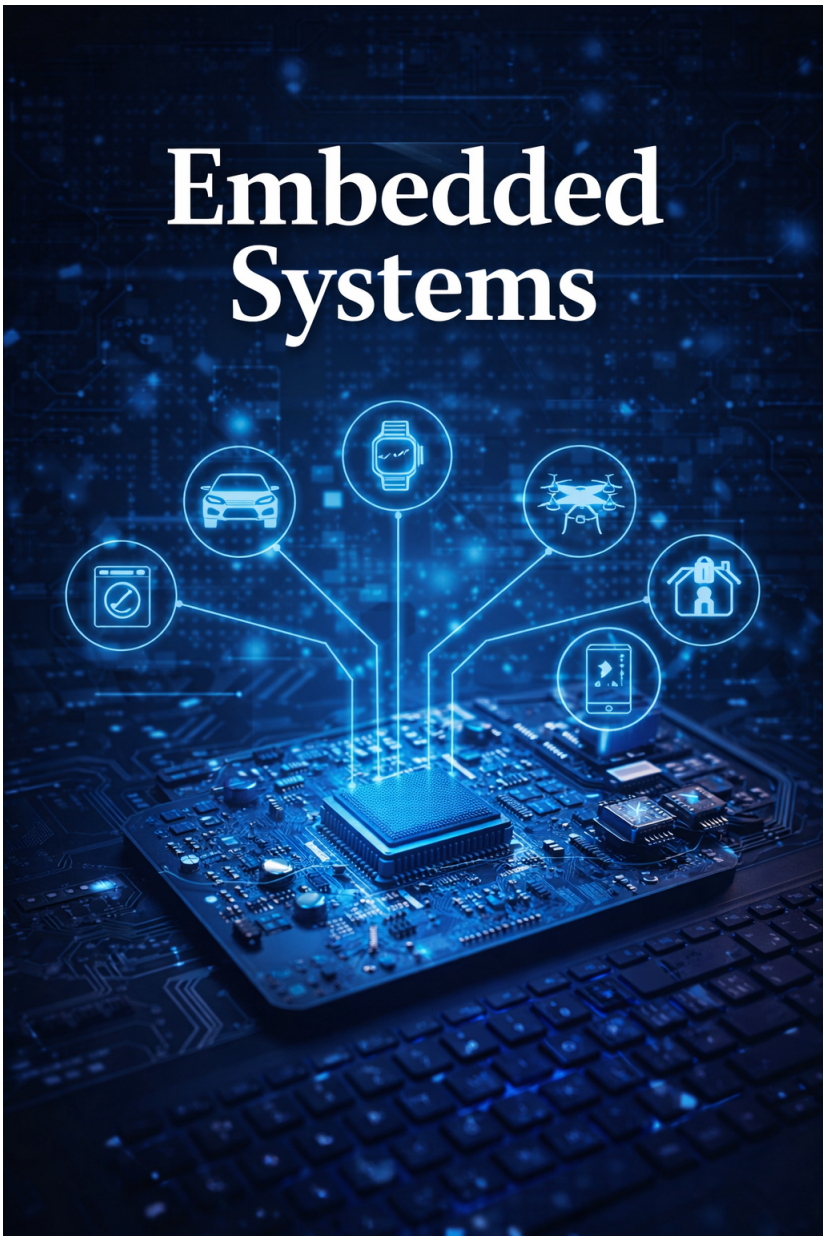
Sisteme Incorporate

Tiberiu T. Cocias



Universitatea
Transilvania
din Brașov

2026

Cuprins

1. Introducere în Sistemele Încorporate

Ce este un sistem încorporat
Caracteristicile sistemelor embedded
Domenii de aplicare ale sistemelor embedded
Diferențe între sistemele embedded și calculatoarele de uz general
Constrângeri de proiectare
Clasificarea sistemelor embedded
Tendințe moderne și Internet of Things
Întrebări de verificare

Sistemele încorporate reprezintă una dintre cele mai importante categorii de sisteme de calcul dezvoltate în ultimele decenii. Deși utilizatorii asociază în mod obișnuit noțiunea de calculator cu un sistem desktop, un laptop sau un server, realitatea este că majoritatea procesoarelor fabricate anual nu sunt utilizate în astfel de echipamente, ci sunt integrate în produse dedicate care îndeplinesc funcții bine definite. Automobilele moderne, echipamentele medicale, aparatele electrocasnice, sistemele industriale sau dispozitivele inteligente din locuințe conțin numeroase sisteme embedded care funcționează continuu, de cele mai multe ori fără ca utilizatorul să fie conștient de existența lor.

Importanța acestor sisteme a crescut semnificativ odată cu miniaturizarea componentelor electronice și cu dezvoltarea comunicațiilor digitale. Apariția conceptului de *Internet of Things* (IoT) a accelerat această evoluție, transformând sistemele embedded din dispozitive izolate în noduri inteligente capabile să comunice și să colaboreze în cadrul unor infrastructuri distribuite [55, 92].

Astăzi, un număr foarte mare dintre echipamentele utilizate în viața de zi cu zi conțin unul sau mai multe microcontrolere dedicate. De exemplu, un automobil modern poate integra peste o sută de unități electronice de control (ECU), fiecare responsabilă pentru o funcționalitate specifică, iar o locuință inteligentă poate include zeci de sisteme embedded care monitorizează și controlează procese precum iluminatul, climatizarea, securitatea sau consumul energetic.

Obiectivul acestui capitol este introducerea conceptelor fundamentale asociate sistemelor încorporate. Vor fi prezentate definițiile de bază, caracteristicile acestor sisteme, principalele

domenii de aplicare, constrângerile de proiectare și tendințele actuale de dezvoltare. Capitolul constituie baza necesară pentru înțelegerea arhitecturilor hardware și software analizate în capitolele următoare.

1.1 Ce este un sistem încorporat

Un sistem încorporat (*Embedded System*) reprezintă un sistem de calcul integrat într-un dispozitiv electronic și proiectat pentru realizarea unei funcții bine definite. Spre deosebire de calculatoarele de uz general, care sunt concepute pentru a executa o varietate foarte mare de aplicații, un sistem embedded este optimizat încă din faza de proiectare pentru un domeniu restrâns de utilizare. Hardware-ul și software-ul sunt dezvoltate împreună astfel încât să satisfacă cerințele specifice ale aplicației, obținând un compromis optim între performanță, cost, consum energetic și fiabilitate [61, 89].

În majoritatea cazurilor, utilizatorul nu interacționează direct cu sistemul de calcul, ci cu produsul final în care acesta este integrat. Din această perspectivă, calculatorul nu mai reprezintă produsul în sine, ci devine o componentă invizibilă care controlează funcționarea echipamentului. Acesta este unul dintre aspectele care diferențiază fundamental sistemele embedded de calculatoarele clasice.

Din punct de vedere conceptual, un sistem embedded poate fi privit ca o combinație între componente hardware, software și interfețe cu mediul fizic. Informațiile provenite de la senzori sunt prelucrate de microcontroler sau microprocesor, iar rezultatele sunt utilizate pentru controlul actuatoarelor sau pentru comunicarea cu alte sisteme. Figura 1.1 ilustrează arhitectura generală a unui sistem embedded.

În forma sa cea mai simplă, un sistem embedded este alcătuit dintr-o unitate de procesare, o memorie pentru stocarea programului și a datelor, periferice de intrare-ieșire, senzori, actuatoare și componente de comunicație. În funcție de complexitatea aplicației, arhitectura poate include și acceleratoare hardware, memorii externe, sisteme de operare de timp real sau module dedicate de securitate.

1.2 Caracteristicile sistemelor embedded

Deși sistemele embedded sunt extrem de diverse din punctul de vedere al aplicațiilor în care sunt utilizate, ele împărtășesc un set de caracteristici fundamentale care le diferențiază de calculatoarele de uz general. Aceste caracteristici nu reprezintă reguli absolute, deoarece există sisteme embedded foarte simple și sisteme embedded de o complexitate comparabilă cu cea a unui calculator personal. Totuși, în majoritatea aplicațiilor industriale și comerciale, proiectarea unui sistem embedded urmărește satisfacerea simultană a unor cerințe privind performanța, consumul energetic, costul și fiabilitatea, ceea ce conduce la compromisuri inginerești specifice [55, 92].

Spre deosebire de un calculator personal, unde utilizatorul poate instala sau elimina aplicații după necesități, funcționalitatea unui sistem embedded este stabilită încă din faza de proiectare. Programul executat de microcontroler este dezvoltat pentru a rezolva o anumită problemă și, în majoritatea cazurilor, nu este modificat pe durata utilizării produsului.

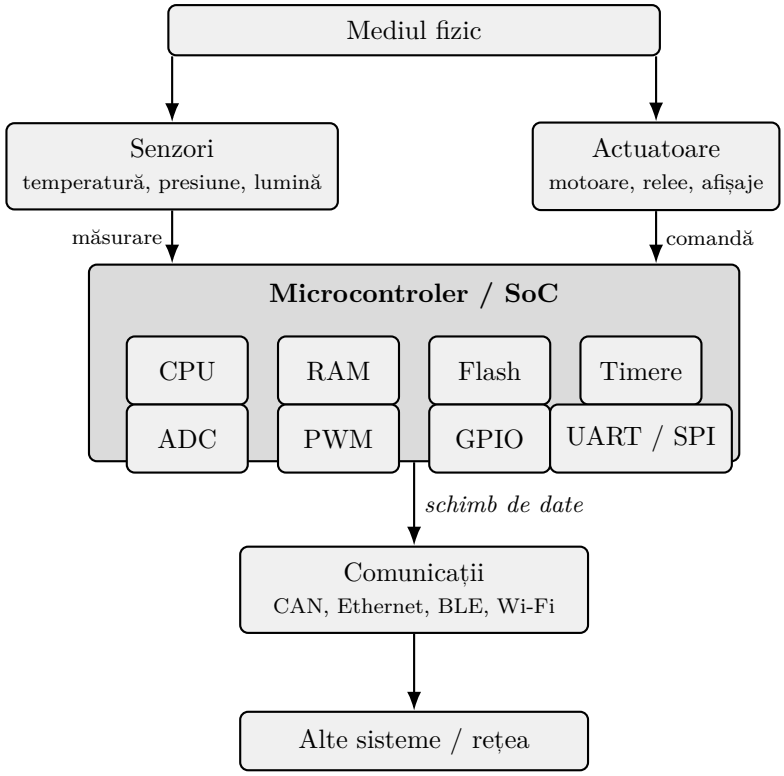


Figura 1.1 Arhitectura generală a unui sistem embedded. Senzorii furnizează informații despre mediul înconjurător, acestea sunt prelucrate de unitatea de calcul, iar rezultatele sunt utilizate pentru controlul actuatoarelor și pentru comunicația cu alte sisteme. Adaptată după [55, 92].

Această abordare permite optimizarea atentă a resurselor hardware și software și conduce la sisteme compacte, eficiente și fiabile.

În continuare sunt prezentate cele mai importante caracteristici care definesc sistemele embedded moderne.

1.2.1 Funcționalitate dedicată

Una dintre caracteristicile definitorii ale sistemelor embedded este faptul că acestea sunt proiectate pentru îndeplinirea unei funcții bine determinate - în loc să execute o gamă foarte variată de aplicații, precum un calculator personal, sistemul embedded realizează una sau câteva operații specifice aplicației în care este integrat. Această specializare permite optimizarea întregului sistem: atât componentele hardware, cât și aplicația software sunt selectate astfel încât să satisfacă exact cerințele produsului, fără a include resurse inutile, ceea ce reduce semnificativ costul, dimensiunile și consumul energetic.

De exemplu, controlerul electronic al unei mașini de spălat execută permanent aceleași funcții: citește informațiile provenite de la senzori, controlează motorul, gestionează electrovalvele și monitorizează desfășurarea programului de spălare. Spre deosebire de un calculator personal, acesta nu trebuie să ruleze aplicații de editare, jocuri sau browsere web.

1.2.2 Resurse hardware limitate

În majoritatea aplicațiilor, sistemele embedded sunt construite în jurul unor microcontrolere care dispun de resurse hardware limitate comparativ cu procesoarele utilizate în calculatoarele personale. Capacitatea memoriei RAM poate varia de la câțiva kilobiți până la câțiva megabiți, iar memoria Flash trebuie să stocheze atât programul executabil, cât și eventualele date permanente. Aceste limitări obligă proiectantul să utilizeze eficient fiecare resursă disponibilă - algoritmii trebuie să fie cât mai simpli, iar consumul de memorie și timpul de execuție trebuie analizate încă din faza de proiectare. În practică, aceasta reprezintă una dintre principalele diferențe dintre dezvoltarea software pentru calculatoare personale și dezvoltarea aplicațiilor embedded.

1.2.3 Consum energetic redus

Numeroase sisteme embedded funcționează utilizând baterii sau alte surse autonome de energie, motiv pentru care reducerea consumului energetic reprezintă una dintre cele mai importante cerințe de proiectare. Consumul poate fi diminuat prin utilizarea microcontrolerelor cu moduri de funcționare de tip *sleep*, prin oprirea perifericelor neutilizate și prin optimizarea algoritmilor software. În multe aplicații moderne, sistemul rămâne în stare de consum redus cea mai mare parte a timpului și se activează doar atunci când trebuie să efectueze o măsurătoare sau să transmită informații.

Acest principiu va fi analizat în detaliu în capitolele dedicate managementului energetic și tehnologiilor de *Energy Harvesting*.

1.2.4 Fiabilitate și funcționare continuă

Sistemele embedded sunt adesea integrate în echipamente care trebuie să funcționeze fără întrerupere perioade foarte lungi de timp. În aplicațiile industriale, medicale sau automotive, o defecțiune software poate conduce nu doar la pierderea unor informații, ci și la deteriorarea echipamentelor sau la punerea în pericol a siguranței utilizatorilor. Din acest motiv, fiabilitatea reprezintă un criteriu de proiectare la fel de important precum performanța: dezvoltarea aplicațiilor embedded presupune utilizarea unor metode riguroase de verificare și validare, precum și respectarea standardelor specifice domeniului de aplicație, de exemplu ISO 26262 pentru industria automotive sau IEC 61508 pentru sistemele critice de control.

După cum se poate observa, aceste caracteristici nu sunt independente, ci influențează reciproc procesul de proiectare. Creșterea performanței poate conduce la un consum energetic mai mare, iar reducerea costurilor poate limita resursele hardware disponibile. Din acest motiv, proiectarea unui sistem embedded presupune identificarea unui compromis optim între cerințele aplicației și resursele disponibile.

Aceste compromisuri reprezintă una dintre principalele provocări ale inginerului proiectant. Spre deosebire de dezvoltarea aplicațiilor software pentru calculatoarele personale, unde resursele hardware pot fi extinse relativ ușor, în sistemele embedded alegerea componentelor hardware și proiectarea software-ului trebuie realizate simultan. În consecință,

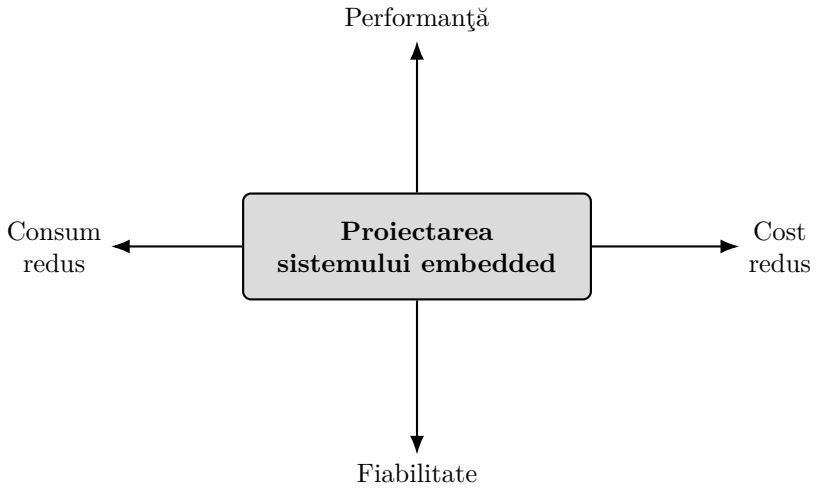


Figura 1.2 Compromisurile fundamentale în proiectarea sistemelor embedded. Creșterea performanței conduce, în general, la creșterea costului și a consumului energetic, motiv pentru care proiectarea unui sistem embedded urmărește identificarea unui echilibru optim între cerințele aplicației și resursele disponibile. Adaptată după [55].

procesul de dezvoltare necesită o abordare integrată, în care fiecare decizie influențează performanțele și costul produsului final.

1.3 Domenii de aplicare ale sistemelor embedded

Sistemele embedded sunt prezente în aproape toate domeniile tehnologiei moderne. Această răspândire se explică prin faptul că foarte multe produse actuale necesită monitorizare, control, comunicație sau procesare locală a informațiilor. În loc ca aceste funcții să fie realizate prin mecanisme pur electromeccanice, ele sunt implementate astăzi prin sisteme de calcul dedicate, integrate direct în produs.

Evoluția microcontrolerelor, reducerea costurilor de fabricație și dezvoltarea comunicațiilor digitale au permis integrarea sistemelor embedded în dispozitive foarte diferite ca dimensiune și complexitate. Astfel, aceeași clasă de principii inginerești poate fi regăsită atât într-un senzor de temperatură de câțiva euro, cât și într-un automobil modern, într-un robot industrial sau într-un satelit [55, 61, 92].

Figura 1.3 sintetizează principalele domenii în care sistemele embedded sunt utilizate în mod frecvent.

În electronica de consum, sistemele embedded controlează funcționarea televizoarelor inteligente, telefoanelor mobile, camerelor digitale, consolelor de jocuri și aparatelor multimedia. În aceste dispozitive, utilizatorul interacționează cu o interfață grafică sau cu un set redus de comenzi, însă în interior rulează aplicații software complexe, optimizate pentru hardware-ul disponibil.

Electrocasnicele moderne reprezintă o altă categorie foarte răspândită de aplicații. O mașină de spălat, de exemplu, include un controler care gestionează nivelul apei, temperatura, turația motorului și durata programului de spălare. În același timp, sistemul poate

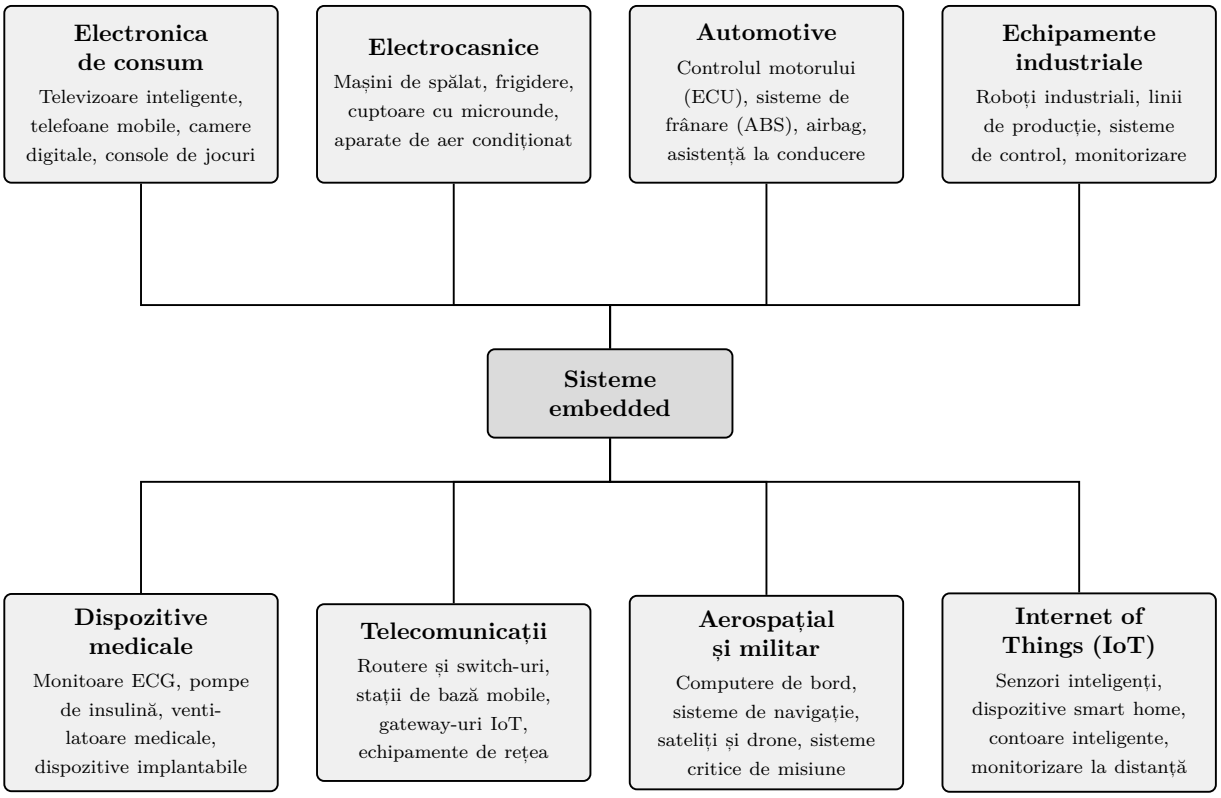


Figura 1.3 Domenii reprezentative de aplicare ale sistemelor embedded. Acestea sunt integrate în produse de consum, automobile, echipamente industriale, dispozitive medicale, infrastructuri de comunicație, sisteme aerospațiale și aplicații IoT.

detecta erori, poate afișa mesaje către utilizator și, în variantele moderne, poate comunica prin rețea cu o aplicație mobilă. Prin utilizarea sistemelor embedded, electrocasnicele au devenit mai eficiente energetic, mai sigure și mai ușor de adaptat la nevoile utilizatorilor.

Industria automotive este unul dintre domeniile în care sistemele embedded au avut un impact major. Un automobil modern conține numeroase unități electronice de control, cunoscute sub denumirea de ECU (*Electronic Control Unit*). Acestea controlează motorul, transmisia, sistemele de frânare, airbag-urile, climatizarea, iluminatul, sistemele multimedia și funcțiile avansate de asistență la conducere. În astfel de aplicații, cerințele privind siguranța funcțională, timpul real și fiabilitatea sunt deosebit de stricte, motiv pentru care dezvoltarea sistemelor embedded automotive este reglementată prin standarde precum ISO 26262 [44].

Exemplu

Sistemul ABS al unui automobil este un sistem embedded de timp real. Acesta monitorizează permanent viteza roților și comandă presiunea de frânare astfel încât roțile să nu se blocheze în timpul frânării. Dacă algoritmul de control ar reacționa prea târziu, funcția de siguranță ar deveni inutilă. Acest exemplu evidențiază legătura directă dintre senzori, procesare, actuatoare și constrângeri temporale.

În domeniul medical, sistemele embedded sunt integrate în echipamente precum moni-

toare ECG, pompe de insulină, ventilatoare medicale, aparate de imagistică și dispozitive implantabile. În aceste cazuri, fiabilitatea și siguranța sunt esențiale, deoarece o defecțiune poate afecta direct sănătatea pacientului. Din acest motiv, dezvoltarea echipamentelor medicale presupune testare riguroasă, validare clinică și respectarea standardelor specifice domeniului.

În automatizarea industrială, sistemele embedded controlează linii de producție, roboți industriali, sisteme de măsurare, controlere programabile și echipamente de monitorizare. Aceste sisteme funcționează adesea în medii dificile, caracterizate prin temperaturi ridicate, vibrații, praf sau zgomot electromagnetic. Din acest motiv, robustețea și disponibilitatea sunt criterii importante de proiectare.

În telecomunicații, routerele, switch-urile, gateway-urile IoT și stațiile de bază pentru rețele mobile sunt exemple de sisteme embedded specializate în procesarea și transferul rapid al datelor. Aceste echipamente rulează continuu și trebuie să ofere disponibilitate ridicată, latență redusă și securitate adecvată.

Aplicațiile aerospațiale și militare reprezintă unele dintre cele mai exigente domenii de utilizare. Calculatoarele de bord, sistemele de navigație, sateliții și dronele autonome trebuie să funcționeze în condiții extreme și să respecte cerințe stricte de fiabilitate. Un exemplu istoric important este Apollo Guidance Computer, considerat unul dintre primele sisteme embedded moderne utilizate într-o aplicație critică [35, 57].

În ultimii ani, Internet of Things a extins semnificativ aria de aplicare a sistemelor embedded. Dispozitivele IoT colectează informații din mediul înconjurător, le procesează local și le transmit către alte dispozitive sau către infrastructuri cloud. Exemplele includ senzori agricoli, contoare inteligente, sisteme de iluminat, dispozitive smart home și echipamente industriale conectate.

Prin urmare, sistemele embedded nu mai pot fi privite doar ca simple controlere locale. Ele au devenit componente fundamentale ale unor ecosisteme complexe, distribuite și conectate, în care funcțiile de calcul, comunicație și control sunt integrate direct în obiectele și infrastructurile utilizate zilnic.

1.4 Diferențe între sistemele embedded și calculatoarele de uz general

La prima vedere, un sistem embedded și un calculator personal par să aibă multe elemente în comun. Ambele conțin un procesor, memorie, dispozitive de intrare-ieșire și execută programe software. Totuși, scopul pentru care au fost proiectate este fundamental diferit. În timp ce un calculator personal trebuie să poată executa o gamă foarte variată de aplicații, un sistem embedded este dezvoltat pentru îndeplinirea uneia sau a unui număr restrâns de funcții specifice.

Această diferență de obiectiv influențează întregul proces de proiectare. În cazul calculatoarelor personale, accentul este pus pe flexibilitate și compatibilitate cu aplicații diverse. Utilizatorul poate instala sau elimina programe, poate înlocui componente hardware și poate modifica configurația sistemului. În schimb, într-un sistem embedded configurația hardware este stabilită încă din faza de proiectare, iar aplicația software este optimizată special pentru

aceasta. De cele mai multe ori, utilizatorul final nici măcar nu este conștient că în interiorul produsului funcționează un sistem de calcul dedicat [55, 92].

O altă diferență importantă este reprezentată de resursele disponibile. Calculatoarele moderne dispun de procesoare multicore performante, memorii de ordinul gigabyților și dispozitive de stocare rapide, fiind capabile să execute simultan numeroase aplicații. În schimb, multe sisteme embedded utilizează microcontrolere cu doar câteva sute de kilobyți sau câțiva megabyți de memorie și funcționează la frecvențe semnificativ mai reduse. Aceste limitări impun utilizarea unor algoritmi eficienți și o proiectare atentă a software-ului.

Din punct de vedere al consumului energetic, diferențele sunt și mai evidente. Calculatoarele personale sunt alimentate, în general, de la rețeaua electrică și urmăresc în principal maximizarea performanței. În schimb, numeroase sisteme embedded sunt alimentate de baterii și trebuie să funcționeze luni sau chiar ani fără înlocuirea sursei de energie. Din acest motiv, optimizarea consumului energetic reprezintă una dintre principalele preocupări ale proiectantului.

În ceea ce privește fiabilitatea, multe sisteme embedded funcționează în aplicații critice, unde o eroare software poate avea consecințe importante asupra siguranței persoanelor sau asupra funcționării echipamentului. Exemple reprezentative sunt sistemele de frânare ABS, controlul airbag-urilor, dispozitivele medicale implantabile sau controlerele utilizate în centrale electrice. În astfel de aplicații sunt utilizate standarde riguroase de dezvoltare și validare, precum ISO 26262 sau IEC 61508.

Figura 1.4 sintetizează principalele diferențe dintre cele două categorii de sisteme.

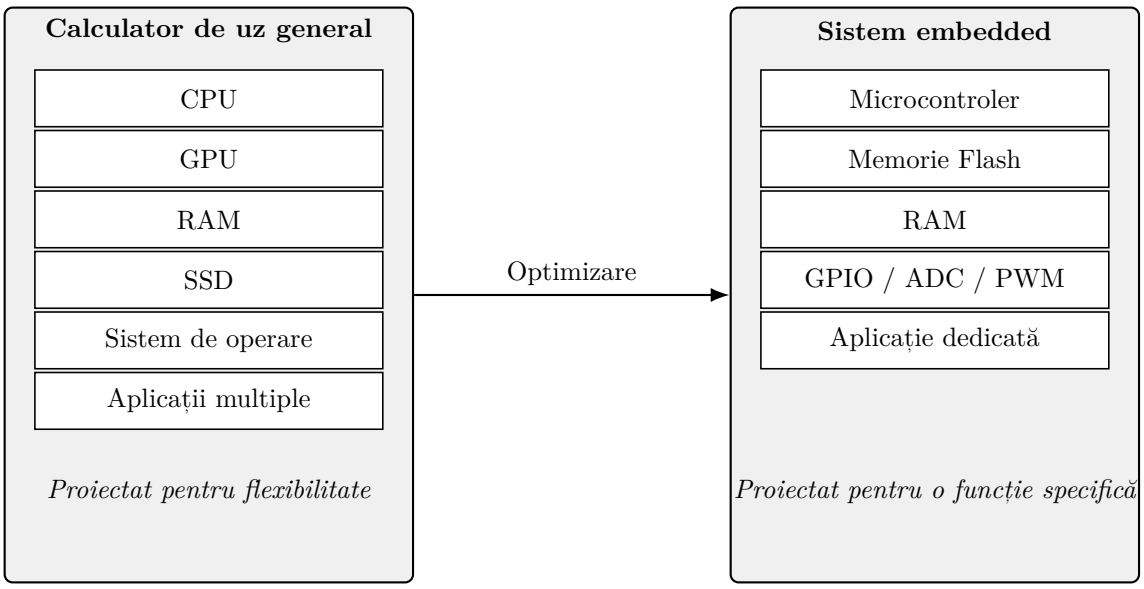


Figura 1.4 Comparație conceptuală între un calculator de uz general și un sistem embedded.

Tabelul 1.1 evidențiază principalele caracteristici care diferențiază cele două categorii de sisteme.

Tabela 1.1 Comparație între calculatoarele de uz general și sistemele embedded

Caracteristică	**Calculator de uz general**	**Sistem embedded**
Scop	Execuția unei game variate de aplicații	Execuția unei funcții dedicate
Hardware	Configurabil și extensibil	Stabilit în faza de proiectare
Sistem de operare	Windows, Linux, macOS	Bare-metal sau RTOS
Consum energetic	Ridicat	Redus
Resurse hardware	Abundente	Limitate
Cost per unitate	Ridicat	Optimizat pentru producție în serie
Fiabilitate	Importantă	Critică în aplicațiile de siguranță
Interacțiunea cu utilizatorul	Directă	De regulă indirectă

Exemplu

Atât un laptop, cât și unitatea electronică de control a motorului unui automobil conțin un procesor și memorie pentru execuția programelor. Totuși, laptopul trebuie să ruleze simultan aplicații foarte diferite, precum un browser web, un editor de text sau un program CAD. În schimb, controlerul motorului execută permanent același algoritm de control al injecției și al aprinderii, fiind optimizat exclusiv pentru această funcție. Acest exemplu evidențiază diferența fundamentală dintre un sistem de calcul de uz general și un sistem embedded.

În concluzie, deși sistemele embedded utilizează aceleași principii fundamentale de calcul precum calculatoarele personale, modul în care acestea sunt proiectate și utilizate este semnificativ diferit. Accentul se mută de la flexibilitate către eficiență, fiabilitate și integrarea strânsă dintre hardware și software. Aceste particularități explică de ce dezvoltarea sistemelor embedded necesită metode și instrumente specifice, care vor fi prezentate în capitolele următoare.

1.5 Constrângeri de proiectare

Proiectarea unui sistem embedded reprezintă un proces de optimizare în care trebuie satisfăcute simultan mai multe cerințe, adesea contradictorii. Spre deosebire de dezvoltarea aplicațiilor pentru calculatoarele de uz general, unde resursele hardware pot fi extinse relativ ușor, în sistemele embedded alegerea componentelor hardware și proiectarea software-ului sunt strâns interdependente. Fiecare decizie influențează performanțele, consumul energetic, costul și fiabilitatea produsului final [55, 61, 92].

În practică, proiectantul nu urmărește maximizarea unui singur criteriu de performanță, ci identificarea unui compromis optim între mai multe constrângeri impuse de aplicație. De exemplu, utilizarea unui procesor mai performant poate reduce timpul de execuție al algoritmilor, însă conduce de regulă la creșterea consumului energetic și a costului sistemului. În mod similar, reducerea costurilor poate impune utilizarea unui microcontroler cu memorie limitată, ceea ce influențează complexitatea aplicației software care poate fi implementată.

Figura 1.2 ilustrează relațiile dintre principalele criterii care trebuie avute în vedere în procesul de proiectare.

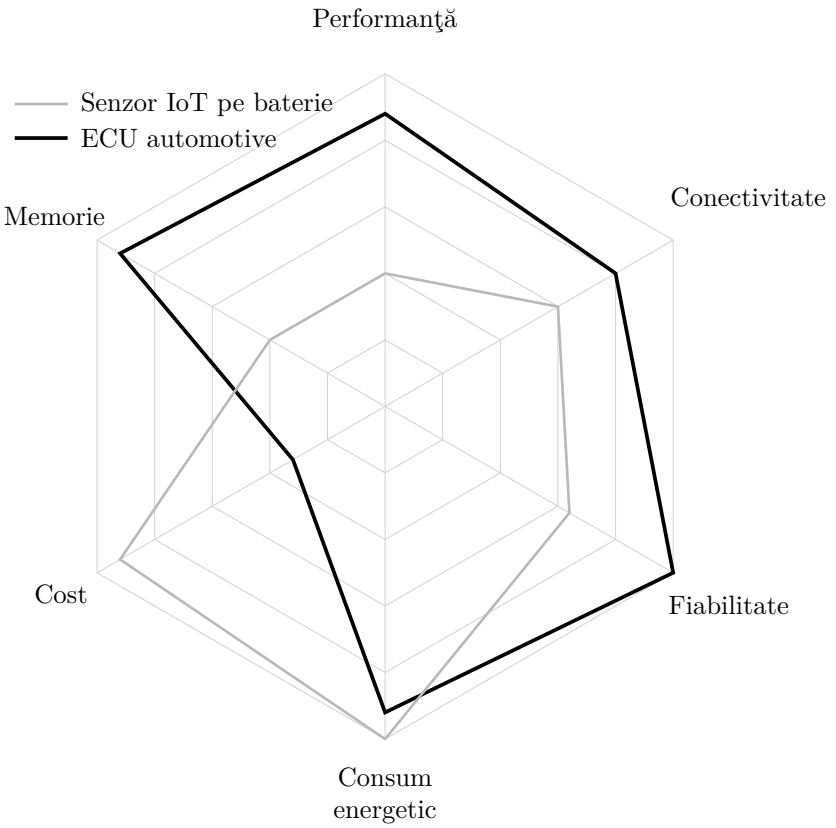


Figura 1.5 Compararea priorităților de proiectare pentru două categorii de sisteme embedded. Un senzor IoT alimentat din baterie prioritizează consumul energetic și costul redus, în timp ce un ECU automotive pune accent pe performanță, fiabilitate și resurse de calcul.

Una dintre cele mai importante constrângeri este reprezentată de timpul de răspuns. Numeroase sisteme embedded trebuie să reacționeze într-un interval de timp bine definit după apariția unui eveniment extern. Într-un automobil, de exemplu, întârzierea activării sistemului de frânare antiblocare (ABS) poate afecta stabilitatea vehiculului. În mod similar, într-un dispozitiv medical, întârzierea prelucrării unui semnal fiziologic poate avea consecințe asupra siguranței pacientului. Din acest motiv, timpul de execuție al algoritmilor trebuie analizat încă din faza de proiectare.

O altă constrângere importantă este consumul energetic. Numeroase sisteme embedded funcționează utilizând baterii sau surse autonome de energie și trebuie să opereze perioade îndelungate fără intervenția utilizatorului. Reducerea consumului se realizează atât prin

alegerea unor componente hardware eficiente, cât și prin dezvoltarea unor algoritmi software care utilizează în mod judicios resursele disponibile. Tehnici precum modurile de funcționare *sleep*, oprirea perifericelor neutilizate sau adaptarea frecvenței procesorului sunt utilizate frecvent în aplicațiile moderne.

Costul reprezintă, de asemenea, un criteriu esențial, în special în cazul produselor fabricate în volume mari. O diferență de câțiva euro în costul unei componente poate conduce la economii semnificative atunci când sunt produse sute de mii sau milioane de unități. Din acest motiv, selecția microcontrolerului și a perifericelor asociate trebuie realizată cu atenție, astfel încât sistemul să satisfacă cerințele aplicației fără utilizarea unor resurse inutile.

În numeroase aplicații industriale și automotive, fiabilitatea este la fel de importantă ca performanța. Sistemele trebuie să funcționeze continuu perioade îndelungate și să își păstreze comportamentul chiar și în condiții dificile de temperatură, vibrații sau perturbații electromagnetice. Pentru aplicațiile critice sunt utilizate standarde precum ISO 26262, IEC 61508 sau DO-178C, care definesc procese riguroase de dezvoltare, verificare și validare.

Pe lângă aceste constrângeri, proiectarea sistemelor embedded trebuie să țină seama și de dimensiunile fizice ale echipamentului, disiparea termică, disponibilitatea memoriei și posibilitățile de actualizare ulterioară a software-ului. În ultimii ani, cerințele privind securitatea cibernetică au devenit la fel de importante, întrucât tot mai multe sisteme embedded sunt conectate la rețele locale sau la Internet.

Exemplu

Un senzor inteligent destinat monitorizării unei culturi agricole trebuie să funcționeze timp de mai mulți ani utilizând o singură baterie. Pentru atingerea acestui obiectiv, microcontrolerul rămâne în modul *sleep* aproape tot timpul și este activat doar periodic pentru efectuarea măsurătorilor și transmiterea datelor. În acest caz, reducerea consumului energetic este mult mai importantă decât maximizarea performanței procesorului, iar alegerea arhitecturii hardware și a algoritmilor software este influențată direct de această constrângere.

În concluzie, proiectarea unui sistem embedded presupune găsirea unui echilibru între numeroase criterii inginerești. Nu există o soluție universal optimă, iar configurația finală depinde întotdeauna de cerințele aplicației. Înțelegerea acestor constrângeri reprezintă primul pas către dezvoltarea unor sisteme embedded eficiente, fiabile și adaptate domeniului în care vor fi utilizate.

1.6 Clasificarea sistemelor embedded

Diversitatea aplicațiilor embedded face dificilă definirea unei singure clasificări care să acopere toate sistemele existente. În practică, acestea sunt grupate în funcție de mai multe criterii, precum complexitatea hardware, constrângerile de timp real, performanțele de calcul, consumul energetic sau domeniul de utilizare. Fiecare dintre aceste clasificări evidențiază anumite caracteristici ale sistemului și facilitează alegerea soluțiilor hardware și software

adecvate aplicației dezvoltate.

Din punctul de vedere al complexității, sistemele embedded pot fi împărțite în trei categorii principale: **sisteme embedded de mică complexitate**, bazate de regulă pe microcontrolere pe 8 sau 16 biți, cu resurse hardware limitate și aplicații relativ simple, precum telecomenzile, termostatele electronice sau electrocasnicele inteligente; **sisteme embedded de complexitate medie**, construite în jurul microcontrolerelor pe 32 de biți, care includ periferice complexe și interfețe multiple de comunicație, categorie în care se încadrează majoritatea sistemelor automotive, industriale și IoT; și **sisteme embedded complexe**, bazate pe procesoare multicore sau pe platforme System-on-Chip (SoC), capabile să execute sisteme de operare precum Linux Embedded sau Android Embedded, precum sistemele infotainment, echipamentele medicale avansate și controlerele industriale performante.

Din punctul de vedere al timpului de răspuns, sistemele embedded sunt împărțite în trei categorii: **sisteme hard real-time**, în care depășirea termenului limită conduce la defectarea sistemului sau la apariția unor situații periculoase; **sisteme firm real-time**, unde depășirea ocazională a termenului limită reduce performanța aplicației, fără a conduce imediat la defectarea acesteia; și **sisteme soft real-time**, în care întârzierile afectează calitatea serviciului, însă funcționarea sistemului continuă.

Figura 1.6 sintetizează principalele criterii utilizate pentru clasificarea sistemelor embedded.

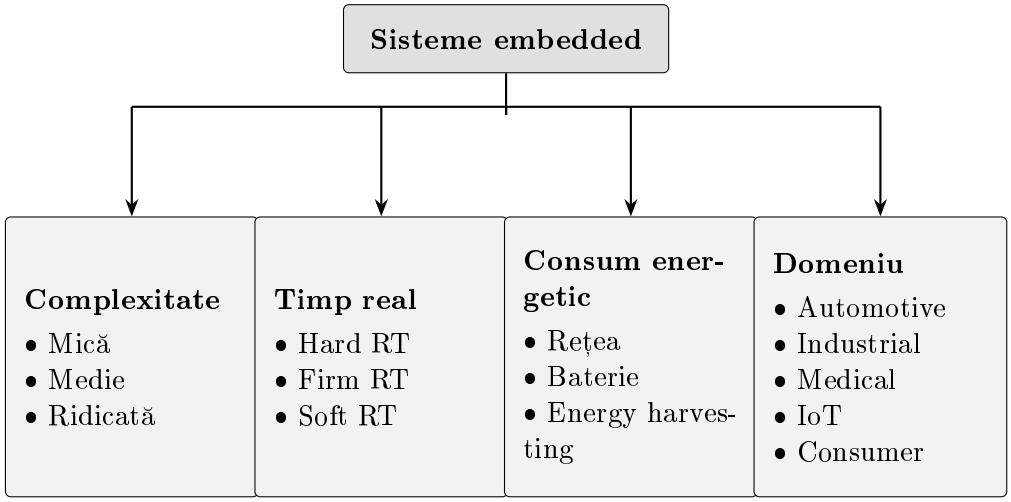


Figura 1.6 Clasificarea sistemelor embedded în funcție de principalele criterii utilizate în practică.

În funcție de sursa de alimentare și autonomia necesară, sistemele embedded pot fi alimentate permanent de la rețeaua electrică sau pot funcționa pe baterii. Această clasificare influențează alegerea microcontrolerului, frecvența de funcționare și strategiile de economisire a energiei.

Din punctul de vedere al aplicației, sistemele embedded sunt întâlnite într-o gamă foarte variată de domenii: automotive, industrie, echipamente medicale, electronică de consum, automatizări, telecomunicații, Internet of Things și sisteme aerospațiale. Deși principiile fundamentale de funcționare sunt similare, fiecare domeniu impune cerințe specifice privind

siguranța, fiabilitatea, consumul energetic și costul.

Exemplu

Un controler destinat unei mașini de spălat poate fi încadrat în categoria sistemelor embedded de complexitate redusă și fără constrângeri severe de timp real. În schimb, un controler pentru sistemul de frânare ABS este un sistem hard real-time, în care orice întârziere peste limita admisă poate afecta siguranța vehiculului. Deși ambele utilizează microcontrolere, cerințele de proiectare sunt semnificativ diferite.

În practică, un sistem embedded aparține simultan mai multor categorii. De exemplu, un nod IoT alimentat din baterie poate fi un sistem de complexitate redusă, cu consum energetic foarte mic și comunicație wireless, în timp ce un controler industrial poate fi un sistem de complexitate ridicată, alimentat permanent și cu cerințe stricte de timp real. Din acest motiv, clasificările prezentate trebuie privite ca perspective complementare asupra aceluiași sistem.

1.7 Tendințe moderne și Internet of Things

În ultimele două decenii, sistemele embedded au evoluat de la dispozitive independente, proiectate pentru executarea unei singure funcții, către platforme inteligente capabile să comunice, să proceseze volume mari de date și să ia decizii autonome. Această evoluție a fost determinată de creșterea performanțelor microcontrolerelor, reducerea consumului energetic și dezvoltarea tehnologiilor de comunicație fără fir [55, 61, 92].

Unul dintre cele mai importante concepte dezvoltate în acest context este *Internet of Things* (IoT), care descrie interconectarea obiectelor fizice prin intermediul rețelelor de comunicație. În cadrul unui sistem IoT, dispozitivele embedded colectează informații utilizând senzori, procesează local o parte dintre date și transmit informațiile relevante către alte dispozitive sau către platforme cloud [12, 71].

Figura 1.7 prezintă arhitectura generală a unui sistem Internet of Things.

Pe măsură ce puterea de calcul a microcontrolerelor a crescut, tot mai multe aplicații au început să execute algoritmi avansați direct pe dispozitiv. Această abordare, cunoscută sub denumirea de *Edge Computing*, reduce latența comunicației, diminuează traficul de date și permite funcționarea chiar și în absența unei conexiuni permanente la Internet [76, 80]. O direcție recentă de cercetare este *TinyML*, care urmărește implementarea algoritmilor de învățare automată pe microcontrolere cu resurse limitate, deschizând noi posibilități pentru monitorizarea inteligentă, mentenanța predictivă și analiza datelor în timp real [15, 91].

O altă tendință importantă este integrarea mecanismelor de actualizare software la distanță (*Over-the-Air Updates – OTA*), care permit îmbunătățirea funcționalităților și remedierea vulnerabilităților fără intervenția directă asupra echipamentului. Această abordare este utilizată pe scară largă în industria automotive, în dispozitivele IoT și în echipamentele industriale conectate [55, 92].

Creșterea gradului de conectivitate aduce însă și provocări suplimentare privind securita-

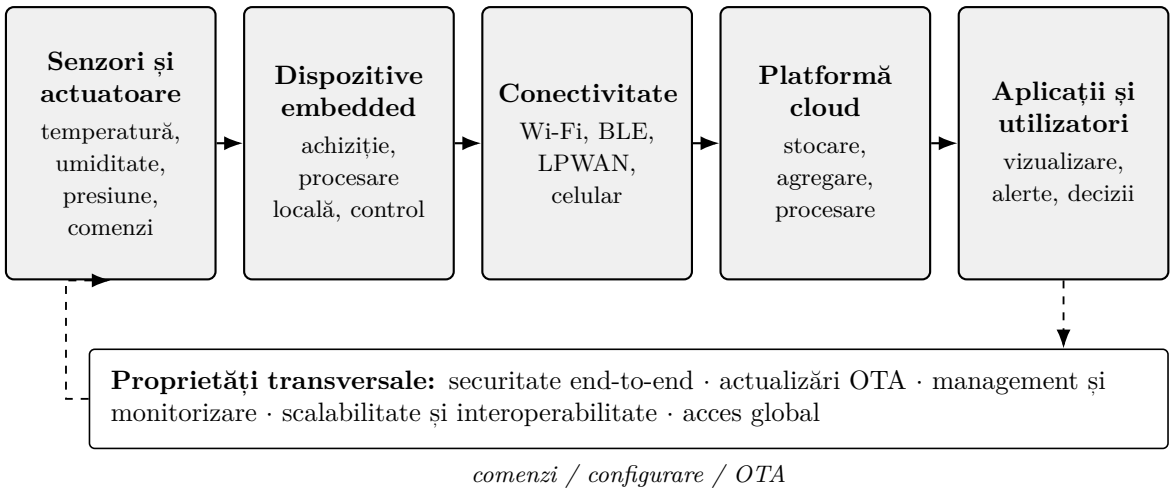


Figura 1.7 Arhitectura generală a unui sistem Internet of Things (IoT).

tea cibernetică. Sistemele embedded moderne trebuie să asigure autentificarea dispozitivelor, criptarea comunicațiilor, protecția memoriei și actualizarea sigură a firmware-ului. În multe aplicații critice, securitatea cibernetică este tratată împreună cu siguranța funcțională, cele două domenii devenind complementare în proiectarea sistemelor moderne [41, 45].

Exemplu

O stație meteorologică inteligentă utilizează senzori pentru măsurarea temperaturii, umidității și presiunii atmosferice. Datele sunt prelucrate local de un microcontroler, iar valorile relevante sunt transmise periodic către o platformă cloud prin intermediul unei conexiuni Wi-Fi sau LoRaWAN. Utilizatorul poate vizualiza informațiile în timp real pe telefonul mobil, poate primi notificări atunci când sunt depășite anumite praguri și poate actualiza firmware-ul dispozitivului fără a interveni fizic asupra acestuia. Acest exemplu ilustrează integrarea într-un singur sistem a conceptelor de IoT, Edge Computing și actualizare OTA.

Evoluția sistemelor embedded este strâns legată de dezvoltarea comunicațiilor, a inteligenței artificiale și a platformelor cloud. În următorii ani se estimează o creștere semnificativă a numărului de dispozitive conectate, precum și extinderea aplicațiilor bazate pe Edge AI, Digital Twins și sisteme autonome [26, 85]. Aceste direcții transformă sistemele embedded din simple controlere dedicate în componente inteligente ale infrastructurilor digitale moderne.

Întrebări de verificare

1. Ce este un sistem embedded și prin ce se diferențiază de un calculator de uz general?
2. Care sunt principalele componente hardware ale unui sistem embedded?
3. Care sunt caracteristicile fundamentale ale sistemelor embedded?
4. De ce reprezintă consumul energetic o constrângere importantă în sistemele embedded?

5. Ce înseamnă funcționalitate dedicată?
6. Care sunt diferențele dintre sistemele hard real-time și soft real-time?
7. Enumerați cel puțin cinci domenii de aplicare ale sistemelor embedded.
8. Care sunt principalele diferențe dintre sistemele desktop, servere și sistemele embedded?
9. Cum influențează costul procesul de proiectare al unui sistem embedded?
10. Ce reprezintă conceptul de Internet of Things?
11. Ce este un dispozitiv MEMS și ce reprezintă conceptul de Smart Dust?
12. Ce este un ASIC și care sunt avantajele sale?
13. Ce este un System-on-Chip și care sunt componentele sale principale?
14. Care sunt avantajele unei platforme SoC?
15. De ce se observă o migrare continuă de la hardware către software?
16. Ce este Edge AI și care sunt avantajele sale?
17. Care sunt principalele tendințe care vor influența viitorul sistemelor embedded?
18. Dați trei exemple de sisteme embedded utilizate în industria automotive și trei exemple din domeniul medical.

2. Arhitectura Sistemelor de Calcul

2.1 Instruction Set Architecture

Orice program executat de un calculator trebuie exprimat într-un limbaj pe care procesorul îl poate interpreta și executa. Acest limbaj este definit de *Instruction Set Architecture* (ISA), care reprezintă interfața dintre software și hardware. Din perspectiva programatorului, ISA descrie comportamentul vizibil al procesorului, fără a impune detalii privind implementarea internă a acestuia. Cu alte cuvinte, ISA stabilește „contractul" dintre aplicațiile software și procesor, permițând dezvoltarea programelor independent de tehnologia utilizată pentru implementarea hardware [38, 65, 82].

Instruction Set Architecture specifică ansamblul caracteristicilor pe care orice implementare hardware trebuie să le respecte pentru a executa aceeași familie de programe. Aceste caracteristici includ setul de instrucțiuni disponibile, registrele accesibile programatorului, tipurile de date suportate, modurile de adresare, organizarea memoriei și mecanismele utilizate pentru tratarea întreruperilor și excepțiilor. Atât compilatorul, cât și programatorul în limbaj de asamblare utilizează exclusiv informațiile definite de ISA, fără a avea nevoie să cunoască modul în care procesorul realizează efectiv execuția instrucțiunilor.

Un aspect important este faptul că două procesoare pot implementa aceeași ISA și totuși să fie construite complet diferit. De exemplu, două generații succesive de procesoare pot avea frecvențe de lucru diferite, dimensiuni diferite ale memoriei cache, număr diferit de

nuclee sau mecanisme distincte de execuție în paralel, însă vor executa același program dacă implementează aceeași arhitectură ISA. Această separare între arhitectura vizibilă și implementarea internă permite evoluția procesoarelor fără modificarea aplicațiilor software existente.

Figura 2.1 ilustrează poziția ISA în cadrul unui sistem de calcul. Software-ul este tradus de compilator în instrucțiuni specifice ISA, iar procesorul implementează aceste instrucțiuni prin intermediul microarhitecturii și al circuitelor hardware.

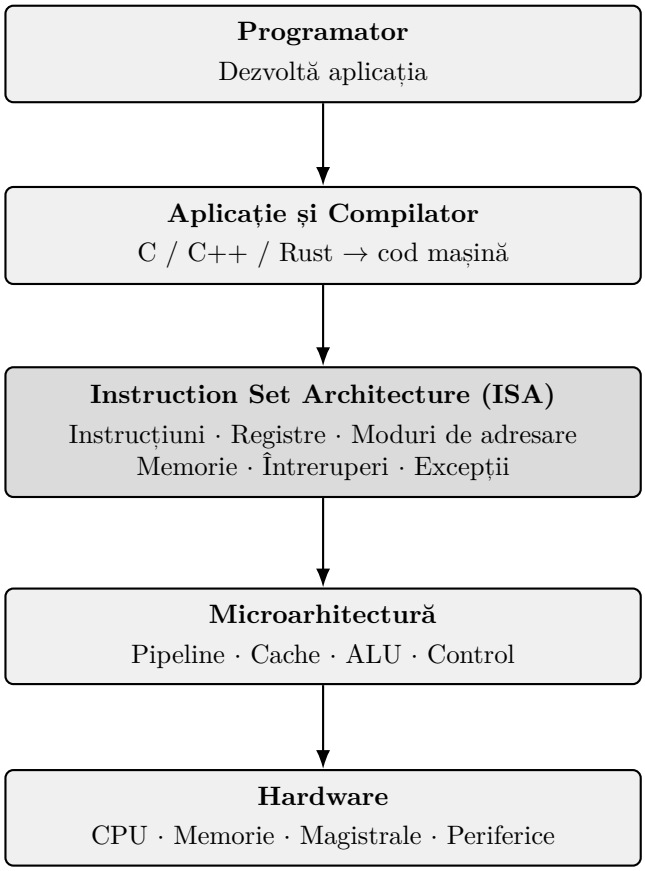


Figura 2.1 Instruction Set Architecture ca interfață dintre software și hardware.

Observație

ISA nu trebuie confundată cu microarhitectura procesorului. ISA descrie funcționalitatea vizibilă programatorului, în timp ce microarhitectura reprezintă implementarea internă utilizată pentru executarea instrucțiunilor. Două procesoare diferite pot implementa aceeași ISA utilizând mecanisme interne complet diferite.

Un exemplu reprezentativ îl constituie arhitectura x86-64, utilizată atât de procesoarele Intel Core, cât și de procesoarele AMD Ryzen. Deși cele două familii folosesc microarhitecturi diferite, ele execută același cod compilat pentru arhitectura x86-64 deoarece implementează aceeași ISA [1, 43].

În domeniul sistemelor embedded, alegerea arhitecturii ISA influențează direct performanța

aplicației, consumul energetic, dimensiunea codului executabil și disponibilitatea instrumentelor software. O arhitectură eficientă permite dezvoltarea unor sisteme rapide, fiabile și cu consum redus de energie, aspect esențial pentru dispozitivele alimentate din baterii.

În prezent, majoritatea microcontrolerelor și procesoarelor dedicate sistemelor embedded utilizează arhitecturi de tip RISC, precum ARM Cortex-M, RISC-V sau AVR, datorită simplității setului de instrucțiuni și eficienței energetice. În schimb, procesoarele destinate calculatoarelor personale și serverelor utilizează în principal arhitecturi din familia x86-64, optimizate pentru performanță ridicată și compatibilitate software.

2.2 Mașini cu trei adrese

Una dintre cele mai intuitive modalități de reprezentare a instrucțiunilor este utilizarea unui format care specifică explicit ambii operanzi sursă și locația în care va fi memorat rezultatul. Acest model este cunoscut sub denumirea de *mașină cu trei adrese* și este frecvent utilizat pentru reprezentarea codului intermediar generat de compilatoare, deoarece reflectă foarte bine expresiile matematice și logice din limbajele de nivel înalt [65, 82].

Într-o instrucțiune cu trei adrese sunt specificate trei câmpuri: doi operanzi utilizați în cadrul operației și un operand destinație în care este memorat rezultatul. Forma generală este

```
ADD DEST, OP1, OP2
```

unde rezultatul operației este

$$DEST = OP1 + OP2.$$

Figura 2.2 ilustrează structura unei instrucțiuni cu trei adrese.

Să considerăm expresia matematică

$$A = B + C \times D - E + F + A. \tag{2.1}$$

O implementare utilizând o mașină cu trei adrese poate fi

```
1 MUL T, C, D
2 ADD T, T, B
3 SUB T, T, E
4 ADD T, T, F
5 ADD A, T, A
```

Observăm că fiecare instrucțiune descrie explicit atât operanzii utilizați, cât și locația în care este memorat rezultatul. Din acest motiv, codul este ușor de urmărit și foarte apropiat de reprezentarea expresiei originale.

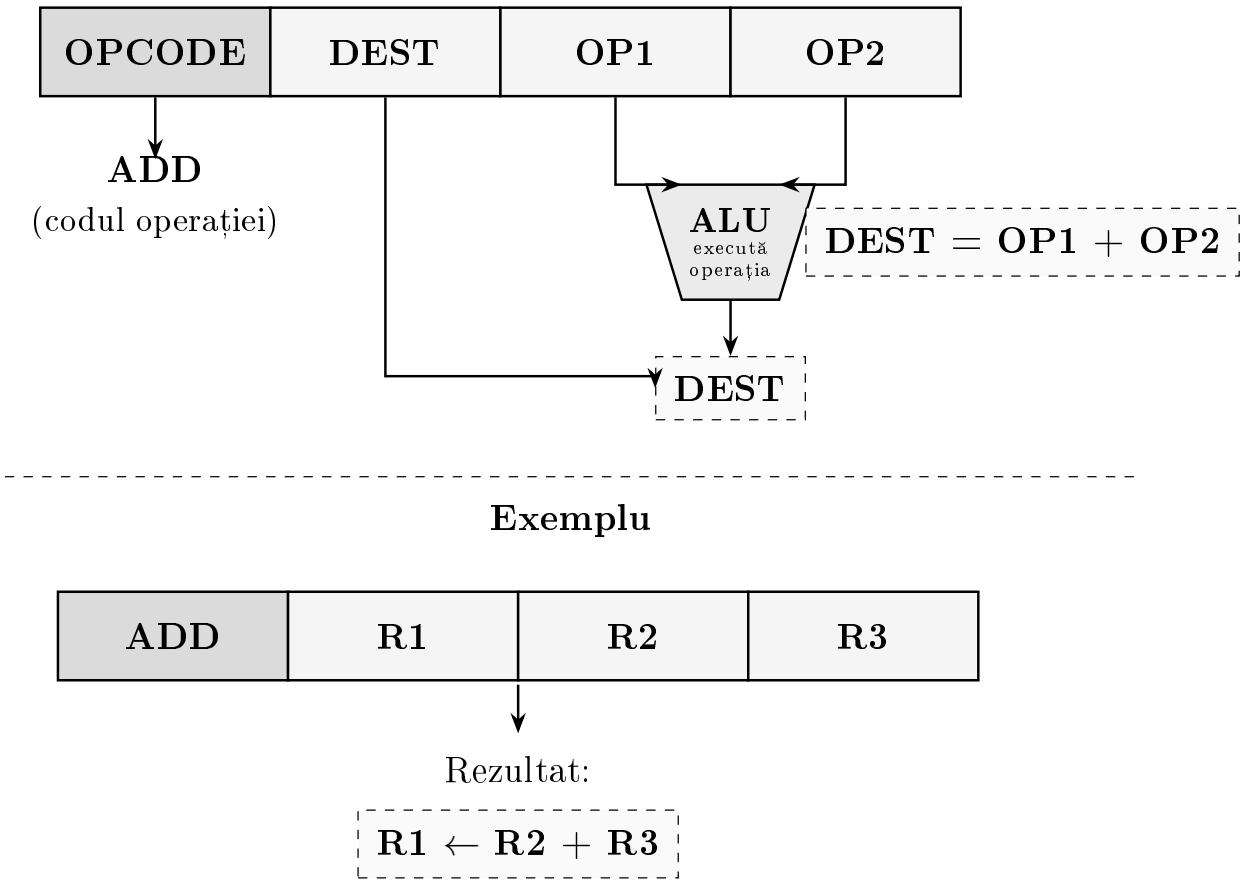


Figura 2.2 Structura unei instrucțiuni cu trei adrese și fluxul operației executate.

Principalul dezavantaj al acestui model constă în dimensiunea instrucțiunilor. Deoarece fiecare instrucțiune trebuie să conțină trei adrese, lungimea codului mașină crește, ceea ce conduce la un consum mai mare de memorie și la o complexitate mai ridicată a procesorului.

Exemplu

Compilatoarele moderne generează frecvent un cod intermediar bazat pe instrucțiuni cu trei adrese înainte de optimizarea și traducerea acestuia către instrucțiunile specifice arhitecturii procesorului. Această reprezentare facilitează aplicarea algoritmilor de optimizare și analiza dependențelor dintre operații.

2.3 Mașini cu două adrese

O arhitectură cu două adrese reduce numărul de câmpuri explicite din instrucțiune. În acest model, unul dintre operanzi este utilizat simultan ca sursă și ca destinație a rezultatului. Această abordare conduce la instrucțiuni mai scurte decât în cazul mașinilor cu trei adrese, dar impune uneori utilizarea unor instrucțiuni suplimentare pentru copierea sau încărcarea valorilor în registre [65, 82].

Forma generală a unei instrucțiuni cu două adrese este:

```
1  ADD OP1, OP2
```

Semnificația operației este:

$$OP1 = OP1 + OP2.$$

Prin urmare, primul operand are un rol dublu: participă la calcul și primește rezultatul. Această proprietate face codul mai compact, dar poate reduce claritatea programului, deoarece valoarea inițială a primului operand este suprascrisă.

Figura 2.3 prezintă structura unei instrucțiuni cu două adrese.

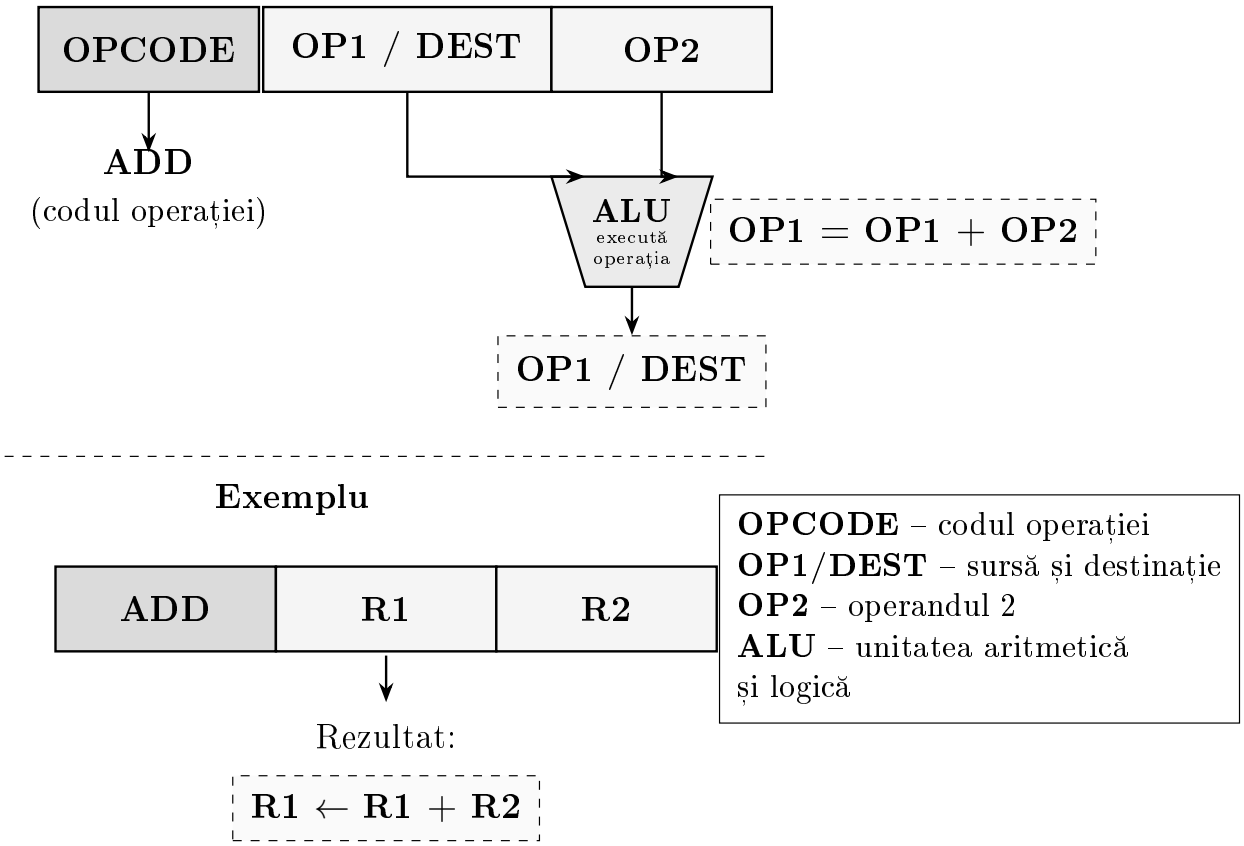


Figura 2.3 Structura unei instrucțiuni cu două adrese. Primul operand este utilizat atât ca sursă, cât și ca destinație a rezultatului.

Pentru expresia:

$$A = B + C \times D - E + F + A \tag{2.2}$$

o implementare posibilă este:

```
1  LOAD T, C
2  MUL T, D
3  ADD T, B
```

```
4  SUB T, E
5  ADD T, F
6  ADD A, T
```

Observăm că variabila temporară `T` este utilizată pentru acumularea rezultatului intermediar. Instrucțiunile aritmetice modifică direct primul operand, ceea ce reduce lungimea instrucțiunilor, dar face necesară o atenție mai mare în organizarea calculelor.

Exemplu

Multe instrucțiuni din familia x86 utilizează un model apropiat de cel cu două adrese. De exemplu, instrucțiunea `ADD EAX, EBX` adună valoarea din registrul `EBX` la valoarea din registrul `EAX`, iar rezultatul este memorat tot în `EAX`. Astfel, registrul `EAX` este simultan operand sursă și destinație [43].

Mașinile cu două adrese reprezintă un compromis între claritatea modelului cu trei adrese și compactitatea modelelor mai simple. Ele au fost utilizate pe scară largă deoarece reduc dimensiunea instrucțiunilor, fără a impune o disciplină la fel de strictă precum arhitecturile bazate pe acumulator sau pe stivă.

2.4 Mașini cu o adresă

Pe măsură ce procesoarele au evoluat, proiectanții au urmărit reducerea dimensiunii instrucțiunilor și simplificarea hardware-ului necesar decodificării acestora. O soluție adoptată în numeroase arhitecturi clasice a fost utilizarea unui registru special, numit *acumulator* (*Accumulator* – AC), care este utilizat implicit de majoritatea operațiilor aritmetice și logice. Astfel a apărut arhitectura cu o singură adresă, în care instrucțiunea specifică explicit un singur operand, celălalt operand și destinația rezultatului fiind acumulatorul [65, 82].

Forma generală a unei instrucțiuni cu o adresă este:

```
1  ADD OP
```

Semnificația operației este:

$$AC = AC + OP.$$

Rezultatul este memorat automat în acumulator, fără ca acesta să fie specificat explicit în instrucțiune. Acest mecanism reduce lungimea codului mașină și simplifică procesul de decodificare, însă impune utilizarea frecventă a instrucțiunilor de încărcare și salvare a valorilor din acumulator.

Figura 2.4 prezintă structura unei instrucțiuni cu o adresă.

Pentru expresia

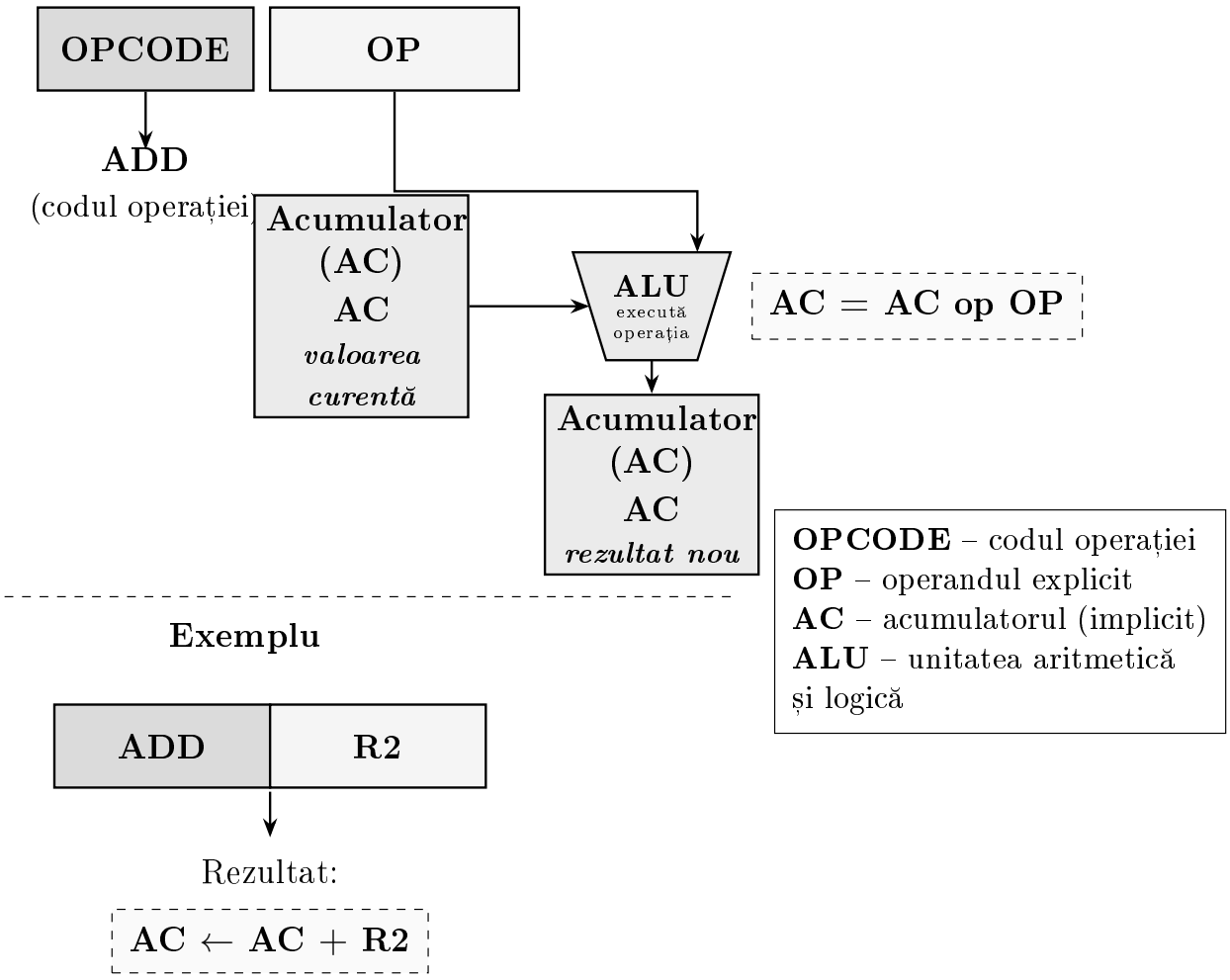


Figura 2.4 Structura unei instrucțiuni cu o adresă. Acumulatorul este utilizat implicit atât ca operand, cât și ca destinație a rezultatului.

$$A = B + C \times D - E + F + A$$

o implementare posibilă este:

```
1 LOAD C
2 MUL D
3 ADD B
4 SUB E
5 ADD F
6 ADD A
7 STORE A
```

În această secvență, toate operațiile sunt executate asupra acumulatorului. Instrucțiunea `LOAD` încarcă primul operand în registrul AC, iar fiecare operație ulterioară utilizează implicit conținutul acestuia. La final, rezultatul este memorat în locația de memorie corespunzătoare

variabilei `A`.

Exemplu

Primele generații de microprocesoare, precum Intel 8080 și MOS Technology 6502, utilizau extensiv registre de tip acumulator. Chiar dacă arhitecturile moderne folosesc în principal registre generale, conceptul de acumulator rămâne important pentru înțelegerea evoluției arhitecturilor procesoarelor și a modului în care au fost proiectate primele sisteme de calcul.

Principalul avantaj al arhitecturilor cu o adresă îl reprezintă dimensiunea redusă a instrucțiunilor și simplitatea hardware-ului necesar implementării procesorului. În schimb, utilizarea unui singur registru implicit conduce la un număr mai mare de accesări ale memoriei și la o flexibilitate redusă în organizarea calculelor complexe.

2.5 Mașini cu zero adrese

Arhitecturile cu zero adrese reprezintă forma cea mai compactă de organizare a instrucțiunilor. Spre deosebire de modelele prezentate anterior, instrucțiunile nu mai conțin câmpuri dedicate operanzilor. Toate operațiile utilizează implicit o structură de date specială, denumită *stivă* (*Stack*), în care operanzii sunt adăugați și extrași conform principiului *Last-In, First-Out* (LIFO) [38, 65, 82].

Într-o mașină cu zero adrese, operațiile aritmetice utilizează automat primele două elemente aflate în vârful stivei. Acestea sunt extrase din stivă, prelucrate de unitatea aritmetică și logică (ALU), iar rezultatul este introdus din nou în vârful stivei. Deoarece operanzii sunt determinați implicit de poziția lor în stivă, instrucțiunile nu mai trebuie să conțină adrese explicite, ceea ce conduce la o lungime minimă a codului mașină.

Forma generală a unei instrucțiuni este:

```
1 ADD
```

Semnificația operației este:

$$TOS = TOS_1 + TOS_2,$$

unde TOS (*Top Of Stack*) reprezintă elementul aflat în vârful stivei.

Figura 2.5 ilustrează funcționarea unei mașini cu zero adrese.

Considerând aceeași expresie

$$A = B + C \times D - E + F + A,$$

o posibilă implementare utilizând o mașină bazată pe stivă este:

Exemplu 2.1 Exemplu de implementare pe o mașină cu zero adrese

```
1 PUSH B
```

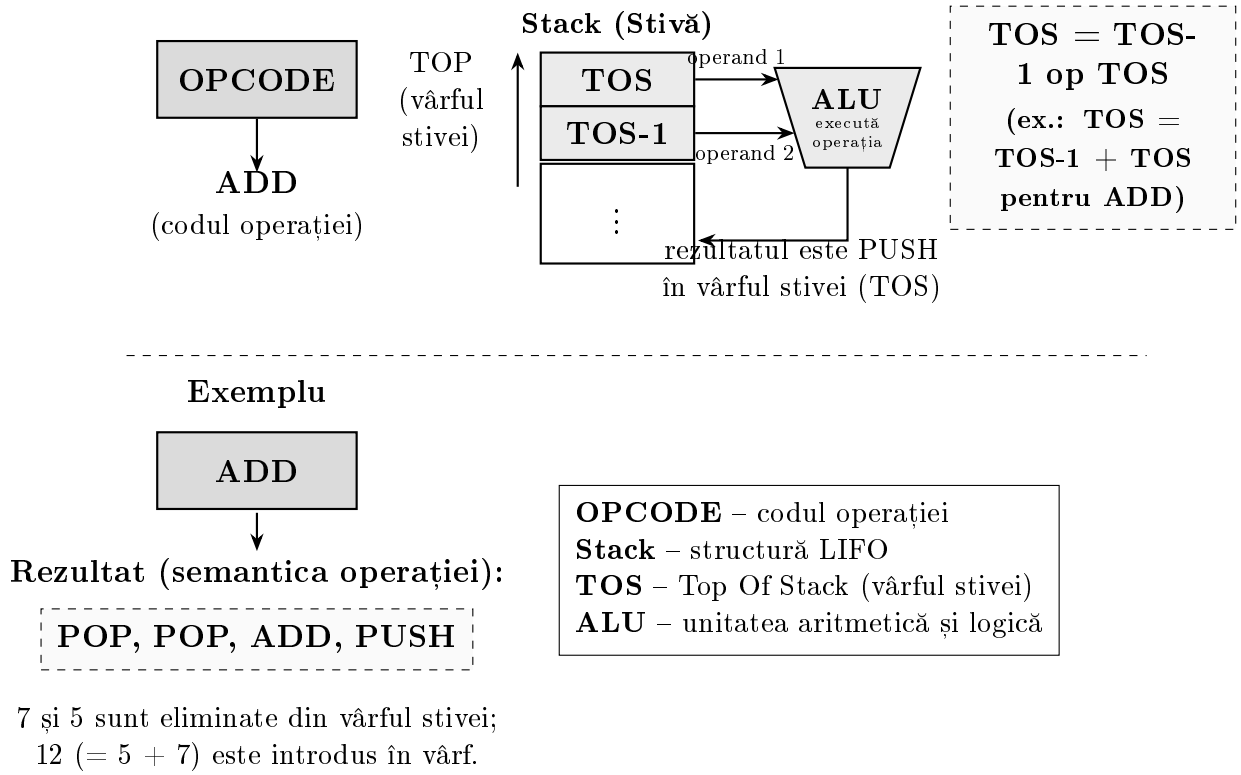


Figura 2.5 Structura unei instrucțiuni cu zero adrese. Operanzii sunt preluați implicit din vârful stivei.

```
2  PUSH C
3  PUSH D
4  MUL
5  ADD
6  PUSH E
7  SUB
8  PUSH F
9  ADD
10 PUSH A
11 ADD
12 POP A
```

Observăm că operațiile aritmetice nu specifică niciun operand. Instrucțiunile `MUL`, `ADD` și `SUB` utilizează automat valorile aflate în vârful stivei, simplificând astfel formatul instrucțiunilor.

Exemplu

Mașina virtuală Java (Java Virtual Machine – JVM) utilizează o arhitectură bazată pe stivă. Instrucțiunile bytecode, precum `iadd`, `imul` sau `isub`, operează implicit asupra elementelor aflate în vârful stivei de execuție. Această abordare contribuie la portabilitatea aplicațiilor Java și simplifică proiectarea mașinii virtuale [63].

Principalul avantaj al arhitecturilor bazate pe stivă este dimensiunea redusă a instrucțiunilor și simplitatea formatului acestora. În schimb, performanța poate fi afectată de numărul mare de operații de încărcare și salvare în stivă, iar accesul direct la valori intermediare este mai puțin flexibil decât în arhitecturile bazate pe registre.

2.6 Compararea modelelor de adresare

Cele patru modele de adresare prezentate anterior ilustrează evoluția arhitecturilor procesoarelor și compromisurile realizate între flexibilitatea programării, dimensiunea instrucțiunilor și complexitatea hardware-ului. Alegerea unui anumit model nu este întâmplătoare, ci reflectă cerințele tehnologice și limitările existente în perioada în care au fost proiectate respectivele arhitecturi [38, 65, 82].

Arhitecturile cu trei adrese oferă cea mai mare flexibilitate, deoarece fiecare instrucțiune specifică explicit cei doi operanzi și locația în care va fi memorat rezultatul. Această abordare conduce la un cod ușor de urmărit și apropiat de reprezentarea expresiilor matematice din limbajele de nivel înalt. Dezavantajul constă în dimensiunea mai mare a instrucțiunilor și într-o logică hardware mai complexă.

Reducerea numărului de operanzi expliciți conduce la instrucțiuni mai compacte. În arhitecturile cu două adrese, unul dintre operanzi este utilizat simultan și ca destinație a rezultatului, ceea ce reduce dimensiunea codului executabil. În arhitecturile cu o adresă, utilizarea unui acumulator implicit simplifică și mai mult formatul instrucțiunilor, însă introduce un număr suplimentar de operații de încărcare și salvare a datelor.

Modelul bazat pe stivă utilizează cele mai scurte instrucțiuni dintre toate variantele prezentate. Operanzii sunt determinați implicit de poziția lor în stivă, iar instrucțiunile nu mai conțin adrese explicite. Deși această abordare reduce dimensiunea codului, accesul indirect la operanzi poate conduce la un număr mai mare de operații asupra memoriei și la o flexibilitate redusă în optimizarea execuției.

Figura 2.6 sintetizează principalele caracteristici ale celor patru modele.

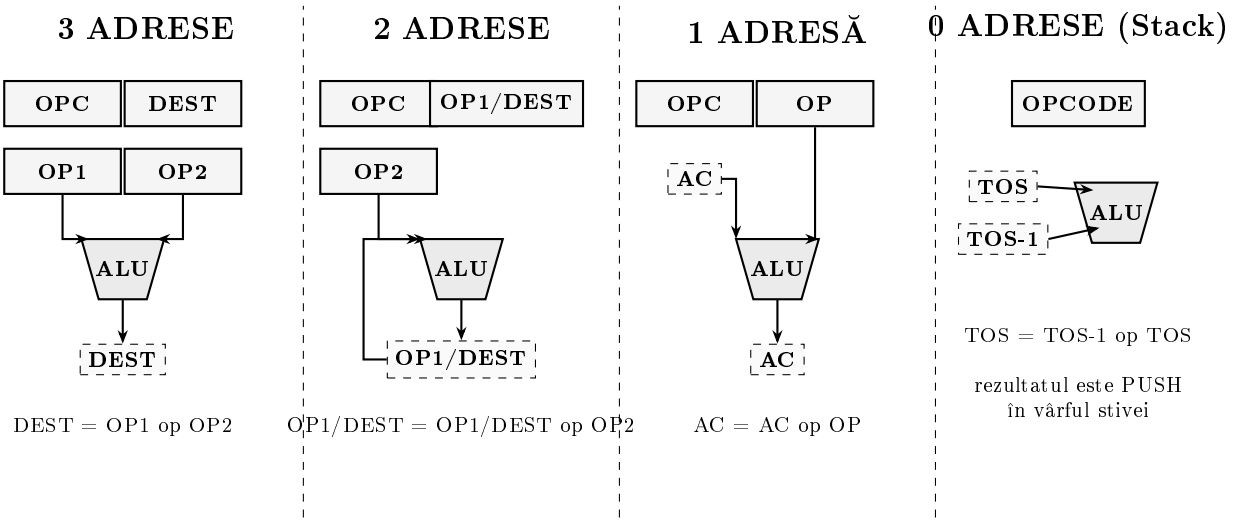


Figura 2.6 Compararea modelelor de adresare utilizate în arhitectura procesoarelor.

Tabelul 2.1 evidențiază diferențele dintre cele patru modele.

Tabela 2.1 Compararea modelelor de adresare.

Caracteristică	3 adrese	2 adrese	1 adresă	0 adrese
Număr operanzi expliciți	3	2	1	0
Registru implicit	Nu	Nu	Acumulator	Stivă
Lungime instrucțiune	Mare	Medie	Mică	Foarte mică
Complexitate hardware	Ridicată	Medie	Redusă	Redusă
Ușurința programării	Foarte bună	Bună	Medie	Redusă
Dimensiunea codului	Mare	Medie	Mică	Foarte mică

În prezent, majoritatea procesoarelor comerciale utilizează arhitecturi bazate pe registre generale, deoarece acestea oferă cel mai bun compromis între performanță și flexibilitate. Totuși, înțelegerea modelelor istorice rămâne importantă, deoarece multe dintre conceptele introduse de acestea sunt utilizate și în arhitecturile moderne.

2.7 Arhitectura CISC

Arhitecturile de tip *Complex Instruction Set Computer* (CISC) au fost dezvoltate în perioada în care memoria principală era costisitoare, iar compilatoarele dispuneau de posibilități limitate de optimizare. Obiectivul principal al acestui model a fost reducerea numărului de instrucțiuni necesare implementării unui algoritm prin introducerea unor instrucțiuni complexe, capabile să execute operații multiple într-un singur ciclu aparent de programare [38, 82].

Procesoarele CISC includ de regulă sute de instrucțiuni, numeroase moduri de adresare și instrucțiuni cu lungime variabilă. Multe dintre acestea pot efectua simultan accesări ale memoriei, operații aritmetice și actualizarea registrelor. Din punctul de vedere al programatorului, acest lucru conduce la programe mai compacte și mai ușor de exprimat.

Implementarea internă a unui procesor CISC este însă considerabil mai complexă. Instrucțiunile sunt decodificate și transformate frecvent în micro-operații interne, executate ulterior de unitățile funcționale ale procesorului. Acest mecanism este realizat prin intermediul microcodului, care permite implementarea instrucțiunilor complexe fără modificarea hardware-ului principal.

Figura 2.7 prezintă procesul general de execuție al unei instrucțiuni CISC.

Procesoarele din familia Intel x86 reprezintă cel mai cunoscut exemplu de arhitectură CISC. Deși instrucțiunile sunt complexe la nivelul ISA, implementările moderne traduc

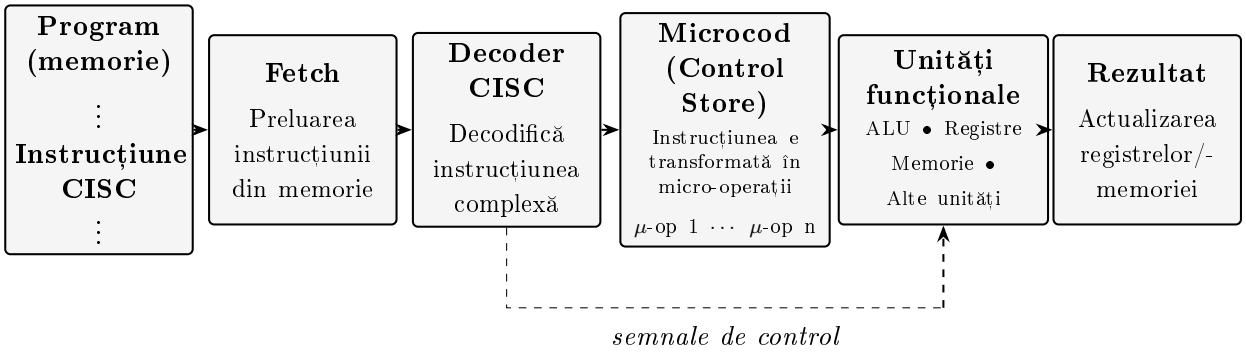


Figura 2.7 Execuția unei instrucțiuni într-o arhitectură CISC.

instrucțiunile x86 în micro-operații interne asemănătoare celor utilizate în arhitecturile RISC [43].

2.8 Arhitectura RISC

Conceptul *Reduced Instruction Set Computer* (RISC) a apărut ca rezultat al studiilor efectuate la Universitatea Berkeley și la Universitatea Stanford în anii 1980. Cercetările au arătat că majoritatea programelor utilizează doar un număr redus dintre instrucțiunile disponibile într-o arhitectură CISC. În consecință, s-a propus simplificarea setului de instrucțiuni și optimizarea execuției acestora [38, 65].

Procesoarele RISC utilizează instrucțiuni de lungime fixă, un număr mare de registre generale și o arhitectură de tip *load/store*. Operațiile aritmetice sunt efectuate exclusiv asupra registrelor, iar accesul la memorie este realizat numai prin instrucțiuni dedicate de încărcare și stocare.

Această organizare simplifică procesul de decodificare și permite implementarea eficientă a execuției în pipeline, contribuind la creșterea performanței procesorului.

Figura 2.8 prezintă fluxul general de execuție într-o arhitectură RISC.

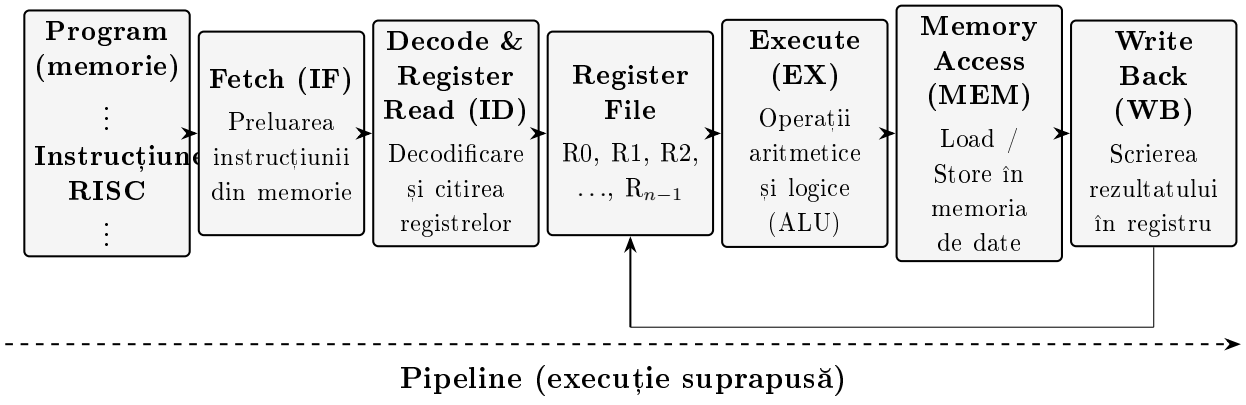


Figura 2.8 Fluxul general de execuție într-o arhitectură RISC.

Majoritatea microcontrolerelor moderne utilizează arhitecturi RISC. Exemple reprezentative sunt familiile ARM Cortex-M, AVR și RISC-V, utilizate pe scară largă în sistemele

embedded datorită eficienței energetice și performanțelor ridicate.

2.9 Comparația dintre arhitecturile RISC și CISC

Deși arhitecturile RISC și CISC au fost considerate mult timp abordări concurente, implementările moderne împrumută frecvent elemente din ambele concepte. Procesoarele x86 utilizează o ISA de tip CISC, însă execută intern micro-operații asemănătoare celor din arhitecturile RISC, în timp ce procesoarele ARM moderne includ mecanisme complexe de predicție, execuție speculativă și execuție în afara ordinii.

Compararea celor două abordări trebuie realizată din perspectiva compromisurilor inginerești și nu prin clasificarea uneia dintre ele ca fiind superioară celeilalte.

Figura 2.9 sintetizează diferențele fundamentale dintre cele două arhitecturi.

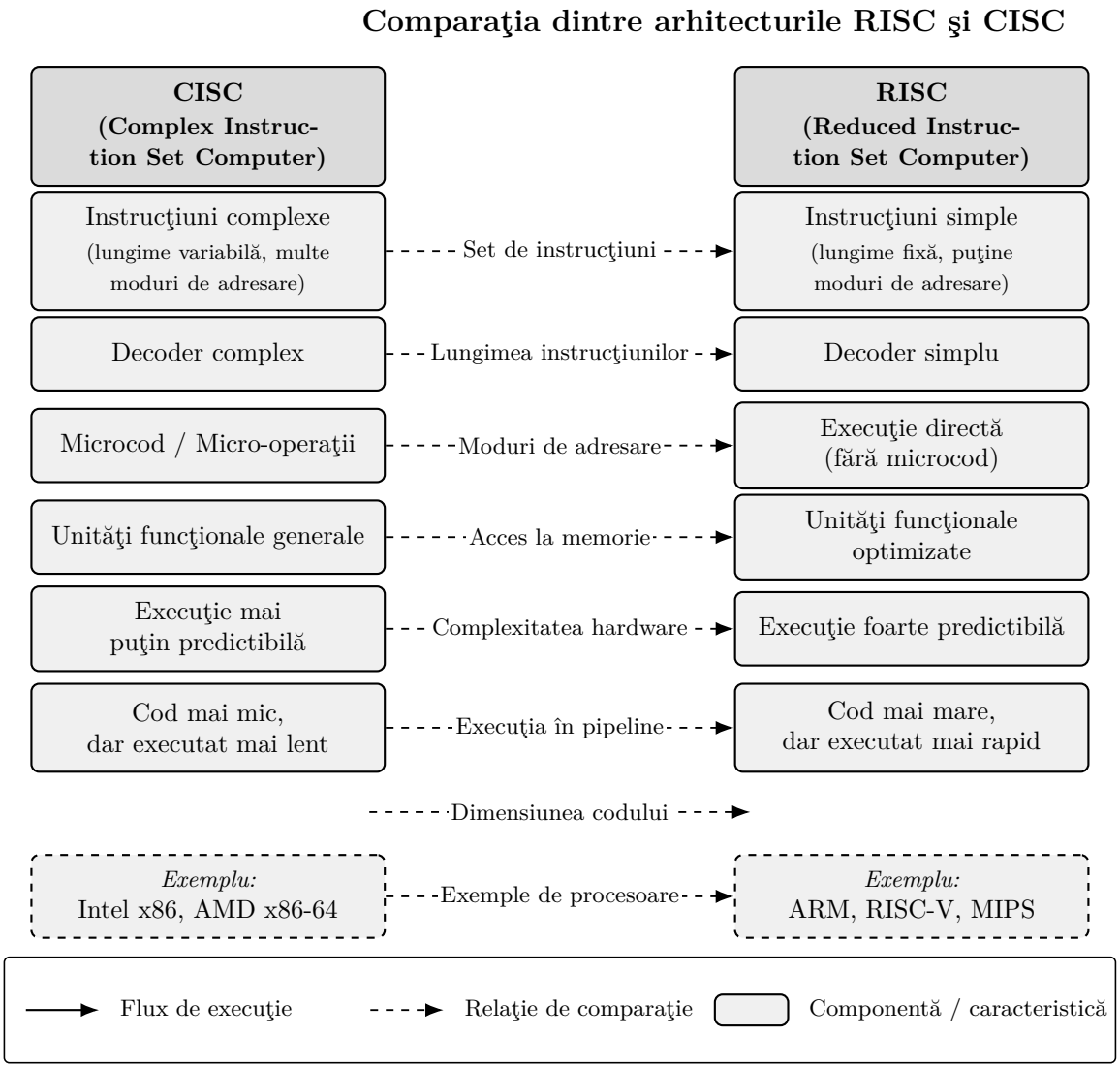


Figura 2.9 Comparația conceptuală dintre arhitecturile RISC și CISC.

În prezent, diferențele dintre cele două familii sunt mai puțin pronunțate decât în trecut. Tendința actuală este reprezentată de combinarea avantajelor ambelor modele, astfel încât

Tabela 2.2 Compararea arhitecturilor RISC și CISC.

Caracteristică	RISC	CISC
Complexitatea instrucțiunilor	Redusă	Ridicată
Lungimea instrucțiunilor	Fixă	Variabilă
Număr registre	Mare	Redus/mediu
Pipeline	Foarte eficient	Mai dificil
Acces la memorie	Load/Store	Direct în instrucțiuni
Complexitate hardware	Redusă	Ridicată
Dimensiunea codului	Mai mare	Mai redusă
Exemple	ARM, AVR, RISC-V	Intel x86, AMD64

performanța ridicată să fie obținută fără sacrificarea compatibilității software.

2.10 Performanța unui procesor

Performanța reprezintă unul dintre cele mai importante criterii utilizate în evaluarea unui sistem de calcul. În cazul sistemelor embedded, performanța nu se rezumă doar la viteza de execuție, ci trebuie analizată împreună cu consumul energetic, costul implementării și constrângerile impuse de aplicație. Un controler utilizat într-un sistem de frânare al unui autovehicul trebuie să răspundă într-un timp determinist, în timp ce un procesor destinat unui server urmărește maximizarea numărului de operații executate într-o unitate de timp.

În mod intuitiv, performanța unui procesor este cu atât mai mare cu cât timpul necesar executării unei aplicații este mai mic. Din acest motiv, evaluarea performanței începe întotdeauna prin măsurarea timpului de execuție al unui program reprezentativ pentru aplicația analizată. Această abordare este preferabilă comparării frecvențelor de ceas, deoarece două procesoare cu aceeași frecvență pot avea performanțe foarte diferite dacă utilizează arhitecturi distincte sau mecanisme diferite de execuție [36, 38, 65].

Figura 2.10 prezintă principalii factori care influențează performanța unui procesor.

În practică, performanța depinde de trei categorii principale de factori: caracteristicile arhitecturii procesorului, eficiența codului executat, precum și compilatorul și optimizările aplicate acestuia.

Aceste elemente determină împreună timpul necesar executării unei aplicații și permit compararea obiectivă a diferitelor arhitecturi hardware.

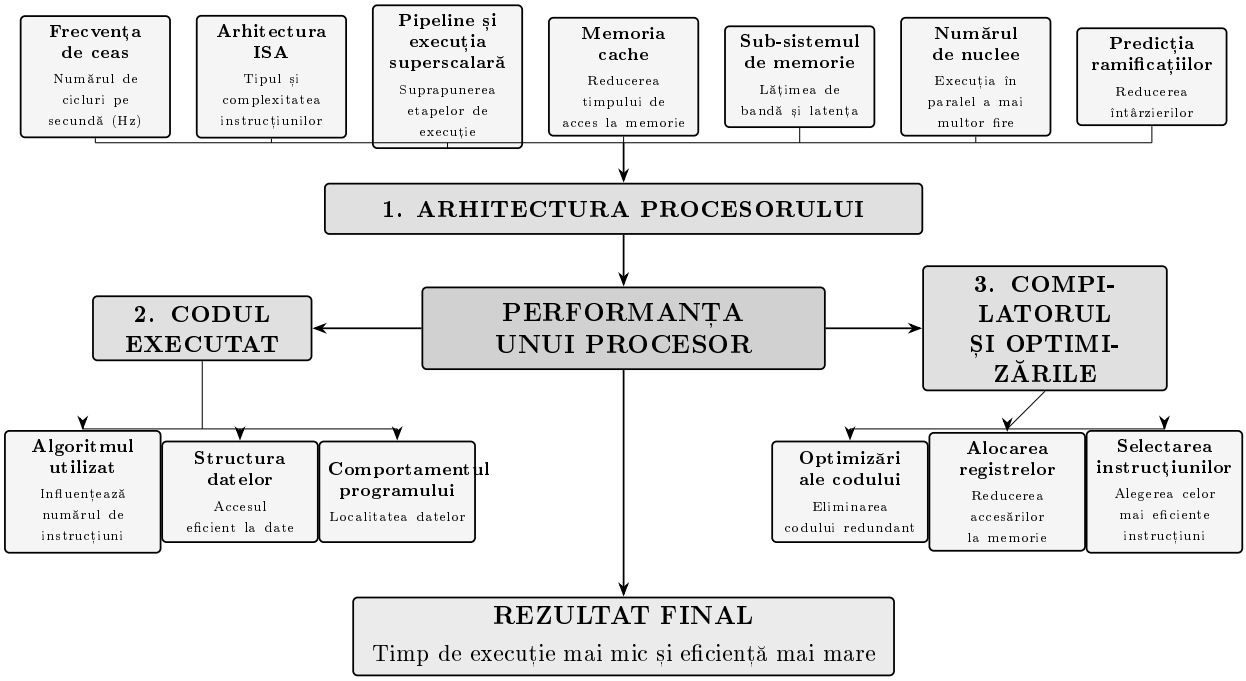


Figura 2.10 Factorii principali care influențează performanța unui procesor.

2.10.1 Timpul de execuție

Indicatorul fundamental utilizat pentru evaluarea performanței este timpul de execuție (*Execution Time*). Acesta reprezintă intervalul de timp necesar procesorului pentru executarea completă a unei aplicații.

Performanța este invers proporțională cu timpul de execuție:

$$Performance = \frac{1}{Execution\ Time} \tag{2.3}$$

În arhitectura calculatoarelor este utilizată relația fundamentală

$$Execution\ Time = Instruction\ Count \times CPI \times Clock\ Cycle \tag{2.4}$$

unde **Instruction Count (IC)** reprezintă numărul total de instrucțiuni executate, **CPI** (*Cycles Per Instruction*) reprezintă numărul mediu de cicluri necesare executării unei instrucțiuni, iar **Clock Cycle** reprezintă durata unui ciclu de ceas.

Ecuația (2.4) evidențiază faptul că reducerea timpului de execuție poate fi realizată prin diminuarea numărului de instrucțiuni executate, prin reducerea valorii CPI sau prin creșterea frecvenței procesorului.

Exemplu

Un procesor execută un program alcătuit din 500 milioane de instrucțiuni. Dacă valoarea medie este $CPI = 2$, iar frecvența este de 1 GHz, timpul de execuție devine

$$Execution\ Time = \frac{500 \cdot 10^6 \times 2}{10^9} = 1\,s$$

Prin reducerea valorii CPI la 1.5, timpul de execuție scade la 0.75 s fără modificarea frecvenței procesorului.

2.10.2 Frecvența de ceas

Frecvența de ceas reprezintă numărul de cicluri generate de oscilator într-o secundă și se exprimă în hertzi (Hz). Procesoarele moderne funcționează în domeniul sutelor de MHz sau al câtorva GHz.

Durata unui ciclu de ceas este invers proporțională cu frecvența:

$$Clock\ Cycle = \frac{1}{Clock\ Frequency} \tag{2.5}$$

De exemplu,

Frecvență	Durata ciclului
100 MHz	10 ns
500 MHz	2 ns
1 GHz	1 ns
2 GHz	0.5 ns

Creșterea frecvenței conduce, în general, la reducerea timpului de execuție. Totuși, aceasta implică un consum energetic mai ridicat și dificultăți suplimentare privind disiparea căldurii. Din acest motiv, arhitecturile moderne urmăresc îmbunătățirea performanței și prin alte mecanisme, precum execuția în pipeline, execuția superscalară sau optimizarea memoriei cache.

2.10.3 Numărul de instrucțiuni executate

Numărul total de instrucțiuni executate depinde atât de algoritmul implementat, cât și de eficiența compilatorului. Două programe care rezolvă aceeași problemă pot necesita un număr diferit de instrucțiuni, în funcție de optimizările aplicate.

De asemenea, aceeași aplicație poate genera cod diferit pentru două arhitecturi ISA distincte. De exemplu, un compilator destinat arhitecturii ARM poate produce un număr diferit de instrucțiuni față de un compilator destinat arhitecturii x86, chiar dacă funcționalitatea aplicației este identică.

Reducerea numărului de instrucțiuni reprezintă una dintre principalele direcții urmărite de compilatoarele moderne, deoarece influențează direct timpul total de execuție.

2.11 CPI și IPC

2.11.1 CPI

CPI (*Cycles Per Instruction*) reprezintă numărul mediu de cicluri de ceas necesare pentru executarea unei instrucțiuni și constituie indicatorul standard prin care este exprimată eficiența cu care o arhitectură își execută propriul set de instrucțiuni:

$$CPI = \frac{Clock\ Cycles}{Instruction\ Count} \tag{2.6}$$

Cu cât valoarea CPI este mai mică, cu atât procesorul finalizează, în medie, o instrucțiune într-un număr mai redus de cicluri, ceea ce indică o utilizare mai eficientă a resurselor interne. În practică, instrucțiunile nu au același cost: o operație aritmetică simplă poate necesita un singur ciclu, în timp ce un acces la memorie sau o instrucțiune de salt condiționat pot necesita mai multe. CPI-ul mediu al unui program se obține prin ponderarea CPI-ului fiecărui tip de instrucțiune cu frecvența sa relativă de apariție:

$$CPI_{avg} = \sum_{i=1}^{n} (CPI_i \times Frequency_i) \tag{2.7}$$

unde CPI_i este numărul de cicluri necesar instrucțiunilor de tip i, iar $Frequency_i$ este ponderea acestora în totalul instrucțiunilor executate.

2.11.2 IPC

IPC (*Instructions Per Cycle*) exprimă aceeași relație din perspectivă inversă, indicând numărul de instrucțiuni finalizate într-un ciclu de ceas:

$$IPC = \frac{Instruction\ Count}{Clock\ Cycles} \tag{2.8}$$

Cele două mărimi sunt reciproce:

$$IPC = \frac{1}{CPI} \tag{2.9}$$

Un procesor scalar clasic, care finalizează cel mult o instrucțiune pe ciclu, are un IPC limitat superior la 1. Procesoarele superscalare moderne pot depăși această limită, deoarece dispun de unități de execuție multiple capabile să lanseze și să finalizeze simultan mai multe instrucțiuni în același ciclu de ceas.

2.12 Exemple de calcul și studii de caz

Exemplu

Un procesor execută un program alcătuit din 500 000 de instrucțiuni, cu $CPI = 2$, la o frecvență de 200 MHz. Timpul de execuție rezultă din ecuația (2.4):

$$CPU\ Time = \frac{500\,000 \times 2}{200 \times 10^6} = 0{,}005\,s = 5\,ms$$

Exemplu

Două procesoare execută același program de 1 000 000 de instrucțiuni. Procesorul A funcționează la 1 GHz cu $CPI = 2$, iar procesorul B funcționează la 800 MHz cu $CPI = 1$:

$$T_A = \frac{1\,000\,000 \times 2}{10^9} = 2\,ms \qquad T_B = \frac{1\,000\,000 \times 1}{800 \times 10^6} = 1{,}25\,ms$$

Deși procesorul B are o frecvență mai mică, acesta oferă performanța mai bună, datorită unui CPI redus. Acest rezultat confirmă faptul că frecvența de ceas, luată izolat, nu reprezintă un indicator suficient pentru evaluarea performanței unui procesor.

Întrebări de verificare

1. Ce reprezintă Instruction Set Architecture și care este rolul acestuia în relația dintre hardware și software?
2. Care sunt avantajele și dezavantajele unei mașini cu trei adrese, respectiv ale unei mașini cu o adresă?
3. Cum funcționează o mașină bazată pe stivă (zero adrese)?
4. Care sunt principalele caracteristici ale arhitecturilor CISC și, respectiv, RISC?
5. De ce arhitecturile RISC sunt preferate în sistemele embedded?
6. Definiți noțiunile de CPI și IPC și precizați relația dintre ele.
7. Scrieți ecuația performanței procesorului și explicați semnificația fiecărui termen.
8. Care sunt factorii care influențează timpul de execuție al unui program?
9. De ce frecvența de ceas, luată izolat, nu este suficientă pentru evaluarea performanței?
10. Calculați timpul de execuție pentru un program care execută 2 000 000 de instrucțiuni, are $CPI = 2$ și rulează la 500 MHz.
11. Explicați avantajele modelului Load/Store.
12. Dați exemple de procesoare RISC și CISC și explicați de ce procesoarele moderne combină idei din ambele filozofii de proiectare.

3. Arhitectura Memoriei și Procesoarele ARM

3.1 Arhitectura von Neumann

Arhitectura von Neumann reprezintă unul dintre modelele fundamentale utilizate pentru descrierea organizării unui sistem de calcul. Originea sa este asociată conceptului de program memorat, formulat în contextul proiectului EDVAC și prezentat în raportul redactat de John von Neumann în anul 1945. Ideea centrală constă în stocarea instrucțiunilor programului și a datelor în aceeași memorie, astfel încât procesorul să le poată accesa utilizând un mecanism comun de adresare [65, 82, 90].

Într-un asemenea sistem, memoria nu face o distincție structurală între o instrucțiune și o valoare numerică. Ambele sunt reprezentate prin secvențe binare, iar semnificația lor este stabilită de contextul în care sunt utilizate de procesor. Această proprietate permite încărcarea programelor în memorie, modificarea lor și utilizarea acelorași mecanisme hardware pentru transferul atât al instrucțiunilor, cât și al datelor.

Din punct de vedere funcțional, procesorul conține unitatea de control, unitatea aritmetică și logică, precum și un set de registre interne. Memoria principală păstrează programul și datele, iar perifericele permit interacțiunea cu mediul exterior. Comunicarea dintre aceste componente este realizată prin magistrale de adrese, de date și de control.

Figura 3.1 prezintă organizarea conceptuală a unei arhitecturi von Neumann.

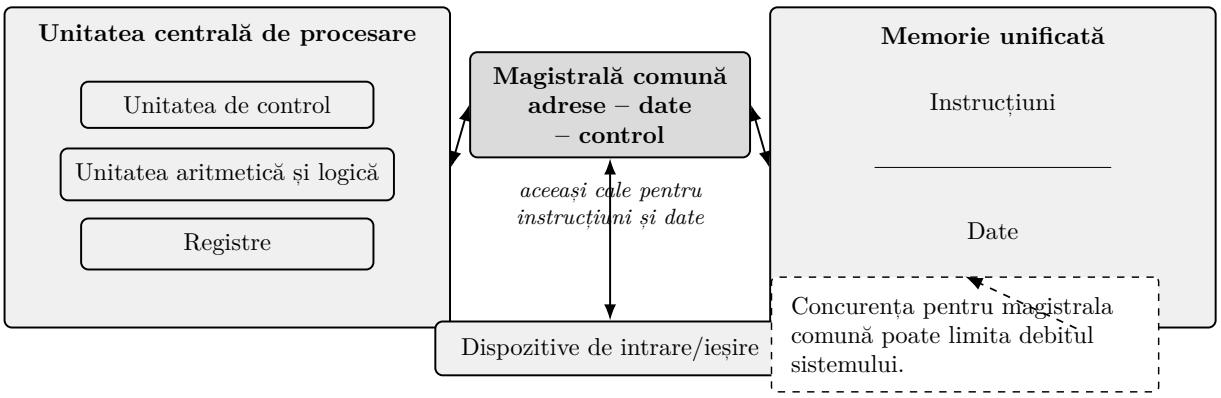


Figura 3.1 Organizarea conceptuală a unei arhitecturi von Neumann. Instrucțiunile și datele sunt păstrate într-o memorie comună și utilizează aceeași cale de transfer către procesor.

Avantajul principal al acestei organizări este flexibilitatea. Spațiul de memorie poate fi distribuit dinamic între program și date, fără ca dimensiunea celor două regiuni să fie stabilită prin structura hardware. De asemenea, existența unui singur spațiu de adrese simplifică modelul de programare și permite tratarea codului ca informație memorată.

Utilizarea aceleiași căi pentru instrucțiuni și date introduce însă o limitare importantă. Procesorul nu poate utiliza simultan o singură interfață de memorie pentru preluarea unei instrucțiuni și pentru transferul unui operand. Cele două operații trebuie arbitrate sau executate succesiv, iar viteza sistemului ajunge să fie limitată de debitul comunicației dintre procesor și memorie.

Această problemă este cunoscută sub denumirea de *von Neumann bottleneck*. În sens modern, ea nu se referă doar la existența unei magistrale comune, ci la diferența tot mai mare dintre viteza cu care procesorul poate executa operații și viteza cu care memoria îi poate furniza instrucțiuni și date [14, 38]. Memoriile cache, preluarea anticipată a instrucțiunilor și transferurile în rafală reduc efectele acestui blocaj, fără a elimina însă complet problema.

Exemplu

Să presupunem că procesorul trebuie să execute instrucțiunea care calculează suma a două valori aflate în memorie. Procesorul trebuie mai întâi să preia instrucțiunea, apoi să citească cei doi operanzi și, în final, să scrie rezultatul. Dacă toate transferurile folosesc aceeași interfață de memorie, ele concurează pentru aceeași resursă. Chiar dacă unitatea aritmetică poate efectua adunarea într-un singur ciclu, timpul total poate fi dominat de accesările memoriei.

Modelul von Neumann rămâne esențial pentru înțelegerea sistemelor de calcul, însă proce-

soarele contemporane implementează numeroase optimizări microarhitecturale care nu sunt vizibile direct programatorului. Din perspectiva software-ului, memoria poate apărea ca un spațiu unificat, chiar dacă în interiorul procesorului instrucțiunile și datele urmează căi diferite.

3.2 Arhitectura Harvard

Arhitectura Harvard descrie o organizare în care instrucțiunile și datele sunt păstrate în memorii distincte și sunt transferate către procesor prin căi separate. Cele două memorii formează, în varianta clasică, spații de adrese independente și pot avea dimensiuni ale cuvântului, capacități și tehnologii de implementare diferite [65, 82].

Această separare permite procesorului să preia instrucțiunea următoare în același timp în care instrucțiunea curentă citește sau scrie un operand. În acest mod, accesul la program nu mai concurează direct cu accesul la date, iar debitul sistemului poate crește. Proprietatea este deosebit de utilă în procesoarele care utilizează execuția în pipeline și în aplicațiile care efectuează în mod repetat operații asupra fluxurilor de date.

Figura 3.2 ilustrează organizarea generală a unei arhitecturi Harvard.

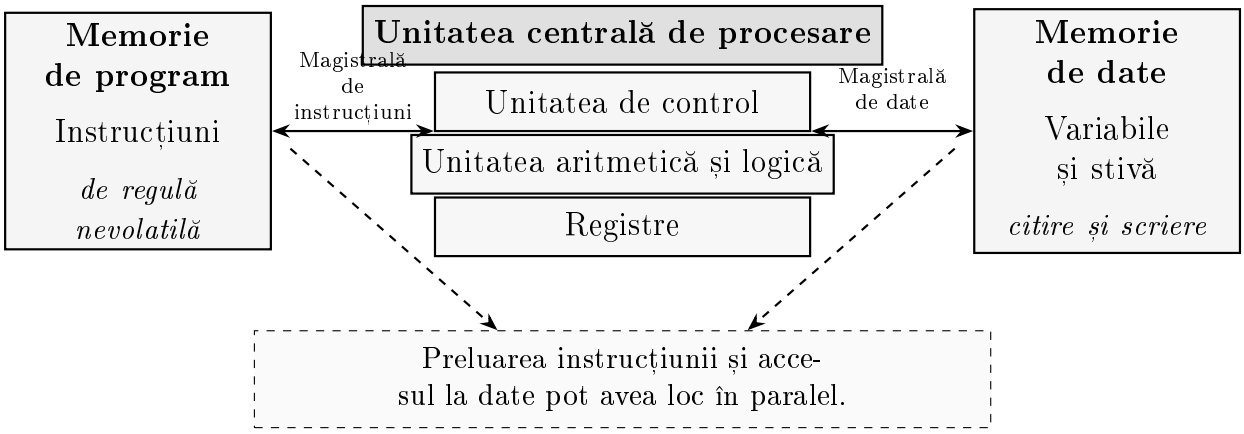


Figura 3.2 Organizarea conceptuală a unei arhitecturi Harvard. Memoria de program și memoria de date sunt accesate prin magistrale distincte.

Separarea fizică permite adaptarea fiecărei memorii la rolul său. Memoria de program poate utiliza, de exemplu, o tehnologie nevolatilă și un cuvânt suficient de larg pentru stocarea unei instrucțiuni complete, în timp ce memoria de date poate utiliza o tehnologie rapidă de tip SRAM și o organizare optimizată pentru citiri și scrieri frecvente.

Această flexibilitate este avantajoasă în sistemele embedded, unde programul este păstrat de regulă în memoria Flash, iar variabilele sunt stocate în memoria RAM. De asemenea, separarea spațiilor poate împiedica executarea accidentală a informațiilor din memoria de date, deși nu trebuie considerată, în sine, un mecanism complet de securitate.

Arhitectura Harvard pură introduce și anumite dificultăți. Deoarece programul și datele ocupă spații de adrese diferite, procesorul nu poate accesa în mod obișnuit memoria de program folosind aceleași instrucțiuni utilizate pentru memoria de date. Constantele, tabelele de căutare și actualizarea firmware-ului necesită mecanisme speciale. În plus, capacitatea

neutilizată dintr-o memorie nu poate fi transferată automat celeilalte memorii.

Exemplu

Un microcontroler poate utiliza o memorie Flash de 256 KB pentru program și o memorie SRAM de 64 KB pentru variabile. Chiar dacă aplicația utilizează numai jumătate din memoria Flash, spațiul rămas nu poate fi folosit direct pentru extinderea stivei sau a variabilelor dinamice. Cele două memorii au roluri și spații de adresare diferite.

Din punct de vedere istoric, denumirea este asociată frecvent cu mașinile construite la Harvard sub coordonarea lui Howard Aiken. Totuși, cercetări istorice recente arată că expresia *Harvard architecture* a fost introdusă ulterior și aplicată retrospectiv acestor sisteme. Din acest motiv, în această carte termenul este utilizat în sensul său tehnic contemporan: separarea memoriilor și a căilor de acces pentru program și date [66].

3.3 Arhitectura Harvard modificată

Separarea strictă a programului și a datelor îmbunătățește debitul, dar limitează flexibilitatea software-ului. În același timp, o memorie complet unificată este ușor de programat, însă poate introduce concurență între preluarea instrucțiunilor și accesarea operanzilor. Arhitectura Harvard modificată urmărește combinarea avantajelor celor două modele.

Nu există o singură implementare universală a acestei arhitecturi. Într-o variantă frecvent întâlnită în procesoarele moderne, memoria principală și spațiul de adrese sunt unificate, dar procesorul utilizează cache-uri distincte pentru instrucțiuni și date. Cache-ul de instrucțiuni, denumit frecvent *I-cache*, furnizează codul executabil, iar cache-ul de date, denumit *D-cache*, gestionează operanzii. Procesorul poate accesa simultan cele două cache-uri, chiar dacă informațiile provin, la nivelurile inferioare ale ierarhiei, din aceeași memorie principală [38, 65].

O altă variantă, întâlnită în microcontrolere, păstrează memoria Flash și memoria SRAM ca resurse distincte, dar introduce mecanisme controlate prin care programul poate citi constante din memoria Flash sau poate modifica firmware-ul. Din perspectiva căilor de acces, organizarea se comportă asemănător unei arhitecturi Harvard; din perspectiva software-ului, separarea nu mai este absolută.

Figura 3.3 prezintă o organizare tipică bazată pe cache-uri separate și memorie principală unificată.

Această organizare permite suprapunerea preluării unei instrucțiuni cu accesarea datelor și păstrează, în același timp, avantajele unui spațiu de memorie flexibil. Implementarea necesită însă mecanisme suplimentare pentru menținerea coerenței dintre cache-uri și memoria principală.

Problema devine vizibilă atunci când programul modifică o regiune de memorie care urmează să fie executată drept cod. Datele scrise pot rămâne temporar în cache-ul de date, în timp ce cache-ul de instrucțiuni păstrează o copie mai veche. Înaintea execuției, software-ul de sistem trebuie să asigure sincronizarea și invalidarea corespunzătoare a cache-urilor.

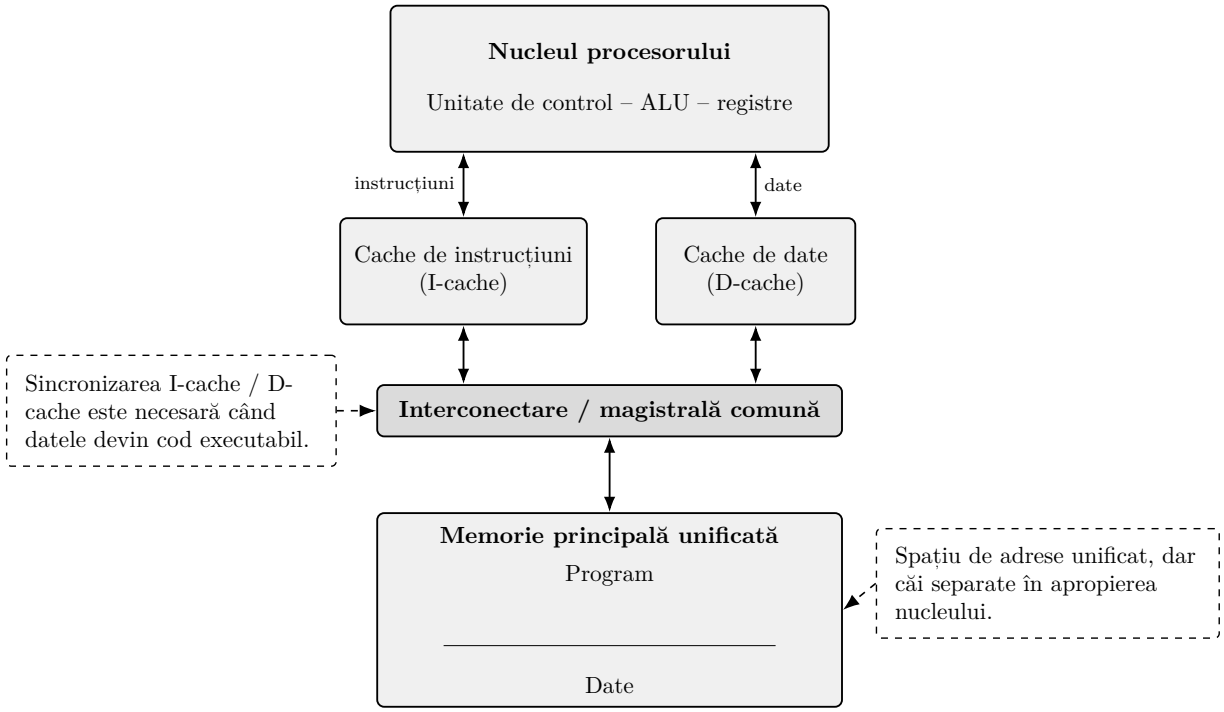


Figura 3.3 Exemplu de arhitectură Harvard modificată. Procesorul utilizează căi separate pentru instrucțiuni și date în apropierea nucleului, dar accesează o memorie principală comună.

Exemplu

Un sistem de operare încarcă un program dintr-un dispozitiv de stocare și îl copiază în memoria RAM. În timpul copierii, informațiile sunt tratate ca date. După finalizarea operației, aceeași regiune este interpretată ca o secvență de instrucțiuni. Într-o arhitectură cu I-cache și D-cache separate, sistemul trebuie să se asigure că nucleul va prelua noul cod și nu o copie mai veche rămasă în cache-ul de instrucțiuni.

Arhitectura Harvard modificată este foarte răspândită deoarece oferă un compromis favorabil între performanță, flexibilitate și compatibilitatea cu modelul de programare bazat pe un spațiu de adrese unificat.

3.4 Comparația arhitecturilor von Neumann, Harvard și Harvard modificată

Cele trei modele nu trebuie privite ca soluții complet opuse, ci ca puncte diferite într-un spațiu de proiectare. Alegerea organizării memoriei depinde de performanța necesară, costul implementării, predictibilitatea temporală și flexibilitatea cerută de software.

Arhitectura von Neumann oferă un model simplu și flexibil, în care memoria poate fi utilizată dinamic pentru cod și date. Arhitectura Harvard urmărește debitul și predictibilitatea prin separarea completă a resurselor. Varianta Harvard modificată introduce căi paralele în zonele critice ale procesorului, păstrând în același timp posibilitatea gestionării mai flexibile a memoriei.

Tabelul 3.1 sintetizează principalele diferențe.

Tabela 3.1 Comparația modelelor de organizare a memoriei

Caracteristică	von Neumann	Harvard	Harvard modificată
Organizarea memoriei	Programul și datele folosesc o memorie sau un spațiu comun.	Programul și datele sunt păstrate în memorii distincte.	Separarea este prezentă numai la anumite niveluri ale ierarhiei.
Căile de acces	O cale comună pentru instrucțiuni și date.	Căi complet separate.	Căi separate în apropierea procesorului, dar posibil unificate ulterior.
Acces concurent	Limitat de interfața comună.	Instrucțiunile și datele pot fi accesate simultan.	Acces simultan prin I-cache și D-cache sau prin memorii dedicate.
Spații de adrese	Unificat.	Separate.	Unificat sau parțial separat, în funcție de implementare.
Accesarea codului ca date	Directă, prin mecanismele obișnuite de memorie.	Restricționată sau imposibilă fără instrucțiuni speciale.	Posibilă prin mecanisme controlate și sincronizarea cache-urilor.
Flexibilitate	Ridicată.	Mai redusă.	Ridicată, cu o complexitate hardware suplimentară.
Predictibilitate	Accesele pot concura pentru aceeași magistrală.	Bună, deoarece fluxurile sunt separate.	Depinde de cache-uri, arbitrare și ierarhia memoriei.
Utilizări reprezentative	Model conceptual al calculatoarelor cu program memorat și unele sisteme simple.	DSP-uri și microcontrolere cu memorii strict separate.	Numeroase procesoare și microcontrolere moderne.

În practică, etichetarea unui procesor drept „von Neumann” sau „Harvard” poate ascunde numeroase detalii. Un procesor poate prezenta software-ului un spațiu de adrese unificat, dar poate utiliza intern cache-uri separate, magistrale multiple și mecanisme de preluare anticipată. Din acest motiv, analiza unui sistem real trebuie să ia în considerare atât modelul vizibil programatorului, cât și organizarea microarhitecturală.

Pentru sistemele embedded, alegerea arhitecturii influențează timpul de răspuns, consumul energetic, complexitatea software-ului și posibilitatea actualizării firmware-ului. Înțelegerea acestor modele oferă baza necesară pentru studiul procesoarelor ARM și al ierarhiei de memorie utilizate de acestea.

3.5 Familia de procesoare Arm

Arhitectura Arm reprezintă una dintre cele mai răspândite familii de arhitecturi de procesoare, fiind utilizată într-o gamă foarte largă de sisteme, de la microcontrolere cu resurse limitate până la telefoane inteligente, sisteme automotive, echipamente de rețea și servere. Succesul său este legat de utilizarea principiilor RISC, de eficiența energetică și de posibilitatea adaptării arhitecturii la cerințe foarte diferite [32, 65, 81].

Primele procesoare ARM au fost dezvoltate în anii 1980 în cadrul companiei Acorn Computers. Obiectivul inițial a fost realizarea unui procesor performant, dar suficient de simplu pentru a putea fi implementat cu un număr redus de tranzistoare. Simplitatea căii de date și a unității de control a permis obținerea unui raport favorabil între performanță, suprafața circuitului și consum energetic.

Denumirea ARM a fost asociată inițial expresiei *Acorn RISC Machine*, iar ulterior expresiei *Advanced RISC Machines*. În prezent, denumirea comercială utilizată este Arm, însă forma scrisă cu majuscule este încă întâlnită frecvent în literatura tehnică, în special atunci când se face referire la generațiile istorice ale arhitecturii.

Un aspect important al modelului de afaceri Arm este faptul că arhitecturile și nucleele de procesare sunt licențiate producătorilor de circuite integrate. Companii diferite pot integra același nucleu Arm în microcontrolere sau sisteme pe cip care includ memorii, periferice și acceleratoare proprii. Din acest motiv, două dispozitive care utilizează același nucleu Cortex-M pot avea caracteristici semnificativ diferite la nivelul perifericelor și al organizării memoriei.

3.5.1 Profilele arhitecturale Arm

Pentru a răspunde cerințelor foarte diferite ale aplicațiilor, familia Arm este organizată în mai multe profile arhitecturale. Cele trei categorii principale sunt profilul A, profilul R și profilul M. Aceste profile nu reprezintă doar niveluri diferite de performanță, ci modele arhitecturale optimizate pentru clase distincte de sisteme [10].

Figura 3.4 sintetizează principalele caracteristici și domenii de utilizare ale celor trei profile.

Profilul A, denumit *Application Profile*, este destinat sistemelor capabile să execute sisteme de operare complexe, precum Linux, Android sau Windows. Aceste procesoare includ în mod obișnuit unități de management al memoriei, memorii cache avansate și mecanisme de execuție speculativă. Nucleele Cortex-A sunt utilizate în telefoane mobile, calculatoare, sisteme multimedia, echipamente de rețea și platforme de calcul de înaltă performanță.

Profilul R, denumit *Real-time Profile*, este proiectat pentru aplicații în care performanța ridicată trebuie combinată cu un comportament temporal predictibil. Nucleele Cortex-R sunt întâlnite în controlere automotive, sisteme de stocare, echipamente industriale și alte aplicații critice. Acestea pot utiliza memorii de tip TCM (*Tightly Coupled Memory*), care oferă un timp de acces predictibil și evită variațiile introduse de cache-uri.

Profilul M, denumit *Microcontroller Profile*, este optimizat pentru microcontrolere, con-

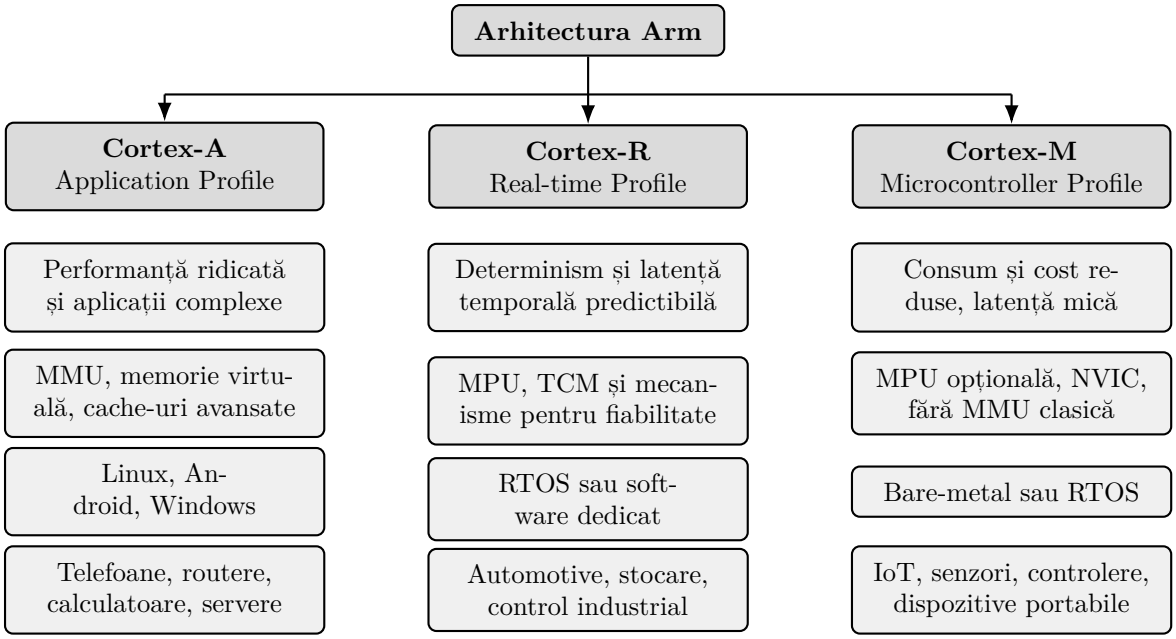


Figura 3.4 Profilele arhitecturale Arm și domeniile lor reprezentative de utilizare.

sum energetic redus și tratarea eficientă a întreruperilor. Nucleele Cortex-M sunt utilizate în senzori, dispozitive IoT, aparatură medicală, controlere industriale și sisteme automotive de complexitate redusă sau medie. Aceste procesoare utilizează un model de programare și un mecanism de tratare a excepțiilor diferit de cel întâlnit în profilele clasice A și R [11, 95].

Tabela 3.2 Caracteristicile generale ale profilelor Arm

Caracteristică	Cortex-A	Cortex-R	Cortex-M
Domeniu principal	Aplicații și sisteme de operare complexe	Control în timp real și aplicații critice	Microcontrolere și sisteme embedded
Managementul memoriei	MMU și memorie virtuală	MPU și memorii deterministe	MPU opțională, fără memorie virtuală clasică
Prioritate de proiectare	Performanță și flexibilitate	Determinism și fiabilitate	Consum redus, cost și latență mică
Sistem de operare	Linux, Android, Windows și alte sisteme complexe	RTOS sau software dedicat	Bare-metal sau RTOS
Exemple de aplicații	Smartphone, router, computer, server	Automotive, stocare, control industrial	Senzori, IoT, controlere și dispozitive portabile

În cadrul acestui capitol vor fi prezentate atât elemente ale modelului clasic AArch32, utile pentru înțelegerea evoluției arhitecturii Arm, cât și caracteristici specifice procesoarelor Cortex-M. Cele două modele trebuie diferențiate, deoarece registrele de stare, modurile de operare și mecanismele de tratare a întreruperilor nu sunt identice.

3.6 Registrele arhitecturii Arm

Registrele reprezintă cele mai rapide elemente de stocare disponibile procesorului. Acestea păstrează operanzii utilizați de instrucțiunile aritmetice și logice, adresele necesare accesului la memorie și informațiile asociate controlului execuției. Utilizarea eficientă a registrelor reduce numărul transferurilor către memoria principală și contribuie direct la creșterea performanței.

Modelul clasic AArch32 prezintă programatorului șaisprezece registre pe 32 de biți, numerotate de la R0 la R15. Registrele R0–R12 pot fi utilizate pentru stocarea operanzilor și a rezultatelor intermediare, în timp ce ultimele trei registre au roluri speciale [9, 81].

R13 este utilizat în mod convențional ca *Stack Pointer* (SP), R14 este utilizat ca *Link Register* (LR), iar R15 reprezintă *Program Counter* (PC).

Figura 3.5 prezintă organizarea registrelor vizibile într-un context de execuție AArch32.

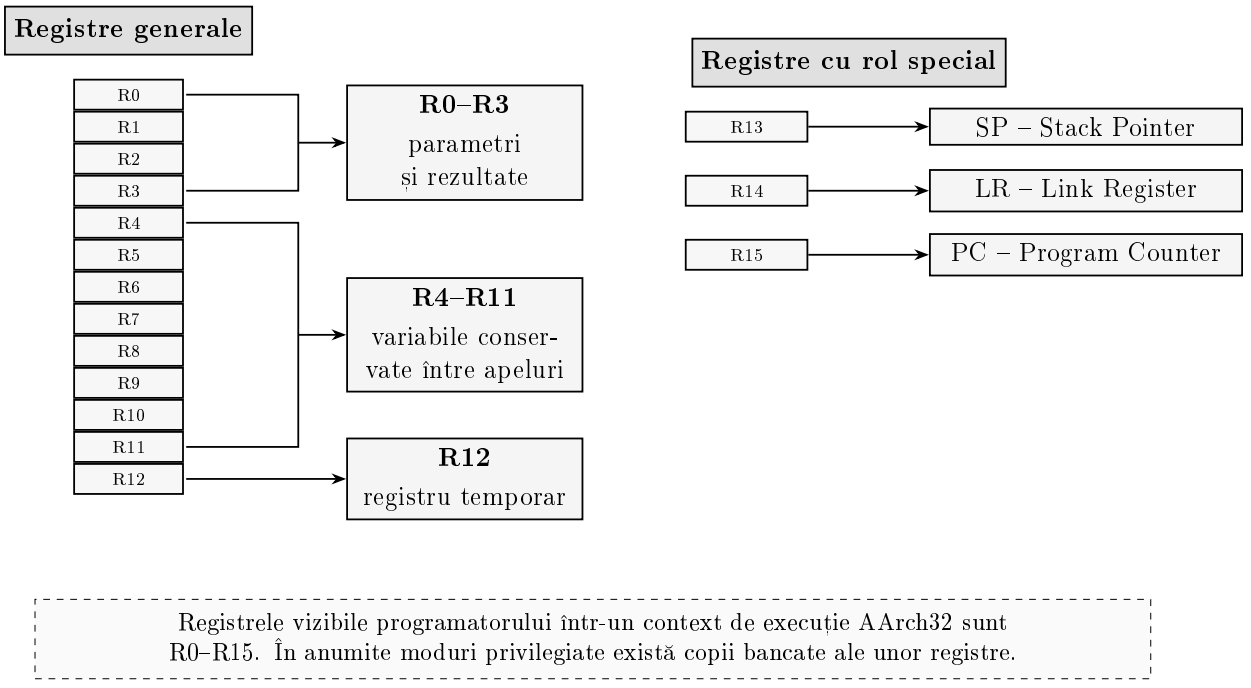


Figura 3.5 Organizarea registrelor generale în modelul AArch32.

Registrele R0–R3 sunt utilizate frecvent pentru transmiterea parametrilor către funcții și pentru returnarea rezultatelor. Registrele R4–R11 păstrează de regulă variabile locale care trebuie conservate între apelurile de funcții, în conformitate cu convenția de apel utilizată. Registrul R12 poate fi utilizat ca registru temporar de către compilator sau de către secvențele de legare.

Registrul SP indică vârful stivei curente. Stiva este utilizată pentru salvarea adreselor de întoarcere, a registrelor, a parametrilor și a variabilelor locale. Registrul LR păstrează adresa la care trebuie să revină execuția după finalizarea unei funcții. Instrucțiunea de apel cu legătură salvează automat adresa de întoarcere în LR.

Exemplu

La executarea unei instrucțiuni de tip `BL functie`, procesorul salvează adresa de întoarcere în registrul LR și transferă execuția către funcția apelată. La final, o instrucțiune precum `BX LR` poate restaura fluxul de execuție, folosind adresa păstrată în registrul de legătură.

3.6.1 Registrele procesoarelor Cortex-M

Procesoarele Cortex-M utilizează tot un set de registre R0–R15, dar modelul de programare include și registre speciale adaptate gestionării eficiente a excepțiilor. R0–R12 sunt registre generale, R13 reprezintă indicatorul de stivă, R14 este registrul de legătură, iar R15 este contorul de program.

O caracteristică importantă este existența a două indicatoare de stivă: MSP (*Main Stack Pointer*) și PSP (*Process Stack Pointer*).

MSP este utilizat după resetare și în modul Handler, în timp ce PSP poate fi utilizat de aplicații sau de taskurile unui sistem de operare de timp real. Această separare permite izolarea stivei nucleului sau a rutinelor de excepție de stivele utilizate de aplicații.

Pe lângă registrele generale, Cortex-M include registre precum xPSR, PRIMASK, BASEPRI, FAULTMASK și CONTROL. Disponibilitatea exactă a acestora depinde de varianta arhitecturală și de nucleul implementat.

Figura 3.6 prezintă o organizare simplificată a modelului de registre Cortex-M.

Diferențierea dintre modelul AArch32 clasic și modelul Cortex-M este importantă în dezvoltarea software-ului embedded. Chiar dacă denumirile registrelor generale sunt similare, modul de tratare a privilegiilor, întreruperilor și excepțiilor este diferit.

3.7 Program Counter și execuția în pipeline

Registrul Program Counter conține informația utilizată pentru identificarea instrucțiunii următoare. Într-un procesor fără pipeline, valoarea acestuia ar putea fi asociată direct cu instrucțiunea aflată în execuție. În procesoarele pipelined, mai multe instrucțiuni se află simultan în etape diferite, iar semnificația valorii citite din PC este influențată de organizarea arhitecturii.

Un pipeline simplu poate fi alcătuit din trei etape: preluarea instrucțiunii din memorie (*Fetch*), decodificarea instrucțiunii (*Decode*) și executarea operației (*Execute*).

În timp ce o instrucțiune este executată, următoarea poate fi decodificată, iar o a treia poate fi preluată din memorie. După umplerea pipeline-ului, procesorul poate finaliza în condiții ideale câte o instrucțiune la fiecare ciclu, chiar dacă fiecare instrucțiune traversează mai multe etape.

Figura 3.7 prezintă suprapunerea etapelor pentru o succesiune de instrucțiuni.

În arhitecturile Arm clasice, citirea registrului PC poate returna o valoare avansată față de adresa instrucțiunii aflate conceptual în execuție. Acest comportament reflectă poziția instrucțiunilor în pipeline și este definit de setul de instrucțiuni utilizat. Software-ul nu

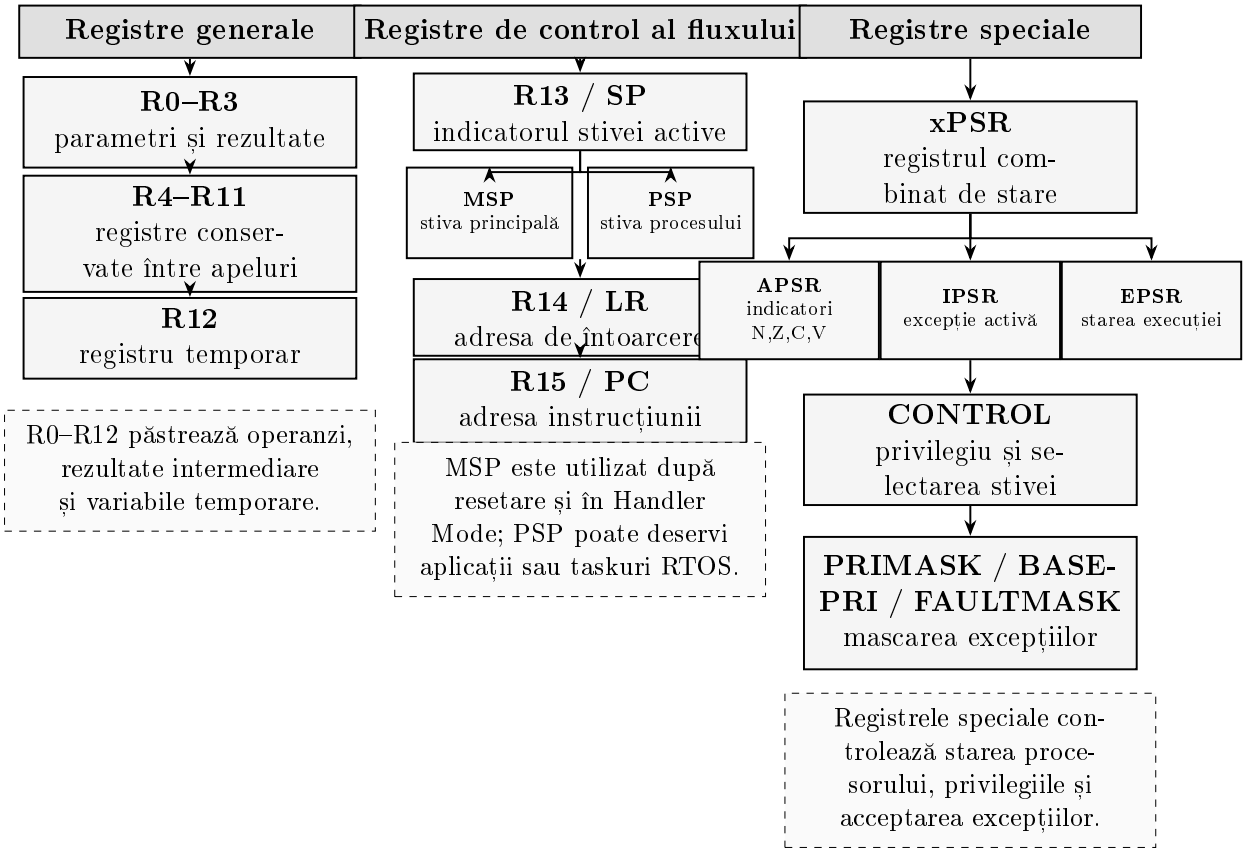


Figura 3.6 Modelul simplificat al registrelor unui procesor Cortex-M.

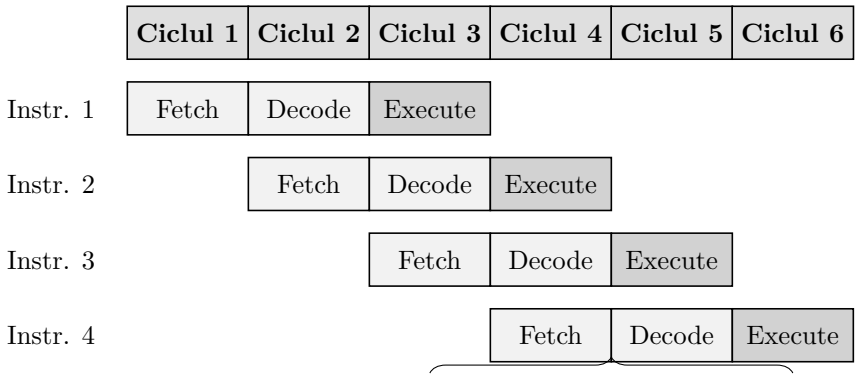


Figura 3.7 Suprapunerea etapelor Fetch, Decode și Execute într-un pipeline simplificat.

trebuie să presupună că valoarea vizibilă a PC reprezintă întotdeauna adresa exactă a instrucțiunii curente.

Instrucțiunile de salt modifică valoarea PC și întrerup fluxul secvențial. Într-o implementare simplă, instrucțiunile care au fost deja preluate pe traseul greșit trebuie eliminate, iar pipeline-ul trebuie reumplut de la noua adresă. Această operație introduce o penalizare de performanță.

Exemplu

Dacă procesorul execută un salt condiționat, instrucțiunile următoare pot fi deja prezente în etapele Fetch și Decode. Atunci când saltul este efectuat, aceste instrucțiuni nu mai trebuie executate și sunt eliminate din pipeline. Procesorul începe apoi preluarea instrucțiunilor de la adresa destinație a saltului.

Procesoarele moderne utilizează pipeline-uri mai complexe și mecanisme precum predicția salturilor și preluarea anticipată. În microcontrolere, pipeline-ul este de regulă mai scurt, deoarece obiectivele principale sunt consumul redus, latența predictibilă și simplitatea implementării.

3.8 Registrele de stare CPSR și xPSR

Pe lângă registrele generale, procesoarele Arm utilizează registre de stare care păstrează indicatorii rezultați din operațiile aritmetice, informații privind nivelul de privilegiu și starea mecanismelor de control. Structura acestor registre diferă între modelul AArch32 clasic și profilul Cortex-M.

3.8.1 Registrul CPSR în modelul AArch32

CPSR (*Current Program Status Register*) este registrul de stare utilizat în modelul AArch32 clasic. Acesta include indicatorii de condiție, biți de control ai întreruperilor, informații privind starea setului de instrucțiuni și câmpul care identifică modul curent al procesorului [9].

Cei mai cunoscuți indicatori sunt:

- N — rezultatul are bitul de semn activ;
- Z — rezultatul operației este zero;
- C — operația a produs transport sau împrumut;
- V — operația cu semn a produs depășire aritmetică.

Figura 3.8 prezintă o structură simplificată a registrului CPSR. Figura evidențiază numai câmpurile importante pentru introducerea conceptului; structura exactă depinde de versiunea arhitecturii.

Instrucțiunea `CMP` efectuează conceptual o scădere fără a memora rezultatul, dar actualizează indicatorii de condiție. Aceștia pot fi utilizați ulterior pentru controlul execuției.

Exemplu 3.1 Actualizarea indicatorilor de condiție

```
1 CMP R0, #0
```

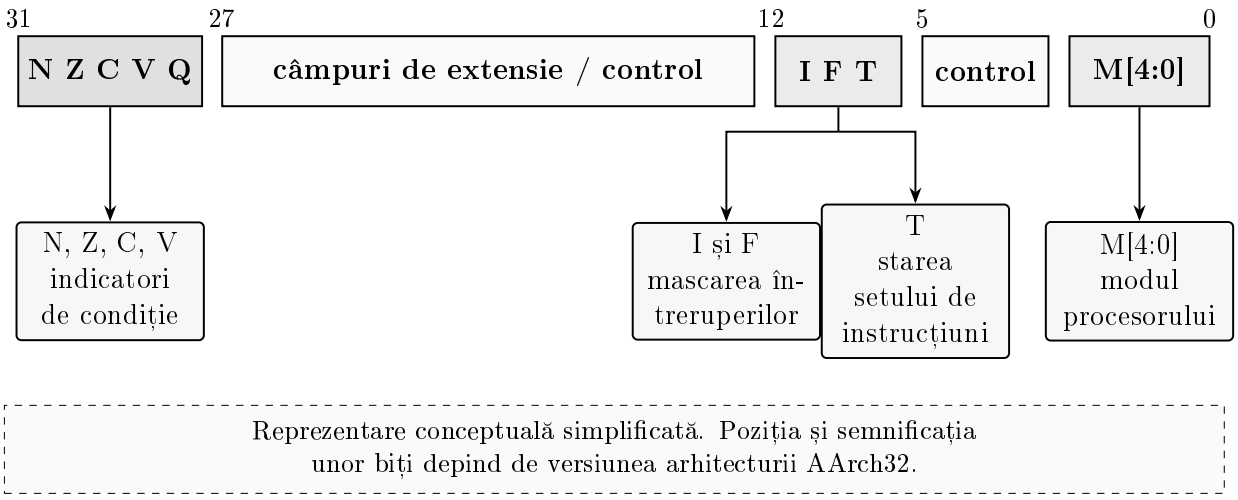


Figura 3.8 Structura conceptuală simplificată a registrului CPSR.

```
2 BEQ valoare_zero
```

Dacă R0 conține valoarea zero, indicatorul Z este setat, iar saltul condiționat poate fi executat.

3.8.2 Registrul xPSR în Cortex-M

Procesoarele Cortex-M utilizează registrul xPSR, care reunește conceptual informații din trei componente: APSR, indicatorii stării aplicației, IPSR, numărul excepției active, și EPSR, informații asociate stării de execuție.

Indicatorii N, Z, C și V sunt păstrați în componenta APSR și au roluri similare celor din CPSR. IPSR permite identificarea excepției sau întreruperii aflate în execuție. Cortex-M nu utilizează câmpul de mod din CPSR și nici modurile IRQ, FIQ sau Supervisor în forma întâlnită în modelul AArch32 clasic [11, 95].

Figura 3.9 prezintă organizarea conceptuală a registrului xPSR.

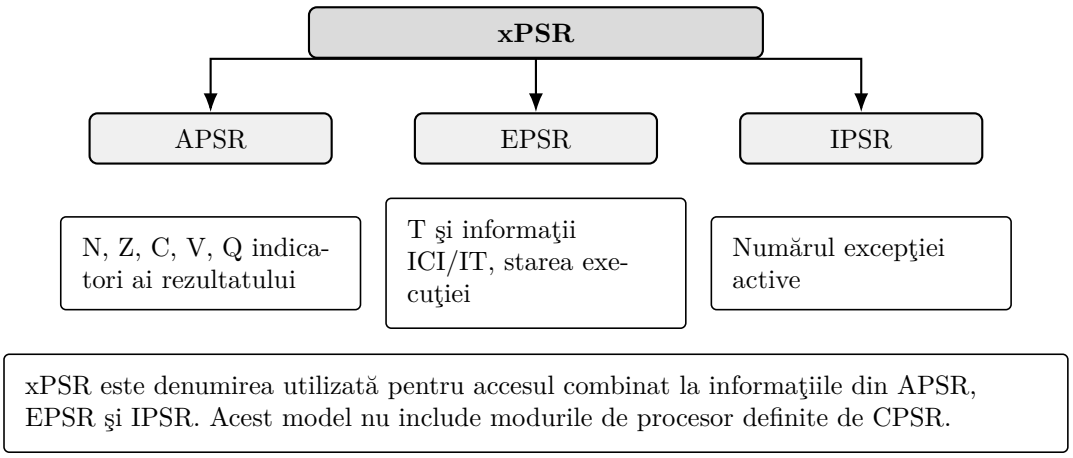


Figura 3.9 Componentele conceptuale ale registrului xPSR în procesoarele Cortex-M.

3.9 Modurile de operare și nivelurile de privilegiu

Modurile de operare permit procesorului să separe execuția normală a aplicațiilor de tratarea excepțiilor și de operațiile privilegiate. Modelul utilizat diferă semnificativ între profilele A/R clasice și profilul M.

3.9.1 Modurile clasice AArch32

În versiunile clasice ale arhitecturii AArch32 sunt definite mai multe moduri de procesor. Modul User este neprivilegiat, iar modurile System, Supervisor, IRQ, FIQ, Abort și Undefined sunt privilegiate. Trecerea către un mod de excepție este realizată automat atunci când procesorul acceptă evenimentul corespunzător.

Modurile IRQ și FIQ sunt asociate întreruperilor. FIQ este conceput pentru evenimente care necesită un răspuns rapid și dispune de un număr mai mare de registre bancate, reducând necesitatea salvării registrelor generale la intrarea în rutină. Modurile Abort și Undefined sunt utilizate pentru tratarea acceselor invalide și a instrucțiunilor nedefinite.

Unele registre sunt bancate, ceea ce înseamnă că un mod privilegiat dispune de propria copie. De exemplu, modurile de excepție au în mod obișnuit propriile registre SP și LR. Acest mecanism permite utilizarea unei stive dedicate și păstrarea adresei de întoarcere fără suprascrierea valorilor utilizate de aplicația întreruptă.

3.9.2 Modurile Cortex-M

Cortex-M utilizează un model mai simplu, bazat pe două moduri de execuție: *Thread Mode*, utilizat pentru execuția aplicației, și *Handler Mode*, utilizat pentru tratarea excepțiilor.

Handler Mode este întotdeauna privilegiat. Thread Mode poate fi privilegiat sau neprivilegiat, în funcție de configurația registrului CONTROL. Această organizare simplifică tratarea întreruperilor și este potrivită pentru microcontrolere și sisteme de operare de timp real.

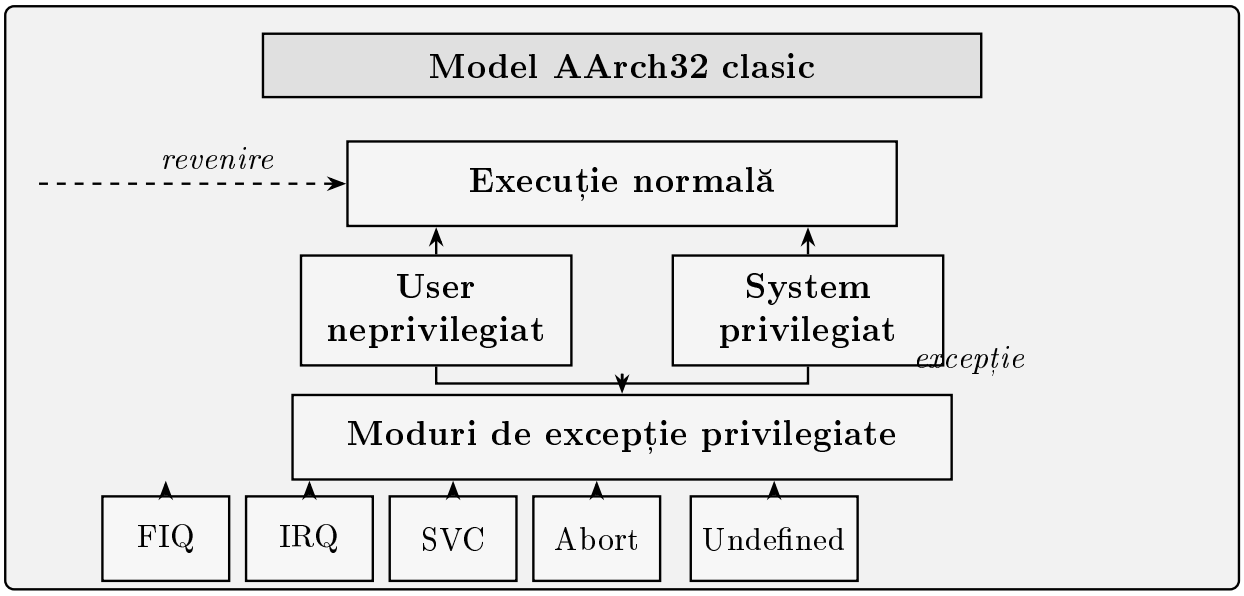
Figura 3.10 compară modelul clasic AArch32 cu modelul Cortex-M.

Această diferențiere este esențială atunci când se studiază documentația unui microcontroler. Un program destinat Cortex-M nu trebuie proiectat presupunând existența modurilor IRQ și FIQ sau a registrului CPSR, chiar dacă aceste concepte apar în materialele referitoare la procesoarele Arm clasice.

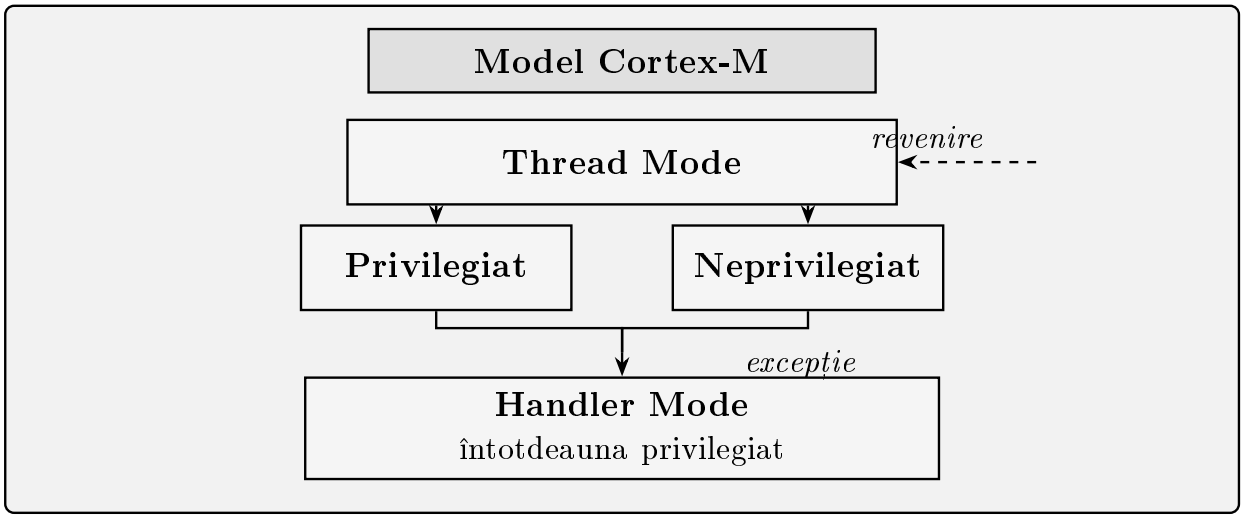
3.10 Gestionarea întreruperilor și a excepțiilor

Întreruperile permit procesorului să răspundă la evenimente fără a verifica permanent starea fiecărui periferic. În loc ca aplicația să interogheze continuu un timer, o interfață serială sau un convertor analog-digital, perifericul poate solicita atenția procesorului numai atunci când apare un eveniment relevant.

În forma sa generală, tratarea unei întreruperi presupune suspendarea temporară a programului curent, salvarea informațiilor necesare reluării sale, executarea unei rutine de tratare și restaurarea contextului. Detaliile exacte depind de arhitectură.



Modurile de excepție pot utiliza copii bancate pentru SP și LR.



Cortex-M utilizează MSP și PSP și nu definește modurile IRQ, FIQ sau Abort în forma modelului AArch32. Handler Mode este utilizat pentru tratarea tuturor excepțiilor.

Figura 3.10 Comparația dintre modurile clasice AArch32 și modelul de execuție Cortex-M.

Tabela 3.3 Comparația modelelor de privilegiu AArch32 clasic și Cortex-M

Caracteristică	AArch32 clasic	Cortex-M
Execuție normală	User sau System	Thread Mode
Tratarea excepțiilor	Moduri distincte: IRQ, FIQ, SVC, Abort și Undefined	Handler Mode
Privilegii	Determinate de modul curent	Privileged sau Unprivileged în Thread Mode; Handler este privilegiat
Registre bancate	Prezente pentru anumite moduri	Nu utilizează același model de registre bancate
Indicatoare de stivă	SP bancat în mai multe moduri	MSP și PSP
Registru principal de stare	CPSR	xPSR

Figura 3.11 prezintă fluxul conceptual al unei întreruperi.

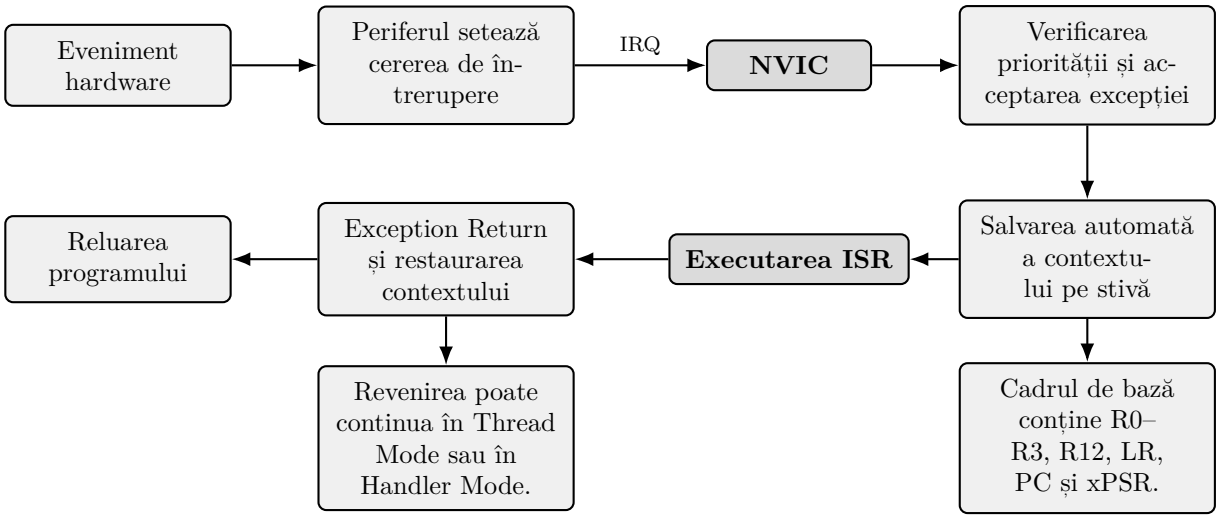


Figura 3.11 Fluxul conceptual de tratare a unei întreruperi.

3.10.1 Tratarea întreruperilor în Cortex-M

Procesoarele Cortex-M integrează controlerul NVIC (*Nested Vectored Interrupt Controller*). Acesta gestionează prioritățile, activarea și imbricarea întreruperilor și permite identificarea rapidă a rutinei care trebuie executată.

La acceptarea unei excepții, procesorul salvează automat pe stivă un set de registre care formează cadrul de excepție. În mod obișnuit, sunt salvate registrele R0–R3, R12, LR, PC și xPSR. Această operație hardware reduce codul necesar la intrarea în rutină și contribuie la obținerea unei latențe predictibile [11, 95].

După salvarea contextului, procesorul intră în Handler Mode și încarcă adresa rutinei

din tabela vectorilor de excepție. La finalizarea rutinei, mecanismul de revenire restaurează contextul și permite reluarea programului întrerupt.

Exemplu

Un timer generează o întrerupere la fiecare milisecundă. La apariția evenimentului, NVIC identifică rutina asociată, procesorul salvează automat contextul minim și execută ISR-ul timerului. Rutina poate incrementa un contor utilizat pentru măsurarea timpului, după care procesorul revine la instrucțiunea întreruptă.

Rutinele ISR trebuie să fie scurte și predictibile. Operațiile lente, așteptările active și apelurile blocante trebuie evitate. Într-un sistem cu RTOS, ISR-ul efectuează de regulă numai operația strict necesară și semnalizează un task care va continua prelucrarea în afara contextului de întrerupere.

Exemplu 3.2 Exemplu conceptual de rutină de întrerupere

```
1 volatile unsigned long counter = 0;
2
3 void Timer_ISR()
4 {
5     counter++;
6 }
```

O variabilă modificată atât de programul principal, cât și de o rutină ISR trebuie declarată și accesată cu atenție. Cuvântul cheie `volatile` informează compilatorul că valoarea se poate modifica în afara fluxului normal al programului, dar nu garantează atomicitatea operațiilor și nu înlocuiește mecanismele de sincronizare.

3.11 Setul de instrucțiuni Arm

Setul de instrucțiuni reprezintă interfața dintre software și procesor. El definește operațiile pe care procesorul le poate executa, registrele accesibile programatorului, modurile de adresare și efectele fiecărei instrucțiuni asupra registrelor de stare.

În familia Arm trebuie diferențiate mai multe stări și codificări ale setului de instrucțiuni. În modelul AArch32, procesoarele din profilele A și R pot utiliza instrucțiuni A32, codificate în mod tradițional pe 32 de biți, și instrucțiuni T32, cunoscute și sub denumirea Thumb sau Thumb-2. Procesoarele Cortex-M execută exclusiv instrucțiuni din setul T32 și nu dispun de starea A32 [9, 11].

Deși detaliile diferă între versiunile arhitecturii, instrucțiunile pot fi grupate în câteva categorii funcționale: instrucțiuni de transfer între memorie și registre, instrucțiuni aritmetice și logice, instrucțiuni de deplasare și rotație, instrucțiuni de comparație, instrucțiuni de salt și apel de funcție, precum și instrucțiuni pentru controlul procesorului și al sistemului.

Figura 3.12 prezintă principalele categorii de instrucțiuni și relația acestora cu registrele, memoria și unitatea aritmetică și logică.

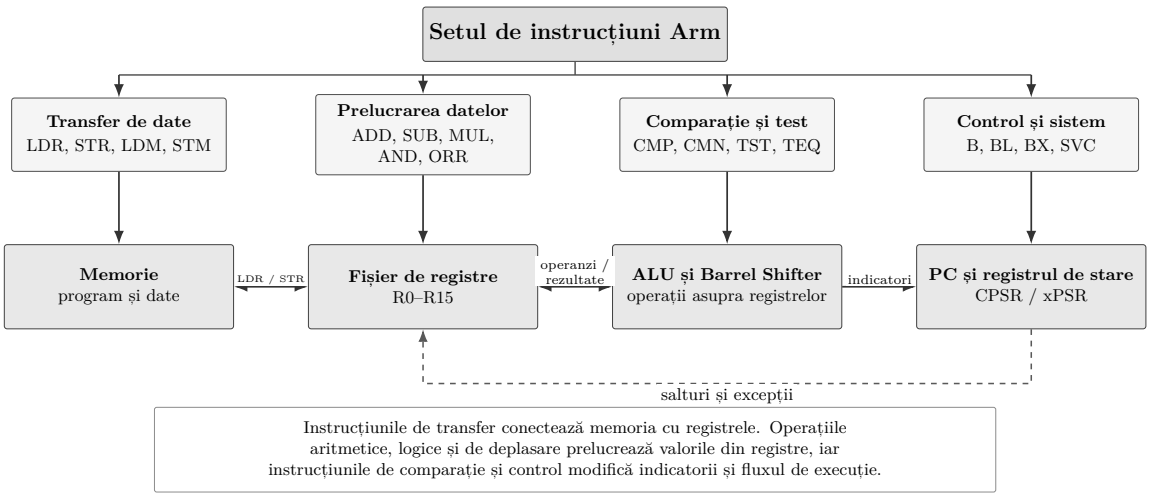


Figura 3.12 Principalele categorii de instrucțiuni ale arhitecturii Arm.

3.11.1 Modelul Load/Store

Arhitectura Arm utilizează un model de tip *Load/Store*. Instrucțiunile aritmetice și logice operează, în general, asupra valorilor aflate în registre. Accesarea memoriei este realizată prin instrucțiuni dedicate de încărcare și stocare.

Instrucțiunea `LDR` transferă o valoare din memorie într-un registru, iar instrucțiunea `STR` transferă conținutul unui registru către memorie.

Exemplu 3.3 Transferul unei valori între memorie și registre

```
1 LDR R0, [R1]
2 STR R0, [R2]
```

Prima instrucțiune citește valoarea de la adresa conținută în R1 și o încarcă în R0. A doua instrucțiune scrie valoarea din R0 la adresa conținută în R2.

Separarea accesului la memorie de operațiile aritmetice simplifică proiectarea căii de date. Un calcul care utilizează două valori memorate trebuie realizat printr-o succesiune de încărcări, operații asupra registrelor și, dacă este necesar, o stocare a rezultatului.

Exemplu 3.4 Calculul unei sume utilizând modelul Load/Store

```
1 LDR R0, [R4]      ; R0 = valoarea de la adresa R4
2 LDR R1, [R5]      ; R1 = valoarea de la adresa R5
3 ADD R2, R0, R1    ; R2 = R0 + R1
4 STR R2, [R6]      ; memoria[R6] = R2
```

Figura 3.13 ilustrează traseul datelor în cazul unei operații de încărcare, urmată de prelucrarea valorii și stocarea rezultatului.

Instrucțiunile de transfer pot lucra cu valori de dimensiuni diferite. Sufixele utilizate depind de dimensiunea și interpretarea informației transferate. De exemplu, `LDRB` și `STRB` operează asupra unui octet, iar `LDRH` și `STRH` asupra unei jumătăți de cuvânt.

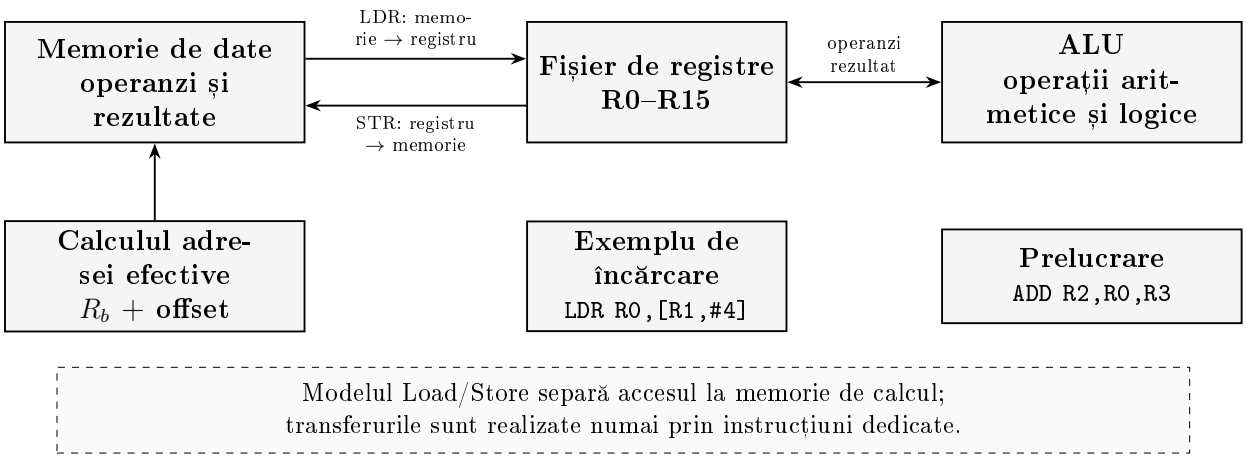


Figura 3.13 Prelucrarea datelor în modelul Load/Store.

Exemplu 3.5 Transferuri de dimensiuni diferite

```
1 LDRB R0, [R1]      ; incarcare octet fara semn
2 LDRH R2, [R3]      ; incarcare valoare pe 16 biti
3 STRB R4, [R5]      ; stocarea celor mai putin semnificativi 8 biti
4 STRH R6, [R7]      ; stocarea celor mai putin semnificativi 16
    ↪ biti
```

În cazul încărcării unor valori cu semn pot fi utilizate instrucțiuni precum `LDRSB` și `LDRSH`. Acestea extind semnul valorii citite pentru a produce rezultatul pe dimensiunea registrului destinație.

3.11.2 Moduri de adresare

Adresa efectivă utilizată de o instrucțiune Load/Store poate fi calculată pornind de la un registru de bază și un deplasament. Deplasamentul poate fi o constantă sau conținutul unui alt registru.

Exemplu 3.6 Adresare cu deplasament imediat

```
1 LDR R0, [R1, #12]
```

În acest caz, adresa efectivă este obținută prin adunarea valorii 12 la adresa de bază din R1:

$$A_{\text{efectiv}} = R1 + 12.$$

Adresarea indexată este utilă pentru accesarea elementelor unui tablou sau a câmpurilor unei structuri. Dacă fiecare element al unui tablou ocupă patru octeți, indicele poate fi deplasat la stânga cu două poziții înainte de calcularea adresei.

Exemplu 3.7 Accesarea unui tablou prin adresare indexată

```
1 LDR R0, [R1, R2, LSL #2]
```

Instrucțiunea citește elementul aflat la adresa

$$A_{\text{efectiv}} = R1 + 4 \times R2.$$

Registrul de bază poate fi actualizat automat. În cazul adresării cu pre-indexare, adresa este calculată înaintea transferului, iar simbolul `!` solicită scrierea noii adrese în registrul de bază.

Exemplu 3.8 Adresare cu pre-indexare și actualizarea registrului de bază

```
1 LDR R0, [R1, #4]!
```

Operația este echivalentă conceptual cu:

$$R1 \leftarrow R1 + 4, \qquad R0 \leftarrow Memorie[R1].$$

În cazul post-indexării, transferul utilizează mai întâi adresa inițială, după care registrul de bază este actualizat.

Exemplu 3.9 Adresare cu post-indexare

```
1 LDR R0, [R1], #4
```

Acest tip de adresare este potrivit pentru parcurgerea secvențială a unui vector.

3.11.3 Instrucțiuni aritmetice

Instrucțiunile aritmetice realizează operații asupra registrelor și, în anumite cazuri, asupra unei valori imediate codificate în instrucțiune.

Exemplu 3.10 Instrucțiuni aritmetice reprezentative

```
1 ADD R0, R1, R2
2 SUB R3, R4, R5
3 MUL R6, R7, R8
```

Instrucțiunea `ADD` adună cei doi operanzi sursă și scrie rezultatul în registrul destinație. Instrucțiunea `SUB` realizează scăderea, iar `MUL` efectuează înmulțirea.

Valoarea unui operand poate fi specificată direct printr-o constantă imediată.

Exemplu 3.11 Utilizarea unei valori imediate

```
1 ADD R0, R0, #1
2 SUB R1, R1, #10
```

În funcție de setul de instrucțiuni și de codificarea disponibilă, nu orice constantă poate fi inclusă direct într-o singură instrucțiune. Asamblorul poate utiliza o succesiune de instrucțiuni sau poate încărca valoarea dintr-o zonă de constante.

Instrucțiunile `ADC` și `SBC` includ indicatorul Carry în calcul și sunt utile pentru realizarea operațiilor asupra numerelor care ocupă mai multe registre.

Exemplu 3.12 Adunarea a două valori pe 64 de biți

```
1 ADDS R0, R0, R2
2 ADC  R1, R1, R3
```

Se consideră că prima valoare pe 64 de biți este păstrată în perechea R1:R0, iar a doua în R3:R2. Instrucțiunea `ADDS` adună cuvintele inferioare și actualizează indicatorul Carry. Instrucțiunea `ADC` adună cuvintele superioare și transportul produs de prima operație.

3.11.4 Instrucțiuni logice și de manipulare a biților

Instrucțiunile logice sunt frecvent utilizate în software-ul embedded pentru accesarea registrelor perifericelor, configurarea câmpurilor de biți și implementarea măștilor.

Exemplu 3.13 Instrucțiuni logice reprezentative

```
1 AND R0, R1, R2
2 ORR R3, R4, R5
3 EOR R6, R7, R8
4 BIC R9, R9, R10
```

Instrucțiunea `AND` păstrează numai biții activi în ambii operanzi. `ORR` setează biții indicați de mască, iar `EOR` realizează operația SAU exclusiv. Instrucțiunea `BIC`, denumită *Bit Clear*, șterge din primul operand biții care sunt setați în al doilea operand.

Pentru activarea bitului 3 al unui registru se poate utiliza:

Exemplu 3.14 Setarea unui bit

```
1 ORR R0, R0, #(1 << 3)
```

Pentru ștergerea aceluiași bit se poate utiliza:

Exemplu 3.15 Ștergerea unui bit

```
1 BIC R0, R0, #(1 << 3)
```

Instrucțiunea `TST` realizează o operație logică AND fără a păstra rezultatul, actualizând numai indicatorii de condiție. Aceasta poate fi utilizată pentru verificarea stării unui bit.

Exemplu 3.16 Testarea unui bit

```
1 TST R0, #(1 << 3)
2 BNE bit_activ
```

3.11.5 Instrucțiuni de comparație

Instrucțiunile de comparație actualizează indicatorii de stare fără a salva rezultatul operației. `CMP` efectuează conceptual o scădere, iar `CMN` realizează conceptual o adunare.

Exemplu 3.17 Compararea a două valori

```
1 CMP R0, R1
```

```
2 BEQ valori_egale
3 BGT prima_mai_mare
```

Instrucțiunile de salt condiționat interpretează indicatorii N, Z, C și V în funcție de tipul comparației. Este importantă diferențierea dintre comparațiile cu semn și cele fără semn. De exemplu, condițiile `GT` și `LT` sunt utilizate pentru numere cu semn, în timp ce `HI` și `LO` corespund comparațiilor fără semn.

Tabela 3.4 Condiții uzuale utilizate după o comparație

Condiție	**Semnificație**	**Interpretare**
EQ	Egal	Indicatorul Z este setat
NE	Diferit	Indicatorul Z nu este setat
CS/HS	Carry set / mai mare sau egal	Comparație fără semn
CC/LO	Carry clear / mai mic	Comparație fără semn
GT	Mai mare	Comparație cu semn
GE	Mai mare sau egal	Comparație cu semn
LT	Mai mic	Comparație cu semn
LE	Mai mic sau egal	Comparație cu semn

3.11.6 Instrucțiuni de salt și apel de funcție

Instrucțiunea `B` modifică fluxul de execuție prin încărcarea unei adrese noi în contorul de program. Destinația este exprimată de regulă printr-o etichetă simbolică, iar asamblorul calculează deplasamentul corespunzător.

Exemplu 3.18 Implementarea unei bucle

```
1 MOV R0, #10
2
3 bucla:
4     SUBS R0, R0, #1
5     BNE bucla
```

Instrucțiunea `BL`, denumită *Branch with Link*, salvează informația necesară revenirii în registrul LR și transferă execuția către funcția apelată.

Exemplu 3.19 Apelul și revenirea dintr-o funcție

```
1 BL functie
2
3 ; continuarea programului
4
5 functie:
```

```
6       ADD R0, R0, #1
7       BX LR
```

Într-un program real, funcția poate salva pe stivă registrele pe care trebuie să le conserve, inclusiv registrul LR atunci când funcția apelează la rândul său alte funcții.

Exemplu 3.20 Salvarea registrelor în cadrul unei funcții

```
1 functie:
2       PUSH {R4, LR}
3
4       MOV  R4, R0
5       BL   alta_functie
6       ADD  R0, R0, R4
7
8       POP  {R4, PC}
```

Încărcarea valorii salvate direct în PC realizează revenirea la apelant.

3.12 Execuția condiționată

Execuția condiționată permite procesorului să execute o instrucțiune numai dacă indicatorii din registrul de stare satisfac o anumită condiție. Obiectivul este reducerea numărului de salturi necesare pentru implementarea deciziilor scurte.

În codificarea A32, numeroase instrucțiuni includ un câmp de condiție. Sufixul condiției este atașat mnemonicii instrucțiunii.

Exemplu 3.21 Execuție condiționată în setul A32

```
1 CMP   R0, #0
2 MOVEQ R1, #1
3 MOVNE R1, #0
```

Dacă R0 este egal cu zero, este executată instrucțiunea `MOVEQ`. În caz contrar, este executată `MOVNE`. Astfel, o secvență echivalentă unei instrucțiuni `if-else` poate fi realizată fără un salt explicit.

Figura 3.14 ilustrează relația dintre comparație, actualizarea indicatorilor și selectarea instrucțiunii executate.

Predicarea instrucțiunilor poate reduce penalizările produse de salturile scurte. Ea nu este însă întotdeauna mai eficientă decât utilizarea unui salt. O instrucțiune a cărei condiție este falsă ocupă în continuare spațiu în cod și consumă resurse pentru preluare și decodificare.

În setul T32, execuția condiționată este mai restrictivă. Salturile condiționate sunt disponibile direct, iar anumite arhitecturi permit utilizarea instrucțiunii `IT`, de la *If-Then*, pentru condiționarea unui bloc scurt de instrucțiuni.

Exemplu 3.22 Bloc condiționat IT în T32

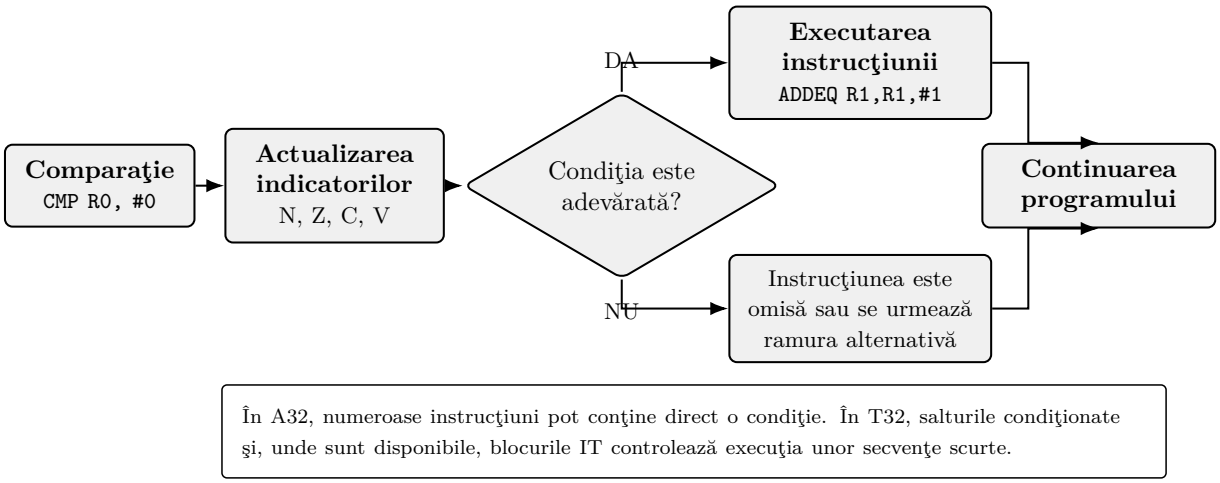


Figura 3.14 Principiul execuției condiționate pe baza indicatorilor de stare.

```
1 CMP   R0, #0
2 ITE   EQ
3 MOVEQ R1, #1
4 MOVNE R1, #0
```

Disponibilitatea și restricțiile blocurilor IT depind de versiunea arhitecturii și de procesor. În codul modern, compilatoarele pot prefera salturi condiționate atunci când acestea oferă un comportament mai clar sau mai eficient.

3.13 Barrel Shifter

Barrel Shifter este un bloc hardware care permite deplasarea sau rotirea unui operand înainte ca acesta să fie transmis unității aritmetice și logice. În numeroase instrucțiuni A32, operația de deplasare poate fi combinată cu o operație aritmetică sau logică, fără introducerea unei instrucțiuni separate [9, 81].

Figura 3.15 prezintă integrarea Barrel Shifter-ului în calea de date.

Principalele operații de deplasare sunt `LSL` (deplasare logică la stânga), `LSR` (deplasare logică la dreapta), `ASR` (deplasare aritmetică la dreapta), `ROR` (rotație la dreapta) și `RRX` (rotație la dreapta prin indicatorul Carry).

Operația `LSL` introduce zerouri în pozițiile eliberate și este echivalentă, în absența depășirii, cu înmulțirea unei valori fără semn cu o putere a lui doi.

$$x \,\mathrm{LSL}\, n = x \times 2^n.$$

De exemplu:

Exemplu 3.23 Combinarea deplasării cu adunarea

```
1 ADD R0, R1, R2, LSL #2
```

Instrucțiunea calculează:

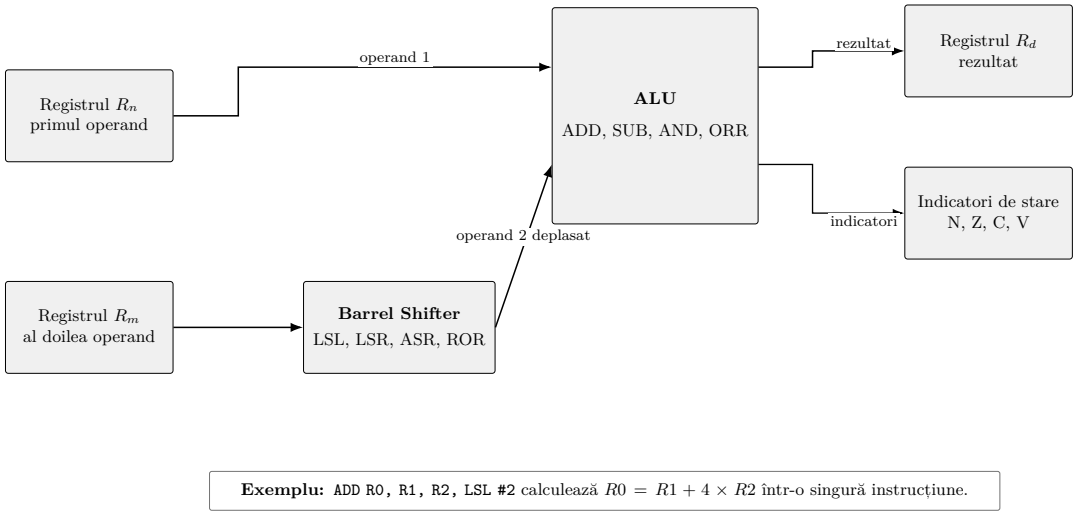


Figura 3.15 Integrarea Barrel Shifter-ului în calea de date a procesorului.

$$R0 = R1 + 4 \times R2.$$

Această operație este utilă pentru calcularea adresei unui element dintr-un tablou ale cărui elemente ocupă patru octeți.

Operația `LSR` introduce zerouri în pozițiile superioare și este potrivită pentru valori fără semn. Operația `ASR` păstrează bitul de semn și este utilizată pentru împărțirea aproximativă la puteri ale lui doi a numerelor cu semn.

Exemplu 3.24 Deplasare logică și deplasare aritmetică

```
1 LSR R0, R1, #3
2 ASR R2, R3, #3
```

Cele două instrucțiuni produc același rezultat pentru valori pozitive, dar se comportă diferit în cazul valorilor negative.

Operația de rotație reintroduce biții eliminați la un capăt în pozițiile eliberate la celălalt capăt. Aceasta este utilizată în algoritmi de criptografie, calculul sumelor de control și manipularea câmpurilor de biți.

Exemplu 3.25 Rotația unui operand

```
1 ROR R0, R1, #8
```

Deși denumirea Barrel Shifter este asociată frecvent cu procesoarele Arm clasice, seturile moderne de instrucțiuni pot implementa operații echivalente prin codificări diferite. Disponibilitatea combinării unei deplasări cu o altă operație depinde de instrucțiunea și de setul A32 sau T32 utilizat.

3.14 Seturile de instrucțiuni Thumb și Thumb-2

Instrucțiunile A32 oferă o codificare regulată și un număr mare de operații, dar utilizarea unei lungimi de 32 de biți pentru fiecare instrucțiune poate conduce la programe de dimensiuni relativ mari. Pentru sistemele embedded, dimensiunea codului influențează costul memoriei, consumul energetic și timpul necesar transferului instrucțiunilor.

Setul Thumb a fost introdus pentru îmbunătățirea densității codului. Primele versiuni utilizau în principal instrucțiuni codificate pe 16 biți, care reprezentau un subset compact al funcționalității disponibile în setul Arm clasic.

Reducerea dimensiunii instrucțiunilor presupunea și anumite limitări: accesul la un număr mai redus de registre în anumite codificări, mai puține moduri de adresare și intervale mai mici pentru constante sau salturi.

Thumb-2 a extins conceptul prin combinarea instrucțiunilor pe 16 și 32 de biți în cadrul aceluiași set T32. Instrucțiunile frecvente pot utiliza codificări compacte, în timp ce operațiile care necesită mai mulți operanzi, constante mai mari sau funcționalități avansate pot utiliza codificări pe 32 de biți.

Figura 3.16 prezintă conceptual diferența dintre codificările A32, Thumb inițial și Thumb-2.

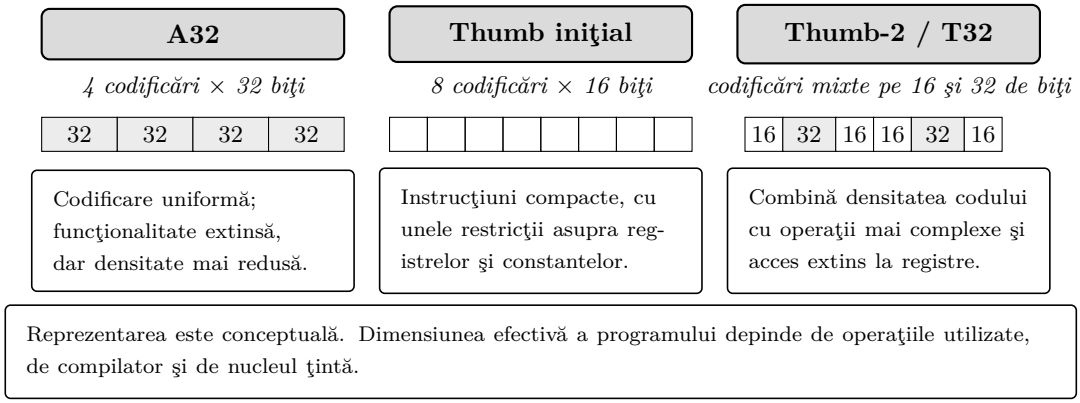


Figura 3.16 Comparația conceptuală a codificărilor A32, Thumb și Thumb-2.

Procesoarele Cortex-M execută instrucțiuni T32. Totuși, nu toate nucleele Cortex-M implementează exact același subset. Cortex-M0 și Cortex-M0+ utilizează un set mai restrâns, optimizat pentru suprafață și consum reduse. Cortex-M3, Cortex-M4 și alte nuclee mai performante includ un număr mai mare de instrucțiuni Thumb-2 și extensii opționale.

Din acest motiv, faptul că două microcontrolere aparțin familiei Cortex-M nu garantează compatibilitatea completă la nivelul tuturor instrucțiunilor. Compilatorul trebuie configurat pentru arhitectura și nucleul țintă.

Exemplu

O aplicație compilată pentru un Cortex-M4 poate conține instrucțiuni de procesare digitală a semnalelor care nu sunt disponibile pe Cortex-M0. Codul sursă în limbaj C poate

Tabela 3.5 Comparația generală a seturilor A32 și T32

Caracteristică	A32	Thumb inițial	Thumb-2/T32
Lungimea instrucțiunii	32 de biți	În principal 16 biți	Codificări pe 16 și 32 de biți
Densitatea codului	Mai redusă	Ridicată	Ridicată, cu funcționalitate extinsă
Accesul la registre	Codificări uniforme pentru registre	Restricții pentru unele instrucțiuni	Acces extins prin instrucțiuni pe 32 de biți
Execuție condiționată	Predicare pentru numeroase instrucțiuni	Predominant salturi condiționate	Salturi și blocuri IT, în funcție de arhitectură
Utilizare reprezentativă	Procesoare AArch32 clasice	Sisteme embedded istorice	Cortex-M și procesoare AArch32 compatibile

fi identic, dar fișierul executabil trebuie generat separat pentru fiecare țintă.

Densitatea codului nu trebuie evaluată numai prin lungimea individuală a unei instrucțiuni. O operație A32 pe 32 de biți poate înlocui, în anumite situații, două instrucțiuni Thumb pe 16 biți. Avantajul real depinde de aplicație, compilator și distribuția instrucțiunilor utilizate.

3.15 Arhitectura de interconectare AMBA

Un sistem pe cip modern conține mai multe componente care trebuie să comunice: unul sau mai multe procesoare, memorii, controlere DMA, acceleratoare, interfețe de comunicație și periferice. Conectarea directă a fiecărei componente la toate celelalte ar conduce la o structură dificil de proiectat și verificat.

AMBA, de la *Advanced Microcontroller Bus Architecture*, reprezintă o familie de protocoale de interconectare definite de Arm. Aceste protocoale standardizează schimbul de adrese, date și semnale de control între blocurile unui sistem pe cip [6–8].

AMBA nu desemnează o singură magistrală și nici o succesiune obligatorie de conexiuni. Proiectantul selectează protocoalele potrivite pentru performanța, complexitatea și consumul energetic ale fiecărei zone a sistemului.

Figura 3.17 prezintă o organizare posibilă în care componentele cu cerințe ridicate sunt conectate prin AXI sau AHB, iar perifericele simple sunt accesate prin APB.

3.15.1 Protocolul AXI

AXI, de la *Advanced eXtensible Interface*, este destinat interconectărilor care necesită debit ridicat și flexibilitate. Protocolul utilizează canale separate pentru adresele și datele operațiilor de citire și scriere.

Separarea canalelor permite suprapunerea mai multor transferuri. Un dispozitiv poate transmite adresa unei operații noi înainte ca operația precedentă să fie finalizată, dacă im-

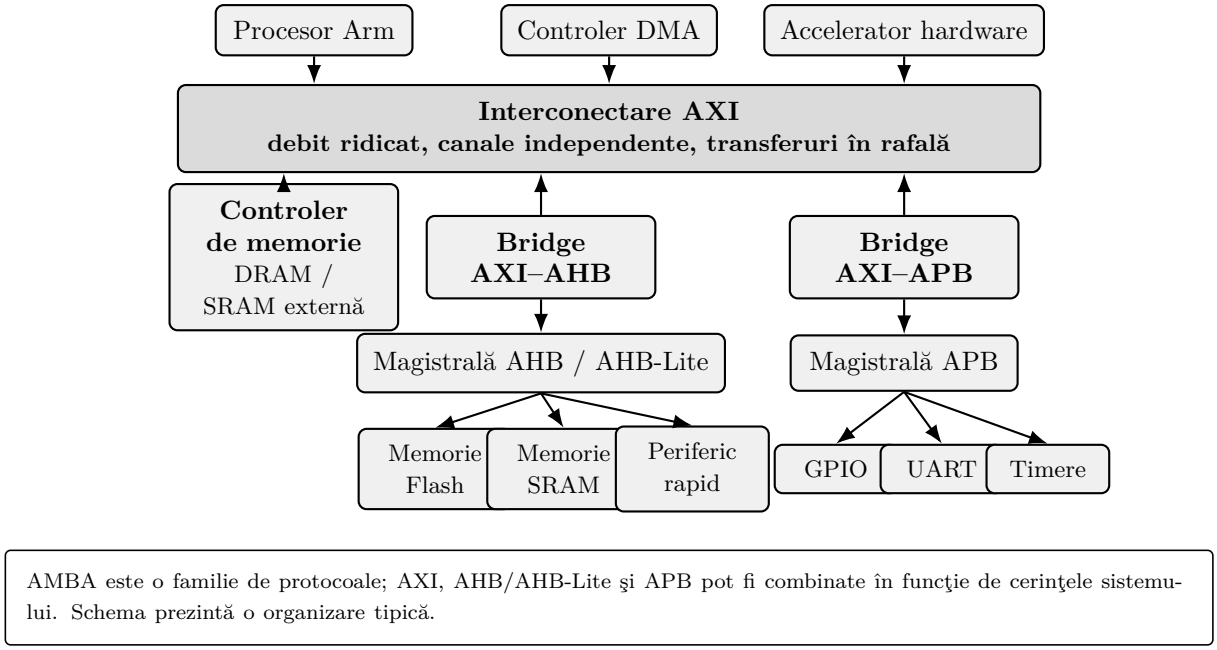


Figura 3.17 Exemplu de organizare a unui sistem pe cip utilizând protocoale AMBA.

plementarea și configurația sistemului permit acest lucru.

Transferurile pot fi organizate în rafale, denumite *bursts*. În locul transmiterii separate a adresei pentru fiecare cuvânt, o singură informație de adresă poate descrie o secvență de transferuri consecutive.

AXI este potrivit pentru conectarea procesoarelor performante, controlerelor de memorie, unităților grafice, acceleratoarelor și controlerelor DMA. Funcționalitățile sale implică însă o logică de control mai complexă decât în cazul protocoalelor destinate perifericelor simple.

3.15.2 Protocoalele AHB și AHB-Lite

AHB, de la *Advanced High-performance Bus*, este un protocol destinat transferurilor cu performanță ridicată, dar cu o complexitate mai redusă decât AXI. Transferul adresei poate fi suprapus cu transferul datelor corespunzător operației precedente.

AHB-Lite reprezintă o variantă simplificată pentru sistemele cu un singur inițiator principal al transferurilor. Este utilizată frecvent în microcontrolere și în magistralele interne ale unor procesoare Cortex-M.

Componentele conectate la AHB sau AHB-Lite pot include memoria Flash, memoria SRAM, controlerele DMA și perifericele care necesită un debit mai ridicat.

3.15.3 Protocolul APB

APB, de la *Advanced Peripheral Bus*, este destinat perifericelor cu cerințe reduse de performanță și cu registre de control relativ simple. Exemplele includ timere, interfețe UART, controlere GPIO și module de supraveghere.

Protocolul utilizează un transfer simplu, desfășurat în faze de configurare și acces. APB

nu urmărește maximizarea debitului, ci reducerea complexității interfeței și a consumului energetic.

Un pod, denumit *bridge*, transformă transferurile provenite de la o interconectare AXI sau AHB în operații APB. Procesorul poate astfel accesa perifericele într-un spațiu de adrese comun, fără a trebui să cunoască în mod direct detaliile protocolului intern.

Tabela 3.6 Caracteristicile generale ale protocoalelor AMBA

Caracteristică	AXI	AHB/AHB-Lite	APB
Domeniu	Interconectare de performanță ridicată	Magistrală de sistem și memorii	Periferice simple
Complexitate	Ridicată	Medie	Redusă
Canale	Canale independente pentru citire și scriere	Adrese și date transferate într-o structură pipelined	Interfață simplă, fără pipeline de performanță
Transferuri în rafală	Suport extins	Suportate	Nu reprezintă obiectivul protocolului
Utilizări	Procesoare, memorii, acceleratoare și DMA	Flash, SRAM, DMA și periferice rapide	GPIO, UART, timere și registre de configurare

3.15.4 Accesarea perifericelor prin mapare în memorie

În numeroase sisteme Arm, perifericele sunt accesate prin mecanismul *memory-mapped I/O*. Fiecărui registru hardware îi este atribuită o adresă în spațiul de memorie al procesorului. Instrucțiunile obișnuite de încărcare și stocare pot fi utilizate pentru citirea și modificarea acestuia.

În limbaj C, un registru periferic poate fi reprezentat conceptual astfel:

Exemplu 3.26 Accesarea conceptuală a unui registru hardware

```
1 #define GPIO_OUTPUT (*(volatile unsigned int*)0x40020014u)
2
3 void set_output(void)
4 {
5     GPIO_OUTPUT |= (1u << 5);
6 }
```

Adresa din exemplu este ilustrativă. Adresele reale și semnificația biților sunt definite de manualul microcontrolerului.

Calificatorul `volatile` indică faptul că accesul nu trebuie eliminat sau combinat necorespunzător de compilator, deoarece valoarea registrului poate fi modificată de hardware. Acesta nu asigură însă sincronizarea dintre nuclee și nu înlocuiește barierele de memorie atunci când acestea sunt necesare.

Întrebări de verificare

1. Care este diferența dintre seturile de instrucțiuni A32 și T32?
2. De ce procesoarele Cortex-M nu trebuie descrise ca executând alternativ instrucțiuni ARM și Thumb?
3. Explicați principiul arhitecturii Load/Store.
4. Ce operație realizează instrucțiunea `LDR R0, [R1, #8]`?
5. Care este diferența dintre pre-indexare și post-indexare?
6. Cum poate fi accesat elementul unui tablou de valori pe 32 de biți utilizând un registru de bază și un indice?
7. Care este rolul instrucțiunilor `ADC` și `SBC`?
8. Care este diferența dintre instrucțiunile `CMP` și `TST`?
9. De ce comparațiile cu semn utilizează alte condiții decât comparațiile fără semn?
10. Care este rolul registrului LR în apelul unei funcții?
11. Ce avantaj poate oferi execuția condiționată?
12. De ce predicarea unei instrucțiuni nu este întotdeauna mai eficientă decât utilizarea unui salt?
13. Ce operație realizează instrucțiunea `ADD R0, R1, R2, LSL #2`?
14. Care este diferența dintre deplasarea logică și deplasarea aritmetică la dreapta?
15. De ce a fost introdus setul de instrucțiuni Thumb?
16. Cum combină Thumb-2 instrucțiunile pe 16 și 32 de biți?
17. De ce codul compilat pentru Cortex-M4 poate să nu fie executabil pe Cortex-M0?
18. Ce reprezintă AMBA?
19. Care sunt diferențele principale dintre AXI, AHB și APB?
20. De ce periferice precum GPIO și UART sunt conectate frecvent la APB?
21. Ce rol are un bridge AXI–APB sau AHB–APB?
22. Ce reprezintă mecanismul *memory-mapped I/O*?
23. De ce registrele hardware sunt declarate frecvent `volatile`?

4. Consumul de Energie în Sistemele Embedded

Constrângeri energetice și obiective de proiectare
Putere, energie și sarcină electrică
Profilul energetic al unui sistem embedded
Metrici energetice
Estimarea autonomiei
Principalii consumatori de energie dintr-un sistem embedded
Consumul energetic al procesoarelor CMOS
Puterea dinamică și puterea statică
Tehnici hardware pentru reducerea consumului
Modurile de consum redus
Tehnici software pentru optimizarea energetică
Costul energetic al comunicațiilor wireless
Procesare locală versus transmiterea datelor
Surse de energie pentru sistemele embedded
Recuperarea energiei din mediul înconjurător
Baterii electrochimice
Tehnologii moderne de baterii
Parametrii și comportamentul real al bateriilor
Modelarea electrică a bateriilor
Influența temperaturii, descărcării și îmbătrânirii
Studii de caz
Metodologie de proiectare energetică
Întrebări și probleme propuse

4.1 Constrângeri energetice și obiective de proiectare

Sistemele embedded trebuie adesea să respecte simultan cerințe stricte privind costul, dimensiunea, performanța, timpul de răspuns, fiabilitatea și consumul energetic - iar în numeroase aplicații, energia disponibilă este resursa care limitează direct funcționarea sistemului. Situația e tipică pentru dispozitivele alimentate din baterii, nodurile IoT din zone greu accesibile, dispozitivele medicale portabile sau implantabile și sistemele autonome care trebuie să funcționeze ani de zile fără intervenție umană [19, 54]. Din acest motiv, consumul energetic trebuie tratat ca o cerință de proiectare încă din primele faze: alegerea procesorului,

a senzorilor, a tehnologiei de comunicație și a sursei de alimentare influențează deopotrivă arhitectura hardware și structura software-ului.

Figura 4.1 prezintă principalele obiective care trebuie echilibrate în proiectare - optimizarea simultană a tuturor este rareori posibilă. De exemplu, creșterea frecvenței procesorului reduce timpul de execuție al unui algoritm, dar crește puterea și temperatura circuitului; efectul net asupra energiei totale depinde de durata execuției, de tensiune și de cât de repede poate reveni sistemul într-un mod de consum redus. De aceea proiectarea energetică nu înseamnă doar alegerea componentei cu cel mai mic consum instantaneu, ci analiza întregului profil de funcționare.

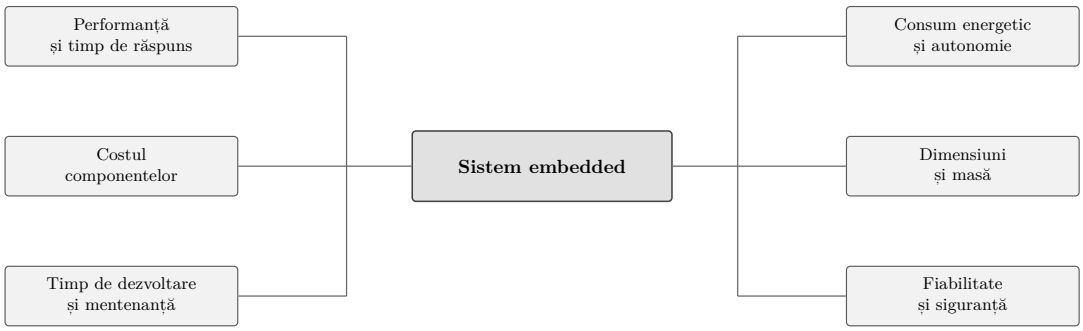


Figura 4.1 Principalele obiective și constrângeri întâlnite în proiectarea unui sistem embedded.

4.1.1 Factori de constrângere energetică

Patru factori leagă, în practică, decizia de proiectare energetică de constrângeri concrete. Primul este **autonomia**: pentru un dispozitiv pe baterie, ea este indicatorul de performanță perceput direct de utilizator și determină costul operațional - un senzor montat pe o conductă izolată, dacă are nevoie de baterie nouă lunar, poate costa în mentenanță mai mult decât valoarea lui, în timp ce o autonomie de cinci-zece ani face aceeași soluție viabilă economic. Ea poate fi estimată, ideal, prin raportarea energiei disponibile la puterea medie consumată:

$$T_{\text{ideal}} = \frac{E_{\text{disponibila}}}{P_{\text{medie}}}. \tag{4.1}$$

Relația e simplă, dar arată un principiu esențial: autonomia depinde de consumul *mediu*, nu doar de cel din modul activ - un sistem poate avea vârfuri mari de curent la transmisie și totuși autonomie bună, dacă restul timpului stă în repaus.

Al doilea factor sunt **limitările termice**. Puterea absorbită de circuitele digitale se transformă în mare parte în căldură, iar temperatura ridicată reduce durata de viață a componentelor și poate forța procesorul să-și reducă frecvența [38, 69]. Sistemele embedded compacte nu au, de regulă, ventilatoare, așa că temperatura trebuie controlată prin consum redus și disipare pasivă - o problemă acută în aplicațiile automotive, industriale și aerospațiale, unde temperatura ambiantă e deja ridicată.

Al treilea este **fiabilitatea și mentenanța**: un consum mai mic înseamnă solicitare termică mai mică și mai puține cicluri de încărcare, deci o mentenanță mai rară - esențial

când mii de noduri dintr-o rețea de senzori ar necesita intervenții individuale.

Al patrulea este **impactul la scară largă**: o economie de energie nesemnificativă la un singur dispozitiv devine importantă înmulțită cu milioane de exemplare, reducând costurile de exploatare, necesarul de răcire și cantitatea de baterii produse și înlocuite. Eficiența energetică este deci deopotrivă un obiectiv tehnic, economic și ecologic.

4.2 Putere, energie și sarcină electrică

Analiza energetică a unui sistem embedded cere să deosebim clar puterea, energia și sarcina electrică - termeni folosiți uneori incorect ca sinonime, dar care descriu mărimi fizice distincte. Figura 4.2 prezintă relația conceptuală dintre tensiune, curent, sarcină, putere și energie.

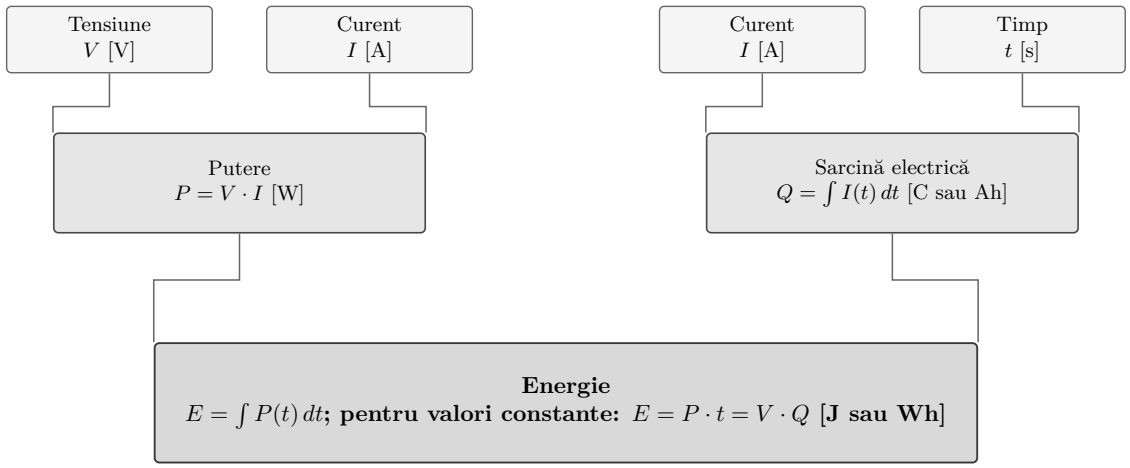


Figura 4.2 Relația dintre mărimile electrice utilizate în analiza energetică.

4.2.1 Putere, energie și sarcină electrică

Puterea descrie viteza cu care energia este consumată, transferată sau transformată, și se măsoară în wați (W). Puterea instantanee absorbită de un circuit este $P(t) = V(t) \cdot I(t)$, unde $V(t)$ e tensiunea la borne și $I(t)$ curentul absorbit; pentru tensiune și curent constante, relația devine simplu $P = V \cdot I$.

Exemplu

Un microcontroler funcționează la 3,3 V și absoarbe 12 mA în modul activ. Puterea consumată este $P = 3{,}3\,\text{V} \times 12\,\text{mA} = 39{,}6\,\text{mW}$ - o valoare instantanee, care nu indică direct energia consumată într-o oră sau autonomia bateriei.

Puterea trebuie privită atât ca valoare medie, cât și ca valoare maximă: media determină energia totală consumată, iar maximul determină dimensionarea sursei de alimentare și capacitatea ei de a susține impulsurile de curent.

Energia este cantitatea totală consumată sau transferată într-un interval de timp, mă-

surată în jouli (J); se obține prin integrarea puterii, $E = \int_0^T P(t)\,dt$, sau simplu $E = P \cdot T$ pentru putere constantă. În electronică și pentru baterii se folosește frecvent watt-ora, $1\,\text{Wh} = 3600\,\text{J}$ - o unitate de energie, nu de putere: o baterie de 10 Wh poate da, ideal, 1 W timp de zece ore sau 10 W timp de o oră.

Exemplu

Un modul radio consumă 100 mW constant timp de zece secunde: $E = 0{,}1\,\text{W} \times 10\,\text{s} = 1\,\text{J} \approx 0{,}278\,\text{mWh}$.

Sarcina electrică transferată printr-un circuit se obține integrând curentul, $Q = \int_0^T I(t)\,dt$, și se măsoară în coulombi (C); pentru baterii se exprimă frecvent în amper-oră, cu $1\,\text{Ah} = 3600\,\text{C}$ și $1\,\text{mAh} = 3{,}6\,\text{C}$. Capacitatea în Ah e o cantitate de sarcină, nu de energie - două baterii cu aceeași capacitate dar tensiuni diferite stochează energii diferite. Energia nominală se aproximează prin $E_{\text{nominal}} \approx V_{\text{nominal}} \cdot Q_{\text{nominal}}$ (volți × amper-oră = watt-oră).

Exemplu

Două baterii de 2000 mAh ($Q_{\text{nominal}} = 2\,\text{Ah}$): la 1,2 V (NiMH), $E_1 \approx 2{,}4\,\text{Wh}$; la 3,7 V (Li-Ion), $E_2 \approx 7{,}4\,\text{Wh}$ - de peste trei ori mai multă energie, deși capacitatea în Ah e identică.

Relația bazată pe tensiunea nominală e doar o estimare: tensiunea reală variază pe durata descărcării, iar energia efectiv utilizabilă depinde și de curent, temperatură, pragul minim acceptat de sistem și eficiența convertorului de alimentare. Tabelul 4.1 sintetizează mărimile de mai sus.

Tabela 4.1 Mărimi fundamentale utilizate în analiza energetică

Mărime	Simbol	Unitate	Semnificație
Tensiune	V	volt, V	Diferența de potențial electric aplicată circuitului.
Curent	I	amper, A	Viteza de transfer a sarcinii electrice.
Sarcină electrică	Q	coulomb, C; Ah	Cantitatea totală de sarcină transferată sau disponibilă.
Putere	P	watt, W	Viteza cu care energia este consumată sau transferată.
Energie	E	joule, J; Wh	Cantitatea totală consumată sau stocată.
Timp de funcționare	T	secundă, oră	Durata pentru care sistemul poate îndeplini funcția cerută.

4.3 Profilul energetic al unui sistem embedded

Consumul unui sistem embedded nu este, în general, constant: procesorul, senzorii și interfețele de comunicație se activează doar când e nevoie, iar între operații sistemul poate

trece într-o stare de consum redus. Un profil energetic descrie această variație a puterii sau curentului în timp și permite identificarea stărilor care domină consumul total și a impulsurilor care solicită sursa de alimentare. Figura 4.3 arată profilul simplificat al unui nod IoT care se trezește periodic, măsoară, procesează și transmite un pachet.

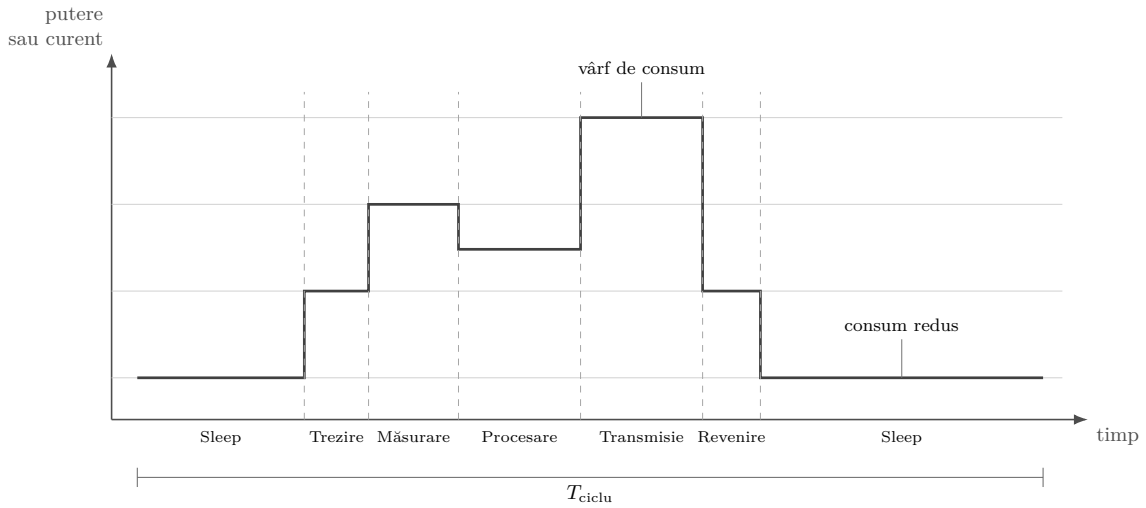


Figura 4.3 Profil energetic simplificat al unui nod IoT cu funcționare periodică.

Din acest profil se extrag patru mărimi. **Puterea instantanee** arată consumul la un moment dat - utilă pentru dimensionarea sursei și analiza căderilor de tensiune, dar insuficientă pentru autonomie: un sistem poate consuma sute de miliwați la transmisie, însă doar câteva milisecunde, și rămâne cu putere medie mică dacă restul timpului stă în microwați.

4.3.1 Puterea medie, puterea maximă și factorul de activitate

Puterea medie pe intervalul T este:

$$P_{\text{medie}} = \frac{1}{T} \int_0^T P(t)\, dt. \tag{4.2}$$

Dacă sistemul funcționează în n stări distincte, puterea medie poate fi calculată prin:

$$P_{\text{medie}} = \frac{\sum_{i=1}^{n} P_i t_i}{\sum_{i=1}^{n} t_i}. \tag{4.3}$$

unde P_i este puterea din starea i, iar t_i durata ei; similar pentru curentul mediu, $I_{\text{mediu}} = \sum I_i t_i / \sum t_i$ - calcul corect pentru autonomie prin raportul Ah/mA doar dacă tensiunile și eficiența lanțului de alimentare sunt tratate corespunzător.

Puterea maximă este cea mai mare valoare înregistrată în interval, $P_{\max} = \max_{t \in [0,T]} P(t)$, și influențează dimensionarea sursei, a convertorului DC–DC, a condensatoarelor de decuplare și a traseelor de alimentare - la modulele radio, curentul de transmisie poate fi de sute

sau mii de ori mai mare decât în sleep, iar bateria și convertorul trebuie să susțină impulsul fără ca tensiunea să scadă sub prag.

Factorul de activitate (*duty cycle*) este fracțiunea din perioadă în care sistemul e activ:

$$D = \frac{t_{\text{activ}}}{T}. \tag{4.4}$$

Pentru un sistem cu două stări, activ și sleep, puterea medie este:

$$P_{\text{medie}} = D \cdot P_{\text{activ}} + (1 - D) \cdot P_{\text{sleep}}. \tag{4.5}$$

Reducerea factorului de activitate e una dintre cele mai eficiente metode de creștere a autonomiei unui sistem periodic - dar nu garantează automat reducerea energiei, pentru că pornirea oscilatoarelor, inițializarea senzorilor și stabilizarea comunicației au un cost fix; dacă sistemul se trezește prea des, energia tranzițiilor devine semnificativă.

Energia pe un ciclu de funcționare. Pentru sisteme periodice e utilă energia dintr-un ciclu complet, $E_{\text{ciclu}} = \sum_{i=1}^{n} P_i t_i$, de unde puterea medie rezultă ca $P_{\text{medie}} = E_{\text{ciclu}}/T_{\text{ciclu}}$ - metodă potrivită pentru noduri IoT și dispozitive care repetă aceeași secvență de măsurare și transmisie.

Exemplu

Un dispozitiv funcționează periodic, cu o perioadă de o secundă. În fiecare ciclu:

- consumă 8 mA timp de 20 ms;
- consumă 10 μA în restul de 980 ms.

Curentul mediu este:

$$I_{\text{mediu}} = \frac{8\,\text{mA} \times 0{,}020\,\text{s} + 0{,}010\,\text{mA} \times 0{,}980\,\text{s}}{1\,\text{s}}.$$

Rezultă:

$$I_{\text{mediu}} = 0{,}1698\,\text{mA} = 169{,}8\,\mu\text{A}.$$

Dacă tensiunea sistemului este de 3,3 V, puterea medie este:

$$P_{\text{medie}} = 3{,}3\,\text{V} \times 0{,}1698\,\text{mA} \approx 0{,}560\,\text{mW}.$$

Deși curentul activ este de 8 mA, valoarea medie este mult mai redusă datorită factorului de activitate de numai 2%.

4.4 Metrici energetice

Nu există o singură metrică potrivită pentru toate sistemele embedded - alegerea depinde de funcția dispozitivului: pentru un nod pe baterie contează autonomia și energia per ciclu; pentru un procesor, energia per operație; pentru un transceiver radio, energia per bit sau per pachet. Tabelul 4.2 le sintetizează.

Energia per operație, $E_{\text{op}} = E_{\text{total}}/N_{\text{op}}$, e utilă pentru compararea a două soluții

Tabela 4.2 Metrici utilizate în evaluarea energetică a sistemelor embedded

Metrică	Relație	Utilizare
Putere instantanee	$P(t) = V(t)I(t)$	Identificarea impulsurilor și dimensionarea sursei.
Putere medie	$P_{\text{medie}} = E/T$	Estimarea autonomiei și a disipării termice medii.
Putere maximă	$P_{\max} = \max P(t)$	Dimensionarea convertoarelor și analiza stabilității alimentării.
Energie per ciclu	$E_{\text{ciclu}} = \sum P_i t_i$	Sisteme periodice, noduri IoT și senzori autonomi.
Energie per operație	$E_{\text{op}} = E_{\text{total}}/N_{\text{op}}$	Compararea procesoarelor, algoritmilor și acceleratoarelor.
Energie per instrucțiune	$E_{\text{instr}} = E_{\text{total}}/N_{\text{instr}}$	Evaluarea consumului unei arhitecturi sau al unui program.
Energie per bit	$E_{\text{bit}} = E_{\text{total}}/N_{\text{bit}}$	Compararea tehnologiilor și configurațiilor de comunicație.
Energie per pachet	$E_{\text{pachet}} = E_{\text{total}}/N_{\text{pachete}}$	Evaluarea comunicațiilor cu costuri de pornire și protocol.

care fac aceeași funcție: un accelerator hardware poate avea putere instantanee mai mare decât un microcontroler, dar termină mult mai repede, deci consumă mai puțină energie totală. Valorile trebuie citite împreună cu tipul operației, tehnologia, tensiunea, frecvența și condițiile de măsurare - altfel spun puțin.

4.4.1 Energia per bit, per pachet și metrici combinate

Energia per bit, $E_{\text{bit}} = E_{\text{comunicație}}/N_{\text{bit}}$, e utilă pentru costul transferului, dar poate ascunde energia fixă de pornire a transceiverului, sincronizare și stabilire a conexiunii. Pentru pachete scurte, **energia per pachet** e mai reprezentativă: $E_{\text{pachet}} = E_{\text{pornire}} + E_{\text{procesare}} + E_{\text{transmisie}} + E_{\text{recepție}}$ - reducerea numărului de pachete poate economisi mai mult decât tăierea câtorva biți din fiecare mesaj (revenim asupra acestui aspect la comunicațiile wireless).

Reducerea energiei nu trebuie să degradeze necontrolat performanța: în aplicațiile cu constrângeri temporale, algoritmul trebuie să respecte un termen limită. O **metrică combinată** des folosită e produsul energie–întârziere:

$$EDP = E \cdot T_{\text{execuție}}. \tag{4.6}$$

O valoare redusă indică un compromis favorabil între energia consumată și timpul de execuție. Totuși, EDP nu este o metrică universală. Pentru un dispozitiv care trebuie să funcționeze ani dintr-o baterie, energia poate avea prioritate. Pentru un sistem de control în timp real, respectarea termenului limită poate fi condiția principală.

4.5 Estimarea autonomiei

Estimarea autonomiei este una dintre cele mai frecvente activități în proiectarea sistemelor alimentate din baterii. Un calcul preliminar poate fi realizat pe baza energiei nominale și a puterii medii:

$$T_{\text{ideal}} = \frac{E_{\text{nominal}}}{P_{\text{medie}}}. \tag{4.7}$$

Această relație presupune că întreaga energie nominală poate fi utilizată și că nu există pierderi. Într-un sistem real, energia disponibilă este mai redusă.

O estimare îmbunătățită poate fi exprimată prin:

$$T_{\text{estimat}} = \frac{E_{\text{nominal}} \cdot k_{\text{utilizare}} \cdot \eta_{\text{alimentare}} \cdot k_{\text{temperatur}} \cdot k_{\text{mbtrnire}}}{P_{\text{medie}}}. \tag{4.8}$$

unde:

- $k_{\text{utilizare}}$ reprezintă fracțiunea din energia nominală care poate fi utilizată înainte ca tensiunea să scadă sub pragul minim;
- $\eta_{\text{alimentare}}$ reprezintă eficiența convertorului;
- $k_{\text{temperatur}}$ modelează reducerea energiei disponibile la temperatura de funcționare;
- k_{mbtrnire} modelează pierderea de capacitate în timp.

Această formulă oferă o aproximație inginerească. Factorii nu sunt complet independenți, iar pentru simulări precise trebuie utilizate curbele de descărcare și modelele furnizate de producătorul bateriei.

Estimarea pe baza capacității în Ah Dacă sistemul este alimentat direct dintr-o baterie și tensiunea poate fi considerată compatibilă pe întreaga durată de funcționare, autonomia ideală poate fi aproximată prin:

$$T_{\text{ideal}} = \frac{Q_{\text{baterie}}}{I_{\text{mediu}}}. \tag{4.9}$$

Dacă Q_{baterie} este exprimată în mAh și I_{mediu} în mA, rezultatul este exprimat în ore.

Această relație nu trebuie aplicată direct atunci când bateria și consumatorul funcționează la tensiuni diferite sau atunci când există un convertor a cărui eficiență variază cu sarcina. În aceste situații, calculul pe baza energiei în Wh este mai corect.

4.5.1 Exemplu complet de estimare

Exemplu

Se consideră dispozitivul analizat anterior, având:

- curent activ de 8 mA timp de 20 ms;
- curent sleep de 10 μA timp de 980 ms;
- tensiunea sistemului de 3,3 V;
- baterie Li-Ion de 3,7 V și 2000 mAh;

- eficiența medie a convertorului de 90%;
- fracție de energie utilizabilă de 80%.

Curentul mediu al sistemului a fost calculat ca:

$$I_{\text{mediu}} = 0{,}1698\,\text{mA}.$$

Puterea medie este:

$$P_{\text{medie}} = 3{,}3\,\text{V} \times 0{,}1698\,\text{mA} \approx 0{,}560\,\text{mW}.$$

Energia nominală a bateriei este:

$$E_{\text{nominal}} = 3{,}7\,\text{V} \times 2\,\text{Ah} = 7{,}4\,\text{Wh}.$$

Energia estimată disponibilă la consumator este:

$$E_{\text{util}} = 7{,}4\,\text{Wh} \times 0{,}8 \times 0{,}9 = 5{,}328\,\text{Wh}.$$

Autonomia estimată este:

$$T_{\text{estimat}} = \frac{5{,}328\,\text{Wh}}{0{,}000560\,\text{W}} \approx 9514\,\text{h}.$$

Exprimată în zile:

$$T_{\text{estimat}} \approx 396\,\text{zile}.$$

Rezultatul nu include autodescărcarea bateriei, consumul circuitelor de protecție, variația eficienței convertorului sau reducerea capacității la temperaturi scăzute. Autonomia reală va fi, prin urmare, mai mică.

Diferența dintre autonomia calculată și cea măsurată vine din surse ca: autodescărcarea bateriei, curentul propriu al regulatorului, curenții de pierderi, reducerea capacității la frig, îmbătrânirea bateriei, impulsurile mari de curent, pragul minim de tensiune, retransmisiile radio neprevăzute și toleranțele dintre exemplare. Pentru sisteme ce trebuie să funcționeze ani de zile, curentul de repaus al regulatorului și autodescărcarea pot deveni comparabile cu consumul microcontrolerului însuși - o baterie mare nu garantează autonomie mare dacă se autodescarcă mult sau dacă sursa consumă permanent un curent ridicat.

Calculele teoretice trebuie validate prin măsurători: un multimetru dă o medie aproximativă, dar poate rata impulsurile scurte, așa că pentru o analiză completă se folosesc rezistențe de șunt cu osciloscop, analizoare de putere, surse programabile cu măsurare de curent sau instrumente dedicate de profilare. Instrumentul trebuie să acopere atât curenți de microamperi în sleep, cât și impulsuri de zeci-sute de miliamperi în activ, iar măsurarea trebuie făcută pe un ciclu reprezentativ - o medie pe interval prea scurt denaturează estimarea autonomiei.

4.6 Principalii consumatori de energie dintr-un sistem embedded

Optimizarea energetică a unui sistem embedded trebuie să înceapă prin identificarea componentelor care contribuie efectiv la consumul total. Presupunerea că procesorul reprezintă întotdeauna principalul consumator poate conduce la decizii de proiectare ineficiente. În funcție de aplicație, energia poate fi dominată de comunicația wireless, de senzori, de afișaj, de memoria externă sau de actuatoare.

Distribuția consumului trebuie analizată pe baza unui profil real de funcționare. O componentă cu o putere instantanee ridicată poate avea o contribuție redusă la energia totală dacă este activată numai pentru perioade scurte. În schimb, un circuit cu un consum aparent nesemnificativ poate deveni important dacă funcționează permanent.

Figura 4.4 prezintă principalele categorii de consumatori întâlnite într-un sistem embedded.

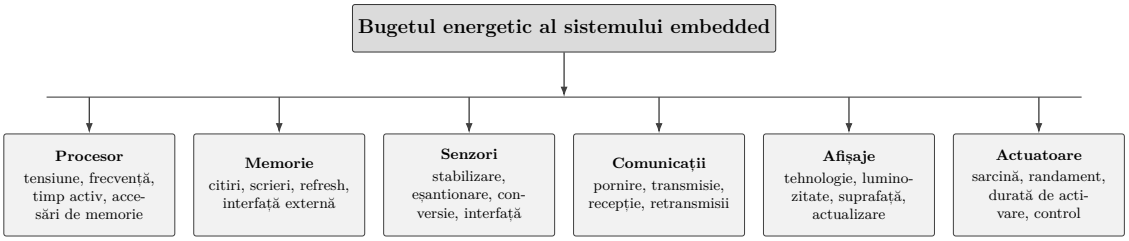


Figura 4.4 Principalele categorii de consumatori energetici dintr-un sistem embedded.

Contribuția unei componente la energia totală poate fi exprimată prin:

$$E_i = \int_0^T P_i(t)\, dt, \tag{4.10}$$

unde $P_i(t)$ reprezintă puterea consumată de componenta i.

Energia totală a sistemului este:

$$E_{\text{total}} = \sum_{i=1}^{n} E_i. \tag{4.11}$$

Ponderea energetică a unei componente este:

$$p_i = \frac{E_i}{E_{\text{total}}} \cdot 100\%. \tag{4.12}$$

Această relație permite prioritizarea corectă a activităților de optimizare. Reducerea cu 50% a consumului unei componente care contribuie cu numai 2% la energia totală va îmbunătăți autonomia sistemului cu aproximativ 1%. În schimb, o reducere modestă aplicată consumatorului dominant poate avea un efect semnificativ.

Procesorul execută instrucțiunile și coordonează perifericele; consumul depinde de arhitectură, tensiune, frecvență, activitatea internă și codul executat, și nu e constant - operații aritmetice pe registre au un profil diferit față de accese intense la memorie sau activarea ac-

celeratoarelor. Trebuie analizat împreună cu timpul de execuție, $E_{\text{activitate}} = P_{\text{activ}} \cdot T_{\text{execuție}}$: un nucleu mai performant poate consuma mai mult instantaneu dar termină mai repede și revine în repaus, deci reducerea frecvenței nu duce automat la reducerea energiei (puterea scade, dar timpul poate crește).

Memoria consumă prin accesări de citire/scriere, curenți de pierderi și, la unele tehnologii, reîmprospătare periodică; accesul extern (magistrale, linii de semnal, distanțe fizice mai mari) e mai costisitor decât operațiile pe registre. Tehnologiile diferă: registrele interne sunt rapide dar ocupă suprafață mare, SRAM e rapidă dar are consum static, DRAM cere reîmprospătare, Flash are consum redus la păstrare dar programare/ștergere costisitoare, iar memoria externă adaugă energia interfeței. Localitatea datelor și cache-urile folosite eficient îmbunătățesc simultan performanța și consumul.

Senzorii transformă mărimi fizice în semnale electrice; consumul include elementul sensibil, condiționarea analogică, conversia A/D și comunicația, iar un senzor trece prin stări succesive (oprit, standby, stabilizare, măsurare, transfer). La unii senzori, stabilizarea durează mai mult decât conversia - pornirea/oprirea prea frecventă poate consuma mai mult decât păstrarea unui mod intermediar. Frecvența de eșantionare și rezoluția trebuie alese după nevoia reală a aplicației: date mai precise sau mai dese decât necesar costă energie suplimentară pe tot lanțul (senzor, convertor, procesor, memorie, comunicație).

Comunicațiile consumă pentru codificare, transmisie, recepție și verificare, plus, la wireless, oscilatoare, sintetizatoare și amplificatoare radio. Energia unui transfer nu ține doar de mărimea mesajului: $E_{\text{transfer}} = E_{\text{pornire}} + E_{\text{date}} + E_{\text{protocol}} + E_{\text{retransmisii}}$ - la pachete scurte, pornirea și sincronizarea pot domina costul total, iar gruparea mai multor valori într-un singur pachet e adesea mai eficientă decât trimiterea imediată a fiecărei măsurători (detaliem costul wireless mai departe în capitol).

Afișajele depind de tehnologie, suprafață, luminozitate, frecvența de actualizare și conținut: la LCD cu iluminare de fundal, sursa de lumină domină consumul (reducerea luminozității sau stingerea ei economisește mult); la OLED, fiecare pixel emite lumină, deci consumul urmează numărul și intensitatea pixelilor activi (interfețe întunecate ajută); e-paper consumă mai ales la actualizare și păstrează imaginea fără alimentare continuă, potrivit pentru conținut rar schimbat, nu pentru video.

4.6.1 Actuatoarele și interpretarea bugetului energetic

Actuatoarele (motoare, servomotoare, electrovalve, relee, rezistențe de încălzire, surse de iluminat, traductoare acustice) transformă energia electrică în mișcare, forță, căldură sau lumină și sunt frecvent consumatorii dominanți în aplicațiile robotice și industriale - un microcontroler consumă miliwați, un motor poate cere zeci-sute de wați, deci optimizarea software-ului procesorului are impact limitat dacă strategia de acționare rămâne neschimbată. Economiile reale vin din: dimensionarea corectă a actuatorului, un convertor de putere eficient, reducerea frecărilor, optimizarea traiectoriei, recuperarea energiei și evitarea menținerii inutile a cuplului sau forței.

Tabelul 4.3 sintetizează factorii care influențează principalele categorii de consumatori.

Tabela 4.3 Consumatori energetici și direcții generale de optimizare

Componentă	Factori care influențează consumul	Direcții de optimizare
Procesor	Tensiune, frecvență, timp activ, accesări de memorie și unități funcționale	DVFS, acceleratoare, moduri sleep, cod eficient și reducerea timpului activ
Memorie	Număr de accesări, tehnologie, dimensiune, refresh și interfață externă	Localitatea datelor, cache, memorii locale și reducerea transferurilor
Senzori	Frecvență de eșantionare, rezoluție, stabilizare și condiționare analogică	Eșantionare adaptivă, duty cycling și alegerea rezoluției necesare
Comunicație	Putere de emisie, timp activ, dimensiunea pachetului, protocol și retransmisii	Gruparea mesajelor, procesare locală și alegerea tehnologiei potrivite
Afișaj	Tehnologie, luminozitate, suprafață și rată de actualizare	Luminozitate adaptivă, stingere automată și actualizări selective
Actuator	Sarcină mecanică, randament, regim de control și durată de activare	Control optim, dimensionare corectă și reducerea pierderilor mecanice

4.7 Consumul energetic al procesoarelor CMOS

Majoritatea microcontrolerelor și procesoarelor embedded moderne sunt realizate în tehnologie CMOS, iar consumul lor se împarte în patru componente:

$$P_{\text{total}} = P_{\text{dinamic}} + P_{\text{scurtcircuit}} + P_{\text{static}} + P_{\text{analogic}}. \tag{4.13}$$

Puterea dinamică vine din încărcarea/descărcarea capacităților interne la comutarea tranzistoarelor; cea de scurtcircuit apare scurt, în timpul tranziției logice, când rețelele de sus și de jos conduc simultan; cea statică e produsă de curenții de pierderi prezenți chiar fără comutare; iar componenta analogică (oscilatoare, referințe de tensiune, convertoare) nu e descrisă de modelul digital CMOS. Figura 4.5 arată aceste componente.

Ponderea fiecărei componente depinde de tehnologie, tensiune, temperatură, frecvență și starea procesorului: în activ domină de regulă puterea dinamică, iar în repaus prelungit contează mai mult curenții de pierderi și circuitele rămase alimentate.

4.8 Puterea dinamică și puterea statică

4.8.1 Puterea dinamică

Puterea dinamică a unui circuit CMOS se aproximează prin $P_{\text{dinamic}} = \alpha C_{\text{ef}} V^2 f$, unde α e factorul de activitate (proporția nodurilor care comută pe unitatea de timp - două programe la aceeași frecvență pot consuma diferit după unitățile funcționale pe care le activează), C_{ef} capacitatea echivalentă comutată (crește cu numărul și dimensiunea tranzistoarelor și lungimea interconexiunilor), V tensiunea și f frecvența. Relația arată că puterea dinamică variază liniar cu frecvența și aproximativ pătratic cu tensiunea [22, 69].

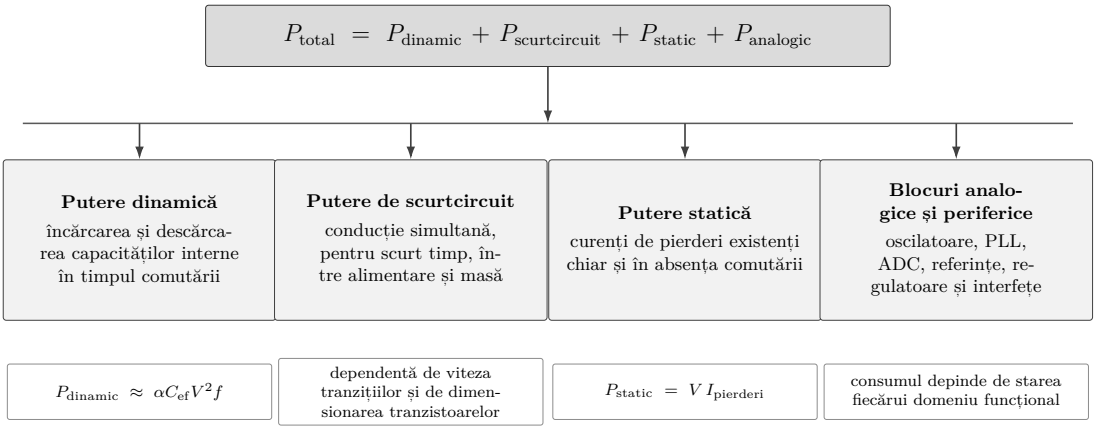


Figura 4.5 Componentele principale ale consumului unui procesor realizat în tehnologie CMOS.

4.8.2 Energia dinamică a unei activități

Pentru o activitate cu un număr fix de cicluri, energia dinamică se aproximează prin:

$$E_{\text{dinamic}} = P_{\text{dinamic}} \cdot T_{\text{execuție}}. \tag{4.14}$$

Dacă numărul de cicluri este N_{cicluri}, atunci:

$$T_{\text{execuție}} = \frac{N_{\text{cicluri}}}{f}. \tag{4.15}$$

Prin înlocuire se obține, într-un model idealizat:

$$E_{\text{dinamic}} = \alpha C_{\text{ef}} V^2 N_{\text{cicluri}}. \tag{4.16}$$

Frecvența nu mai apare explicit: la cicluri fixe și tensiune constantă, reducerea frecvenței scade puterea dar crește proporțional timpul, iar energia dinamică ideală rămâne aproape constantă. În practică însă, reducerea frecvenței permite adesea și reducerea tensiunii - iar economiile reale vin din scăderea lui V^2, nu doar din frecvență.

Exemplu

Un procesor execută aceeași activitate în două puncte de funcționare:

- configurația 1: $V_1 = 1{,}2\,\text{V}$ și $f_1 = 100\,\text{MHz}$;
- configurația 2: $V_2 = 0{,}9\,\text{V}$ și $f_2 = 70\,\text{MHz}$.

Presupunând că factorul de activitate și capacitatea echivalentă nu se modifică, raportul puterilor dinamice este:

$$\frac{P_2}{P_1} = \left(\frac{0{,}9}{1{,}2}\right)^2 \cdot \frac{70}{100} = 0{,}39375.$$

Puterea dinamică scade cu aproximativ 60,6%.

Timpul de execuție crește însă cu factorul:

$$\frac{T_2}{T_1} = \frac{100}{70} \approx 1{,}43.$$

Raportul energiilor dinamice devine:

$$\frac{E_2}{E_1} = 0{,}39375 \times 1{,}43 \approx 0{,}5625.$$

Energia dinamică scade cu aproximativ 43,8%, datorită reducerii tensiunii.

Puterea de scurtcircuit apare când, pentru un interval scurt în timpul tranziției logice, tranzistoarele rețelei superioare și inferioare conduc simultan, creând un curent direct între alimentare și masă; depinde de viteza tranzițiilor, tensiune și dimensionare, și e de regulă mai mică decât puterea de încărcare a capacităților, dar nu nulă.

4.8.3 Puterea statică

Puterea statică, $P_{\text{static}} = V \cdot I_{\text{pierderi}}$, vine din curenți de pierderi prezenți chiar fără comutare, dependenți de tehnologie, temperatură, tensiune și starea internă a circuitului. Temperatura mai mare crește pierderile, ceea ce crește puterea și deci temperatura - o reacție pozitivă care, în tehnologiile moderne, poate face din puterea statică o parte importantă a consumului total, mai ales în modurile cu activitate de comutare redusă.

Ecuația CMOS dinamică nu descrie complet oscilatoarele, convertoarele A/D, comparatoarele, referințele interne și regulatoarele de tensiune - un microcontroler „inactiv” poate consuma în continuare dacă rămân active oscilatorul principal, bucla PLL, referința ADC, detectorul brown-out, watchdog-ul, SRAM-ul sau interfețele de depanare. Un consum minim real cere analiza fiecărui domeniu funcțional, nu doar a nucleului CPU.

4.9 Tehnici hardware pentru reducerea consumului

Tehnicile hardware urmăresc reducerea tensiunii, a activității de comutare, a capacității active sau a curenților de pierderi. Ele pot fi aplicate la nivelul procesorului, al perifericelor sau al întregului sistem.

Figura 4.6 sintetizează principalele mecanisme hardware utilizate în sistemele embedded.

Figura 4.6 Tehnici hardware utilizate pentru reducerea consumului energetic.

4.9.1 Prezentare generală a tehnicilor hardware

Scalarea dinamică a tensiunii și frecvenței DVFS, de la *Dynamic Voltage and Frequency Scaling*, permite adaptarea tensiunii și a frecvenței în funcție de sarcina de calcul.

Atunci când performanța maximă nu este necesară, procesorul poate funcționa la o frecvență mai redusă și, dacă tehnologia permite, la o tensiune mai mică. Reducerea tensiunii produce economii importante datorită dependenței pătratice a puterii dinamice.

Scăderea tensiunii reduce însă viteza de comutare a tranzistoarelor, ceea ce limitează frecvența maximă. Din acest motiv, tensiunea și frecvența sunt controlate împreună prin puncte de funcționare validate.

DVFS presupune anumite costuri:

- timpul necesar stabilizării tensiunii;
- reconfigurarea ceasului;
- energia consumată în timpul tranziției;
- complexitatea algoritmului de control;
- necesitatea respectării termenelor temporale.

Clock gating oprește semnalul de ceas pentru blocurile digitale care nu sunt utilizate. În absența ceasului, registrele interne nu își schimbă starea, iar activitatea de comutare este redusă.

Clock gating diminuează în principal puterea dinamică. Blocul rămâne alimentat, iar informațiile din registre sunt păstrate. Curenții de pierderi continuă însă să existe.

Tehnica este potrivită pentru periferice sau unități funcționale care trebuie reactivate rapid. De exemplu, ceasul unui timer, al unei interfețe SPI sau al unui accelerator poate fi oprit atunci când modulul nu este utilizat.

Power gating deconectează alimentarea unui bloc hardware, reducând atât activitatea de comutare, cât și o parte din puterea statică - cu prețul pierderii stării interne (dacă nu se folosesc registre de retenție sau memorii alimentate separat) și al unei reveniri mai costisitoare (realimentare, stabilizarea tensiunii, reinițializare). Oferă economii mai mari decât clock gating, dar cu latență și cost de tranziție mai ridicate.

Domenii independente de alimentare și ceas Un sistem pe cip poate fi împărțit în domenii independente (procesor, memorie, periferice analogice, comunicație), fiecare cu ceas sau alimentare proprie - de exemplu domeniul radio poate fi deconectat cât timp timerul de consum redus continuă să funcționeze pentru trezirea periodică. Separarea cere însă circuite suplimentare pentru izolarea semnalelor, conversia nivelurilor logice și păstrarea stării.

Acceleratoarele hardware (unități criptografice, controlere DMA, DSP, acceleratoare grafice, unități pentru rețele neuronale, calcul de sume de control) execută o funcție specializată cu mai puține transferuri și instrucțiuni decât un procesor general; pot avea putere instantanee mai mare, dar termină mult mai repede, așa că eficiența trebuie evaluată prin energia per activitate, nu prin putere.

4.9.2 Controlul accesului la memorie

Reducerea transferurilor de date e o tehnică hardware/arhitecturală importantă: memorii locale, cache-uri, buffere și controlere DMA limitează accesările externe. DMA transferă un bloc de date fără implicarea procesorului pentru fiecare element, lăsând nucleul liber pentru altă activitate sau pentru repaus - dar la transferuri foarte scurte, costul configurării DMA poate egala economia obținută, deci alegerea depinde de dimensiunea și frecvența transferurilor.

Tabela 4.4 Comparația principalelor tehnici hardware de reducere a consumului

Tehnică	Componenta redusă	Avantaj principal	Limitare principală
DVFS	Puterea dinamică	Economii importante prin reducerea tensiunii	Performanță mai redusă și cost de tranziție
Clock gating	Activitatea de comutare	Trezire rapidă și păstrarea stării	Nu elimină puterea statică
Power gating	Puterea dinamică și statică	Consum foarte redus al blocului oprit	Pierderea stării și latență de reactivare
Domenii independente	Consum selectiv al subsistemelor	Oprirea numai a blocurilor neutilizate	Complexitate hardware suplimentară
Accelerator hardware	Energia per activitate	Execuție rapidă și specializată	Flexibilitate redusă și suprafață suplimentară
DMA și memorii locale	Transferurile și activitatea CPU	Mai puține treziri și accesări externe	Cost de configurare și resurse suplimentare

4.10 Modurile de consum redus

Procesoarele și microcontrolerele moderne oferă mai multe stări energetice. Denumirile exacte diferă între producători, dar principiile generale sunt similare.

Figura 4.7 prezintă o ierarhie generică a stărilor de consum.

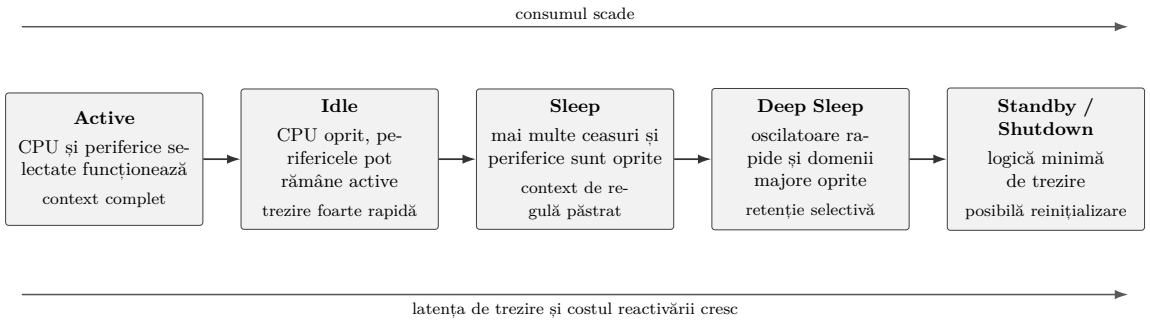


Figura 4.7 Relația generală dintre consum, păstrarea stării și latența de trezire.

4.10.1 Stările de consum redus

Cinci stări tipice apar pe majoritatea platformelor, de la cel mai activ la cel mai profund consum. În **modul activ**, nucleul execută instrucțiuni cu ceasurile și perifericele necesare funcționale - chiar și aici sunt posibile optimizări (oprirea ceasului perifericelor neutilizate, reducerea frecvenței, DMA, dezactivarea blocurilor analogice inactive). În **Idle**, execuția nucleului e suspendată dar restul sistemului rămâne alimentat, cu trezire rapidă - potrivit pentru pauze scurte, unde un mod profund nu s-ar amortiza. În **Sleep** se opresc mai multe ceasuri și periferice, cu registre și SRAM păstrate și trezire prin timer, întrerupere externă sau periferic selectat - consum mai mic decât Idle, dar latență mai mare. În **Deep Sleep** se opresc oscilatoarele rapide, PLL-urile și majoritatea perifericelor, rămânând activ doar un domeniu minim (timer, controler de trezire, o parte din memorie); revenirea cere pornirea și stabilizarea surselor de ceas și, uneori, reconfigurarea perifericelor. În **Standby/Shutdown**, memoria principală și registrele se pot pierde, sistemul păstrează doar informații minime într-un domeniu de retenție sau memorie nevolatilă, iar revenirea seamănă cu o resetare - potrivit pentru inactivitate îndelungată.

Tabela 4.5 Caracteristicile generale ale modurilor de consum redus

Stare	Elemente active	Păstrarea contextului	Latența de revenire
Active	CPU, memorii și periferice selectate	Completă	Nu este necesară trezirea
Idle	Memorie și majoritatea perifericelor	Completă	Foarte redusă
Sleep	Domenii și surse de trezire selectate	De regulă completă	Redusă sau medie
Deep Sleep	Domeniu de consum redus și memorie de retenție	Parțială sau completă	Medie sau ridicată
Standby/Shutdown	Doar logică minimă de trezire	Limitată	Ridicată; posibilă reinițializare

Valorile concrete de curent și latență trebuie luate din fișa tehnică a microcontrolerului - două moduri cu același nume pot avea comportamente diferite pe procesoare diferite.

Nu toate evenimentele pot trezi procesorul din orice stare; sursele posibile includ timerul de timp real, un pin extern, watchdog-ul, un comparator de consum redus, o interfață de comunicație, o variație analogică sau un eveniment de senzor - dacă trezirea cere păstrarea unui periferic activ, consumul stării de repaus crește, așa că proiectantul trebuie să identifice mecanismul minim necesar.

4.10.2 Energia tranzițiilor și timpul de rentabilizare

Intrarea și ieșirea dintr-un mod de consum redus necesită energie (oprirea/repornirea oscilatoarelor, stabilizarea tensiunii, eventuala reinițializare), deci un mod profund e avan-

tajos doar dacă pauza e suficient de lungă cât să compenseze acest cost. Timpul minim de rentabilizare, într-un model simplificat, este:

$$T_{\text{BE}} = \frac{E_{\text{intrare}} + E_{\text{ieșire}}}{P_{\text{stare inițial}} - P_{\text{sleep}}}, \tag{4.17}$$

unde T_{BE} este timpul de *break-even*.

Exemplu

Un sistem consumă 3 mW în Idle și 0,03 mW în Deep Sleep. Intrarea și ieșirea din Deep Sleep consumă împreună:

$$E_{\text{tranziții}} = 0{,}15\,\text{mJ}.$$

Timpul de rentabilizare este:

$$T_{\text{BE}} = \frac{0{,}15\,\text{mJ}}{3\,\text{mW} - 0{,}03\,\text{mW}} \approx 50{,}5\,\text{ms}.$$

Dacă pauza estimată este mai scurtă de aproximativ 50 ms, intrarea în Deep Sleep nu produce economie energetică în modelul analizat. Pentru pauze mai lungi, modul profund devine avantajos.

Păstrarea întregii memorii SRAM reduce timpul de revenire, dar cere alimentarea permanentă a celulelor - retenția selectivă a unei regiuni reduce consumul. Software-ul poate salva înainte, în memorie nevolatilă, doar informațiile strict necesare, dar scrierile Flash frecvente consumă energie și uzează memoria; alegerea trebuie să echilibreze energia de retenție, costul salvării și timpul de reinițializare.

4.11 Tehnici software pentru optimizarea energetică

Software-ul controlează cât timp rămân componentele active și cât de des e trezit sistemul - modurile hardware eficiente nu produc economii dacă programul face verificări permanente sau ține perifericele active inutil. Figura 4.8 prezintă principalele direcții de optimizare software.

Execuția bazată pe evenimente. Un program bazat pe polling verifică repetat starea perifericelor, executând instrucțiuni fără activitate utilă:

Exemplu 4.1 Verificarea activă a unui eveniment

```
1 while ((STATUS_REGISTER & READY_MASK) == 0u)
2 {
3     /* asteptare activa */
4 }
```

O soluție bazată pe întreruperi permite procesorului să intre în repaus și să fie trezit la apariția evenimentului - dar nu e întotdeauna mai eficientă: dacă evenimentele sunt foarte

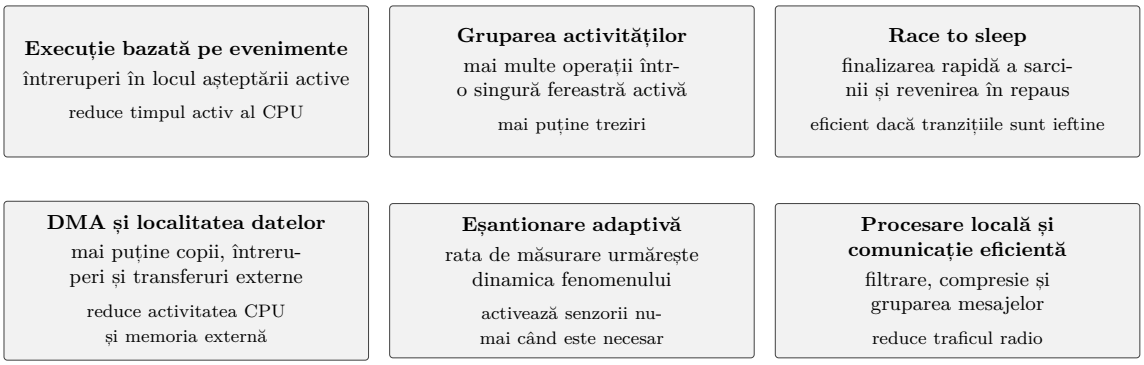


Figura 4.8 Direcții software pentru reducerea consumului energetic.

frecvente, costul salvării contextului și al tranzițiilor poate depăși avantajul repausului. Alegerea se face pe baza frecvenței evenimentelor și a latenței acceptate.

Reducerea numărului de treziri. Fiecare trezire implică pornirea ceasurilor, execuția codului și eventual activarea senzorilor sau comunicației, deci gruparea activităților (*batching* sau *wake-up coalescing*) reduce costurile fixe: în loc ca trei periferice să trezească procesorul separat, sistemul sincronizează măsurările și procesează datele într-o singură fereastră activă.

Race to sleep. Strategia presupune executarea rapidă a unei activități urmată de revenirea în repaus, avantajoasă atunci când consumul în sleep e mult mai mic decât cel activ, creșterea frecvenței nu cere o creștere excesivă a tensiunii, activitatea se termină mai repede, costul tranzițiilor e redus și există o pauză suficientă. Strategia opusă, *pace to idle*, folosește o frecvență redusă pentru limitarea puterii; niciuna nu e universal optimă - decizia se ia pe baza energiei măsurate pentru activitatea completă.

Reducerea accesărilor de memorie. Algoritmii trebuie să favorizeze reutilizarea datelor din registre, cache sau memoria locală - parcurgerea secvențială a unui vector e de regulă mai eficientă decât accesarea unor locații dispersate, iar structurile de date prea mari sau copierea inutilă a bufferelor cresc atât timpul, cât și energia. Tehnicile utile includ procesarea în blocuri, evitarea copiilor inutile, buffere circulare, tipuri de date potrivite, reutilizarea rezultatelor intermediare și alinierea datelor conform arhitecturii.

Utilizarea DMA. Controlerul DMA transferă date între periferic și memorie fără ca procesorul să gestioneze fiecare octet - de exemplu, un ADC poate stoca automat un bloc de eșantioane în memorie, procesorul fiind trezit doar după completarea bufferului, ceea ce reduce numărul întreruperilor și timpul activ al nucleului.

Eșantionarea adaptivă. Un senzor nu trebuie utilizat mereu la rata maximă: sistemul poate adapta frecvența de eșantionare la dinamica fenomenului, reducând rata când valorile se modifică lent și crescând-o temporar la o variație semnificativă. Aceasta reduce energia consumată de senzor, ADC, procesor, memorie și comunicație.

Procesarea locală și reducerea datelor. Filtrarea, compresia și extragerea caracteristicilor pot reduce cantitatea de informație transmisă. Un algoritm local consumă energie de calcul, dar poate evita activarea prelungită a transceiverului; decizia se evaluează comparând

$E_{\text{procesare}}$ cu $E_{\text{comunicație evitat}}$, procesarea locală fiind avantajoasă dacă

$$E_{\text{procesare}} < E_{\text{comunicație evitat}}. \tag{4.18}$$

Alegerea algoritmului. Algoritmul cu cel mai mic timp de execuție nu e întotdeauna cel mai eficient energetic - poate folosi mai puține operații aritmetice, dar mai multe accesări de memorie - astfel încât evaluarea trebuie făcută pe platforma țintă, cu date reprezentative. Pentru procesoare fără FPU, aritmetica fixed-point e adesea mai eficientă decât floating-point, diferența fiind mult mai mică pe un procesor cu FPU performantă. Optimizarea manuală nu trebuie aplicată înaintea profilării - compilatorul poate genera cod mai eficient decât o simplificare manuală aparentă.

Gestionarea perifericelor. Software-ul trebuie să dezactiveze explicit perifericele neutilizate: ceasurile modulelor, ADC-ul și referința sa, interfețele de comunicație, senzorii externi, LED-urile, interfața de depanare, pull-up/pull-down-urile inutile. Un pin lăsat flotant poate produce comutări și consum suplimentar, deci configurarea corectă a pinilor neutilizați contează în modurile de repaus.

Optimizarea comunicației. Software-ul poate reduce energia comunicațiilor prin gruparea mai multor valori într-un pachet, eliminarea datelor redundante, compresie, alegerea adaptivă a puterii de emisie, reducerea mesajelor de control, evitarea retransmisiilor, stocarea locală cu transmitere periodică și utilizarea ferestrelor de comunicație. Un pachet foarte mare are însă o probabilitate mai mare de eroare și poate necesita retransmiterea unei cantități mari de date - există deci o dimensiune optimă, dependentă de protocol și de calitatea legăturii. In tabelul 4.6 sunt sintetizat tehnicile de optimizare software.

4.11.1 Profilarea energetică a software-ului

Optimizarea trebuie bazată pe măsurători: profilarea temporală arată unde petrece procesorul timpul, dar nu întotdeauna unde e consumată energia. Un profil energetic corelează funcțiile executate, stările procesorului, activarea perifericelor, curentul măsurat și evenimentele de comunicație. Marcarea unor momente prin pini GPIO permite corelarea execuției software cu forma de undă a curentului măsurat la osciloscop.

Exemplu

Un nod de senzori se trezește la fiecare secundă și execută următoarele activități:

- pornirea senzorului;
- așteptarea stabilizării;
- citirea valorii;
- procesarea;
- transmiterea imediată a unui pachet.

Profilarea arată că procesorul finalizează procesarea în 2 ms, dar așteaptă activ 15 ms stabilizarea senzorului.

O versiune optimizată pornește senzorul, introduce procesorul în Sleep și utilizează un

Tabela 4.6 Tehnici software pentru optimizarea consumului energetic

Tehnică	Efect principal	Observație
Execuție pe bază de evenimente	Elimină așteptarea activă	Întreruperile frecvente pot introduce cost suplimentar
Gruparea activităților	Reduce numărul trezirilor	Poate introduce latență suplimentară
Race to sleep	Reduce durata stării active	Trebuie evaluat împreună cu tensiunea și tranzițiile
DMA	Reduce activitatea procesorului	Eficient pentru transferuri suficient de mari
Eșantionare adaptivă	Reduce activarea senzorilor și procesarea	Necesită o strategie corectă de detectare a evenimentelor
Procesare locală	Reduce traficul transmis	Consumă energie suplimentară pentru calcul
Localitatea datelor	Reduce transferurile către memorie	Depinde de arhitectura memoriei
Oprirea perifericelor	Reduce consumul static și dinamic	Necesită inițializare corectă la reactivare
Profilare energetică	Identifică optimizările relevante	Necesită instrumente și cicluri reprezentative

timer pentru trezire. Mai multe măsurători sunt grupate într-un singur pachet transmis la fiecare zece secunde.

Optimizarea reduce:

- timpul activ al procesorului;
- numărul pornirilor radio;
- numărul pachetelor;
- energia totală a ciclului.

4.12 Costul energetic al comunicațiilor wireless

Comunicațiile wireless permit sistemelor embedded să transmită informații fără o conexiune fizică, dar introduc un cost energetic care poate domina bugetul total al unui dispozitiv - situație frecventă în nodurile IoT, rețelele de senzori, dispozitivele wearable și sistemele industriale distribuite.

Un transceiver radio nu consumă energie doar la transmitere: funcționarea presupune activarea oscilatoarelor, sintetizarea frecvenței purtătoare, amplificarea semnalului, conversia analog-digitală/digital-analogică și procesarea protocolului. Recepția poate avea un consum comparabil cu transmisia, deoarece etajele radio și circuitele de procesare rămân active pe toată durata ascultării canalului [4, 70].

Energia totală asociată unei comunicații poate fi exprimată conceptual prin:

$$E_{\text{radio}} = E_{\text{pornire}} + E_{\text{ascultare}} + E_{\text{transmisie}} + E_{\text{recepție}} + E_{\text{protocol}} + E_{\text{retransmisii}}. \tag{4.19}$$

Relația arată că dimensiunea mesajului e doar unul dintre factorii care determină consumul: pentru mesaje scurte, energia fixă de pornire și sincronizare a transceiverului poate reprezenta o parte importantă din total.

Figura 4.9 prezintă un profil energetic simplificat al unui transceiver.

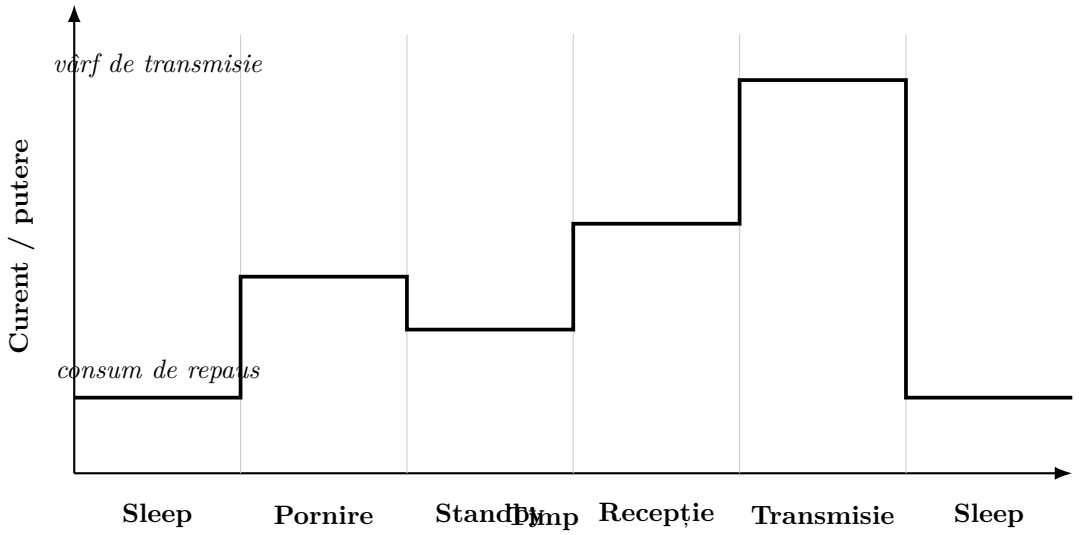


Figura 4.9 Profilul energetic simplificat al unui transceiver wireless.

Stările energetice ale unui transceiver. Majoritatea transceiverelor wireless funcționează în mai multe stări: *shutdown*, *sleep*, *standby*, recepție și transmisie. În shutdown sunt alimentate doar circuitele strict necesare reactivării, consumul e foarte redus, dar revenirea poate cere reinițializarea completă a modulului. În sleep sunt păstrate anumite registre/configurări - consum mai mare decât shutdown, dar revenire mai rapidă. În standby, oscilatoarele sau unele etaje radio sunt deja active, permițând tranziție rapidă spre recepție sau transmisie, dar menținerea îndelungată în această stare poate consuma semnificativ. În recepție, transceiverul amplifică, filtrează și decodează permanent semnalul, motiv pentru care ascultarea continuă a canalului poate fi mai costisitoare decât transmiterea periodică a unor pachete scurte. În transmisie, consumul depinde de puterea de emisie și eficiența amplificatorului - o putere mai mare îmbunătățește raza și robustețea legăturii, dar reduce autonomia.

4.12.1 Energia per bit și energia per informație utilă

Energia per bit transmis este definită prin:

$$E_{\text{bit}} = \frac{E_{\text{radio}}}{N_{\text{bit}}}. \tag{4.20}$$

Metrica e utilă pentru comparații, dar poate induce concluzii incomplete, deoarece un

pachet nu conține doar datele aplicației, ci și antete, adrese, informații de control, coduri de detecție a erorilor și secvențe de sincronizare. De aceea poate fi definită și energia per bit util:

$$E_{\text{bit,util}} = \frac{E_{\text{radio}}}{N_{\text{bit,util}}}. \tag{4.21}$$

Dacă pachetul conține puține date utile și un antet relativ mare, energia per bit util poate fi semnificativ mai ridicată decât cea calculată pe întregul cadru radio. O altă metrică relevantă este energia per pachet recepționat corect:

$$E_{\text{pachet,livrat}} = \frac{E_{\text{radio,total}}}{N_{\text{pachete,livrate}}}. \tag{4.22}$$

Aceasta include energia pierdută prin coliziuni, pachete corupte și retransmisii.

4.12.2 Influența dimensiunii pachetului

Gruparea mai multor măsurători într-un singur pachet reduce numărul pornirilor transceiverului și ponderea antetelor, dar un pachet foarte lung cere un timp mai mare de transmisie și e mai expus interferențelor. Energia unui pachet poate fi aproximată prin:

$$E_{\text{pachet}} = E_{\text{fix}} + P_{\text{TX}} \cdot T_{\text{pachet}}, \tag{4.23}$$

unde E_{fix} include pornirea, stabilizarea și procesarea protocolului. Pentru un debit radio R, timpul de transmisie este:

$$T_{\text{pachet}} = \frac{N_{\text{bit}}}{R}. \tag{4.24}$$

Reducerea numărului de pachete produce economii importante când energia fixă e ridicată; în condiții de canal nefavorabile însă, retransmiterea unui pachet foarte mare poate fi costisitoare.

4.12.3 Raza de comunicație și puterea de emisie

Puterea necesară transmisiei crește odată cu distanța, obstacolele și atenuarea mediului: în spațiu liber, puterea semnalului recepționat scade aproximativ cu pătratul distanței, iar în medii industriale, urbane sau interioare reflexiile și obstacolele produc o atenuare mai severă. Un model simplificat al energiei de transmisie:

$$E_{\text{TX}} = N_{\text{bit}} \left(E_{\text{electronic}} + E_{\text{amplificator}} d^n \right), \tag{4.25}$$

unde d e distanța, n exponentul de propagare, $E_{\text{electronic}}$ energia circuitelor radio și $E_{\text{amplificator}}$ energia etajului de emisie. Relația e un model conceptual - consumul real trebuie determinat din documentația transceiverului și prin măsurători. Creșterea puterii de emisie nu garantează întotdeauna reducerea energiei totale: o putere mai mare reduce numărul retransmisiilor, dar consumă mai mult per încercare; configurația optimă depinde de calitatea

legăturii și de cerințele aplicației.

Recepția și ascultarea canalului. Un nod care ascultă permanent canalul radio poate consuma mai multă energie decât unul care transmite periodic. Pentru reducerea acestui cost pot fi folosite ferestre programate de recepție, verificarea periodică a canalului, sincronizarea nodurilor, semnale dedicate de trezire, receptoare secundare cu consum foarte redus și protocoale de tip rendezvous - reducerea timpului de recepție poate introduce însă o latență mai mare, deoarece emițătorul trebuie să aștepte următoarea fereastră activă a receptorului.

4.12.4 Retransmisiile și calitatea legăturii

Interferențele, coliziunile și atenuarea pot conduce la pierderea pachetelor, iar energia consumată pentru o informație livrată cu succes crește odată cu numărul retransmisiilor. Dacă probabilitatea de livrare a unei încercări este p, numărul mediu idealizat de încercări este:

$$N_{\text{ncercri}} = \frac{1}{p}. \tag{4.26}$$

Energia medie necesară livrării unui pachet poate fi aproximată prin:

$$E_{\text{livrare}} = \frac{E_{\text{ncercare}}}{p}. \tag{4.27}$$

Pentru $p = 0{,}5$, sunt necesare în medie două încercări, iar energia se dublează față de o legătură fără pierderi.

4.12.5 Comparația tehnologiilor wireless

Tehnologiile wireless sunt optimizate pentru cerințe diferite - nu există o interfață care să ofere simultan raza maximă, debitul maxim, latența minimă și consumul minim [72].

Raza, debitul și consumul nu sunt valori fixe - depind de puterea de emisie, rata de date, codare, antenă, mediul de propagare și parametrii protocolului. Figura 4.10 ilustrează compromisurile generale dintre rază, debit, latență și consum.

4.12.6 Exemplu de profil energetic radio

Exemplu

Un nod wireless repetă următorul ciclu la fiecare 60 de secunde:

- pornire radio: 5 mA timp de 5 ms;
- recepție: 12 mA timp de 100 ms;
- transmisie: 30 mA timp de 20 ms;
- sleep: 2 μA în restul perioadei.

Durata stării sleep este:

$$t_{\text{sleep}} = 60 - 0{,}005 - 0{,}100 - 0{,}020 = 59{,}875\,\text{s}.$$

Tabela 4.7 Caracteristici generale ale unor tehnologii wireless

Tehnologie	Caracteristică principală	Avantaj energetic	Limitare
Bluetooth Low Energy	Comunicații locale și trafic redus sau periodic	Moduri sleep eficiente și pachete scurte	Rază și debit dependente de configurație
Wi-Fi	Debit ridicat și integrare IP	Transfer rapid al unor cantități mari de date	Pornire, asociere și recepție cu consum relativ ridicat
IEEE 802.15.4	Rețele de senzori și comunicație locală	Transceivere simple și consum redus	Debit și dimensiune a pachetului limitate
LoRa/ LoRa-WAN	Distanțe mari și trafic redus	Comunicații sporadice la putere moderată	Timp de transmisie mare la rate reduse
NB-IoT/ LTE-M	Acoperire celulară și infrastructură existentă	Conectivitate pe distanțe mari	Înregistrarea în rețea și semnalizarea pot consuma mult

Tehnologie	**Rază**	**Debit**	**Consum**	**Latență**
Bluetooth Low Energy	■■□□□	■■□□□	■□□□□	■■□□□
Wi-Fi	■■□□□	■■■■■	■■■■■	■□□□□
IEEE 802.15.4	■■□□□	■■□□□	■■□□□	■■■□□
LoRa / LoRaWAN	■■■■■	■□□□□	■■□□□	■■■■■
NB-IoT / LTE-M	■■■■■	■■□□□	■■■□□	■■■■□

Figura 4.10 Compromisuri generale în alegerea unei tehnologii wireless.

Sarcina consumată într-un ciclu este:

$$Q_{\text{ciclu}} = 5 \cdot 0{,}005 + 12 \cdot 0{,}100 + 30 \cdot 0{,}020 + 0{,}002 \cdot 59{,}875.$$

Rezultă:

$$Q_{\text{ciclu}} \approx 1{,}945\,\text{mA s}.$$

Curentul mediu este:

$$I_{\text{mediu}} = \frac{1{,}945}{60} \approx 0{,}0324\,\text{mA} = 32{,}4\,\mu\text{A}.$$

Deși curentul de transmisie este de 30 mA, valoarea medie este redusă datorită duratei scurte de activare.

4.13 Procesare locală versus transmiterea datelor

Un sistem embedded poate transmite datele brute către un server sau le poate procesa local și transmite numai informația relevantă. Alegerea influențează consumul, latența, confidențialitatea și necesarul de calcul.

Energia primei strategii este:

$$E_{\text{brut}} = E_{\text{achiziție}} + E_{\text{transmitere brut}}. \tag{4.28}$$

Energia procesării locale este:

$$E_{\text{local}} = E_{\text{achiziție}} + E_{\text{procesare}} + E_{\text{transmitere rezultat}}. \tag{4.29}$$

Procesarea locală este avantajoasă energetic atunci când:

$$E_{\text{procesare}} + E_{\text{transmitere rezultat}} < E_{\text{transmitere brut}}. \tag{4.30}$$

Figura 4.11 compară cele două abordări.

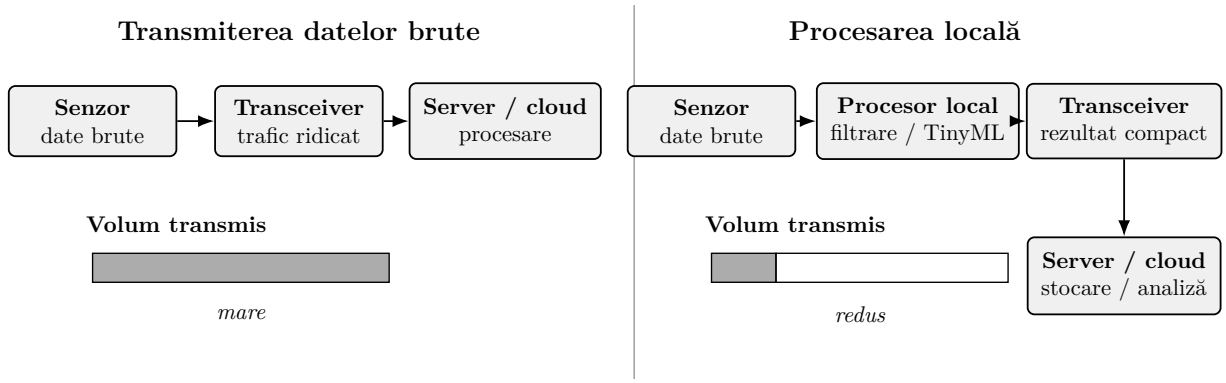


Figura 4.11 Procesarea locală în comparație cu transmiterea datelor brute.

Reducerea volumului de date. Procesarea locală poate include filtrare, eliminarea valorilor redundante, compresie, extragerea caracteristicilor, detecția evenimentelor, clasificare sau inferență TinyML. Un senzor de vibrații poate genera mii de eșantioane pe secundă - transmiterea permanentă cere activarea îndelungată a radio-ului, în timp ce un algoritm local poate calcula indicatori statistici și transmite doar la detectarea unei anomalii.

Costurile procesării locale. Procesarea locală necesită un procesor, memorie și eventual un accelerator hardware; algoritmii complecși pot crește timpul activ, iar modelele de inteligență artificială pot cere memorie suplimentară. Nu e întotdeauna optimă - trans-

miterea poate fi preferabilă dacă datele sunt deja foarte compacte, legătura e permanent disponibilă și eficientă, algoritmul necesită resurse mari, modelul trebuie actualizat foarte frecvent, serverul trebuie să păstreze informația brută, sau consumul radio e redus comparativ cu procesarea.

Latență, confidențialitate și robustețe. Procesarea locală poate reduce latența, deoarece decizia nu mai depinde de rețea, și poate îmbunătăți confidențialitatea, deoarece semnalul brut nu părăsește dispozitivul; un sistem local poate continua să funcționeze și în absența conexiunii, dar actualizarea algoritmului și colectarea datelor pentru diagnostic pot fi mai dificile. Decizia trebuie luată pe baza întregului sistem, nu doar a consumului procesorului sau radio-ului.

4.13.1 Exemplu comparativ

Exemplu

Un senzor produce 1000 de eșantioane pe secundă, fiecare reprezentat pe 16 biți:

$$R_{\text{brut}} = 1000 \times 16 = 16\,000\,\text{bit/s}.$$

Sistemul poate transmite permanent fluxul sau poate executa local un algoritm care produce un rezultat de 32 de octeți la fiecare minut.
Numărul de biți transmiși într-un minut în prima variantă este:

$$N_{\text{brut}} = 16\,000 \times 60 = 960\,000\,\text{bit}.$$

În varianta locală:

$$N_{\text{rezultat}} = 32 \times 8 = 256\,\text{bit}.$$

Raportul dintre volumele transmise este:

$$\frac{N_{\text{brut}}}{N_{\text{rezultat}}} = 3750.$$

Dacă energia procesării locale este mai mică decât energia necesară transmiterii celor 959 744 de biți evitați, procesarea locală reduce consumul total.

4.14 Surse de energie pentru sistemele embedded

Sursa de energie influențează autonomia, dimensiunea, costul, fiabilitatea și condițiile de funcționare ale unui sistem embedded. Principalele categorii sunt alimentarea de la rețeaua electrică, sursele industriale de curent continuu, bateriile primare, bateriile reîncărcabile, supercondensatoarele, pilele de combustie și sistemele de recuperare a energiei. Figura 4.12 prezintă principalele opțiuni de alimentare.

Alimentarea de la rețea sau de la o sursă externă. Echipamentele fixe pot fi alimentate de la rețeaua electrică printr-o sursă AC–DC sau dintr-o magistrală industrială

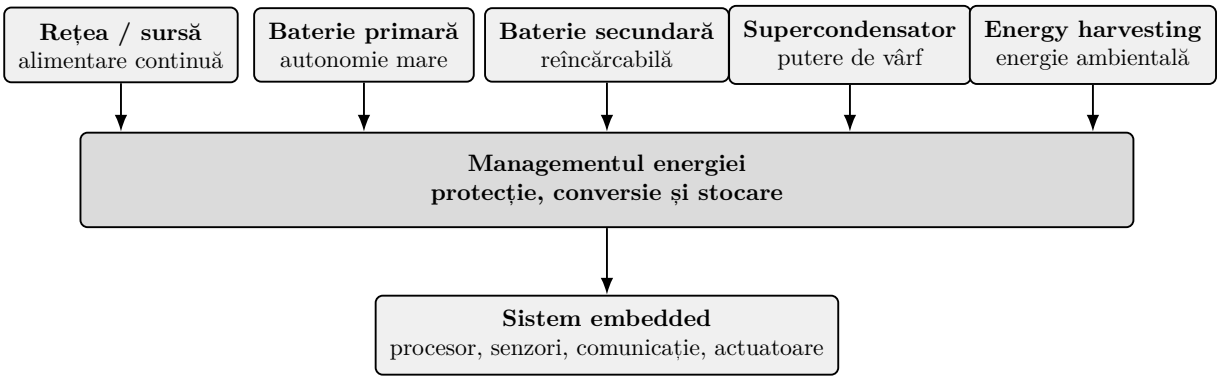


Figura 4.12 Surse de energie utilizate în sistemele embedded.

de curent continuu - soluție cu energie disponibilă pe termen lung, dar cu dependență de infrastructură, ceea ce poate cere protecții la supratensiune, izolație galvanică, filtrare electromagnetică și surse de rezervă. În aplicații critice, o baterie sau un supercondensator poate asigura funcționarea temporară la întreruperea alimentării.

Bateriile primare nu sunt proiectate pentru reîncărcare și sunt potrivite pentru dispozitive cu consum redus și durată mare de funcționare; avantajele includ autodescărcare redusă, durată mare de stocare, integrare simplă și cost redus, iar dezavantajele sunt necesitatea înlocuirii și impactul asupra mediului.

Bateriile reîncărcabile (secundare) permit cicluri repetate de încărcare și descărcare și sunt utilizate în dispozitive mobile, roboți, drone și sisteme conectate periodic la o sursă; integrarea necesită un circuit de încărcare și, pentru anumite tehnologii, un sistem de protecție și management.

Supercondensatoarele stochează energie electrostatic, oferind putere specifică ridicată, curenți mari de încărcare/descărcare, un număr foarte mare de cicluri și încărcare rapidă. Energia stocată este

$$E_C = \frac{1}{2} C V^2. \tag{4.31}$$

Tensiunea scade continuu în timpul descărcării, iar densitatea energetică e mai mică decât a bateriilor, motiv pentru care e adesea necesar un convertor DC–DC. Sunt potrivite pentru susținerea impulsurilor de curent, recuperarea energiei și funcționarea de scurtă durată.

4.14.1 Alegerea sursei

Alegerea trebuie să considere:

- energia totală necesară;
- puterea maximă;
- durata de funcționare;
- temperatura;
- dimensiunea și masa;
- numărul de cicluri;

- mentenanța;
- siguranța;
- costul.

Tabela 4.8 Comparația generală a surselor de energie

Sursă	Avantaj principal	Limitare principală	Utilizare reprezentativă
Rețea / sursă externă	Energie disponibilă continuu	Lipsa autonomiei	Echipamente fixe și industriale
Baterie primară	Stocare îndelungată și integrare simplă	Nu poate fi reîncărcată	Senzori cu mentenanță rară
Baterie secundară	Reutilizare prin reîncărcare	Îmbătrânire și necesitatea încărcătorului	Dispozitive mobile și roboți
Super condensator	Putere mare și număr ridicat de cicluri	Energie stocată relativ redusă	Impulsuri, backup și harvesting
Energy harvesting	Posibilitatea funcționării autonome	Putere variabilă și dependentă de mediu	Noduri de senzori și sisteme distribuite

4.15 Recuperarea energiei din mediul înconjurător

Recuperarea energiei, denumită *energy harvesting*, constă în captarea unei cantități de energie disponibile în mediul înconjurător și transformarea ei în energie electrică [64, 83]. Sursele posibile includ radiația solară, lumina artificială, diferențele de temperatură, vibrațiile mecanice, mișcarea, fluxurile de aer sau apă, câmpurile electromagnetice și semnalele radio. Energia disponibilă e de regulă redusă și variabilă, așa că sistemul trebuie să combine captarea cu stocarea și managementul energetic. Figura 4.13 prezintă lanțul funcțional al unui sistem de recuperare a energiei.

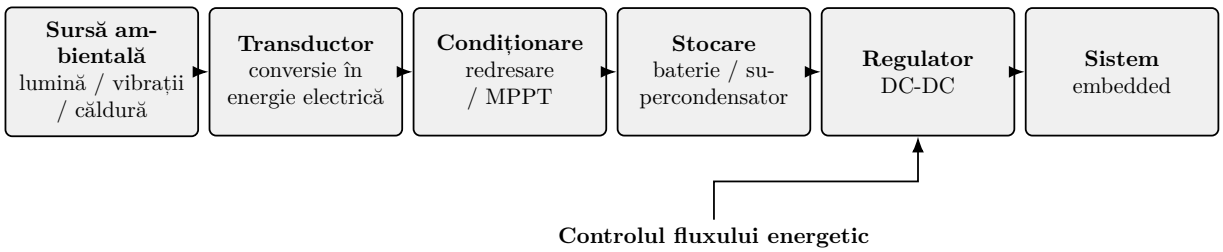


Figura 4.13 Lanțul funcțional al unui sistem de recuperare a energiei.

4.15.1 Surse de recuperare a energiei din mediu

Conversia fotovoltaică. Celulele fotovoltaice transformă energia luminoasă în energie electrică, puterea depinzând de iluminare, suprafață, orientare, spectrul sursei și randamentul celulei. În exterior, energia solară poate susține noduri care transmit periodic; în interior,

nivelul de iluminare e mult mai redus, iar celulele trebuie optimizate pentru spectrul surselor artificiale. Un convertor cu urmărirea punctului de putere maximă poate adapta sarcina astfel încât panoul să funcționeze în regiunea optimă.

Recuperarea energiei termice. Generatoarele termoelectrice produc o tensiune la o diferență de temperatură între cele două fețe și pot fi utilizate pe conducte, motoare, corpul uman sau echipamente industriale, puterea scăzând pe măsură ce temperaturile se egalizează. Avantajul constă în lipsa pieselor mobile, dezavantajul principal fiind randamentul redus pentru diferențe mici de temperatură.

Recuperarea energiei din vibrații. Vibrațiile pot fi convertite prin mecanisme piezoelectrice, electromagnetice sau electrostatice, puterea obținută depinzând de amplitudine, frecvență și acordarea mecanică a recuperatorului - un dispozitiv rezonant poate fi eficient într-un interval îngust, dar mai puțin eficient dacă frecvența vibrațiilor variază. Aplicațiile includ monitorizarea utilajelor, transportul și infrastructura.

Recuperarea energiei radio. Energia electromagnetică poate fi captată printr-o antenă și convertită în curent continuu printr-un circuit de redresare. În absența unui emițător dedicat, densitatea de putere ambientală e de regulă redusă, deci recuperarea RF e potrivită pentru dispozitive cu consum extrem de mic sau pentru sisteme alimentate intenționat de un cititor, cum sunt anumite etichete RFID.

Condiția de neutralitate energetică. Un sistem poate funcționa sustenabil dacă energia captată pe termen lung e cel puțin egală cu energia consumată și pierderile asociate stocării:

$$\int_{t_0}^{t_1} \eta_{\text{harv}} P_{\text{harv}}(t)\, dt \geq \int_{t_0}^{t_1} P_{\text{sistem}}(t)\, dt + E_{\text{pierderi}}. \tag{4.32}$$

Condiția trebuie verificată pe un interval reprezentativ: un sistem solar poate avea surplus în timpul zilei, dar trebuie să stocheze suficientă energie pentru noapte și pentru perioadele cu iluminare redusă.

Stocarea energiei recuperate. Energia poate fi stocată într-o baterie reîncărcabilă, într-un supercondensator sau într-o combinație a celor două - bateria oferă energie ridicată, supercondensatorul susține impulsurile de curent, iar o arhitectură hibridă poate reduce solicitarea bateriei. Circuitul de management trebuie să funcționeze cu un curent propriu foarte redus: într-un sistem de ordinul microwaților, consumul convertorului poate reprezenta o parte importantă din energia captată.

4.16 Baterii electrochimice

O baterie electrochimică transformă energia chimică în energie electrică prin reacții de oxidare și reducere. O celulă conține electrod negativ, electrod pozitiv, electrolit, separator, colectoare de curent și carcasă; în timpul descărcării, electronii circulă prin circuitul exterior, iar ionii se deplasează prin electrolit. Figura 4.14 prezintă structura conceptuală a unei celule.

Celulă, baterie și pachet. O celulă reprezintă unitatea electrochimică elementară; o baterie poate conține una sau mai multe celule conectate în serie sau în paralel. Conectarea

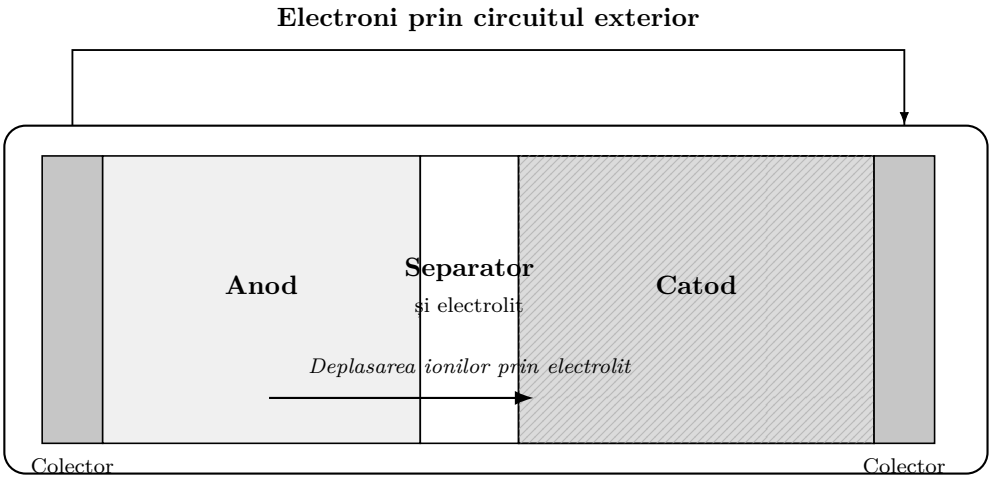


Figura 4.14 Structura conceptuală a unei celule electrochimice.

în serie crește tensiunea:

$$V_{\text{pachet}} = N_s V_{\text{celul}}. \tag{4.33}$$

Conectarea în paralel crește capacitatea și curentul disponibil:

$$Q_{\text{pachet}} = N_p Q_{\text{celul}}. \tag{4.34}$$

Un pachet poate include și senzori, siguranțe, circuite de protecție și un sistem de management.

Tensiunea unei celule nu e constantă - depinde de chimie, starea de încărcare, curent, temperatură și îmbătrânire. Tensiunea nominală e o valoare convențională utilizată pentru descrierea și estimarea energiei; proiectarea trebuie să considere tensiunea maximă, minimă și curba de descărcare.

Capacitatea și energia. Capacitatea nominală e exprimată în Ah sau mAh și descrie sarcina electrică disponibilă în condițiile de testare specificate. Energia nominală poate fi aproximată prin:

$$E_{\text{nominal}} \approx V_{\text{nominal}} Q_{\text{nominal}}. \tag{4.35}$$

Rezultatul e exprimat în Wh dacă tensiunea e în V, iar capacitatea în Ah. Energia efectiv utilizabilă e mai mică din cauza pragului minim de tensiune, rezistenței interne, temperaturii și eficienței convertorului.

4.16.1 Rata C

Rata C exprimă curentul raportat la capacitatea nominală. Pentru o baterie cu capacitatea Q_{nominal}, curentul corespunzător ratei C_r este:

$$I = C_r Q_{\text{nominal}}. \tag{4.36}$$

O baterie de 2 Ah descărcată la $0{,}5C$ furnizează:

$$I = 0{,}5 \times 2 = 1\,\text{A}.$$

Descărcarea la o rată mai mare poate reduce capacitatea utilizabilă și crește încălzirea.

4.16.2 Rezistența internă

Un model simplu reprezintă bateria printr-o sursă ideală de tensiune și o rezistență internă. Tensiunea la borne în timpul descărcării este:

$$V_{\text{terminal}} = V_{\text{OC}} - I R_{\text{intern}}. \tag{4.37}$$

Puterea disipată intern este:

$$P_{\text{pierdere}} = I^2 R_{\text{intern}}. \tag{4.38}$$

Impulsurile mari de curent pot produce o cădere de tensiune care resetează sistemul, chiar dacă bateria mai conține energie - problemă frecventă pentru celulele tip monedă care alimentează transceivere radio. Un condensator poate susține impulsul, dar soluția trebuie verificată pentru întregul profil de funcționare.

Autodescărcarea. O baterie pierde treptat sarcină chiar dacă nu alimentează un consumator; pentru dispozitivele cu o autonomie de mai mulți ani, autodescărcarea poate deveni comparabilă cu energia consumată de circuit, deci estimarea autonomiei trebuie să includă și pierderea proprie a bateriei.

Durata de viață calendaristică și cicluri. Durata de viață calendaristică descrie degradarea produsă de timp, temperatură și starea de încărcare; durata de viață în cicluri descrie numărul de cicluri înainte ca performanța să scadă sub un prag - un ciclu nu trebuie să fie o descărcare completă, mai multe descărcări parțiale putând forma echivalentul unui ciclu complet.

Densitatea energetică și densitatea de putere. Densitatea energetică descrie energia raportată la masă sau volum:

$$\rho_E = \frac{E}{m} \quad \text{sau} \quad \frac{E}{V_{\text{volum}}}. \tag{4.39}$$

Densitatea de putere descrie capacitatea de a furniza energie rapid:

$$\rho_P = \frac{P}{m}. \tag{4.40}$$

O baterie poate avea o energie specifică ridicată, dar să nu poată furniza impulsuri mari - alegerea trebuie să considere ambele caracteristici.

Siguranța și managementul bateriei. Bateriile reîncărcabile necesită respectarea

limitelor de tensiune, curent și temperatură. Un sistem de management al bateriei poate asigura protecție la supraîncărcare, la descărcare excesivă și la supracurent, monitorizarea temperaturii, estimarea stării de încărcare, echilibrarea celulelor și comunicarea cu sistemul principal - complexitatea depinde de chimie, numărul de celule și aplicație.

4.17 Tehnologii moderne de baterii

Tehnologiile de baterii oferă compromisuri diferite între energie, putere, durată de viață, temperatură, siguranță și cost [33, 73, 86].

4.17.1 Tipuri de baterii primare și reîncărcabile

Bateriile alcaline sunt baterii primare, utilizate în dispozitive cu consum redus sau moderat, cu tensiune nominală în jurul a 1,5 V, cost redus, disponibilitate ridicată și durată bună de stocare, dar performanță mai slabă la curenți mari și temperaturi scăzute; sunt potrivite pentru telecomenzi, senzori simpli și electronice de consum.

Bateriile primare cu litiu oferă o densitate energetică și o durată de stocare mai ridicate decât cele alcaline. Celulele tip monedă litiu–dioxid de mangan furnizează frecvent circa 3 V și sunt utilizate în ceasuri, memorii de backup și senzori; chimia litiu–clorură de tionil oferă o tensiune nominală de circa 3,6 V, autodescărcare redusă și autonomie foarte mare, dar comportamentul la impulsuri și pasivarea trebuie analizate cu atenție.

Bateriile NiMH (nichel–metal hidrură) sunt reîncărcabile, cu tensiune nominală de circa 1,2 V; avantajele includ robustețea și toleranța la utilizare, dezavantajele autodescărcarea mai ridicată și densitatea energetică mai redusă decât Li-Ion. Sunt utilizate în echipamente portabile, dispozitive medicale și aplicații cu formatul standard AA/AAA.

Familia Li-Ion descrie chimiile bazate pe migrarea ionilor de litiu între electrozi [60]. Chimiile pe bază de oxizi de cobalt, nichel și mangan oferă densitate energetică ridicată și sunt utilizate în electronice portabile, drone și vehicule, cu tensiune nominală frecvent 3,6–3,7 V (limitele exacte depind de chimie). Celulele necesită control atent al încărcării și protecție la supratensiune, subtensiune, supracurent și temperatură excesivă.

Litiu–fier–fosfat ($LiFePO_4$, LFP) oferă stabilitate termică bună, durată mare de viață în cicluri, capacitate bună de furnizare a puterii și tensiune nominală de circa 3,2 V, cu densitate energetică mai redusă decât NMC sau NCA; e utilizată în sisteme industriale, stocare staționară, vehicule și aplicații unde siguranța și durata de viață contează.

Litiu–titanat (LTO) folosește titanat de litiu la electrodul negativ, oferind încărcare rapidă, putere mare și număr ridicat de cicluri, cu tensiune nominală și densitate energetică mai reduse decât celulele Li-Ion convenționale - potrivită unde durata de viață, temperatura și încărcarea rapidă contează mai mult decât masa.

Bateriile Li-Polymer sunt de regulă celule Li-Ion într-o carcasă flexibilă de tip pouch, cu electrolit sub formă de gel sau polimer - nu reprezintă automat o chimie distinctă, proprietățile depinzând de materialele electrozilor. Formatul pouch permite forme subțiri și utilizare eficientă a volumului, dar cere protecție mecanică și controlul umflării.

Bateriile cu electrolit solid urmăresc înlocuirea electrolitului lichid cu un material

solid, cu avantaje potențiale de siguranță și densitate energetică ridicată, dar provocări legate de rezistența interfețelor, fabricație, cost și comportament pe termen lung - tehnologie în evoluție continuă, nu o înlocuire universală imediată a bateriilor Li-Ion.

4.17.2 Comparația tehnologiilor

Tabela 4.9 Comparația generală a unor tehnologii de baterii

Tehnologie	Tensiune nominală	Avantaj principal	Limitare principală
Alcalină	1,5 V	Cost redus și disponibilitate	Nereîncărcabilă și performanță redusă la impulsuri mari
Litiu primar tip monedă	Aproximativ 3 V	Autodescărcare redusă și dimensiune mică	Curent continuu și impulsuri limitate
Li-SOCl$_2$	Aproximativ 3,6 V	Energie și durată de stocare ridicate	Pasivare și cerințe speciale pentru impulsuri
NiMH	Aproximativ 1,2 V	Reîncărcabilă și robustă	Autodescărcare și energie specifică moderate
Li-Ion NM-C/NCA/LCO	3,6–3,7 V	Densitate energetică ridicată	Necesită protecție și management atent
LiFePO$_4$	Aproximativ 3,2 V	Siguranță și durată mare de viață	Densitate energetică mai redusă
LTO	Aproximativ 2,3 V	Încărcare rapidă și foarte multe cicluri	Tensiune și energie specifică reduse

Criterii de alegere. Pentru un sistem embedded trebuie analizate tensiunea minimă și maximă, energia necesară, curentul continuu și impulsurile, temperatura, durata calendaristică, numărul de cicluri, autodescărcarea, masa și volumul, siguranța, costul, disponibilitatea și mentenanța. Nu e suficientă alegerea bateriei cu cea mai mare capacitate în mAh - tensiunea, energia în Wh, rezistența internă și comportamentul la sarcina reală pot determina o soluție diferită.

4.18 Parametrii și comportamentul real al bateriilor

Capacitatea și tensiunea nominale nu sunt suficiente pentru a descrie comportamentul unei baterii într-un sistem embedded real - energia efectiv disponibilă depinde de profilul curentului, temperatura de funcționare, starea de încărcare, rezistența internă, pragul minim acceptat de circuit și gradul de îmbătrânire. Două baterii cu aceeași capacitate în Ah pot produce autonomii diferite dacă au tensiuni, rezistențe interne sau caracteristici de descărcare diferite, iar o parte din energia electrochimică poate rămâne neutilizată dacă tensiunea la borne scade sub pragul minim al regulatorului sau microcontrolerului. Figura 4.15 prezintă principalele elemente care determină energia efectiv disponibilă.

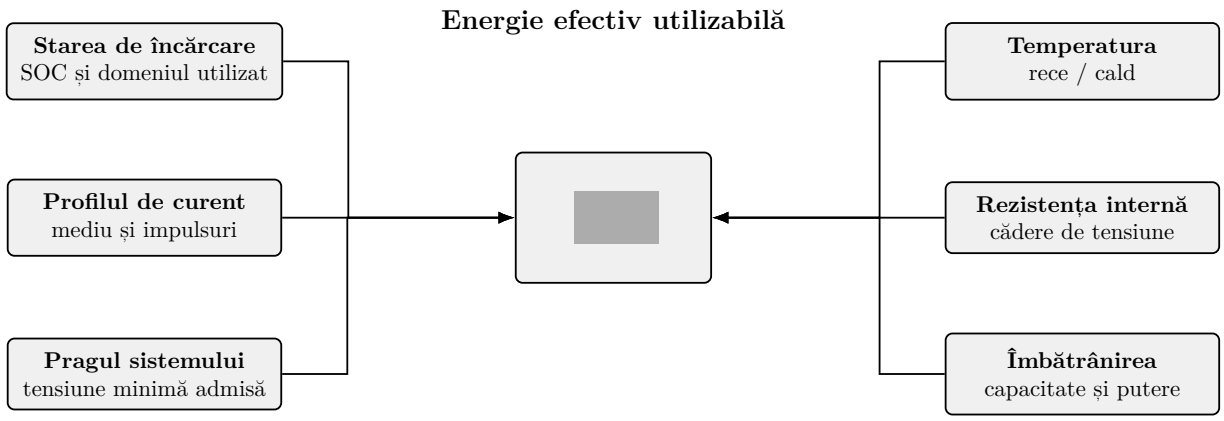


Figura 4.15 Factorii care determină comportamentul real al unei baterii.

4.18.1 Starea de încărcare

Starea de încărcare (*State of Charge*, SOC) exprimă proporția de sarcină disponibilă la un moment dat.

$$\mathrm{SOC} = \frac{Q_{\mathrm{rmas}}}{Q_{\mathrm{maxim}}} \cdot 100\%. \tag{4.41}$$

O baterie complet încărcată are, ideal, SOC = 100%, iar una complet descărcată SOC = 0%. În practică, sistemul nu utilizează întotdeauna întregul domeniu - pentru protejarea bateriei sau respectarea pragului minim de tensiune, funcționarea poate fi limitată, de exemplu, la 10%–90% SOC. Adâncimea de descărcare (*Depth of Discharge*, DOD) este:

$$\mathrm{DOD} = 100\% - \mathrm{SOC}. \tag{4.42}$$

O descărcare de la 100% la 30% SOC corespunde unei adâncimi de descărcare de 70%.

4.18.2 Estimarea SOC prin numărarea sarcinii

O metodă frecventă pentru estimarea SOC este integrarea curentului (*coulomb counting*). Considerând curentul pozitiv în timpul descărcării:

$$\mathrm{SOC}(t) = \mathrm{SOC}(t_0) - \frac{1}{Q_{\max}} \int_{t_0}^{t} \eta_I I(\tau)\, d\tau, \tag{4.43}$$

unde $Q_{\max}$ e capacitatea efectivă a bateriei, $I(t)$ curentul bateriei și η_I randamentul de încărcare/descărcare. Dacă $Q_{\max}$ e exprimată în Ah, timpul trebuie exprimat în ore; dacă e în coulombi, timpul e în secunde.

Metoda e simplă și poate urmări variațiile rapide ale SOC, dar are două limitări: acumularea erorilor de măsurare și necesitatea cunoașterii stării inițiale - o eroare mică a senzorului de curent, integrată zile la rând, poate produce o eroare importantă a SOC. Corectarea periodică poate folosi tensiunea în circuit deschis, identificarea unei stări complet încărcate sau algoritmi de estimare bazați pe modele [67].

Tensiunea în circuit deschis (V_{OC}) este tensiunea măsurată după ce bateria a rămas suficient timp fără sarcină, ajungând într-o stare apropiată de echilibrul electrochimic. Există o relație între V_{OC} și SOC:

$$V_{\mathrm{OC}} = f(\mathrm{SOC}, T, \mathrm{SOH}), \tag{4.44}$$

unde T e temperatura, iar SOH starea de sănătate a bateriei. Relația nu e identică pentru toate tehnologiile: pentru unele chimii, curba tensiunii e suficient de înclinată pentru a permite o estimare aproximativă a SOC, dar pentru altele, precum $\mathrm{LiFePO_4}$, tensiunea rămâne aproape constantă într-o parte mare a descărcării, unde o diferență mică de tensiune poate corespunde unei variații mari a SOC. Tensiunea măsurată sub sarcină nu trebuie confundată cu tensiunea în circuit deschis - căderea pe rezistența internă și efectele dinamice modifică valoarea observată.

4.18.3 Starea de sănătate

Starea de sănătate (*State of Health*, SOH) descrie degradarea bateriei față de starea inițială. O definiție bazată pe capacitate este:

$$\mathrm{SOH}_Q = \frac{Q_{\mathrm{max,actual}}}{Q_{\mathrm{nominal,inițial}}} \cdot 100\%. \tag{4.45}$$

Dacă o baterie evaluată inițial la 2 Ah mai poate furniza numai 1,6 Ah, rezultă:

$$\mathrm{SOH}_Q = \frac{1{,}6}{2} \cdot 100\% = 80\%.$$

SOH poate fi evaluată și prin creșterea rezistenței interne - o baterie poate păstra o parte importantă din capacitate, dar poate deveni inutilizabilă într-o aplicație cu impulsuri mari de curent. Starea de sănătate trebuie deci raportată la cerințele aplicației: capacitatea disponibilă, rezistența internă, puterea maximă, autodescărcarea și siguranța.

4.18.4 Energia efectiv utilizabilă

Energia extrasă din baterie până la oprirea sistemului este:

$$E_{\mathrm{util}} = \int_{t_0}^{t_{\mathrm{oprire}}} V_{\mathrm{terminal}}(t) I(t)\, dt. \tag{4.46}$$

Momentul t_{oprire} poate fi determinat de atingerea tensiunii minime a microcontrolerului, pragul de protecție al bateriei, oprirea convertorului DC–DC, incapacitatea de a furniza un impuls de curent sau limita minimă de SOC impusă de aplicație. O estimare inginerească poate utiliza:

$$E_{\mathrm{util}} \approx E_{\mathrm{nominal}} \cdot k_{\mathrm{SOC}} \cdot k_T \cdot k_{\mathrm{rat}} \cdot k_{\mathrm{mbtrnire}} \cdot \eta_{\mathrm{conv}}, \tag{4.47}$$

unde fiecare coeficient reprezintă o fracție subunitară; relația nu înlocuiește modelarea sau testarea, dar evidențiază necesitatea aplicării unor factori de corecție.

Curba de descărcare reprezintă variația tensiunii în funcție de timpul de funcționare, sarcina extrasă sau SOC. Figura 4.16 prezintă o curbă generică de descărcare.

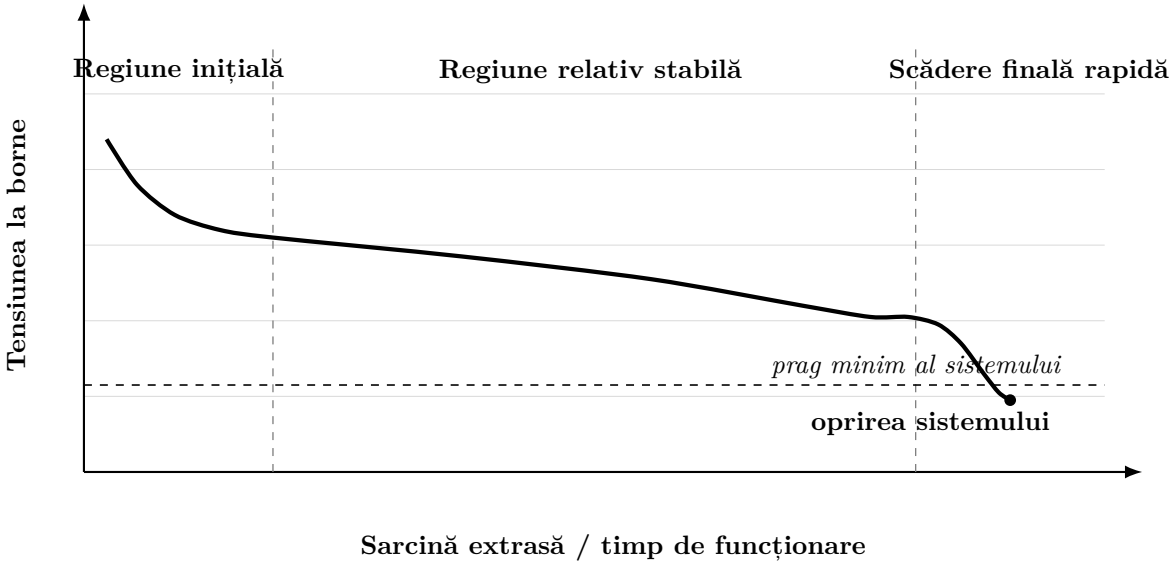


Figura 4.16 Curba generică de descărcare a unei baterii.

Pot fi identificate trei regiuni generale: o variație inițială după conectarea sarcinii, o regiune de funcționare relativ stabilă și o scădere rapidă a tensiunii spre sfârșitul descărcării. Forma exactă depinde de chimie - unele baterii au o curbă relativ plată, altele o scădere progresivă - deci pragul de oprire trebuie ales în funcție de caracteristica reală a bateriei, nu doar de tensiunea nominală.

4.18.5 Impulsurile de curent

Sistemele embedded moderne au frecvent un consum pulsatoriu: nodul poate petrece aproape întregul timp în sleep și poate solicita un curent de sute sau mii de ori mai mare în timpul transmiterii radio. Căderea instantanee de tensiune poate fi aproximată prin:

$$\Delta V = I_{\text{impuls}} R_{\text{intern}}. \tag{4.48}$$

Dacă tensiunea ajunge sub pragul minim, sistemul se poate reseta chiar dacă SOC e încă ridicat. Un condensator plasat în apropierea consumatorului poate furniza o parte din energia impulsului. Într-un model simplificat:

$$C \geq \frac{I_{\text{impuls}} \Delta t}{\Delta V_{\text{acceptat}}}. \tag{4.49}$$

Relația presupune că întregul curent e furnizat de condensator; în realitate, bateria și condensatorul contribuie simultan, iar rezistențele și impedanțele traseelor trebuie incluse în analiză.

4.19 Modelarea electrică a bateriilor

Modelele de baterie permit simularea tensiunii, estimarea autonomiei, proiectarea regulatorului și dezvoltarea algoritmilor pentru SOC și SOH. Un model trebuie ales în funcție de scop: unul simplu poate fi suficient pentru dimensionarea preliminară, în timp ce estimarea precisă a SOC necesită un model dinamic și identificarea parametrilor. Principalele categorii sunt modelele electrice echivalente, electrochimice, empirice și bazate pe date; cele electrochimice descriu procesele fizice în detaliu, dar au cost de calcul ridicat, motiv pentru care în sistemele embedded și algoritmii BMS sunt folosite frecvent modelele electrice echivalente [23, 40].

Modelul sursei ideale. Cel mai simplu model consideră bateria o sursă ideală de tensiune:

$$V_{\text{terminal}} = V_{\text{ideal}}. \tag{4.50}$$

Acest model nu include variația tensiunii cu SOC, rezistența internă, efectele dinamice, temperatura sau îmbătrânirea, deci poate fi utilizat doar pentru analize preliminare în care tensiunea e aproximată ca fiind constantă.

4.19.1 Modelul cu rezistență internă

Modelul Rint adaugă o rezistență serie R_0 la sursa de tensiune în circuit deschis. Pentru curent pozitiv la descărcare:

$$V_{\text{terminal}} = V_{\text{OC}}(\text{SOC}) - IR_0. \tag{4.51}$$

Figura 4.17 prezintă modelul Rint și modelul Thevenin de ordinul întâi.

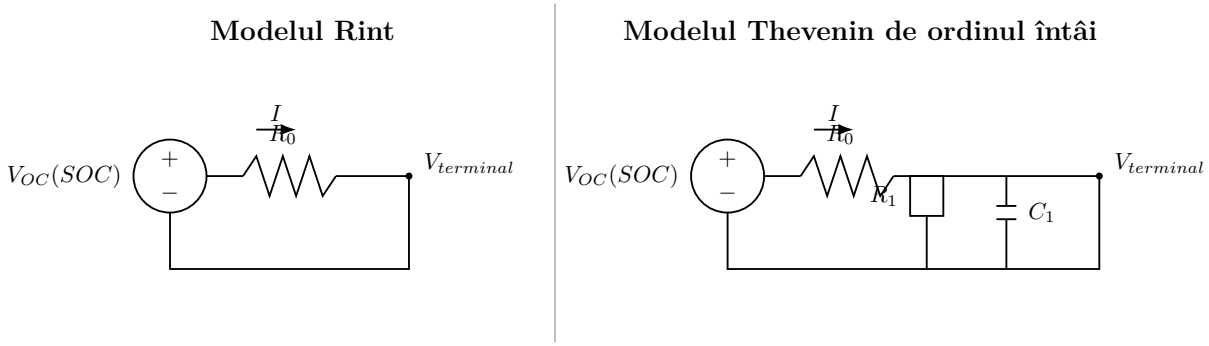


Figura 4.17 Modele electrice echivalente pentru o baterie.

Modelul Rint descrie căderea instantanee de tensiune, dar nu reproduce revenirea lentă observată după eliminarea sarcinii. Rezistența internă nu e constantă - depinde de:

$$R_0 = f(\text{SOC}, T, \text{SOH}, I). \tag{4.52}$$

Pentru o analiză simplificată poate fi utilizată o valoare constantă determinată experi-

mental în regiunea relevantă de funcționare.

4.19.2 Modelul Thevenin de ordinul întâi

Pentru descrierea efectelor dinamice se adaugă o ramură paralelă R_1C_1; tensiunea pe această ramură, V_1, modelează polarizarea electrochimică. Ecuația tensiunii la borne este:

$$V_{\text{terminal}} = V_{\text{OC}}(\text{SOC}) - IR_0 - V_1. \tag{4.53}$$

Dinamica ramurii, cu constanta de timp $\tau_1 = R_1C_1$, este:

$$\frac{dV_1}{dt} = -\frac{V_1}{R_1C_1} + \frac{I}{C_1}. \tag{4.54}$$

După conectarea unei sarcini apar două efecte: o cădere instantanee produsă de R_0 și o variație graduală produsă de ramura R_1C_1; la eliminarea curentului, tensiunea nu revine instantaneu la V_{OC}, ci prezintă un fenomen de relaxare.

4.19.3 Modelele de ordin superior

O singură ramură RC nu poate descrie simultan toate fenomenele rapide și lente; pot fi adăugate mai multe ramuri:

$$V_{\text{terminal}} = V_{\text{OC}} - IR_0 - \sum_{k=1}^{n} V_k, \tag{4.55}$$

iar pentru fiecare ramură:

$$\frac{dV_k}{dt} = -\frac{V_k}{R_kC_k} + \frac{I}{C_k}. \tag{4.56}$$

Creșterea ordinului poate îmbunătăți precizia, dar introduce mai mulți parametri și mărește costul de calcul - pentru un microcontroler cu resurse reduse, modelul de ordinul întâi rămâne adesea un compromis adecvat.

4.19.4 Modelul discret

Pentru implementarea într-un microcontroler, ecuațiile continue sunt transformate într-o formă discretă. Pentru perioada de eșantionare T_s, ecuația ramurii RC poate fi scrisă:

$$V_1[k+1] = aV_1[k] + bI[k], \tag{4.57}$$

unde $a = e^{-T_s/(R_1C_1)}$ și $b = R_1\left(1 - e^{-T_s/(R_1C_1)}\right)$. Tensiunea terminală devine:

$$V_{\text{terminal}}[k] = V_{\text{OC}}(\text{SOC}[k]) - I[k]R_0 - V_1[k]. \tag{4.58}$$

Această formă poate fi implementată eficient și poate fi utilizată într-un filtru Kalman pentru estimarea SOC.

4.19.5 Identificarea parametrilor

Parametrii R_0, R_1 și C_1 pot fi identificați prin aplicarea unor impulsuri de curent și măsurarea răspunsului tensiunii. Într-un test simplificat: bateria e lăsată să ajungă la echilibru, se aplică un impuls de curent cunoscut, se măsoară căderea instantanee de tensiune, se urmărește evoluția lentă a tensiunii, apoi sarcina e eliminată și se observă relaxarea. Rezistența serie poate fi aproximată prin:

$$R_0 \approx \frac{\Delta V_{\text{instantanee}}}{\Delta I}. \tag{4.59}$$

Constanta de timp poate fi estimată din răspunsul exponențial, iar apoi pot fi determinate R_1 și C_1. Testele trebuie repetate la diferite valori SOC și temperaturi, deoarece parametrii nu sunt constanți pe întregul domeniu.

4.19.6 Modele electrochimice și modele bazate pe date

Modelele electrochimice descriu transportul ionilor, difuzia și reacțiile de la suprafața electrozilor, oferind informații detaliate, dar cu numeroși parametri și resurse de calcul mari. Modelele bazate pe date utilizează regresie, rețele neuronale sau alte metode de învățare automată, aproximând relații neliniare fără descrierea explicită a proceselor electrochimice - limitarea principală fiind dependența de setul de date, un model antrenat pentru o anumită celulă, temperatură și profil de sarcină putând produce erori în alte condiții. În aplicațiile critice se folosește frecvent o combinație între un model fizic simplificat și o metodă de estimare adaptivă.

Tabela 4.10 Comparația principalelor modele de baterie

Model	Avantaj	Limitare	Utilizare
Sursă ideală	Foarte simplu	Nu descrie comportamentul real	Estimări preliminare
Rint	Include căderea instantanee de tensiune	Nu descrie polarizarea și relaxarea	Dimensionarea sursei și analiza impulsurilor
Thevenin 1RC	Compromis bun între precizie și complexitate	Necesită identificarea parametrilor	Estimare SOC și simulare embedded
Model cu mai multe ramuri RC	Descrie mai multe constante de timp	Complexitate și număr mare de parametri	Simulări mai precise
Electrochimic	Descriere fizică detaliată	Cost de calcul ridicat	Cercetare, proiectare și diagnostic
Bazat pe date	Poate aproxima neliniarități complexe	Depinde de datele de antrenare	Estimare și predicție pe platforme performante

4.20 Influența temperaturii, descărcării și îmbătrânirii

Bateria și sistemul embedded trebuie evaluate în întregul domeniu de funcționare: o soluție care funcționează în laborator la temperatura camerei poate deveni instabilă la temperaturi scăzute sau după mai mulți ani de utilizare.

Figura 4.18 sintetizează principalele influențe asupra performanței bateriei.

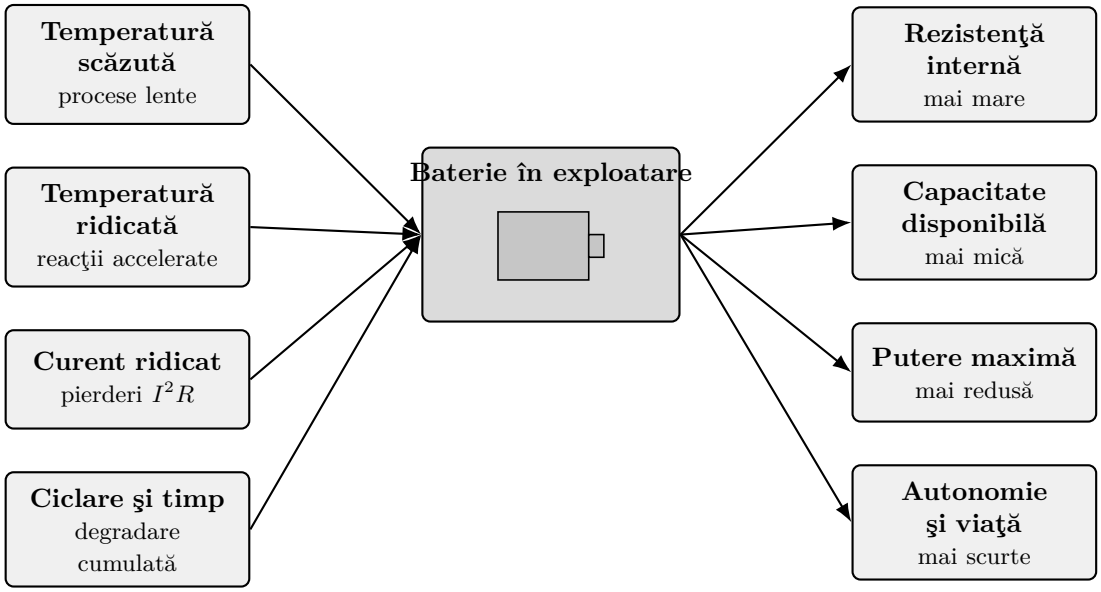


Figura 4.18 Influența temperaturii, curentului și îmbătrânirii asupra bateriei.

Temperatura scăzută. La temperaturi scăzute, procesele electrochimice devin mai lente, iar rezistența internă crește, ceea ce produce căderi mai mari de tensiune, putere maximă redusă, capacitate aparentă mai mică, timp de încărcare mai mare și risc de oprire prematură a sistemului. O parte din capacitate poate redeveni disponibilă după încălzirea bateriei - pierderea la temperatură scăzută nu e deci în întregime o degradare permanentă. Pentru bateriile Li-Ion, încărcarea la temperaturi foarte scăzute poate produce depunerea litiului metalic și degradarea celulei; limitele exacte trebuie preluate din documentația producătorului [46].

Temperatura ridicată poate reduce temporar rezistența internă și îmbunătăți puterea furnizată, dar accelerează reacțiile secundare și îmbătrânirea, ducând pe termen lung la pierderea capacității, creșterea rezistenței interne, autodescărcare mai rapidă, degradarea electrozilor și electrolitului și reducerea siguranței. O baterie nu trebuie evaluată doar după performanța instantanee - funcționarea continuă la temperatură ridicată poate reduce semnificativ durata calendaristică.

Curentul de descărcare. Creșterea curentului produce o cădere de tensiune mai mare:

$$V_{\text{terminal}} = V_{\text{OC}} - IR_{\text{intern}}. \tag{4.60}$$

Pierderile interne cresc pătratic:

$$P_{\text{pierderi}} = I^2 R_{\text{intern}}. \tag{4.61}$$

Dublarea curentului conduce, pentru aceeași rezistență, la creșterea de patru ori a pierderilor rezistive. La curenți mari, sistemul poate atinge pragul minim de tensiune înainte ca bateria să fie descărcată electrochimic, iar capacitatea utilizabilă pentru aplicație devine astfel mai mică decât cea nominală.

Regimul pulsatoriu și relaxarea. În timpul unui impuls, tensiunea scade din cauza rezistenței interne și a polarizării; după eliminarea sarcinii, tensiunea revine parțial - fenomen numit relaxare, care explică de ce un dispozitiv se poate opri în timpul unei activități intense chiar dacă tensiunea bateriei măsurată ulterior pare normală. Pentru un nod radio, intervalul dintre transmisii poate permite o revenire parțială, dar dacă impulsurile sunt prea frecvente, bateria nu mai ajunge la echilibru.

Îmbătrânirea calendaristică apare chiar dacă bateria nu e ciclată și depinde în special de timp, temperatură, starea de încărcare la depozitare și chimie. Pentru multe baterii Li-Ion, păstrarea timp îndelungat la temperatură ridicată și SOC ridicat accelerează degradarea; condițiile recomandate de depozitare trebuie preluate din specificațiile producătorului.

Îmbătrânirea prin ciclare. Fiecare ciclu produce modificări fizice și chimice, gradul de degradare depinzând de adâncimea ciclului, curentul de încărcare/descărcare, temperatură, limitele de tensiune și timpul petrecut la SOC ridicat sau redus. O mie de cicluri cu adâncime redusă nu sunt neapărat echivalente cu o mie de cicluri complete - pentru comparare se folosește conceptul de cicluri complete echivalente (de exemplu, două descărcări de câte 25% într-o zi corespund împreună la 50% dintr-un ciclu complet echivalent). Mecanismele de îmbătrânire conduc în general la pierderea capacității și creșterea rezistenței interne [16, 77].

4.20.1 Consumurile auxiliare

Într-un sistem cu autonomie mare, consumul circuitelor auxiliare poate deveni determinant: curentul propriu al regulatorului, circuitul de protecție, divizorul pentru măsurarea tensiunii, LED-urile, memoria de retenție și autodescărcarea bateriei. Curentul mediu total poate fi scris:

$$I_{\text{mediu,total}} = I_{\text{aplicație}} + I_{\text{regulator}} + I_{\text{protecție}} + I_{\text{msurare}} + I_{\text{alte pierderi}}. \tag{4.62}$$

Un regulator cu un curent propriu de $20\,\mu$A nu este potrivit pentru un nod a cărui electronică principală consumă în medie $5\,\mu$A.

4.20.2 Marginea de proiectare

Dimensionarea bateriei nu trebuie realizată pentru condițiile ideale - e necesară o rezervă pentru variația componentelor, temperatură, îmbătrânire și incertitudinea profilului de utilizare. O formulă preliminară este:

$$E_{\text{baterie,necesar}} = \frac{P_{\text{medie}} T_{\text{cerut}}}{\eta_{\text{conv}} k_{\text{util}}} \cdot M, \tag{4.63}$$

unde k_{util} e fracția utilizabilă a energiei nominale, iar M un factor de rezervă mai mare decât 1, justificat prin analiza riscurilor și prin teste - o rezervă arbitrară foarte mare poate conduce la o baterie supradimensionată, iar una prea mică poate compromite autonomia.

4.21 Studii de caz

Studiile următoare arată modul în care puterea medie, impulsurile, randamentul și condițiile de mediu influențează dimensionarea energetică.

4.21.1 Studiul de caz 1: nod IoT alimentat de o celulă tip monedă

Se consideră un nod wireless alimentat de o baterie de:

$$V_{\text{nominal}} = 3\,\text{V}, \qquad Q_{\text{nominal}} = 220\,\text{mAh}.$$

Nodul efectuează un ciclu la fiecare 60 de secunde:

Tabela 4.11 Profilul energetic al nodului IoT

Stare	Curent	Durată
Sleep	$3\,\mu$A	59,842 s
Măsurare	2 mA	100 ms
Procesare	4 mA	50 ms
Transmisie	15 mA	8 ms

Sarcina consumată într-un ciclu este:

$$\begin{aligned} Q_{\text{ciclu}} = {} & 0{,}003 \cdot 59{,}842 + 2 \cdot 0{,}100 \\ & + 4 \cdot 0{,}050 + 15 \cdot 0{,}008. \end{aligned} \tag{4.64}$$

Rezultă:

$$Q_{\text{ciclu}} \approx 0{,}700\,\text{mA s}.$$

Curentul mediu este:

$$I_{\text{mediu}} = \frac{0{,}700}{60} \approx 0{,}0117\,\text{mA} = 11{,}7\,\mu\text{A}. \tag{4.65}$$

Autonomia ideală este:

$$T_{\text{ideal}} = \frac{220\,\text{mAh}}{0{,}0117\,\text{mA}} \approx 18\,800\,\text{h}. \tag{4.66}$$

Aceasta corespunde unei autonomii de aproximativ:

$$T_{\text{ideal}} \approx 2{,}15\,\text{ani}.$$

Dacă se consideră utilizabilă numai 70% din capacitatea nominală:

$$T_{\text{corectat}} = 2{,}15 \cdot 0{,}70 \approx 1{,}5\,\text{ani}. \tag{4.67}$$

Calculul nu demonstrează însă că bateria poate furniza impulsul de 15 mA. Dacă rezistența internă este ridicată, tensiunea poate scădea sub pragul microcontrolerului.

Pentru o variație acceptată de 0,2 V, condensatorul care ar furniza singur impulsul ar avea:

$$C \geq \frac{0{,}015 \cdot 0{,}008}{0{,}2} = 600\,\mu\text{F}. \tag{4.68}$$

În practică, bateria furnizează o parte din curent, iar condensatorul necesar poate fi mai mic. Studiul evidențiază însă că autonomia calculată din mAh nu este suficientă. Trebuie verificată stabilitatea tensiunii în timpul transmisiei.

Figura 4.19 prezintă principalele elemente ale acestei analize.

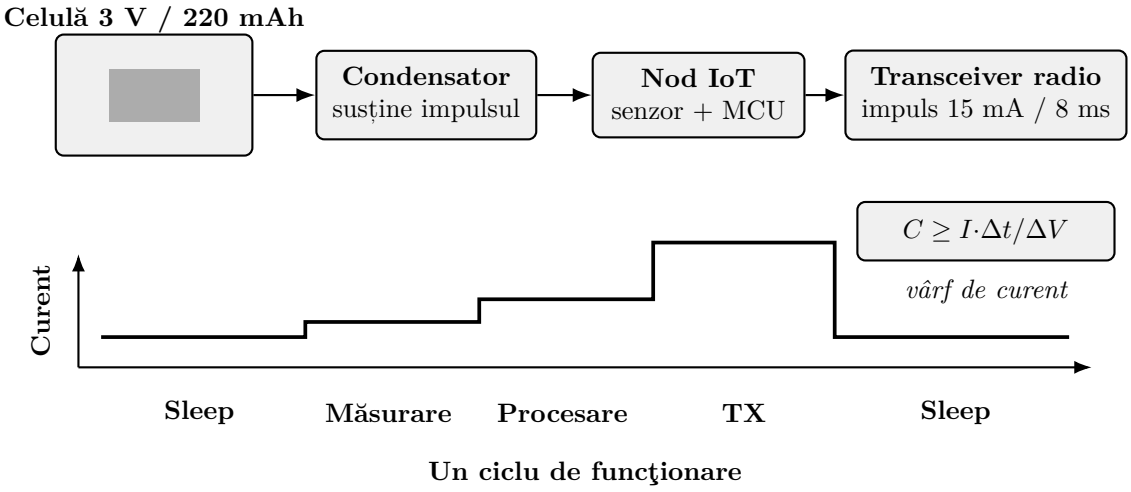


Figura 4.19 Analiza energetică a unui nod IoT alimentat de o celulă tip monedă.

4.21.2 Studiul de caz 2: nod alimentat prin energy harvesting

Se consideră un nod alimentat de o celulă fotovoltaică. Puterea medie consumată este:

$$P_{\text{medie}} = 0{,}5\,\text{mW}.$$

Energia necesară zilnic este:

$$E_{\text{zi}} = 0{,}5\,\text{mW} \cdot 24\,\text{h} = 12\,\text{mWh}. \tag{4.69}$$

Dacă randamentul lanțului de conversie este 75%, energia care trebuie captată este:

$$E_{\text{captat}} = \frac{12}{0{,}75} = 16\,\text{mWh/zi}. \tag{4.70}$$

Se presupun patru ore solare echivalente pe zi și un factor de reducere de 60%, care include orientarea și variația iluminării.

Puterea minimă a panoului este:

$$P_{\text{panou}} \geq \frac{16\,\text{mWh}}{4\,\text{h} \cdot 0{,}60} \approx 6{,}7\,\text{mW}. \tag{4.71}$$

Această valoare reprezintă un minim teoretic. În practică, panoul trebuie dimensionat pentru sezonul nefavorabil, depuneri, umbrire și variații ale consumului.

Pentru trei zile fără lumină, energia necesară stocării este:

$$E_{\text{stocare}} \geq \frac{12 \cdot 3}{0{,}8 \cdot 0{,}9} = 50\,\text{mWh}, \tag{4.72}$$

unde 0,8 reprezintă fracția utilizabilă a stocării, iar 0,9 randamentul conversiei.

Pentru o baterie cu tensiunea nominală de 3,7 V:

$$Q_{\text{min}} = \frac{50\,\text{mWh}}{3{,}7\,\text{V}} \approx 13{,}5\,\text{mAh}. \tag{4.73}$$

Într-un produs real va fi aleasă o capacitate mai mare pentru a compensa îmbătrânirea, temperatura și perioadele mai lungi cu iluminare redusă.

4.21.3 Studiul de caz 3: robot mobil

Se consideră un robot mobil alimentat de un pachet Li-Ion 4S:

$$V_{\text{nominal}} = 14{,}8\,\text{V}, \qquad Q_{\text{nominal}} = 5\,\text{Ah}.$$

Energia nominală este:

$$E_{\text{nominal}} = 14{,}8 \cdot 5 = 74\,\text{Wh}. \tag{4.74}$$

Puterea medie este distribuită astfel:

- motoare: 38 W;
- procesor și electronică: 8 W;
- senzori și comunicație: 4 W.

Puterea medie totală este:

$$P_{\text{medie}} = 38 + 8 + 4 = 50\,\text{W}. \tag{4.75}$$

Autonomia ideală este:

$$T_{\text{ideal}} = \frac{74}{50} = 1{,}48\,\text{h}. \tag{4.76}$$

Dacă se utilizează 80% din energia nominală, iar randamentul global este 90%:

$$T_{\text{real}} = \frac{74 \cdot 0{,}8 \cdot 0{,}9}{50} \approx 1{,}07\,\text{h}. \tag{4.77}$$

Autonomia estimată este astfel de aproximativ 64 de minute.

Să presupunem că robotul solicită un curent de vârf de 12 A, iar rezistența totală a pachetului și conexiunilor este:

$$R_{\text{total}} = 60\,\text{m}\Omega.$$

Căderea de tensiune este:

$$\Delta V = 12 \cdot 0{,}060 = 0{,}72\,\text{V}. \tag{4.78}$$

Puterea disipată în timpul impulsului este:

$$P_{\text{pierderi}} = 12^2 \cdot 0{,}060 = 8{,}64\,\text{W}. \tag{4.79}$$

Aceste pierderi apar numai în timpul impulsului, dar pot produce încălzire și pot activa protecția pachetului. Prin urmare, selecția bateriei trebuie să considere curentul maxim, nu numai energia nominală.

În cazul robotului, cele mai mari economii pot fi obținute prin:

- reducerea masei;
- optimizarea traiectoriei;
- reducerea frecărilor;
- limitarea accelerațiilor;
- alegerea unor motoare și reductoare eficiente.

Optimizarea consumului microcontrolerului are un impact redus dacă motoarele domină energia totală.

4.22 Metodologie de proiectare energetică

Proiectarea energetică trebuie realizată iterativ - estimarea exclusiv pe baza valorilor tipice din fișele tehnice poate conduce la erori, deoarece acestea sunt măsurate în condiții specifice. O metodologie recomandată este prezentată în Figura 4.20.

Etapele principale sunt:

1. definirea autonomiei și a condițiilor de funcționare;
2. identificarea tuturor stărilor energetice;
3. determinarea duratei și consumului fiecărei stări;
4. calcularea energiei unui ciclu reprezentativ;
5. identificarea consumatorilor dominanți;

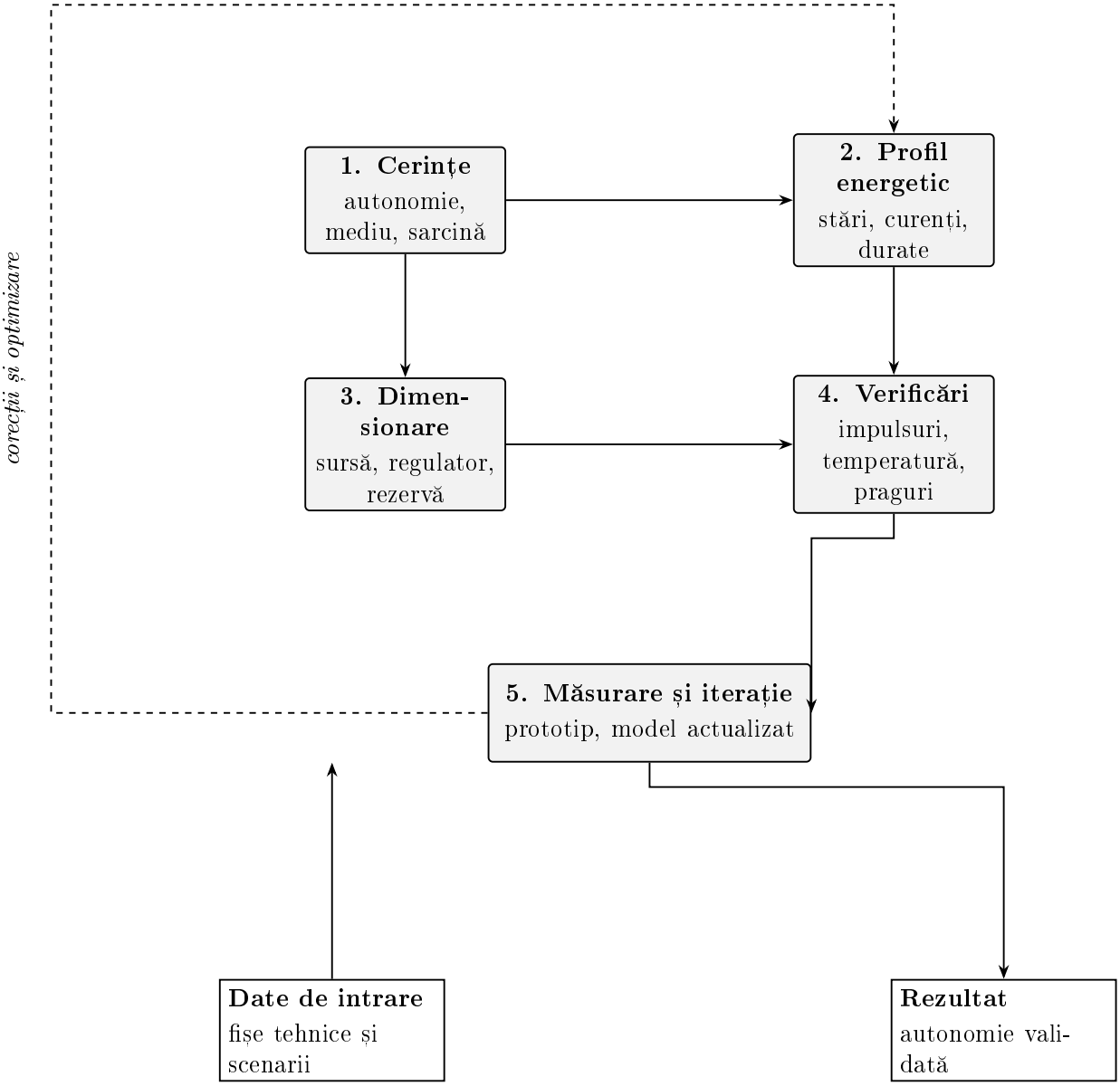


Figura 4.20 Fluxul de proiectare energetică a unui sistem embedded.

6. alegerea sursei și a regulatorului;
7. verificarea curenților de vârf și a pragurilor de tensiune;
8. aplicarea factorilor pentru temperatură și îmbătrânire;
9. realizarea măsurătorilor pe prototip;
10. actualizarea modelului și repetarea calculului.

Măsurarea trebuie realizată pe un ciclu complet și reprezentativ - o măsurare scurtă, efectuată numai în modul activ sau numai în sleep, nu descrie energia totală.

4.23 Întrebări și probleme propuse

1. Care este diferența dintre putere, energie și sarcină electrică?
2. De ce o baterie de 1000 mAh nu conține obligatoriu mai multă energie decât una de 800 mAh?
3. Un sistem consumă 20 mA timp de două secunde și 10 μA timp de 58 de secunde. Determinați curentul mediu.
4. Explicați diferența dintre puterea dinamică și puterea statică a unui circuit CMOS.
5. De ce reducerea frecvenței procesorului nu conduce întotdeauna la o reducere proporțională a energiei?
6. Care este diferența dintre clock gating și power gating?
7. Ce reprezintă timpul de break-even al unui mod de consum redus?
8. De ce recepția continuă poate consuma mai multă energie decât transmiterea periodică?
9. În ce condiții este avantajoasă procesarea locală a datelor?
10. Care este diferența dintre SOC, DOD și SOH?
11. O baterie are $V_{OC} = 3{,}1\,\text{V}$, $R_{intern} = 20\,\Omega$ și furnizează un curent de 40 mA. Determinați tensiunea la borne.
12. Determinați condensatorul minim necesar pentru un impuls de 30 mA cu durata de 10 ms, dacă scăderea maximă acceptată a tensiunii este 0,15 V.
13. Explicați de ce o baterie poate produce resetarea sistemului în timpul transmisiei radio, deși tensiunea măsurată fără sarcină este suficientă.
14. Comparați modelul Rint cu modelul Thevenin de ordinul întâi.
15. De ce parametrii modelului unei baterii trebuie identificați la mai multe temperaturi și valori SOC?
16. Un pachet de 12 V și 4 Ah alimentează un sistem care consumă în medie 24 W. Determinați autonomia ideală și autonomia pentru o energie utilizabilă de 75%.
17. Proiectați profilul energetic al unui senzor care măsoară temperatura la fiecare cinci minute și transmite datele o dată pe oră.
18. Identificați consumatorul dominant într-un robot mobil și propuneți trei metode de reducere a energiei totale.
19. Explicați modul în care temperatura scăzută poate reduce autonomia fără a produce în mod obligatoriu o pierdere permanentă de capacitate.

20. Elaborați o metodă experimentală pentru determinarea rezistenței interne a unei baterii.

5. Tehnici de Optimizare a Consumului Energetic

Introducere în optimizarea energetică
Obiective și metrici de optimizare
Modelul energetic al circuitelor CMOS
Reducerea activității de comutare
Reducerea frecvenței de ceas
Scalarea dinamică a tensiunii și frecvenței
Clock Gating
Power Gating
Paralelismul și eficiența energetică
Pipeline și eficiența energetică
Arhitecturi VLIW și acceleratoare specializate
Managementul dinamic al puterii

5.1 Introducere în optimizarea energetică

Capitolul anterior a prezentat principalele surse de consum energetic ale unui sistem embedded, profilul temporal al consumului, metodele de estimare a autonomiei și comportamentul real al surselor de alimentare. După identificarea și măsurarea consumului, următoarea etapă este optimizarea energetică — procesul prin care energia necesară realizării funcțiilor sistemului este redusă fără încălcarea cerințelor de performanță, timp real, siguranță, precizie și fiabilitate.

Obiectivul nu este întotdeauna cel mai mic consum instantaneu: un sistem poate funcționa la putere redusă, dar pentru o durată foarte mare, caz în care energia totală crește; în alte situații, executarea rapidă a unei activități urmată de revenirea imediată în repaus poate fi mai eficientă. Optimizarea trebuie realizată pentru un scenariu complet de funcționare:

$$E_{\text{total}} = \int_{t_0}^{t_1} P(t)\,dt. \tag{5.1}$$

Într-un sistem embedded, consumul energetic influențează direct autonomia, temperatura de funcționare, dimensiunea și masa bateriei, complexitatea sursei de alimentare, fiabilitatea

componentelor, costul mentenanței și impactul asupra mediului. Importanța fiecărui obiectiv depinde de aplicație: pentru un senzor greu accesibil, autonomia și mentenanța pot fi criteriile principale; pentru un controler automotive, prioritare sunt cerințele de timp real și limitele termice; pentru un robot mobil, consumul actuatoarelor poate domina întregul buget energetic.

5.1.1 Optimizarea energetică drept problemă cu constrângeri

Reducerea consumului nu trebuie analizată separat de celelalte cerințe ale sistemului. Într-o formă generală, problema poate fi exprimată astfel:

$$\min_{\mathbf{x}} E(\mathbf{x}), \tag{5.2}$$

sub constrângerile:

$$T_{\text{exec}}(\mathbf{x}) \leq D, \tag{5.3}$$

$$P_{\max}(\mathbf{x}) \leq P_{\text{admis}}, \tag{5.4}$$

$$T_{\text{funcțiune}}(\mathbf{x}) \leq T_{\max}, \tag{5.5}$$

$$Q_{\text{serviciu}}(\mathbf{x}) \geq Q_{\min}, \tag{5.6}$$

unde $\mathbf{x}$ reprezintă parametrii care pot fi modificați (frecvența, tensiunea, algoritmul, numărul de nuclee active, rata de eșantionare, politica de management energetic), D e deadline-ul activității, iar Q_{serviciu} descrie nivelul minim acceptabil de calitate — precizia unui algoritm, rata de cadre, timpul de răspuns, probabilitatea de livrare a unui pachet sau eroarea de control, în funcție de aplicație. Un punct de funcționare care minimizează energia dar nu respectă termenul limită nu e o soluție validă, la fel cum reducerea ratei de eșantionare nu e acceptabilă dacă determină pierderea unor evenimente critice.

5.1.2 Nivelurile de optimizare

Consumul energetic poate fi influențat la mai multe niveluri ale sistemului. Figura 5.1 prezintă o organizare ierarhică a principalelor niveluri de optimizare.

La nivel tehnologic și de circuit sunt influențate tensiunea de alimentare, capacitățile comutate, curenții de pierderi și dimensiunea tranzistoarelor. La nivel logic pot fi reduse numărul tranzițiilor, activitatea inutilă și comutările produse de hazarduri. La nivel microarhitectural sunt relevante pipeline-ul, memoria cache, acceleratoarele, paralelismul, clock gating și power gating. La nivelul sistemului de operare sau al platformei software pot fi controlate stările de performanță, modurile de repaus și planificarea sarcinilor. La nivel algoritmic și al aplicației pot fi reduse numărul operațiilor, transferurile de memorie, cantitatea de date transmisă și rata activării componentelor.

În general, optimizările aplicate la nivelurile superioare pot avea impact important, deoarece modifică volumul total de activitate solicitat hardware-ului, dar trebuie susținute de

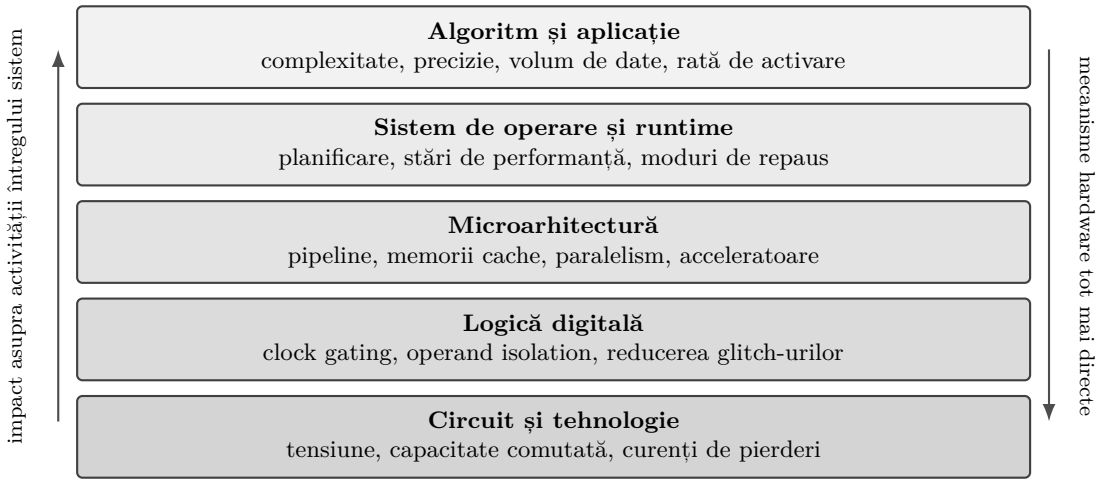


Figura 5.1 Nivelurile la care poate fi realizată optimizarea energetică.

mecanisme hardware care permit oprirea sau adaptarea efectivă a resurselor.

5.1.3 Optimizarea locală și optimizarea globală

Reducerea consumului unei singure componente nu garantează reducerea energiei întregului sistem: comprimarea datelor poate crește energia consumată de procesor, dar poate reduce mai mult energia necesară comunicației; într-o altă aplicație, un algoritm mai rapid poate necesita o memorie externă cu consum ridicat și poate deveni mai puțin eficient la nivel de sistem. Energia totală poate fi exprimată prin:

$$E_{\text{total}} = E_{\text{procesor}} + E_{\text{memorie}} + E_{\text{comunicație}} + E_{\text{senzori}} + E_{\text{actuatoare}} + E_{\text{conversie}}. \tag{5.7}$$

O optimizare trebuie evaluată prin variația întregii expresii, nu doar a unui termen.

Exemplu

Într-un sistem, comunicația reprezintă 60% din energia totală. O tehnică de compresie reduce energia comunicației cu 40%, dar crește energia procesorului cu o valoare egală cu 5% din energia inițială a sistemului.
Economia produsă în comunicație este:

$$0{,}60 \cdot 0{,}40 = 0{,}24.$$

Creșterea consumului procesorului este:

$$0{,}05.$$

Economia netă este:

$$0{,}24 - 0{,}05 = 0{,}19.$$

Energia totală este redusă cu 19%, chiar dacă energia procesorului crește.

5.1.4 Compromisuri fundamentale

Optimizarea energetică presupune frecvent compromisuri între energie și performanță, latență, precizie, fiabilitate, suprafața circuitului, complexitatea software și cost. Figura 5.2 prezintă principalele obiective care trebuie echilibrate.

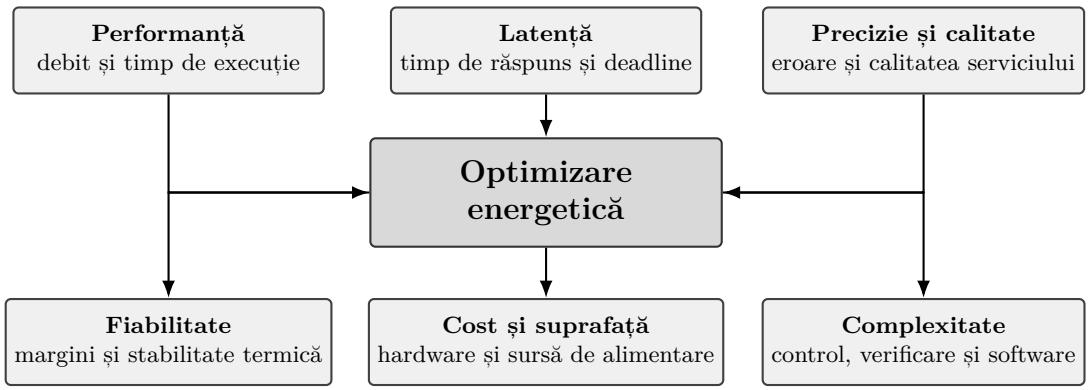


Figura 5.2 Compromisuri întâlnite în optimizarea energetică.

Un accelerator hardware poate reduce energia per operație, dar ocupă suprafață suplimentară; o frecvență redusă poate diminua puterea, dar poate crește timpul de execuție; un mod profund de repaus poate avea un consum foarte mic, dar o latență mare de reactivare. Nu există deci o tehnică universal optimă — soluția trebuie aleasă pentru profilul real al aplicației.

5.2 Obiective și metrici de optimizare

Pentru compararea variantelor de implementare trebuie stabilite metrici care reflectă obiectivele reale ale sistemului. Utilizarea exclusivă a puterii instantanee sau a curentului maxim poate conduce la concluzii greșite.

5.2.1 Puterea instantanee, medie și maximă

Puterea instantanee este:

$$P(t) = V(t)I(t). \tag{5.8}$$

Puterea medie pe intervalul T este:

$$P_{\text{medie}} = \frac{1}{T}\int_0^T P(t)\,dt. \tag{5.9}$$

Puterea maximă este:

$$P_{\text{vrf}} = \max_{t \in [0,T]} P(t). \tag{5.10}$$

Puterea medie influențează temperatura și consumul energetic pe termen lung; puterea de vârf determină dimensionarea regulatorului, a traseelor de alimentare, a condensatoarelor și a sursei. Un sistem poate avea o putere medie redusă și impulsuri foarte mari — profil întâlnit frecvent la dispozitivele wireless.

Energia per activitate. Pentru optimizarea software și arhitecturală este utilă energia consumată pentru realizarea unei activități:

$$E_{\text{activitate}} = \int_{t_{\text{start}}}^{t_{\text{stop}}} P(t)\, dt. \tag{5.11}$$

Activitatea poate fi procesarea unui eșantion, executarea unei iterații de control, codificarea unui cadru, clasificarea unei imagini, transmiterea unui pachet sau finalizarea unui ciclu de măsurare. Această metrică permite compararea unor implementări cu timpi de execuție diferiți.

5.2.2 Energia per operație și eficiența energetică

Pentru procesoare și acceleratoare poate fi utilizată energia per operație:

$$E_{\text{op}} = \frac{E_{\text{total}}}{N_{\text{operații}}}. \tag{5.12}$$

Eficiența energetică poate fi exprimată prin numărul de operații realizate pentru o unitate de energie:

$$\eta_{\text{energetic}} = \frac{N_{\text{operații}}}{E_{\text{total}}}. \tag{5.13}$$

În funcție de domeniu, pot fi utilizate metrici precum operații/J, eșantioane/J, cadre/J sau inferențe/J. Numărul de operații trebuie definit clar — o comparație între platforme nu e relevantă dacă fiecare folosește o definiție diferită a operației sau execută algoritmi cu precizii diferite.

5.2.3 Energia per informație utilă

În sistemele de comunicație este importantă energia necesară livrării informației utile:

$$E_{\text{bit,util}} = \frac{E_{\text{comunicație}}}{N_{\text{biți,utili}}}. \tag{5.14}$$

Această metrică include indirect efectele antetelor, retransmisiilor și pachetelor pierdute. Într-un sistem de senzori poate fi utilizată energia per măsurătoare validă:

$$E_{\text{msurtoare}} = \frac{E_{\text{ciclu}}}{N_{\text{msurtori,valide}}}. \tag{5.15}$$

5.2.4 Economia energetică relativă

Economia energetică față de o implementare de referință e:

$$S_E = \frac{E_{\text{referință}} - E_{\text{optimizat}}}{E_{\text{referință}}} \cdot 100\%. \tag{5.16}$$

În mod similar, reducerea puterii este:

$$S_P = \frac{P_{\text{referință}} - P_{\text{optimizat}}}{P_{\text{referință}}} \cdot 100\%. \tag{5.17}$$

Cele două valori nu sunt obligatoriu egale — reducerea puterii poate fi însoțită de creșterea timpului de execuție.

5.2.5 Energy–Delay Product

Atunci când trebuie evaluate simultan energia și timpul de execuție, poate fi utilizată metrica *Energy–Delay Product*:

$$\text{EDP} = E \cdot T. \tag{5.18}$$

O valoare redusă indică un compromis favorabil între energie și performanță. Dacă timpul trebuie penalizat mai puternic, poate fi utilizată:

$$\text{ED}^2\text{P} = E \cdot T^2. \tag{5.19}$$

EDP și ED^2P nu reprezintă mărimi fizice fundamentale, ci indicatori de comparare. Alegerea lor trebuie justificată prin cerințele aplicației.

Exemplu

Două configurații execută aceeași activitate.
Pentru configurația A:

$$E_A = 120\,\text{mJ}, \qquad T_A = 0{,}4\,\text{s}.$$

Pentru configurația B:

$$E_B = 80\,\text{mJ}, \qquad T_B = 0{,}9\,\text{s}.$$

Rezultă:

$$\text{EDP}_A = 0{,}12 \cdot 0{,}4 = 0{,}048\,\text{J s},$$

iar:

$$\mathrm{EDP}_B = 0{,}08 \cdot 0{,}9 = 0{,}072\,\mathrm{J\,s}.$$

Configurația B consumă mai puțină energie, dar configurația A are o valoare EDP mai mică. Alegerea depinde de importanța autonomiei și a timpului de răspuns.

5.2.6 Metrici dependente de aplicație

O metrică generală nu poate descrie toate cerințele unui produs. Exemple de metrici dependente de aplicație sunt:

- energie per kilometru pentru un robot mobil;
- energie per decizie de control;
- energie per imagine procesată;
- energie per eveniment detectat;
- energie per pachet livrat;
- energie per inferență corectă;
- autonomie pentru o anumită calitate a serviciului.

Pentru algoritmii aproximativi sau de inteligență artificială trebuie analizată și calitatea rezultatului. O implementare care consumă mai puțină energie, dar produce o rată inacceptabilă de eroare, nu reprezintă o optimizare validă.

Poate fi definită, de exemplu:

$$\eta_{\text{decizie}} = \frac{N_{\text{decizii,corecte}}}{E_{\text{total}}}. \tag{5.20}$$

5.2.7 Prioritizarea optimizărilor

Optimizările trebuie aplicate cu prioritate componentelor dominante. Dacă o componentă reprezintă fracția p_i din energia totală și energia ei este redusă cu fracția r_i, reducerea relativă a energiei sistemului este:

$$r_{\text{total}} = p_i r_i. \tag{5.21}$$

Exemplu

Memoria reprezintă 20% din energia totală. O optimizare reduce energia memoriei cu 50%.
Reducerea energiei întregului sistem este:

$$r_{\text{total}} = 0{,}20 \cdot 0{,}50 = 0{,}10.$$

Energia totală este redusă cu 10%, nu cu 50%.

Acest principiu arată de ce profilarea energetică trebuie realizată înaintea optimizării.

5.2.8 Tabel de sinteză a metricilor

Tabela 5.1 Metrici utilizate în optimizarea energetică

Metrică	Semnificație	Utilizare principală
Putere medie	Viteza medie de consum a energiei	Analiza termică și estimarea consumului pe termen lung
Putere de vârf	Valoarea maximă solicitată sursei	Dimensionarea regulatorului și verificarea stabilității
Energie per activitate	Energia necesară finalizării unei sarcini	Compararea implementărilor software și hardware
Energie per operație	Costul energetic mediu al unei operații	Evaluarea procesoarelor și acceleratoarelor
Operații per joule	Volumul de calcul realizat pentru o unitate de energie	Compararea eficienței energetice
Energie per bit util	Energia necesară livrării informației utile	Evaluarea sistemelor de comunicație
EDP	Produsul dintre energie și timp	Compromis între consum și performanță
ED^{2}P	Metrică ce penalizează mai puternic timpul	Sisteme cu cerințe ridicate de performanță
Economie relativă	Reducerea față de o implementare de referință	Raportarea rezultatelor optimizării

5.3 Modelul energetic al circuitelor CMOS

Majoritatea procesoarelor, microcontrolerelor și circuitelor digitale moderne sunt realizate în tehnologie CMOS; înțelegerea consumului acestei tehnologii permite identificarea parametrilor care pot fi modificați prin tehnici precum DVFS, clock gating și power gating. Modelul energetic trebuie privit ca o aproximație — consumul real depinde de tehnologia de fabricație, temperatură, distribuția semnalului de ceas, activitatea datelor, memorii și circuitele analogice integrate.

Invertorul CMOS este format dintr-un tranzistor PMOS conectat la alimentare și un tranzistor NMOS conectat la masă. Figura 5.3 prezintă structura sa conceptuală și cele două stări logice stabile.

Pentru o intrare logică 0, PMOS conduce și NMOS e blocat, ieșirea fiind conectată la alimentare (logic 1); pentru o intrare logică 1, PMOS e blocat și NMOS conduce, ieșirea fiind conectată la masă (logic 0). În modelul ideal, în stările logice stabile nu există o cale conductoare directă între alimentare și masă, dar circuitul real prezintă curenți de pierderi.

5.3.1 Capacitatea de sarcină

Ieșirea unei porți CMOS poate fi reprezentată printr-o capacitate echivalentă C_L, care include capacitățile tranzistoarelor, capacitatea interconexiunilor, intrările porților coman-

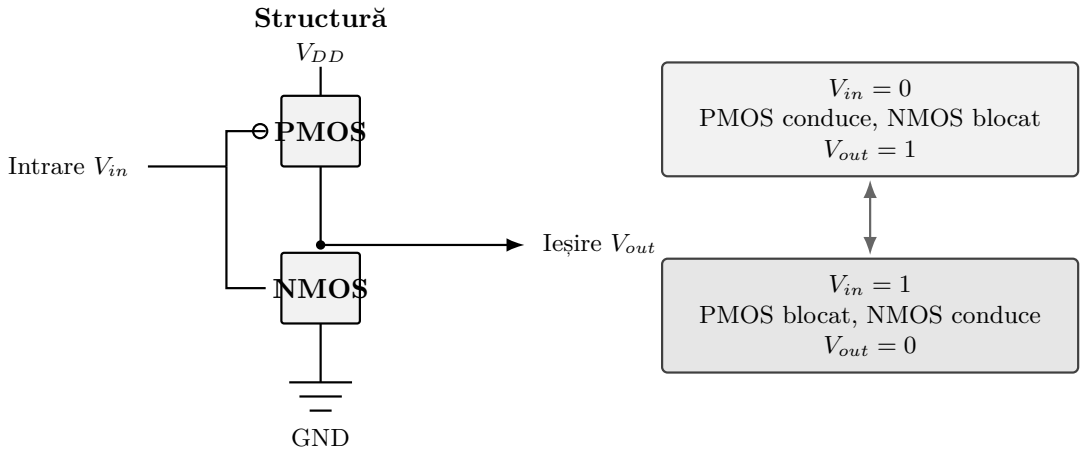


Figura 5.3 Structura și funcționarea logică a unui invertor CMOS.

date și capacitățile parazite. La tranziția ieșirii de la 0 la 1, sursa de alimentare încarcă această capacitate. Sarcina acumulată este:

$$Q = C_L V_{DD}. \tag{5.22}$$

Energia stocată în condensator este:

$$E_C = \frac{1}{2} C_L V_{DD}^2. \tag{5.23}$$

Energia preluată de la sursă în timpul încărcării este:

$$E_{\text{surs}} = C_L V_{DD}^2. \tag{5.24}$$

Jumătate este stocată temporar în condensator, iar cealaltă jumătate este disipată în rețeaua de tranzistoare. La descărcare, energia stocată este disipată către masă.

Pentru un ciclu complet $0 \to 1 \to 0$, energia extrasă din sursă este aproximativ:

$$E_{\text{ciclu}} = C_L V_{DD}^2. \tag{5.25}$$

5.3.2 Puterea de comutare

Într-un circuit format din mai multe noduri, puterea de comutare poate fi exprimată prin:

$$P_{\text{comutare}} = V_{DD}^2 f \sum_{i=1}^{n} \alpha_i C_i, \tag{5.26}$$

unde C_i e capacitatea asociată nodului i, α_i activitatea de comutare a nodului, f frecvența de ceas și V_{DD} tensiunea de alimentare. Prin introducerea unei capacități efective și a unui factor mediu de activitate se obține relația cunoscută:

$$P_{\text{comutare}} = \alpha C_{\text{ef}} V_{DD}^2 f. \tag{5.27}$$

În literatură poate apărea și un factor $\frac{1}{2}$, în funcție de modul în care e definit factorul de activitate, așa că definiția lui α trebuie precizată la compararea unor modele diferite [22, 69]. Relația evidențiază patru direcții de optimizare: reducerea activității α, a capacității efective C_{ef}, a tensiunii V_{DD} și a frecvenței f. Tensiunea are o influență pătratică asupra puterii de comutare, dar nu poate fi redusă independent de cerințele de frecvență și de stabilitatea circuitului.

5.3.3 Puterea de scurtcircuit

În timpul unei tranziții a semnalului de intrare există un interval în care tranzistoarele PMOS și NMOS conduc simultan, formând temporar o cale între alimentare și masă. Puterea medie asociată poate fi exprimată prin:

$$P_{\text{scurtcircuit}} = V_{DD} I_{\text{sc,mediu}}. \tag{5.28}$$

Aceasta depinde de timpul de creștere/descreștere al intrării, tensiunea de alimentare, tensiunile de prag, dimensionarea tranzistoarelor și sarcina de la ieșire. O tranziție lentă poate menține ambele tranzistoare în conducție mai mult timp; proiectarea corectă a etajelor de comandă și evitarea fronturilor excesiv de lente limitează această componentă. Figura 5.4 prezintă conceptual curentul de scurtcircuit în timpul unei tranziții.

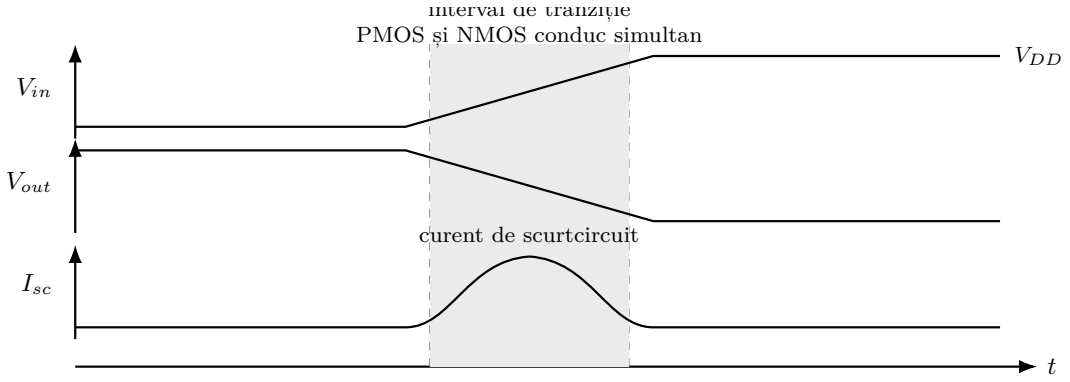


Figura 5.4 Apariția curentului de scurtcircuit în timpul comutării CMOS.

5.3.4 Puterea statică

Puterea statică este produsă de curenții care circulă chiar și atunci când circuitul nu efectuează tranziții logice. Un model simplificat este:

$$P_{\text{static}} = V_{DD} I_{\text{leak}}. \tag{5.29}$$

Curentul total de pierderi include mai multe mecanisme fizice; pentru analiza la nivel

de sistem e suficientă observarea dependenței sale de tehnologia de fabricație, tensiunea de alimentare, temperatura joncțiunii, starea tranzistoarelor și numărul de componente alimentate. Creșterea temperaturii conduce, în general, la creșterea curenților de pierderi, ceea ce poate produce o reacție pozitivă între consum și temperatură:

$$P_{\text{static}} \uparrow \quad \Rightarrow \quad T_{\text{joncțiune}} \uparrow \quad \Rightarrow \quad I_{\text{leak}} \uparrow .$$

Puterea statică e importantă în special pentru blocurile care rămân alimentate, dar inactive, timp îndelungat.

5.3.5 Modelul total

Pentru un circuit digital, puterea totală poate fi aproximată prin:

$$P_{\text{CMOS}} = P_{\text{comutare}} + P_{\text{scurtcircuit}} + P_{\text{static}}. \tag{5.30}$$

Prin înlocuire:

$$P_{\text{CMOS}} = \alpha C_{\text{ef}} V_{DD}^2 f + V_{DD} I_{\text{sc,mediu}} + V_{DD} I_{\text{leak}}. \tag{5.31}$$

Într-un microcontroler real trebuie adăugat și consumul blocurilor analogice, al memoriilor, al oscilatoarelor și al regulatoarelor:

$$P_{\text{sistem}} = P_{\text{CMOS}} + P_{\text{memorie}} + P_{\text{analogic}} + P_{\text{ceas}} + P_{\text{I/O}} + P_{\text{conversie}}. \tag{5.32}$$

Figura 5.5 prezintă componentele modelului.

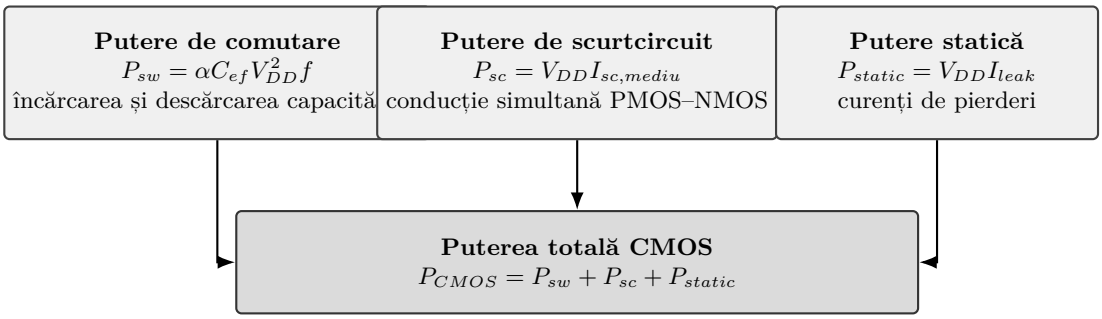


Figura 5.5 Componentele modelului energetic al unui circuit CMOS.

5.3.6 Energia unei activități cu număr fix de cicluri

Considerăm o activitate care necesită N cicluri de ceas. Timpul de execuție idealizat este:

$$T_{\text{exec}} = \frac{N}{f}. \tag{5.33}$$

Energia de comutare este:

$$\begin{aligned} E_{\text{comutare}} &= P_{\text{comutare}} T_{\text{exec}} \\ &= \alpha C_{\text{ef}} V_{DD}^2 f \frac{N}{f}. \end{aligned} \tag{5.34}$$

Rezultă:

$$E_{\text{comutare}} = \alpha C_{\text{ef}} V_{DD}^2 N. \tag{5.35}$$

În modelul ideal, frecvența nu mai apare explicit — reducerea frecvenței la tensiune constantă reduce puterea, dar mărește timpul, iar energia de comutare rămâne aproximativ constantă. Energia statică este:

$$E_{\text{static}} = P_{\text{static}} \frac{N}{f}. \tag{5.36}$$

Reducerea frecvenței poate crește energia statică deoarece circuitul rămâne alimentat mai mult timp.

Aceste relații explică de ce reducerea frecvenței este eficientă energetic mai ales atunci când permite și reducerea tensiunii sau când modifică favorabil stările de repaus ale sistemului.

5.3.7 Exemplu numeric

Exemplu

Se consideră un circuit cu:

$$\alpha = 0{,}2, \qquad C_{\text{ef}} = 100\,\text{pF},$$

$$V_{DD} = 3{,}3\,\text{V}, \qquad f = 100\,\text{MHz}.$$

Puterea de comutare este:

$$P_{\text{comutare}} = 0{,}2 \cdot 100 \cdot 10^{-12} \cdot 3{,}3^2 \cdot 100 \cdot 10^6.$$

Rezultă:

$$P_{\text{comutare}} \approx 21{,}8\,\text{mW}.$$

Dacă tensiunea este redusă la 2,5 V, iar ceilalți parametri rămân neschimbați:

$$\frac{P_{\text{nou}}}{P_{\text{inițial}}} = \left(\frac{2{,}5}{3{,}3}\right)^2 \approx 0{,}574.$$

Puterea de comutare scade la aproximativ 57,4% din valoarea inițială. În practică trebuie verificat dacă frecvența de 100 MHz mai poate fi susținută la tensiunea redusă.

Limitele modelului. Ecuația CMOS e utilă pentru înțelegerea tendințelor, dar nu permite singură determinarea consumului exact al unui microcontroler — nu include explicit dependența frecvenței maxime de tensiune, variația curenților de pierderi cu temperatura, activitatea diferită a instrucțiunilor, accesările memoriei, consumul oscilatoarelor și al PLL-urilor, randamentul regulatorului sau tranzițiile între stările de putere. Valorile finale trebuie obținute din fișele tehnice și validate prin măsurători.

5.4 Reducerea activității de comutare

Conform modelului CMOS, puterea dinamică este direct proporțională cu factorul de activitate:

$$P_{\text{comutare}} \propto \alpha. \tag{5.37}$$

Reducerea numărului tranzițiilor inutile poate diminua consumul fără modificarea tensiunii sau a frecvenței. Activitatea de comutare poate fi redusă la nivel logic, arhitectural, algoritmic și software.

5.4.1 Definirea factorului de activitate

Pentru un nod digital, factorul de activitate poate reprezenta numărul mediu de tranziții $0 \to 1$ într-un ciclu de ceas. Pentru N_{cicluri} și $N_{0\to1}$ tranziții:

$$\alpha = \frac{N_{0\to1}}{N_{\text{cicluri}}}. \tag{5.38}$$

În alte modele, α poate include toate tranzițiile sau probabilitatea ca nodul să își schimbe starea, deci convenția trebuie specificată. Activitatea unui procesor depinde de instrucțiunile executate, valorile operanzilor, accesările de memorie, ramificații, întreruperi, perifericele active și organizarea magistralelor. Figura 5.6 prezintă diferența dintre activitatea utilă și tranzițiile care nu contribuie la rezultatul final.

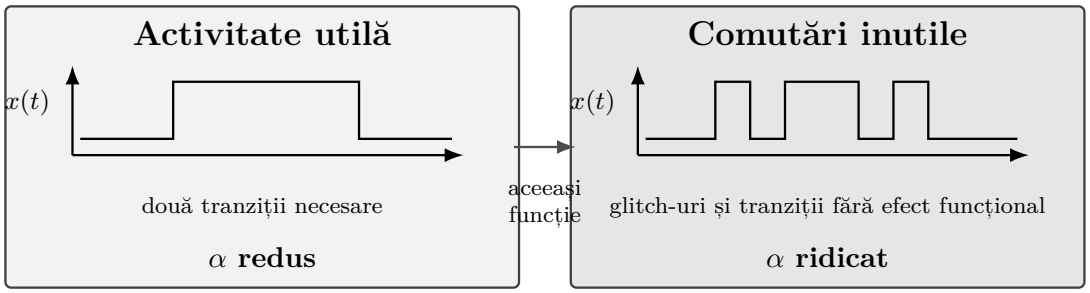


Figura 5.6 Activitatea utilă și comutările energetice inutile.

Eliminarea operațiilor inutile. Prima direcție de optimizare constă în reducerea numărului de operații executate: eliminarea calculelor redundante, reutilizarea rezultatelor intermediare, evitarea conversiilor repetate, precalcularea constantelor, oprirea algoritmului când rezultatul e deja determinat, procesarea numai a datelor modificate. O valoare calculată de mai multe ori poate fi stocată și reutilizată, dar această decizie introduce accesări de memorie — avantajul trebuie evaluat în funcție de costul calculului și costul memorării.

Execuția condiționată. Blocurile de calcul trebuie activate numai când rezultatul lor e necesar — un algoritm care filtrează permanent un semnal, deși aplicația utilizează rezultatul doar la apariția unui eveniment, produce activitate inutilă. Execuția bazată pe evenimente permite activarea condiționată a procesării:

$$\text{activare} = \begin{cases} 1, & \text{dacă evenimentul este relevant,} \\ 0, & \text{în caz contrar.} \end{cases} \tag{5.39}$$

Detecția evenimentului trebuie să consume mai puțină energie decât procesarea evitată.

Operand isolation. *Operand isolation* împiedică propagarea modificărilor către un bloc combinațional atunci când rezultatul blocului nu e utilizat — de exemplu, intrările unui multiplicator pot continua să se schimbe chiar dacă ieșirea nu e selectată, producând tranziții interne și consum dinamic. Prin blocarea intrărilor sau păstrarea lor la o valoare constantă, activitatea internă e redusă; tehnica introduce însă logică suplimentară și trebuie aplicată blocurilor suficient de complexe încât economia să depășească energia logicii de izolare.

Reducerea hazardurilor și a glitch-urilor. Într-un circuit combinațional, semnalele pot ajunge la o poartă în momente diferite din cauza întârzierilor de propagare, iar ieșirea poate efectua tranziții temporare (*glitch-uri*) înainte de stabilizarea valorii corecte, consumând energie chiar dacă nu modifică rezultatul final. Reducerea lor se face prin echilibrarea întârzierilor pe căile logice, reorganizarea expresiilor booleene, introducerea registrelor în puncte potrivite, evitarea logicii combinaționale excesiv de adânci și instrumente de sinteză orientate spre consum. Pipeline-ul poate reduce adâncimea logicii combinaționale, dar introduce registre suplimentare comandate de semnalul de ceas — compromis analizat într-o secțiune ulterioară.

Codificarea datelor. Reprezentarea datelor influențează numărul de biți care se modifică între două valori consecutive: într-un contor binar, tranziția de la 7 la 8 ($0111 \rightarrow 1000$) schimbă toți cei patru biți, în timp ce în cod Gray două valori consecutive diferă printr-un singur bit, ceea ce poate reduce activitatea pe anumite magistrale și în contoare. Codificarea nu e întotdeauna avantajoasă — conversia între reprezentări necesită logică suplimentară, iar economiile depind de statistica datelor. Pentru magistrale largi poate fi utilizată și codificarea *bus-invert*: dacă distanța Hamming dintre două cuvinte depășește jumătate din numărul de biți, e transmis complementul împreună cu un bit de control.

Reducerea lățimii datelor. Utilizarea unei precizii mai mari decât cea necesară poate activa mai multe porți, registre și linii de magistrală — un algoritm care necesită valori

în domeniul $0 \ldots 1000$ poate fi reprezentat pe 16 biți, fără a utiliza obligatoriu 32 sau 64. Reducerea lățimii datelor poate diminua capacitatea comutată, dimensiunea memoriei, traficul pe magistrală, timpul unor operații și energia transferurilor, dar pe anumite procesoare operațiile pe tipuri mai mici pot necesita instrucțiuni suplimentare de extindere sau mascare — rezultatul trebuie măsurat pe arhitectura țintă.

Reducerea transferurilor de date. Deplasarea datelor între procesor, memorie și periferice produce activitate pe magistrale și în circuitele de interfață. Tehnicile utile includ reutilizarea datelor locale, procesarea în blocuri, utilizarea registrelor și a memoriilor locale, reducerea copiilor de buffer, evitarea accesărilor externe inutile și comprimarea sau filtrarea datelor înaintea transferului. În numeroase arhitecturi, energia deplasării datelor poate deveni comparabilă sau mai mare decât energia operației aritmetice executate asupra lor [39].

Activarea selectivă a perifericelor. Un periferic nu trebuie menținut activ când nu e utilizat — dezactivarea ceasului reduce activitatea de comutare, iar deconectarea alimentării poate reduce și consumul static. La acest nivel trebuie controlate:

- semnalele de enable;
- ceasurile perifericelor;
- bufferele de intrare și ieșire;
- convertoarele de date;
- referințele analogice;
- interfețele externe.

Clock gating și power gating vor fi analizate separat, deoarece implică mecanisme hardware, condiții de sincronizare și costuri de reactivare.

Alegerea algoritmului Complexitatea algoritmică influențează numărul total de comutări. Pentru un volum mare de date, alegerea unui algoritm cu o complexitate mai redusă poate produce economii importante.

Totuși, complexitatea asimptotică nu descrie singură energia. Trebuie analizate:

- numărul real de operații;
- tipul operațiilor;
- accesările de memorie;
- localitatea datelor;
- ramificațiile;
- posibilitatea utilizării unui accelerator.

Un algoritm cu mai puține operații aritmetice poate consuma mai mult dacă transferă un volum mare de date din memoria externă.

5.4.2 Exemplu de reducere a activității

Exemplu

Un sistem achiziționează 1000 de eșantioane pe secundă. Algoritmul inițial recalculează o statistică asupra ultimelor 100 de valori la fiecare eșantion.
Implementarea directă necesită aproximativ:

$$1000 \cdot 100 = 100\,000$$

operații de acumulare pe secundă.
Prin utilizarea unei sume actualizate incremental:

$$S[k] = S[k-1] + x[k] - x[k-100],$$

sunt necesare numai două operații principale pentru fiecare eșantion:

$$1000 \cdot 2 = 2000$$

operații pe secundă.
Numărul operațiilor este redus de aproximativ 50 de ori. Implementarea necesită însă păstrarea ultimelor 100 de valori într-un buffer circular.

5.4.3 Relația dintre activitate și energie

Dacă tensiunea, frecvența și capacitatea rămân constante, raportul puterilor de comutare este:

$$\frac{P_2}{P_1} = \frac{\alpha_2}{\alpha_1}. \tag{5.40}$$

Dacă activitatea este redusă de la $\alpha_1 = 0{,}4$ la $\alpha_2 = 0{,}25$, rezultă:

$$\frac{P_2}{P_1} = \frac{0{,}25}{0{,}4} = 0{,}625.$$

Puterea de comutare scade cu 37,5%.

Energia totală nu scade obligatoriu cu aceeași valoare, deoarece puterea statică și celelalte componente rămân prezente.

5.4.4 Costul mecanismelor de optimizare

Orice mecanism de reducere a activității introduce un cost:

- logică suplimentară;
- registre de control;
- timp de decizie;
- memorie;
- energie pentru activare și dezactivare;

- complexitate de verificare.

Energia netă economisită este:

$$E_{\text{net}} = E_{\text{evitat}} - E_{\text{mecanism}}. \tag{5.41}$$

Tehnica este avantajoasă numai dacă:

$$E_{\text{evitat}} > E_{\text{mecanism}}. \tag{5.42}$$

Această condiție este importantă pentru blocurile mici sau pentru perioadele scurte de inactivitate.

Validarea reducerii activității. Reducerea teoretică a numărului de operații nu garantează o reducere proporțională a energiei — compilatorul, memoria cache, pipeline-ul și perifericele pot modifica profilul real. Validarea trebuie să includă măsurarea implementării de referință, aplicarea unei singure modificări controlate, măsurarea energiei pentru aceeași sarcină, verificarea timpului de execuție și a rezultatului funcțional, și repetarea măsurătorilor în scenarii reprezentative. Pentru corelarea software-ului cu forma de undă a curentului pot fi utilizați pini GPIO care marchează începutul și sfârșitul unei activități.

5.5 Reducerea frecvenței de ceas

Frecvența de ceas determină rata cu care registrele și blocurile secvențiale ale unui circuit digital își pot actualiza starea; într-un procesor, ea influențează direct numărul maxim de instrucțiuni executabile într-o unitate de timp. Conform modelului CMOS prezentat anterior, puterea de comutare este:

$$P_{\text{comutare}} = \alpha C_{\text{ef}} V_{DD}^2 f. \tag{5.43}$$

Dacă tensiunea, capacitatea și activitatea rămân constante, rezultă:

$$P_{\text{comutare}} \propto f. \tag{5.44}$$

Reducerea frecvenței conduce astfel la reducerea aproximativ liniară a puterii dinamice, dar efectul asupra energiei totale depinde de timpul necesar finalizării activității. Figura 5.7 ilustrează efectul reducerii frecvenței asupra puterii și timpului de execuție.

5.5.1 Influența asupra timpului de execuție

Pentru o activitate care necesită N cicluri de ceas, timpul ideal de execuție este:

$$T_{\text{exec}} = \frac{N}{f}. \tag{5.45}$$

Dacă frecvența e redusă de la f_1 la f_2, iar numărul de cicluri rămâne constant, raportul timpilor este:

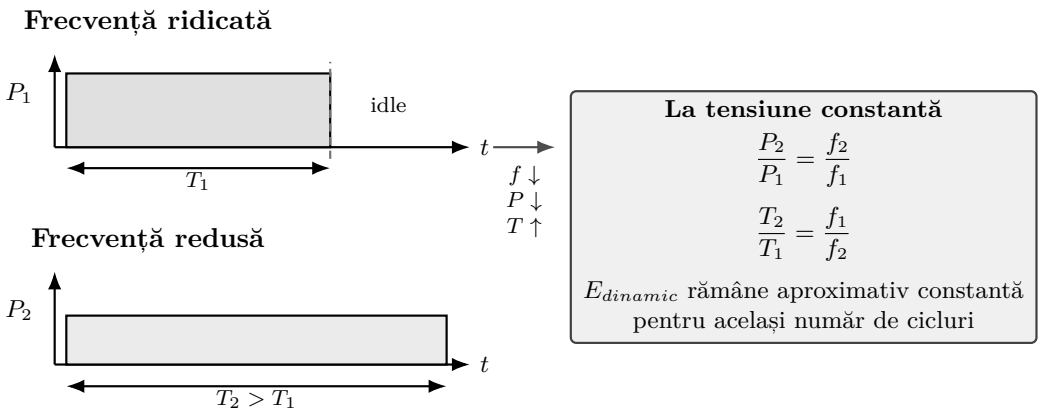


Figura 5.7 Efectul reducerii frecvenței asupra puterii și timpului de execuție.

$$\frac{T_2}{T_1} = \frac{f_1}{f_2}. \tag{5.46}$$

Reducerea frecvenței la jumătate conduce, într-un model ideal, la dublarea timpului de execuție. În practică, timpul nu e întotdeauna invers proporțional cu frecvența — accesările memoriei externe, transferurile prin periferice și operațiile de intrare-ieșire pot avea durate care nu se modifică odată cu frecvența procesorului. Timpul total poate fi descris prin:

$$T_{\text{total}} = T_{\text{CPU}}(f) + T_{\text{memorie}} + T_{\text{I/O}} + T_{\text{așteptare}}. \tag{5.47}$$

Reducerea frecvenței influențează în principal termenul $T_{\text{CPU}}(f)$ — dacă aplicația petrece o mare parte din timp așteptând un periferic, scăderea frecvenței poate avea un efect redus asupra duratei totale.

5.5.2 Influența asupra energiei dinamice

Energia dinamică asociată activității e:

$$E_{\text{dinamic}} = P_{\text{comutare}} T_{\text{exec}}. \tag{5.48}$$

Înlocuind relațiile anterioare:

$$\begin{aligned} E_{\text{dinamic}} &= \alpha C_{\text{ef}} V_{DD}^2 f \frac{N}{f} \\ &= \alpha C_{\text{ef}} V_{DD}^2 N. \end{aligned} \tag{5.49}$$

În modelul idealizat, frecvența se simplifică: pentru o tensiune constantă și un număr fix de cicluri, reducerea frecvenței scade puterea, dar nu reduce în mod obligatoriu energia dinamică necesară realizării activității. Acest rezultat explică diferența dintre *low power* (reducerea puterii instantanee) și *low energy* (reducerea energiei totale). Reducerea frecvenței e

foarte utilă pentru limitarea puterii și a temperaturii, dar economiile energetice importante apar în special când e redusă și tensiunea de alimentare.

5.5.3 Influența puterii statice

Puterea statică rămâne prezentă pe toată durata execuției:

$$E_{\text{static}} = P_{\text{static}} T_{\text{exec}}. \tag{5.50}$$

Pentru o activitate cu N cicluri:

$$E_{\text{static}} = P_{\text{static}} \frac{N}{f}. \tag{5.51}$$

Reducerea frecvenței prelungește timpul în care circuitul rămâne alimentat și poate crește energia statică. Energia totală simplificată este:

$$E_{\text{total}} = \alpha C_{\text{ef}} V_{DD}^2 N + P_{\text{static}} \frac{N}{f} + E_{\text{alte}}. \tag{5.52}$$

Termenul E_{alte} poate include energia memoriei, a perifericelor, a regulatorului și a comunicațiilor.

5.5.4 Exemplu numeric

Exemplu

Un procesor execută o activitate care necesită:

$$N = 100 \cdot 10^6$$

cicluri.
În prima configurație:

$$f_1 = 200\,\text{MHz}, \qquad P_{\text{dinamic},1} = 120\,\text{mW}.$$

Timpul de execuție este:

$$T_1 = \frac{100 \cdot 10^6}{200 \cdot 10^6} = 0{,}5\,\text{s}.$$

Energia dinamică este:

$$E_{\text{dinamic},1} = 120\,\text{mW} \cdot 0{,}5\,\text{s} = 60\,\text{mJ}.$$

Frecvența este redusă la:

$$f_2 = 100\,\text{MHz}.$$

Presupunând aceeași tensiune, puterea dinamică devine aproximativ:

$$P_{\text{dinamic},2} = 60\,\text{mW}.$$

Timpul de execuție se dublează:

$$T_2 = 1\,\text{s}.$$

Energia dinamică este:

$$E_{\text{dinamic},2} = 60\,\text{mW} \cdot 1\,\text{s} = 60\,\text{mJ}.$$

Puterea dinamică a fost redusă la jumătate, dar energia dinamică a rămas aproximativ constantă.

Dacă procesorul consumă suplimentar o putere statică de 10 mW, energia statică este:

$$E_{\text{static},1} = 10\,\text{mW} \cdot 0{,}5\,\text{s} = 5\,\text{mJ},$$

respectiv:

$$E_{\text{static},2} = 10\,\text{mW} \cdot 1\,\text{s} = 10\,\text{mJ}.$$

Energia totală crește de la:

$$E_{\text{total},1} = 65\,\text{mJ}$$

la:

$$E_{\text{total},2} = 70\,\text{mJ}.$$

Exemplul arată că reducerea frecvenței fără reducerea tensiunii poate conduce chiar la creșterea energiei totale.

5.5.5 Frecvența minimă necesară

Într-un sistem cu deadline, frecvența trebuie aleasă astfel încât activitatea să fie finalizată la timp. Dacă o activitate necesită N cicluri și are deadline-ul relativ D, frecvența minimă ideală este:

$$f_{\min} = \frac{N}{D}. \tag{5.53}$$

În practică e necesară o rezervă pentru întreruperi, variația timpului de execuție, accesări imprevizibile ale memoriei, activități ale sistemului de operare și latența schimbării frecvenței. Frecvența selectată poate fi:

$$f_{\text{selectat}} = M_f \frac{N_{\max}}{D}, \tag{5.54}$$

unde $M_f > 1$ reprezintă o marjă de siguranță.

5.5.6 Race to idle și pace to idle

Două strategii generale pot fi utilizate pentru executarea unei activități. În *race to idle*, procesorul rulează la frecvență ridicată, finalizează rapid activitatea și intră în repaus; în *pace to idle*, procesorul rulează la frecvență mai mică și utilizează o parte mai mare din intervalul disponibil. Race to idle e avantajoasă dacă puterea în repaus e foarte redusă, timpul de tranziție spre repaus e mic, funcționarea rapidă nu cere o creștere excesivă a tensiunii și celelalte componente pot fi dezactivate după finalizarea activității. Pace to idle e avantajoasă dacă tensiunea poate fi redusă semnificativ, puterea de repaus nu e mult mai mică decât cea activă, există constrângeri termice sau de putere maximă, ori tranzițiile între stări sunt costisitoare. Alegerea trebuie realizată pe baza energiei măsurate pentru întregul ciclu.

Limitările reducerii frecvenței. Reducerea frecvenței poate produce efecte secundare: creșterea latenței, încălcarea deadline-urilor, creșterea energiei statice, menținerea mai îndelungată a memoriei și perifericelor active, modificarea temporizărilor interfețelor și reducerea debitului sistemului. Unele periferice utilizează ceasuri derivate din ceasul procesorului, iar schimbarea frecvenței poate necesita reconfigurarea baud rate-ului, a timerelor, a magistralelor și a convertoarelor.

5.6 Scalarea dinamică a tensiunii și frecvenței

Scalarea dinamică a tensiunii și frecvenței (DVFS, *Dynamic Voltage and Frequency Scaling*) adaptează punctul de funcționare al procesorului la performanța necesară. Reducerea frecvenței permite reducerea puterii dinamice liniar; reducerea tensiunii produce un efect mai important, deoarece puterea de comutare depinde aproximativ de pătratul tensiunii:

$$P_{\text{comutare}} = \alpha C_{\text{ef}} V_{DD}^2 f. \tag{5.55}$$

DVFS selectează simultan o tensiune și o frecvență compatibile cu cerințele aplicației și cu limitele circuitului [18, 22]. Figura 5.8 prezintă conceptual mai multe puncte de funcționare tensiune-frecvență.

DVS, DFS și DVFS. Pot fi diferențiate trei mecanisme: *Dynamic Frequency Scaling* modifică frecvența, *Dynamic Voltage Scaling* modifică tensiunea, iar DVFS le modifică în mod coordonat. Reducerea exclusivă a frecvenței scade puterea dinamică, dar nu produce neapărat o reducere importantă a energiei; reducerea tensiunii e mai eficientă energetic, dar limitează frecvența maximă — de aceea sistemele reale utilizează de regulă perechi validate de tensiune și frecvență.

5.6.1 Relația dintre tensiune și frecvența maximă

Viteza de comutare a tranzistoarelor scade odată cu reducerea tensiunii de alimentare. O aproximație utilizată pentru întârzierea unei porți CMOS este:

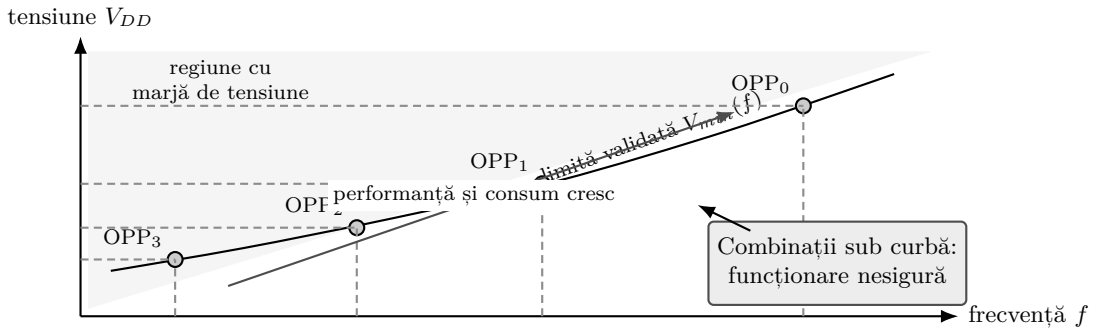


Figura 5.8 Puncte de funcționare utilizate pentru scalarea dinamică a tensiunii și frecvenței.

$$t_{\text{pd}} \propto \frac{V_{DD}}{(V_{DD} - V_T)^{\gamma}}, \tag{5.56}$$

unde:

- V_T este tensiunea de prag;
- γ este un parametru dependent de tehnologie.

Frecvența maximă este aproximativ invers proporțională cu întârzierea:

$$f_{\text{max}} \propto \frac{(V_{DD} - V_T)^{\gamma}}{V_{DD}}. \tag{5.57}$$

Pe măsură ce tensiunea se apropie de tensiunea de prag, frecvența maximă scade rapid [75].

Tensiunea nu poate fi redusă sub limita pentru care circuitul funcționează corect în toate condițiile de proces, temperatură și variație a alimentării.

5.6.2 Puncte de operare

Un punct de operare conține cel puțin:

$$\text{OPP}_i = (V_i, f_i)\,, \tag{5.58}$$

unde OPP provine de la *Operating Performance Point*.

Un exemplu conceptual este prezentat în Tabelul 5.2. Valorile nu reprezintă specificațiile unui procesor concret.

Tabela 5.2 Exemplu conceptual de puncte de operare DVFS

Punct	**Tensiune**	**Frecvență**
OPP$_0$	1,20 V	200 MHz
OPP$_1$	1,05 V	150 MHz
OPP$_2$	0,90 V	100 MHz
OPP$_3$	0,75 V	50 MHz

Punctele sunt validate de producător. Utilizarea unei combinații nevalidate poate produce erori de calcul sau funcționare instabilă.

5.6.3 Efectul DVFS asupra puterii

Pentru două puncte de operare, presupunând aceeași activitate și capacitate, raportul puterilor dinamice este:

$$\frac{P_2}{P_1} = \left(\frac{V_2}{V_1}\right)^2 \frac{f_2}{f_1}. \tag{5.59}$$

Exemplu

Un procesor trece de la:

$$V_1 = 1{,}2\,\mathrm{V}, \qquad f_1 = 200\,\mathrm{MHz}$$

la:

$$V_2 = 0{,}9\,\mathrm{V}, \qquad f_2 = 100\,\mathrm{MHz}.$$

Raportul puterilor dinamice este:

$$\frac{P_2}{P_1} = \left(\frac{0{,}9}{1{,}2}\right)^2 \frac{100}{200}.$$

Rezultă:

$$\frac{P_2}{P_1} = 0{,}28125.$$

Puterea dinamică scade la aproximativ 28,1% din valoarea inițială.

5.6.4 Efectul DVFS asupra energiei

Pentru un număr fix de cicluri, timpul de execuție crește cu raportul invers al frecvenței:

$$\frac{T_2}{T_1} = \frac{f_1}{f_2}. \tag{5.60}$$

Raportul energiilor dinamice este:

$$\begin{aligned} \frac{E_2}{E_1} &= \frac{P_2 T_2}{P_1 T_1} \\ &= \left(\frac{V_2}{V_1}\right)^2. \end{aligned} \tag{5.61}$$

În modelul idealizat, pentru același număr de cicluri, energia dinamică depinde de pătratul raportului tensiunilor. Pentru exemplul anterior:

$$\frac{E_2}{E_1} = \left(\frac{0{,}9}{1{,}2}\right)^2 = 0{,}5625.$$

Energia dinamică scade la aproximativ 56,3% din valoarea inițială. Figura 5.9 prezintă compromisul general dintre tensiune, frecvență, timp și energie.

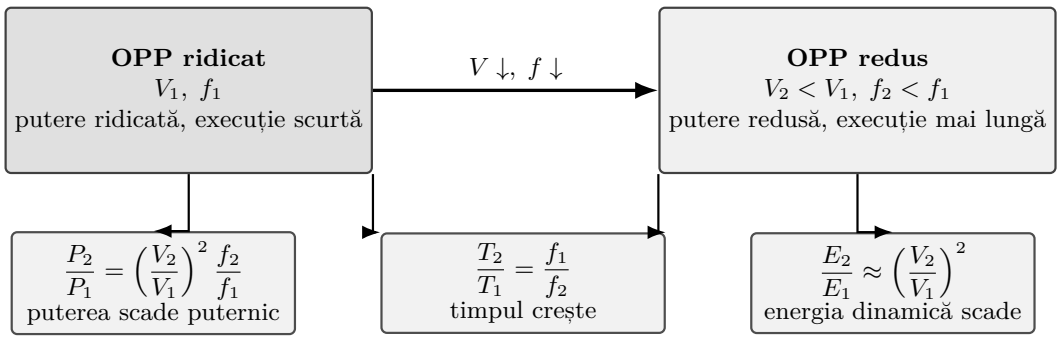


Figura 5.9 Influența DVFS asupra puterii, timpului de execuție și energiei.

5.6.5 Punctul de energie minimă

Reducerea tensiunii nu conduce nelimitat la scăderea energiei totale — la tensiuni foarte mici, timpul de execuție crește, iar energia statică poate deveni dominantă. Energia totală poate fi aproximată prin:

$$E_{\text{total}}(V, f) = \alpha C_{\text{ef}} V^2 N + P_{\text{static}}(V, T)\frac{N}{f} + E_{\text{tranziții}}. \tag{5.62}$$

Primul termen scade odată cu tensiunea, al doilea poate crește din cauza duratei mai mari de execuție, deci poate exista un punct de energie minimă, care depinde de tehnologia circuitului, temperatură, activitatea aplicației, numărul accesărilor de memorie, puterea statică și starea de repaus disponibilă.

5.6.6 Tranziția între punctele de operare

Schimbarea tensiunii și frecvenței nu e instantanee. O tranziție poate presupune reducerea temporară a frecvenței, modificarea tensiunii regulatorului, așteptarea stabilizării alimentării, configurarea sursei de ceas sau a PLL-ului, trecerea la noua frecvență și actualizarea temporizărilor perifericelor. La creșterea performanței, tensiunea trebuie ridicată înaintea frecvenței — altfel circuitul ar putea funcționa temporar la o frecvență prea mare pentru tensiunea disponibilă; la reducerea performanței, frecvența poate fi redusă înaintea tensiunii. Figura 5.10 prezintă secvența conceptuală a unei tranziții DVFS.

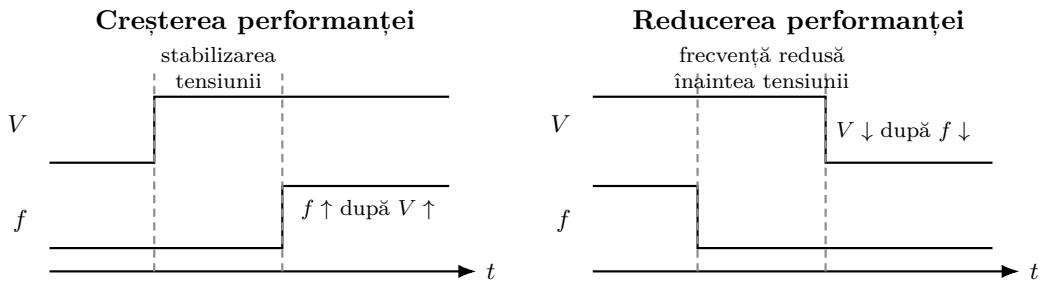


Figura 5.10 Secvența generală a unei tranziții între două puncte DVFS.

5.6.7 Costul tranzițiilor

Tranzițiile consumă timp și energie — dacă punctul de operare e schimbat prea frecvent, economiile pot fi anulate de costul mecanismului. Energia netă economisită este:

$$E_{\text{economie,net}} = E_{\text{evitat}} - E_{\text{tranziții}}. \tag{5.63}$$

DVFS este avantajos numai dacă:

$$E_{\text{evitat}} > E_{\text{tranziții}}. \tag{5.64}$$

Un controler DVFS trebuie să evite oscilațiile rapide între două puncte de operare — pot fi utilizate praguri diferite pentru creșterea și reducerea performanței, introducând astfel histerezis.

Influența memoriei și a perifericelor. Reducerea frecvenței procesorului nu reduce obligatoriu frecvența tuturor componentelor — un sistem poate conține domeniu de ceas pentru procesor, domeniu separat pentru memorie, ceasuri independente pentru periferice, domeniu radio și domeniu permanent activ. Dacă memoria și perifericele continuă să consume aceeași putere, economia obținută prin DVFS la nivelul procesorului poate reprezenta doar o parte din economia sistemului; de asemenea, la frecvențe reduse, accesările memoriei pot reprezenta o fracție mai mare din timpul total de execuție.

Cerințe pentru utilizarea DVFS. Implementarea necesită:

- regulator cu tensiune programabilă;
- sursă de ceas configurabilă;
- puncte de operare validate;
- control al secvenței tensiune–frecvență;
- mecanism software sau hardware de decizie;
- monitorizarea deadline-urilor;
- reconfigurarea temporizărilor dependente de ceas.

Nu toate microcontrolerele permit modificarea dinamică a tensiunii nucleului. Unele platforme oferă numai reducerea frecvenței sau mai multe niveluri fixe de reglare internă.

DVFS în sisteme real-time și utilizarea slack-ului temporal vor fi analizate într-un bloc

ulterior.

5.7 Clock Gating

Semnalul de ceas produce o activitate intensă deoarece este distribuit către un număr mare de registre și comută periodic, chiar dacă unele blocuri nu efectuează o activitate utilă.

Clock Gating întrerupe propagarea ceasului către un bloc care nu este utilizat. Registrele acelui bloc nu mai primesc fronturi de ceas, iar activitatea dinamică este redusă.

Figura 5.11 prezintă principiul general al tehnicii.

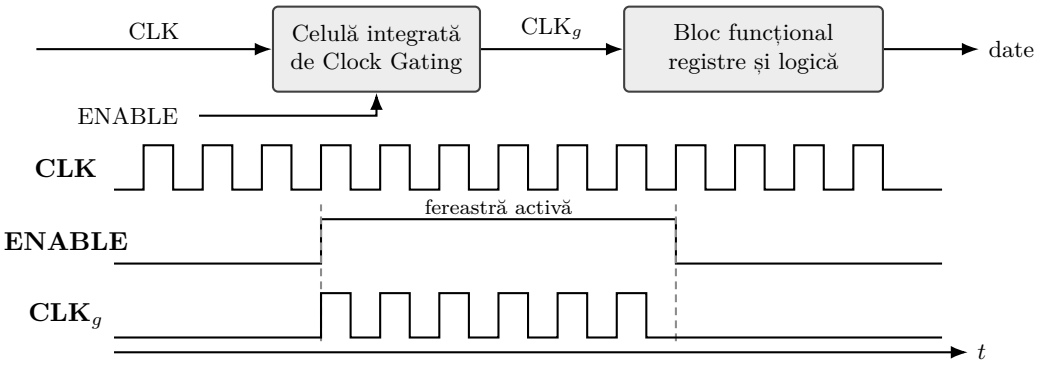


Figura 5.11 Principiul Clock Gating pentru un bloc funcțional.

5.7.1 Rețeaua de distribuție a ceasului

Rețeaua de ceas include oscilatorul, PLL-ul, divizoarele, bufferele, traseele de distribuție și intrările de ceas ale registrelor; poate comuta la fiecare ciclu și poate avea o capacitate echivalentă importantă. Puterea sa poate fi aproximată prin:

$$P_{\text{ceas}} = C_{\text{ceas}} V_{DD}^2 f. \tag{5.65}$$

Factorul de activitate e apropiat de valoarea maximă, deoarece semnalul de ceas efectuează tranziții permanent în modul activ.

Poarta de ceas. O implementare simplă ar putea utiliza o poartă logică AND:

$$\text{CLK}_{\text{gated}} = \text{CLK} \wedge \text{ENABLE}. \tag{5.66}$$

Modificarea semnalului ENABLE în timp ce ceasul e activ poate produce impulsuri scurte sau fronturi suplimentare, care pot modifica accidental registrele. O celulă integrată de Clock Gating (ICG) memorează semnalul ENABLE într-o fază sigură a ceasului și previne apariția glitch-urilor; semnalul de control trebuie să respecte cerințele temporale ale celulei de gating.

5.7.2 Efectul asupra puterii

Considerăm un bloc activ pentru fracția d din timp. Dacă puterea dinamică e neglijabilă când ceasul e oprit, puterea medie idealizată este:

$$P_{\text{medie}} \approx dP_{\text{activ}} + (1-d)\,P_{\text{static}}. \tag{5.67}$$

Clock Gating reduce comutarea registrelor, comutarea logicii comandate de registre și activitatea ramurii locale a rețelei de ceas, dar nu elimină direct curenții de pierderi, consumul alimentării blocului, consumul domeniilor analogice sau consumul memoriei păstrate active.

5.7.3 Exemplu numeric

Exemplu

Un periferic consumă:

$$P_{\text{activ}} = 15\,\text{mW}.$$

Puterea sa statică este:

$$P_{\text{static}} = 0{,}8\,\text{mW}.$$

Perifericul este utilizat 10% din timp.
Fără Clock Gating, puterea medie este aproximativ:

$$P_{\text{fr}} = 15\,\text{mW}.$$

Cu Clock Gating:

$$P_{\text{cu}} = 0{,}10 \cdot 15 + 0{,}90 \cdot 0{,}8.$$

Rezultă:

$$P_{\text{cu}} = 2{,}22\,\text{mW}.$$

Reducerea este:

$$S_P = \frac{15 - 2{,}22}{15} \cdot 100\% \approx 85{,}2\%.$$

Puterea nu devine zero în intervalul inactiv deoarece blocul rămâne alimentat.

5.7.4 Clock Gating la nivelul microcontrolerului

Microcontrolerele permit frecvent activarea și dezactivarea ceasului pentru periferice prin registre de control. Un exemplu generic este:

Exemplu 5.1 Controlul generic al ceasului unui periferic

```
1 /* Activarea ceasului pentru periferic. */
```

```
2  CLOCK_ENABLE_REGISTER |= PERIPHERAL_CLOCK_MASK;
3
4  /* Configurarea si utilizarea perifericulului. */
5  peripheral_init();
6  peripheral_transfer();
7
8  /* Asteptarea finalizarii transferului. */
9  while (peripheral_busy())
10 {
11     /* Asteptare sau intrare intr-un mod de repaus. */
12 }
13
14 /* Dezactivarea perifericulului si a ceasului. */
15 peripheral_disable();
16 CLOCK_ENABLE_REGISTER &= ~PERIPHERAL_CLOCK_MASK;
```

Ordinea operațiilor e importantă — ceasul nu trebuie oprit în timpul unei tranzacții, iar starea perifericului trebuie verificată înaintea dezactivării; denumirile registrelor și secvența exactă depind de microcontroler.

Gating la nivel fin și la nivel grosier. Clock Gating poate fi aplicat la niveluri diferite: gating-ul fin poate opri un registru, o unitate aritmetică, o etapă de pipeline sau o ramură locală a rețelei de ceas, iar gating-ul grosier poate opri un periferic, un accelerator, un nucleu sau un subsistem. Gating-ul fin oferă un control mai precis, dar introduce mai multe celule, semnale de control și dificultăți de verificare.

Clock Gating automat. Instrumentele de sinteză pot identifica registre al căror conținut e păstrat pentru anumite condiții și pot introduce automat celule de Clock Gating. La nivel RTL, o descriere conceptuală de tipul

dacă ENABLE, registru ← valoare nouă

poate fi implementată prin multiplexor la intrarea registrului, celulă de Clock Gating sau semnal de enable integrat în registru — alegerea depinde de tehnologie, constrângerile temporale și politica de sinteză.

Costuri și limitări. Clock Gating introduce celule suplimentare, logică de control, constrângeri temporale, complexitate pentru testare, posibilitatea unor erori de secvențiere și energie pentru comanda mecanismului. Tehnica trebuie utilizată când blocul rămâne inactiv suficient timp pentru a compensa costul controlului.

5.8 Power Gating

Clock Gating reduce activitatea dinamică, dar blocul rămâne conectat la alimentare și continuă să prezinte curenți de pierderi. *Power Gating* deconectează alimentarea unui bloc prin tranzistoare de putere controlate, reducând atât puterea dinamică, cât și o parte

importantă a puterii statice. Figura 5.12 prezintă elementele principale ale unui domeniu cu Power Gating.

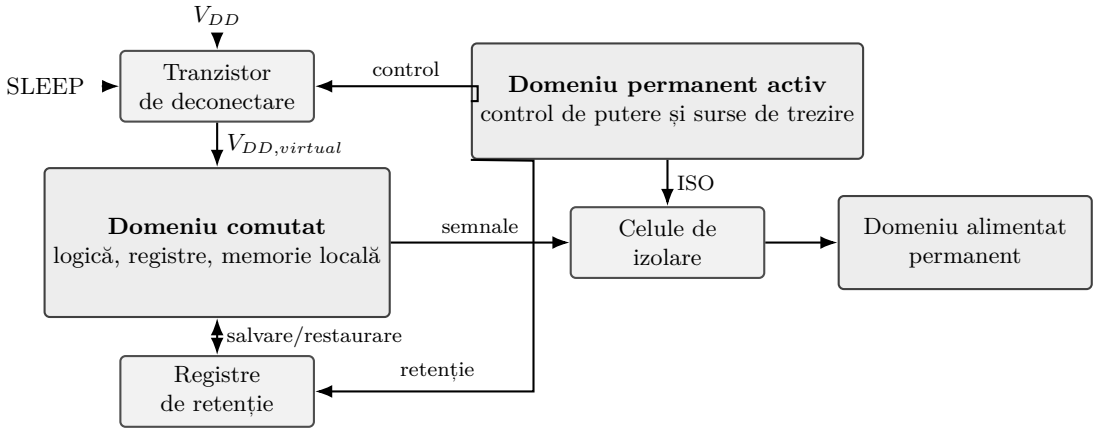


Figura 5.12 Structura conceptuală a unui domeniu controlat prin Power Gating.

Tranzistoarele de deconectare. Alimentarea poate fi întreruptă printr-un tranzistor de tip header (între V_{DD} și bloc), de tip footer (între bloc și masă) sau o combinație — denumite frecvent *sleep transistors*. În modul activ, tranzistorul trebuie să aibă o rezistență suficient de mică pentru a evita o cădere importantă de tensiune; în modul oprit, trebuie să limiteze curentul de pierderi. Dimensionarea lor e un compromis între suprafață, căderea de tensiune, curent maxim, timp de reactivare și curent de pierderi.

Domeniul virtual de alimentare. Tranzistorul de power gating creează o alimentare virtuală pentru blocul controlat: activ, $V_{\text{virtual}} \approx V_{DD}$; oprit, tensiunea virtuală scade și blocul nu mai funcționează normal. Semnalele care pleacă din domeniul oprit pot avea niveluri nedefinite, de aceea sunt necesare celule de izolare.

Celulele de izolare forțează ieșirile domeniului oprit la o valoare logică sigură (0, 1 sau determinată de interfață). Izolarea trebuie activată înaintea întreruperii alimentării și eliminată doar după stabilizarea domeniului la reactivare. O secvență conceptuală de oprire e: oprirea activităților blocului, salvarea stării necesare, activarea izolării, oprirea ceasului, deconectarea alimentării.

Păstrarea stării. Într-un domeniu oprit, starea registrelor obișnuite e pierdută. Soluțiile includ reinițializarea completă la pornire, salvarea stării într-o memorie permanent alimentată, registre de retenție sau păstrarea unui context minimal. Registrele de retenție conțin o parte alimentată dintr-un domeniu permanent activ — starea e salvată înaintea opririi și restaurată la reactivare. Retenția reduce timpul de revenire, dar crește suprafața și consumul domeniului permanent activ.

Domeniul permanent activ. Un sistem cu Power Gating necesită de regulă un domeniu care rămâne alimentat, conținând controlerul de putere, sursele de trezire, timerul de consum redus, registrele de retenție, logica de reset și monitorizarea tensiunii. Consumul acestui domeniu stabilește limita minimă a puterii sistemului în starea profundă de repaus.

Secvența de reactivare trebuie realizată într-o ordine controlată: conectarea alimentării, așteptarea stabilizării tensiunii, pornirea surselor de ceas, resetarea sau restaurarea stării, eliminarea izolării, reluarea activității. Activarea simultană a unui domeniu mare poate produce un curent de pornire ridicat, pentru limitarea căruia tranzistoarele de alimentare pot fi activate gradual.

5.8.1 Energia și timpul de break-even

Intrarea și ieșirea din starea oprită necesită energie:

$$E_{\text{tranziție}} = E_{\text{oprire}} + E_{\text{pornire}} + E_{\text{salvare}} + E_{\text{restaurare}}. \tag{5.68}$$

Dacă blocul ar consuma P_{idle} fără Power Gating și P_{off} în starea oprită, timpul de rentabilizare este:

$$T_{\text{BE}} = \frac{E_{\text{tranziție}}}{P_{\text{idle}} - P_{\text{off}}}. \tag{5.69}$$

Power Gating este avantajos dacă:

$$T_{\text{inactiv}} > T_{\text{BE}}. \tag{5.70}$$

Exemplu

Un bloc consumă în starea alimentată, dar inactivă:

$$P_{\text{idle}} = 8\,\text{mW}.$$

În starea Power Gated consumă:

$$P_{\text{off}} = 0{,}1\,\text{mW}.$$

Energia totală a opririi și reactivării este:

$$E_{\text{tranziție}} = 0{,}4\,\text{mJ}.$$

Timpul de break-even este:

$$T_{\text{BE}} = \frac{0{,}4\,\text{mJ}}{8\,\text{mW} - 0{,}1\,\text{mW}} \approx 50{,}6\,\text{ms}.$$

Pentru o pauză mai scurtă de aproximativ 51 ms, Power Gating nu produce o economie netă în modelul considerat.

5.8.2 Clock Gating versus Power Gating

Figura 5.13 compară principalele efecte ale celor două tehnici.

Cele două tehnici sunt complementare — un sistem poate aplica mai întâi Clock Gating pentru o perioadă scurtă de inactivitate și poate trece ulterior la Power Gating dacă pauza

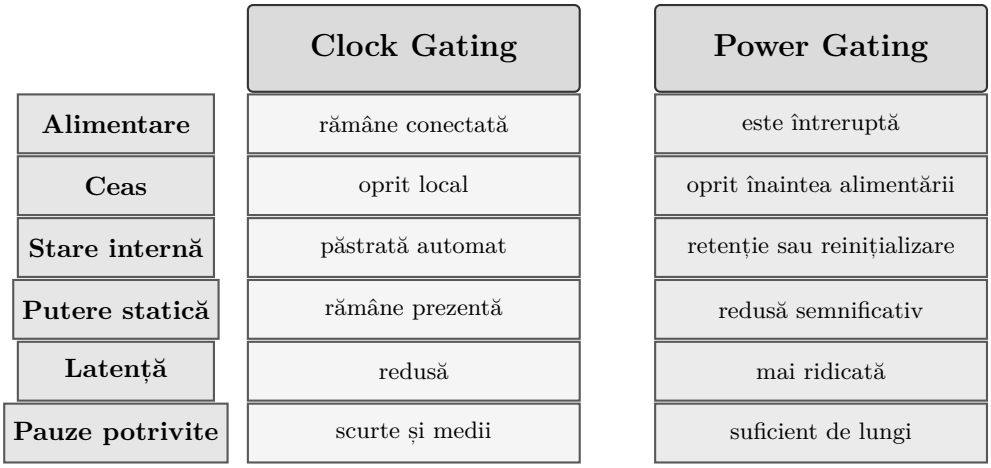


Figura 5.13 Comparație între Clock Gating și Power Gating.

Tabela 5.3 Caracteristicile Clock Gating și Power Gating

Caracteristică	Clock Gating	Power Gating
Alimentarea blocului	Rămâne activă	Este întreruptă parțial sau complet
Semnalul de ceas	Este oprit	Este oprit înaintea deconectării
Puterea dinamică	Este redusă semnificativ	Este eliminată pentru domeniul oprit
Puterea statică	Rămâne prezentă	Este redusă semnificativ
Păstrarea stării	Este realizată automat	Necesită retenție sau reinițializare
Latența de revenire	Redusă	Mai ridicată
Complexitate	Moderată	Ridicată
Interval potrivit	Pauze scurte și medii	Pauze suficient de lungi

continuă.

5.8.3 Power Gating în microcontrolere

La nivelul unui microcontroler, Power Gating poate apărea sub forma modurilor Deep Sleep, Standby, Shutdown, Hibernate sau Backup, în care pot fi deconectate nucleul procesorului, memoria cache, perifericele rapide, PLL-ul, domeniul radio sau anumite regiuni SRAM. Fișa tehnică trebuie consultată pentru a determina ce regiuni rămân alimentate, ce informații sunt păstrate, sursele de trezire disponibile, timpul de reactivare, curentul tipic și maxim și secvența de configurare. Denumirile modurilor nu sunt standardizate — două dispozitive pot utiliza aceeași denumire pentru stări energetice diferite.

Impactul asupra fiabilității și verificării. Introducerea mai multor domenii de ali-

mentare complică verificarea sistemului — trebuie analizate ordinea semnalelor de control, comportamentul la întreruperea alimentării, resetarea incompletă, accesarea unui domeniu oprit, starea semnalelor la interfețe, restaurarea corectă a contextului și comportamentul la treziri simultane. Erorile de Power Gating pot produce blocări dificil de reprodus, mai ales dacă apar doar pentru anumite secvențe de oprire și trezire.

5.8.4 Alegerea tehnicii potrivite

Alegerea între funcționare activă, Clock Gating și Power Gating trebuie realizată pe baza duratei estimate a inactivității. O politică simplificată poate fi:

$$\text{stare} = \begin{cases} \text{activ}, & T_{\text{inactiv}} < T_1, \\ \text{clock gated}, & T_1 \leq T_{\text{inactiv}} < T_2, \\ \text{power gated}, & T_{\text{inactiv}} \geq T_2. \end{cases} \tag{5.71}$$

Pragurile T_1 și T_2 depind de energia tranzițiilor, latența acceptată, puterea statică, necesitatea păstrării stării și probabilitatea unei treziri rapide.

5.9 Paralelismul și eficiența energetică

Paralelismul e utilizat în mod obișnuit pentru creșterea performanței, dar poate contribui și la reducerea energiei consumate: ideea fundamentală e distribuirea activității între mai multe resurse de calcul, astfel încât fiecare să poată funcționa la o frecvență și, eventual, o tensiune mai reduse.

La prima vedere, activarea mai multor nuclee sau unități funcționale pare să conducă inevitabil la creșterea consumului, dar rezultatul energetic depinde de tensiunea și frecvența fiecărei resurse, gradul de paralelizare al algoritmului, timpul total de execuție, costul sincronizării, traficul suplimentar de memorie, puterea statică a resurselor active și posibilitatea revenirii rapide într-un mod de repaus. Figura 5.14 prezintă comparația conceptuală dintre execuția serială la frecvență ridicată și execuția paralelă la frecvență redusă.

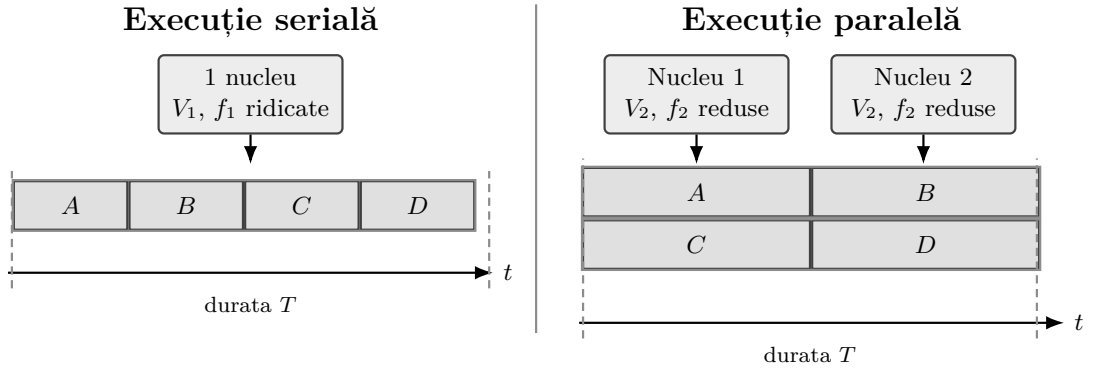


Figura 5.14 Utilizarea paralelismului pentru reducerea tensiunii și frecvenței.

Tipuri de paralelism. În sistemele embedded pot fi întâlnite mai multe forme de

paralelism: la nivel de instrucțiuni (execuția simultană a mai multor instrucțiuni independente, exploatată prin pipeline, procesoare superscalare și arhitecturi VLIW), la nivel de date (aceeași operație aplicată asupra mai multor elemente, precum SIMD și acceleratoarele pentru procesarea semnalelor), la nivel de sarcini (distribuirea unor activități diferite către mai multe nuclee sau procesoare), de tip pipeline (procesarea simultană a unor date diferite în etape succesive) și eterogen (resurse specializate, precum un nucleu general, un DSP, un accelerator criptografic și o unitate pentru rețele neuronale). Fiecare formă are costuri energetice și limitări diferite.

5.9.1 Modelul ideal al execuției paralele

Considerăm o activitate care necesită N operații și e executată de un singur nucleu la frecvența f_1. Într-un model simplificat:

$$T_1 = \frac{N}{r_1}, \tag{5.72}$$

unde r_1 e rata de procesare a nucleului. Dacă activitatea e împărțită perfect între p resurse identice:

$$T_p = \frac{T_1}{p}. \tag{5.73}$$

Accelerarea ideală este:

$$S_p = \frac{T_1}{T_p} = p. \tag{5.74}$$

În practică apar costuri de distribuire, sincronizare și comunicare:

$$T_p = \frac{T_{\text{paralelizabil}}}{p} + T_{\text{serial}} + T_{\text{sincronizare}} + T_{\text{comunicație}}. \tag{5.75}$$

Din acest motiv, accelerarea reală e mai mică decât numărul resurselor.

5.9.2 Legea lui Amdahl

Legea lui Amdahl descrie limita de accelerare produsă de paralelizare [3]. Dacă fracția paralelizabilă a programului e q, iar fracția serială $1 - q$, accelerarea pe p procesoare este:

$$S(p) = \frac{1}{(1-q) + \dfrac{q}{p}}. \tag{5.76}$$

Pentru $p \to \infty$:

$$S_{\max} = \frac{1}{1-q}. \tag{5.77}$$

Dacă 10% din activitate e obligatoriu serială, $q = 0{,}9$, iar accelerarea maximă este:

$$S_{\max} = \frac{1}{0{,}1} = 10.$$

Creșterea numărului de nuclee peste această limită nu poate elimina partea serială. Figura 5.15 prezintă efectul fracției seriale asupra accelerării și eficienței utilizării resurselor.

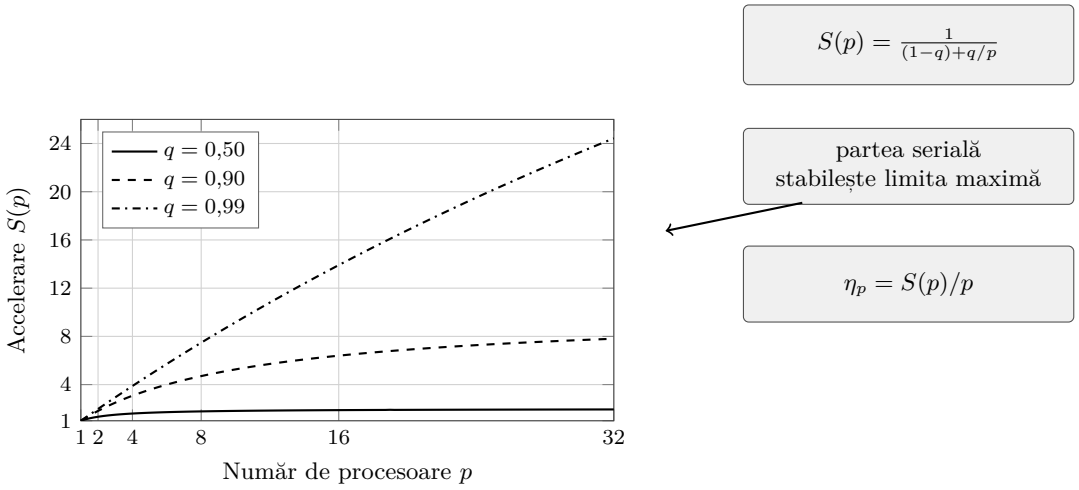


Figura 5.15 Limitarea paralelismului prin partea serială a aplicației.

Eficiența paralelă este:

$$\eta_p = \frac{S(p)}{p}. \tag{5.78}$$

O eficiență redusă indică existența unor resurse active care nu contribuie proporțional la reducerea timpului de execuție.

5.9.3 Modelul energetic al execuției paralele

Puterea dinamică totală a p nuclee identice poate fi aproximată prin:

$$P_{\text{dinamic},p} = p\alpha C_{\text{ef}} V_p^2 f_p. \tag{5.79}$$

Puterea statică totală este:

$$P_{\text{static},p} = pV_p I_{\text{leak},p}. \tag{5.80}$$

Energia totală este:

$$E_p = \left(P_{\text{dinamic},p} + P_{\text{static},p} + P_{\text{memorie},p}\right) T_p + E_{\text{sincronizare}}. \tag{5.81}$$

Paralelismul este avantajos energetic dacă:

$$E_p < E_1. \tag{5.82}$$

Simpla reducere a timpului de execuție nu garantează îndeplinirea acestei condiții.

5.9.4 Reducerea tensiunii prin paralelism

Un avantaj important al paralelismului e posibilitatea menținerii debitului la o frecvență mai mică. Considerăm un nucleu care execută o activitate la:

$$V_1 = 1{,}2\,\mathrm{V}, \qquad f_1 = 1\,\mathrm{GHz}.$$

Activitatea este distribuită ideal între două nuclee care funcționează la:

$$V_2 = 0{,}9\,\mathrm{V}, \qquad f_2 = 500\,\mathrm{MHz}.$$

Raportul puterilor dinamice totale este:

$$\begin{aligned} \frac{P_2}{P_1} &= 2\left(\frac{0{,}9}{1{,}2}\right)^2 \frac{0{,}5}{1} \\ &= 0{,}5625. \end{aligned} \tag{5.83}$$

În modelul ideal, cele două nuclee păstrează același debit, iar puterea dinamică totală scade la aproximativ 56,3% din valoarea inițială. Economia provine din reducerea tensiunii — dacă tensiunea ar rămâne constantă, două nuclee la jumătatea frecvenței ar avea aproximativ aceeași putere dinamică totală ca un singur nucleu la frecvența maximă.

5.9.5 Costurile paralelismului

Execuția paralelă introduce resurse și activități suplimentare: fire de execuție, buffere, semafoare și bariere, cozi de mesaje, mecanisme de coerență, interconectări, transferuri suplimentare de memorie și controlere pentru distribuirea sarcinilor. Energia mecanismelor suplimentare poate fi exprimată prin:

$$E_{\text{suplimentar}} = E_{\text{sincronizare}} + E_{\text{comunicație}} + E_{\text{coerent}} + E_{\text{control}}. \tag{5.84}$$

Pentru activități foarte scurte, acest cost poate depăși energia economisită prin reducerea timpului de execuție.

5.9.6 Dezechilibrarea sarcinii

Dacă activitatea nu e distribuită uniform, unele nuclee finalizează execuția mai devreme și așteaptă finalizarea celui mai lent. Timpul total e determinat de:

$$T_p = \max\left(T_1, T_2, \ldots, T_p\right) + T_{\text{sincronizare}}. \tag{5.85}$$

În timpul așteptării, nucleele inactive continuă să consume putere statică și pot produce activitate în sistemul de operare. Tehnicile de echilibrare includ partiționarea uniformă a

datelor, distribuirea dinamică a activităților, work stealing, migrarea sarcinilor și oprirea nucleelor neutilizate.

Limitarea prin memorie. Creșterea numărului de nuclee nu produce o creștere proporțională a performanței dacă toate resursele accesează aceeași memorie. Lățimea de bandă necesară poate fi aproximată prin:

$$B_{\text{necesar}} = pB_{\text{nucleu}}. \tag{5.86}$$

Dacă memoria nu poate furniza această lățime de bandă, nucleele intră în așteptare, iar eficiența energetică scade. Localitatea datelor, memorii locale și reducerea transferurilor sunt esențiale pentru un avantaj energetic real.

Paralelism omogen și eterogen. Un sistem omogen conține nuclee similare, ceea ce simplifică distribuirea sarcinilor, dar nu oferă întotdeauna eficiența maximă. Un sistem eterogen poate conține nuclee performante, nuclee cu consum redus, DSP, GPU, accelerator criptografic și accelerator pentru inteligență artificială — o activitate trebuie executată pe resursa care oferă energia minimă pentru cerințele sale de performanță.

Paralelismul și puterea statică. Mai multe resurse active înseamnă o suprafață alimentată mai mare și, de regulă, curenți de pierderi mai mari. Paralelismul poate reduce energia statică dacă activitatea e finalizată suficient de repede și resursele sunt apoi deconectate; dacă rămân alimentate permanent, energia statică suplimentară poate anula avantajul. Acest compromis e asociat conceptului de *dark silicon*: nu toate resursele unui circuit complex pot fi utilizate simultan la performanță maximă din cauza limitelor energetice și termice [27].

5.10 Pipeline și eficiența energetică

Pipeline-ul împarte prelucrarea unei activități în mai multe etape succesive — în loc ca o singură unitate combinațională să execute toate operațiile într-un ciclu, rezultatele intermediare sunt memorate în registre. Figura 5.16 prezintă o structură generică formată din mai multe etape.

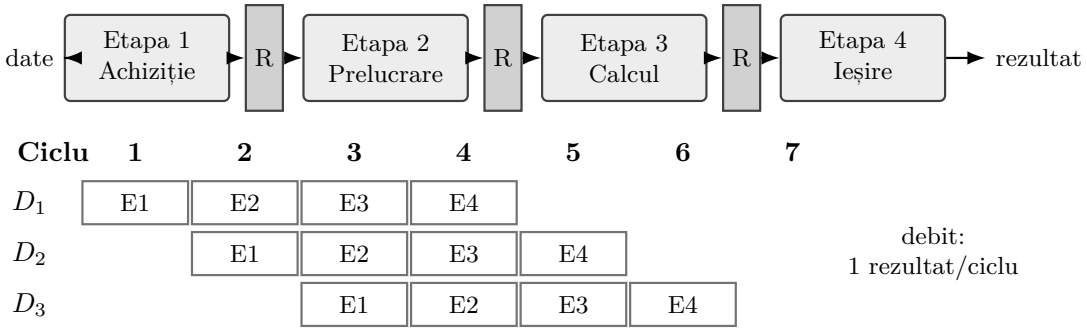


Figura 5.16 Împărțirea unei activități în etape de pipeline.

Pipeline-ul urmărește în principal creșterea debitului, dar poate contribui la reducerea

energiei dacă permite reducerea tensiunii sau utilizarea mai eficientă a resurselor.

5.10.1 Latență și debit

Latența reprezintă timpul dintre introducerea unei date și obținerea rezultatului, iar debitul numărul de rezultate produse într-o unitate de timp. Pentru un pipeline cu k etape și perioada de ceas T_{clk}, latența ideală este:

$$T_{\text{latență}} = kT_{\text{clk}}. \tag{5.87}$$

După umplerea pipeline-ului, poate fi produs un rezultat la fiecare ciclu:

$$R_{\text{pipeline}} \approx \frac{1}{T_{\text{clk}}}. \tag{5.88}$$

Pipeline-ul poate crește debitul chiar dacă latența individuală nu scade.

5.10.2 Calea critică

Perioada minimă de ceas este determinată de cea mai lentă etapă:

$$T_{\text{clk}} \geq \max_i \left(T_{\text{logic},i}\right) + T_{\text{registru}}. \tag{5.89}$$

Termenul T_{registru} include timpul de propagare al registrului, setup time și incertitudinea ceasului. Dacă o cale combinațională lungă e împărțită în mai multe etape, întârzierea maximă a unei etape scade. Circuitul poate:

- funcționa la o frecvență mai mare;
- menține aceeași frecvență la o tensiune mai redusă;
- utiliza porți mai mici și mai lente.

Pentru optimizarea energetică este relevantă a doua posibilitate: reducerea tensiunii la același debit.

5.10.3 Pipeline și reducerea tensiunii

Considerăm un bloc nepipelineizat cu întârzierea critică T_C.

După împărțirea ideală în k etape:

$$T_{\text{logic,etap}} \approx \frac{T_C}{k}. \tag{5.90}$$

În practică, întârzierea registrului și dezechilibrarea etapelor limitează câștigul.

Dacă noua structură poate respecta aceeași frecvență la o tensiune mai mică, puterea dinamică poate fi redusă conform:

$$P_{\text{nou}} \propto C_{\text{nou}} V_{\text{nou}}^2 f. \tag{5.91}$$

Capacitatea C_{nou} crește din cauza registrelor suplimentare. Pipeline-ul este avantajos energetic numai dacă economia produsă prin reducerea tensiunii depășește costul registrelor

și al rețelei de ceas.

5.10.4 Costul registrelor de pipeline

Fiecare etapă suplimentară introduce:

- registre;
- logică de control;
- sarcină suplimentară pentru rețeaua de ceas;
- semnale de validare;
- consum static;
- latență.

Puterea pipeline-ului poate fi exprimată conceptual prin:

$$P_{\text{pipeline}} = P_{\text{logic}} + P_{\text{registre}} + P_{\text{ceas}} + P_{\text{control}}. \tag{5.92}$$

Pe măsură ce numărul etapelor crește, termenii asociați registrelor și ceasului devin mai importanți, deci există un număr optim de etape pentru o anumită tehnologie și aplicație.

Dezechilibrarea etapelor. Frecvența e determinată de etapa cea mai lentă — dacă etapele au întârzieri foarte diferite, unele resurse nu sunt utilizate eficient. Gradul de dezechilibrare poate fi exprimat prin:

$$\Delta T = \max_i T_i - \frac{1}{k}\sum_{i=1}^{k} T_i. \tag{5.93}$$

O valoare mare a lui ΔT indică faptul că împărțirea logicii nu e uniformă. Retiming-ul și redistribuirea operațiilor pot îmbunătăți echilibrarea.

5.10.5 Hazarduri și opriri

Pipeline-ul unui procesor poate fi afectat de dependențe de date, ramificații, accesări lente ale memoriei, conflicte între resurse și întreruperi, situații care pot produce stall-uri sau golirea pipeline-ului. Energia consumată pentru instrucțiunile executate speculativ și apoi eliminate nu contribuie la rezultatul final. Energia per instrucțiune utilă devine:

$$E_{\text{instrucțiune,util}} = \frac{E_{\text{total}}}{N_{\text{instrucțiuni,finalizate}}}. \tag{5.94}$$

O rată mare de ramificații greșit prezise reduce eficiența energetică.

Clock Gating pentru etapele inactive. Etapele care nu conțin date valide pot fi oprite prin Clock Gating. Un semnal de validare poate controla actualizarea registrelor:

$$\text{CLK}_{i,\text{activ}} = \text{CLK} \wedge \text{VALID}_i. \tag{5.95}$$

Implementarea reală trebuie să utilizeze celule sigure de Clock Gating și să respecte cerințele temporale — tehnica e utilă pentru pipeline-uri care funcționează intermitent sau procesează fluxuri cu goluri.

Pipeline elastic. Un pipeline elastic utilizează semnale de validare și confirmare pentru a permite fiecărei etape să avanseze doar când etapa următoare poate accepta date. Avantajele includ evitarea comutărilor inutile, adaptarea la rate variabile, oprirea locală a etapelor și integrarea modulelor cu latențe diferite — dar controlul suplimentar consumă energie și crește complexitatea.

5.10.6 Alegerea adâncimii pipeline-ului

Un pipeline foarte scurt poate avea cale critică lungă, frecvență redusă și necesitatea unei tensiuni ridicate. Un pipeline foarte adânc poate avea multe registre, putere mare a rețelei de ceas, penalizare ridicată la ramificații, latență mai mare și complexitate de control. Alegerea trebuie realizată prin minimizarea unei funcții care poate include energia și timpul:

$$J(k) = E(k) + \lambda T(k), \tag{5.96}$$

unde λ reprezintă importanța performanței în raport cu energia.

5.11 Arhitecturi VLIW și acceleratoare specializate

Arhitecturile VLIW și acceleratoarele specializate urmăresc realizarea unui volum mare de calcul cu un cost energetic redus. VLIW exploatează paralelismul la nivel de instrucțiuni prin programarea statică a mai multor unități funcționale, în timp ce acceleratoarele reduc costul energetic prin adaptarea directă a structurii hardware la o anumită clasă de operații.

5.11.1 Principiul VLIW

VLIW provine de la *Very Long Instruction Word*. O instrucțiune VLIW conține mai multe operații care pot fi lansate simultan către unități funcționale diferite. Figura 5.17 prezintă structura conceptuală a unui procesor VLIW.

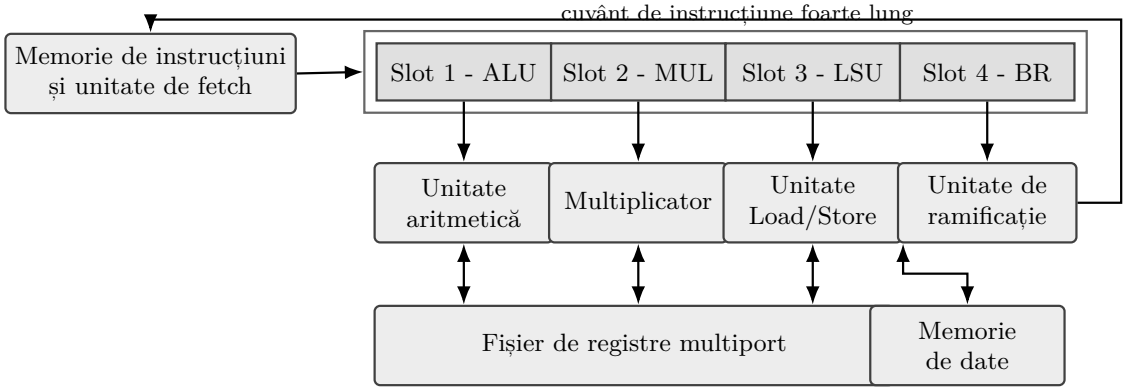


Figura 5.17 Structura conceptuală a unei arhitecturi VLIW.

Un cuvânt de instrucțiune poate conține, de exemplu, o operație aritmetică, o înmulțire, o operație logică și un transfer de memorie:

Exemplu 5.2 Instrucțiune VLIW conceptuală

```
1 ADD R1, R2, R3 | MUL R4, R5, R6 | LOAD R7, [R8]
```

Operațiile sunt independente și pot fi executate în paralel.

Planificarea statică. Într-un procesor superscalar, hardware-ul detectează dinamic dependențele și alege instrucțiunile care pot fi executate în paralel; într-o arhitectură VLIW, această activitate e realizată în principal de compilator, care trebuie să identifice operațiile independente, respecte latențele unităților funcționale, aloce registre, programeze accesările de memorie, trateze ramificațiile și completeze sloturile neutilizate. Reducerea logicii dinamice de planificare poate diminua suprafața și consumul controlului procesorului [29].

Avantajele energetice ale VLIW. Arhitecturile VLIW pot oferi performanță ridicată la o frecvență moderată, logică de control mai simplă decât în execuția superscalară dinamică, utilizarea explicită a unităților specializate, predictibilitate temporală mai bună și posibilitatea aplicării Clock Gating unităților neutilizate. Energia per operație poate fi redusă dacă mai multe unități execută operații utile în același ciclu.

5.11.2 Limitările VLIW

Eficiența depinde de capacitatea compilatorului de a găsi operații independente. Dacă paralelismul disponibil e redus, unele sloturi rămân neutilizate și pot fi completate cu operații NOP. Gradul de utilizare este:

$$U_{\text{VLIW}} = \frac{N_{\text{sloturi,utile}}}{N_{\text{sloturi,totale}}}. \tag{5.97}$$

O utilizare redusă conduce la cod mai mare, trafic mai mare în memoria de instrucțiuni, unități funcționale neutilizate și eficiență energetică redusă. Compatibilitatea binară poate fi dificilă între implementări cu număr diferit de unități sau latențe diferite.

Predicția și execuția speculativă. Pentru menținerea unităților ocupate, unele arhitecturi pot utiliza predicție, execuție condiționată sau software pipelining. Execuția speculativă poate crește performanța, dar operațiile anulate consumă energie fără a contribui la rezultatul final — o optimizare energetică trebuie să compare energia suplimentară a speculației cu timpul evitat.

5.11.3 Acceleratoarele specializate

Un accelerator specializat implementează direct în hardware o funcție sau o clasă restrânsă de algoritmi. Exemplele includ unități DSP, acceleratoare criptografice, controlere DMA, acceleratoare grafice, unități pentru procesarea imaginilor, acceleratoare pentru rețele neuronale și unități pentru codificare/decodificare. Figura 5.18 prezintă integrarea unui accelerator într-un sistem embedded.

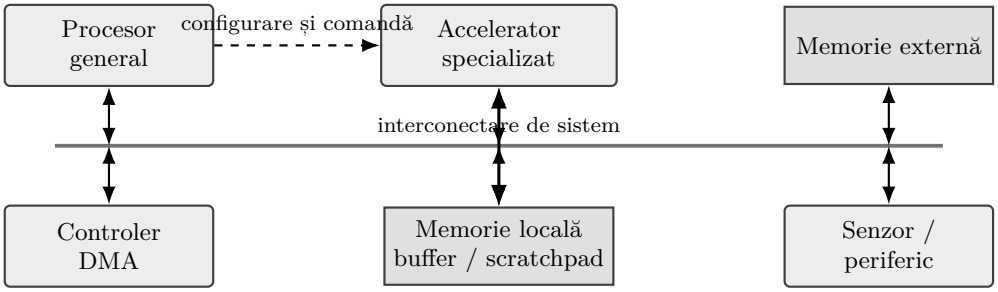


Figura 5.18 Integrarea unui accelerator specializat într-un sistem embedded.

5.11.4 Sursele eficienței energetice

Un accelerator poate reduce energia prin eliminarea preluării și decodificării repetate a instrucțiunilor, utilizarea unor căi de date adaptate operației, reducerea lățimii operanzilor, reutilizarea locală a datelor, evitarea transferurilor către memoria externă, eliminarea logicii generale neutilizate și utilizarea masivă a paralelismului la frecvență redusă. Energia acceleratorului este:

$$E_{\text{acc}} = E_{\text{calcul}} + E_{\text{memorie local}} + E_{\text{transfer}} + E_{\text{control}}. \tag{5.98}$$

Un accelerator nu e eficient dacă datele trebuie transferate frecvent printr-o interfață cu un consum ridicat.

Energia deplasării datelor. În multe aplicații moderne, energia transferului datelor poate depăși energia operației aritmetice [39]. O arhitectură eficientă trebuie să maximizeze reutilizarea locală:

$$R_{\text{reutilizare}} = \frac{N_{\text{operații}}}{N_{\text{accesri memorie extern}}}. \tag{5.99}$$

O valoare mare indică faptul că fiecare valoare transferată e utilizată în mai multe operații — caracteristică importantă în acceleratoarele pentru convoluții, filtrare și procesare matricială [84].

Precizia numerică. Acceleratoarele pot utiliza o precizie adaptată aplicației: 32 de biți în virgulă mobilă, 16 biți, aritmetică fixed-point, 8 biți întregi sau reprezentări binare/ternare. Reducerea preciziei poate diminua suprafața unităților aritmetice, capacitatea comutată, dimensiunea memoriei, traficul de date și energia per operație — dar nu trebuie redusă sub limita impusă de calitatea rezultatului.

Utilizarea acceleratorului. Un accelerator ocupă suprafață și prezintă curenți de pierderi chiar dacă e utilizat rar. Gradul de utilizare poate fi definit prin:

$$U_{\text{acc}} = \frac{T_{\text{activ}}}{T_{\text{total}}}. \tag{5.100}$$

Pentru un accelerator utilizat rar, Power Gating poate fi necesar pentru reducerea energiei

statice.

5.11.5 Condiția de eficiență a acceleratorului

Considerăm energia executării software, E_{SW}. Energia utilizării acceleratorului este:

$$E_{\text{HW}} = E_{\text{configurare}} + E_{\text{transfer}} + E_{\text{execuție}} + E_{\text{reactivare}}. \tag{5.101}$$

Accelerarea este avantajoasă energetic dacă:

$$E_{\text{HW}} < E_{\text{SW}}. \tag{5.102}$$

Pentru volume mici de date, costul configurării și al transferului poate domina.

5.11.6 Exemplu conceptual

Exemplu

Un algoritm criptografic consumă pe procesor:

$$E_{\text{CPU}} = 20\,\mu\text{J}$$

pentru procesarea unui bloc.
Acceleratorul consumă:

$$E_{\text{configurare}} = 2\,\mu\text{J},$$

$$E_{\text{transfer}} = 1\,\mu\text{J},$$

și:

$$E_{\text{calcul}} = 3\,\mu\text{J}.$$

Energia totală a acceleratorului este:

$$E_{\text{acc}} = 2 + 1 + 3 = 6\,\mu\text{J}.$$

Economia energetică este:

$$S_E = \frac{20 - 6}{20} \cdot 100\% = 70\%.$$

Dacă acceleratorul ar fi utilizat pentru un volum foarte mic de date, energia de configurare ar putea reprezenta o parte dominantă.

5.12 Managementul dinamic al puterii

Managementul dinamic al puterii (DPM, *Dynamic Power Management*) controlează starea energetică a resurselor în funcție de activitatea sistemului. Spre deosebire de DVFS, care

adaptează punctul de performanță al unei resurse active, DPM selectează între stări precum activ, idle, sleep, deep sleep și off; cele două tehnici pot fi utilizate împreună.

5.12.1 Modelul bazat pe stări energetice

O resursă poate fi descrisă printr-un set de stări:

$$\mathcal{S} = \{S_0, S_1, \ldots, S_n\}. \tag{5.103}$$

Fiecărei stări îi sunt asociate puterea P_i, nivelul de performanță, latența de trezire, contextul păstrat și sursele de reactivare; trecerea dintre stările S_i și S_j are energia E_{ij} și durata T_{ij}. Figura 5.19 prezintă un automat generic de management energetic.

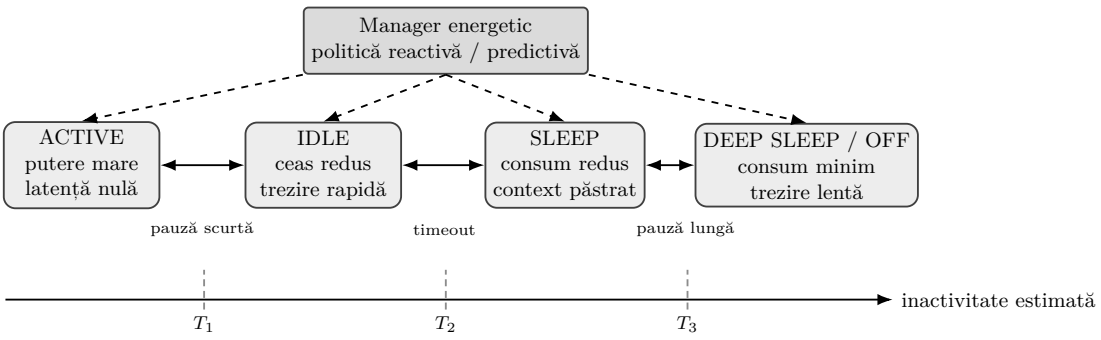


Figura 5.19 Model generic de management dinamic al puterii.

5.12.2 Energia unui profil de stări

Pentru un interval T, energia consumată în stări e:

$$E_{\text{stri}} = \sum_{i=0}^{n} P_i T_i. \tag{5.104}$$

Energia tranzițiilor e:

$$E_{\text{tranziții}} = \sum_{i=0}^{n} \sum_{j=0}^{n} N_{ij} E_{ij}, \tag{5.105}$$

unde N_{ij} e numărul tranzițiilor dintre stările i și j. Energia totală devine:

$$E_{\text{DPM}} = E_{\text{stri}} + E_{\text{tranziții}}. \tag{5.106}$$

O politică ce schimbă foarte frecvent stările poate consuma mai mult decât o politică mai simplă.

5.12.3 Timpul de break-even

Considerăm o stare activă sau idle cu puterea P_A și o stare de repaus cu puterea P_S.

Energia de intrare și ieșire din repaus este:

$$E_{\text{tr}} = E_{A \to S} + E_{S \to A}. \tag{5.107}$$

Timpul minim necesar pentru recuperarea costului tranziției este:

$$T_{\text{BE}} = \frac{E_{\text{tr}}}{P_A - P_S}. \tag{5.108}$$

Starea de repaus este avantajoasă dacă:

$$T_{\text{inactiv}} > T_{\text{BE}}. \tag{5.109}$$

În plus, trebuie verificată latența:

$$T_{\text{trezire}} \leq L_{\max}, \tag{5.110}$$

unde $L_{\max}$ este latența maximă acceptată.

5.12.4 Politici reactive

O politică reactivă decide schimbarea stării după observarea inactivității. Cea mai simplă e politica bazată pe timeout:

$$\text{intrare n sleep} \quad \text{dacă} \quad T_{\text{idle}} \geq T_{\text{timeout}}. \tag{5.111}$$

Avantajele sunt implementarea simplă, costul redus de calcul și comportamentul ușor de verificat; limitările sunt energia consumată în timpul timeout-ului, reacția lentă la perioade lungi de inactivitate, alegerea dificilă a pragului și performanța redusă pentru sarcini neregulate. Timeout-ul trebuie să fie suficient de mare pentru a evita opririle inutile, dar suficient de mic pentru a nu pierde o parte importantă din intervalul de repaus.

5.12.5 Politici predictive

O politică predictivă încearcă să estimeze durata viitoare a inactivității, folosind istoricul cererilor, periodicitatea activității, starea aplicației, evenimente programate, modele statistice sau algoritmi de învățare automată. Starea de repaus poate fi selectată dacă:

$$\widehat{T}_{\text{inactiv}} > T_{\text{BE}}, \tag{5.112}$$

unde $\widehat{T}_{\text{inactiv}}$ e durata estimată. O predicție greșită poate produce reactivare prematură, latență suplimentară, energie de tranziție inutilă sau pierderea unor evenimente.

Politici hibride. O politică hibridă combină reacția la evenimente cu predicția — de exemplu: la începutul inactivității se selectează Idle, după un timp scurt se trece în Sleep, dacă inactivitatea continuă se trece în Deep Sleep, iar pentru un eveniment periodic cunoscut trezirea e programată. Această abordare poate reduce riscul unei predicții incorecte.

Politici bazate pe praguri. Pentru un sistem cu coadă de activități, starea poate fi

aleasă pe baza numărului de cereri, de exemplu:

$$\text{stare} = \begin{cases} \text{performanț ridicat}, & N_q > N_H, \\ \text{performanț redus}, & N_L < N_q \leq N_H, \\ \text{sleep}, & N_q = 0. \end{cases} \tag{5.113}$$

Pragurile diferite pentru activare și dezactivare introduc histerezis și evită oscilațiile rapide.

5.12.6 Coordonarea mai multor componente

Într-un sistem embedded, componentele nu sunt independente. Oprirea procesorului poate necesita finalizarea unui transfer DMA, oprirea memoriei externe, salvarea contextului, configurarea sursei de trezire, suspendarea comunicației și trecerea senzorului în standby — ordinea tranzițiilor e importantă. Un exemplu de oprire: finalizarea operațiilor în curs, salvarea datelor, oprirea perifericelor, dezactivarea surselor rapide de ceas, configurarea timerului de trezire, intrarea procesorului în repaus; la trezire, ordinea e aproximativ inversă.

Management centralizat și distribuit. În managementul centralizat, un singur controler decide starea tuturor componentelor, oferind vedere globală, coordonare simplă și o politică comună, dar cu complexitate și riscul ca decizia centrală să devină un punct critic. În managementul distribuit, fiecare componentă poate controla local anumite stări, fiind necesar un protocol pentru evitarea conflictelor.

5.12.7 Exemplu de politică pentru un periferic

Exemplu

Un periferic are trei stări:

$$P_{\text{activ}} = 20\,\text{mW},$$

$$P_{\text{idle}} = 5\,\text{mW},$$

$$P_{\text{sleep}} = 0{,}2\,\text{mW}.$$

Energia intrării și ieșirii din Sleep este:

$$E_{\text{tr}} = 0{,}3\,\text{mJ}.$$

Timpul de break-even față de Idle este:

$$T_{\text{BE}} = \frac{0{,}3\,\text{mJ}}{5\,\text{mW} - 0{,}2\,\text{mW}} = 62{,}5\,\text{ms}.$$

Pentru pauze estimate sub 62,5 ms, perifericul rămâne în Idle. Pentru pauze mai lungi,

intrarea în Sleep poate produce o economie netă.

5.12.8 Calitatea serviciului

Politica energetică trebuie să respecte cerințele de performanță: latența maximă, deadline-uri, debit minim, pierderi maxime de date, timp de răspuns, disponibilitate. O funcție de cost poate combina energia și penalizarea întârzierilor:

$$J = E + \lambda_L L + \lambda_D N_{\text{deadline ratate}}, \tag{5.114}$$

unde λ_L și λ_D exprimă importanța latenței și a deadline-urilor.

Implementarea software. Un manager energetic poate fi implementat în firmware, în sistemul de operare, într-un controler hardware sau într-o combinație hardware-software. Interfața software trebuie să permită componentelor să indice dacă sunt ocupate, când pot fi suspendate, sursele de trezire necesare, latența maximă acceptată și starea care trebuie păstrată. O componentă nu trebuie oprită cât timp e utilizată de un alt subsistem.

Validarea politicii DPM trebuie realizată pentru profiluri variate: activitate periodică, activitate în rafale, cereri rare, încărcare maximă, treziri neașteptate, variații ale timpului de execuție. Trebuie măsurate energia totală, numărul tranzițiilor, timpul petrecut în fiecare stare, latența de răspuns și numărul deadline-urilor ratate — o politică eficientă pentru o sarcină periodică poate avea rezultate slabe pentru un trafic aleator.

6. Software Power Management și Optimizarea Energetică la Nivel Software

Rolul software-ului în consumul energetic
Optimizarea energetică la nivel de algoritm și cod
Optimizarea accesului la memorie și a transferurilor de date
Power-Aware Software și controlul resurselor
Modele software pentru estimarea consumului
Managementul software al stărilor energetice
Break-Even Time și alegerea stării de consum
Politici software de management energetic
Predicția încărcării și DVFS controlat software
Management energetic în sistemele embedded moderne
Studii de caz
Tendințe actuale în optimizarea energetică software
Întrebări și probleme propuse

6.1 Rolul software-ului în consumul energetic

Atunci când e analizat consumul energetic al unui sistem embedded, atenția e orientată frecvent către componentele hardware: procesor, memorii, convertoare de tensiune, senzori, module de comunicație și actuatoare. Aceste componente stabilesc limitele fizice ale sistemului, însă software-ul determină modul concret în care resursele sunt utilizate.

Două aplicații care rulează pe aceeași platformă hardware pot avea consumuri energetice foarte diferite — diferența nu vine din modificarea componentelor, ci din durata în care acestea rămân active, frecvența cu care sunt utilizate, numărul operațiilor executate și volumul datelor transferate. Software-ul influențează consumul prin alegerea algoritmilor, numărul și tipul instrucțiunilor executate, organizarea datelor în memorie, frecvența accesărilor memoriei, activarea/dezactivarea perifericelor, utilizarea comunicațiilor, alegerea stărilor energetice, controlul frecvenței și tensiunii și planificarea sarcinilor. Figura 6.1 prezintă principalele căi prin care software-ul influențează energia totală a sistemului.

Energia consumată de aplicație poate fi exprimată prin:

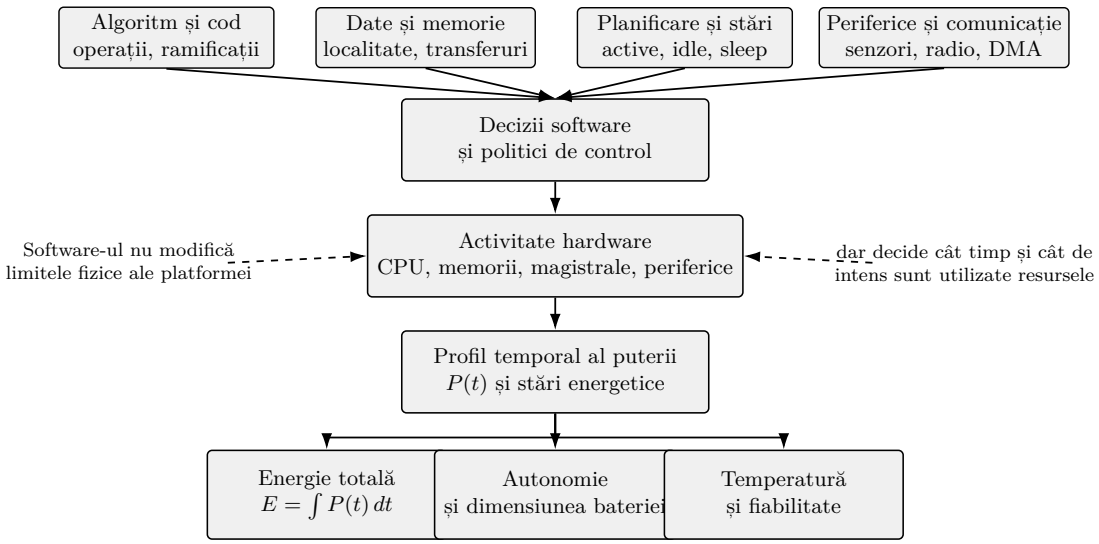


Figura 6.1 Influența software-ului asupra consumului energetic al sistemului.

$$E_{\text{software}} = \int_{t_0}^{t_1} P_{\text{sistem}}(t)\,dt. \tag{6.1}$$

Puterea sistemului nu depinde doar de procesor — în timpul executării unei funcții pot fi active memoria, magistralele, senzorii, convertoarele și interfețele de comunicație. O reprezentare conceptuală este:

$$P_{\text{sistem}} = P_{\text{CPU}} + P_{\text{memorie}} + P_{\text{periferice}} + P_{\text{comunicație}} + P_{\text{conversie}}. \tag{6.2}$$

Software-ul controlează direct sau indirect fiecare termen — de exemplu, o funcție care solicită repetat date de la un senzor menține active procesorul, interfața de comunicație și circuitul de măsurare; o implementare care grupează mai multe achiziții poate reduce numărul activărilor și energia asociată pornirii perifericelor. Consumul energetic al unei aplicații poate fi aproximat și prin energia petrecută în diferite stări:

$$E_{\text{total}} = \sum_{i=1}^{n} P_i T_i + E_{\text{tranziții}}, \tag{6.3}$$

unde P_i e puterea stării i, iar T_i durata acesteia. Într-un nod IoT, stările pot fi pornirea sistemului, achiziția datelor, procesarea, transmisia, așteptarea și modul Deep Sleep.

O optimizare software poate reduce energia în două moduri: diminuarea puterii unei stări (de exemplu prin reducerea frecvenței) sau reducerea timpului petrecut în ea (de exemplu prin alegerea unui algoritm mai rapid și revenirea mai devreme în Sleep). Un program nu consumă energie doar la calcule utile — energia poate fi consumată și pentru așteptare activă, verificări repetate ale unui registru, recalcularea unor valori neschimbate, copierea inutilă a datelor, retransmiterea mesajelor, tratarea frecventă a întreruperilor și menținerea

perifericelor pornite. Aceste activități nu modifică funcționalitatea observată de utilizator, dar influențează autonomia și temperatura sistemului.

O buclă de așteptare activă este un exemplu simplu:

Exemplu 6.1 Așteptare activă a unui eveniment

```
1 while ((STATUS_REGISTER & READY_MASK) == 0u)
2 {
3     /* Procesorul executa permanent instructiuni. */
4 }
```

Procesorul rămâne în modul activ până când perifericul finalizează operația. O abordare bazată pe întreruperi permite procesorului să intre într-un mod de repaus:

Exemplu 6.2 Așteptarea unui eveniment prin întrerupere

```
1 start_peripheral_operation();
2
3 while (!operation_completed)
4 {
5     enter_sleep_mode();
6 }
```

În această variantă, întreruperea perifericului setează variabila `operation_completed` și trezește procesorul. Soluția bazată pe întreruperi nu e automat optimă în toate situațiile — dacă evenimentul apare după numai câteva cicluri, energia tranziției către Sleep poate fi mai mare decât economia realizată; decizia trebuie luată în funcție de durata așteptării și de timpul de revenire.

Fiecare instrucțiune executată produce activitate în procesor. Costul energetic depinde de operația executată, valorile operanzilor, starea pipeline-ului, accesările memoriei, ratările cache, frecvența și tensiunea procesorului și activitatea celorlalte unități. Modelele care asociază un cost energetic fiecărei instrucțiuni pot fi utilizate pentru estimare, dar energia reală nu e întotdeauna suma independentă a costurilor instrucțiunilor — tranzițiile dintre instrucțiuni, pipeline-ul și memoria influențează rezultatul [88].

O aproximare simplificată este:

$$E_{\text{program}} \approx \sum_{i=1}^{m} n_i E_i + E_{\text{memorie}} + E_{\text{periferice}} + E_{\text{așteptare}}, \tag{6.4}$$

unde n_i este numărul execuțiilor instrucțiunii i, iar E_i costul energetic mediu asociat acesteia.

Această relație evidențiază faptul că reducerea numărului de instrucțiuni poate fi utilă, dar nu este suficientă. O implementare cu mai puține instrucțiuni poate accesa mai frecvent memoria externă și poate consuma mai mult.

Software-ul influențează și autonomia prin controlul perifericelor. Un modul radio care rămâne în recepție poate consuma mult mai mult decât procesorul. Dezactivarea corectă a

transceiverului poate avea un efect mai important decât optimizarea câtorva instrucțiuni.

Exemplu

Se consideră un nod care efectuează o măsurare la fiecare minut.
În prima versiune, procesorul și senzorul rămân permanent active:

$$I_{\text{versiunea 1}} = 8\,\text{mA}.$$

În a doua versiune, sistemul consumă 8 mA timp de 100 ms și 10 μA în restul intervalului. Curentul mediu este:

$$I_{\text{mediu}} = \frac{8 \cdot 0{,}1 + 0{,}01 \cdot 59{,}9}{60}.$$

Rezultă:

$$I_{\text{mediu}} \approx 0{,}0233\,\text{mA} = 23{,}3\,\mu\text{A}.$$

Hardware-ul este identic, dar politica software reduce curentul mediu de peste 300 de ori.

Acest exemplu arată că eficiența software nu trebuie evaluată numai prin durata unei funcții. Trebuie analizat întregul ciclu de funcționare și timpul petrecut în fiecare stare.

6.2 Optimizarea energetică la nivel de algoritm și cod

Cea mai eficientă metodă de reducere a energiei software e eliminarea activității care nu trebuie executată — o optimizare la nivel de algoritm poate evita mii sau milioane de instrucțiuni, în timp ce o optimizare locală a codului economisește uneori doar câteva cicluri.

Complexitatea algoritmică nu trebuie însă confundată direct cu energia: notația asimptotică descrie evoluția numărului operațiilor pentru dimensiuni mari ale problemei, dar nu include costul diferit al operațiilor, accesările de memorie, utilizarea cache-ului, paralelismul, costul inițializării, caracteristicile datelor sau optimizările compilatorului. Energia unui algoritm poate fi aproximată prin:

$$E_{\text{algoritm}} = N_{\text{op}} E_{\text{op}} + N_{\text{mem}} E_{\text{mem}} + E_{\text{control}} + E_{\text{I/O}}, \tag{6.5}$$

unde N_{op} e numărul operațiilor de calcul, iar N_{mem} numărul accesărilor de memorie. Un algoritm cu mai puține operații poate consuma mai mult dacă utilizează structuri de date mari sau accesări aleatoare în memoria externă. Figura 6.2 prezintă relația dintre alegerea algoritmului, implementarea codului și energia finală.

O comparație clasică e cea dintre căutarea liniară și căutarea binară. Pentru un vector ordonat cu N elemente, căutarea liniară necesită în cazul cel mai nefavorabil:

$$N_{\text{comp,liniar}} = N \tag{6.6}$$

comparații. Căutarea binară necesită aproximativ:

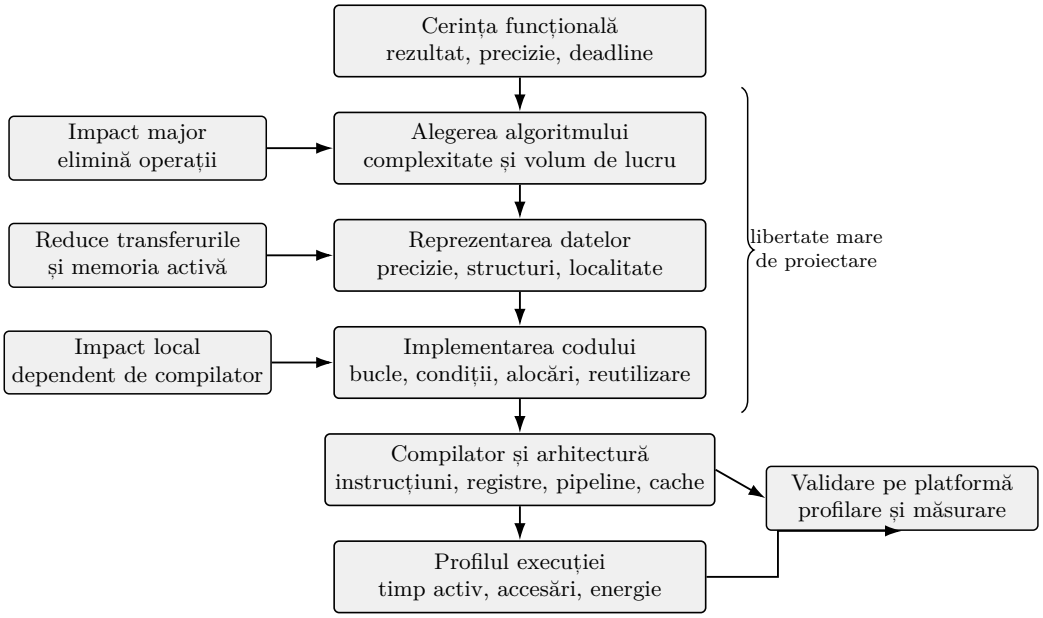


Figura 6.2 Nivelurile optimizării energetice a algoritmului și codului.

$$N_{\text{comp,binar}} = \lceil \log_2 N \rceil \,. \tag{6.7}$$

Pentru $N = 1\,000\,000$, rezultă $\log_2(1\,000\,000) \approx 20$ — diferența față de sute de mii de comparații e semnificativă. Totuși, căutarea binară presupune date ordonate și acces aleator; dacă vectorul trebuie sortat pentru o singură căutare, costul sortării poate depăși economia obținută. Alegerea algoritmului trebuie făcută în funcție de întregul scenariu: numărul căutărilor, frecvența modificării datelor, memoria disponibilă, structura memoriei, termenul limită și energia inițializării.

Un alt exemplu e calcularea mediei glisante. O implementare directă recalculează suma tuturor elementelor:

Exemplu 6.3 Calcul direct al unei medii glisante

```
1  float moving_average(const float* data, std::size_t window)
2  {
3      float sum = 0.0f;
4
5      for (std::size_t i = 0; i < window; ++i)
6      {
7          sum += data[i];
8      }
9
10     return sum / static_cast<float>(window);
11 }
```

Dacă fereastra are 100 de elemente și rezultatul e actualizat la fiecare eșantion, sunt executate aproximativ 100 de adunări pentru fiecare valoare nouă. O implementare incrementală actualizează suma:

$$S[k] = S[k-1] + x[k] - x[k-N]. \tag{6.8}$$

Codul poate fi:

Exemplu 6.4 Actualizarea incrementală a mediei

```
1 running_sum += new_value;
2 running_sum -= oldest_value;
3
4 const float average =
5     running_sum / static_cast<float>(window_size);
```

Numărul operațiilor e redus, dar trebuie păstrat un buffer circular cu valorile precedente — energia economisită prin calcul trebuie comparată cu energia accesărilor bufferului.

După alegerea algoritmului, implementarea poate fi optimizată, dar aceste optimizări trebuie aplicate după profilare și după activarea opțiunilor corespunzătoare ale compilatorului. Compilatoarele moderne efectuează automat eliminarea codului mort, propagarea constantelor, eliminarea expresiilor comune, mutarea calculelor invariabile în afara buclelor, înlocuirea anumitor operații, inlining, vectorizare și loop unrolling. Optimizarea manuală poate uneori împiedica compilatorul să genereze cod eficient, de aceea rezultatul trebuie verificat prin măsurarea timpului, analiza codului generat și măsurarea energiei. Un exemplu de calcul invariant este:

Exemplu 6.5 Calcul redundant în interiorul unei bucle

```
1 for (std::size_t i = 0; i < size; ++i)
2 {
3     const float coefficient = gain * offset;
4     output[i] = coefficient * input[i];
5 }
```

Calculul poate fi mutat înaintea buclei:

Exemplu 6.6 Eliminarea unui calcul redundant

```
1 const float coefficient = gain * offset;
2
3 for (std::size_t i = 0; i < size; ++i)
4 {
5     output[i] = coefficient * input[i];
6 }
```

În mod normal, un compilator optimizant identifică această situație dacă poate demonstra că variabilele nu se modifică; utilizarea nejustificată a calificatorului `volatile` poate împiedica optimizarea și poate produce accesări repetate ale memoriei.

Alegerea tipurilor de date influențează memoria și calculul, dar un tip mai mic nu e automat mai rapid sau mai eficient. Exemplul:

Exemplu 6.7 Alegerea unui tip cu dimensiune explicită

```
1 std::uint8_t counter = 0u;
```

garantează un obiect pe 8 biți, dar procesorul poate efectua operațiile aritmetice în registre de 32 de biți, ceea ce poate introduce extensii, măști sau operații suplimentare. Tipurile cu dimensiune fixă sunt utile când formatul datelor e specificat, datele sunt transmise printr-un protocol, memoria e limitată sau e necesar un domeniu numeric precis. Pentru performanță pot fi utilizate și tipuri precum `std::uint_fast8_t`, a căror dimensiune concretă depinde de platformă.

Reducerea preciziei numerice poate economisi energie, în special pentru procesarea semnalelor și algoritmii de inteligență artificială, dar trebuie verificat efectul asupra erorii rezultatului. Aritmetica în virgulă mobilă nu e universal mai costisitoare decât cea întreagă — pe un microcontroler fără FPU, operațiile floating-point sunt implementate prin biblioteci software și pot necesita multe instrucțiuni, dar pe un procesor cu FPU hardware costul poate fi apropiat de cel al operațiilor întregi.

Înlocuirea împărțirii la o putere a lui doi cu o deplasare e un exemplu frecvent:

Exemplu 6.8 Împărțire și deplasare

```
1 const std::uint32_t result_a = value / 8u;
2 const std::uint32_t result_b = value >> 3u;
```

Pentru o valoare fără semn, cele două expresii sunt echivalente. Compilatorul realizează de regulă automat transformarea.

Pentru valori cu semn, deplasarea și împărțirea pot produce rezultate diferite din cauza regulilor de rotunjire și a modului în care este implementată deplasarea aritmetică. O asemenea optimizare nu trebuie aplicată fără analiza semnificației datelor.

Expresiile condiționale beneficiază de evaluarea short-circuit. Pentru operatorul logic AND, evaluarea se oprește la prima condiție falsă:

Exemplu 6.9 Evaluarea short-circuit a condițiilor

```
1 if (sensor_available() &&
2     sample_is_valid() &&
3     battery_level > minimum_level)
4 {
5     process_sample();
6 }
```

Ordinea optimă depinde de două caracteristici:

- probabilitatea ca o condiție să oprească evaluarea;
- costul calculării condiției.

O verificare ieftină și frecvent falsă poate fi plasată înaintea unei operații costisitoare. Totuși, ordinea nu trebuie schimbată dacă funcțiile au efecte secundare sau dacă semantica programului depinde de succesiunea lor.

Ramificațiile pot influența și pipeline-ul procesorului. O ramificație greu de prezis poate produce golirea pipeline-ului și execuție inutilă. În aplicațiile cu date regulate, reorganizarea codului poate reduce aceste penalizări.

Loop unrolling reduce numărul operațiilor de control ale buclei:

Exemplu 6.10 Buclă standard

```
1  for (std::size_t i = 0; i < size; ++i)
2  {
3      sum += data[i];
4  }
```

O variantă desfășurată este:

Exemplu 6.11 Loop unrolling cu factor patru

```
1  std::size_t i = 0;
2
3  for (; i + 3 < size; i += 4)
4  {
5      sum += data[i];
6      sum += data[i + 1];
7      sum += data[i + 2];
8      sum += data[i + 3];
9  }
10
11 for (; i < size; ++i)
12 {
13     sum += data[i];
14 }
```

Avantajele posibile sunt:

- mai puține comparații;
- mai puține ramificații;
- paralelism mai bun între instrucțiuni;
- posibilitatea vectorizării.

Dezavantajele includ creșterea dimensiunii codului și presiunea mai mare asupra registrelor. Pe un sistem cu memorie de instrucțiuni redusă, codul mai mare poate produce mai multe accesări Flash și poate crește energia.

Inlining-ul funcțiilor are un compromis similar. Eliminarea apelului poate reduce timpul, dar duplicarea codului mărește memoria ocupată.

Recursivitatea poate fi elegantă, însă utilizează stiva pentru fiecare apel. În sistemele cu memorie limitată, o implementare iterativă poate fi mai predictibilă. Aceasta nu înseamnă că recursivitatea este întotdeauna ineficientă; rezultatul depinde de adâncime, compilator și arhitectură.

Alocarea dinamică a memoriei poate introduce:

- timp de administrare;
- fragmentare;
- comportament temporal variabil;
- accesări suplimentare ale memoriei.

În sistemele real-time sunt utilizate frecvent buffere statice, pool-uri de obiecte sau alocarea realizată numai la inițializare.

Figura 6.3 prezintă efectul unei optimizări care reduce timpul activ și permite intrarea mai devreme în repaus.

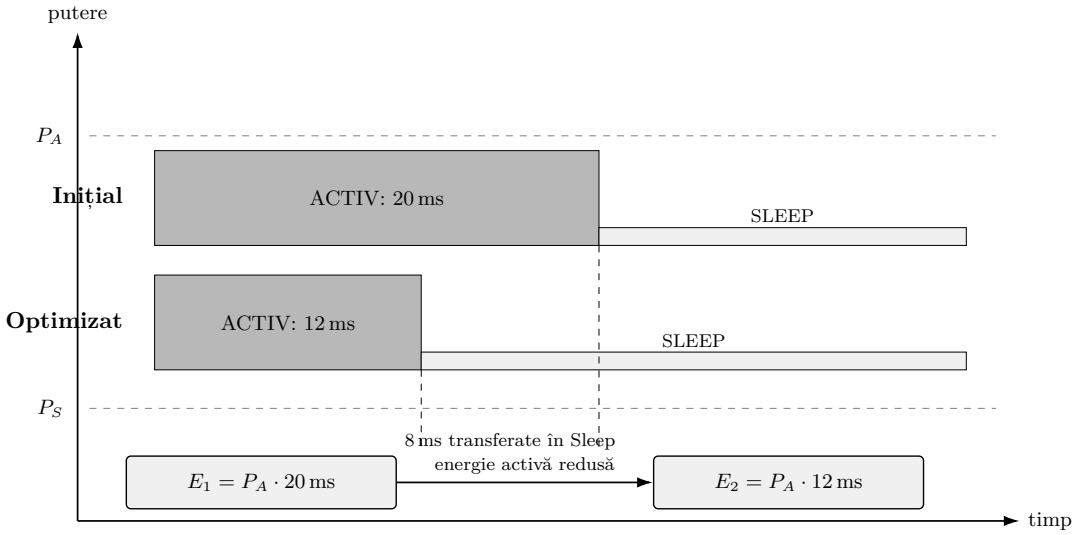


Figura 6.3 Reducerea energiei prin scurtarea intervalului activ.

Exemplu

O activitate este executată inițial în:

$$T_1 = 20\,\text{ms}$$

la o putere activă de:

$$P_A = 50\,\text{mW}.$$

După optimizare, timpul devine:

$$T_2 = 12\,\text{ms}.$$

Energia activă inițială este:

$$E_1 = 50\,\text{mW} \cdot 20\,\text{ms} = 1\,\text{mJ}.$$

Energia după optimizare este:

$$E_2 = 50\,\text{mW} \cdot 12\,\text{ms} = 0{,}6\,\text{mJ}.$$

Economia este:

$$S_E = \frac{1 - 0{,}6}{1} \cdot 100\% = 40\%.$$

Dacă sistemul intră apoi într-un mod de repaus, cele opt milisecunde eliberate pot produce o economie suplimentară.

Executarea cât mai rapidă nu e totuși întotdeauna soluția optimă — dacă scurtarea timpului cere creșterea tensiunii, energia poate crește, deci strategia trebuie evaluată pentru întregul punct de funcționare.

6.3 Optimizarea accesului la memorie și a transferurilor de date

În sistemele embedded moderne, energia deplasării datelor poate fi comparabilă sau chiar mai mare decât energia operațiilor aritmetice efectuate asupra lor [39], deci optimizarea memoriei nu e doar o problemă de performanță, ci și una energetică. Memoria e organizată ierarhic: nivelurile apropiate de procesor sunt rapide și eficiente energetic, dar au capacitate redusă, iar cele îndepărtate oferă capacitate mare, cu transferuri mai lente și mai costisitoare. Figura 6.4 prezintă ierarhia tipică.

O ierarhie poate include registre, memorie cache, SRAM locală, Flash intern, RAM externă, memorie Flash externă și stocare nevolatilă. Costul exact depinde de tehnologie și arhitectură, dar accesarea unei memorii externe necesită de regulă activarea unor buffere, magistrale și controlere suplimentare. Energia transferului datelor poate fi aproximată prin:

$$E_{\text{transfer}} = N_{\text{accesri}} E_{\text{acces}} + N_{\text{biți}} E_{\text{bit}}. \tag{6.9}$$

Reducerea energiei poate fi obținută prin scăderea numărului accesărilor, a numărului biților transferați sau a distanței dintre memoria utilizată și unitatea de calcul. Localitatea datelor e unul dintre principiile fundamentale: localitatea temporală înseamnă că o valoare utilizată recent are o probabilitate ridicată de a fi utilizată din nou, iar localitatea spațială înseamnă că după accesarea unei locații e probabilă accesarea locațiilor învecinate — aceste proprietăți permit funcționarea eficientă a memoriilor cache.

Un acces secvențial:

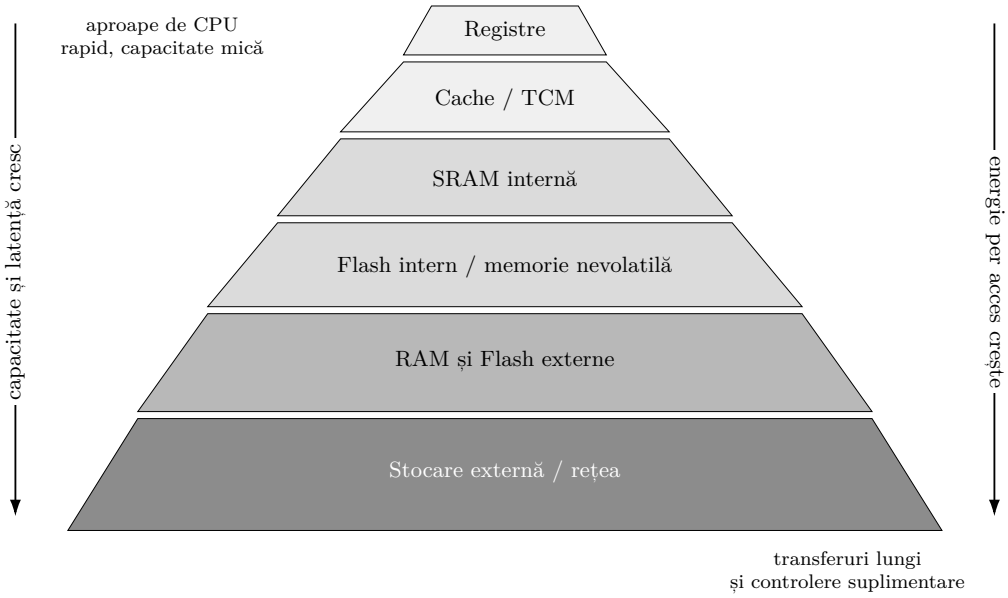


Figura 6.4 Ierarhia memoriei și costul energetic al accesului la date.

Exemplu 6.12 Parcurgerea secvențială a unui vector

```
1 std::int64_t sum = 0;
2
3 for (std::size_t i = 0; i < size; ++i)
4 {
5     sum += data[i];
6 }
```

utilizează eficient liniile cache și transferurile în rafală. Un acces indirect:

Exemplu 6.13 Acces indirect la elementele unui vector

```
1 std::int64_t sum = 0;
2
3 for (std::size_t i = 0; i < size; ++i)
4 {
5     sum += data[index[i]];
6 }
```

poate accesa locații îndepărtate și poate genera mai multe ratări cache, plus citirea vectorului de indici. Cele două variante nu sunt întotdeauna interschimbabile — accesul indirect poate fi impus de algoritm, dar ordonarea sau gruparea datelor poate îmbunătăți localitatea. În cazul matricelor, ordinea buclelor trebuie corelată cu organizarea în memorie; în C și C++, matricile sunt stocate de regulă pe linii. Parcurgerea eficientă este:

Exemplu 6.14 Parcurgerea unei matrice pe linii

```
for (std::size_t row = 0; row < rows; ++row)
{
    for (std::size_t column = 0; column < columns; ++column)
    {
        sum += matrix[row][column];
    }
}
```

Inversarea buclelor poate produce salturi mari între accesări și o utilizare mai slabă a cache-ului. Procesarea în blocuri (*blocking* sau *tiling*) permite păstrarea unei regiuni de date în memoria locală și reutilizarea ei înaintea încărcării următorului bloc — tehnică utilizată pentru înmulțirea matricelor, procesarea imaginilor, convoluții, filtrare și algoritmi de inteligență artificială. Reutilizarea datelor poate fi exprimată prin:

$$R_{\text{reutilizare}} = \frac{N_{\text{operații}}}{N_{\text{transferuri externe}}}. \tag{6.10}$$

O valoare ridicată indică utilizarea de mai multe ori a datelor încărcate. Copierea inutilă a bufferelor trebuie evitată:

Exemplu 6.15 Procesarea directă a unui buffer

```
void process_samples(const std::uint16_t* samples,
                     std::size_t sample_count)
{
    for (std::size_t i = 0; i < sample_count; ++i)
    {
        process_sample(samples[i]);
    }
}
```

Transmiterea bufferului prin pointer evită o copie, dar trebuie controlată durata de viață a datelor și accesul concurent. Bufferul local e util când datele sunt citite dintr-o memorie externă și apoi reutilizate:

Exemplu 6.16 Transferul datelor într-un buffer local

```
std::array<std::uint16_t, 128> local_buffer{};

read_external_memory(local_buffer.data(),
                     local_buffer.size());

process_samples(local_buffer.data(),
                local_buffer.size());
```

Această strategie e eficientă dacă datele sunt procesate de mai multe ori — dacă fiecare

valoare e utilizată o singură dată, copierea în buffer poate introduce un transfer suplimentar fără beneficiu.

DMA permite transferul datelor fără implicarea procesorului pentru fiecare element; cât timp DMA funcționează, procesorul poate executa altă sarcină sau poate intra în repaus. Energia cu DMA poate fi reprezentată prin:

$$E_{\text{DMA}} = E_{\text{configurare}} + E_{\text{transfer}} + E_{\text{finalizare}}. \tag{6.11}$$

Transferul realizat direct de procesor consumă:

$$E_{\text{CPU}} = E_{\text{instrucțiuni}} + E_{\text{transfer}} + E_{\text{așteptare}}. \tag{6.12}$$

DMA este avantajos atunci când economia procesorului depășește energia de configurare:

$$E_{\text{DMA}} < E_{\text{CPU}}. \tag{6.13}$$

Pentru transferuri foarte scurte, configurarea DMA poate fi mai costisitoare decât o copiere simplă; pentru blocuri mari sau transferuri periodice, avantajul e de regulă mai important.

Structurile de date influențează atât memoria ocupată, cât și numărul transferurilor. Compilatorul poate introduce spații de aliniere între câmpuri:

Exemplu 6.17 Structură de date cu posibile spații de aliniere

```
1 struct SensorData
2 {
3     std::uint8_t status;
4     std::uint32_t timestamp;
5     std::uint16_t value;
6 };
```

Dimensiunea structurii poate fi mai mare decât suma dimensiunilor câmpurilor. Reordonarea poate reduce spațiile:

Exemplu 6.18 Reordonarea câmpurilor unei structuri

```
1 struct SensorDataOptimized
2 {
3     std::uint32_t timestamp;
4     std::uint16_t value;
5     std::uint8_t status;
6 };
```

Rezultatul exact depinde de regulile de aliniere ale compilatorului. Utilizarea unei structuri împachetate forțat poate reduce dimensiunea:

Exemplu 6.19 Exemplu conceptual de structură împachetată

```
#pragma pack(push, 1)

struct PackedSensorData
{
    std::uint8_t status;
    std::uint32_t timestamp;
    std::uint16_t value;
};

#pragma pack(pop)
```

Câmpurile pot deveni însă nealiniate: pe unele procesoare accesul nealiniat e mai lent, pe altele poate produce o excepție. Împachetarea trebuie utilizată în special pentru formate de comunicație sau stocare, nu automat pentru toate structurile interne — o soluție robustă e separarea formatului transmis de reprezentarea internă, cu date serializate explicit într-un buffer de octeți, controlând ordinea și alinierea.

Reducerea lățimii datelor poate diminua traficul: un vector de 1000 de valori pe 16 biți ocupă 2000 octeți, față de 4000 octeți pe 32 de biți — dar reducerea preciziei trebuie să respecte domeniul numeric și eroarea acceptată.

Memoria Flash are caracteristici diferite față de SRAM: citirile pot fi rapide, dar scrierea și ștergerea sunt mai costisitoare și au o durată de viață limitată. Salvarea frecventă a unor variabile în Flash poate consuma energie, bloca procesorul, uza memoria și necesita ștergerea unei pagini întregi. Pentru date actualizate des pot fi utilizate buffer RAM, scriere în loturi, jurnalizare circulară, wear leveling sau memorii nevolatile dedicate.

Accesul la memorie trebuie analizat și din perspectiva alimentării — unele memorii pot fi trecute în standby sau Deep Power-Down, iar software-ul trebuie să grupeze accesările astfel încât memoria să poată rămâne oprită perioade mai lungi. Figura 6.5 prezintă principalele direcții de optimizare.

Exemplu

Un algoritm citește un bloc de 1024 de valori dintr-o memorie externă de patru ori. În varianta inițială sunt realizate $4 \cdot 1024 = 4096$ accesări externe. În varianta optimizată, blocul e transferat o singură dată într-o memorie locală și e apoi reutilizat, iar numărul accesărilor externe scade la 1024. Dacă energia unui acces extern e E_{ext}, energia evitată este:

$$E_{\text{economisit}} = 3072 E_{\text{ext}} - E_{\text{buffer}},$$

unde E_{buffer} include energia accesărilor memoriei locale. Optimizarea e avantajoasă deoarece accesul local e, în general, mai eficient decât accesul la memoria externă.

Optimizarea memoriei trebuie realizată împreună cu profilarea aplicației — o structură

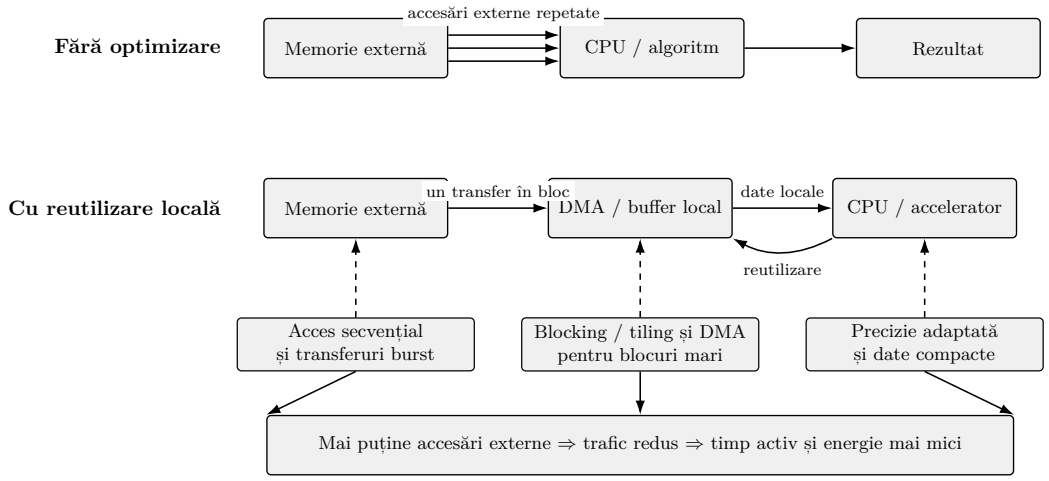


Figura 6.5 Reducerea energiei asociate memoriei și transferurilor de date.

Tabela 6.1 Tehnici software pentru optimizarea accesului la memorie

Tehnică	Avantaj energetic	Posibilă limitare
Acces secvențial	Utilizează eficient cache-ul și transferurile în rafală	Nu este posibil pentru orice algoritm
Reutilizare locală	Reduce accesările memoriei externe	Necesită memorie locală suplimentară
Procesare în blocuri	Crește localitatea și reutilizarea	Introduce complexitate și alegerea dimensiunii blocului
Eliminarea copiilor	Reduce traficul și timpul de execuție	Necesită gestionarea atentă a duratei de viață a datelor
DMA	Reduce activitatea procesorului	Are un cost de configurare
Reducerea preciziei	Reduce dimensiunea datelor și traficul	Poate afecta acuratețea rezultatului
Reordonarea structurilor	Reduce spațiile de aliniere	Poate modifica formatul binar
Data packing	Reduce memoria ocupată și volumul transmis	Poate produce accesări nealiniate
Gruparea scrierilor Flash	Reduce numărul operațiilor costisitoare	Necesită buffer și strategie pentru pierderea alimentării

mai compactă, o copie eliminată sau utilizarea DMA pot produce economii, dar rezultatul depinde de platformă, compilator și profilul datelor.

6.4 Power-Aware Software și controlul resurselor

Conceptul de *Power-Aware Software* descrie dezvoltarea aplicațiilor care iau în considerare explicit energia necesară executării funcționalităților — obiectivul nu e doar rezultatul corect, ci obținerea lui cu un consum energetic minim, fără încălcarea cerințelor de timp real, performanță, precizie și fiabilitate.

Software-ul power-aware trebuie să cunoască, cel puțin la nivel funcțional, caracteristicile energetice ale platformei: ce resurse pot fi oprite, care e timpul lor de pornire, ce informații sunt pierdute la dezactivare și ce surse pot produce reactivarea sistemului. Într-o aplicație convențională, software-ul solicită o resursă când are nevoie de ea; într-o aplicație power-aware, software-ul controlează întregul ciclu de viață al resursei:

$$\text{dezactivat} \rightarrow \text{inițializare} \rightarrow \text{activ} \rightarrow \text{finalizare} \rightarrow \text{dezactivat}.$$

Această abordare e importantă pentru senzori, module radio, memorii externe, convertoare analog-digitale, acceleratoare și domenii de ceas.

6.4.1 Principii de proiectare power-aware

O aplicație eficientă energetic trebuie să reducă atât durata activității, cât și numărul activărilor resurselor. Pentru un ciclu de funcționare, energia poate fi exprimată prin:

$$E_{\text{ciclu}} = \sum_{i=1}^{n} P_i T_i + \sum_{j=1}^{m} E_{\text{tranziție},j}, \tag{6.14}$$

unde P_i și T_i reprezintă puterea și durata stării i, iar $E_{\text{tranziție},j}$ energia necesară unei schimbări de stare. Relația arată că energia poate fi redusă prin scurtarea stărilor cu putere ridicată, selectarea unor stări cu putere mai redusă, reducerea numărului tranzițiilor, gruparea activităților și eliminarea activărilor inutile.

Gruparea activităților (*batching*) — dacă un senzor trebuie citit de mai multe ori într-un interval scurt, poate fi mai eficient să rămână activ până la finalizarea întregului grup de măsurători decât să fie pornit și oprit pentru fiecare eșantion.

În schimb, dacă măsurătorile sunt separate de intervale lungi, menținerea senzorului activ produce energie inutilă. Decizia depinde de durata inactivității, de timpul de stabilizare și de energia pornirii.

Un software power-aware poate fi descris printr-o buclă de control formată din:

1. monitorizarea stării sistemului;

2. estimarea necesarului de performanță;

3. alegerea configurației energetice;

4. aplicarea configurației;

5. verificarea rezultatului și a deadline-urilor.

Figura 6.6 prezintă această buclă.

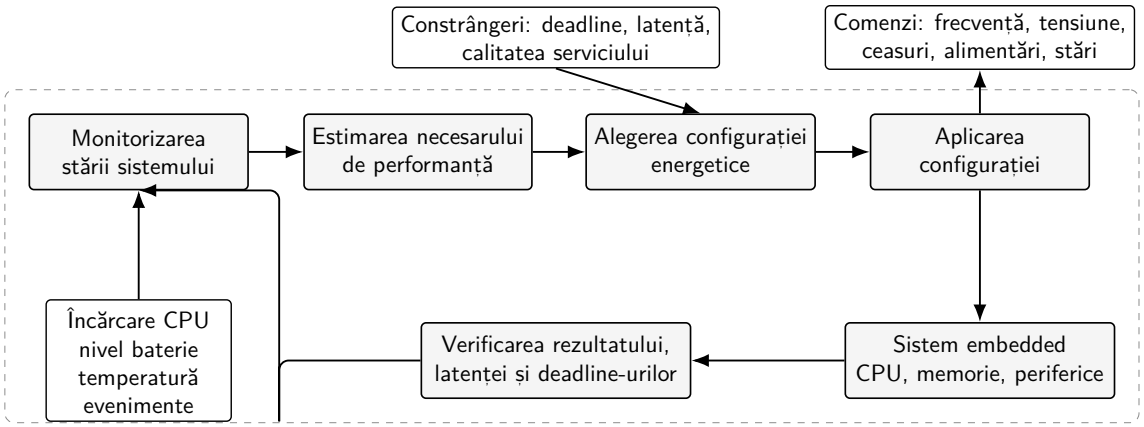


Figura 6.6 Bucla software de monitorizare și control energetic.

Intrările buclei pot include:

- încărcarea procesorului;
- numărul activităților din coadă;
- nivelul bateriei;
- temperatura;
- deadline-urile;
- starea comunicației;
- disponibilitatea unei surse externe de energie.

Ieșirile pot controla frecvența, tensiunea, starea procesorului, ceasurile periferice și alimentarea componentelor externe.

Politica nu trebuie să urmărească exclusiv minimizarea energiei. Un sistem de control trebuie să răspundă la timp, iar un dispozitiv de comunicație trebuie să respecte ferestrele protocolului. Problema poate fi formulată prin:

$$\min E_{\text{total}}, \tag{6.15}$$

sub constrângerile:

$$T_{\text{rspuns}} \leq T_{\max}, \tag{6.16}$$

$$N_{\text{deadline ratate}} = 0, \tag{6.17}$$

$$Q_{\text{serviciu}} \geq Q_{\min}. \tag{6.18}$$

Conceptul de calitate a serviciului poate include rata de eșantionare, precizia măsurătorilor, debitul comunicației sau rata de cadre.

6.4.2 Controlul perifericelor și al comunicațiilor

În numeroase sisteme embedded, procesorul nu este componenta dominantă din punct de vedere energetic. Un modul radio, un receptor GNSS, un senzor optic sau un actuator poate consuma mult mai mult decât nucleul de calcul.

Din acest motiv, software-ul trebuie să controleze explicit:

- alimentarea perifericului;
- semnalul de ceas;
- modul său intern de consum;
- timpul de stabilizare;
- transferurile aflate în desfășurare;
- starea pinilor de intrare și ieșire.

O secvență generică de utilizare a unui senzor este:

1. activarea alimentării;
2. așteptarea stabilizării;
3. configurarea interfeței;
4. efectuarea măsurătorii;
5. citirea și validarea rezultatului;
6. trecerea senzorului în standby;
7. oprirea alimentării, dacă este permisă.

Timpul de stabilizare poate fi important — un senzor care necesită 100 ms pentru stabilizare nu poate fi oprit între două măsurători separate de doar câteva milisecunde. Energia unei utilizări poate fi exprimată prin:

$$E_{\text{periferic}} = E_{\text{pornire}} + P_{\text{stabilizare}} T_{\text{stabilizare}} + P_{\text{activ}} T_{\text{activ}} + E_{\text{oprire}}. \tag{6.19}$$

Figura 6.7 prezintă un ciclu de activare, utilizare și oprire.

Pentru comunicații, energia depinde de cantitatea de informație, protocol, distanță, puterea de emisie și numărul retransmisiilor. Software-ul poate reduce energia prin gruparea mai multor valori într-un pachet, eliminarea datelor redundante, transmiterea doar la apariția unui eveniment, adaptarea ratei de raportare, utilizarea confirmărilor doar când sunt necesare și oprirea completă a transceiverului între transmisii.

Gruparea datelor reduce numărul pornirilor radio și numărul antetelor transmise, dar un pachet mai mare poate avea o probabilitate mai mare de a fi afectat de erori și poate necesita retransmiterea unei cantități mai mari de date. Considerăm n măsurători, fiecare cu L_D biți

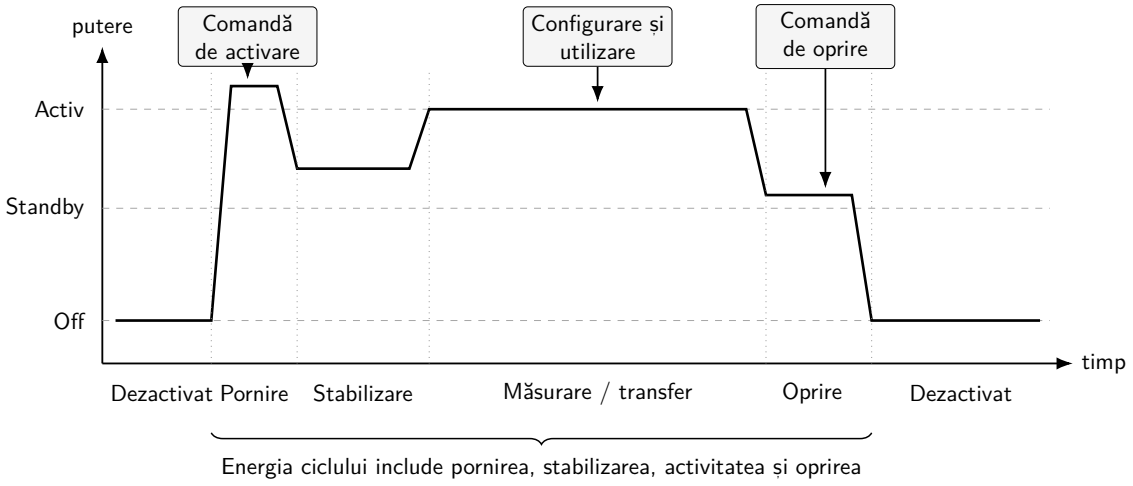


Figura 6.7 Controlul software al ciclului de funcționare al unui periferic.

utili. Dacă fiecare valoare e transmisă separat, iar antetul are L_H biți, numărul total de biți este:

$$L_{\text{separat}} = n\,(L_D + L_H)\,. \tag{6.20}$$

Dacă valorile sunt grupate într-un singur pachet:

$$L_{\text{grupat}} = nL_D + L_H. \tag{6.21}$$

Numărul de biți evitați e:

$$\Delta L = (n-1)\,L_H. \tag{6.22}$$

Pe lângă energia transmisiei, sunt evitate și pornirile repetate ale transceiverului.

Exemplu

Un nod produce zece măsurători de câte 16 biți. Antetul fiecărui pachet are 80 de biți. Transmiterea separată necesită:

$$L_{\text{separat}} = 10\,(16 + 80) = 960\,\text{biți}.$$

Gruparea într-un singur pachet necesită:

$$L_{\text{grupat}} = 10 \cdot 16 + 80 = 240\,\text{biți}.$$

Volumul transmis este redus cu:

$$960 - 240 = 720\,\text{biți}.$$

La această economie se adaugă evitarea a nouă secvențe de activare și acces la canal.

Software-ul trebuie să țină cont și de starea pinilor — un pin lăsat flotant poate produce tranziții și curenți suplimentari, deci la intrarea într-un mod de consum redus, pinii trebuie configurați într-o stare cunoscută, compatibilă cu circuitele externe.

6.4.3 Execuția orientată pe evenimente și planificarea activității

O arhitectură bazată pe evenimente permite procesorului să execute cod doar când există o activitate reală. Evenimentele pot proveni de la un timer, un senzor, o interfață de comunicație, un pin extern, un transfer DMA, un comparator analogic sau un watchdog. Într-o arhitectură bazată pe polling, procesorul verifică repetat starea perifericelor; într-una bazată pe întreruperi, procesorul poate intra în repaus și e reactivat doar la apariția evenimentului. Un model generic este:

Exemplu 6.20 Buclă principală orientată pe evenimente

```
1  int main()
2  {
3      system_init();
4
5      while (true)
6      {
7          process_pending_events();
8          prepare_low_power_state();
9          enter_low_power_state();
10     }
11 }
```

Funcția `process_pending_events()` trebuie să proceseze toate activitățile pregătite. Înaintea intrării în repaus, software-ul trebuie să verifice din nou dacă nu a apărut un eveniment între verificarea cozii și executarea instrucțiunii de sleep — situație care e o condiție de cursă:

verificare → apariție eveniment → sleep.

Dacă evenimentul nu rămâne memorat sau nu produce o nouă întrerupere, sistemul poate rămâne în repaus deși există activitate de procesat. Microcontrolerele oferă mecanisme precum instrucțiunile `WFI` (*Wait For Interrupt*) și `WFE` (*Wait For Event*); secvența exactă trebuie implementată conform manualului procesorului și microcontrolerului.

Într-un sistem de operare real-time, planificatorul poate introduce procesorul în repaus când nu există sarcini gata de execuție. Un sistem *tickless* poate opri tick-ul periodic și poate programa următoarea trezire pentru primul timer software care expiră. Într-un sistem cu tick periodic, procesorul e trezit chiar dacă nu există activitate utilă:

$$N_{\text{treziri}} = f_{\text{tick}} T.$$

Pentru un tick de 1 kHz, într-o secundă apar 1000 de treziri; într-un sistem care trebuie să execute o activitate doar o dată pe secundă, majoritatea trezirilor pot fi inutile.

Planificarea power-aware poate grupa activitățile apropiate în timp: în loc să trezească procesorul separat pentru fiecare timer, sistemul poate executa mai multe activități în aceeași fereastră activă. Această tehnică reduce numărul tranzițiilor, dar introduce o abatere temporală, și poate fi utilizată doar dacă deadline-urile permit.

6.5 Modele software pentru estimarea consumului

Optimizarea software necesită o metodă prin care variantele de implementare să poată fi comparate. Măsurarea directă pe hardware e metoda de referință, dar nu e întotdeauna disponibilă în etapele inițiale. Modelele software estimează consumul pe baza codului executat, a evenimentelor arhitecturale sau a timpului petrecut în diferite stări, și pot fi utilizate pentru compararea algoritmilor, identificarea funcțiilor costisitoare, evaluarea compilatoarelor, explorarea punctelor DVFS, estimarea autonomiei și verificarea politicilor de power management. Niciun model nu e universal — precizia trebuie evaluată pentru platforma, configurația și profilul aplicației analizate.

6.5.1 Modele bazate pe instrucțiuni și blocuri de cod

Modelele bazate pe instrucțiuni asociază fiecărei clase de instrucțiuni o energie medie [88]. Energia poate fi aproximată prin:

$$\widehat{E}_{\text{instr}} = \sum_{i=1}^{m} n_i e_i, \tag{6.23}$$

unde:

- n_i este numărul execuțiilor instrucțiunii i;
- e_i este energia medie a instrucțiunii.

Un model extins poate include tranzițiile dintre instrucțiuni:

$$\widehat{E}_{\text{program}} = \sum_{i} n_i e_i + \sum_{i} \sum_{j} n_{ij} e_{ij} + E_{\text{cache}} + E_{\text{stall}}, \tag{6.24}$$

unde n_{ij} reprezintă numărul tranzițiilor dintre instrucțiunile i și j.

Necesitatea acestui termen apare deoarece activitatea internă depinde și de diferența dintre operanzi și de starea precedentă a procesorului.

Modelele pot fi aplicate la nivel de:

- instrucțiune;
- bloc de bază;
- funcție;
- activitate software;

- întreg program.

Un bloc de bază este o secvență de instrucțiuni cu un singur punct de intrare și un singur punct de ieșire. Dacă energia blocului este cunoscută, energia programului poate fi estimată prin:

$$\widehat{E} = \sum_{k=1}^{b} N_k E_k, \tag{6.25}$$

unde N_k este numărul execuțiilor blocului k.

Modelele la nivel de bloc reduc complexitatea, dar pot pierde efectele produse la interfața dintre blocuri.

Un simulator al setului de instrucțiuni, denumit ISS, poate furniza numărul instrucțiunilor și traseul execuției. Un ISS funcțional urmărește corectitudinea instrucțiunilor, dar nu reproduce obligatoriu pipeline-ul, cache-ul și temporizarea exactă.

Un simulator cycle-accurate oferă o precizie mai bună, însă necesită mai mult timp și un model detaliat al procesorului.

6.5.2 Modele bazate pe profilare și stări

În numeroase microcontrolere, un model bazat pe stări este mai practic decât un model bazat pe fiecare instrucțiune.

Energia este aproximată prin:

$$\widehat{E}_{\text{stri}} = \sum_{i=1}^{n} P_i T_i, \tag{6.26}$$

unde stările pot reprezenta procesor activ la o anumită frecvență, procesor în Idle, transmisie radio, recepție radio, achiziție ADC, memorie externă activă sau Deep Sleep. Pentru fiecare stare sunt necesare o putere și o durată — puterea poate proveni din fișa tehnică, măsurători, modelul regulatorului sau caracterizarea unui prototip, iar durata poate fi determinată prin instrumentarea codului, timere hardware, trace-uri sau pini GPIO. Figura 6.8 prezintă principalele surse de informație utilizate de modelele software.

Un model statistic poate estima puterea pe baza mai multor variabile observate:

$$\widehat{P} = \beta_0 + \sum_{j=1}^{r} \beta_j x_j, \tag{6.27}$$

unde x_j poate reprezenta gradul de utilizare a procesorului, numărul accesărilor memoriei, ratările cache, numărul instrucțiunilor, traficul pe magistrală sau activitatea unui periferic; coeficienții β_j sunt determinați prin măsurători. Modelele neliniare pot surprinde mai bine relația dintre activitate, tensiune, frecvență și temperatură, însă necesită mai multe date și sunt mai dificil de interpretat. Pentru un model DVFS trebuie utilizate coeficiente diferite pentru puncte de operare diferite:

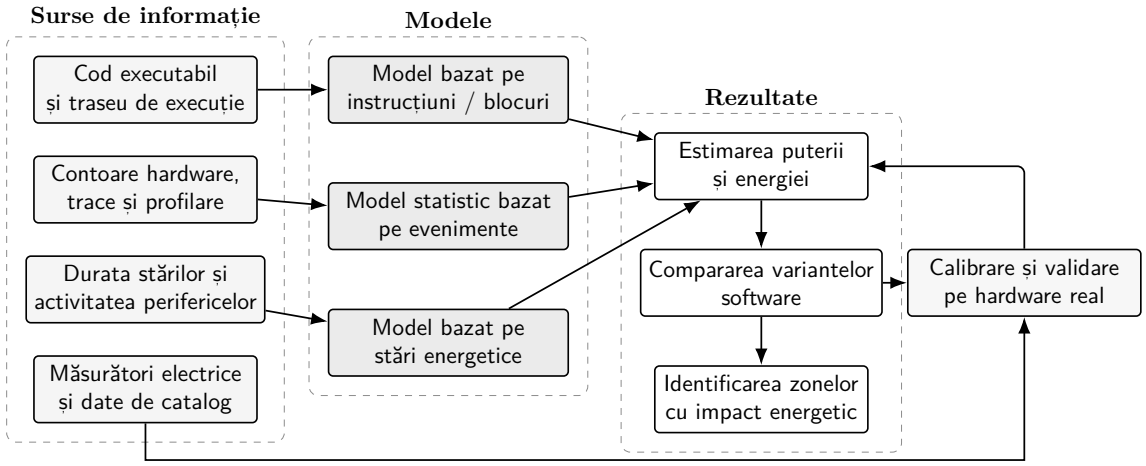


Figura 6.8 Modele software utilizate pentru estimarea consumului energetic.

$$\widehat{P} = f\left(\mathrm{OPP}, \mathbf{x}\right). \tag{6.28}$$

Puterea măsurată la o frecvență nu trebuie extrapolată automat către toate punctele de funcționare.

6.5.3 Calibrarea și validarea modelului

Un model energetic trebuie calibrat cu date reprezentative — dacă este calibrat numai pentru o buclă aritmetică, poate avea erori mari pentru o aplicație dominată de memorie sau comunicație. Procesul de construire a modelului include:

1. definirea variabilelor observabile;
2. alegerea scenariilor de caracterizare;
3. măsurarea energiei;
4. identificarea parametrilor;
5. validarea cu scenarii diferite;
6. analiza erorilor.

Eroarea relativă pentru o măsurătoare este:

$$\varepsilon_i = \frac{\left|E_i - \widehat{E}_i\right|}{E_i} \cdot 100\%. \tag{6.29}$$

Eroarea medie absolută procentuală poate fi exprimată prin:

$$\mathrm{MAPE} = \frac{1}{N} \sum_{i=1}^{N} \frac{\left|E_i - \widehat{E}_i\right|}{E_i} \cdot 100\%. \tag{6.30}$$

MAPE trebuie interpretată cu atenție pentru energii foarte mici, deoarece numitorul poate amplifica eroarea. Un model trebuie validat pentru aplicații orientate spre calcul, aplicații orientate spre memorie, activitate de comunicație, mai multe frecvențe și tensiuni, precum și temperaturi reprezentative. Temperatura influențează curenții de pierderi, iar randamentul regulatorului depinde de curent — un model care utilizează exclusiv puterea nominală a procesorului poate ignora pierderile sursei de alimentare.

Energia preluată din baterie este:

$$E_{\text{baterie}} = \frac{E_{\text{sarcin}}}{\eta_{\text{regulator}}}, \tag{6.31}$$

unde randamentul $\eta_{\text{regulator}}$ poate varia în funcție de starea sistemului. Un regulator eficient în modul activ poate avea un curent propriu important în Deep Sleep, motiv pentru care modelele trebuie construite la nivelul întregului sistem, nu numai la ieșirea regulatorului.

Tabela 6.2 Comparația modelelor software de estimare energetică

Model	Avantaje	Limitări
Bazat pe instrucțiuni	Permite asocierea consumului cu secvențe de cod	Necesită caracterizarea procesorului și nu surprinde toate efectele arhitecturale
Bazat pe blocuri	Reduce complexitatea analizei	Poate pierde interacțiunile dintre blocuri
Bazat pe stări	Simplu și potrivit pentru cicluri funcționale	Necesită delimitarea corectă a stărilor
Bazat pe contoare	Urmărește activitatea arhitecturală reală	Contoarele disponibile sunt limitate
Statistic	Poate combina mai multe surse de activitate	Depinde de datele de calibrare
Măsurare directă	Include comportamentul real al platformei	Necesită hardware și instrumentație

Modelele software nu înlocuiesc măsurarea finală. Ele permit însă identificarea rapidă a direcțiilor de optimizare și compararea variantelor înaintea integrării complete.

6.6 Managementul software al stărilor energetice

Microcontrolerele și procesoarele embedded oferă mai multe stări energetice. Software-ul decide momentul intrării într-o stare, resursele care pot fi oprite și condițiile de revenire.

Denumirile stărilor diferă între producători. Termeni precum *Sleep*, *Stop*, *Standby*, *Shutdown* și *Hibernate* nu definesc în mod universal același comportament.

Pentru fiecare platformă trebuie verificat ce ceasuri sunt oprite, ce domenii rămân alimentate, dacă SRAM-ul este păstrat, ce registre își păstrează valoarea, ce surse de trezire sunt disponibile, cât durează revenirea și ce secvență software este obligatorie.

6.6.1 Stări energetice și păstrarea contextului

O clasificare generală include stările Active, Idle, Sleep, Deep Sleep și Shutdown. În starea Active, procesorul execută instrucțiuni, iar resursele necesare sunt alimentate, putând exista mai multe puncte de performanță definite prin frecvență și tensiune. În Idle, nucleul nu execută instrucțiuni, dar majoritatea ceasurilor și perifericelor rămân disponibile, iar revenirea este rapidă cu contextul păstrat complet. În Sleep sunt oprite mai multe ceasuri — procesorul își păstrează contextul, dar anumite periferice nu mai funcționează. În Deep Sleep pot fi oprite oscilatoarele rapide, PLL-ul, nucleul și unele regiuni de memorie, rămânând activ doar un domeniu de consum redus care poate conține un timer și logica de wake-up. În Shutdown sau Hibernate, alimentarea este întreruptă aproape complet, revenirea poate fi echivalentă cu o resetare, iar pot fi păstrate numai câteva registre de backup sau o memorie specială. Figura 6.9 prezintă o ierarhie conceptuală a stărilor.

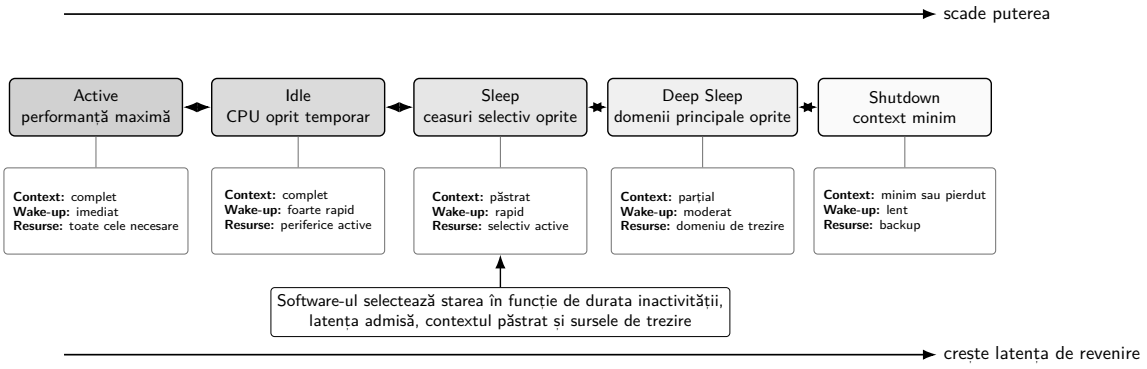


Figura 6.9 Stări energetice și tranziții controlate software.

Tabela 6.3 Caracteristici generale ale stărilor energetice

Stare	Resurse active	Context	Revenire
Active	Procesor și resurse solicitate	Păstrat	Nu este necesară
Idle	Ceasuri și periferice principale	Păstrat complet	Foarte rapidă
Sleep	Domeniu principal parțial oprit	De regulă păstrat	Rapidă sau moderată
Deep Sleep	Domeniu de wake-up și resurse selectate	Parțial păstrat	Moderată
Shutdown	Numai domeniu de backup sau wake-up	Minimal sau pierdut	Lentă, apropiată de resetare

Consumul scade în general către stările profunde, dar cresc latența și energia tranziției — alegerea nu trebuie realizată numai pe baza curentului minim.

6.6.2 Secvența software de intrare și revenire

Intrarea într-un mod de consum redus trebuie tratată ca o tranziție controlată, nu ca o singură instrucțiune. Înaintea intrării în repaus, software-ul trebuie să:

1. finalizeze sau suspende operațiile în curs;
2. salveze informațiile care nu vor fi păstrate;
3. configureze pinii;
4. dezactiveze perifericele neutilizate;
5. configureze sursele de trezire;
6. elimine flag-urile vechi;
7. selecteze starea energetică;
8. execute instrucțiunea de așteptare.

Un exemplu generic este:

Exemplu 6.21 Secvență generică de intrare într-un mod de consum redus

```
1  void enter_system_sleep()
2  {
3      disable_nonessential_interrupts();
4
5      finish_pending_transfers();
6      save_required_context();
7      configure_low_power_pins();
8      disable_unused_peripherals();
9
10     clear_wakeup_flags();
11     configure_wakeup_sources();
12     select_power_mode();
13
14     enable_required_interrupts();
15
16     data_synchronization_barrier();
17     wait_for_interrupt();
18 }
```

Funcțiile sunt conceptuale, implementarea reală depinzând de procesor și de microcontroler. La revenire, software-ul trebuie să determine sursa trezirii și să restaureze resursele în ordinea corectă:

1. verificarea cauzei trezirii;
2. stabilizarea alimentărilor;
3. pornirea oscilatoarelor;
4. configurarea PLL-ului;
5. restaurarea frecvenței;
6. reinițializarea perifericelor;
7. restaurarea contextului;
8. reluarea aplicației.

Ceasul rapid și PLL-ul pot necesita un timp de stabilizare, procesorul putând rula temporar dintr-un oscilator lent până la disponibilitatea sursei principale. Dacă revenirea este tratată ca resetare, aplicația trebuie să distingă între pornire normală, resetare produsă de watchdog, revenire din Shutdown și pierderea alimentării; această informație poate fi stocată în registre de backup.

O problemă importantă este accesarea unui periferic care nu a fost încă reactivat — driverele trebuie să cunoască starea energetică și să refuze sau să amâne operația până la restaurarea resursei. Managementul stărilor trebuie integrat cu mecanismele de concurență, deoarece o componentă nu poate fi oprită cât timp este utilizată de o sarcină sau de un transfer DMA. Pot fi utilizate contoare de referință:

$$N_{\text{utilizatori}} = \sum_{k=1}^{r} u_k, \tag{6.32}$$

unde $u_k = 1$ dacă modulul k utilizează resursa.

Resursa poate fi oprită numai dacă:

$$N_{\text{utilizatori}} = 0. \tag{6.33}$$

6.6.3 Surse de trezire și latența revenirii

O stare energetică este utilă numai dacă sistemul poate reveni la momentul necesar. Sursele de trezire pot include timer de consum redus, ceas de timp real, pin GPIO, interfață serială, modul radio, comparator analogic, watchdog sau senzor autonom.

Nu toate sursele sunt disponibile în toate stările. De exemplu, o interfață UART alimentată din domeniul principal nu poate trezi sistemul dintr-o stare în care domeniul respectiv este oprit.

Latența totală de răspuns este:

$$T_{\text{rspuns}} = T_{\text{detectare}} + T_{\text{wake-up}} + T_{\text{restaurare}} + T_{\text{execuție}}. \tag{6.34}$$

Aceasta trebuie să respecte deadline-ul:

$$T_{\text{rspuns}} \leq D. \tag{6.35}$$

Pentru activități periodice, software-ul poate utiliza *pre-wakeup*. Sistemul este trezit înaintea momentului la care rezultatul este necesar:

$$T_{\text{trezire}} = T_{\text{deadline}} - T_{\text{pregtire}} - T_{\text{execuție}}. \tag{6.36}$$

De exemplu, dacă un senzor necesită 50 ms pentru stabilizare, iar procesarea necesită 10 ms, sistemul trebuie trezit cu cel puțin 60 ms înaintea deadline-ului.

Pre-wakeup ascunde latența, dar reduce durata efectivă a repausului. Momentul trezirii trebuie ales suficient de devreme pentru respectarea deadline-ului, dar nu excesiv de devreme.

Unele surse de trezire pot produce evenimente false sau zgomot. Un pin extern poate necesita rezistență de pull-up sau pull-down, filtrare, debounce sau confirmarea evenimentului după trezire. O trezire falsă poate fi costisitoare deoarece pornește oscilatoare, regulator, memorie și procesor fără a exista activitate utilă.

Rata trezirilor false poate fi inclusă în energie:

$$E_{\text{fals}} = N_{\text{treziri false}} E_{\text{wake-up}}. \tag{6.37}$$

Software-ul trebuie să monitorizeze cauza trezirii și să identifice sursele care produc activări nejustificate. Selecția stării energetice depinde astfel de durata estimată a inactivității, energia tranziției, latența revenirii, contextul care trebuie păstrat, sursele de trezire necesare și consumul domeniului permanent activ. Durata minimă pentru care o stare devine avantajoasă va fi analizată prin conceptul de Break-Even Time în blocul următor.

6.7 Break-Even Time și alegerea stării de consum

Trecerea într-o stare energetică mai profundă nu produce automat o economie. Intrarea și revenirea necesită timp, energie și o anumită secvență software. Dacă intervalul de inactivitate este prea scurt, costul tranziției poate depăși energia economisită.

Din acest motiv, alegerea stării trebuie realizată prin compararea energiei consumate în două scenarii: menținerea sistemului într-o stare superficială sau trecerea într-o stare mai profundă și revenirea ulterioară. Conceptul de *Break-Even Time* exprimă durata minimă de inactivitate pentru care a doua variantă devine avantajoasă energetic.

6.7.1 Modelul energetic al tranziției

Considerăm două stări: starea A, cu puterea P_A, și starea S, cu puterea redusă P_S. Pentru un interval de inactivitate T_I, menținerea sistemului în starea A consumă:

$$E_A = P_A T_I. \tag{6.38}$$

Dacă sistemul trece în starea S, energia totală este:

$$E_S = E_{A\to S} + P_S T_S + E_{S\to A}, \tag{6.39}$$

unde $E_{A\to S}$ este energia intrării în starea redusă, $E_{S\to A}$ este energia revenirii, iar T_S este timpul petrecut efectiv în starea redusă. Durata efectivă a repausului nu este întotdeauna egală cu întregul interval de inactivitate — din interval trebuie scăzute perioadele de tranziție:

$$T_S = T_I - T_{A\to S} - T_{S\to A}. \tag{6.40}$$

Pentru un model simplificat, energia totală a tranzițiilor este:

$$E_T = E_{A\to S} + E_{S\to A}. \tag{6.41}$$

Timpul de break-even este obținut prin egalarea energiilor:

$$P_A T_{\mathrm{BE}} = E_T + P_S T_{\mathrm{BE}}. \tag{6.42}$$

Rezultă:

$$T_{\mathrm{BE}} = \frac{E_T}{P_A - P_S}. \tag{6.43}$$

Relația este validă pentru modelul în care duratele tranzițiilor sunt incluse în energia E_T; într-un model mai detaliat, acestea trebuie tratate explicit.

Starea redusă este avantajoasă energetic dacă:

$$T_I > T_{\mathrm{BE}}. \tag{6.44}$$

Figura 6.10 prezintă evoluția energiei pentru cele două scenarii.

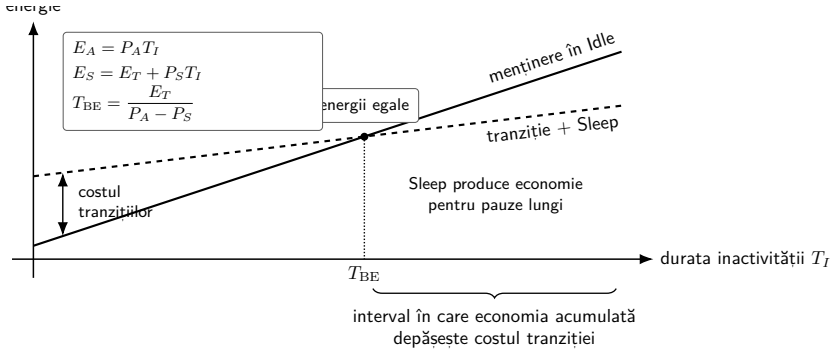


Figura 6.10 Interpretarea energetică a timpului de break-even.

Exemplu

Un periferic consumă în starea Idle:

$$P_A = 12\,\text{mW},$$

iar în Sleep:

$$P_S = 0{,}4\,\text{mW}.$$

Energia totală necesară pentru intrare și revenire este:

$$E_T = 0{,}58\,\text{mJ}.$$

Timpul de break-even este:

$$T_{\text{BE}} = \frac{0{,}58\,\text{mJ}}{12\,\text{mW} - 0{,}4\,\text{mW}}.$$

Rezultă:

$$T_{\text{BE}} = 50\,\text{ms}.$$

Pentru o inactivitate mai scurtă de 50 ms, menținerea perifericului în Idle este mai eficientă; pentru o perioadă mai lungă, trecerea în Sleep poate produce o economie netă.

6.7.2 Mai multe stări și praguri succesive

Un sistem real poate oferi mai multe stări:

$$S_0 = \text{Active}, \qquad S_1 = \text{Idle}, \qquad S_2 = \text{Sleep}, \qquad S_3 = \text{Deep Sleep}.$$

Fiecărei stări îi sunt asociate o putere, o latență de revenire, o energie de tranziție, un nivel de retenție și un set de surse de trezire. Pentru fiecare pereche de stări poate fi calculat un timp de break-even:

$$T_{\text{BE},i\to j} = \frac{E_{i\to j} + E_{j\to i}}{P_i - P_j}. \tag{6.45}$$

O politică bazată pe praguri poate selecta starea astfel:

$$S(T_I) = \begin{cases} S_1, & T_I < T_1, \\ S_2, & T_1 \le T_I < T_2, \\ S_3, & T_I \ge T_2. \end{cases} \tag{6.46}$$

Pragurile nu sunt determinate numai de energie — starea selectată trebuie să permită și revenirea la timp:

$$T_{\text{wake},j} + T_{\text{exec}} \leq D, \tag{6.47}$$

unde D este deadline-ul.

O stare profundă poate avea un consum foarte mic, dar poate fi invalidă dacă latența sa de revenire este prea mare. În plus, unele stări nu păstrează memoria sau nu permit trezirea prin perifericul necesar, iar selecția trebuie să respecte:

$$R_j \supseteq R_{\text{necesar}}, \tag{6.48}$$

unde R_j este setul resurselor păstrate în starea j.

6.7.3 Determinarea practică a timpului de break-even

Valorile din fișele tehnice sunt utile pentru o estimare inițială, dar timpul de break-even trebuie verificat pe sistemul real. Energia tranziției poate include salvarea registrelor, oprirea perifericelor, comutarea regulatorului, pornirea oscilatorului, stabilizarea PLL-ului, restaurarea memoriei, reinițializarea driverelor și executarea rutinelor de wake-up. Măsurarea poate fi realizată prin integrarea puterii:

$$E_T = \int_{t_0}^{t_1} V(t) I(t)\, dt. \tag{6.49}$$

Pentru identificarea intervalelor pot fi utilizați pini GPIO care marchează începutul opririi, intrarea efectivă în Sleep, începutul trezirii și reluarea aplicației. Timpul de break-even poate varia cu tensiunea bateriei, temperatura, punctul DVFS, numărul perifericelor active, cantitatea de context salvat și starea memoriei externe. Din acest motiv, o implementare robustă poate utiliza o valoare conservatoare:

$$T_{\text{prag}} = M_{\text{BE}} T_{\text{BE}}, \tag{6.50}$$

unde $M_{\text{BE}} > 1$ reprezintă o marjă de siguranță.

6.8 Politici software de management energetic

După caracterizarea stărilor energetice, software-ul trebuie să decidă momentul tranziției. Această decizie este realizată de o politică de management energetic.

O politică primește informații despre starea sistemului și selectează o acțiune:

$$\pi : \mathcal{X} \to \mathcal{A}, \tag{6.51}$$

unde $\mathcal{X}$ este spațiul stărilor observate, iar $\mathcal{A}$ este setul acțiunilor energetice.

Starea observată poate include încărcarea procesorului, lungimea cozilor, deadline-urile, temperatura, nivelul bateriei și timpul până la următorul eveniment.

Politicile pot fi reactive, predictive sau hibride [17].

6.8.1 Politici reactive și timeout

Politicile reactive iau decizia pe baza evenimentelor deja observate. Cea mai simplă metodă este politica bazată pe timeout.

După ce sistemul devine inactiv, este pornit un temporizator. Dacă nu apare o nouă activitate înaintea expirării sale, sistemul intră într-o stare mai profundă:

$$\text{Sleep} \quad \text{dacă} \quad T_{\text{idle}} \geq T_{\text{timeout}}. \tag{6.52}$$

Avantajele sunt implementarea simplă, comportamentul predictibil, costul redus de calcul și verificarea relativ ușoară. Principalul dezavantaj este energia consumată în timpul timeout-ului.

Energia pierdută înaintea intrării în Sleep este:

$$E_{\text{timeout}} = P_{\text{idle}} T_{\text{timeout}}. \tag{6.53}$$

Un timeout foarte scurt poate produce tranziții frecvente. Un timeout foarte lung poate pierde o parte importantă din perioada disponibilă pentru repaus.

Pragul trebuie să fie cel puțin comparabil cu timpul de break-even:

$$T_{\text{timeout}} \geq T_{\text{BE}}. \tag{6.54}$$

Totuși, această condiție nu garantează optimul, deoarece durata viitoare a inactivității nu este cunoscută.

Pentru mai multe stări poate fi utilizată o politică în trepte:

$$\text{Idle} \rightarrow \text{Sleep} \rightarrow \text{Deep Sleep}. \tag{6.55}$$

Fiecare tranziție este declanșată de un timeout diferit.

Figura 6.11 compară conceptual politicile reactive, predictive și hibride.

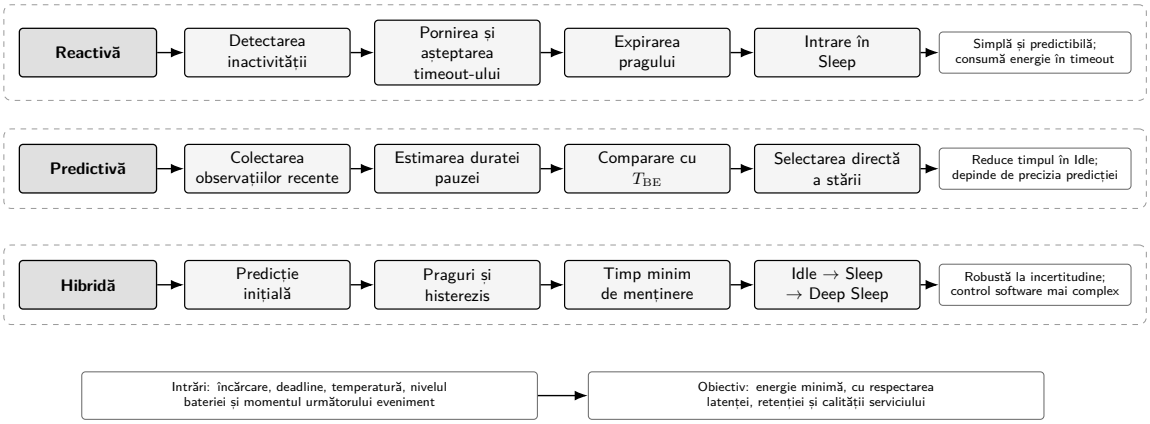


Figura 6.11 Principalele categorii de politici software de management energetic.

6.8.2 Politici predictive și pre-wakeup

O politică predictivă estimează durata viitoare a inactivității:

$$\widehat{T}_I = f\left(x_n, x_{n-1}, \ldots, x_{n-k}\right), \tag{6.56}$$

unde x_n reprezintă observațiile recente.

Starea este selectată numai dacă durata estimată depășește timpul de break-even:

$$\widehat{T}_I > T_{\text{BE}}. \tag{6.57}$$

Avantajele politicilor predictive sunt intrarea rapidă într-o stare profundă, reducerea timpului consumat în Idle și adaptarea la comportamentul aplicației. O predicție eronată poate conduce la tranziții inutile, trezire prea târzie, deadline ratat sau menținerea sistemului într-o stare prea performantă. Pentru un eveniment periodic, momentul trezirii poate fi programat înaintea deadline-ului:

$$t_{\text{wake}} = t_{\text{deadline}} - T_{\text{reactivare}} - T_{\text{execuție}} - M_T, \tag{6.58}$$

unde M_T este o marjă temporală.

Această tehnică, denumită *pre-wakeup*, permite utilizarea unei stări profunde fără expunerea aplicației la întreaga latență de revenire.

Predicția timpului de execuție poate utiliza valoarea maximă observată:

$$\widehat{T}_{\text{exec}} = \max\left(T_{n-k}, \ldots, T_n\right). \tag{6.59}$$

Metoda este conservatoare și reduce riscul încălcării deadline-urilor, dar scurtează intervalul de repaus.

6.8.3 Politici hibride, histerezis și limitarea tranzițiilor

O politică hibridă combină o predicție inițială cu praguri reactive. De exemplu:

1. dacă predictorul estimează o pauză lungă, sistemul intră direct în Deep Sleep;
2. dacă predicția este incertă, sistemul intră în Idle;
3. după un timeout, Idle este înlocuit cu Sleep;
4. dacă inactivitatea continuă, se selectează Deep Sleep.

Această abordare reduce riscul unei predicții greșite. Pentru evitarea oscilațiilor între două stări pot fi utilizate praguri diferite:

$$\begin{aligned} &\text{creștere performanț} \quad \text{dacă} \quad U > U_H, \\ &\text{reducere performanț} \quad \text{dacă} \quad U < U_L, \end{aligned} \tag{6.60}$$

unde:

$$U_L < U_H. \tag{6.61}$$

Zona dintre praguri reprezintă histerezisul. O politică trebuie să limiteze și frecvența tranzițiilor, putând fi definit un timp minim de menținere:

$$T_{\text{menținere}} \geq T_{\min}. \tag{6.62}$$

Schimbarea stării este permisă numai după expirarea acestui interval.

Costul total al politicii este:

$$J = E_{\text{total}} + \lambda_L L + \lambda_D N_D + \lambda_T N_T, \tag{6.63}$$

unde L reprezintă latența, N_D este numărul deadline-urilor ratate, N_T este numărul tranzițiilor, iar coeficienții λ exprimă penalizările asociate. O politică nu trebuie evaluată numai prin energia minimă, ci și prin predictibilitate și robustețe.

6.9 Predicția încărcării și DVFS controlat software

DVFS controlat software adaptează tensiunea și frecvența la volumul de lucru estimat. Predictorul determină performanța necesară pentru intervalul următor, iar controlerul selectează un punct de operare valid.

Fluxul general este:

$$\text{msurare} \rightarrow \text{predicție} \rightarrow \text{selecție OPP} \rightarrow \text{execuție} \rightarrow \text{actualizare}.$$

Predicția trebuie să ofere suficientă performanță pentru respectarea deadline-urilor, dar să evite selectarea inutilă a unui punct cu tensiune și frecvență ridicate.

6.9.1 Estimarea încărcării viitoare

Încărcarea unui interval poate fi definită prin:

$$W_n = \frac{T_{\text{activ},n}}{T_{\text{interval}}}. \tag{6.64}$$

O valoare de $W_n = 0{,}7$ indică faptul că procesorul a fost ocupat 70% din interval.

Poate fi utilizat și numărul de cicluri:

$$C_n = f_n T_{\text{activ},n}. \tag{6.65}$$

Această reprezentare este utilă atunci când frecvența diferă între intervale.

În continuare sunt prezentate șase strategii simple. Denumirile sunt utilizate ca etichete descriptive; ele nu reprezintă o nomenclatură standardizată în toate sistemele de operare sau platformele DVFS.

Metoda **PAST** presupune că încărcarea viitoare este egală cu cea mai recentă valoare:

$$\widehat{W}_{n+1} = W_n. \tag{6.66}$$

Metoda reacționează rapid, dar este sensibilă la variațiile bruște.

Metoda **FLAT** utilizează o valoare fixă:

$$\widehat{W}_{n+1} = W_{\text{fix}}. \tag{6.67}$$

Este stabilă, dar nu se adaptează încărcării reale, putând fi utilizată ca referință.

Metoda **LONG_SHORT** combină o medie pe termen scurt cu una pe termen lung:

$$\widehat{W}_{n+1} = \alpha \overline{W}_{\text{scurt}} + (1-\alpha)\,\overline{W}_{\text{lung}}, \tag{6.68}$$

unde:

$$0 \leq \alpha \leq 1. \tag{6.69}$$

O valoare mare a lui α oferă reacție rapidă, iar o valoare mică produce o estimare mai stabilă.

Metoda **AGED_AVERAGE** utilizează o medie exponențială:

$$\widehat{W}_{n+1} = \beta W_n + (1-\beta)\,\widehat{W}_n, \tag{6.70}$$

unde:

$$0 < \beta < 1. \tag{6.71}$$

Această metodă necesită numai predicția precedentă și ultima observație.

Metoda **CYCLE** utilizează valoarea din poziția corespunzătoare a ciclului precedent:

$$\widehat{W}_{n+1} = W_{n+1-k}, \tag{6.72}$$

unde k este lungimea perioadei.

Metoda este potrivită pentru aplicații periodice, dar poate avea rezultate slabe dacă periodicitatea se modifică.

Metoda **PEAK** selectează cea mai mare încărcare dintr-o fereastră:

$$\widehat{W}_{n+1} = \max\left(W_{n-k+1}, \ldots, W_n\right). \tag{6.73}$$

Aceasta reduce probabilitatea subestimării, dar tinde să selecteze frecvențe mai mari.

Figura 6.12 prezintă comportamentul conceptual al metodelor.

Exemplu

Se consideră încărcările recente:

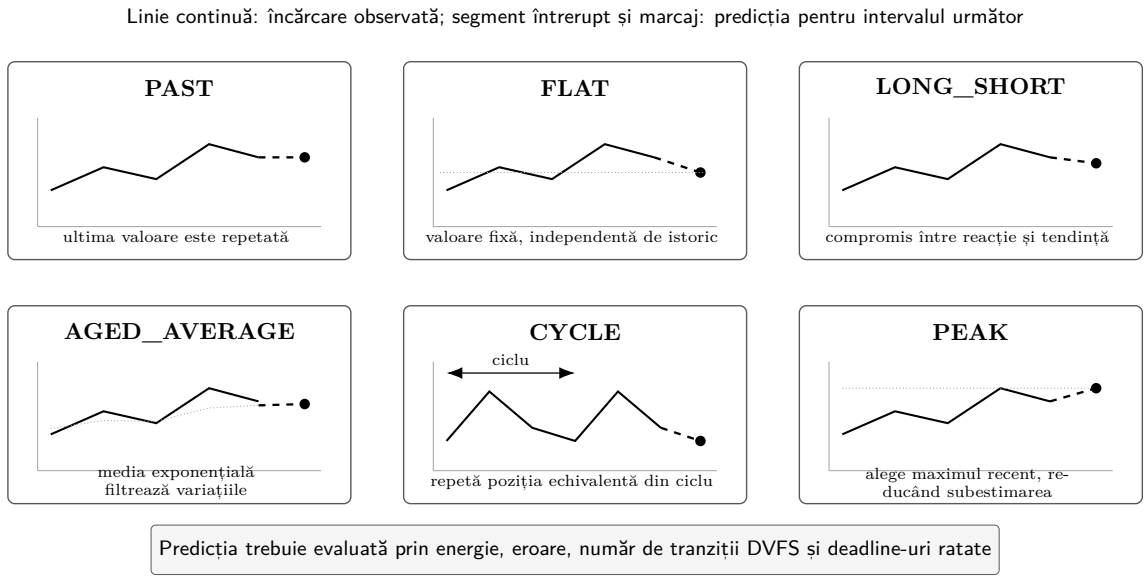


Figura 6.12 Comportamentul principalelor metode de predicție a încărcării.

$$W_{n-2} = 40\%, \qquad W_{n-1} = 55\%, \qquad W_n = 70\%.$$

Predicția PAST este:

$$\widehat{W}_{n+1} = 70\%.$$

Pentru AGED_AVERAGE, dacă predicția precedentă este 50%, iar $\beta = 0{,}3$:

$$\widehat{W}_{n+1} = 0{,}3 \cdot 70 + 0{,}7 \cdot 50 = 56\%.$$

Pentru PEAK, într-o fereastră de trei intervale:

$$\widehat{W}_{n+1} = 70\%.$$

Metodele produc rezultate diferite chiar pentru aceeași secvență de observații.

6.9.2 Selecția punctului DVFS

Presupunem că procesorul oferă punctele:

$$\mathcal{O} = \{\mathrm{OPP}_0, \mathrm{OPP}_1, \ldots, \mathrm{OPP}_{m-1}\}\,. \tag{6.74}$$

Fiecare punct conține:

$$\mathrm{OPP}_i = (V_i, f_i)\,. \tag{6.75}$$

Pentru un interval de durată D, frecvența necesară poate fi estimată prin:

$$f_{\text{necesar}} = \frac{\widehat{C}_{n+1}}{D}, \tag{6.76}$$

unde $\widehat{C}_{n+1}$ este numărul estimat de cicluri.

Se selectează cel mai mic punct care satisface:

$$f_i \geq M_f f_{\text{necesar}}, \tag{6.77}$$

unde $M_f > 1$ este o marjă.

Selecția poate fi exprimată prin:

$$i^\star = \min \{ i \mid f_i \geq M_f f_{\text{necesar}} \}. \tag{6.78}$$

Algoritmul generic este:

Exemplu 6.22 Selecția conceptuală a unui punct DVFS

```
OperatingPoint select_operating_point(
    std::uint64_t predicted_cycles,
    std::uint64_t interval_us)
{
    const double required_frequency =
        static_cast<double>(predicted_cycles) /
        static_cast<double>(interval_us);

    const double target_frequency =
        required_frequency * frequency_margin;

    for (const auto& operating_point : operating_points)
    {
        if (operating_point.frequency >= target_frequency)
        {
            return operating_point;
        }
    }

    return operating_points.back();
}
```

Lista punctelor trebuie ordonată crescător după frecvență.

Într-un sistem real, timpul disponibil trebuie redus cu durata tranziției:

$$D_{\text{util}} = D - T_{\text{DVFS}}. \tag{6.79}$$

Frecvența necesară devine:

$$f_{\text{necesar}} = \frac{\widehat{C}_{n+1}}{D - T_{\text{DVFS}}}. \tag{6.80}$$

Dacă tranziția este mai lungă decât economia potențială, schimbarea punctului nu este justificată.

6.9.3 Evaluarea predictorului și controlul erorilor

Eroarea de predicție este:

$$e_{n+1} = W_{n+1} - \widehat{W}_{n+1}. \tag{6.81}$$

Pentru:

$$e_{n+1} > 0,$$

încărcarea a fost subestimată. Aceasta este situația periculoasă pentru deadline-uri.
Pentru:

$$e_{n+1} < 0,$$

încărcarea a fost supraestimată. Sistemul poate consuma mai multă energie decât este necesar.

Pot fi definite două erori separate:

$$e^{+} = \max\left(0, W - \widehat{W}\right), \tag{6.82}$$

și:

$$e^{-} = \max\left(0, \widehat{W} - W\right). \tag{6.83}$$

Funcția de cost poate penaliza mai puternic subestimarea:

$$J_{\text{pred}} = \lambda_{+} e^{+} + \lambda_{-} e^{-}, \qquad \lambda_{+} > \lambda_{-}. \tag{6.84}$$

Pentru aplicațiile hard real-time, rata deadline-urilor ratate trebuie să fie:

$$N_{\text{deadline ratate}} = 0. \tag{6.85}$$

În aplicațiile soft real-time poate fi acceptată o rată limitată:

$$R_D = \frac{N_{\text{deadline ratate}}}{N_{\text{activități}}}. \tag{6.86}$$

Figura 6.13 prezintă bucla completă de control.

Un predictor trebuie evaluat prin energia totală, eroarea medie, eroarea maximă, rata deadline-urilor ratate, numărul schimbărilor DVFS și timpul petrecut în fiecare punct de

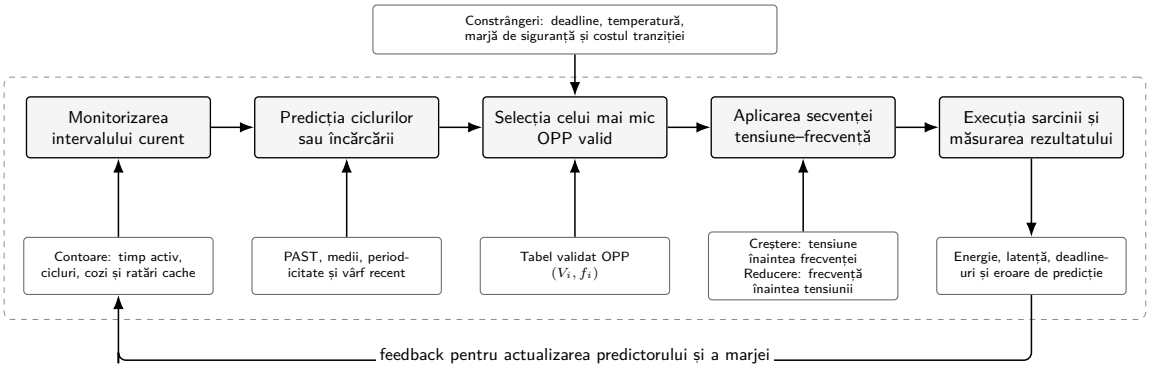


Figura 6.13 Controlul software al DVFS pe baza predicției încărcării.

Tabela 6.4 Caracteristicile metodelor simple de predicție

Metodă	Avantaj principal	Limitare principală
PAST	Reacție rapidă și implementare minimă	Sensibilitate la schimbări bruște
FLAT	Stabilitate și comportament determinist	Nu se adaptează la încărcare
LONG_SHORT	Combină reacția rapidă cu tendința pe termen lung	Necesită alegerea ferestrelor și a ponderii
AGED_AVERAGES	Cost redus și filtrarea variațiilor	Parametrul influențează puternic răspunsul
CYCLE	Potrivit pentru activitate periodică	Necesită o periodicitate stabilă
PEAK	Reduce riscul subestimării	Produce frecvent supraestimare

operare. Cea mai precisă predicție nu conduce obligatoriu la cea mai mică energie — un model complex poate avea propriul cost de calcul, iar schimbările frecvente ale punctului de operare pot anula economiile.

6.10 Management energetic în sistemele embedded moderne

Managementul energetic al unui sistem embedded modern nu este realizat de un singur algoritm și nici de o singură componentă. Consumul rezultă din interacțiunea dintre hardware, firmware, drivere, sistemul de operare și aplicații.

Hardware-ul furnizează mecanismele fizice: puncte de operare tensiune–frecvență, Clock Gating, Power Gating, moduri de repaus, domenii independente de alimentare, accelaratoare specializate și surse de trezire. Software-ul decide când și în ce condiții sunt utilizate aceste mecanisme, iar o decizie locală poate influența întregul sistem — de exemplu, reducerea frecvenței procesorului poate prelungi durata pentru care memoria externă, senzorul și transceiverul trebuie să rămână active.

Din acest motiv, obiectivul trebuie formulat la nivelul întregului sistem:

$$\min E_{\text{sistem}}, \tag{6.87}$$

sub constrângerile:

$$T_{\text{rspuns}} \leq T_{\max}, \tag{6.88}$$

$$N_{\text{deadline ratate}} \leq N_{\text{admis}}, \tag{6.89}$$

$$Q_{\text{serviciu}} \geq Q_{\min}, \tag{6.90}$$

$$T_{\text{joncțiune}} \leq T_{\max,\text{j}}. \tag{6.91}$$

Energia minimă nu trebuie obținută prin încălcarea cerințelor funcționale, temporale sau termice.

6.10.1 Coordonarea nivelurilor hardware și software

Managementul energetic poate fi organizat pe mai multe niveluri. La nivelul aplicației sunt cunoscute semnificația activității, deadline-ul și calitatea minimă acceptată — aplicația poate decide reducerea ratei de eșantionare, gruparea mesajelor sau amânarea unei activități necritice. Sistemul de operare cunoaște starea globală a procesorului, sarcinile gata de execuție, timerele și resursele utilizate, putând selecta starea de repaus, modifica frecvența și grupa trezirile. Driverele cunosc starea perifericelor și secvențele corecte de pornire, suspendare și restaurare, iar hardware-ul execută tranzițiile și poate monitoriza temperatura, tensiunea, curentul sau activitatea internă.

Figura 6.14 prezintă această organizare.

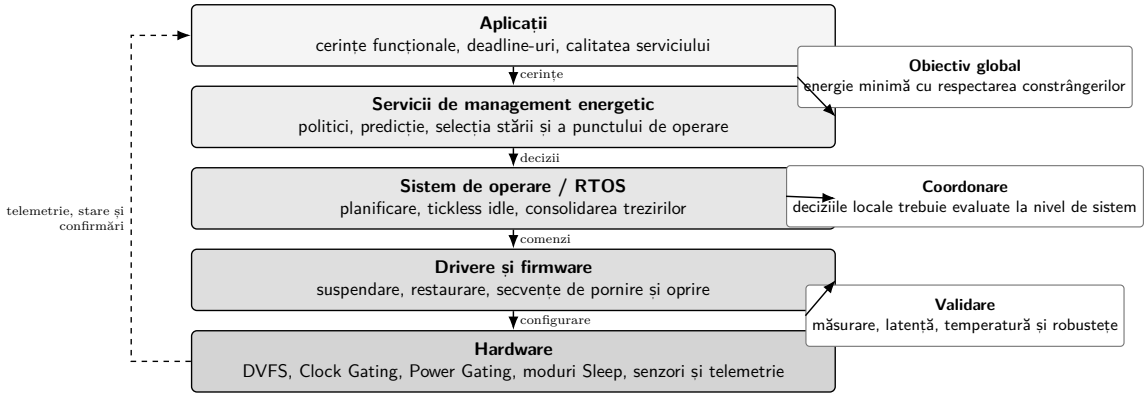


Figura 6.14 Nivelurile implicate în managementul energetic al unui sistem embedded.

O interfață power-aware trebuie să permită aplicației să declare cerința, fără a controla direct fiecare registru hardware.

De exemplu, aplicația poate solicita:

$$\text{latență} \leq 20\,\text{ms},$$

iar managerul energetic poate alege punctul DVFS și starea de repaus care respectă această cerință.

Această separare reduce dependența aplicației de o anumită platformă și permite modificarea politicii fără rescrierea funcționalității. Într-un sistem complex pot exista mai multe domenii: domeniul procesorului, domeniul memoriei, domeniul perifericelor, domeniul radio, domeniul permanent activ și domeniul unui accelerator. Tranzițiile trebuie coordonate — procesorul nu poate opri memoria dacă un controler DMA continuă un transfer, iar un modul radio nu poate fi dezactivat înaintea confirmării finalizării transmisiei.

Dependențele pot fi reprezentate printr-un graf:

$$G = (\mathcal{R}, \mathcal{D}), \tag{6.92}$$

unde $\mathcal{R}$ este setul resurselor, iar $\mathcal{D}$ conține dependențele dintre ele. Dacă resursa R_i depinde de R_j, oprirea lui R_j este permisă numai după suspendarea lui R_i.

6.10.2 Planificarea sarcinilor și arbitrarea resurselor

Planificatorul influențează energia prin ordinea și momentul executării activităților. Două strategii generale sunt distribuirea activităților pentru limitarea puterii instantanee și gruparea activităților pentru crearea unor intervale lungi de repaus. Distribuirea poate fi utilă atunci când temperatura sau puterea maximă trebuie limitate; gruparea poate fi mai eficientă atunci când starea profundă de repaus are un consum foarte mic.

Considerăm trei activități necritice, programate la momente apropiate. Executarea lor separat produce trei treziri:

$$N_{\text{wake}} = 3.$$

Dacă deadline-urile permit gruparea, ele pot fi executate într-o singură fereastră activă:

$$N_{\text{wake}} = 1.$$

Energia evitată este aproximativ:

$$E_{\text{evitat}} = 2E_{\text{tranziție}} + E_{\text{intervale inactive}}, \tag{6.93}$$

unde al doilea termen depinde de stările care devin disponibile după grupare.

Într-un sistem multicore, planificatorul poate alege între distribuirea sarcinilor pe mai multe nuclee, consolidarea lor pe un număr redus de nuclee sau oprirea nucleelor rămase neutilizate. Consolidarea este avantajoasă dacă nucleele neutilizate pot fi trecute într-o stare profundă; distribuirea poate fi preferată dacă permite reducerea tensiunii sau evitarea supraîncălzirii. Pentru fiecare activitate i pot fi definite momentul de eliberare r_i, deadline-ul

d_i, numărul de cicluri C_i, resursele necesare și criticitatea. Planificarea trebuie să satisfacă:

$$r_i \leq t_i^{\text{start}} < t_i^{\text{stop}} \leq d_i. \tag{6.94}$$

Energia procesorului pentru activitatea i poate fi aproximată prin:

$$E_i = E_i\left(V_i, f_i, C_i\right). \tag{6.95}$$

Problema constă în alegerea momentului și a punctului de operare pentru fiecare activitate, astfel încât suma energiilor să fie minimă. În practică sunt utilizate politici euristice, deoarece soluția globală poate fi costisitoare computațional — controlerul energetic trebuie să consume mult mai puțină energie decât economia pe care o produce.

6.10.3 Telemetrie, robustețe și funcționare degradată

O politică energetică nu poate lua decizii corecte fără informații despre starea sistemului.

Telemetria poate include:

- utilizarea procesorului;
- timpul petrecut în fiecare stare;
- numărul trezirilor;
- temperatura;
- tensiunea bateriei;
- curentul instantaneu;
- numărul retransmisiilor;
- deadline-urile ratate;
- erorile de predicție.

Monitorizarea introduce propriul consum. Eșantionarea prea frecventă poate produce treziri și accesări inutile.

Perioada de monitorizare trebuie aleasă astfel încât:

$$E_{\text{monitorizare}} \ll E_{\text{economisit}}. \tag{6.96}$$

Figura 6.15 prezintă integrarea monitorizării, politicii și mecanismelor hardware.

Nivelul bateriei poate fi utilizat pentru selectarea unui mod degradat. Exemplele includ:

- reducerea ratei de eșantionare;
- diminuarea frecvenței transmisiilor;
- dezactivarea funcțiilor neesențiale;
- reducerea preciziei;
- utilizarea unui predictor mai conservator;
- trecerea mai rapidă în Deep Sleep.

Politica trebuie să evite oscilațiile produse de variația tensiunii bateriei. Pot fi utilizate praguri cu histerezis:

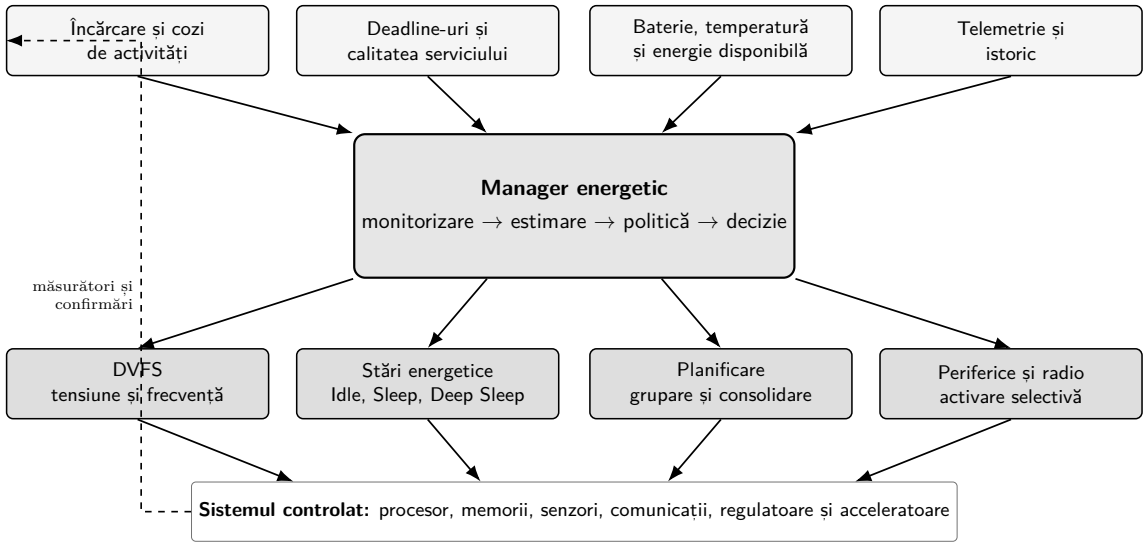


Figura 6.15 Arhitectura conceptuală a unui manager energetic modern.

$$V_{\text{revenire}} > V_{\text{degradare}}. \tag{6.97}$$

Managementul energetic trebuie să trateze și erorile.

Dacă schimbarea frecvenței eșuează, dacă oscilatorul nu se stabilizează sau dacă un periferic nu răspunde după trezire, sistemul trebuie să intre într-o stare sigură.

În aplicațiile critice, economisirea energiei nu trebuie să dezactiveze:

- mecanismele de siguranță;
- watchdog-ul necesar;
- monitorizarea tensiunii;
- sursele obligatorii de trezire;
- funcțiile de diagnostic.

6.11 Studii de caz

Studiile de caz din această secțiune folosesc valori explicite pentru a ilustra metoda de calcul. Valorile active depind de dispozitiv, placa de dezvoltare, regulator, periferice, puterea radio și condițiile de mediu.

Rezultatele nu trebuie interpretate drept garanții de autonomie. Ele reprezintă estimări inițiale care trebuie validate prin măsurători.

6.11.1 Nod IoT Wi-Fi bazat pe ESP32

Se consideră un nod care măsoară temperatura și transmite rezultatul prin Wi-Fi o dată la 15 minute.

Durata ciclului este:

$$T_C = 900\,\text{s}. \tag{6.98}$$

Pentru scenariul ilustrativ se presupun:

- curent activ mediu: $I_A = 120\,\text{mA}$;
- durată activă: $T_A = 3\,\text{s}$;
- curent în Deep Sleep: $I_S = 0{,}010\,\text{mA}$;
- durată în Deep Sleep: $T_S = 897\,\text{s}$.

Valoarea activă de 120 mA este o ipoteză la nivelul sistemului și include procesarea și comunicația. Ea nu reprezintă o valoare universală pentru toate variantele ESP32 și toate plăcile.

Curentul mediu este:

$$I_{\text{mediu}} = \frac{I_A T_A + I_S T_S}{T_C}. \tag{6.99}$$

Prin înlocuire:

$$\begin{aligned} I_{\text{mediu}} &= \frac{120 \cdot 3 + 0{,}010 \cdot 897}{900} \\ &\approx 0{,}410\,\text{mA}. \end{aligned} \tag{6.100}$$

Figura 6.16 prezintă ciclul energetic.

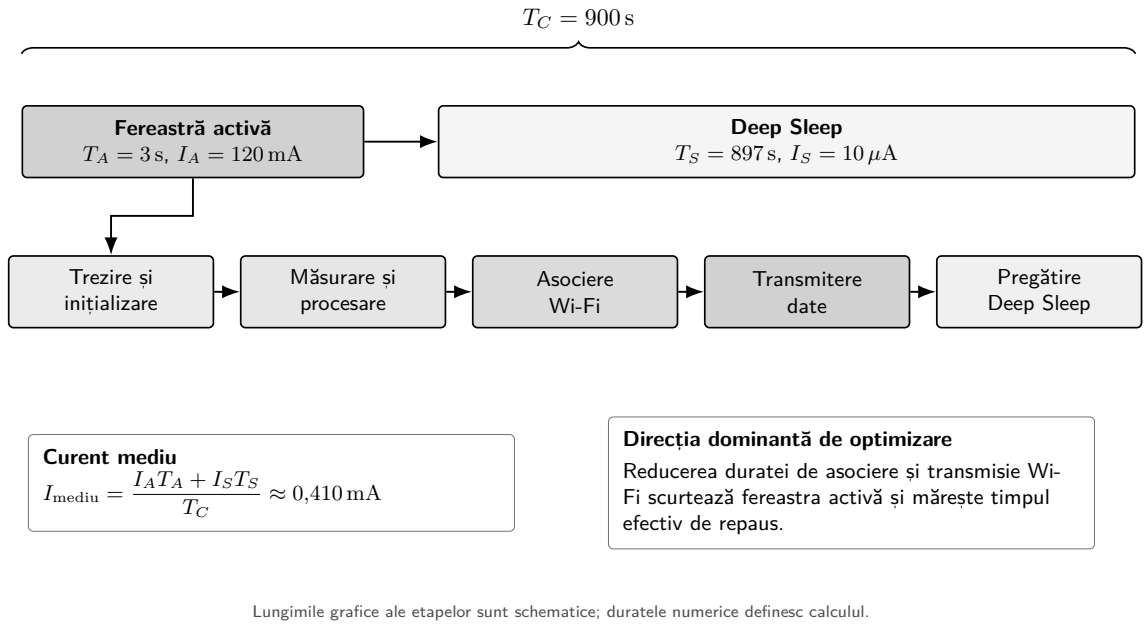


Figura 6.16 Ciclul de funcționare al unui nod IoT Wi-Fi bazat pe ESP32.

Pentru o baterie ideală cu capacitatea:

$$C_B = 2400\,\text{mAh},$$

autonomia teoretică este:

$$T_{\text{ideal}} = \frac{C_B}{I_{\text{mediu}}}. \tag{6.101}$$

Rezultă:

$$T_{\text{ideal}} \approx 5854\,\text{h} \approx 244\,\text{zile}.$$

Această valoare nu include:

- randamentul regulatorului;
- autoconsumul plăcii;
- convertorul USB–serial;
- LED-urile;
- variația duratei conectării Wi-Fi;
- retransmisiile;
- autodescărcarea bateriei;
- efectul temperaturii.

Pe o placă de dezvoltare, consumul în Deep Sleep poate fi mult mai mare decât consumul circuitului integrat, din cauza componentelor auxiliare.

Optimizarea principală constă frecvent în reducerea duratei conexiunii Wi-Fi. Dacă timpul activ este redus la:

$$T'_A = 1{,}5\,\text{s},$$

iar timpul de repaus devine:

$$T'_S = 898{,}5\,\text{s},$$

curentul mediu este:

$$\begin{aligned} I'_{\text{mediu}} &= \frac{120 \cdot 1{,}5 + 0{,}010 \cdot 898{,}5}{900} \\ &\approx 0{,}210\,\text{mA}. \end{aligned} \tag{6.102}$$

Reducerea la jumătate a timpului activ reduce curentul mediu cu aproape 49% în scenariul analizat.

Acest rezultat arată că optimizarea protocolului, memorarea parametrilor de conectare și reducerea timpului de asociere pot avea un impact mai mare decât optimizarea unor instrucțiuni izolate.

6.11.2 Senzor industrial cu STM32L4 și comunicație LoRa

Se consideră un senzor industrial care măsoară temperatura și transmite un pachet LoRa la fiecare 15 minute.

Platforma conceptuală conține:

- microcontroler din familia STM32L4;
- senzor digital;
- transceiver LoRa;
- baterie de 2400 mAh.

Pentru fiecare ciclu se presupun:

Tabela 6.5 Parametrii energetici ai nodului LoRa

Stare	Curent	Durată
Procesare și măsurare	8 mA	1 s
Transmisie LoRa	45 mA	2 s
Deep Sleep	0,005 mA	897 s

Valorile sunt ilustrative. Curentul și durata transmisiei depind de puterea de emisie, spreading factor, lățimea de bandă, dimensiunea pachetului și retransmisii.

Curentul mediu este:

$$I_{\text{mediu}} = \frac{8 \cdot 1 + 45 \cdot 2 + 0{,}005 \cdot 897}{900}. \tag{6.103}$$

Rezultă:

$$I_{\text{mediu}} \approx 0{,}114\,\text{mA}. \tag{6.104}$$

Autonomia ideală este:

$$T_{\text{ideal}} = \frac{2400}{0{,}114} \approx 21053\,\text{h}. \tag{6.105}$$

Aceasta corespunde aproximativ:

$$877\,\text{zile} \approx 2{,}4\,\text{ani}.$$

Figura 6.17 prezintă distribuția temporală a stărilor.

Deși transmisia ocupă numai două secunde, contribuția sa la sarcina electrică este:

$$Q_{\text{TX}} = 45\,\text{mA} \cdot 2\,\text{s} = 90\,\text{mA s}.$$

Contribuția Deep Sleep este:

$$Q_{\text{sleep}} = 0{,}005\,\text{mA} \cdot 897\,\text{s} = 4{,}485\,\text{mA s}.$$

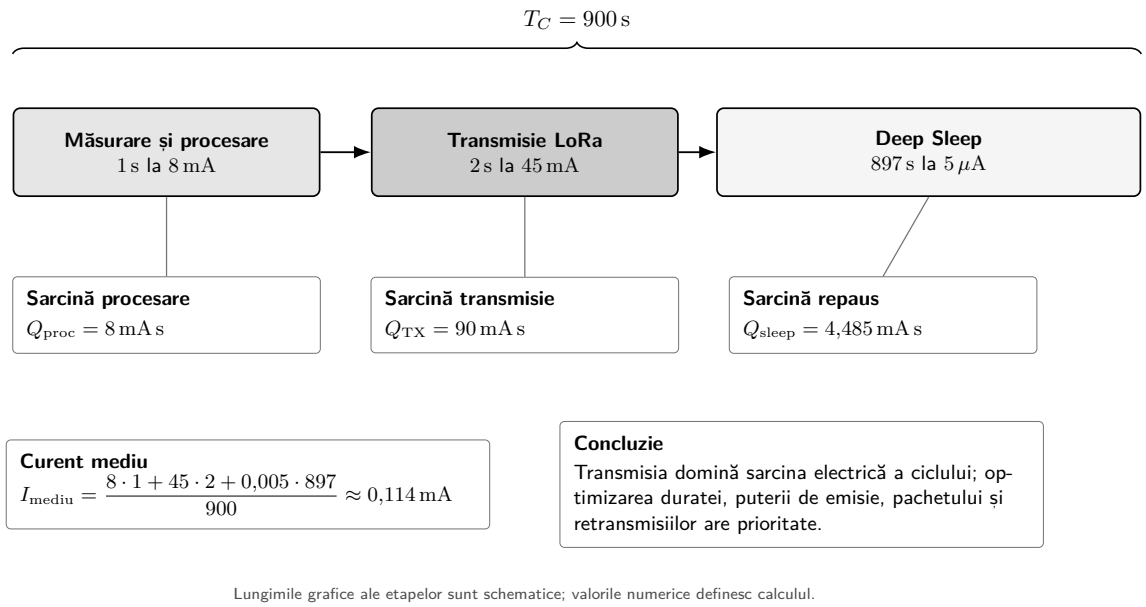


Figura 6.17 Ciclul energetic al unui senzor industrial cu comunicație LoRa.

Prin urmare, transmisia domină consumul ciclului.

Direcțiile de optimizare sunt:

- reducerea duratei transmisiei;
- reducerea puterii de emisie atunci când legătura permite;
- gruparea măsurătorilor;
- reducerea dimensiunii pachetului;
- evitarea retransmisiilor;
- adaptarea intervalului de raportare.

Dacă perioada de transmitere este dublată, consumul mediu asociat transmisiilor se reduce aproximativ la jumătate, presupunând că restul condițiilor rămân constante.

Totuși, reducerea frecvenței raportării poate afecta detecția rapidă a unui eveniment. O soluție este utilizarea unei politici adaptive:

- raportare rară în regim normal;
- raportare frecventă la depășirea unui prag;
- transmisie imediată pentru o alarmă.

6.11.3 Analiza sensibilității și autonomia realistă

Estimarea autonomiei prin raportul dintre capacitate și curentul mediu este utilă, dar incompletă.

O estimare îmbunătățită poate utiliza:

$$T_{\mathrm{real}} = \frac{k_C C_{\mathrm{nominal}}}{I_{\mathrm{mediu}} + I_{\mathrm{pierderi}}}, \tag{6.106}$$

unde:

- k_C este fracția utilizabilă din capacitatea nominală;
- I_{pierderi} include regulatorul și circuitele auxiliare.

Factorul k_C poate include efectele temperaturii, îmbătrânirii, curentului de descărcare și tensiunii minime acceptate de sistem.

Pentru studiul LoRa, dacă se presupune:

$$k_C = 0{,}75$$

și:

$$I_{\text{pierderi}} = 0{,}010\,\text{mA},$$

rezultă:

$$\begin{aligned} T_{\text{real}} &= \frac{0{,}75 \cdot 2400}{0{,}114 + 0{,}010} \\ &\approx 14516\,\text{h} \\ &\approx 605\,\text{zile}. \end{aligned} \tag{6.107}$$

Estimarea scade de la aproximativ 2,4 ani la aproximativ 1,66 ani.

Analiza sensibilității arată ce parametru trebuie optimizat. Pentru un ciclu format din n stări:

$$I_{\text{mediu}} = \frac{\sum_{i=1}^{n} I_i T_i}{T_C}. \tag{6.108}$$

Sensibilitatea față de durata stării j este aproximativ:

$$\frac{\partial I_{\text{mediu}}}{\partial T_j} = \frac{I_j - I_{\text{mediu}}}{T_C}. \tag{6.109}$$

O stare cu un curent mult mai mare decât media produce un impact important chiar pentru o variație mică a duratei.

Pentru validare trebuie măsurate:

- distribuția reală a timpului de conectare;
- numărul retransmisiilor;
- curentul regulatorului în repaus;
- curentul senzorului după oprire;
- consumul pinilor și rezistențelor;
- variația cu temperatura;
- capacitatea efectivă a bateriei.

Valorile tipice din fișa tehnică nu trebuie înlocuite cu valorile maxime în toate calculele, dar nici utilizate fără o marjă. Pentru dimensionarea autonomiei trebuie analizate un scenariu nominal și unul nefavorabil.

6.12 Tendințe actuale în optimizarea energetică software

Creșterea complexității sistemelor embedded a condus la politici energetice care combină modele statistice, telemetrie, acceleratoare și surse variabile de energie. Obiectivul nu mai este numai reducerea consumului mediu — sistemele moderne trebuie să gestioneze simultan energia, temperatura, performanța, durata bateriei, disponibilitatea energiei recoltate, îmbătrânirea componentelor și calitatea serviciului.

6.12.1 Management predictiv și adaptiv

Politicile predictive simple folosesc istoricul recent sau periodicitatea. Sistemele moderne pot utiliza modele mai complexe pentru estimarea încărcării procesorului, momentului următoarei cereri, duratei comunicației, energiei disponibile, temperaturii viitoare sau probabilității unei alarme. Un manager adaptiv își actualizează parametrii în timpul funcționării:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n + \Delta\boldsymbol{\theta}_n, \tag{6.110}$$

unde $\boldsymbol{\theta}$ reprezintă parametrii predictorului sau ai politicii.

Actualizarea trebuie limitată pentru a evita instabilitatea — o schimbare temporară a sarcinii nu trebuie să producă modificarea excesivă a politicii. Metodele de învățare automată pot surprinde relații neliniare, dar introduc memorie pentru model, calcule suplimentare, date pentru antrenare, dificultăți de verificare și risc de comportament neașteptat. Energia netă economisită trebuie să satisfacă:

$$E_{\text{economie}} > E_{\text{predicție}} + E_{\text{monitorizare}} + E_{\text{adaptare}}. \tag{6.111}$$

Pentru aplicațiile critice sunt importante limitele explicite și o politică de rezervă — dacă predictorul nu este sigur, sistemul poate selecta un punct de operare conservator.

6.12.2 Edge AI și utilizarea acceleratoarelor

Aplicațiile Edge AI execută local clasificare, detecție, estimare sau control. Procesarea locală poate reduce comunicația cu un server, dar introduce o sarcină de calcul importantă.

Energia unei inferențe poate fi exprimată prin:

$$E_{\text{inferență}} = E_{\text{calcul}} + E_{\text{memorie}} + E_{\text{senzor}} + E_{\text{transfer}}. \tag{6.112}$$

Reducerea exclusivă a numărului operațiilor nu garantează energia minimă — dimensiunea modelului și accesările memoriei pot domina. Tehnicile utilizate includ cuantizare, pruning, distilarea modelului, activare condiționată, inferență în mai multe etape, acceleratoare specializate și procesare în apropierea senzorului.

Un clasificator în cascadă poate utiliza mai întâi un model simplu. Modelul complex este activat numai pentru exemplele incerte:

$$\text{model} = \begin{cases} M_{\text{simplu}}, & c \geq c_{\min}, \\ M_{\text{complex}}, & c < c_{\min}, \end{cases} \tag{6.113}$$

unde c este încrederea rezultatului.

Această strategie reduce energia medie dacă majoritatea cazurilor pot fi rezolvate de primul model. Metricile relevante sunt inferențe/J, energie per inferență corectă, latența inferenței, memorie ocupată, acuratețe și energia întregului lanț senzor–decizie.

Figura 6.18 prezintă principalele direcții moderne.

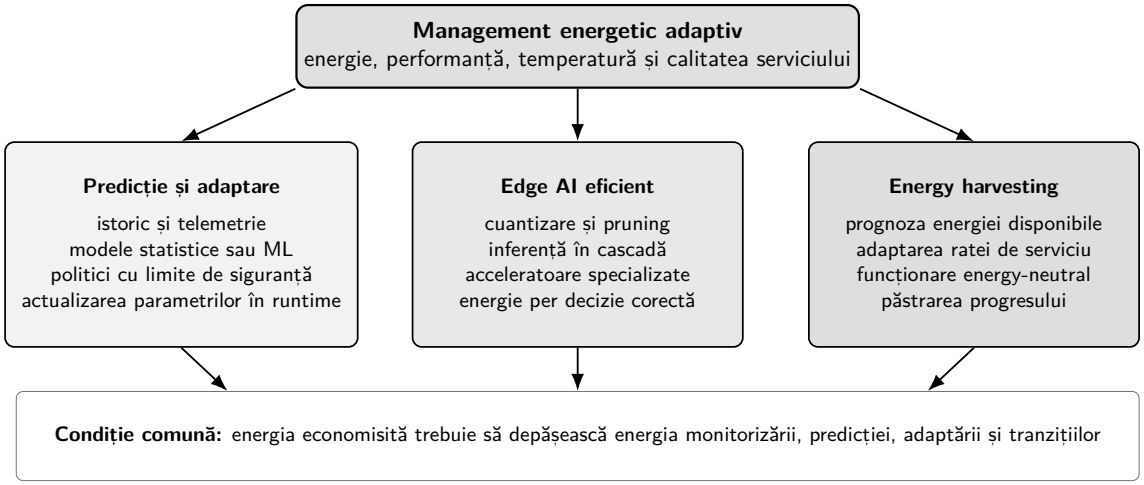


Figura 6.18 Direcții actuale în optimizarea energetică software.

6.12.3 Energy harvesting și funcționarea energy-neutral

Unele sisteme recuperează energie din mediul înconjurător: lumină, vibrații, diferențe termice, unde radio sau mișcare. În aceste sisteme, energia disponibilă variază în timp, iar software-ul trebuie să adapteze activitatea la energia stocată și la energia estimată pentru viitor.

Pentru un interval T, condiția de funcționare sustenabilă este:

$$E_{\text{consumat}} \leq E_{\text{recoltat}} + E_{\text{stocat,inițial}} - E_{\text{rezerv}}. \tag{6.114}$$

Pe termen lung, un sistem energy-neutral urmărește:

$$\overline{P}_{\text{consumat}} \leq \overline{P}_{\text{recoltat}}. \tag{6.115}$$

Puterea instantanee poate depăși puterea recoltată dacă diferența este furnizată de un acumulator sau supercondensator. Politica poate ajusta rata de eșantionare, numărul inferențelor, precizia algoritmului, frecvența transmisiei, puterea radio sau activarea unor

funcții opționale.

Poate fi definit un nivel de serviciu Q în funcție de energia disponibilă:

$$Q = g\left(E_{\text{stocat}}, \widehat{E}_{\text{recoltat}}\right). \tag{6.116}$$

Când energia este abundentă, sistemul oferă o rată mare de măsurare; când energia scade, trece într-un regim de conservare. O dificultate importantă este păstrarea progresului la pierderea alimentării — sistemele intermitente trebuie să salveze starea în memorie nevolatilă, dar salvarea prea frecventă consumă energie și uzează memoria.

6.13 Întrebări și probleme propuse

6.13.1 Întrebări de verificare

1. De ce două aplicații care rulează pe aceeași platformă pot avea consumuri energetice diferite?
2. Care este diferența dintre reducerea puterii și reducerea energiei?
3. De ce complexitatea algoritmică nu determină singură consumul energetic?
4. În ce condiții un tip de date cu dimensiune redusă poate crește timpul de execuție?
5. De ce loop unrolling nu este întotdeauna avantajos energetic?
6. Care este diferența dintre localitatea temporală și localitatea spațială?
7. În ce situații utilizarea DMA este eficientă energetic?
8. De ce împachetarea forțată a structurilor poate produce penalizări?
9. Ce înseamnă o aplicație power-aware?
10. Cum contribuie gruparea mesajelor la reducerea consumului radio?
11. Care este diferența dintre un model energetic bazat pe instrucțiuni și unul bazat pe stări?
12. De ce un model energetic trebuie calibrat pentru mai multe tipuri de sarcini?
13. Care este diferența generală dintre Idle, Sleep și Deep Sleep?
14. Ce informații trebuie verificate înaintea intrării într-o stare de consum redus?
15. Ce reprezintă timpul de break-even?
16. De ce energia tranziției trebuie să includă atât intrarea, cât și revenirea?
17. Care sunt avantajele și limitările unei politici bazate pe timeout?
18. Ce rol are histerezisul într-o politică energetică?

19. Cum diferă metodele PAST și AGED_AVERAGE?

20. În ce tip de aplicație este potrivit predictorul CYCLE?

21. De ce metoda PEAK este conservatoare?

22. Cum este transformată predicția încărcării într-un punct DVFS?

23. De ce subestimarea și supraestimarea încărcării au consecințe diferite?

24. Cum poate planificatorul crea intervale mai lungi de repaus?

25. De ce autonomia ideală calculată din capacitate și curent mediu poate fi prea optimistă?

26. Ce condiție trebuie satisfăcută pentru funcționarea energy-neutral?

6.13.2 Probleme de calcul și analiză

1. Un sistem consumă 40 mW în Idle și 0,5 mW în Sleep. Intrarea și revenirea consumă împreună 0,79 mJ. Calculați timpul de break-even.

2. Un nod execută o activitate timp de 200 ms la 25 mA și rămâne în Deep Sleep timp de 59,8 s la 8 μA. Calculați curentul mediu.

3. Pentru nodul de la problema anterioară, calculați autonomia ideală pentru o baterie de 2000 mAh.

4. Recalculați autonomia presupunând că numai 75% din capacitatea nominală este utilizabilă și regulatorul consumă suplimentar 15 μA.

5. Un procesor oferă frecvențele 40, 80, 120 și 160 MHz. O activitate necesită $6 \cdot 10^6$ cicluri și are un deadline de 60 ms. Selectați frecvența minimă care respectă deadline-ul, fără marjă.

6. Repetați calculul anterior pentru o marjă de frecvență de 20%.

7. O aplicație produce următoarele încărcări: 30%, 40%, 65%, 50%. Calculați predicția PAST și predicția PEAK pentru o fereastră de patru intervale.

8. Pentru aceeași aplicație, calculați AGED_AVERAGE dacă predicția precedentă este 45%, ultima încărcare este 50%, iar $\beta = 0{,}25$.

9. Un transceiver consumă 50 mA timp de 1 s pentru fiecare transmisie. Comparați contribuția la curentul mediu pentru transmisii realizate la fiecare minut și la fiecare 15 minute.

10. Zece măsurători de câte 24 de biți sunt transmise separat. Fiecare pachet are un antet de 96 de biți. Calculați numărul total de biți și comparați-l cu gruparea într-un singur pachet.

11. Un sistem are trei stări cu puterile: $P_{\text{Idle}} = 20\,\text{mW}$, $P_{\text{Sleep}} = 2\,\text{mW}$ și $P_{\text{Deep}} = 0{,}1\,\text{mW}$. Energia tranziției Idle–Sleep–Idle este 0,36 mJ, iar energia Idle–Deep–Idle este 1,99 mJ. Calculați pragurile de break-even.

12. Propuneți o politică în trepte pentru sistemul de la problema anterioară și justificați pragurile alese.

13. Analizați o aplicație în care un senzor necesită 100 ms pentru stabilizare și este citit la fiecare 150 ms. Discutați dacă oprirea sa după fiecare citire este justificată.

14. Proiectați o politică pentru un nod agricol alimentat de un panou fotovoltaic. Politica trebuie să adapteze rata de măsurare la energia stocată și la prognoza energiei recoltate.

15. Comparați două implementări ale unui algoritm: prima consumă 30 mW timp de 20 ms, iar a doua consumă 45 mW timp de 10 ms. Determinați energia fiecărei variante și indicați implementarea mai eficientă.

7. Modele de Execuție și Planificare în Sistemele Embedded de Timp Real

Introducere în sistemele embedded de timp real
Constrângeri temporale, predictibilitate și clase de sisteme
Modele de execuție software în sistemele embedded
Rolul și structura unui sistem de operare de timp real
Task-uri, stări și comutarea contextului
Multitasking și organizarea aplicației
Multitasking și strategii de planificare
Analiza schedulabilității și algoritmi clasici de planificare

7.1 Introducere în sistemele embedded de timp real

Numeroase sisteme embedded interacționează direct cu procese fizice. Acestea achiziționează informații de la senzori, execută algoritmi de prelucrare și generează comenzi pentru actuatoare, iar valoarea rezultatului nu este determinată numai de corectitudinea sa logică, ci și de momentul în care acesta devine disponibil.

Într-o aplicație convențională, o întârziere poate fi percepută doar ca o scădere a performanței. Într-un sistem de timp real, aceeași întârziere poate conduce la pierderea stabilității unei bucle de control, la omiterea unui eveniment sau la activarea tardivă a unui mecanism de protecție — un rezultat corect furnizat după momentul la care putea fi utilizat poate fi, din perspectiva sistemului, echivalent cu un rezultat incorect.

Sistemele de timp real nu sunt definite prin faptul că execută foarte rapid. O platformă lentă poate constitui un sistem de timp real dacă oferă limite temporale cunoscute și respectă toate termenele impuse; în același timp, un procesor foarte performant poate fi nepotrivit dacă timpul său de răspuns variază într-un mod care nu poate fi controlat sau analizat.

7.1.1 Corectitudinea logică și temporală

Corectitudinea unui sistem de timp real are două componente:

$$\text{corectitudine} = \text{corectitudine logic} + \text{corectitudine temporal}. \tag{7.1}$$

Corectitudinea logică arată că algoritmul produce rezultatul dorit pentru datele de intrare, iar corectitudinea temporală arată că rezultatul este produs în intervalul în care acesta este util.

Considerăm o activitate software care trebuie să determine comanda $u(t)$ pe baza unei măsurători $y(t)$. Rezultatul logic poate fi corect, însă comanda este aplicată după o întârziere Δt:

$$u_{\text{aplicat}}(t) = u_{\text{calculat}}(t - \Delta t). \tag{7.2}$$

Dacă întârzierea depășește limita acceptată de procesul controlat, funcționarea poate deveni instabilă sau periculoasă. Un sistem de timp real trebuie să ofere limite pentru duratele importante — în loc să se cunoască numai timpul mediu de execuție, este necesară o limită superioară:

$$T_{\text{exec}} \leq T_{\text{exec,max}}. \tag{7.3}$$

În mod similar, timpul de răspuns trebuie să respecte:

$$R_i \leq D_i, \tag{7.4}$$

unde R_i este timpul de răspuns al activității i, iar D_i este deadline-ul relativ al acesteia. Timpul mediu poate fi util pentru evaluarea performanței generale, dar nu demonstrează respectarea unui deadline — un sistem poate avea un timp mediu redus și, ocazional, întârzieri foarte mari.

Exemplu

O funcție de control este executată de 1000 de ori. În 999 de cazuri, timpul de răspuns este:

$$R = 0{,}5\,\text{ms}.$$

Într-un singur caz, timpul de răspuns este:

$$R = 25\,\text{ms}.$$

Timpul mediu este aproximativ:

$$R_{\text{mediu}} = \frac{999 \cdot 0{,}5 + 25}{1000} \approx 0{,}5245\,\text{ms}.$$

Dacă deadline-ul este de 5 ms, valoarea medie pare foarte bună, dar sistemul nu respectă cerința temporală în toate cazurile.

În sistemele de timp real sunt importante limita maximă a timpului de execuție, latența întreruperilor, timpul de blocare pe resurse, timpul de comutare a contextului, interferența produsă de activitățile cu prioritate mai mare și variația temporală introdusă de hardware

și software. Analiza temporală trebuie efectuată pentru scenariul nefavorabil credibil, nu numai pentru cazul mediu măsurat în timpul dezvoltării.

7.1.2 Sistemul reactiv și lanțul senzor–decizie–actuator

Sistemele embedded de timp real sunt frecvent sisteme reactive — ele nu execută un calcul izolat, ci răspund continuu la modificări ale mediului.

Modelul general conține:

1. procesul fizic;
2. senzorii;
3. achiziția și condiționarea semnalelor;
4. algoritmul de estimare sau control;
5. actuatoarele;
6. feedback-ul către proces.

Figura 7.1 prezintă acest lanț.

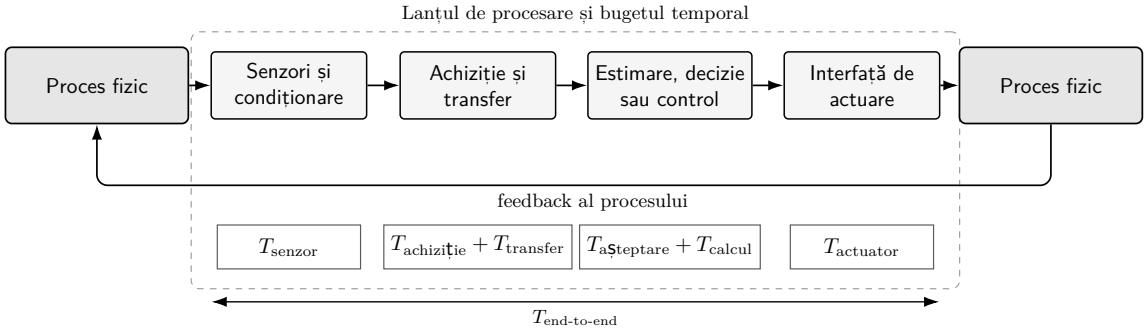


Figura 7.1 Lanțul temporal al unui sistem embedded reactiv.

Latența totală dintre apariția unui fenomen și aplicarea comenzii poate fi exprimată prin:

$$T_{\text{end-to-end}} = T_{\text{senzor}} + T_{\text{achiziție}} + T_{\text{comunicație}} + T_{\text{așteptare}} + T_{\text{calcul}} + T_{\text{actuator}}. \tag{7.5}$$

Optimizarea exclusivă a algoritmului nu garantează îndeplinirea cerinței end-to-end — dacă senzorul actualizează rezultatul lent sau mesajul este întârziat de rețea, timpul total poate rămâne prea mare.

Pentru un sistem de control se poate impune:

$$T_{\text{end-to-end}} \leq D_{\text{control}}. \tag{7.6}$$

Lanțul trebuie analizat integral, fiecare componentă primind un buget temporal:

$$D_{\text{control}} = D_{\text{senzor}} + D_{\text{transfer}} + D_{\text{software}} + D_{\text{actuator}} + M_T, \tag{7.7}$$

unde M_T reprezintă o marjă temporală.

Un sistem reactiv poate răspunde la evenimente:

- periodice, precum o buclă de control executată la fiecare 10 ms;
- sporadice, precum apariția unei erori;
- aperiodice, precum o comandă transmisă de utilizator;
- sincrone sau asincrone față de ceasul procesorului.

Aceste tipuri de activare influențează arhitectura software și metoda de planificare.

Exemplu

Exemplu: robot mobil.

Un robot detectează un obstacol la distanța d. Dacă viteza sa este v, timpul ideal disponibil până la atingerea obstacolului este:

$$T_{\text{disponibil}} = \frac{d}{v}.$$

Pentru:

$$d = 0{,}5\,\text{m}, \qquad v = 1\,\text{m/s},$$

rezultă:

$$T_{\text{disponibil}} = 0{,}5\,\text{s}.$$

Din acest interval trebuie scăzute timpul de frânare, întârzierea actuatorului, achiziția senzorului și marja de siguranță — software-ul nu dispune de întreaga durată de 0,5 s pentru luarea deciziei.

7.1.3 Domenii de aplicare și cerințe asociate

Sistemele de timp real sunt întâlnite în numeroase domenii, însă nivelul criticității și consecințele întârzierilor diferă. În automotive, aplicațiile includ controlul motorului, frânarea, controlul stabilității, direcția asistată, managementul bateriei și sistemele avansate de asistență. În automatizări industriale sunt relevante controlul mișcării, sincronizarea roboților, protecțiile, achiziția deterministă și comunicațiile industriale.

În domeniul medical, întârzierea poate afecta monitorizarea, administrarea unui tratament sau activarea unei alarme. În aviație și transport feroviar, predictibilitatea temporală este asociată direct funcțiilor de siguranță. În aplicațiile multimedia, depășirea ocazională a unui deadline poate produce pierderea unui cadru, fără a conduce la defectarea completă a sistemului.

Cerințele trebuie formulate cantitativ — o expresie precum „sistemul trebuie să răspundă

rapid" nu poate fi verificată. O cerință temporală verificabilă poate fi:

> Pentru fiecare activare a funcției de protecție, comanda actuatorului trebuie generată în maximum 2 ms, în toate condițiile de operare definite.

Cerința trebuie să precizeze evenimentul care pornește măsurarea, evenimentul care marchează finalizarea, limita temporală, condițiile de operare și comportamentul la depășirea limitei.

7.2 Constrângeri temporale, predictibilitate și clase de sisteme

Descrierea riguroasă a unei aplicații de timp real necesită parametri care definesc activarea, execuția și termenul limită al fiecărei activități.

O activitate software poate fi modelată ca un task τ_i:

$$\tau_i = (C_i, T_i, D_i, J_i)\,, \tag{7.8}$$

unde C_i este timpul de execuție în cazul cel mai nefavorabil, T_i este perioada sau intervalul minim dintre activări, D_i este deadline-ul relativ, iar J_i este jitter-ul de activare. Modelul poate fi extins cu prioritate, timp de blocare, offset și resurse partajate.

7.2.1 Parametrii temporali ai unei activități

Pentru instanța k a task-ului τ_i sunt definite momentul de activare sau eliberare $r_{i,k}$, momentul începerii execuției $s_{i,k}$, momentul finalizării $f_{i,k}$ și deadline-ul absolut $d_{i,k}$. Deadline-ul absolut este:

$$d_{i,k} = r_{i,k} + D_i. \tag{7.9}$$

Timpul de răspuns este:

$$R_{i,k} = f_{i,k} - r_{i,k}. \tag{7.10}$$

Timpul de așteptare înaintea primei execuții este:

$$W_{i,k} = s_{i,k} - r_{i,k}. \tag{7.11}$$

Pentru un task preemptiv, execuția poate fi împărțită în mai multe intervale. În acest caz, timpul de răspuns include execuția proprie, preempțiile, blocarea, întârzierile scheduler-ului și timpul de comutare a contextului. Respectarea deadline-ului necesită:

$$f_{i,k} \leq d_{i,k}. \tag{7.12}$$

Lateness-ul este definit prin:

$$L_{i,k} = f_{i,k} - d_{i,k}. \tag{7.13}$$

O valoare negativă arată că task-ul a fost finalizat înaintea deadline-ului. Tardiness-ul consideră numai depășirile:

$$\mathcal{T}_{i,k} = \max\left(0, L_{i,k}\right). \tag{7.14}$$

Figura 7.2 prezintă relația dintre acești parametri.

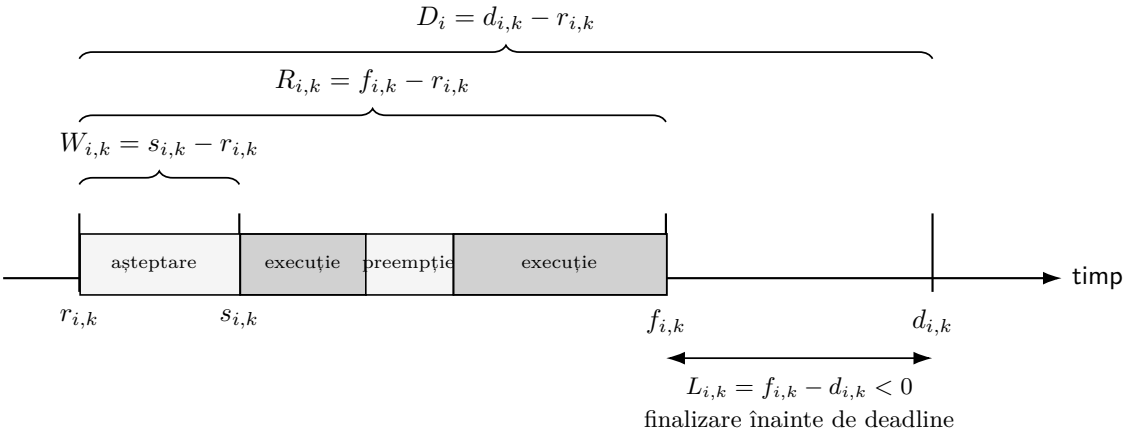


Figura 7.2 Parametrii temporali ai unei instanțe de task.

Pentru un task periodic, activările ideale sunt:

$$r_{i,k} = r_{i,0} + kT_i. \tag{7.15}$$

Într-un sistem real, activarea poate prezenta jitter:

$$r_{i,k} = r_{i,0} + kT_i + \delta_{i,k}, \tag{7.16}$$

unde:

$$|\delta_{i,k}| \leq J_i. \tag{7.17}$$

Timpul de execuție C_i trebuie interpretat ca WCET, de la *Worst-Case Execution Time*. Determinarea sa poate fi dificilă deoarece durata este influențată de:

- căile de execuție ale programului;
- valorile datelor;
- memoria cache;
- pipeline;
- predicția ramificațiilor;
- accesările memoriei;
- întreruperile și DMA;
- frecvența procesorului.

Măsurarea repetată poate identifica valori mari, dar nu demonstrează automat că a fost observat cazul cel mai nefavorabil.

7.2.2 Determinism, latență și jitter

Determinismul temporal reprezintă capacitatea de a stabili limite cunoscute pentru momentele importante ale execuției.

Un sistem determinist nu trebuie să producă exact aceeași durată la fiecare execuție. Este suficient ca variația să fie limitată și compatibilă cu cerințele.

Pentru un timp de răspuns cu valorile minimă și maximă:

$$R_{\min} \leq R \leq R_{\max}, \tag{7.18}$$

jitter-ul de răspuns poate fi exprimat prin:

$$J_R = R_{\max} - R_{\min}. \tag{7.19}$$

Figura 7.3 compară două sisteme cu aceeași valoare medie, dar cu variații diferite.

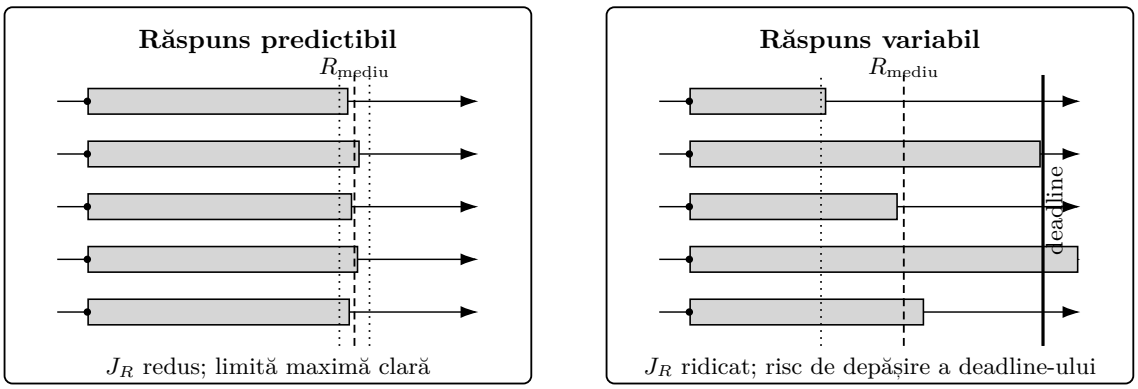


Figura 7.3 Influența jitter-ului asupra predictibilității temporale.

Într-o buclă de control periodică, jitter-ul modifică intervalul de eșantionare:

$$T_k = T_{\text{nominal}} + \Delta T_k. \tag{7.20}$$

Dacă variația este mare, algoritmul proiectat pentru o perioadă constantă poate avea performanțe mai slabe. Sursele de nedeterminism pot include întreruperi cu durată variabilă, secțiuni critice lungi, alocare dinamică, memorii cache, colectare automată a memoriei, blocare pe resurse, comunicații cu timp variabil, drivere care folosesc polling blocant și management energetic cu schimbări DVFS. Aceste mecanisme nu sunt interzise în mod absolut, dar influența lor trebuie analizată și limitată.

Rapiditatea și determinismul sunt proprietăți distincte. Considerăm două implementări:

$$R_A \in [1, 10]\,\text{ms},$$

și:

$$R_B \in [4, 5]\,\text{ms}.$$

Implementarea A poate fi uneori mai rapidă, dar implementarea B este mai ușor de utilizat într-un sistem cu deadline de 6 ms. Pentru verificarea temporală este necesară limita maximă, nu numai cea mai bună valoare observată.

7.2.3 Clase de sisteme și modele de activare

Sistemele de timp real sunt clasificate frecvent în hard, firm și soft real-time. Diferența este determinată de consecința depășirii deadline-ului, nu numai de probabilitatea producerii acesteia [21, 50].

Într-un sistem **hard real-time**, depășirea unui deadline relevant este considerată o încălcare a cerinței sistemului, iar consecința poate fi pierderea controlului, activarea tardivă a unei protecții sau apariția unui pericol. Într-un sistem **firm real-time**, rezultatul nu mai are valoare după deadline, dar depășirea ocazională nu produce neapărat defectarea completă a sistemului — rezultatul întârziat este abandonat. Într-un sistem **soft real-time**, valoarea rezultatului scade după deadline, însă acesta poate rămâne utilizabil, întârzierile fiind reflectate în calitatea serviciului.

Această diferență poate fi descrisă printr-o funcție de utilitate $U(t)$ asociată momentului finalizării.

Pentru hard real-time:

$$U_{\text{hard}}(t) = \begin{cases} U_0, & t \le d, \\ -\infty, & t > d. \end{cases} \tag{7.21}$$

Pentru firm real-time:

$$U_{\text{firm}}(t) = \begin{cases} U_0, & t \le d, \\ 0, & t > d. \end{cases} \tag{7.22}$$

Pentru soft real-time, utilitatea poate scădea gradual după deadline.

Tabela 7.1 Clasificarea sistemelor în funcție de consecința depășirii deadline-ului

Clasă	Valoarea rezultatului întârziat	Consecință tipică
Hard real-time	Încălcarea limitei este inacceptabilă	Cerință critică nerespectată sau stare periculoasă
Firm real-time	Rezultatul devine inutil după deadline	Eșantion, cadru sau decizie abandonată
Soft real-time	Rezultatul își pierde gradual valoarea	Reducerea calității serviciului

Clasificarea trebuie aplicată funcțiilor, nu obligatoriu întregului produs. Același sistem

poate conține control hard real-time, comunicație firm real-time, diagnosticare soft real-time și logare fără deadline strict.

Activitățile pot fi clasificate și după modul de activare. Un task **periodic** este activat la intervale regulate T_i.

Un task **sporadic** este activat de evenimente, dar există un interval minim între două activări:

$$r_{i,k+1} - r_{i,k} \geq T_i. \tag{7.23}$$

Un task **aperiodic** nu are o separare minimă cunoscută — pentru analiza unui sistem critic, evenimentele aperiodice trebuie limitate prin mecanisme precum cozi, servere de execuție sau controlul ratei.

7.3 Modele de execuție software în sistemele embedded

Arhitectura de execuție stabilește modul în care funcțiile software primesc procesorul și răspund la evenimente. Alegerea depinde de numărul activităților, frecvența evenimentelor, deadline-uri, resursele disponibile, necesitatea izolării, complexitatea sincronizării și cerințele de mentenanță. Principalele modele sunt bucla principală, executivul ciclic, execuția bazată pe întreruperi, arhitectura Foreground/Background și multitasking-ul controlat de un RTOS.

RTOS-ul va fi analizat în blocurile următoare. În această secțiune sunt prezentate modelele bare-metal și tranziția către o arhitectură multitasking.

Figura 7.4 oferă o vedere comparativă.

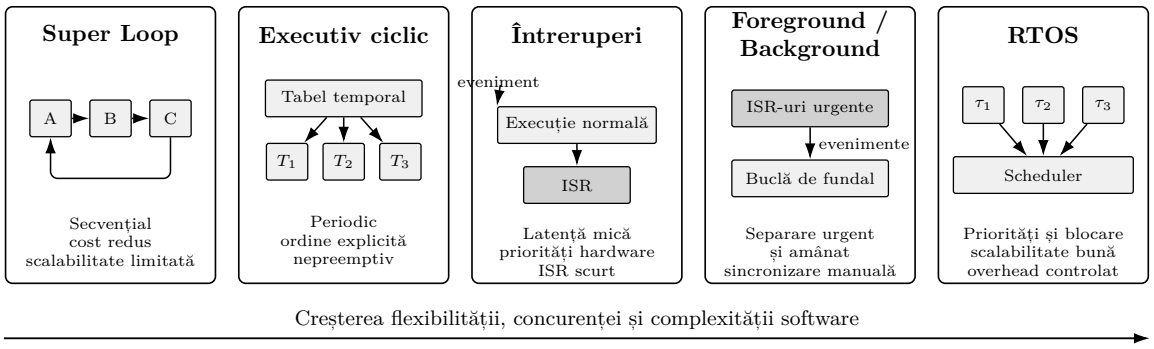


Figura 7.4 Modele de execuție software utilizate în sistemele embedded.

7.3.1 Bucla principală și executivul ciclic

Cel mai simplu model este bucla principală, denumită frecvent *Super Loop*. Funcțiile sunt apelate secvențial și ciclul se repetă.

Exemplu 7.1 Structura de bază a unei aplicații Super Loop

```
1 int main()
2 {
```

```
3       initialize_hardware();
4
5       while (true)
6       {
7           read_sensors();
8           process_data();
9           update_actuators();
10          service_communication();
11      }
12  }
```

Avantajele sunt:

- structură simplă;
- memorie ocupată redusă;
- lipsa comutărilor de context;
- depanare relativ simplă;
- ordine explicită a operațiilor.

Predictibilitatea este bună numai dacă toate funcțiile:

- au durată limitată;
- nu se blochează pe termen nedefinit;
- nu conțin bucle de așteptare necontrolate;
- nu depind de periferice cu durată necunoscută.

Durata unei iterații este:

$$T_{\text{loop}} = \sum_{i=1}^{n} C_i + C_{\text{control}}, \tag{7.24}$$

unde C_{control} include verificările și administrarea buclei.

Dacă un eveniment este verificat o singură dată în fiecare iterație, întârzierea de polling poate ajunge aproape de:

$$L_{\text{poll,max}} \approx T_{\text{loop}}. \tag{7.25}$$

Timpul până la finalizarea tratării evenimentului include și funcția sa de procesare.

Figura 7.5 prezintă situația în care un eveniment apare imediat după verificarea sa.

Exemplu

Se consideră funcțiile:

$$C_{\text{senzori}} = 1\,\text{ms},$$

$$C_{\text{procesare}} = 4\,\text{ms},$$

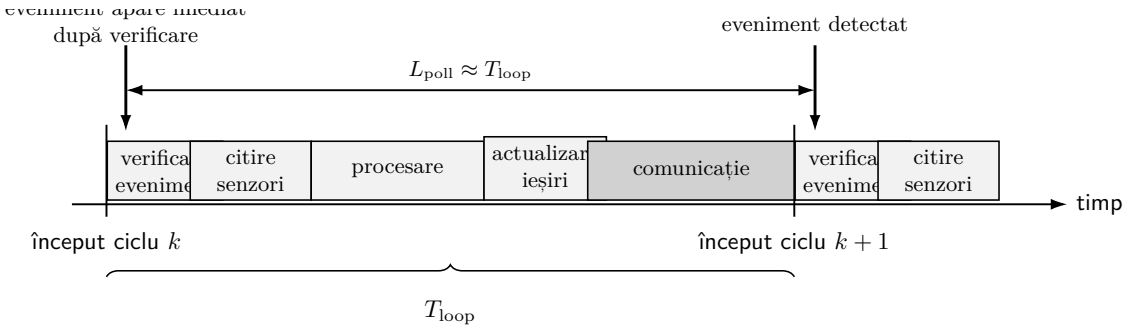


Figura 7.5 Latența maximă produsă de verificarea periodică într-un Super Loop.

$$C_{\text{actuatoare}} = 1\,\text{ms},$$

și:

$$C_{\text{comunicație}} = 14\,\text{ms}.$$

Durata buclei este:

$$T_{\text{loop}} = 1 + 4 + 1 + 14 = 20\,\text{ms}.$$

Dacă un eveniment este verificat numai la începutul buclei și apare imediat după verificare, acesta poate aștepta aproape 20 ms înainte de a fi observat. La această valoare se adaugă timpul necesar procesării sale.
Rezultatul este valabil numai dacă toate duratele sunt limitate — dacă funcția de comunicație poate aștepta nedefinit, nu mai poate fi garantată o latență maximă.

Pentru activități cu perioade diferite poate fi utilizat un executiv ciclic, în care o bază de timp declanșează funcțiile conform unui tabel.

Exemplu 7.2 Executiv ciclic simplificat

```
1  void scheduler_tick()
2  {
3      const std::uint32_t now = system_time_ms();
4
5      if (time_reached(now, next_control))
6      {
7          next_control += control_period;
8          run_control();
9      }
10
11     if (time_reached(now, next_sensor))
12     {
```

```
13          next_sensor += sensor_period;
14          read_sensors();
15      }
16
17      if (time_reached(now, next_diagnostic))
18      {
19          next_diagnostic += diagnostic_period;
20          run_diagnostic();
21      }
22  }
```

Compararea timpilor trebuie implementată astfel încât să funcționeze corect și la revenirea contorului de timp la zero. Un executiv ciclic este potrivit când activitățile sunt în principal periodice, ordinea este cunoscută, numărul funcțiilor este redus, timpii de execuție sunt bine limitați și nu este necesară preempția între funcții. Limitarea principală este blocarea nepreemptivă — o funcție lungă întârzie toate funcțiile care urmează.

7.3.2 Execuția bazată pe întreruperi

Întreruperile permit tratarea rapidă a evenimentelor asincrone — la apariția unei întreruperi, procesorul suspendă temporar execuția curentă și rulează o rutină ISR.

Secvența conceptuală este:

1. perifericul sau controlerul de întreruperi semnalizează evenimentul;
2. procesorul finalizează sau întrerupe instrucțiunea curentă;
3. este salvat contextul necesar;
4. este determinată rutina ISR;
5. ISR-ul confirmă și tratează evenimentul;
6. contextul este restaurat;
7. execuția întreruptă continuă.

Latența întreruperii poate fi exprimată prin:

$$L_{\text{IRQ}} = T_{\text{mascare}} + T_{\text{instrucțiune}} + T_{\text{priorități}} + T_{\text{intrare}}. \tag{7.26}$$

Termenii pot include timpul în care întreruperile sunt dezactivate, finalizarea instrucțiunii curente, interferența ISR-urilor cu prioritate mai mare, precum și salvarea contextului și accesarea vectorului. Durata totală până la finalizarea reacției este:

$$R_{\text{IRQ}} = L_{\text{IRQ}} + C_{\text{ISR}} + C_{\text{deferred}}, \tag{7.27}$$

unde C_{deferred} reprezintă procesarea amânată în afara ISR-ului.

ISR-urile trebuie să execute numai activitățile strict necesare: confirmarea sursei, citirea sau scrierea datelor urgente, salvarea informației care s-ar pierde și semnalizarea procesării ulterioare. Operații precum filtrarea complexă, formatarea mesajelor, scrierea în Flash și așteptarea unui periferic trebuie evitate în ISR.

Figura 7.6 prezintă separarea dintre tratarea imediată și procesarea amânată.

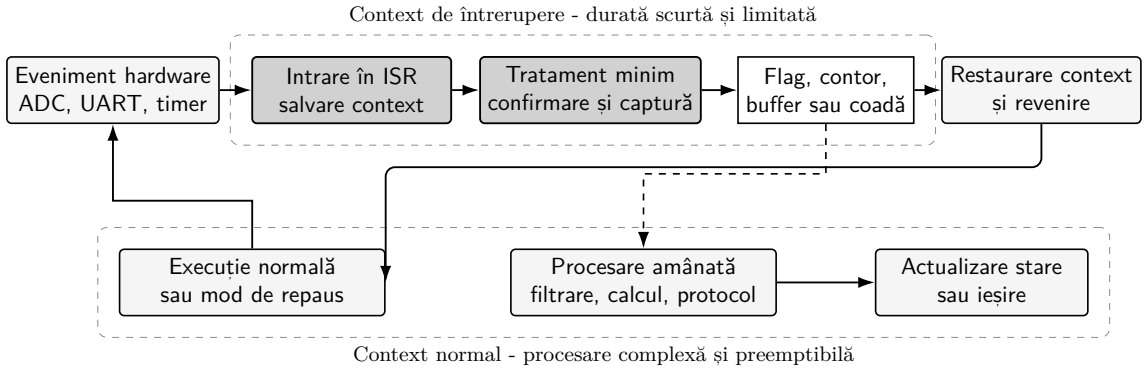


Figura 7.6 Tratarea rapidă a întreruperii și procesarea amânată.

Un exemplu C++ poate utiliza un indicator atomic:

Exemplu 7.3 Semnalizarea sigură a unui eveniment din ISR

```
1  #include <atomic>
2
3  static_assert(
4      std::atomic_bool::is_always_lock_free,
5      "Atomic operations used in ISR must be lock-free.");
6
7  std::atomic_bool measurement_ready{false};
8
9  extern "C" void ADC_IRQHandler()
10 {
11     acknowledge_adc_interrupt();
12
13     measurement_ready.store(
14         true,
15         std::memory_order_release);
16 }
17
18 void process_background()
19 {
```

```
20      if (measurement_ready.exchange(
21              false,
22              std::memory_order_acq_rel))
23      {
24          process_measurement();
25      }
26  }
```

Exemplul presupune că operația atomică este lock-free pe platforma țintă. Într-un proiect concret pot fi utilizate primitivele furnizate de microcontroler, secțiuni critice scurte sau mecanismele RTOS dedicate comunicației din ISR.

Simpla declarare a unei variabile ca `volatile` obligă compilatorul să efectueze accesările, dar nu garantează în mod general atomicitate, sincronizare sau protecție împotriva pierderii evenimentelor.

Un flag boolean poate pierde evenimente dacă întreruperea apare de mai multe ori înainte ca procesarea să îl reseteze. Pentru evenimente repetabile pot fi utilizate:

- un contor;
- un buffer circular;
- o coadă;
- DMA;
- registre hardware cu flag-uri persistente.

O furtună de întreruperi poate ocupa întregul procesor. Rata maximă trebuie să satisfacă aproximativ:

$$U_{\text{ISR}} = \sum_{i=1}^{m} \lambda_i C_{\text{ISR},i} < 1, \tag{7.28}$$

unde λ_i este rata maximă a întreruperii i. În practică trebuie păstrată capacitate și pentru codul principal.

7.3.3 Arhitectura Foreground/Background și alegerea modelului

Arhitectura Foreground/Background combină întreruperile cu o buclă principală. Foreground-ul este format din ISR-uri și tratează evenimentele cu cerințe stricte de latență, iar background-ul execută activitățile amânate și funcțiile care nu necesită o reacție imediată. Structura conceptuală este:

Exemplu 7.4 Arhitectură Foreground/Background

```
1  int main()
2  {
3      initialize_hardware();
4      enable_interrupts();
5
6      while (true)
```

```
7       {
8           process_adc_events();
9           process_communication_events();
10          run_diagnostics();
11          update_logs();
12          enter_idle_if_possible();
13      }
14  }
```

ISR-urile nu execută procesarea complexă, ci introduc datele în buffere sau setează evenimente. Avantajele sunt latența redusă pentru evenimente urgente, memoria ocupată mai mică decât într-un RTOS, structura relativ simplă, posibilitatea intrării procesorului în repaus și separarea procesării urgente de cea necritică. Limitările sunt faptul că background-ul este în general nepreemptiv, o funcție lungă întârzie celelalte activități de background, prioritizarea în background trebuie implementată manual, datele partajate cu ISR-urile necesită protecție, iar scalabilitatea scade odată cu creșterea numărului de funcții.

Un model Foreground/Background nu conține obligatoriu numai două niveluri hardware de prioritate. Controlerul de întreruperi poate permite mai multe priorități și întreruperi imbricate. Totuși, din perspectiva aplicației, distincția principală rămâne între codul de întrerupere și bucla de fundal.

Tabela 7.2 Compararea modelelor software de execuție

Model	Avantaj principal	Limitare temporală	Aplicații potrivite
Super Loop	Simplitate și cost redus	O funcție lungă întârzie întregul ciclu	Aplicații mici, predominant secvențiale
Executiv ciclic	Ordine și perioade explicite	Execuție nepreemptivă și tabel rigid	Activități periodice bine cunoscute
Întreruperi	Latență redusă la evenimente	Interferență, imbricare și risc de suprasarcină	Evenimente asincrone și achiziții urgente
Foreground/Background	Separarea reacției urgente de procesare	Background nepreemptiv și sincronizare manuală	Aplicații bare-metal de complexitate medie
RTOS	Task-uri, priorități și sincronizare	Overhead și complexitate de configurare	Aplicații concurente și extensibile

Alegerea unui RTOS nu trebuie făcută numai deoarece aplicația conține mai multe funcții. Un sistem bare-metal poate fi suficient atunci când funcționalitatea este redusă, activitățile sunt bine cunoscute, timpii sunt ușor de limitat, nu există multe resurse partajate și mentenanța rămâne gestionabilă. Un RTOS devine justificat când aplicația necesită mai multe activități concurente, priorități independente, blocare eficientă pe evenimente, cozi

și mecanisme de sincronizare, temporizatoare și deadline-uri multiple, precum și separarea clară a componentelor software.

Complexitatea unui sistem nu este redusă automat prin introducerea unui RTOS — ea este reorganizată în task-uri, priorități, resurse și relații de sincronizare. O arhitectură incorectă poate produce comportamente mai dificil de analizat decât un executiv ciclic bine proiectat.

7.4 Rolul și structura unui sistem de operare de timp real

Pe măsură ce numărul activităților software crește, organizarea lor într-o singură buclă principală devine dificilă. Aplicația trebuie să gestioneze activități cu perioade, priorități și condiții de activare diferite, precum achiziția periodică a datelor, controlul actuatoarelor, tratarea comunicațiilor, diagnosticarea sistemului, înregistrarea evenimentelor și actualizarea interfeței cu utilizatorul.

Un sistem de operare de timp real, denumit RTOS (*Real-Time Operating System*), oferă mecanisme pentru organizarea acestor activități sub forma unor unități independente de execuție.

RTOS-ul nu garantează automat respectarea deadline-urilor — el furnizează mecanisme predictibile de planificare, sincronizare și temporizare, însă arhitectura aplicației, prioritățile și timpii de execuție trebuie proiectate și analizate corect. Principalele responsabilități ale unui RTOS sunt gestionarea task-urilor, planificarea accesului la procesor, gestionarea timpului, sincronizarea activităților, comunicația între task-uri, integrarea cu întreruperile și administrarea resurselor software.

7.4.1 Nivelul de abstractizare oferit de RTOS

Într-o aplicație bare-metal, programatorul controlează direct ordinea apelării funcțiilor, verificarea evenimentelor și accesul la periferice. Într-un sistem bazat pe RTOS, aceste activități sunt distribuite între task-uri, iar kernelul decide momentul în care fiecare task primește procesorul.

Figura 7.7 prezintă poziția RTOS-ului între aplicație și hardware.

O arhitectură tipică poate fi organizată pe următoarele niveluri:

1. hardware;
2. drivere și Hardware Abstraction Layer;
3. kernel RTOS;
4. servicii middleware;
5. task-uri ale aplicației.

Hardware Abstraction Layer-ul, prescurtat HAL, ascunde o parte dintre particularitățile microcontrolerului; kernelul oferă primitive pentru planificare și sincronizare, iar middleware-ul poate implementa protocoale de comunicație, sisteme de fișiere sau stive de rețea.

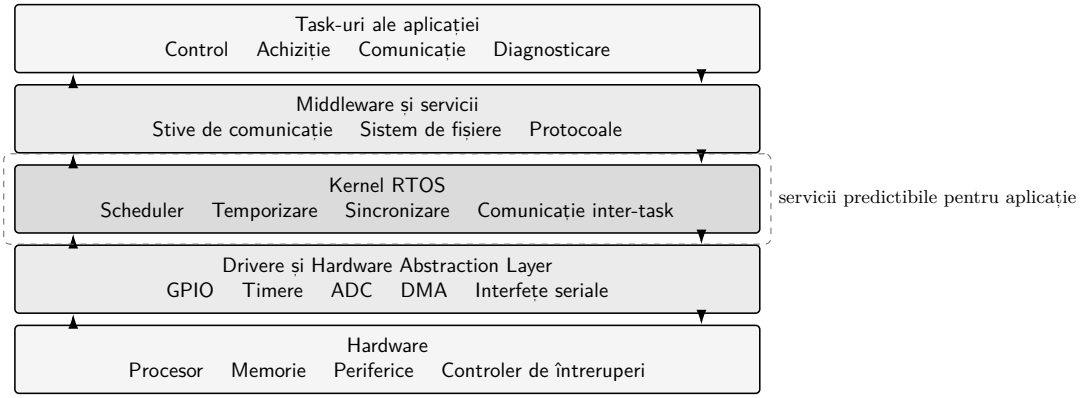


Figura 7.7 Poziția RTOS-ului în arhitectura unui sistem embedded.

Aplicația nu mai trebuie să verifice permanent toate evenimentele într-o buclă. Un task poate fi blocat până la apariția evenimentului necesar:

$$\text{așteptare} \rightarrow \text{eveniment} \rightarrow \text{execuție}.$$

Blocarea unui task nu este echivalentă cu o buclă de așteptare activă. În timpul blocării, procesorul poate executa alt task sau poate intra într-o stare de consum redus.

Un RTOS permite și separarea funcționalităților. De exemplu, într-un sistem de control pot exista:

- un task pentru senzori;
- un task pentru algoritmul de control;
- un task pentru comunicație;
- un task pentru diagnosticare.

Această separare îmbunătățește modularitatea, dar introduce dependențe temporale și necesitatea comunicării între task-uri.

RTOS-ul nu trebuie privit ca o soluție pentru orice aplicație. Pentru un sistem foarte simplu, un executiv ciclic poate avea o structură mai clară și un cost mai mic. Utilizarea RTOS-ului este justificată atunci când beneficiile planificării, sincronizării și modularității depășesc costurile introduse.

7.4.2 Kernel, scheduler și dispatcher

Kernelul reprezintă componenta centrală a RTOS-ului. Acesta păstrează informațiile despre task-uri și implementează mecanismele prin care acestea sunt create, suspendate, blocate și reactivate.

Scheduler-ul este componenta care selectează următorul task ce trebuie executat. Decizia este realizată pe baza politicii de planificare.

Într-un sistem cu priorități fixe, regula generală este:

$$\tau_{\text{selectat}} = \arg\max_{\tau_i \in \mathcal{R}} (p_i), \tag{7.29}$$

unde:

- $\mathcal{R}$ este mulțimea task-urilor Ready;
- p_i este prioritatea task-ului τ_i.

Dacă mai multe task-uri au aceeași prioritate, kernelul poate utiliza:

- Round Robin;
- ordinea activării;
- un time slice;
- o regulă specifică implementării.

Dispatcher-ul aplică decizia scheduler-ului. El realizează transferul efectiv al procesorului de la task-ul curent către task-ul selectat.

Procesul include:

1. salvarea contextului task-ului curent;
2. actualizarea stării acestuia;
3. selectarea contextului următor;
4. restaurarea registrelor;
5. reluarea execuției task-ului selectat.

Figura 7.8 prezintă relația dintre task-uri, scheduler și dispatcher.

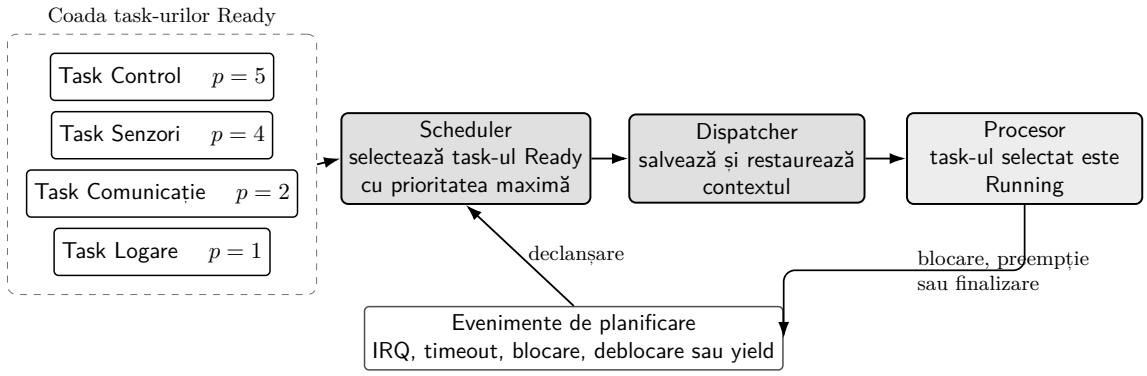


Figura 7.8 Relația dintre scheduler, dispatcher și task-urile aplicației.

Scheduler-ul poate fi activat de expirarea unui timer, blocarea task-ului curent, deblocarea unui task cu prioritate mai mare, finalizarea unei operații, o întrerupere, cedarea voluntară a procesorului sau modificarea priorității unui task. Durata dintre apariția evenimentului și începerea execuției task-ului selectat include:

$$L_{\text{schedule}} = L_{\text{IRQ}} + C_{\text{kernel}} + C_{\text{switch}}, \tag{7.30}$$

unde L_{IRQ} este latența tratării întreruperii, C_{kernel} este timpul deciziei de planificare, iar C_{switch} este timpul comutării contextului. Într-un RTOS destinat aplicațiilor de timp real, aceste valori trebuie să fie limitate și documentate.

7.4.3 Servicii temporale și integrarea cu întreruperile

Kernelul utilizează o bază de timp pentru gestionarea întârzierilor, timeout-urilor și activităților periodice.

O implementare clasică folosește un tick periodic cu perioada:

$$T_{\text{tick}} = \frac{1}{f_{\text{tick}}}. \tag{7.31}$$

Pentru:

$$f_{\text{tick}} = 1000\,\text{Hz},$$

rezultă:

$$T_{\text{tick}} = 1\,\text{ms}.$$

Rezoluția temporală nu este identică în toate situațiile cu precizia temporală — un timeout de un tick poate expira într-un interval dependent de momentul la care a fost solicitat.

Într-un model simplificat, întârzierea reală pentru n tick-uri poate fi:

$$(n-1)T_{\text{tick}} < T_{\text{ntrziere}} \leq nT_{\text{tick}}. \tag{7.32}$$

Valoarea exactă depinde de semantica API-ului și de implementarea kernelului. Pentru activități periodice este preferabilă programarea în raport cu un reper absolut, deoarece o întârziere relativă introdusă după finalizarea activității poate acumula o deplasare temporală.

O implementare conceptuală bazată pe întârziere relativă este:

Exemplu 7.5 Task cu întârziere relativă

```
1 void SensorTask(void* argument)
2 {
3     while (true)
4     {
5         read_sensors();
6         delay_ticks(sensor_period_ticks);
7     }
8 }
```

Perioada reală devine aproximativ:

$$T_{\text{real}} = C_{\text{senzor}} + T_{\text{delay}}. \tag{7.33}$$

Pentru menținerea perioadei față de un reper se poate utiliza:

Exemplu 7.6 Task periodic raportat la un reper temporal

```
1  void SensorTask(void* argument)
2  {
3      TickType next_release = get_current_tick();
4
5      while (true)
6      {
7          read_sensors();
8
9          delay_until(
10             next_release,
11             sensor_period_ticks);
12     }
13 }
```

Funcția este conceptuală, API-ul exact depinzând de RTOS. Întreruperile continuă să fie utilizate și într-un sistem cu RTOS — diferența este că ISR-ul poate debloca un task care așteaptă evenimentul.

Fluxul este:

$$\text{periferic} \rightarrow \text{ISR} \rightarrow \text{semnalizare} \rightarrow \text{task Ready} \rightarrow \text{scheduler}.$$

Dacă task-ul deblocat are prioritate mai mare decât task-ul curent, poate fi realizată o preempție imediată după ieșirea din ISR.

Nu toate funcțiile kernelului pot fi apelate din întreruperi. RTOS-urile oferă frecvent variante speciale ale API-urilor pentru ISR, deoarece ISR-ul nu poate fi blocat, contextul său este diferit de cel al unui task, schimbarea planificării trebuie amânată până la ieșirea din ISR, iar anumite structuri interne necesită protecție specială.

Regula generală este:

> Din ISR trebuie apelate numai funcțiile documentate explicit ca fiind sigure pentru contextul de întrerupere.

7.5 Task-uri, stări și comutarea contextului

Task-ul reprezintă unitatea de execuție planificată de kernel. Acesta poate fi considerat un flux independent de control care dispune de o funcție de intrare, o stivă proprie, o prioritate, o stare de planificare, un context al procesorului și informații pentru sincronizare și temporizare.

În literatura de specialitate sunt utilizați termenii *task* și *thread*. Semnificația exactă depinde de sistemul de operare — într-un RTOS pentru microcontrolere, cei doi termeni descriu frecvent unități similare de execuție care partajează același spațiu de adrese.

7.5.1 Modelul task-ului și Task Control Block

Un task este definit printr-o funcție care, în mod obișnuit, conține o buclă principală:

Exemplu 7.7 Structura generică a unui task

```
void ControlTask(void* argument)
{
    initialize_control();

    while (true)
    {
        wait_for_control_event();
        read_control_inputs();
        compute_control_command();
        update_actuators();
    }
}
```

Task-ul nu verifică permanent evenimentul prin polling activ. Funcția `wait_for_control_event()` blochează task-ul până la producerea evenimentului.

Pentru fiecare task, kernelul păstrează o structură denumită *Task Control Block*, prescurtat TCB.

TCB-ul poate conține:

- indicatorul către stiva task-ului;
- prioritatea de bază;
- prioritatea efectivă;
- starea curentă;
- poziția în listele scheduler-ului;
- informații despre timeout;
- numele sau identificatorul task-ului;
- statistici de execuție;
- informații de protecție a memoriei.

Figura 7.9 prezintă conceptual relația dintre task, TCB și stivă.

Stiva task-ului păstrează variabile locale, adresele de revenire, parametrii funcțiilor, registre salvate și contextul necesar comutării. Dimensiunea stivei trebuie aleasă în funcție de adâncimea apelurilor, dimensiunea variabilelor locale, utilizarea bibliotecilor, tratarea întreruperilor, arhitectura procesorului și scenariul de execuție nefavorabil. O stivă prea mică poate produce coruperea memoriei, iar o stivă excesivă consumă RAM fără beneficiu.

În sistemele embedded se utilizează frecvent mecanisme precum umplerea inițială a stivei

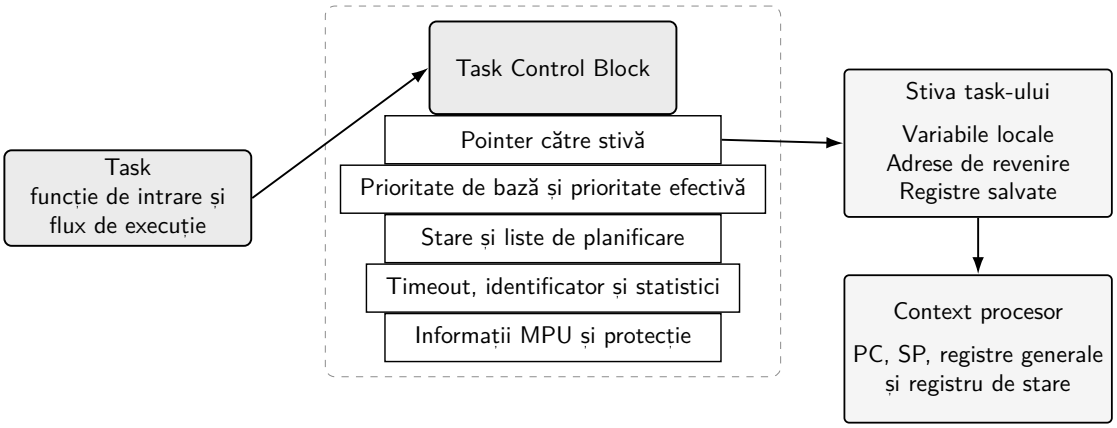


Figura 7.9 Relația dintre task, Task Control Block și stiva proprie.

cu un model cunoscut, măsurarea valorii *high-water mark*, regiuni de protecție MPU, canary values și verificarea limitelor la comutarea contextului. Monitorizarea stivei trebuie realizată în scenarii reprezentative, inclusiv în cazurile rare care produc cea mai mare adâncime de apel.

7.5.2 Stările task-ului și tranzițiile dintre acestea

Un task nu rulează permanent — pe durata existenței sale, acesta trece prin mai multe stări. Cele mai utilizate sunt Running, Ready, Blocked și Suspended.

În starea **Running**, task-ul execută instrucțiuni pe procesor; pe un sistem cu un singur nucleu, cel mult un task poate fi Running la un moment dat. În starea **Ready**, task-ul este pregătit să ruleze, dar procesorul este ocupat de un alt task selectat de scheduler. În starea **Blocked**, task-ul așteaptă un eveniment sau expirarea unui interval, de exemplu sosirea unui mesaj, eliberarea unui semafor, date de la un periferic, expirarea unui timer sau disponibilitatea unei resurse. În starea **Suspended**, task-ul a fost exclus explicit din planificare — el nu devine Ready numai prin expirarea unui timeout obișnuit, ci trebuie reluat printr-o operație specifică.

Figura 7.10 prezintă principalele tranziții.

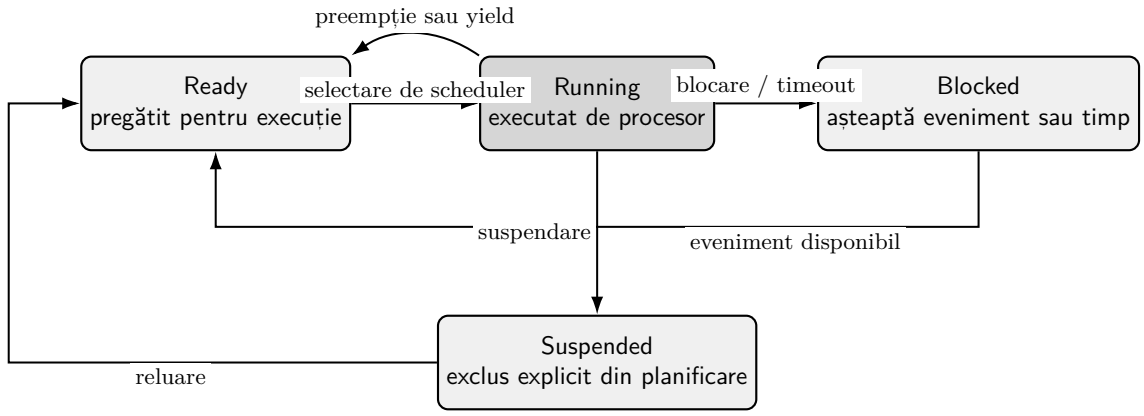
Un task trece din Running în Blocked atunci când solicită o operație care nu poate fi finalizată imediat:

$$\text{Running} \rightarrow \text{Blocked}.$$

La apariția evenimentului:

$$\text{Blocked} \rightarrow \text{Ready}.$$

Task-ul nu trece direct obligatoriu în Running — el este introdus în lista task-urilor



Un task deblocat devine Ready; execuția sa depinde de prioritate și de politica scheduler-ului.

Figura 7.10 Stările unui task și tranzițiile controlate de kernel.

Ready și va fi executat conform priorității.

Dacă task-ul deblocat are o prioritate mai mare decât task-ul curent:

$$p_{\text{deblocat}} > p_{\text{curent}},$$

scheduler-ul preemptiv poate realiza tranziția:

$$\text{Running}_{\text{curent}} \rightarrow \text{Ready},$$

iar task-ul deblocat devine Running.

Un task care așteaptă un eveniment trebuie să utilizeze și un timeout atunci când blocarea nelimitată nu este acceptabilă:

Exemplu 7.8 Așteptarea unui eveniment cu timeout

```
1  const WaitResult result =
2      wait_for_message(message_timeout);
3
4  if (result == WaitResult::MessageReceived)
5  {
6      process_message();
7  }
8  else
9  {
10     handle_communication_timeout();
11 }
```

Timeout-ul permite detectarea pierderii unui mesaj sau defectării unei componente. Un

task nu trebuie să rămână Ready în timp ce așteaptă activ un eveniment:

Exemplu 7.9 Așteptare activă nerecomandată într-un task

```
1 while (!data_available())
2 {
3     /* Task-ul ramane Ready sau Running. */
4 }
```

Această implementare consumă procesor și poate împiedica executarea task-urilor cu prioritate egală sau mai mică.

7.5.3 Comutarea contextului și costul acesteia

Comutarea contextului permite suspendarea unui task și reluarea lui ulterioară de la aceeași instrucțiune. Contextul procesorului poate include registre generale, program counter, stack pointer, registrul de stare, registre floating-point și registre specifice arhitecturii.

Secvența conceptuală este:

1. apare un eveniment de planificare;
2. contextul task-ului curent este salvat pe stiva sa;
3. stack pointer-ul este salvat în TCB;
4. scheduler-ul selectează următorul task;
5. stack pointer-ul noului task este restaurat;
6. registrele acestuia sunt încărcate;
7. execuția este reluată.

Figura 7.11 prezintă această secvență.

Comutarea contextului introduce timp în care nu este executată funcționalitate utilă a aplicației.

Dacă durata unei comutări este C_{switch}, iar numărul de comutări în intervalul T este N_{switch}, utilizarea asociată este:

$$U_{\text{switch}} = \frac{N_{\text{switch}} C_{\text{switch}}}{T}. \tag{7.34}$$

Exemplu

Un sistem realizează:

$$N_{\text{switch}} = 5000$$

comutări pe secundă. Fiecare comutare necesită:

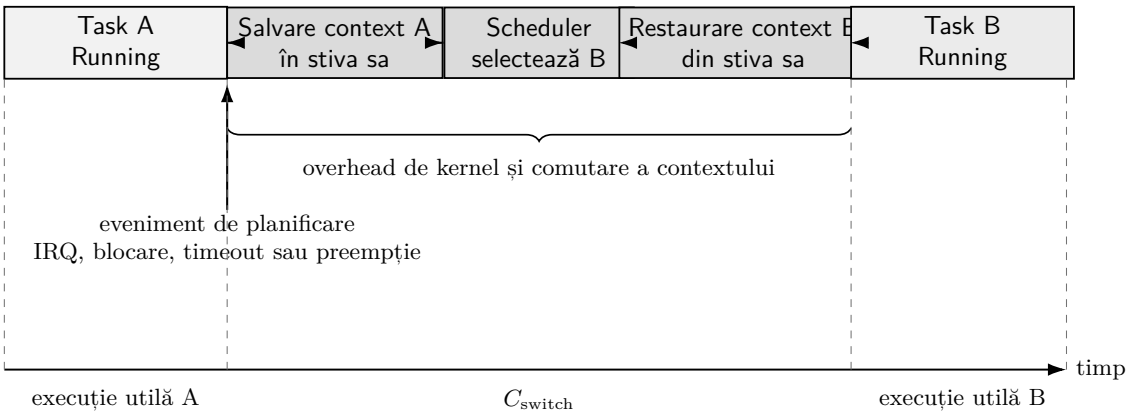


Figura 7.11 Secvența conceptuală a unei comutări de context.

$$C_{\text{switch}} = 5\,\mu\text{s}.$$

Timpul total consumat este:

$$T_{\text{switch}} = 5000 \cdot 5\,\mu\text{s} = 25\,\text{ms}.$$

Utilizarea procesorului pentru comutarea contextului este:

$$U_{\text{switch}} = \frac{25\,\text{ms}}{1000\,\text{ms}} = 2{,}5\%.$$

La această valoare se adaugă timpul executat în scheduler, întreruperi și alte servicii ale kernelului.

Costul comutării depinde de:

- arhitectura procesorului;
- numărul registrelor;
- utilizarea unității floating-point;
- tipul memoriei;
- mecanismele de protecție;
- starea cache-ului;
- implementarea kernelului.

Pe unele procesoare, o parte a contextului este salvată automat la intrarea într-o excepție. Restul este gestionat software de portarea RTOS-ului.

Numărul schimbărilor de context poate fi redus prin:

- evitarea time slice-urilor excesiv de mici;
- gruparea activităților;
- reducerea semnalizărilor inutile;

- utilizarea eficientă a bufferelor;
- alegerea corectă a priorităților;
- evitarea task-urilor fragmentate excesiv.

Reducerea numărului task-urilor nu trebuie realizată în detrimentul clarității arhitecturii sau al izolării funcționalităților critice.

7.6 Multitasking și organizarea aplicației

Multitasking-ul permite existența mai multor fluxuri de execuție în aceeași aplicație. Pe un procesor cu un singur nucleu, task-urile nu rulează fizic simultan, ci împart timpul procesorului, iar impresia de execuție simultană este produsă prin preempție, blocarea task-urilor, time slicing și comutarea rapidă a contextului. Pe un procesor multicore, mai multe task-uri pot rula în paralel, ceea ce introduce probleme suplimentare de sincronizare și coerență a memoriei.

7.6.1 Concurență, paralelism și preempție

Concurența descrie situația în care mai multe activități progresează în intervale suprapuse, iar paralelismul descrie execuția efectiv simultană pe unități de calcul diferite. Un sistem uniprocesor poate fi concurent fără a fi paralel:

$$\tau_1 \rightarrow \tau_2 \rightarrow \tau_1 \rightarrow \tau_3.$$

Figura 7.12 prezintă distribuirea procesorului între mai multe task-uri.

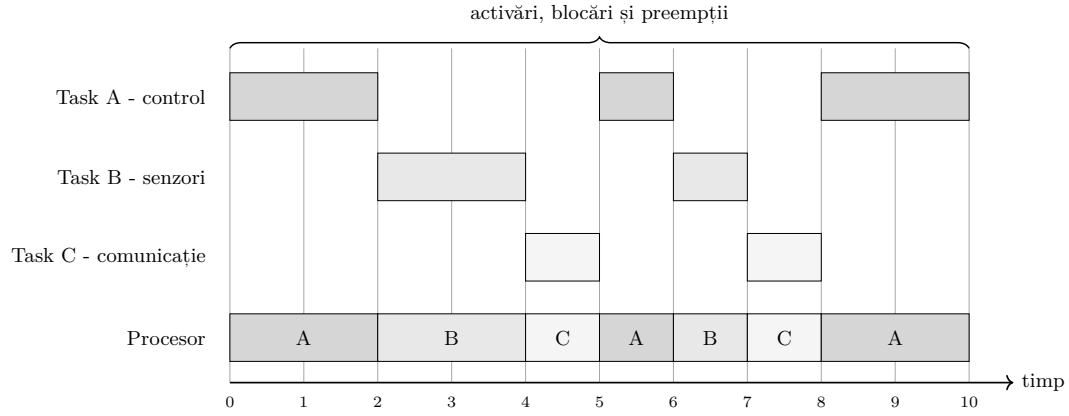


Figura 7.12 Execuția concurentă a task-urilor pe un procesor uniprocesor.

Într-un sistem cooperativ, task-ul curent păstrează procesorul până când se blochează, se termină sau cedează voluntar execuția. Într-un sistem preemptiv, kernelul poate întrerupe task-ul atunci când un task cu prioritate mai mare devine Ready. Preempția îmbunătățește timpul de răspuns pentru activitățile critice, dar introduce comutări de context, posibilitatea întreruperii în aproape orice punct, necesitatea protejării datelor partajate și dificultăți suplimentare de depanare. Codul trebuie proiectat presupunând că un task poate fi preemptat

între două instrucțiuni, cu excepția regiunilor protejate explicit.

Într-un sistem cu priorități, task-ul de prioritate mare nu trebuie să execute permanent — el trebuie să se blocheze atunci când nu are activitate utilă, permițând task-urilor cu prioritate mai mică să ruleze.

7.6.2 Task-uri periodice, sporadice și orientate pe evenimente

Un task periodic este activat la intervale regulate — exemplele includ bucle de control, achiziții și actualizări ale actuatoarelor.

O implementare conceptuală este:

Exemplu 7.10 Task periodic de control

```
void MotorControlTask(void* argument)
{
    TickType next_release = get_current_tick();

    while (true)
    {
        acquire_feedback();
        compute_motor_command();
        apply_motor_command();

        delay_until(
            next_release,
            motor_control_period);
    }
}
```

Task-ul sporadic este activat de evenimente între care există o separare minimă, precum un mesaj de comunicație, detectarea unei erori, o comandă externă sau un semnal de la un senzor. Un task orientat pe evenimente poate aștepta blocat pe o coadă:

Exemplu 7.11 Task activat printr-un mesaj

```
void CommunicationTask(void* argument)
{
    Message message{};

    while (true)
    {
        if (receive_message(
                communication_queue,
                message,
                receive_timeout))
```

```
11          {
12              process_message(message);
13          }
14          else
15          {
16              handle_receive_timeout();
17          }
18      }
19  }
```

Modelul orientat pe evenimente este eficient deoarece task-ul nu consumă procesor în absența mesajelor. Un task poate combina activări periodice și evenimente — de exemplu, task-ul de diagnostic poate fi activat periodic, pentru verificări regulate, și imediat, la apariția unei erori. Această combinație trebuie proiectată astfel încât evenimentele frecvente să nu împiedice activitatea periodică.

Task-urile care execută operații de intrare-ieșire trebuie să utilizeze mecanisme blocante eficiente, DMA sau întreruperi. Așteptarea activă menține procesorul ocupat și afectează planificarea.

7.6.3 Granularitatea task-urilor și separarea responsabilităților

Una dintre deciziile importante este alegerea numărului și dimensiunii task-urilor. O arhitectură cu un număr foarte mic de task-uri poate conduce la funcții mari și greu de întreținut, amestecarea responsabilităților, blocarea unor activități critice de către operații lente și dificultăți la stabilirea priorităților. O arhitectură cu prea multe task-uri poate produce consum mare de RAM pentru stive, multe comutări de context, sincronizare complexă, fragmentarea fluxului funcțional și dificultăți de analiză.

Un task trebuie să reprezinte o activitate cu condiție clară de activare, cerință temporală coerentă, prioritate justificată, resurse bine definite și interfață explicită cu celelalte componente. Nu este recomandată crearea unui task pentru fiecare funcție software — funcțiile cu aceeași periodicitate, criticitate și context pot fi grupate. În schimb, două funcționalități trebuie separate atunci când au priorități foarte diferite, una poate fi blocată pentru perioade lungi, au deadline-uri incompatibile, necesită izolare sau sunt dezvoltate ori validate independent.

Exemplu

Exemplu: robot mobil.
Un robot conține următoarele activități:

- controlul motoarelor la fiecare 5 ms;
- citirea senzorilor la fiecare 10 ms;
- localizarea la fiecare 20 ms;
- comunicația atunci când sosesc mesaje;

- logarea la fiecare 500 ms.

Controlul motoarelor trebuie separat de logare deoarece are un deadline mult mai strict. Comunicarea poate fi implementată ca task orientat pe evenimente, iar logarea ca task periodic cu prioritate redusă. Localizarea și achiziția pot fi separate dacă timpul de procesare al localizării este variabil și nu trebuie să întârzie citirea senzorilor.

Interfața dintre task-uri trebuie să evite dependențele ascunse. Datele pot fi transferate prin:

- cozi de mesaje;
- buffere protejate;
- notificări;
- evenimente;
- mecanisme publish–subscribe.

Accesul direct al mai multor task-uri la variabile globale produce dependențe dificil de analizat și va fi tratat în secțiunile dedicate sincronizării.

O arhitectură bine proiectată trebuie să permită identificarea traseului end-to-end:

$$\text{senzor} \rightarrow \text{achiziție} \rightarrow \text{procesare} \rightarrow \text{control} \rightarrow \text{actuator}.$$

Separarea în task-uri nu trebuie să ascundă bugetul temporal total al acestui lanț.

7.7 Multitasking și strategii de planificare

Într-un sistem multitasking, mai multe task-uri pot fi simultan pregătite pentru execuție, însă numărul procesoarelor disponibile este limitat. Pe o platformă cu un singur nucleu, la un moment dat poate rula un singur task, iar kernelul trebuie să decidă ordinea în care task-urile primesc procesorul.

Procesul prin care este ales următorul task se numește *planificare* sau *scheduling*. Componenta kernelului care realizează această alegere este scheduler-ul.

Decizia scheduler-ului influențează direct:

- timpul de răspuns;
- respectarea deadline-urilor;
- predictibilitatea temporală;
- gradul de utilizare a procesorului;
- numărul comutărilor de context;
- consumul energetic;
- echitatea între task-uri.

Nu există o strategie universală care să fie optimă pentru toate sistemele. Aplicațiile interactive pot urmări echitatea și timpul mediu de răspuns, în timp ce sistemele hard real-time urmăresc garantarea deadline-urilor activităților critice.

Un scheduler poate utiliza priorități fixe, priorități dinamice, intervale de timp sau reguli cooperative. Alegerea trebuie corelată cu modelul task-urilor și cu cerințele sistemului.

7.7.1 Planificarea cooperativă și planificarea preemptivă

Într-un sistem cooperativ, task-ul aflat în execuție păstrează procesorul până când:

- finalizează activitatea;
- se blochează în așteptarea unui eveniment;
- cedează voluntar procesorul.

Un task cooperativ poate ceda execuția printr-o operație de tip `yield`:

Exemplu 7.12 Task cooperativ care cedează voluntar procesorul

```
1 void DiagnosticTask(void* argument)
2 {
3     while (true)
4     {
5         execute_diagnostic_step();
6         task_yield();
7     }
8 }
```

Avantajele planificării cooperative sunt:

- implementarea simplă;
- numărul redus de comutări de context;
- controlul explicit al punctelor de întrerupere;
- protejarea mai simplă a unor date partajate;
- comportamentul relativ ușor de urmărit.

Principala limitare este dependența întregului sistem de comportamentul task-ului curent — dacă acesta execută o buclă lungă sau nu cedează procesorul, celelalte task-uri nu pot rula.

Timpul maxim în care un task de prioritate mare poate aștepta într-un sistem cooperativ depinde de cea mai lungă secțiune nepreemptivă:

$$B_{\text{coop}} = \max_{j} \left(C_{j,\text{nepreemptiv}} \right). \tag{7.35}$$

Dacă un task cu prioritate redusă execută neîntrerupt timp de 20 ms, un eveniment critic apărut imediat după începerea sa poate aștepta aproape întreaga durată. Într-un sistem preemptiv, scheduler-ul poate întrerupe task-ul curent atunci când un task cu prioritate mai mare devine Ready.

Secvența este:

Task cu prioritate redus → eveniment → Task cu prioritate ridicat.

Figura 7.13 compară cele două strategii.

Preempția permite reducerea timpului de răspuns al activităților critice, dar introduce costuri suplimentare: salvarea și restaurarea contextului, acces concurent la date, necesita-

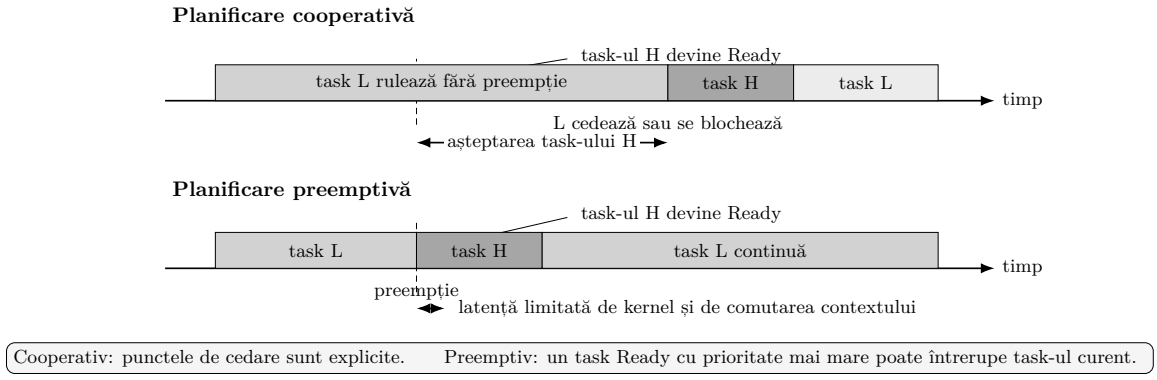


Figura 7.13 Compararea planificării cooperative și preemptive.

tea secțiunilor critice, posibilitatea apariției unor condiții de cursă și depanare mai dificilă. Într-un sistem preemptiv, codul unui task poate fi întrerupt între aproape oricare două instrucțiuni — accesul la o structură partajată nu trebuie considerat sigur doar pentru că secvența de cod este scurtă.

O arhitectură poate combina cele două modele: task-urile cu priorități diferite pot fi planificate preemptiv, iar task-urile cu aceeași prioritate pot coopera sau pot utiliza Round Robin.

7.7.2 Planificarea pe priorități și Round Robin

În majoritatea RTOS-urilor pentru microcontrolere, fiecare task are o prioritate numerică. Scheduler-ul selectează task-ul Ready cu prioritatea efectivă cea mai mare:

$$\tau^{\star} = \arg\max_{\tau_i \in \mathcal{R}} (p_i)\,, \tag{7.36}$$

unde $\mathcal{R}$ este mulțimea task-urilor Ready. Prioritatea trebuie asociată cerinței temporale, nu importanței generale a funcției pentru utilizator. Un task de logare poate manipula informații importante, dar poate avea un deadline relaxat; un task de control aparent simplu poate necesita prioritate mare deoarece trebuie să reacționeze în câteva sute de microsecunde.

Considerăm următoarele task-uri:

Tabela 7.3 Exemplu de priorități într-un sistem de control

Task	Prioritate	Justificare
MotorControlTask	5	Deadline redus și impact direct asupra stabilității
SensorTask	4	Furnizează datele necesare controlului
CommunicationTask	2	Acceptă o latență mai mare
LoggerTask	1	Activitate necritică din punct de vedere temporal

Un task de prioritate mare trebuie să se blocheze atunci când nu are activitate — în caz contrar, task-urile de prioritate mai mică nu vor primi procesorul.

O implementare problematică este:

Exemplu 7.13 Task de prioritate mare care monopolizează procesorul

```
1  void HighPriorityTask(void* argument)
2  {
3      while (true)
4      {
5          if (event_available())
6          {
7              process_event();
8          }
9
10         /* Task-ul nu se blocheaza. */
11     }
12 }
```

O variantă corespunzătoare așteaptă evenimentul printr-o primitivă blocantă:

Exemplu 7.14 Task de prioritate mare blocat în așteptarea unui eveniment

```
1  void HighPriorityTask(void* argument)
2  {
3      while (true)
4      {
5          wait_for_event();
6          process_event();
7      }
8  }
```

Atunci când mai multe task-uri Ready au aceeași prioritate, poate fi utilizată planificarea Round Robin. Fiecărui task i se acordă un interval denumit *time slice* sau *quantum*:

$$Q = N_Q T_{\text{tick}}, \tag{7.37}$$

unde N_Q este numărul de tick-uri alocat.

La expirarea intervalului, task-ul curent este mutat la sfârșitul listei corespunzătoare priorității sale.

Figura 7.14 prezintă planificarea mai multor task-uri.

Un quantum foarte scurt oferă o distribuție aparent uniformă a procesorului, dar crește numărul comutărilor de context; un quantum foarte lung reduce overhead-ul, însă poate crește timpul de așteptare al task-urilor de aceeași prioritate.

Într-un sistem hard real-time, utilizarea mai multor task-uri cu aceeași prioritate trebuie analizată atent, deoarece ordinea Round Robin poate introduce interferență dificil de observat dacă analiza consideră numai prioritățile numerice.

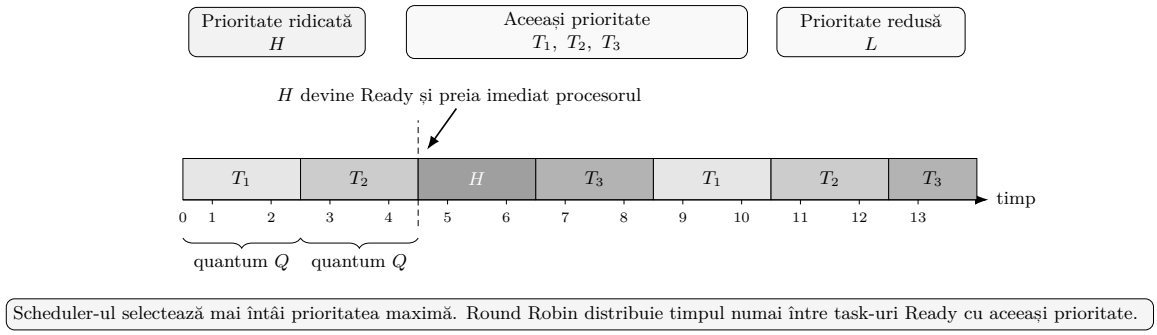


Figura 7.14 Planificare pe priorități și Round Robin între task-uri egale.

7.7.3 Overhead-ul planificării și criteriile de proiectare

Scheduler-ul, tick-ul periodic, ISR-urile kernelului și comutările de context consumă timp de procesor, iar acest timp trebuie inclus în analiza sistemului.

Utilizarea totală poate fi aproximată prin:

$$U_{\text{total}} = U_{\text{task}} + U_{\text{ISR}} + U_{\text{kernel}}, \tag{7.38}$$

unde:

$$U_{\text{task}} = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{7.39}$$

Costul comutărilor de context într-un interval T este:

$$U_{\text{switch}} = \frac{N_{\text{switch}} C_{\text{switch}}}{T}. \tag{7.40}$$

Costul tick-ului este aproximativ:

$$U_{\text{tick}} = f_{\text{tick}} C_{\text{tick}}. \tag{7.41}$$

Pentru:

$$f_{\text{tick}} = 1000\,\text{Hz}, \qquad C_{\text{tick}} = 2\,\mu\text{s},$$

rezultă:

$$U_{\text{tick}} = 1000 \cdot 2\,\mu\text{s} = 0{,}002,$$

adică:

$$U_{\text{tick}} = 0{,}2\%.$$

La această valoare se adaugă comutările de context și funcțiile apelate în interiorul tick-ului.

Figura 7.15 prezintă componentele overhead-ului de planificare.

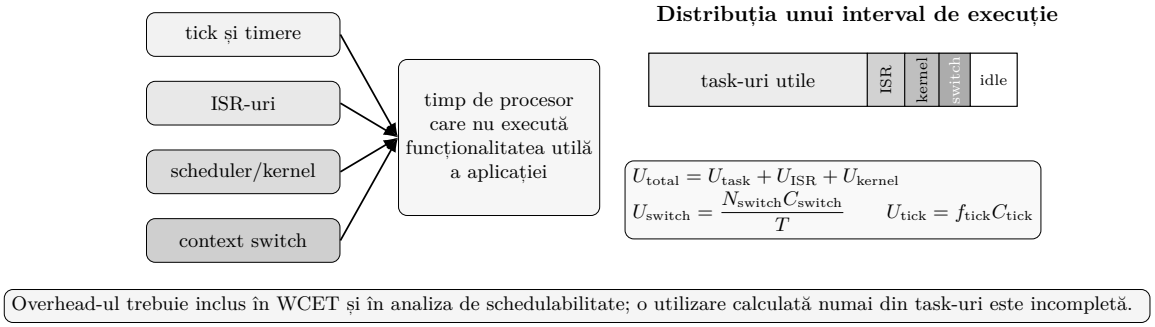


Figura 7.15 Componentele costului temporal introdus de planificare.

O utilizare totală apropiată de 100% lasă o marjă redusă pentru variația timpilor de execuție, întreruperi rare, tratarea erorilor, comunicații neperiodice, operații de mentenanță și extinderea ulterioară a aplicației. Totuși, alegerea unei limite fixe precum 70% sau 80% nu demonstrează automat schedulabilitatea — limita adecvată depinde de algoritmul de planificare, de deadline-uri, de blocări și de caracteristicile task-urilor.

Criteriile de alegere a strategiei includ existența task-urilor hard real-time, necesitatea preempției, numărul nivelurilor de prioritate, costul context switching-ului, frecvența evenimentelor, gradul de partajare a resurselor și posibilitatea realizării unei analize formale.

7.8 Analiza schedulabilității și algoritmi clasici de planificare

Un set de task-uri este *schedulabil* dacă toate instanțele sale pot respecta deadline-urile sub politica de planificare analizată. Schedulabilitatea nu este o proprietate exclusivă a gradului de utilizare — aceasta depinde și de perioade, deadline-uri, priorități, momente de activare, timpi de blocare, preempții, overhead-ul kernelului, relațiile de precedență și resursele partajate.

Un test de schedulabilitate poate fi suficient, dar nu necesar, necesar, dar nu suficient, sau exact pentru modelul analizat. Dacă un set nu satisface un test suficient, nu rezultă automat că este neschedulabil — poate fi necesar un test mai precis.

7.8.1 Modelul task-urilor și gradul de utilizare

Considerăm un set de n task-uri periodice:

$$\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}\,. \tag{7.42}$$

Fiecare task este descris prin:

$$\tau_i = (C_i, T_i, D_i)\,, \tag{7.43}$$

unde C_i este WCET-ul, T_i este perioada, iar D_i este deadline-ul relativ. Utilizarea individuală este:

$$U_i = \frac{C_i}{T_i}, \tag{7.44}$$

iar utilizarea totală este:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} \frac{C_i}{T_i}. \tag{7.45}$$

Condiția:

$$U \leq 1 \tag{7.46}$$

este necesară pentru un set de task-uri periodice independente pe un singur procesor, dacă nu sunt incluse alte costuri, dar nu este suficientă pentru orice algoritm de planificare.

Exemplu

Se consideră task-urile:

Task	C_i	T_i
τ_1	1 ms	5 ms
τ_2	2 ms	10 ms
τ_3	5 ms	50 ms

Utilizarea este:

$$U = \frac{1}{5} + \frac{2}{10} + \frac{5}{50} = 0{,}5.$$

Procesorul este utilizat în proporție de 50% de task-urile modelate. Această valoare nu demonstrează singură respectarea deadline-urilor — trebuie cunoscute politica de planificare, deadline-urile și timpii de blocare.

Pentru task-uri periodice, intervalul după care modelul activărilor se repetă este hiperperioada:

$$H = \operatorname{lcm}(T_1, T_2, \ldots, T_n). \tag{7.47}$$

Construirea unei diagrame complete pe hiperperioadă poate fi utilă pentru exemple mici, dar devine dificilă atunci când perioadele sunt numeroase sau nu sunt armonice.

7.8.2 Rate Monotonic Scheduling și limita Liu–Layland

Rate Monotonic Scheduling, prescurtat RMS, este o strategie cu priorități fixe pentru task-uri periodice. Prioritatea este invers proporțională cu perioada:

$$T_i < T_j \quad \Longrightarrow \quad p_i > p_j. \tag{7.48}$$

Task-ul cu perioada cea mai mică primește prioritatea cea mai mare.

În modelul clasic sunt presupuse:

- task-uri periodice;
- procesor unic;
- planificare preemptivă;
- task-uri independente;
- deadline-uri egale cu perioadele;
- timpi de execuție cunoscuți;
- costuri de kernel neglijabile;
- absența blocării pe resurse.

În aceste condiții, RMS este optim între algoritmii cu priorități fixe: dacă un set poate fi planificat printr-o anumită atribuire de priorități fixe, poate fi planificat și prin ordonarea Rate Monotonic [53].

Pentru n task-uri, Liu și Layland au formulat condiția suficientă:

$$U \leq n\left(2^{\frac{1}{n}} - 1\right). \tag{7.49}$$

Limita este:

$$U_{\mathrm{LL}}(n) = n\left(2^{\frac{1}{n}} - 1\right). \tag{7.50}$$

Pentru un număr mare de task-uri:

$$\lim_{n\to\infty} U_{\mathrm{LL}}(n) = \ln 2 \approx 0{,}693. \tag{7.51}$$

Tabela 7.4 Limita suficientă Liu–Layland pentru RMS

Număr de task-uri	U_{LL}
1	100,0%
2	82,8%
3	78,0%
4	75,7%
5	74,3%
∞	69,3%

Limita este suficientă, nu necesară. Dacă:

$$U \leq U_{\mathrm{LL}},$$

setul este garantat schedulabil în condițiile modelului. Dacă:

$$U > U_{\mathrm{LL}},$$

testul este neconcludent, setul putând fi în continuare schedulabil.

Figura 7.16 prezintă un exemplu de planificare RMS.

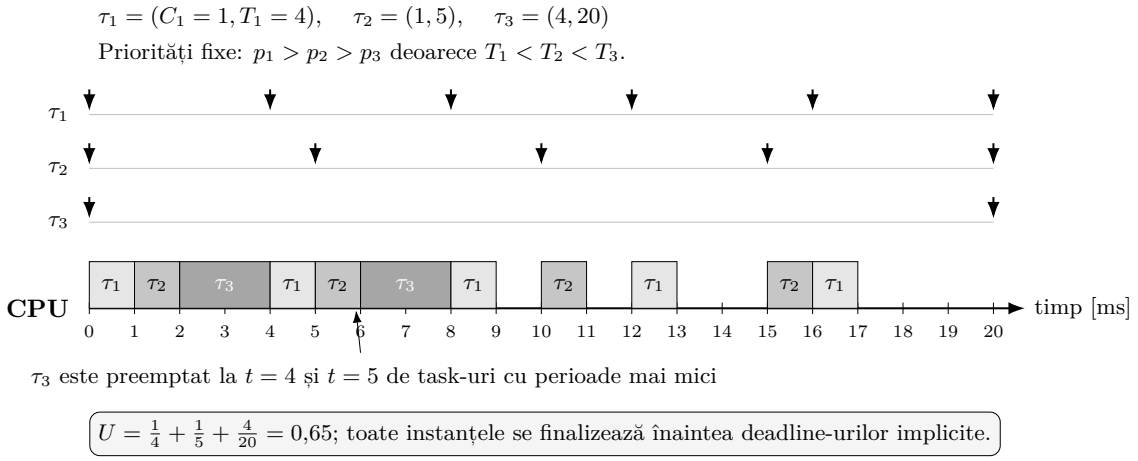


Figura 7.16 Exemplu de planificare Rate Monotonic Scheduling.

Exemplu

Exemplu RMS.

Considerăm:

$$\tau_1 = (1, 5, 5),$$

$$\tau_2 = (2, 10, 10),$$

și:

$$\tau_3 = (4, 40, 40).$$

Utilizarea este:

$$U = \frac{1}{5} + \frac{2}{10} + \frac{4}{40} = 0{,}5.$$

Pentru trei task-uri:

$$U_{\text{LL}} = 3\left(2^{1/3} - 1\right) \approx 0{,}780.$$

Deoarece:

$$0{,}5 < 0{,}780,$$

setul este garantat schedulabil prin RMS în ipotezele modelului.

Pentru perioade armonice, în care fiecare perioadă este un multiplu întreg al perioadelor

mai mici, RMS poate utiliza procesorul până aproape de 100%. Prin urmare, valoarea de 69,3% nu reprezintă o limită maximă universală pentru RMS.

7.8.3 Earliest Deadline First

Earliest Deadline First, prescurtat EDF, utilizează priorități dinamice, iar instanța cu deadline-ul absolut cel mai apropiat primește prioritatea cea mai mare:

$$J^\star = \arg\min_{J_{i,k}\in\mathcal{R}} (d_{i,k})\,. \tag{7.52}$$

Prioritatea unui task se poate modifica de la o instanță la alta, deoarece deadline-urile absolute evoluează în timp. În modelul clasic — cu task-uri periodice independente, procesor unic, preempție, deadline-uri egale cu perioadele și costuri de kernel neglijabile — setul este schedulabil prin EDF dacă și numai dacă:

$$U \leq 1. \tag{7.53}$$

EDF poate utiliza teoretic 100% din capacitatea procesorului pentru acest model.

Figura 7.17 prezintă un exemplu de priorități dinamice.

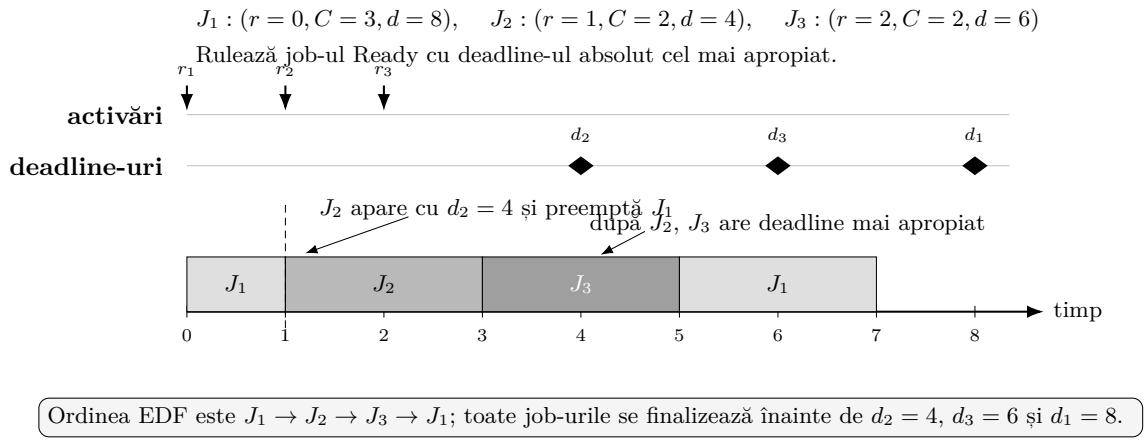


Figura 7.17 Exemplu de planificare Earliest Deadline First.

Avantajele EDF includ utilizarea eficientă a procesorului, adaptarea priorităților la deadline-urile curente și optimalitatea pentru modelul preemptiv uniprocesor.

Limitările practice includ:

- administrarea unei cozi ordonate după deadline;
- priorități dinamice;
- comportament mai dificil de urmărit;
- degradare mai puțin intuitivă la suprasarcină;
- integrarea mai complexă cu unele mecanisme de moștenire a priorității.

Într-o suprasarcină, EDF poate produce depășiri pentru mai multe task-uri. Într-un sistem cu priorități fixe, task-urile cu prioritate mare pot rămâne protejate, în timp ce

activitățile de prioritate mică ratează deadline-urile.

Condiția simplă $U \leq 1$ nu este suficientă pentru toate modelele EDF. Dacă deadline-urile sunt mai mici decât perioadele, este necesară analiza cererii de procesor într-un interval.

7.8.4 Analiza timpului de răspuns și compararea RMS–EDF

Limita Liu–Layland este simplă, dar poate fi conservatoare. Pentru planificarea preemptivă cu priorități fixe poate fi utilizată analiza timpului de răspuns.

Pentru task-ul τ_i, timpul de răspuns în cazul cel mai nefavorabil poate fi calculat iterativ:

$$R_i^{(k+1)} = C_i + B_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j, \tag{7.54}$$

unde B_i este timpul maxim de blocare, $hp(i)$ este mulțimea task-urilor cu prioritate mai mare, iar termenul din sumă reprezintă interferența task-urilor superioare. Inițializarea poate fi:

$$R_i^{(0)} = C_i + B_i. \tag{7.55}$$

Iterația continuă până când:

$$R_i^{(k+1)} = R_i^{(k)}, \tag{7.56}$$

sau până când:

$$R_i^{(k+1)} > D_i. \tag{7.57}$$

Task-ul este schedulabil dacă:

$$R_i \leq D_i. \tag{7.58}$$

Exemplu

Exemplu de analiză a timpului de răspuns.
Considerăm:

$$\tau_1 = (C_1 = 1,\ T_1 = 4,\ D_1 = 4),$$

și:

$$\tau_2 = (C_2 = 2,\ T_2 = 6,\ D_2 = 6).$$

Task-ul τ_1 are prioritate mai mare.
Pentru τ_1:

$$R_1 = C_1 = 1\,\text{ms}.$$

Rezultă:

$$R_1 \leq D_1.$$

Pentru τ_2:

$$R_2^{(0)} = 2.$$

Prima iterație este:

$$R_2^{(1)} = 2 + \left\lceil \frac{2}{4} \right\rceil \cdot 1 = 3.$$

A doua iterație este:

$$R_2^{(2)} = 2 + \left\lceil \frac{3}{4} \right\rceil \cdot 1 = 3.$$

Calculul converge la:

$$R_2 = 3\,\text{ms}.$$

Deoarece:

$$3\,\text{ms} \leq 6\,\text{ms},$$

task-ul respectă deadline-ul.

Figura 7.18 prezintă nivelurile unei analize de schedulabilitate.

RMS și EDF pot fi comparate astfel:

Alegerea algoritmului nu trebuie realizată numai pe baza utilizării teoretice maxime. În sistemele industriale sunt importante și:

- simplitatea certificării;
- comportamentul la suprasarcină;
- suportul oferit de RTOS;
- integrarea resurselor partajate;
- ușurința depanării;
- modificarea ulterioară a aplicației.

Analiza trebuie să includă costurile reale — timpul de execuție utilizat în formule poate fi extins:

$$C_i' = C_i + C_{\text{kernel},i} + C_{\text{switch},i} + C_{\text{cache},i}. \tag{7.59}$$

Timpii de blocare introduși de mutex-uri și secțiuni critice vor fi analizați în blocul următor.

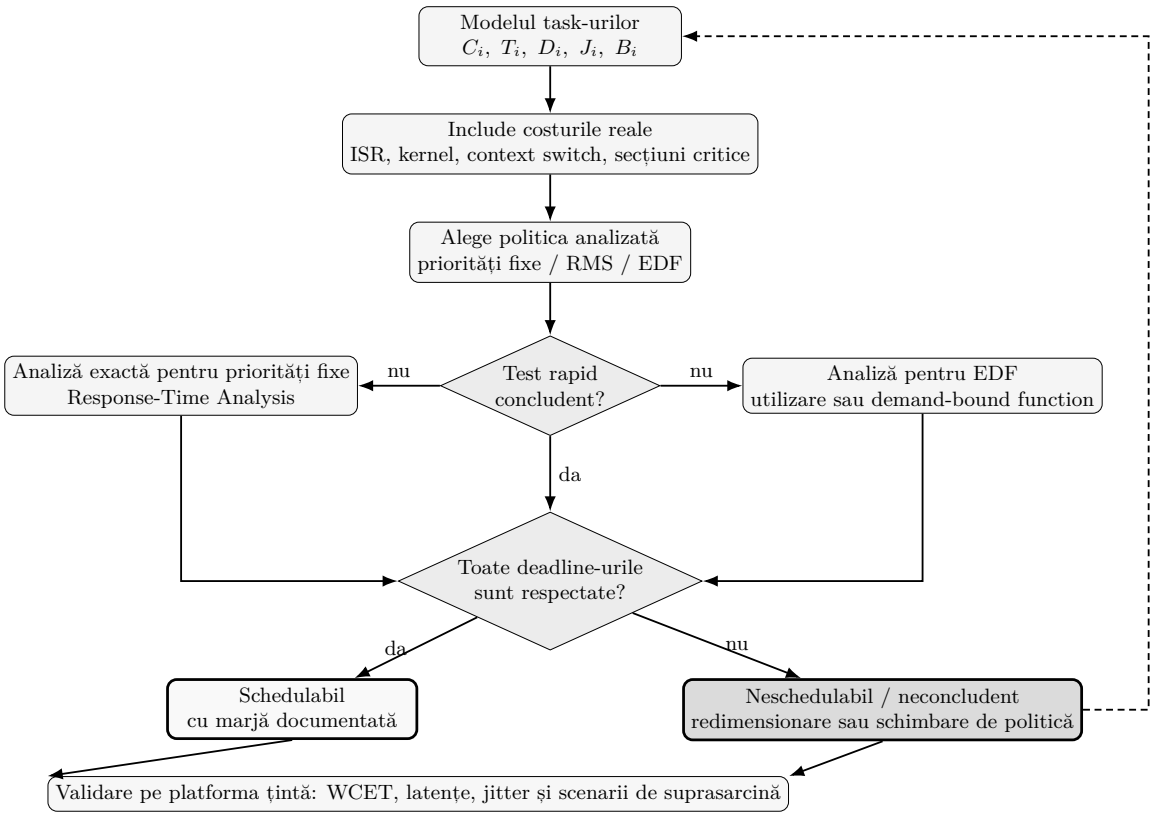


Figura 7.18 Etapele analizei de schedulabilitate a unui sistem de timp real.

Tabela 7.5 Compararea planificării RMS și EDF

Caracteristică	RMS	EDF
Tipul priorităților	Fixe	Dinamice
Criteriul priorității	Perioada task-ului	Deadline-ul absolut curent
Test simplu	Limita Liu–Layland, suficientă	$U \leq 1$ pentru deadline-uri implicite
Utilizare teoretică	Poate ajunge la 100%, dar limita suficientă este mai redusă	Până la 100% în modelul clasic
Implementare	Simplă și potrivită pentru multe RTOS-uri	Necesită administrarea deadline-urilor dinamice
Suprasarcină	Task-urile cu prioritate mare pot rămâne protejate	Pot apărea depășiri distribuite între task-uri
Analiză	Limite de utilizare și analiză a timpului de răspuns	Analiză de utilizare sau demand-bound

8. Sincronizare, Arhitecturi de Kernel și RTOS-uri pentru Sisteme Embedded

Sincronizare, comunicație și protejarea resurselor partajate
Probleme de concurență și protocoale de control al priorităților
Arhitecturi de kernel și izolare software
RTOS-uri moderne
Criterii de alegere a unui RTOS
Studii de caz
Întrebări și probleme propuse

8.1 Sincronizare, comunicație și protejarea resurselor partajate

Separarea aplicației în mai multe task-uri permite organizarea clară a funcționalităților, dar introduce necesitatea coordonării acestora. Task-urile pot utiliza aceleași date, periferice, buffere sau servicii software, iar ordinea accesărilor nu mai este determinată exclusiv de succesiunea instrucțiunilor din cod.

Într-un sistem preemptiv, un task poate fi întrerupt aproape în orice moment, iar într-un sistem multicore, două task-uri pot executa efectiv în paralel. Prin urmare, o secvență care pare indivizibilă la nivelul limbajului sursă poate fi observată într-o stare intermediară de un alt context de execuție.

Sincronizarea urmărește două obiective principale: protejarea integrității resurselor partajate și coordonarea ordinii în care sunt executate activitățile. Aceste obiective nu trebuie confundate — un mutex este utilizat în principal pentru excludere mutuală, în timp ce un semafor sau o notificare poate fi utilizat pentru semnalizarea producerii unui eveniment. Alegerea incorectă a primitivei poate conduce la condiții de cursă, pierderea evenimentelor, blocări nelimitate, inversarea priorităților, inconsistența datelor sau încălcarea deadline-urilor.

8.1.1 Condiții de cursă și secțiuni critice

O condiție de cursă, denumită *race condition*, apare atunci când rezultatul programului depinde de ordinea temporală a două sau mai multe accesări concurente.

Considerăm o variabilă globală:

Exemplu 8.1 Incrementarea neprotejată a unei variabile partajate

```
1  std::uint32_t event_count = 0;
2
3  void TaskA()
4  {
5      ++event_count;
6  }
7
8  void TaskB()
9  {
10     ++event_count;
11 }
```

Operația:

`event_count++`

nu este obligatoriu o singură instrucțiune atomică, putând fi descompusă în:

1. citirea valorii din memorie;
2. incrementarea valorii;
3. scrierea rezultatului.

Dacă două task-uri execută această secvență simultan, poate apărea următoarea ordine:

τ_A : citește 10,
τ_B : citește 10,
τ_A : scrie 11,
τ_B : scrie 11.

Deși au fost executate două incrementări, valoarea finală este 11 în loc de 12.

Figura 8.1 prezintă pierderea unei actualizări.

O regiune de cod care accesează o resursă partajată și trebuie executată în mod indivizibil se numește *secțiune critică*.

Pentru o resursă R, excluderea mutuală impune:

$$N_{\text{task-uri n secțiunea critic}}(R) \leq 1. \tag{8.1}$$

O secțiune critică trebuie să fie cât mai scurtă, lipsită de operații blocante, lipsită de întârzieri voluntare, lipsită de calcule care pot fi executate în exterior și bine delimitată.

Un exemplu nerecomandat este:

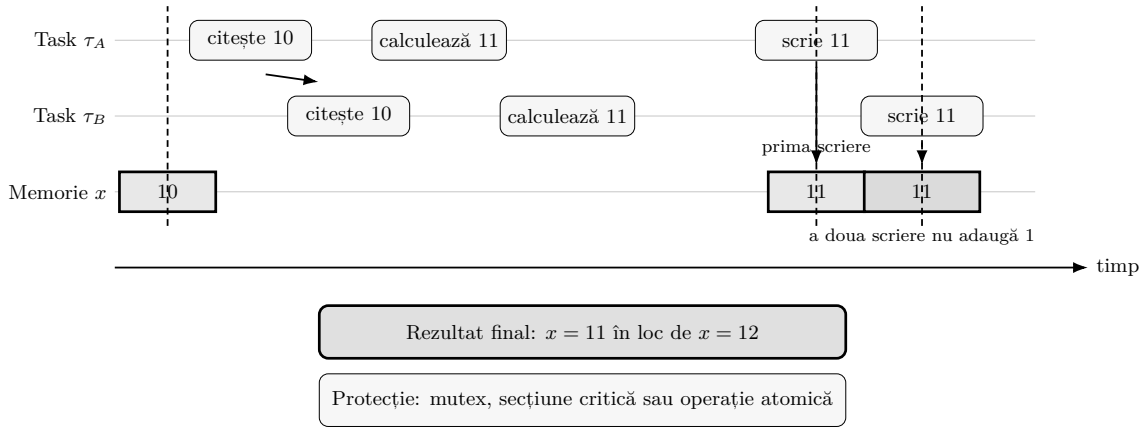


Figura 8.1 Condiție de cursă produsă de actualizarea concurentă a unei variabile.

Exemplu 8.2 Secțiune critică excesiv de lungă

```
1 lock(data_mutex);
2
3 read_shared_data();
4 perform_complex_filtering();
5 write_external_flash();
6 send_network_packet();
7
8 unlock(data_mutex);
```

Mutex-ul este păstrat în timpul filtrării, accesului Flash și comunicației, deși numai o parte dintre aceste operații necesită protejarea datelor.

O variantă mai bună copiază datele sub protecția mutex-ului și efectuează procesarea în exterior:

Exemplu 8.3 Reducerea duratei secțiunii critice

```
1 SensorData local_copy{};
2
3 lock(data_mutex);
4 local_copy = shared_sensor_data;
5 unlock(data_mutex);
6
7 perform_complex_filtering(local_copy);
8 write_external_flash(local_copy);
9 send_network_packet(local_copy);
```

Durata maximă a secțiunii critice influențează timpul de blocare al celorlalte task-uri:

$$B_i \geq C_{\text{critical,max}}, \tag{8.2}$$

dacă task-ul τ_i poate solicita resursa în timp ce aceasta este deținută de alt task. Dezactivarea întreruperilor poate fi utilizată pentru protejarea unor secvențe foarte scurte între task și ISR:

Exemplu 8.4 Secțiune critică locală prin mascarea temporară a întreruperilor

```
1 enter_critical_section();
2
3 shared_status = new_status;
4 shared_counter++;
5
6 exit_critical_section();
```

Această metodă trebuie utilizată cu precauție, deoarece durata în care întreruperile sunt dezactivate contribuie direct la latența maximă:

$$L_{\text{IRQ,max}} \geq T_{\text{interrupts disabled,max}}. \tag{8.3}$$

Dezactivarea globală a întreruperilor nu este o soluție generală pentru sincronizarea task-urilor — pe un sistem multicore, mascarea întreruperilor pe un nucleu nu împiedică alt nucleu să acceseze aceeași memorie.

Nici calificatorul `volatile` nu oferă excludere mutuală. Acesta influențează optimizările compilatorului, dar nu transformă automat o secvență de citire–modificare–scriere într-o operație atomică.

8.1.2 Mutex-uri, semafoare și grupuri de evenimente

Mutex-ul este o primitivă de sincronizare destinată protejării unei resurse partajate. Termenul provine de la *mutual exclusion*. Un mutex are conceptul de proprietar: task-ul care îl obține devine proprietar, numai proprietarul trebuie să îl elibereze, iar task-ul poate fi blocat dacă mutex-ul este deja deținut.

Utilizarea generică este:

Exemplu 8.5 Protejarea unei resurse prin mutex

```
1  void update_shared_configuration(
2      const Configuration& new_configuration)
3  {
4      if (lock_mutex(
5              configuration_mutex,
6              mutex_timeout))
7      {
8          shared_configuration = new_configuration;
9          unlock_mutex(configuration_mutex);
10     }
11     else
```

```
12      {
13          handle_mutex_timeout();
14      }
15  }
```

Timeout-ul previne blocarea nelimitată și permite detectarea unei funcționări anormale.

Un mutex nu trebuie utilizat, în general, din ISR. ISR-ul nu poate aștepta eliberarea sa, iar conceptul de proprietar este asociat unui task.

Semaforul este o primitivă mai generală. Un semafor de numărare poate fi reprezentat printr-un contor:

$$0 \leq S \leq S_{\max}. \tag{8.4}$$

Operația de eliberare incrementează contorul, iar operația de așteptare îl decrementează dacă valoarea este pozitivă. Dacă valoarea este zero, task-ul poate fi blocat.

Un semafor binar are valorile:

$$S \in \{0, 1\}.$$

El poate fi utilizat pentru semnalizarea unui eveniment:

Exemplu 8.6 Semnalizarea unui task prin semafor binar

```
1  extern "C" void ADC_IRQHandler()
2  {
3      const Sample sample = read_adc_result();
4      store_sample_from_isr(sample);
5
6      give_semaphore_from_isr(
7          adc_completed_semaphore);
8  }
9
10 void ProcessingTask(void* argument)
11 {
12     while (true)
13     {
14         take_semaphore(
15             adc_completed_semaphore,
16             wait_forever);
17
18         process_available_samples();
19     }
20 }
```

Spre deosebire de mutex, semaforul binar nu presupune obligatoriu că același context îl obține și îl eliberează — acesta poate fi eliberat de ISR și consumat de un task. Un semafor de numărare poate reprezenta numărul elementelor disponibile într-un buffer, numărul resurselor identice libere sau numărul evenimentelor produse și neprocesate.

Pentru un pool de N buffere, contorul este inițializat cu:

$$S_0 = N. \tag{8.5}$$

Fiecare utilizator decrementează contorul la obținerea unui buffer și îl incrementează după eliberare. Grupurile de evenimente permit unui task să aștepte una sau mai multe condiții reprezentate prin biți:

$$E = e_0 \vee e_1 \vee \ldots \vee e_{m-1}. \tag{8.6}$$

De exemplu, bitul 0 poate indica finalizarea achiziției, bitul 1 stabilirea comunicației, iar bitul 2 disponibilitatea configurației. Un task poate aștepta oricare dintre evenimente, toate evenimentele sau un eveniment cu timeout.

Figura 8.2 compară rolurile principalelor primitive.

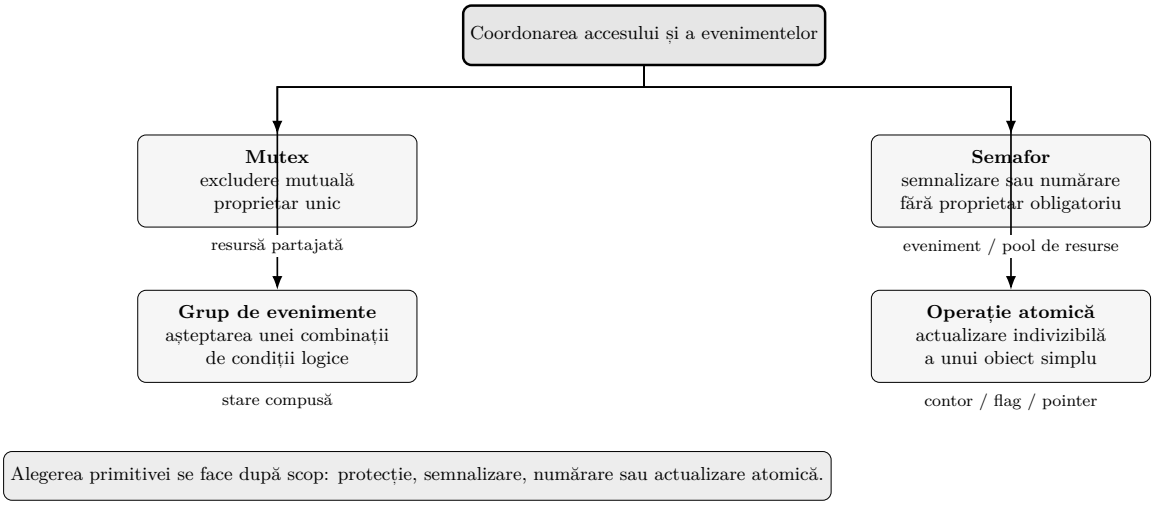


Figura 8.2 Rolurile mutex-urilor, semafoarelor și grupurilor de evenimente.

Un grup de evenimente poate pierde informația despre numărul aparițiilor — dacă același bit este setat de zece ori înainte de a fi procesat, starea sa rămâne doar setată. Pentru numărarea aparițiilor trebuie utilizat un semafor de numărare sau o coadă.

8.1.3 Cozi de mesaje și proprietatea datelor

Comunicarea prin mesaje reduce accesul direct la variabile globale — un task producător introduce date într-o coadă, iar un task consumator le extrage.

Fluxul este:

Tabela 8.1 Utilizarea principalelor primitive de sincronizare

Primitivă	Utilizare recomandată	Observație importantă
Mutex	Protejarea unei resurse cu proprietar unic	Poate implementa Priority Inheritance
Semafor binar	Semnalizarea unui eveniment	Nu are obligatoriu conceptul de proprietar
Semafor de numărare	Numărarea evenimentelor sau resurselor	Contorul are o valoare maximă
Grup de evenimente	Așteptarea unei combinații de condiții	Biții nu numără aparițiile repetate
Secțiune critică	Protejarea unei secvențe foarte scurte	Poate crește latența întreruperilor
Operație atomică	Actualizarea unui obiect simplu	Trebuie verificat suportul arhitecturii

$$\text{productor} \rightarrow \text{coad} \rightarrow \text{consumator}.$$

Figura 8.3 prezintă comunicarea dintre task-uri.

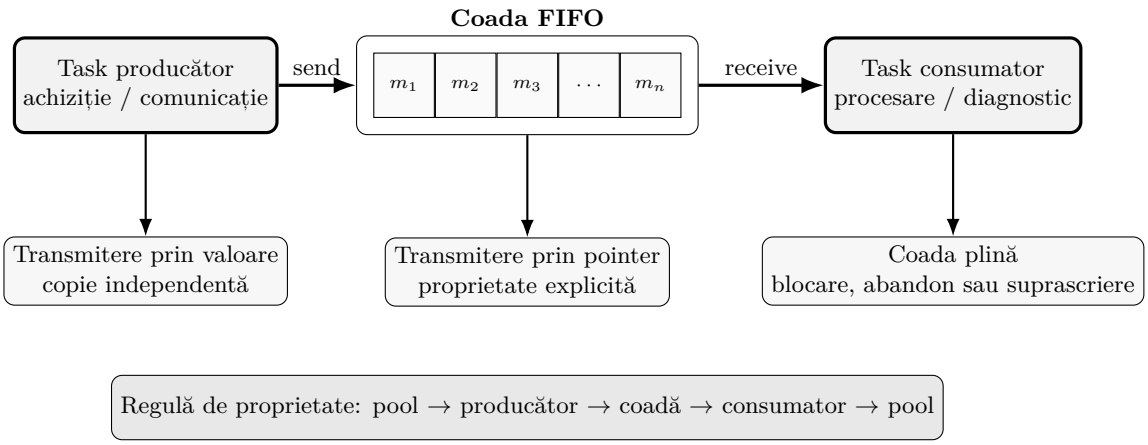


Figura 8.3 Transferul datelor între task-uri printr-o coadă de mesaje.

Un mesaj poate conține valoarea completă:

Exemplu 8.7 Transmiterea prin valoare a unei măsurători

```
1 struct Measurement
2 {
3     std::uint32_t timestamp;
4     float temperature;
5     float vibration;
6 };
```

```
7
8  void MeasurementTask(void* argument)
9  {
10     while (true)
11     {
12         const Measurement measurement =
13             acquire_measurement();
14
15         send_to_queue(
16             measurement_queue,
17             measurement,
18             queue_timeout);
19     }
20 }
```

Task-ul consumator primește propria copie:

Exemplu 8.8 Consumarea unui mesaj din coadă

```
1  void DiagnosisTask(void* argument)
2  {
3      Measurement measurement{};
4
5      while (true)
6      {
7          if (receive_from_queue(
8                  measurement_queue,
9                  measurement,
10                 wait_forever))
11         {
12             diagnose_equipment(measurement);
13         }
14     }
15 }
```

Transmiterea prin valoare oferă o regulă clară de proprietate: după copiere, producătorul și consumatorul nu mai accesează același obiect. Pentru mesaje mari, copierea poate deveni costisitoare, situație în care se pot transmite pointeri către buffere:

Exemplu 8.9 Transmiterea unui buffer prin pointer

```
1  SampleBuffer* buffer =
2      acquire_buffer_from_pool();
3
4  fill_sample_buffer(*buffer);
```

```
5
6 send_to_queue(
7       processing_queue ,
8       buffer ,
9       queue_timeout);
```

Această abordare necesită definirea explicită a proprietarului — după transmiterea cu succes a pointerului, producătorul nu mai trebuie să modifice bufferul până când acesta este returnat în pool.

Ciclul de proprietate poate fi:

$$\text{pool} \rightarrow \text{productor} \rightarrow \text{coad} \rightarrow \text{consumator} \rightarrow \text{pool}.$$

O eroare de proprietate poate conduce la:

- utilizarea unui buffer după eliberare;
- modificarea simultană de două task-uri;
- pierderea bufferului;
- returnarea de două ori în pool.

Capacitatea cozii trebuie dimensionată în funcție de rata de producere și rata de consum.

Dacă producătorul generează mesaje cu rata λ_P, iar consumatorul procesează cu rata λ_C, funcționarea stabilă pe termen lung necesită:

$$\lambda_C \geq \lambda_P. \tag{8.7}$$

Chiar dacă această condiție este îndeplinită în medie, pot apărea rafale. Capacitatea trebuie să acopere numărul maxim de mesaje acumulate într-o rafală:

$$N_Q \geq N_{\text{burst,max}}. \tag{8.8}$$

O coadă plină necesită o politică explicită: blocarea producătorului, abandonarea mesajului nou, eliminarea celui mai vechi mesaj, suprascrierea unei valori sau semnalizarea unei erori. Politica depinde de semnificația datelor — pentru o comandă critică, pierderea unui mesaj poate fi inacceptabilă, iar pentru o măsurătoare periodică, cea mai nouă valoare poate fi mai importantă decât valorile vechi.

Pentru un semnal de stare poate fi utilizat un mecanism de tip *mailbox*, care păstrează numai ultima valoare. Pentru un flux de evenimente trebuie păstrată ordinea și numărul mesajelor.

8.1.4 Comunicarea dintre ISR și task-uri

ISR-ul și task-urile reprezintă contexte de execuție diferite — ISR-ul trebuie să finalizeze rapid, iar procesarea complexă trebuie transferată unui task.

Modelul recomandat este:

eveniment hardware → ISR scurt → semnalizare → task de procesare.

Figura 8.4 prezintă acest transfer.

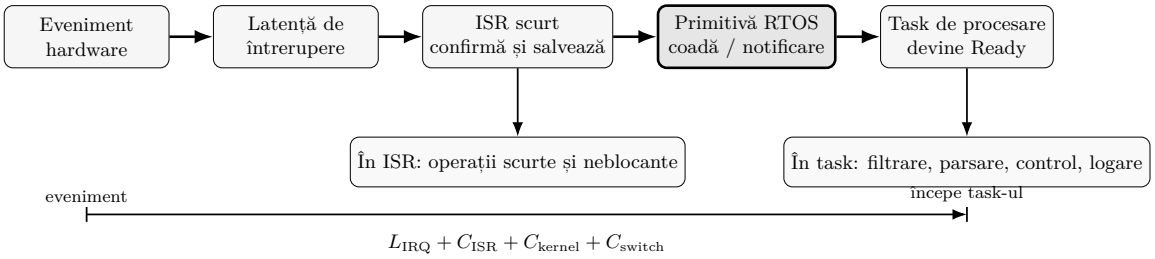


Figura 8.4 Transferul procesării de la ISR către un task.

ISR-ul poate utiliza notificarea directă a unui task, semafor binar, semafor de numărare, introducerea unui mesaj într-o coadă, actualizarea unui buffer DMA sau setarea unui eveniment.

Un exemplu generic este:

Exemplu 8.10 Transferul unui eveniment UART către un task

```
1  extern "C" void UART_IRQHandler()
2  {
3      const std::uint8_t received_byte =
4          read_uart_register();
5
6      const bool task_woken =
7          queue_send_from_isr(
8              uart_rx_queue,
9              received_byte);
10
11     request_context_switch_from_isr(
12         task_woken);
13 }
```

Task-ul consumator este:

Exemplu 8.11 Procesarea datelor UART într-un task

```
1  void UartProcessingTask(void* argument)
2  {
3      std::uint8_t received_byte = 0;
4
5      while (true)
```

```
6       {
7           receive_from_queue(
8               uart_rx_queue,
9               received_byte,
10              wait_forever);
11
12          process_uart_byte(received_byte);
13      }
14  }
```

Dacă ISR-ul deblochează un task cu prioritate mai mare, kernelul poate solicita o comutare de context imediat după ieșirea din întrerupere. Latența până la task este aproximativ:

$$L_{\text{ISR}\to\text{task}} = L_{\text{IRQ}} + C_{\text{ISR}} + C_{\text{kernel}} + C_{\text{switch}}. \tag{8.9}$$

Această valoare trebuie inclusă în bugetul temporal end-to-end. Funcțiile obișnuite ale RTOS-ului nu trebuie apelate automat din ISR — trebuie utilizate numai variantele documentate ca fiind sigure pentru întreruperi. Prioritatea hardware a ISR-ului poate impune restricții suplimentare: în unele portări, întreruperile aflate peste un anumit nivel de prioritate nu pot apela niciun serviciu al kernelului.

O rafală de date poate depăși capacitatea cozii. Pentru comunicații cu debit ridicat se utilizează frecvent DMA, buffer circular, double buffering, transferuri în bloc sau zero-copy. În double buffering, DMA scrie într-un buffer în timp ce task-ul procesează celălalt buffer:

$$B_0 \leftrightarrow B_1.$$

Sincronizarea trebuie să garanteze că procesorul nu citește bufferul în timp ce DMA îl modifică.

8.2 Probleme de concurență și protocoale de control al priorităților

Mecanismele de sincronizare rezolvă accesul concurent, dar pot introduce noi dependențe temporale. Un task poate fi blocat de un alt task, iar ordinea obținerii resurselor poate conduce la imposibilitatea progresului. Principalele probleme analizate sunt starvation, livelock, deadlock și priority inversion. Într-un sistem de timp real, problema nu este numai faptul că un task așteaptă, ci dacă durata maximă a așteptării poate fi limitată și inclusă în analiza timpului de răspuns.

8.2.1 Starvation, livelock și deadlock

Starvation apare atunci când un task rămâne pregătit sau solicită repetat o resursă, dar nu primește suficient timp de procesor ori acces la resursă. Într-un sistem cu priorități fixe, un task de prioritate redusă poate suferi starvation dacă task-urile superioare utilizează permanent procesorul:

$$\sum_{\tau_j \in hp(i)} \frac{C_j}{T_j} \approx 1. \tag{8.10}$$

Chiar dacă procesorul nu este utilizat complet în medie, rafalele frecvente pot produce întârzieri foarte mari. Starvation poate apărea și la mutex-uri dacă mecanismul de trezire favorizează repetat aceleași task-uri. Măsurile de prevenire includ analiza utilizării și a timpului de răspuns, limitarea timpului executat de task-urile superioare, blocarea task-urilor când nu au activitate, politici echitabile pentru resurse necritice și mecanisme de aging în sisteme care permit priorități dinamice.

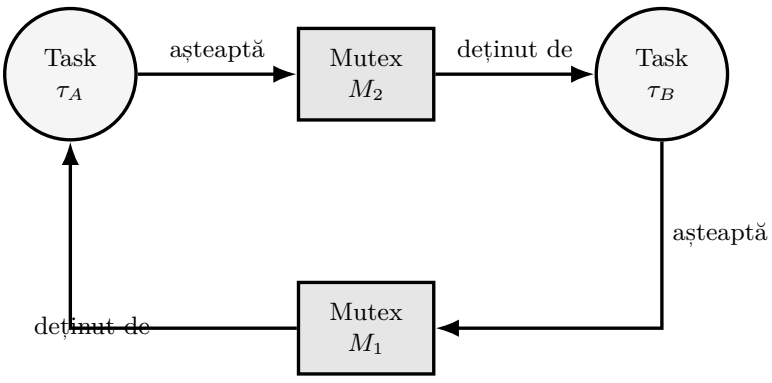
Livelock-ul apare când task-urile își modifică permanent starea ca răspuns unul la celălalt, dar nu finalizează activitatea utilă. De exemplu, două task-uri pot elibera și solicita repetat resurse pentru a evita o coliziune, fără ca vreunul să progreseze.

În deadlock, două sau mai multe task-uri rămân blocate permanent deoarece fiecare așteaptă o resursă deținută de alt task. Considerăm că task-ul τ_A deține mutex-ul M_1 și solicită M_2, iar task-ul τ_B deține mutex-ul M_2 și solicită M_1.

Rezultă un ciclu:

$$\tau_A \rightarrow M_2 \rightarrow \tau_B \rightarrow M_1 \rightarrow \tau_A.$$

Figura 8.5 prezintă acest graf de așteptare.



Ciclu: $\tau_A \rightarrow M_2 \rightarrow \tau_B \rightarrow M_1 \rightarrow \tau_A$

Prevenire: ordine globală a mutex-urilor, evitarea hold-and-wait, timeout și protocoale de plafon

Figura 8.5 Deadlock produs de obținerea resurselor într-o ordine incompatibilă.

Condițiile clasice asociate apariției deadlock-ului sunt:

1. excludere mutuală;

2. păstrarea unei resurse în timp ce este solicitată alta;

3. absența preluării forțate a resursei;

4. existența unei așteptări circulare.

Deadlock-ul poate fi prevenit prin eliminarea cel puțin a uneia dintre aceste condiții.

O metodă practică este impunerea unei ordini globale a resurselor:

$$M_1 < M_2 < M_3.$$

Toate task-urile trebuie să obțină mutex-urile numai în ordine crescătoare și să le elibereze în ordine inversă.

Exemplul corect este:

Exemplu 8.12 Obținerea mutex-urilor într-o ordine globală

```
1 lock(mutex_1);
2 lock(mutex_2);
3
4 use_shared_resources();
5
6 unlock(mutex_2);
7 unlock(mutex_1);
```

Timeout-urile pot detecta o așteptare excesivă, dar nu rezolvă automat consistența datelor. După expirarea timeout-ului, task-ul trebuie să elibereze resursele deja obținute și să readucă sistemul într-o stare validă.

8.2.2 Priority Inversion

Priority Inversion apare atunci când un task cu prioritate mare este blocat de un task cu prioritate mai mică ce deține o resursă necesară.

Blocarea directă este uneori inevitabilă:

$$\tau_H \rightarrow M \rightarrow \tau_L,$$

unde τ_H este task-ul cu prioritate ridicată, τ_L este task-ul cu prioritate redusă, iar M este mutex-ul deținut de τ_L. Problema devine severă atunci când un task de prioritate medie τ_M preemptă task-ul τ_L.

Secvența este:

1. τ_L obține mutex-ul;

2. τ_H devine Ready și solicită mutex-ul;

3. τ_H este blocat;
4. τ_L continuă pentru a elibera resursa;
5. τ_M devine Ready și preemptă τ_L;
6. τ_H rămâne blocat pe durata execuției lui τ_M.

Figura 8.6 prezintă situația.

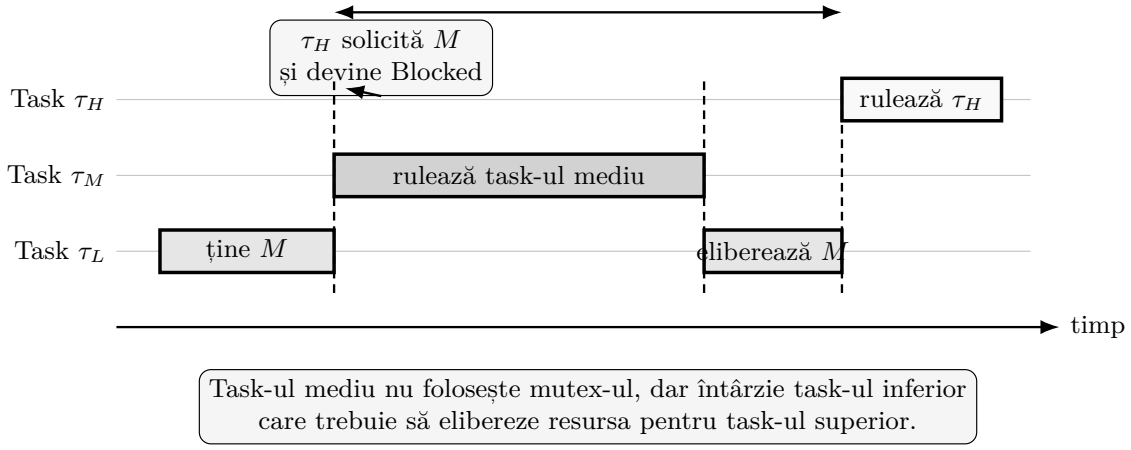


Figura 8.6 Priority Inversion produsă de un task intermediar.

Deși τ_M nu utilizează mutex-ul, execuția sa prelungește indirect blocarea task-ului τ_H. Fără un protocol de control, durata blocării poate depinde de toate task-urile de prioritate intermediară:

$$B_H = C_{L,\text{critic}} + \sum_{\tau_j \in mp(H,L)} I_j, \tag{8.11}$$

unde $mp(H, L)$ reprezintă task-urile cu prioritate între τ_H și τ_L.

Termenul „inversare nelimitată” nu înseamnă neapărat infinită într-o implementare concretă, ci faptul că blocarea nu mai este limitată doar de durata secțiunii critice a task-ului inferior.

Exemplu

Task-ul de logare τ_L deține un mutex timp de:

$$C_{L,\text{critic}} = 2\,\text{ms}.$$

Task-ul de control τ_H solicită mutex-ul și este blocat.
În acest moment, un task de diagnostic τ_M execută timp de:

$$C_M = 15\,\text{ms}.$$

Fără un protocol de control al priorităților, timpul de blocare al task-ului de control

poate ajunge la cel puțin:

$$B_H = 2 + 15 = 17\,\text{ms}.$$

Deși secțiunea critică durează numai 2 ms, interferența task-ului mediu mărește semnificativ întârzierea.

Un exemplu istoric bine cunoscut este sistemul misiunii Mars Pathfinder. O secvență de blocare pe o resursă partajată, combinată cu preempția produsă de activități cu prioritate intermediară, putea întârzia un task critic până la declanșarea watchdog-ului. Activarea mecanismului de Priority Inheritance pentru mutex-ul implicat a eliminat resetările observate [47].

Exemplul demonstrează că utilizarea unui mutex nu este suficientă — trebuie analizată și interacțiunea dintre mutex și prioritățile task-urilor.

8.2.3 Priority Inheritance Protocol

Priority Inheritance Protocol, prescurtat PIP, reduce inversarea priorităților prin ridicarea temporară a priorității task-ului care deține resursa.

Dacă task-ul τ_H este blocat de task-ul τ_L, atunci:

$$p_L^{\text{efectiv}} = \max\left(p_L^{\text{baz}}, p_H\right). \tag{8.12}$$

Task-ul inferior moștenește prioritatea task-ului blocat și poate finaliza secțiunea critică fără a fi preemptat de task-urile cu priorități intermediare. După eliberarea resursei, prioritatea sa revine la valoarea corespunzătoare:

$$p_L^{\text{efectiv}} \rightarrow p_L^{\text{baz}}. \tag{8.13}$$

Figura 8.7 prezintă efectul protocolului.

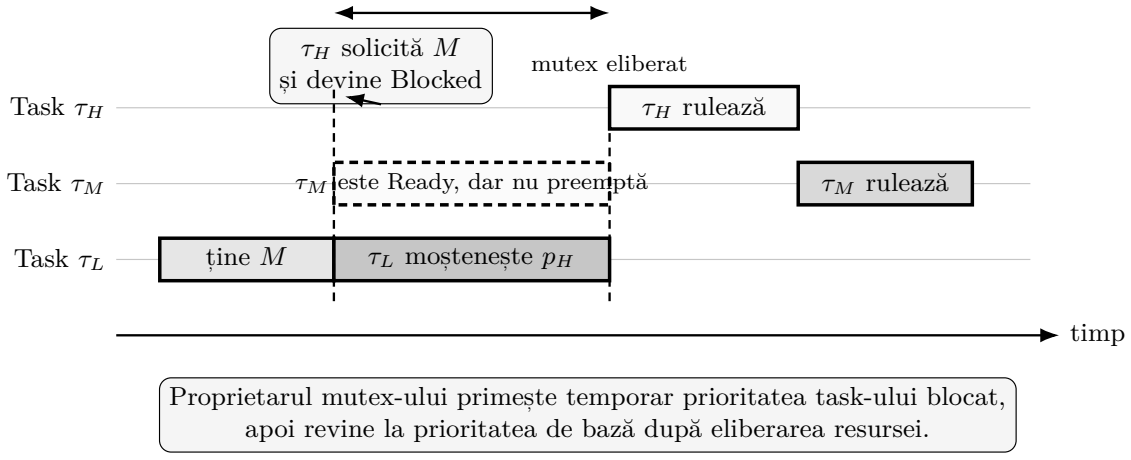


Figura 8.7 Reducerea blocării prin Priority Inheritance Protocol.

Pentru exemplul anterior:

1. τ_L deține mutex-ul;
2. τ_H solicită mutex-ul;
3. τ_L moștenește prioritatea lui τ_H;
4. τ_M nu mai poate preempta τ_L;
5. τ_L eliberează mutex-ul;
6. τ_H este deblocat;
7. τ_L revine la prioritatea de bază.

PIP trebuie să trateze și moștenirea tranzitivă. Dacă:

$$\tau_H \rightarrow M_1 \rightarrow \tau_L,$$

iar τ_L așteaptă un alt mutex deținut de τ_X, prioritatea ridicată trebuie propagată către τ_X — această situație este denumită blocare în lanț.

Priority Inheritance oferă avantaje importante: elimină interferența task-urilor cu prioritate intermediară, este integrat în numeroase mutex-uri RTOS și păstrează modelul obișnuit de obținere și eliberare a resursei. Protocolul are și limitări: nu previne automat deadlock-ul, poate permite blocări în lanț, analiza duratei maxime poate fi complexă, funcționează numai dacă este utilizat mutex-ul corespunzător și nu se aplică automat semafoarelor binare.

În multe RTOS-uri, Priority Inheritance este implementat pentru mutex-uri, nu pentru semafoare binare — înlocuirea unui mutex cu un semafor poate elimina protecția împotriva inversării priorităților. Task-ul care deține mutex-ul nu trebuie suspendat, întârziat voluntar sau blocat pe o operație lentă în interiorul secțiunii critice; Priority Inheritance nu poate compensa o secțiune critică proiectată necorespunzător.

8.2.4 Priority Ceiling și reguli de proiectare

Protocoalele bazate pe plafonul de prioritate atribuie fiecărei resurse o valoare numită *priority ceiling*. Pentru resursa R_k, plafonul este cea mai mare prioritate dintre task-urile care o pot utiliza:

$$\Pi(R_k) = \max_{\tau_i \text{ utilizeaz } R_k} (p_i)\,. \tag{8.14}$$

Există mai multe variante ale protocolului.

În *Original Priority Ceiling Protocol*, un task poate obține o resursă numai dacă prioritatea sa este mai mare decât plafonul curent al resurselor blocate de alte task-uri. Protocolul introduce o regulă de admitere care previne anumite secvențe periculoase.

În *Immediate Ceiling Priority Protocol*, denumit și protocol cu plafon imediat, task-ul care obține o resursă primește imediat prioritatea plafonului:

$$p_i^{\text{efectiv}} = \max\left(p_i^{\text{baz}}, \Pi(R_k)\right). \tag{8.15}$$

Figura 8.8 prezintă varianta cu plafon imediat.

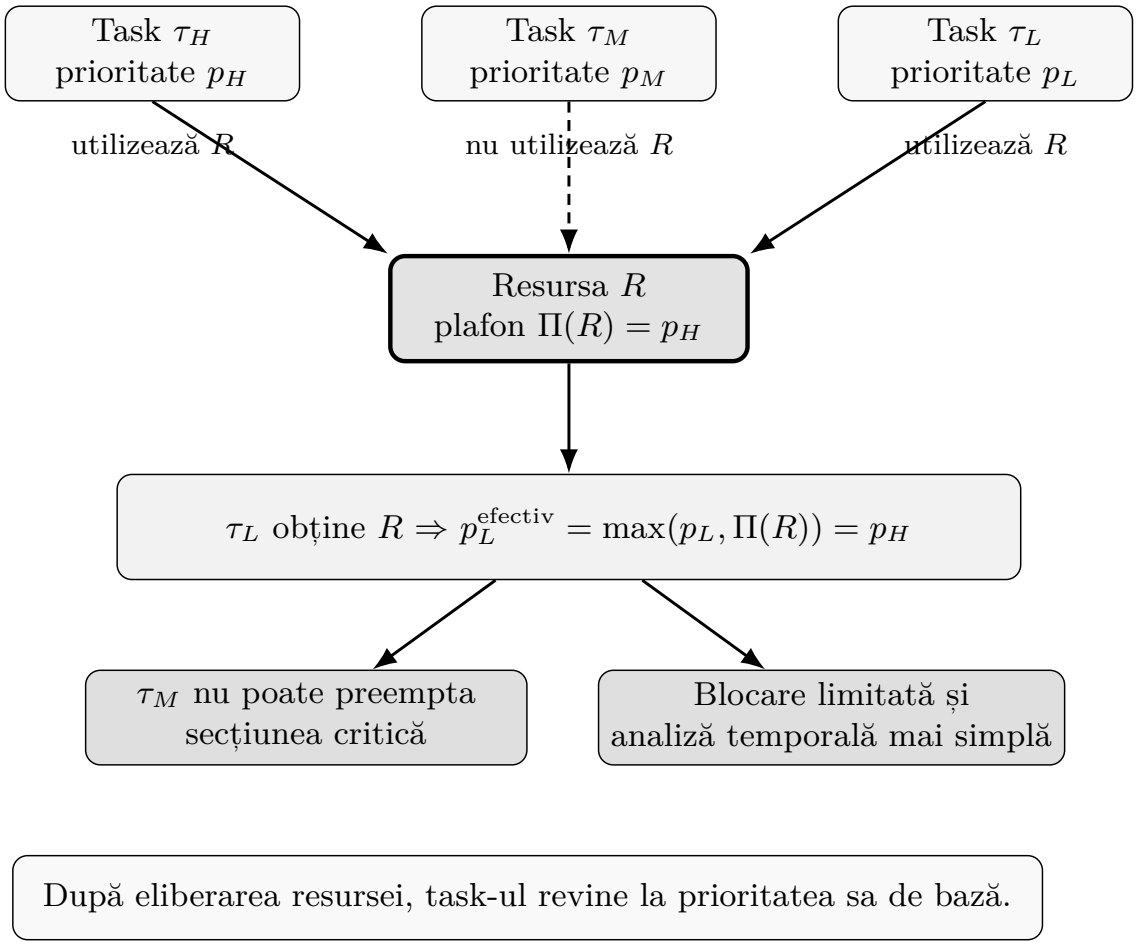


Figura 8.8 Protejarea unei resurse prin plafonul imediat de prioritate.

Protocoalele de tip Priority Ceiling pot oferi, în ipotezele modelului clasic:

- prevenirea deadlock-ului produs de resursele gestionate;
- limitarea blocării în lanț;
- o limită mai simplă pentru timpul de blocare;
- predictibilitate temporală mai bună.

În modelul clasic, un task poate fi blocat de cel mult o secțiune critică a unui task cu prioritate mai mică:

$$B_i \leq \max_{\tau_j \in lp(i)} \left(C_{j,\text{critic relevant}}\right), \tag{8.16}$$

unde $lp(i)$ reprezintă task-urile cu prioritate mai mică ce utilizează resurse cu plafon relevant pentru τ_i.

Această proprietate simplifică analiza timpului de răspuns:

$$R_i^{(k+1)} = C_i + B_i + \sum_{\tau_j \in hp(i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j. \tag{8.17}$$

Alegerea între Priority Inheritance și Priority Ceiling depinde de serviciile oferite de RTOS, posibilitatea cunoașterii tuturor utilizatorilor resursei, cerințele de analiză, dinamica priorităților și nivelul de criticitate. Priority Ceiling necesită cunoașterea anticipată a priorității maxime a task-urilor care pot accesa resursa — această condiție este potrivită pentru sisteme statice și bine analizate, dar poate fi dificilă pentru arhitecturi foarte dinamice.

Tabela 8.2 Compararea protocoalelor de control al priorităților

Protocol	Avantaj principal	Limitare principală
Fără protocol	Implementare minimală	Inversare de prioritate potențial nelimitată
Priority Inheritance	Elimină interferența priorităților intermediare	Nu previne automat deadlock-ul și blocarea în lanț
Original Priority Ceiling	Previne deadlock-ul și limitează blocarea	Necesită administrarea plafonului sistemului
Immediate Ceiling	Comportament predictibil și implementare eficientă	Necesită stabilirea corectă a plafonului fiecărei resurse

Regulile practice pentru resurse partajate sunt:

1. secțiunile critice trebuie menținute scurte;
2. task-ul nu trebuie să execute întârzieri în timp ce deține un mutex;
3. nu trebuie realizate operații de intrare–ieșire cu durată necontrolată în interiorul secțiunii critice;
4. mutex-urile multiple trebuie obținute într-o ordine globală;
5. fiecare operație de obținere trebuie să aibă o strategie pentru timeout și eroare;
6. trebuie utilizat un mutex cu protocol de prioritate atunci când resursa este partajată de task-uri cu priorități diferite;
7. semafoarele trebuie utilizate pentru semnalizare, nu ca înlocuitor automat al mutex-urilor;
8. ISR-urile nu trebuie să aștepte mutex-uri;
9. timpul maxim de blocare trebuie inclus în analiza schedulabilității;
10. proprietatea datelor trebuie definită explicit pentru bufferele transmise între task-uri.

8.3 Arhitecturi de kernel și izolare software

Arhitectura kernelului stabilește distribuția responsabilităților între codul privilegiat și componentele aplicației. Această decizie influențează latența serviciilor sistemului de operare, costul comunicației dintre componente, izolarea erorilor, suprafața software executată cu privilegii, posibilitatea certificării, complexitatea integrării driverelor și extensibilitatea sistemului.

Clasificările precum *kernel monolitic* și *microkernel* descriu principii arhitecturale, iar implementările concrete pot combina elemente din mai multe modele.

De exemplu, un sistem poate produce o singură imagine executabilă, dar poate utiliza MPU pentru izolarea anumitor task-uri; în alt caz, un microkernel poate rula servicii complexe în componente separate, deși întregul produs este distribuit ca o singură imagine firmware.

8.3.1 Kernel în spațiu unic și arhitectură monolitică

Într-o arhitectură cu spațiu unic de adrese, kernelul, driverele și aplicația pot accesa aceeași memorie, iar componentele sunt legate frecvent într-o singură imagine executabilă.

Structura conceptuală este:

$$\text{Aplicație} + \text{servicii} + \text{kernel} + \text{drivere} \rightarrow \text{imagine firmware}.$$

Apelarea unui serviciu poate fi realizată direct printr-o funcție:

$$C_{\text{serviciu}} \approx C_{\text{apel}} + C_{\text{kernel}}. \tag{8.18}$$

Nu sunt necesare obligatoriu schimbarea spațiului de adrese sau copierea mesajelor între procese izolate. Avantajele principale sunt latența redusă, overhead-ul mic, utilizarea eficientă a memoriei, integrarea simplă pe microcontrolere, accesul direct la periferice și configurarea statică. Acest model este potrivit pentru platforme cu memorie Flash și RAM limitate, microcontrolere fără MMU, număr redus de componente, cerințe stricte de latență și arhitectură software controlată integral de aceeași echipă.

Limitarea principală este propagarea erorilor — un pointer invalid dintr-un driver poate modifica memoria kernelului sau stiva altui task. Probabilitatea propagării nu depinde numai de dimensiunea codului, însă o bază mare de cod privilegiat mărește numărul componentelor care trebuie considerate de încredere.

Baza software de încredere poate fi notată prin:

$$\text{TCB}_{\text{security}} = \{\text{kernel}, \text{drivere}, \text{servicii privilegiate}, \text{configurație}\}, \tag{8.19}$$

unde TCB provine, în acest context, de la *Trusted Computing Base* și nu trebuie confundat cu *Task Control Block*.

Figura 8.9 compară organizarea într-un spațiu unic cu modelul microkernel.

În sistemele mici, lipsa izolării poate fi acceptată dacă:

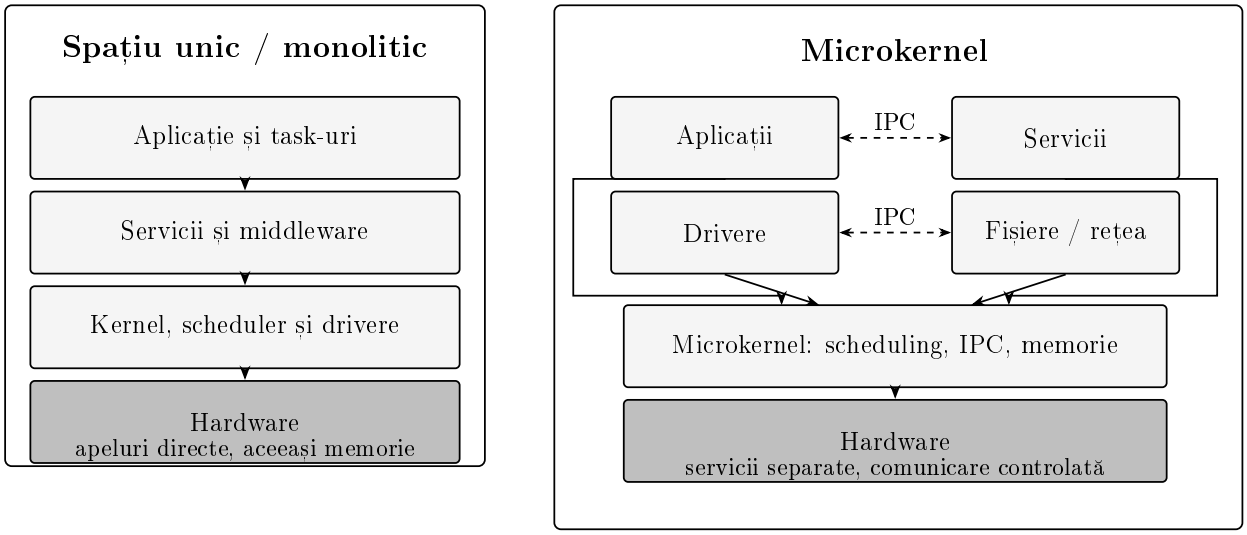


Figura 8.9 Compararea arhitecturii cu spațiu unic și a arhitecturii microkernel.

- software-ul este redus;
- toate componentele sunt verificate împreună;
- nu se execută cod provenit din surse diferite;
- mecanismele de diagnosticare detectează rapid coruperea;
- riscul este compatibil cu cerințele produsului.

Totuși, simplitatea arhitecturală nu elimină necesitatea protejării stivelor, validării pointerilor și controlului accesului concurent.

8.3.2 Microkernel și servicii separate

Un microkernel păstrează în modul privilegiat numai mecanismele considerate esențiale. Acestea pot include:

- planificarea;
- gestionarea firelor de execuție;
- comunicația inter-proces;
- gestionarea spațiilor de adrese;
- întreruperile;
- controlul accesului la resurse.

Driverele, sistemele de fișiere, stivele de rețea și alte servicii pot fi executate în componente separate.

Un serviciu este accesat prin schimb de mesaje:

$$\text{client} \rightarrow \text{IPC} \rightarrow \text{server}.$$

Costul unei operații poate fi aproximat prin:

$$C_{\text{microkernel}} = C_{\text{apel}} + C_{\text{IPC}} + C_{\text{schedule}} + C_{\text{serviciu}} + C_{\text{retur}}. \tag{8.20}$$

Costul IPC depinde de numărul schimbărilor de context, dimensiunea mesajului, copierea datelor, modificarea spațiului de adrese, mecanismele hardware de protecție și implementarea kernelului.

Avantajele microkernelului sunt izolarea mai bună a componentelor, reducerea codului privilegiat, repornirea separată a unor servicii, controlul explicit al comunicației, posibilitatea construirii sistemelor cu criticitate mixtă și analizarea separată a componentelor. Dacă un driver executat neprivilegiat eșuează, kernelul poate împiedica accesul acestuia la memoria altor componente, sistemul putând decide oprirea driverului, repornirea serviciului, trecerea într-un mod degradat sau notificarea unei componente de supraveghere.

Microkernelul nu garantează singur siguranța sau securitatea — proprietățile depind de corectitudinea kernelului, configurarea drepturilor, serviciile executate deasupra kernelului, protocolul IPC, gestionarea erorilor și arhitectura hardware. Un microkernel configurat greșit poate acorda unui serviciu acces excesiv la memorie sau periferice.

Separarea componentelor trebuie să fie însoțită de o politică explicită:

$$\text{Permisiuni}\,(C_i) = \{R_1, R_2, \ldots, R_m\}\,, \tag{8.21}$$

unde C_i este componenta, iar R_j sunt resursele pe care aceasta le poate accesa.

Principiul privilegiului minim impune:

$$\text{Permisiuni}\,(C_i) = \text{minimum necesar}. \tag{8.22}$$

8.3.3 Protecția memoriei și domenii de privilegii

Izolarea software necesită suport hardware sau mecanisme de instrumentare. Două componente hardware importante sunt MPU (*Memory Protection Unit*) și MMU (*Memory Management Unit*). Un MPU definește regiuni de memorie și drepturi precum citire, scriere, execuție, acces privilegiat și acces neprivilegiat.

Pentru o regiune R_k poate fi definită politica:

$$P(R_k) = (r_k, w_k, x_k, q_k)\,, \tag{8.23}$$

unde r_k permite citirea, w_k permite scrierea, x_k permite execuția, iar q_k reprezintă nivelul de privilegiu. MMU-ul poate realiza suplimentar traducerea adreselor virtuale în adrese fizice:

$$A_{\text{fizic}} = \mathcal{M}\,(A_{\text{virtual}})\,. \tag{8.24}$$

MMU-ul permite spații de adrese separate, paginare și politici complexe, costul hardware și software fiind în general mai mare decât pentru un MPU.

Figura 8.10 prezintă izolarea task-urilor prin regiuni protejate.

Într-un sistem cu MPU, un task neprivilegiat poate avea acces la propria stivă, datele proprii, un buffer partajat explicit, anumite periferice și funcțiile kernelului prin apeluri con-

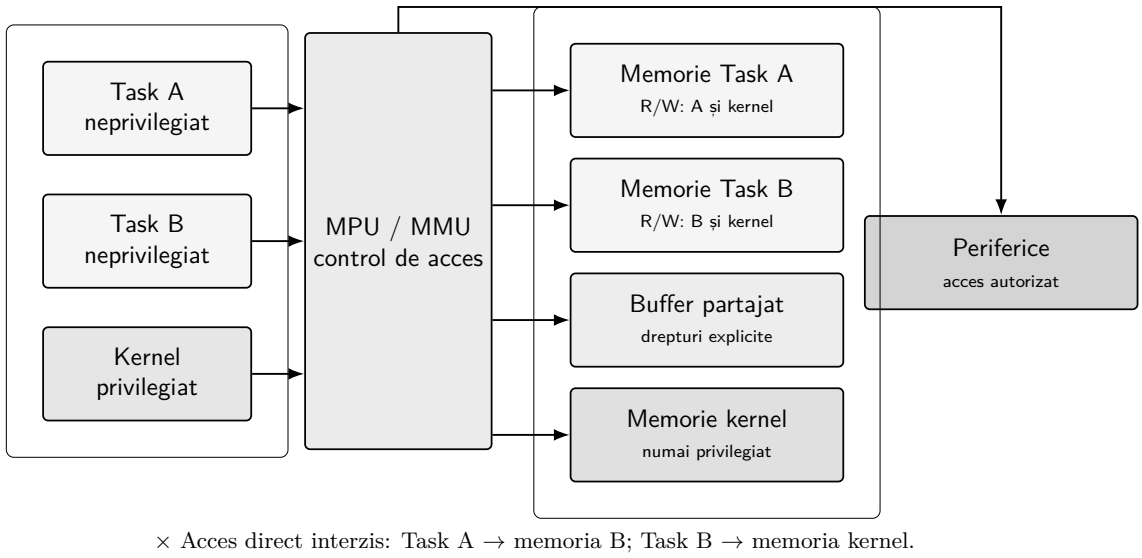


Figura 8.10 Izolarea componentelor software prin domenii de memorie.

trolate. O încercare de acces în afara regiunilor permise generează o excepție, iar handlerul de eroare poate identifica task-ul care a produs accesul, adresa accesată, tipul operației, registrul program counter și starea procesorului.

Protecția memoriei poate detecta depășirea stivei, scrierea în memorie read-only, execuția dintr-o regiune de date, accesul neautorizat la periferice și coruperea memoriei altui task. Ea nu detectează automat erori logice în interiorul regiunii permise, utilizarea incorectă a unei valori valide, deadline-uri ratate, protocoale incorecte sau date eronate furnizate printr-o interfață autorizată. Izolarea trebuie combinată cu validarea parametrilor, timeout-uri, supraveghere, verificarea integrității, politici de recuperare și analiză temporală.

8.3.4 Multicore și sisteme cu criticitate mixtă

Procesoarele embedded moderne pot conține mai multe nuclee. Două modele principale sunt AMP (*Asymmetric Multiprocessing*) și SMP (*Symmetric Multiprocessing*).

În AMP, fiecare nucleu poate executa o instanță separată de software:

$$\text{Core}_0 \rightarrow \text{RTOS}_0, \qquad \text{Core}_1 \rightarrow \text{RTOS}_1.$$

În SMP, o singură instanță a kernelului planifică task-urile pe mai multe nuclee:

$$\text{RTOS} \rightarrow \{\text{Core}_0, \text{Core}_1, \ldots, \text{Core}_{m-1}\}.$$

Figura 8.11 compară cele două modele.

AMP oferă o separare arhitecturală mai clară. De exemplu:

- un nucleu execută funcții de control;
- un nucleu execută comunicații;
- un nucleu execută funcții necritice.

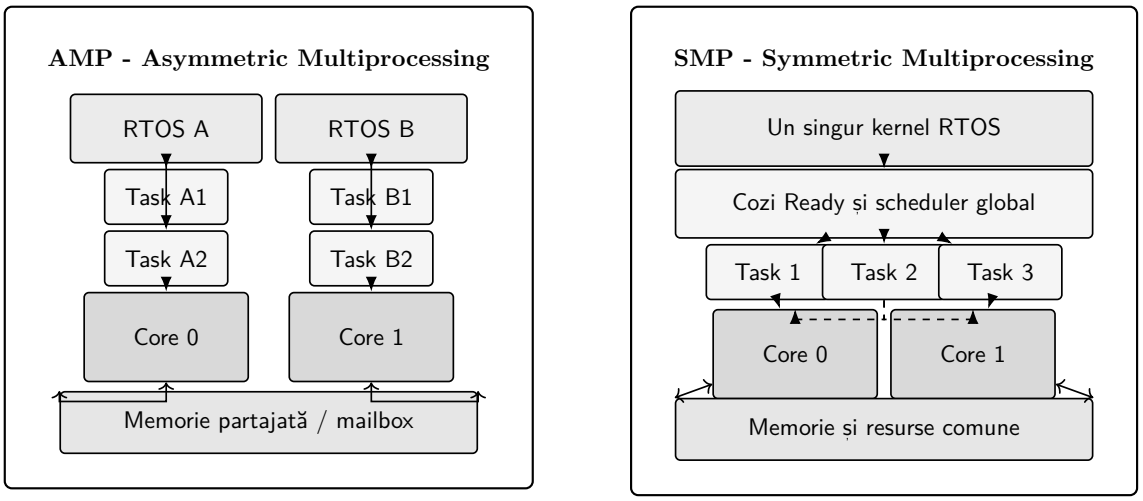


Figura 8.11 Modelele AMP și SMP pentru sisteme embedded multicore.

Comunicarea se realizează prin:

- memorie partajată;
- mailbox-uri hardware;
- întreruperi inter-core;
- cozi de mesaje;
- periferice de comunicație.

SMP permite utilizarea flexibilă a nucleelor, dar introduce:

- execuție paralelă reală;
- migrarea task-urilor;
- lock-uri între nuclee;
- interferență asupra cache-ului;
- interferență pe magistrale și memorie;
- dificultăți suplimentare în analiza WCET.

Utilizarea totală nu mai este descrisă suficient prin simpla condiție:

$$U \leq m,$$

unde m este numărul nucleelor. Task-urile pot avea afinitate, secțiuni seriale, resurse comune și interferență hardware.

Afinitatea poate restricționa task-ul τ_i la o mulțime de nuclee:

$$A_i \subseteq \{0, 1, \ldots, m-1\}. \tag{8.25}$$

Într-un sistem cu criticitate mixtă, funcții cu niveluri diferite de criticitate utilizează aceeași platformă.

Exemple:

- control critic și interfață grafică;

- funcție de siguranță și conectivitate;
- monitorizare critică și diagnosticare;
- control de zbor și funcții de misiune.

Separarea trebuie realizată atât spațial, cât și temporal. Izolarea spațială impune:

$$M_i \cap M_j = \varnothing, \tag{8.26}$$

pentru regiunile private ale componentelor i și j.

Izolarea temporală limitează consumul de procesor al unei componente:

$$C_i \leq Q_i \quad \text{în fiecare interval } P_i, \tag{8.27}$$

unde Q_i este bugetul, iar P_i este perioada de reîncărcare. Un sistem poate trece într-un mod de criticitate ridicată atunci când timpul de execuție al unei funcții critice depășește estimarea normală, iar activitățile necritice pot fi reduse sau suspendate pentru păstrarea funcțiilor esențiale.

8.4 RTOS-uri moderne

Ecosistemul sistemelor embedded include kerneluri compacte pentru microcontrolere, sisteme apropiate de POSIX, platforme Linux cu preempție în timp real și microkerneluri cu izolare avansată.

Comparația trebuie realizată la nivelul unei versiuni și configurații concrete. Numele produsului nu determină singur latența maximă, memoria ocupată, certificabilitatea, securitatea sau suportul pentru o anumită platformă — opțiunile de compilare, driverele, BSP-ul și configurația aplicației pot modifica semnificativ rezultatul.

8.4.1 FreeRTOS

FreeRTOS este un kernel open-source destinat microcontrolerelor și microprocesoarelor embedded de dimensiune redusă [30].

Kernelul oferă mecanisme precum task-uri, priorități, planificare preemptivă sau cooperativă, cozi, semafoare, mutex-uri, notificări de task, grupuri de evenimente, temporizatoare software și mecanisme pentru integrarea cu ISR-uri. Arhitectura obișnuită produce o imagine statică în care aplicația și kernelul sunt legate împreună.

Un task poate fi creat conceptual prin:

Exemplu 8.13 Crearea unui task FreeRTOS

```
1 BaseType_t result =
2     xTaskCreate(
3         SensorTask,
4         "Sensor",
5         sensor_stack_size,
6         nullptr,
```

```
7           sensor_priority,
8           &sensor_task_handle);
9
10  if (result != pdPASS)
11  {
12      handle_task_creation_failure();
13  }
```

Un task periodic poate utiliza:

Exemplu 8.14 Task periodic FreeRTOS

```
1   void SensorTask(void* parameters)
2   {
3       TickType_t last_wake_time =
4           xTaskGetTickCount();
5
6       while (true)
7       {
8           acquire_sensor_data();
9
10          vTaskDelayUntil(
11              &last_wake_time,
12              pdMS_TO_TICKS(10));
13      }
14  }
```

Kernelul poate fi configurat pentru planificare single-core, configurații AMP, configurații SMP pe platformele suportate, alocare statică sau dinamică, funcționare tickless și protecție cu MPU în anumite portări. Suportul SMP nu transformă automat o aplicație single-core într-o aplicație sigură pentru execuție paralelă — datele globale, driverele și secțiunile critice trebuie reevaluate.

FreeRTOS este potrivit când sunt importante amprenta redusă, integrarea rapidă pe un microcontroler, controlul configurației, un API relativ compact și suportul larg oferit de producătorii de procesoare. Kernelul de bază nu reprezintă singur un sistem complet — rețeaua, stocarea, securitatea și actualizarea software necesită biblioteci suplimentare.

Există versiuni LTS care păstrează stabilitatea funcțională și primesc remedieri critice pentru intervalul de mentenanță declarat [31]. Pentru aplicații cu cerințe de siguranță trebuie analizată varianta concretă, documentația de certificare, configurația și lanțul de dezvoltare — utilizarea kernelului open-source standard nu echivalează automat cu utilizarea unui produs precertificat.

8.4.2 Zephyr

Zephyr este un sistem de operare open-source destinat platformelor embedded, de la dispozitive cu resurse limitate până la controlere și produse IoT mai complexe [96].

Ecosistemul său include kernel preemptiv, model de drivere, Devicetree, configurare prin Kconfig, stive de comunicație, sisteme de fișiere, management energetic, servicii criptografice, mecanisme de actualizare și infrastructură de testare. Descrierea hardware este separată de logica aplicației prin Devicetree.

Conceptual:

Exemplu 8.15 Accesarea unui dispozitiv descris prin Devicetree

```
1 const device* sensor =
2     DEVICE_DT_GET(DT_ALIAS(environment_sensor));
3
4 if (!device_is_ready(sensor))
5 {
6     handle_sensor_initialization_failure();
7 }
```

Configurarea selectează numai componentele necesare, această abordare permițând eliminarea codului neutilizat și adaptarea aceleiași aplicații la mai multe platforme. Zephyr oferă mecanisme de userspace și domenii de memorie pe arhitecturile care furnizează suportul hardware necesar — un fir neprivilegiat poate primi acces numai la obiectele kernelului și regiunile autorizate.

Avantajele principale sunt ecosistemul integrat, suportul pentru numeroase clase de dispozitive, subsistemele de conectivitate, configurarea declarativă, suportul pentru protecția memoriei și infrastructura pentru testare automată. Complexitatea ecosistemului este mai mare decât în cazul unui kernel minimal — echipa trebuie să gestioneze configurația Kconfig, Devicetree, versiunea toolchain-ului, modulele externe, migrarea între versiuni și dimensiunea imaginii rezultate.

Pentru produse cu durată mare de viață trebuie selectată o versiune menținută și trebuie planificată actualizarea componentelor.

8.4.3 Apache NuttX și Linux cu PREEMPT_RT

Apache NuttX este un RTOS open-source cu accent pe standarde și pe interfețe apropiate de POSIX [5].

Acesta oferă concepte familiare dezvoltatorilor Unix:

- fire POSIX;
- descriptori de fișiere;
- sisteme de fișiere;
- socket-uri;
- semnale;

- API-uri standardizate;
- interfețe pentru drivere.

O aplicație poate utiliza:

Exemplu 8.16 Crearea unui fir prin API POSIX

```
pthread_t thread{};
pthread_attr_t attributes{};

pthread_attr_init(&attributes);

const int result =
    pthread_create(
        &thread,
        &attributes,
        worker_thread,
        nullptr);

if (result != 0)
{
    handle_thread_creation_failure(result);
}
```

Avantajul principal este reducerea distanței conceptuale dintre dezvoltarea embedded și dezvoltarea pe sisteme POSIX.

NuttX este potrivit pentru aplicații care necesită:

- mai multe servicii de sistem;
- portarea codului POSIX;
- sisteme de fișiere;
- rețea;
- interfețe uniforme pentru drivere;
- o arhitectură mai apropiată de un sistem de operare complet.

Costul poate fi mai mare decât pentru un kernel minimal, în funcție de configurația selectată.

Linux cu PREEMPT_RT reprezintă o categorie diferită. Linux oferă:

- procese și spații virtuale de adrese;
- drivere numeroase;
- stive de rețea complexe;
- sisteme de fișiere;
- containere și virtualizare;
- instrumente avansate de diagnostic.

Mecanismele PREEMPT_RT reduc regiunile nepreemptibile și transformă numeroase întreruperi în fire planificabile [52].

Timpul de răspuns al unei aplicații Linux depinde de:

- versiunea și configurația kernelului;
- politica de scheduling;
- prioritățile firelor;
- afinitatea CPU;
- drivere;
- firmware;
- hardware;
- trafic de memorie și I/O;
- management energetic.

Utilizarea PREEMPT_RT nu demonstrează singură respectarea unui deadline. Platforma trebuie măsurată și analizată în configurația produsului.

Linux este potrivit atunci când aplicația necesită funcționalități extinse, iar platforma dispune de:

- procesor performant;
- MMU;
- memorie suficientă;
- stocare;
- un proces de mentenanță și actualizare.

O arhitectură eterogenă poate combina Linux cu un RTOS:

$$\text{Linux} \leftrightarrow \text{RTOS}.$$

Linux execută funcțiile complexe și interfața, iar RTOS-ul execută buclele de control cu deadline strict.

8.4.4 Platforme comerciale și kerneluri cu asigurare ridicată

Platformele comerciale, precum VxWorks și QNX, sunt utilizate în produse care necesită suport profesional, BSP-uri industriale, instrumente de analiză și pachete asociate unor procese de certificare. Selecția unui produs comercial poate oferi suport contractual, versiuni menținute, documentație pentru siguranță, instrumente de trasare și analiză, componente de rețea și stocare, integrare cu hypervisor și consultanță pentru certificare. Costurile includ licențe, redevențe sau taxe de distribuție, dependență de furnizor, costul actualizărilor, restricții contractuale și disponibilitatea specialiștilor.

Un kernel cu asigurare ridicată urmărește reducerea și verificarea bazei de cod privilegiate.

seL4 este un microkernel bazat pe capabilități și verificat formal pentru anumite configurații și proprietăți [49, 78].

O capabilitate reprezintă o referință controlată către o resursă și drepturile asociate:

$$\text{cap} = (\text{obiect}, \text{drepturi})\,. \tag{8.28}$$

O componentă nu poate accesa un obiect numai prin cunoașterea adresei — ea trebuie

să dețină capabilitatea corespunzătoare. Acest model permite izolare cu granulație fină, control explicit al autorității, construcția sistemelor cu componente de încredere diferită și utilizarea kernelului ca RTOS sau hypervisor. Verificarea formală a kernelului nu verifică automat aplicația utilizatorului, driverele externe, configurația capabilităților, hardware-ul, cerințele produsului sau analiza temporală completă.

Figura 8.12 prezintă poziționarea conceptuală a principalelor familii.

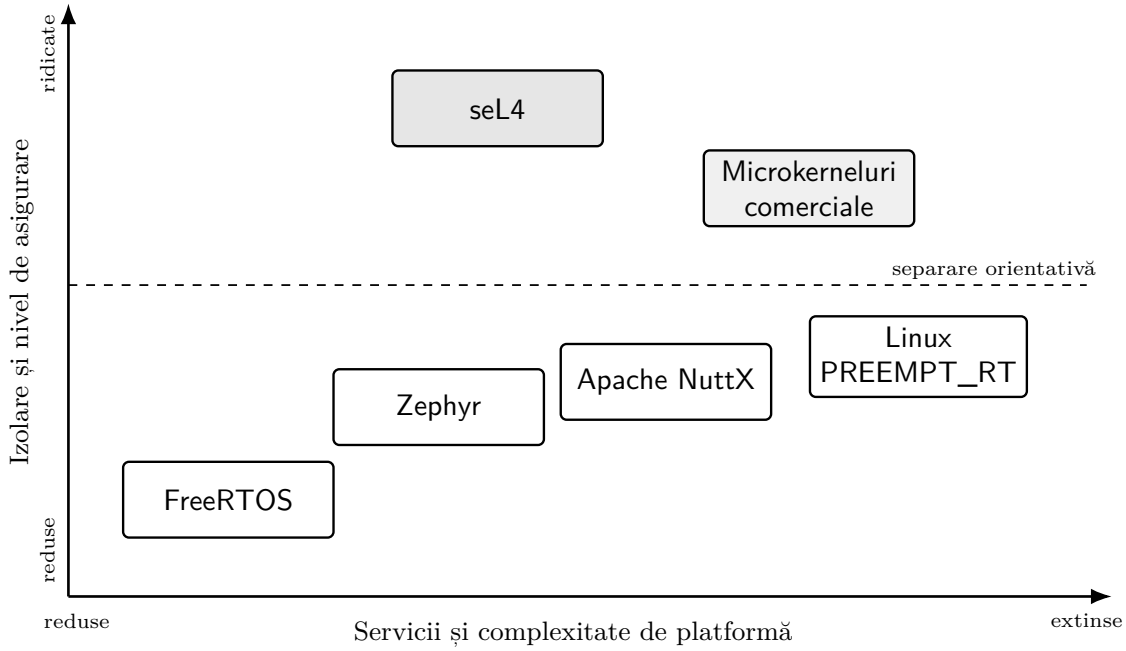


Figura 8.12 Poziționarea conceptuală a familiilor de sisteme de operare embedded.

Tabelul prezintă orientări generale, nu rezultate de benchmark și nici garanții — evaluarea trebuie realizată pentru configurația concretă.

8.5 Criterii de alegere a unui RTOS

Nu există un RTOS universal optim — selecția trebuie să pornească de la cerințele sistemului și nu de la popularitatea unei tehnologii.

Procesul trebuie să răspundă la întrebarea:

> Care platformă permite demonstrarea cerințelor produsului, cu un cost și un risc acceptabile pe întreaga durată de viață?

Criteriile trebuie ponderate — pentru un dispozitiv simplu, memoria și costul pot domina, iar pentru un sistem critic, izolarea, certificarea și predictibilitatea pot avea prioritate.

Tabela 8.3 Orientarea generală a unor familii de sisteme de operare embedded

Familie	Platformă tipică	Orientare dominantă	Aspect care trebuie evaluat
FreeRTOS	Microcontrolere și procesoare mici	Kernel compact și integrare directă	Serviciile suplimentare necesare produsului
Zephyr	Microcontrolere și dispozitive conectate	Ecosistem integrat și configurabil	Complexitatea configurației și a actualizării
Apache NuttX	Platforme embedded de dimensiuni variate	Interfețe apropiate de POSIX	Costul serviciilor activate
Linux PREEMPT_RT	Microprocesoare cu MMU	Funcționalitate generală și latență redusă	Configurația completă hardware–software
Microkernel comercial	Platforme industriale și critice	Izolare, suport și pachete de certificare	Costul și dependența de furnizor
seL4	Sisteme cu asigurare ridicată	Izolare bazată pe capabilități și verificare	Construcția serviciilor și integrarea aplicației

8.5.1 Cerințe funcționale, temporale și de resurse

Prima etapă constă în definirea funcționalităților obligatorii: task-uri și priorități, temporizatoare, mecanisme IPC, rețea, Bluetooth, Wi-Fi sau alte protocoale, USB, sistem de fișiere, actualizare software, management energetic, suport multicore și suport POSIX. Pentru fiecare funcționalitate trebuie precizat dacă este obligatorie, recomandată, opțională sau exclusă.

Cerințele temporale includ latența maximă a întreruperilor, timpul de comutare a contextului, jitter-ul, rezoluția timerelor, numărul de priorități, politica de planificare, suportul pentru protocoale de sincronizare și comportamentul la suprasarcină.

O cerință verificabilă este:

> În configurația de producție, task-ul de control trebuie să înceapă execuția în maximum $50\,\mu$s de la activarea întreruperii, cu toate funcțiile de comunicație active.

Resursele necesare trebuie măsurate pentru configurația reală:

$$M_{\mathrm{RAM}} = M_{\mathrm{kernel}} + \sum_{i=1}^{n} M_{\mathrm{stack},i} + M_{\mathrm{heap}} + M_{\mathrm{buffer}} + M_{\mathrm{driver}}. \tag{8.29}$$

Memoria Flash este:

$$M_{\mathrm{Flash}} = M_{\mathrm{kernel}} + M_{\mathrm{aplicație}} + M_{\mathrm{middleware}} + M_{\mathrm{date\ constante}}. \tag{8.30}$$

O comparație care utilizează numai dimensiunea kernelului gol este insuficientă. Produsul trebuie evaluat cu driverele, protocoalele și mecanismele de diagnostic necesare.

8.5.2 Siguranță, securitate și izolare

Pentru funcțiile critice trebuie analizate:

- nivelul de siguranță cerut;
- standardul aplicabil;
- existența documentației de certificare;
- calificarea toolchain-ului;
- trasabilitatea cerințelor;
- strategia de testare;
- istoricul versiunii.

Un certificat sau un pachet de siguranță este relevant numai dacă:

- versiunea utilizată este cea acoperită;
- configurația este permisă;
- procesorul și compilatorul sunt suportate;
- integratorul respectă manualele de utilizare;
- aplicația este dezvoltată prin procesul cerut.

Securitatea trebuie evaluată prin modelul de amenințări, separarea privilegiilor, MPU sau MMU, secure boot, actualizare autentică, stocarea cheilor, criptografie, gestionarea vulnerabilităților și durata suportului pentru remedieri. Protecția memoriei reduce impactul unor erori, dar nu înlocuiește secure boot-ul sau validarea actualizărilor.

O matrice de acces poate fi definită prin:

$$A_{ij} = \begin{cases} 1, & \text{componenta } C_i \text{ poate accesa resursa } R_j, \\ 0, & \text{accesul este interzis.} \end{cases} \tag{8.31}$$

Obiectivul este minimizarea drepturilor:

$$\min \sum_{i} \sum_{j} A_{ij}, \tag{8.32}$$

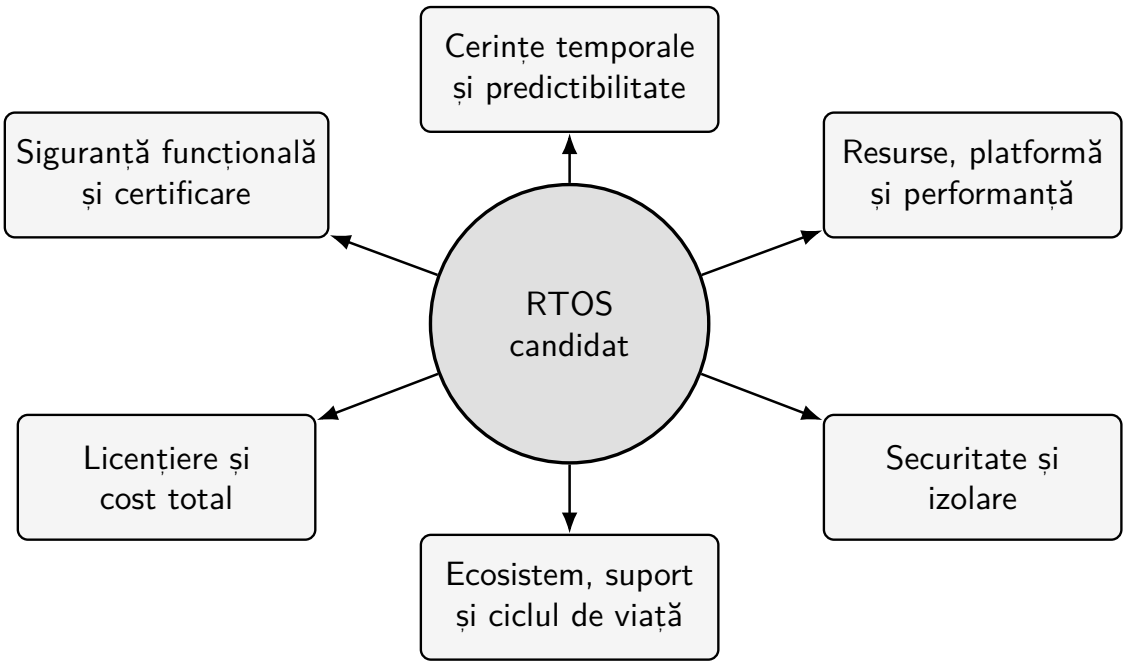
sub constrângerea păstrării funcționalității.

Figura 8.13 prezintă dimensiunile principale ale evaluării.

8.5.3 Ecosistem, licențiere și ciclul de viață

Costul unui RTOS nu este reprezentat numai de prețul licenței. Costul total poate fi estimat prin:

$$C_{\text{total}} = C_{\text{licență}} + C_{\text{integrare}} + C_{\text{dezvoltare}} + C_{\text{validare}} + C_{\text{mentenanță}} + C_{\text{migrare}}. \tag{8.33}$$



Selecția rezultă din satisfacerea simultană a cerințelor obligatorii.

Figura 8.13 Dimensiunile principale utilizate la evaluarea unui RTOS.

Un RTOS gratuit poate necesita un efort mare de integrare, iar un produs comercial poate reduce riscul tehnic, dar introduce costuri contractuale și dependență de furnizor. Trebuie analizate tipul licenței, obligațiile de distribuție, licențele componentelor terțe, disponibilitatea suportului comercial, activitatea comunității, frecvența versiunilor, politica LTS, procesul de raportare a vulnerabilităților, disponibilitatea BSP-urilor și suportul pentru instrumentele utilizate.

Un BSP disponibil nu garantează calitatea industrială. Trebuie verificate perifericele implementate, modul DMA, consumul energetic, suportul multicore, eratele procesorului, testele automate și mentenanța versiunii.

Durata de viață a produsului poate fi mai mare decât durata de suport a unei versiuni RTOS. Este necesar un plan pentru actualizări de securitate, migrarea toolchain-ului, actualizarea bibliotecilor, înlocuirea componentelor hardware, reproducerea build-ului și arhivarea surselor și configurațiilor. Un build reproductibil trebuie să păstreze versiunea exactă a surselor, configurația, toolchain-ul, scripturile, dependențele și hash-urile artefactelor.

8.5.4 Proces de selecție și validare experimentală

Selecția poate fi realizată în mai multe etape. Prima etapă elimină soluțiile care nu satisfac cerințele obligatorii:

$$\mathcal{C}_{\text{valid}} = \{R \mid R \text{ satisface toate cerințele obligatorii}\}. \tag{8.34}$$

A doua etapă atribuie ponderi criteriilor:

$$\sum_{k=1}^{m} w_k = 1. \tag{8.35}$$

Scorul candidatului R_j poate fi:

$$S_j = \sum_{k=1}^{m} w_k s_{j,k} - P_j, \tag{8.36}$$

unde $s_{j,k}$ este scorul candidatului pentru criteriul k, w_k este ponderea criteriului, iar P_j reprezintă penalizările și riscurile. Un scor numeric nu trebuie să ascundă o cerință obligatorie — o soluție fără suport pentru procesorul țintă nu poate compensa această lipsă printr-un scor bun la documentație.

După analiza documentației trebuie construit un prototip reprezentativ, care să includă procesorul țintă, driverele principale, numărul realist de task-uri, comunicațiile, încărcarea procesorului, mecanismele de protecție și scenarii de eroare. Trebuie măsurate latența întreruperilor, timpul ISR–task, timpul de comutare a contextului, jitter-ul task-urilor periodice, utilizarea RAM și Flash, timpul de pornire, consumul energetic, comportamentul la suprasarcină și recuperarea după erori.

Figura 8.14 prezintă procesul recomandat.

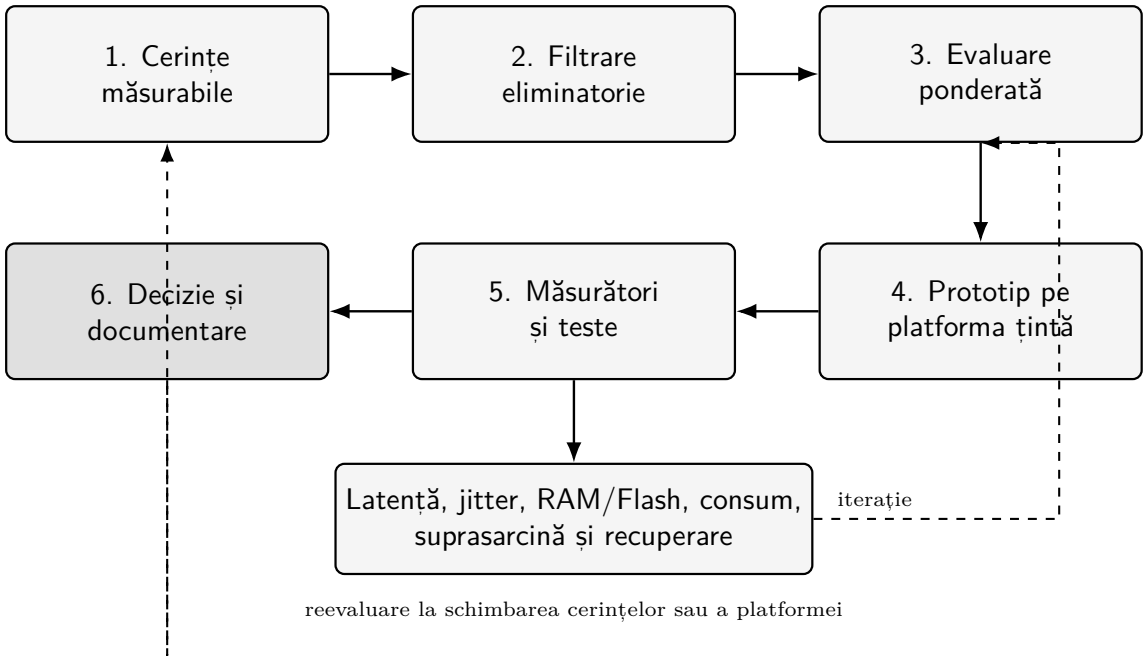


Figura 8.14 Proces recomandat pentru selecția și validarea unui RTOS.

Un benchmark trebuie să utilizeze aceeași platformă hardware, aceeași frecvență, aceleași optimizări ale compilatorului, aceeași metodă de măsurare, aceeași sarcină și aceeași definiție a latenței.

Compararea unor valori publicate de surse diferite poate fi incorectă deoarece metodele și configurațiile pot fi incompatibile. Rezultatul selecției trebuie documentat prin cerințele evaluate, candidații eliminați, ipotezele, rezultatele măsurătorilor, riscurile, justificarea soluției și condițiile care ar impune reevaluarea.

8.6 Studii de caz

Conceptele prezentate în acest capitol trebuie aplicate împreună. Alegerea task-urilor, priorităților și mecanismelor de comunicare influențează atât structura software, cât și comportamentul temporal al sistemului.

În cadrul acestei secțiuni sunt analizate două aplicații reprezentative:

- un robot mobil autonom;
- un sistem industrial de monitorizare a unui motor electric.

Valorile numerice au rol didactic. Într-un proiect real, timpii de execuție trebuie determinați pentru platforma, compilatorul și configurația software utilizate în produs.

8.6.1 Robot mobil autonom

Se consideră un robot mobil destinat navigației într-un mediu interior.

Platforma hardware conține:

- un microcontroler ARM Cortex-M7;
- encodere pentru măsurarea vitezei roților;
- o unitate de măsurare inerțială;
- un senzor LiDAR;
- drivere pentru motoare;
- un modul de comunicație wireless.

Aplicația trebuie să realizeze:

- achiziția datelor;
- controlul vitezei motoarelor;
- estimarea poziției;
- calculul traiectoriei;
- comunicația cu o stație externă;
- diagnosticarea sistemului.

O posibilă descompunere în task-uri este prezentată în Tabelul 8.4.

Prioritățile sunt atribuite conform principiului Rate Monotonic. Task-ul cu perioada cea mai scurtă primește prioritatea cea mai mare.

Figura 8.15 prezintă fluxul principal al datelor.

Encoderele și unitatea inerțială produc date pentru `SensorTask`. Rezultatele sunt transmise către `LocalizationTask`, care actualizează estimarea poziției.

`NavigationTask` utilizează poziția și datele LiDAR pentru calculul traiectoriei. Comanda de viteză rezultată este transmisă către `MotorControlTask`.

Comunicarea între task-uri poate utiliza:

- o coadă pentru măsurători;

Tabela 8.4 Task-urile periodice ale robotului mobil

Task	C_i	T_i	D_i	Prioritate
MotorControlTask	0,8 ms	5 ms	5 ms	Foarte ridicată
SensorTask	1,2 ms	10 ms	10 ms	Ridicată
LocalizationTask	2,5 ms	20 ms	20 ms	Medie
CommunicationTask	2 ms	50 ms	50 ms	Redusă
NavigationTask	5 ms	100 ms	100 ms	Foarte redusă

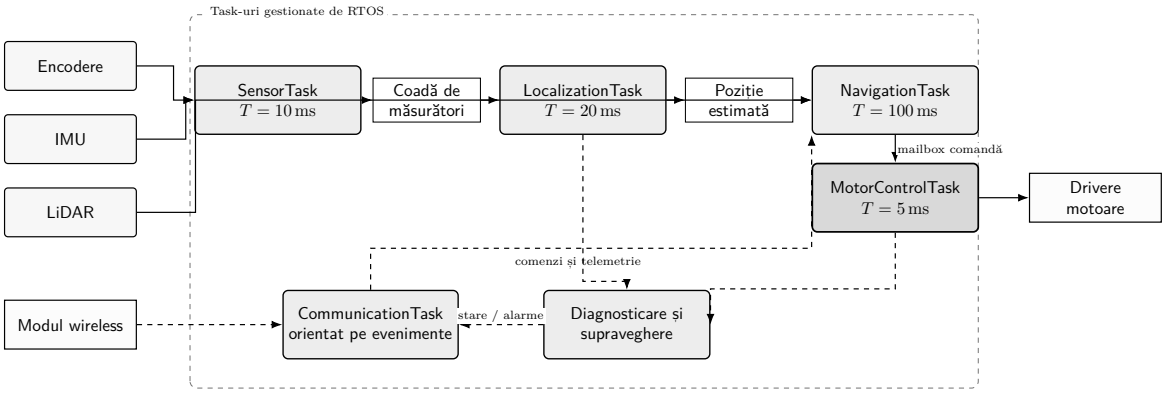


Figura 8.15 Arhitectura software bazată pe RTOS a robotului mobil.

- un buffer protejat pentru ultima poziție estimată;
- o coadă pentru comenzile de mișcare;
- un grup de evenimente pentru stările sistemului.

Task-ul de control trebuie să utilizeze cea mai recentă comandă disponibilă. Din acest motiv, o structură de tip mailbox poate fi mai potrivită decât o coadă care păstrează numeroase comenzi vechi.

Gradul de utilizare al procesorului este:

$$U = \frac{0{,}8}{5} + \frac{1{,}2}{10} + \frac{2{,}5}{20} + \frac{2}{50} + \frac{5}{100}. \tag{8.37}$$

Rezultă:

$$\begin{aligned} U &= 0{,}16 + 0{,}12 + 0{,}125 + 0{,}04 + 0{,}05 \\ &= 0{,}495. \end{aligned} \tag{8.38}$$

Prin urmare:

$$U = 49{,}5\%. \tag{8.39}$$

Pentru cinci task-uri, limita suficientă Liu–Layland este:

$$U_{\mathrm{LL}} = 5\left(2^{1/5} - 1\right) \approx 0{,}743. \tag{8.40}$$

Deoarece:

$$0{,}495 < 0{,}743,$$

setul este garantat schedulabil prin testul suficient RMS, în ipotezele modelului clasic.
Acest calcul nu include:

- timpii ISR;
- comutările de context;
- timpii de blocare;
- operațiile DMA;
- variația timpilor de execuție;
- eventualele task-uri sporadice.

Pentru o analiză mai precisă poate fi calculat timpul de răspuns.
Pentru `MotorControlTask`:

$$R_1 = C_1 = 0{,}8\,\mathrm{ms}. \tag{8.41}$$

Pentru `SensorTask`:

$$R_2^{(0)} = 1{,}2. \tag{8.42}$$

Prima iterație este:

$$\begin{aligned} R_2^{(1)} &= 1{,}2 + \left\lceil \frac{1{,}2}{5} \right\rceil 0{,}8 \\ &= 2\,\mathrm{ms}. \end{aligned} \tag{8.43}$$

A doua iterație produce aceeași valoare:

$$R_2 = 2\,\mathrm{ms} < D_2.$$

Pentru `LocalizationTask`:

$$R_3^{(0)} = 2{,}5, \tag{8.44}$$

$$\begin{aligned} R_3^{(1)} &= 2{,}5 + \left\lceil \frac{2{,}5}{5} \right\rceil 0{,}8 + \left\lceil \frac{2{,}5}{10} \right\rceil 1{,}2 \\ &= 4{,}5\,\mathrm{ms}. \end{aligned} \tag{8.45}$$

Următoarea iterație nu modifică rezultatul:

$$R_3 = 4{,}5\,\text{ms} < 20\,\text{ms}.$$

Pentru `CommunicationTask`, calculul converge la:

$$R_4 = 7{,}3\,\text{ms}.$$

Pentru `NavigationTask`, calculul converge la:

$$R_5 = 14{,}3\,\text{ms}.$$

Toate valorile respectă deadline-urile corespunzătoare.
Figura 8.16 prezintă o porțiune din planificarea task-urilor.

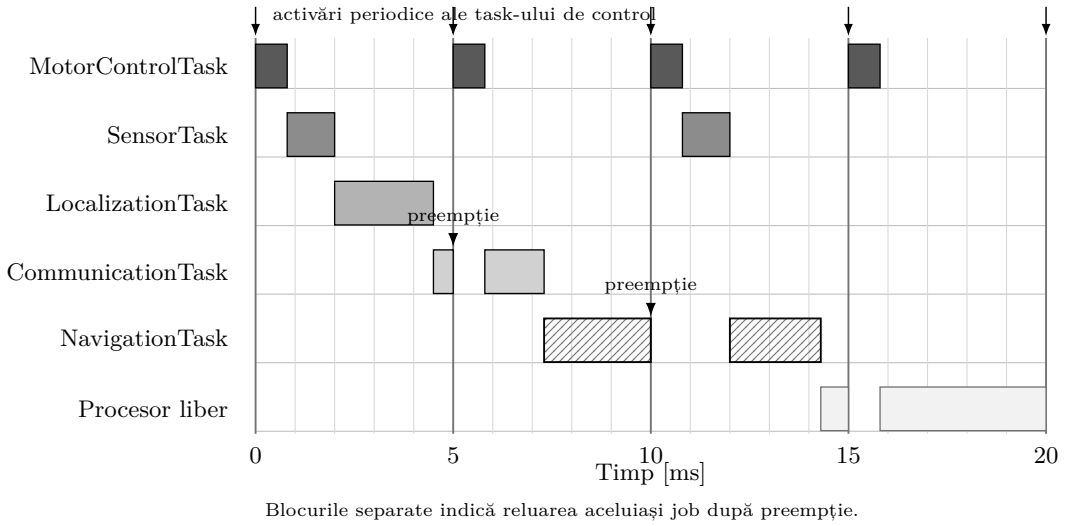


Figura 8.16 Exemplu de planificare a task-urilor robotului mobil.

Lanțul critic este:

$$\text{senzori} \rightarrow \text{localizare} \rightarrow \text{navigație} \rightarrow \text{control}.$$

Timpul end-to-end nu este egal cu timpul de răspuns al unui singur task. Acesta depinde de momentele de activare și de transferul datelor între task-uri.

O comandă calculată de `NavigationTask` poate aștepta până la activarea următoare a `MotorControlTask`. În cazul cel mai nefavorabil, această așteptare poate fi apropiată de:

$$T_{\text{MotorControl}} = 5\,\text{ms}.$$

Pentru reducerea latenței, task-ul de control poate fi deblocat imediat la sosirea unei comenzi noi, păstrând în același timp o activare periodică de siguranță.

Trebuie tratate și situațiile de eroare:

- lipsa datelor LiDAR;
- pierderea comunicației;
- depășirea timpului de execuție al localizării;
- blocarea unui motor;
- depășirea cozii de măsurători;
- coruperea unei stive.

Pentru fiecare situație trebuie definit un răspuns sigur. De exemplu, pierderea actualizărilor de localizare poate determina reducerea vitezei și oprirea controlată a robotului.

8.6.2 Sistem industrial de monitorizare

Se consideră un sistem care monitorizează temperatura și vibrațiile unui motor electric. Platforma conține:

- senzori de temperatură și vibrații;
- un convertor analog–digital;
- un microcontroler;
- memorie nevolatilă;
- interfață Ethernet sau RS-485;
- ieșire pentru alarmă.

Funcționalitățile principale sunt:

- achiziția măsurătorilor;
- filtrarea semnalelor;
- extragerea caracteristicilor;
- detectarea anomaliilor;
- înregistrarea evenimentelor;
- transmiterea alarmelor.

O posibilă descompunere este prezentată în Tabelul 8.5.

Tabela 8.5 Task-urile sistemului industrial de monitorizare

Task	C_i	Perioadă	Rol
AcquisitionTask	1 ms	10 ms	Achiziția eșantioanelor
FeatureTask	2 ms	20 ms	Filtrare și extragere de caracteristici
DiagnosisTask	4 ms	100 ms	Detectarea anomaliilor
CommunicationTask	2 ms	100 ms	Transmiterea stării și alarmelor
LoggingTask	3 ms	500 ms	Salvarea datelor agregate

Utilizarea este:

$$\begin{aligned} U &= \frac{1}{10} + \frac{2}{20} + \frac{4}{100} + \frac{2}{100} + \frac{3}{500} \\ &= 0{,}1 + 0{,}1 + 0{,}04 + 0{,}02 + 0{,}006 \\ &= 0{,}266. \end{aligned} \tag{8.46}$$

Rezultă:

$$U = 26{,}6\%. \tag{8.47}$$

Această valoare oferă o marjă pentru întreruperi, comunicații sporadice și tratarea alarmelor, dar nu demonstrează singură respectarea deadline-urilor.

Figura 8.17 prezintă fluxul producer–consumer.

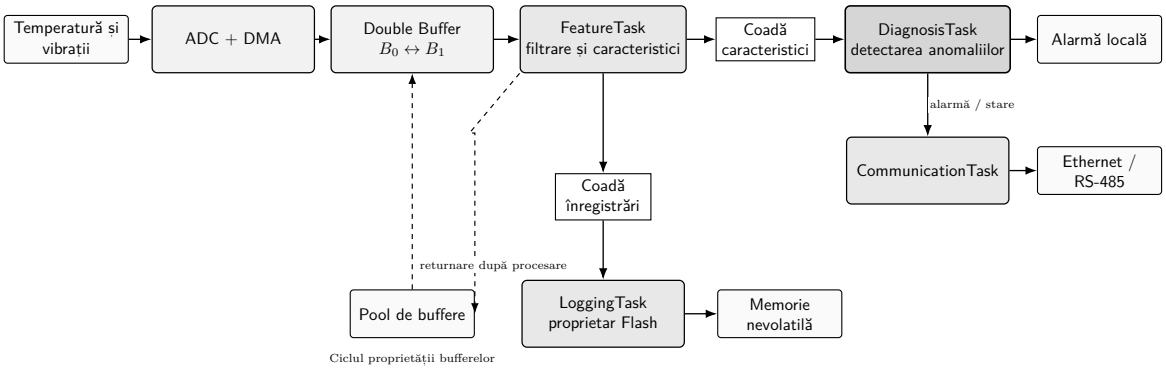


Figura 8.17 Fluxul de procesare al sistemului industrial de monitorizare.

Achiziția poate utiliza DMA cu două buffere:

$$B_0 \leftrightarrow B_1.$$

În timp ce DMA completează un buffer, `FeatureTask` îl procesează pe celălalt.

La finalizarea unui bloc de eșantioane, ISR-ul DMA introduce în coadă un pointer către buffer:

Exemplu 8.17 Transferul unui buffer DMA către task-ul de procesare

```
1  extern "C" void DMA_IRQHandler()
2  {
3      SampleBuffer* completed_buffer =
4          get_completed_dma_buffer();
5
6      const bool task_woken =
7          queue_send_from_isr(
```

```
8               feature_queue,
9               completed_buffer);
10
11      switch_dma_buffer();
12
13      request_context_switch_from_isr(
14          task_woken);
15  }
```

Task-ul de procesare preia proprietatea temporară:

Exemplu 8.18 Procesarea unui buffer primit de la DMA

```
1  void FeatureTask(void* argument)
2  {
3      SampleBuffer* buffer = nullptr;
4
5      while (true)
6      {
7          receive_from_queue(
8              feature_queue,
9              buffer,
10             wait_forever);
11
12         const FeatureVector features =
13             extract_features(*buffer);
14
15         return_buffer_to_pool(buffer);
16
17         send_to_queue(
18             diagnosis_queue,
19             features,
20             diagnosis_timeout);
21     }
22 }
```

După transmiterea pointerului, ISR-ul nu trebuie să reutilizeze bufferul până când acesta este returnat în pool.

Pentru protejarea împotriva supraîncărcării, trebuie monitorizate:

- numărul bufferelor disponibile;
- ocuparea maximă a cozilor;
- numărul mesajelor pierdute;
- timpul maxim de procesare;

- numărul deadline-urilor ratate.

Dacă achiziția produce un buffer la fiecare:

$$T_P = 20\,\text{ms},$$

rata producătorului este:

$$\lambda_P = \frac{1}{T_P} = 50\,\text{buffere/s}. \tag{8.48}$$

Dacă procesarea unui buffer necesită cel mult:

$$C_F = 2\,\text{ms},$$

rata teoretică maximă a consumatorului este:

$$\lambda_C = 500\,\text{buffere/s}.$$

În această situație, procesarea este suficient de rapidă în regimul nominal. Totuși, preempțiile și funcțiile suplimentare trebuie incluse în analiza reală.

Înregistrarea în memoria nevolatilă nu trebuie executată în interiorul unui mutex utilizat și de task-ul de alarmă. O operație Flash poate avea o durată mare și variabilă.

O arhitectură recomandată este:

1. task-urile produc înregistrări;
2. înregistrările sunt introduse într-o coadă;
3. `LoggingTask` este singurul proprietar al memoriei Flash;
4. celelalte task-uri nu accesează direct driverul de stocare.

Această organizare elimină mutex-ul din traseul critic și definește clar proprietatea perifericului.

Alarmele trebuie tratate diferit de telemetria obișnuită. Un eveniment critic poate debloca imediat `CommunicationTask`, fără a aștepta următoarea activare periodică.

Sistemul trebuie să definească politica pentru o coadă de alarmă plină:

- păstrarea celei mai grave alarme;
- eliminarea mesajelor de stare necritice;
- activarea unei ieșiri locale;
- memorarea unui cod minimal de eroare;
- trecerea în mod degradat.

Pentru datele de monitorizare poate fi acceptabilă pierderea unor eșantioane. Pentru o alarmă de protecție, pierderea mesajului poate fi inacceptabilă.

8.6.3 Validarea temporală și testarea sistemului

Analiza matematică trebuie completată prin măsurători pe hardware.

Validarea unui sistem bazat pe RTOS trebuie să urmărească:

- respectarea deadline-urilor;
- latența întreruperilor;
- timpul ISR–task;
- jitter-ul task-urilor periodice;
- utilizarea stivei;
- ocuparea cozilor;
- timpii de blocare;
- comportamentul la suprasarcină;
- recuperarea după defecte.

Figura 8.18 prezintă fluxul recomandat.

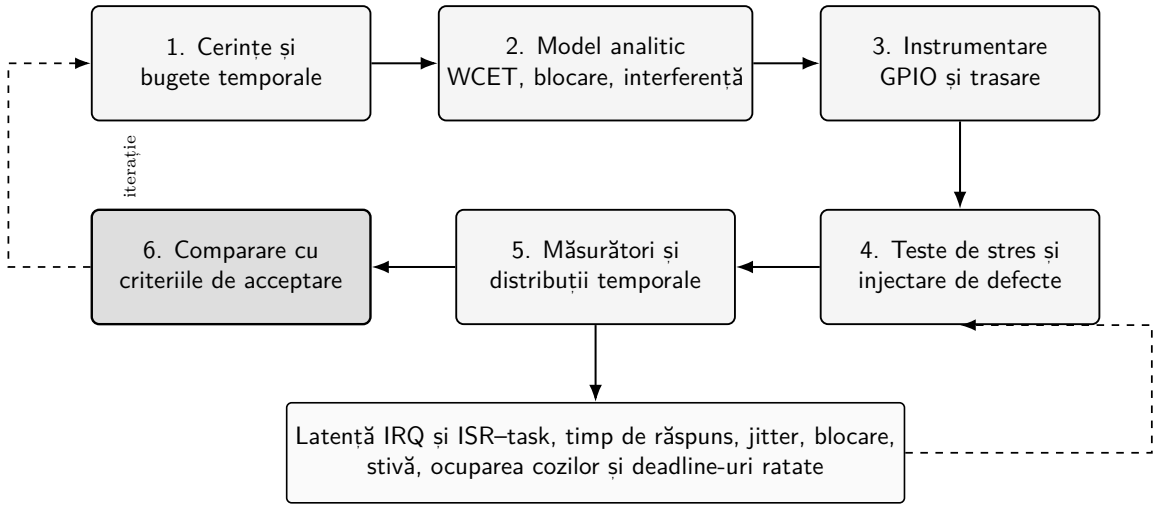


Figura 8.18 Fluxul de validare temporală a unei aplicații bazate pe RTOS.

Pentru măsurarea latenței poate fi utilizat un pin GPIO.

La apariția întreruperii:

Exemplu 8.19 Marcarea începutului unei întreruperi

```
1 extern "C" void Sensor_IRQHandler()
2 {
3     set_debug_pin();
4
5     acknowledge_sensor_interrupt();
6     notify_task_from_isr(sensor_task);
7
8     clear_debug_pin();
```

```
9 }
```

Task-ul poate activa un alt pin sau poate genera un impuls la începerea procesării. Un osciloscop sau analizor logic permite măsurarea:

$$L_{\text{IRQ}}, \qquad C_{\text{ISR}}, \qquad L_{\text{ISR}\to\text{task}}.$$

Pentru analiza software pot fi utilizate evenimente de trasare:

- task switched in;
- task switched out;
- ISR entered;
- ISR exited;
- mutex obtained;
- mutex released;
- queue send;
- queue receive.

Trasarea trebuie configurată astfel încât overhead-ul să nu modifice semnificativ comportamentul măsurat.

Pentru task-ul τ_i se pot colecta:

$$R_{i,\max} = \max_k \left(f_{i,k} - r_{i,k}\right), \tag{8.49}$$

și:

$$J_i = R_{i,\max} - R_{i,\min}. \tag{8.50}$$

Măsurarea nu înlocuiește demonstrația limitei maxime, deoarece testele pot să nu activeze toate cazurile nefavorabile. Ea verifică însă ipotezele și poate evidenția costuri omise.

Scenariile de test trebuie să includă:

- toate comunicațiile active;
- rata maximă a întreruperilor;
- cozi aproape pline;
- acces simultan la resurse;
- temperatura și tensiunea limită;
- frecvența minimă permisă;
- erori și timeout-uri;
- repornirea unor componente;
- suprasarcina controlată.

Testarea prin injectarea defectelor poate simula:

- pierderea unui senzor;
- blocarea unei comunicații;
- expirarea unui mutex;
- coruperea unui mesaj;

- depășirea unei cozi;
- depășirea timpului de execuție;
- resetarea unui periferic.

Un criteriu de acceptare trebuie să fie cantitativ.
Exemplu:

> Într-un test de 24 de ore, cu rata maximă de comunicație și toate mecanismele de diagnosticare active, `MotorControlTask` trebuie să prezinte un timp de răspuns maxim mai mic de 2 ms și niciun deadline ratat.

Trebuie monitorizată și stiva fiecărui task:

$$M_{\text{rezerv},i} = M_{\text{stiv},i} - M_{\text{max utilizat},i}. \tag{8.51}$$

Rezerva trebuie să acopere scenarii rare, apeluri de eroare și modificări viitoare.
Validarea trebuie repetată după:

- schimbarea compilatorului;
- modificarea nivelului de optimizare;
- actualizarea RTOS-ului;
- schimbarea driverelor;
- modificarea frecvenței procesorului;
- introducerea unui nou task;
- modificarea priorităților.

8.7 Întrebări și probleme propuse

8.7.1 Întrebări de verificare

1. Ce proprietăți determină corectitudinea unui sistem de timp real?
2. De ce un sistem de timp real nu este definit numai prin viteza procesorului?
3. Care este diferența dintre un deadline relativ și unul absolut?
4. Ce reprezintă timpul de răspuns al unui task?
5. Care este diferența dintre latență și jitter?
6. Cum se diferențiază sistemele hard, firm și soft real-time?
7. Care este diferența dintre un task periodic și unul sporadic?
8. Care sunt avantajele și limitările unei arhitecturi Super Loop?
9. În ce condiții este potrivit un executiv ciclic?
10. De ce ISR-urile trebuie să fie cât mai scurte?

11. Ce reprezintă procesarea amânată a unei întreruperi?
12. Care este diferența dintre `volatile` și o operație atomică?
13. Ce rol are kernelul unui RTOS?
14. Care este diferența dintre scheduler și dispatcher?
15. Ce informații poate conține un Task Control Block?
16. Care sunt principalele stări ale unui task?
17. Ce evenimente determină trecerea unui task din Blocked în Ready?
18. Ce informații sunt salvate la o comutare de context?
19. Care este diferența dintre concurență și paralelism?
20. Care este diferența dintre planificarea cooperativă și cea preemptivă?
21. Cum funcționează Round Robin?
22. De ce un task de prioritate mare trebuie să se blocheze atunci când nu are activitate?
23. Ce reprezintă gradul de utilizare al procesorului?
24. Care sunt ipotezele modelului clasic Rate Monotonic?
25. De ce limita Liu–Layland este considerată un test suficient?
26. Care este principiul algoritmului Earliest Deadline First?
27. În ce condiții este valabil testul simplu $U \leq 1$ pentru EDF?
28. Ce componente include analiza timpului de răspuns?
29. Ce este o condiție de cursă?
30. Care este diferența dintre un mutex și un semafor binar?
31. Pentru ce este utilizat un semafor de numărare?
32. Care este limitarea unui grup de evenimente pentru numărarea aparițiilor?
33. Care sunt avantajele comunicării prin cozi de mesaje?
34. Ce înseamnă proprietatea unui buffer?
35. Cum se transferă în siguranță procesarea de la ISR la un task?
36. Ce reprezintă starvation?

37. Care este diferența dintre livelock și deadlock?
38. Care sunt cele patru condiții asociate apariției deadlock-ului?
39. Ce este Priority Inversion?
40. Cum funcționează Priority Inheritance?
41. De ce Priority Inheritance nu previne automat deadlock-ul?
42. Cum este determinat plafonul unei resurse în Priority Ceiling?
43. Care este diferența dintre arhitectura cu spațiu unic și microkernel?
44. Ce rol are un MPU?
45. Care este diferența dintre MPU și MMU?
46. Care este diferența dintre AMP și SMP?
47. Ce reprezintă izolarea temporală?
48. Care sunt principalele criterii pentru alegerea unui RTOS?
49. De ce măsurarea timpilor pe hardware nu înlocuiește complet analiza teoretică?
50. Ce trebuie reverificat după modificarea priorităților sau adăugarea unui task?

8.7.2 Probleme de calcul și proiectare

1. Un task este activat la momentul $r = 12\,\text{ms}$ și finalizează la $f = 19\,\text{ms}$. Deadline-ul relativ este $D = 10\,\text{ms}$.

   Calculați deadline-ul absolut, timpul de răspuns și latența față de deadline.

2. Un task periodic trebuie activat la fiecare 20 ms. Momentele reale de activare sunt: 0, 21, 39, 61 și 80 ms.

   Calculați deviația fiecărei activări față de momentele ideale.

3. Un Super Loop conține patru funcții cu duratele: 1, 3, 2 și 10 ms.

   Calculați durata buclei și latența maximă aproximativă pentru un eveniment verificat o singură dată pe iterație.

4. O întrerupere are următoarele componente:

$$T_{\text{mascare}} = 4\,\mu\text{s},$$

$$T_{\text{instrucțiune}} = 1\,\mu\text{s},$$

$$T_{\text{priorități}} = 3\,\mu\text{s},$$

și:

$$T_{\text{intrare}} = 2\,\mu\text{s}.$$

Calculați latența totală.

5. Un RTOS utilizează un tick de 500 Hz. Calculați perioada tick-ului și discutați rezoluția unui timeout de trei tick-uri.

6. Un sistem realizează 4000 de comutări de context pe secundă. Fiecare comutare necesită $3\,\mu$s.

   Calculați procentul de procesor utilizat pentru comutări.

7. Se consideră task-urile:

$$\tau_1 = (1, 5, 5), \qquad \tau_2 = (2, 10, 10), \qquad \tau_3 = (5, 50, 50).$$

   Calculați utilizarea totală și verificați condiția Liu–Layland.

8. Se consideră task-urile:

$$\tau_1 = (2, 5, 5), \qquad \tau_2 = (2, 8, 8), \qquad \tau_3 = (3, 20, 20).$$

   Calculați utilizarea. Dacă limita Liu–Layland nu este satisfăcută, explicați dacă se poate concluziona că sistemul este neschedulabil.

9. Pentru task-urile:

$$\tau_1 = (1, 4, 4), \qquad \tau_2 = (2, 6, 6), \qquad \tau_3 = (3, 15, 15),$$

   calculați timpul de răspuns al fiecărui task prin iterație.

10. Un task are:

$$C_i = 3\,\text{ms},$$

$$B_i = 2\,\text{ms},$$

    și este interferat de un task superior cu:

$$C_h = 1\,\text{ms}, \qquad T_h = 5\,\text{ms}.$$

Calculați timpul de răspuns iterativ pentru deadline-ul $D_i = 12\,\text{ms}$.

11. Două task-uri incrementează aceeași variabilă fără protecție. Descrieți o ordine a execuției care conduce la pierderea unei actualizări.

12. Un producător generează 100 de mesaje pe secundă, iar consumatorul poate procesa 120 de mesaje pe secundă. Într-o rafală pot fi produse suplimentar 15 mesaje.

    Propuneți capacitatea minimă a cozii și justificați alegerea.

13. Un pool conține șase buffere. Trei buffere sunt utilizate de DMA, două sunt procesate, iar unul este în coada de transmitere.

    Calculați valoarea semaforului care reprezintă numărul bufferelor libere.

14. Task-ul de prioritate redusă deține un mutex timp de 3 ms. Un task de prioritate medie execută timp de 12 ms, iar task-ul de prioritate mare așteaptă mutex-ul.

    Calculați blocarea task-ului de prioritate mare fără Priority Inheritance și cu Priority Inheritance, într-un model simplificat.

15. Trei mutex-uri trebuie obținute de mai multe task-uri. Propuneți o ordine globală care previne așteptarea circulară și descrieți regula de eliberare.

16. O resursă este utilizată de task-uri cu priorităților 2, 5 și 7. Determinați plafonul resursei pentru un protocol Priority Ceiling.

17. Un sistem conține patru task-uri cu stive de: 512, 768, 1024 și 1536 bytes. Kernelul utilizează 8 KB, iar bufferele utilizează 12 KB.

    Calculați memoria RAM totală, fără heap și alte structuri.

18. Un task are o stivă de 2048 bytes, iar valoarea maximă măsurată este 1536 bytes.

    Calculați rezerva și procentul de stivă rămas.

19. Proiectați arhitectura software a unui controler de dronă care execută:

    - controlul atitudinii la 1 kHz;
    - estimarea poziției la 100 Hz;
    - navigația la 20 Hz;
    - telemetria la 10 Hz;
    - logarea la 2 Hz.

    Propuneți task-urile, prioritățile și mecanismele de comunicație.

20. Proiectați un sistem de monitorizare în care achiziția este realizată prin DMA, procesarea printr-un task, iar transmiterea prin Ethernet. Definiți proprietatea bufferelor și comportamentul la umplerea cozii.

21. Comparați utilizarea unui mutex și a unei cozi pentru transferul unei structuri de stare între un task de achiziție și unul de diagnosticare.

22. Propuneți criterii de alegere între FreeRTOS, Zephyr și Linux cu PREEMPT_RT pentru:

    - un senzor alimentat din baterie;
    - un gateway industrial;
    - un robot cu interfață grafică și control în timp real.

23. Definiți un plan experimental pentru măsurarea latenței ISR–task și a jitter-ului unui task periodic.

24. Propuneți cinci scenarii de injectare a defectelor pentru validarea unui sistem industrial bazat pe RTOS.

25. Explicați ce analize trebuie repetate după introducerea unui nou task periodic într-o aplicație existentă.

9. Sisteme Reconfigurabile și Accelerare Hardware pentru Aplicații Embedded

Introducere în calculul reconfigurabil
Motivația accelerării hardware
Compararea procesoarelor, FPGA-urilor și circuitelor ASIC
Arhitectura internă a unui FPGA
LUT-uri, registre și implementarea funcțiilor logice
Resurse hardware specializate
Fluxul de proiectare FPGA
Sisteme heterogene CPU–FPGA
Partiționarea hardware–software
Performanță, latență și utilizarea resurselor
Configurarea și securitatea sistemelor reconfigurabile
Studiu de caz: accelerarea procesării imaginilor
Întrebări și probleme propuse

9.1 Introducere în calculul reconfigurabil

Sistemele embedded moderne trebuie să proceseze volume tot mai mari de date, respectând simultan constrângeri de latență, consum energetic, cost și dimensiune. În aplicații precum procesarea imaginilor, comunicațiile digitale, robotica sau inteligența artificială, optimizarea exclusiv software poate deveni insuficientă.

Creșterea performanței unui procesor nu poate fi obținută nelimitat prin majorarea frecvenței de ceas. Consumul dinamic al unui circuit digital poate fi aproximat prin:

$$P_{\text{dinamic}} = \alpha C_L V^2 f, \tag{9.1}$$

unde α este factorul de activitate, C_L este capacitatea comutată, V este tensiunea de alimentare, iar f este frecvența de funcționare. Creșterea frecvenței mărește puterea consumată, iar reducerea tensiunii este limitată de cerințele de funcționare ale tranzistoarelor. Din acest motiv, arhitecturile moderne urmăresc nu doar executarea mai rapidă a acelorași instrucțiuni, ci utilizarea unor structuri de calcul specializate [38].

Calculul reconfigurabil oferă o soluție intermediară între execuția software pe un procesor general și implementarea unei funcții într-un circuit dedicat — funcționalitatea hardware poate fi modificată după fabricarea dispozitivului, prin încărcarea unei noi configurații.

În practică, principalele platforme utilizate pentru calcul reconfigurabil sunt circuitele FPGA, denumire provenită de la *Field-Programmable Gate Array*. Acestea conțin blocuri logice, registre, memorii, resurse aritmetice și conexiuni programabile care pot fi configurate pentru a forma un circuit digital specific aplicației [37].

9.1.1 De la execuția software la implementarea hardware

Un procesor general execută succesiv instrucțiuni stocate în memorie. Funcționalitatea sistemului este modificată prin schimbarea programului, fără modificarea structurii interne a procesorului.

Modelul simplificat al execuției este:

$$\text{preluare} \rightarrow \text{decodificare} \rightarrow \text{execuție} \rightarrow \text{memorare}.$$

Același set de unități de calcul este reutilizat pentru numeroase operații. Această caracteristică oferă flexibilitate ridicată, dar poate introduce overhead asociat cu preluarea și decodificarea instrucțiunilor, transferul repetat al datelor, controlul execuției și utilizarea unei arhitecturi generale pentru operații specializate.

Într-o implementare hardware, operațiile algoritmului sunt transformate în structuri digitale dedicate. În loc ca aceeași unitate aritmetică să fie reutilizată succesiv, pot fi construite mai multe unități care funcționează simultan.

De exemplu, expresia:

$$y = a \cdot b + c \cdot d \tag{9.2}$$

poate fi executată software prin două înmulțiri succesive și o adunare. Într-o implementare hardware pot fi utilizate două multiplicatoare paralele, urmate de un sumator.

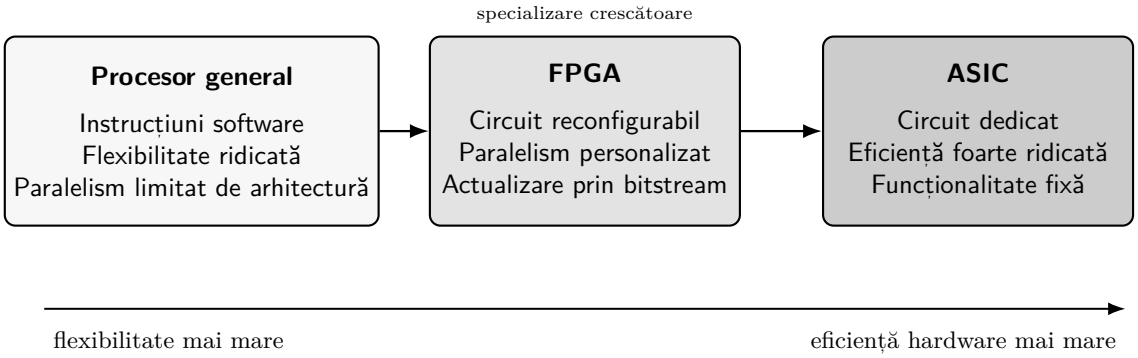
Figura 9.1 prezintă poziționarea principalelor soluții de calcul.

Procesorul general oferă flexibilitatea cea mai ridicată, circuitul ASIC oferă eficiență maximă pentru funcția pentru care a fost proiectat, iar FPGA-ul ocupă o poziție intermediară, combinând posibilitatea reconfigurării cu execuția hardware.

Este importantă diferența dintre programarea unui procesor și configurarea unui FPGA: un program descrie o succesiune de instrucțiuni, în timp ce o descriere hardware definește structura și relațiile temporale ale unui circuit.

9.1.2 Paralelismul spațial

Avantajul principal al logicii reconfigurabile este posibilitatea exploatării paralelismului spațial. Considerăm N canale independente, fiecare necesitând un timp de procesare T_C. Într-o implementare secvențială simplificată:



FPGA-ul ocupă poziția intermediară dintre execuția software și hardware-ul dedicat

Figura 9.1 Spectrul soluțiilor de calcul, de la procesor general la circuit dedicat.

$$T_{\text{secvențial}} \approx NT_C. \tag{9.3}$$

Dacă sunt implementate N unități hardware care funcționează simultan:

$$T_{\text{paralel}} \approx T_C, \tag{9.4}$$

la care se adaugă latențele de intrare, sincronizare și colectare a rezultatelor.

Pentru opt canale, fiecare necesitând $10\,\mu$s:

$$T_{\text{secvențial}} \approx 8 \cdot 10\,\mu\text{s} = 80\,\mu\text{s}.$$

Într-o arhitectură cu opt unități paralele, timpul de procesare poate rămâne apropiat de:

$$T_{\text{paralel}} \approx 10\,\mu\text{s}.$$

Această reducere nu este automată — ea necesită replicarea resurselor hardware, suficiente porturi de memorie, lățime de bandă adecvată, absența dependențelor între date și o arhitectură de control corespunzătoare. Paralelizarea crește de regulă utilizarea LUT-urilor, registrelor, blocurilor DSP și memoriei, iar proiectarea FPGA reprezintă astfel un compromis între performanță și resurse.

O altă formă importantă de paralelism este pipeline-ul. Un algoritm este împărțit în mai multe etape, iar fiecare etapă procesează simultan un element diferit.

După umplerea pipeline-ului, poate fi produs un rezultat la fiecare interval de inițiere:

$$II = \text{numărul de cicluri dintre două intrări succesive.} \tag{9.5}$$

Latența totală poate fi de mai multe cicluri, chiar dacă intervalul de inițiere este egal cu un singur ciclu.

9.1.3 Domenii de aplicare

Calculul reconfigurabil este adecvat în special algoritmilor care prezintă paralelism între date, operații repetitive, fluxuri continue de date, cerințe stricte de latență, aritmetică regulată și posibilitatea utilizării unui pipeline. Domeniile reprezentative includ:

- procesarea imaginilor și a semnalelor;
- comunicațiile digitale;
- criptografia;
- controlul industrial;
- robotică;
- accelerarea inferenței rețelelor neuronale;
- procesarea pachetelor de rețea;
- instrumentația științifică.

Un FPGA nu înlocuiește obligatoriu procesorul principal. În numeroase sisteme, procesorul execută funcțiile de control, comunicație și configurare, iar FPGA-ul accelerează numai operațiile intensive.

Arhitectura rezultată este eterogenă:

$$\text{CPU} + \text{logic reconfigurabil} + \text{memorie} + \text{periferice}.$$

Această împărțire permite menținerea funcțiilor cu evoluție frecventă în software și implementarea operațiilor stabile și costisitoare în hardware.

9.2 Motivația accelerării hardware

Accelerarea hardware nu trebuie aplicată întregii aplicații. În cele mai multe cazuri, numai o parte redusă a codului consumă o proporție semnificativă din timpul total de execuție.

Aceste porțiuni sunt denumite frecvent:

- zone critice;
- nuclee de calcul;
- hotspot-uri;
- funcții dominante.

Procesul corect începe prin profilarea aplicației, nu prin alegerea anticipată a unui accelerator.

9.2.1 Limitările execuției exclusiv software

Execuția software este avantajoasă pentru algoritmi cu:

- ramificații numeroase;
- structuri de date dinamice;
- control complex;
- modificări frecvente;
- volum redus de calcul.

Ea poate deveni ineficientă pentru operații regulate aplicate unor cantități mari de date.

Exemplele includ:

- convoluții asupra imaginilor;
- filtre FIR;
- codare și decodare;
- multiplicări de matrice;
- criptare în flux;
- prelucrarea mai multor canale independente.

Timpul total al unei aplicații poate fi exprimat prin:

$$T_{\text{aplicație}} = T_{\text{control}} + T_{\text{calcul}} + T_{\text{memorie}} + T_{\text{I/O}}. \tag{9.6}$$

Accelerarea componentei T_{calcul} poate avea un efect redus dacă aplicația este dominată de transferurile de memorie sau de intrare–ieșire. În consecință, trebuie analizat întregul traseu al datelor, nu numai funcția matematică.

9.2.2 Identificarea zonelor critice

Profilarea măsoară contribuția fiecărei funcții la timpul total de execuție. Pentru funcția F_i, proporția timpului este:

$$p_i = \frac{T_i}{T_{\text{total}}}. \tag{9.7}$$

O funcție este un candidat bun pentru accelerare dacă are o contribuție mare la timpul total, este executată frecvent, operează asupra unor date regulate, poate fi paralelizată, transferul datelor nu anulează câștigul, iar implementarea hardware este stabilă.

Figura 9.2 prezintă procesul de identificare a unei funcții candidate.

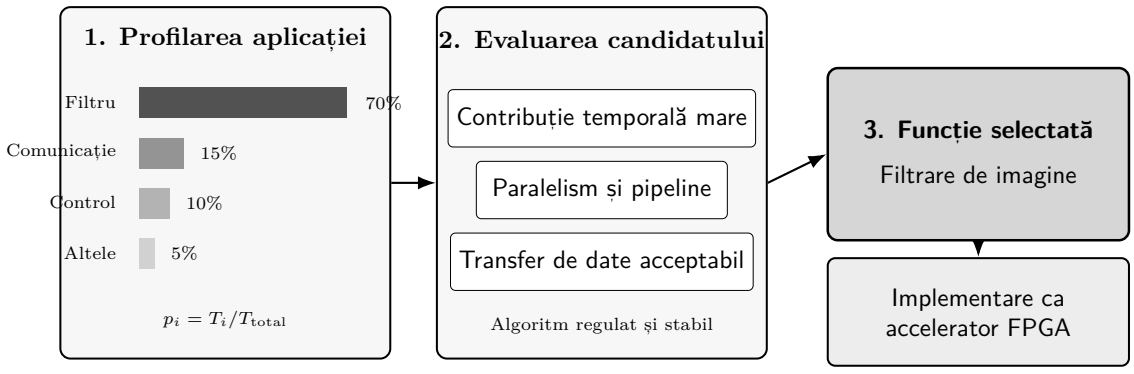


Figura 9.2 Identificarea zonelor critice care pot fi migrate în hardware.

Să presupunem că o aplicație necesită 100 ms, iar filtrarea imaginilor consumă 70 ms. Funcția de filtrare reprezintă:

$$p_{\text{filtru}} = \frac{70}{100} = 0{,}7.$$

Accelerarea filtrului poate îmbunătăți semnificativ aplicația. În schimb, accelerarea unei funcții care consumă numai 2% din timp va avea un efect redus, indiferent de performanța acceleratorului.

Un candidat aparent bun poate fi respins dacă necesită transferuri mari între CPU și FPGA. Timpul efectiv al acceleratorului este:

$$T_{\text{accelerat}} = T_{\text{trimitere}} + T_{\text{configurare}} + T_{\text{execuție}} + T_{\text{primire}}. \tag{9.8}$$

Pentru operații foarte scurte, costul comunicării poate fi mai mare decât timpul economisit prin execuția hardware.

9.2.3 Performanță, energie și flexibilitate

Accelerarea hardware urmărește frecvent trei obiective: reducerea latenței, creșterea throughput-ului și reducerea energiei necesare pentru o operație. Latența reprezintă timpul dintre furnizarea intrării și obținerea rezultatului:

$$L = t_{\text{ieșire}} - t_{\text{intrare}}. \tag{9.9}$$

Throughput-ul reprezintă numărul de rezultate produse într-o unitate de timp:

$$\Theta = \frac{N_{\text{rezultate}}}{T}. \tag{9.10}$$

O arhitectură pipeline poate avea o latență de zece cicluri și poate produce, după umplere, câte un rezultat în fiecare ciclu — prin urmare, latența și throughput-ul trebuie evaluate separat.

Energia consumată pentru o operație este:

$$E_{\text{operație}} = P_{\text{medie}} T_{\text{operație}}. \tag{9.11}$$

Un FPGA poate consuma o putere instantanee comparabilă sau chiar mai mare decât un procesor mic, dar poate finaliza operația într-un timp mai redus — indicatorul relevant este energia pentru sarcina completă, nu numai puterea instantanee.

Flexibilitatea FPGA-ului provine din posibilitatea încărcării unei noi configurații. Această flexibilitate are însă limite: modificarea necesită un nou flux de sinteză și implementare, timpul de compilare hardware este mai mare decât compilarea software, verificarea temporală trebuie repetată, configurația trebuie distribuită și protejată, iar resursele fizice rămân limitate.

9.3 Compararea procesoarelor, FPGA-urilor și circuitelor ASIC

Alegerea platformei de calcul influențează performanța, consumul, costul și durata dezvoltării. Cele trei categorii principale analizate sunt procesorul general, FPGA-ul și circuitul ASIC. Ele nu trebuie privite numai ca soluții concurente — un produs poate utiliza simultan un procesor, un FPGA și mai multe blocuri ASIC integrate.

9.3.1 Procesor general și accelerator specializat

Procesorul general este adecvat pentru control și decizie, protocoale, sisteme de operare, interfețe grafice, algoritmi cu ramificații complexe și funcționalități actualizate frecvent. Avantajele sale sunt programarea relativ simplă, ecosistemul software matur, depanarea facilă, reutilizarea bibliotecilor și flexibilitatea ridicată.

Acceleratorul hardware este potrivit pentru operații regulate și repetitive. Acesta poate funcționa ca un periferic comandat de procesor:

$$\text{CPU} \rightarrow \text{configurare} \rightarrow \text{accelerator} \rightarrow \text{rezultat}.$$

Performanța sistemului depinde de colaborarea dintre cele două componente — un accelerator foarte rapid poate rămâne inactiv dacă procesorul sau memoria nu furnizează datele cu rata necesară.

9.3.2 ASIC și FPGA

ASIC provine din *Application-Specific Integrated Circuit*. Circuitul este proiectat și fabricat pentru o funcționalitate sau o clasă restrânsă de funcționalități. Avantajele principale sunt performanța ridicată, consumul energetic redus, suprafața optimizată și costul unitar redus la volume mari. Limitările sunt costul inițial foarte ridicat, timpul mare de dezvoltare, riscul asociat erorilor de proiectare și imposibilitatea modificării după fabricație.

FPGA-ul este fabricat ca o structură programabilă generală, funcționalitatea fiind stabilită după fabricație. Avantajele sunt reconfigurarea, costul inițial redus, prototiparea rapidă, actualizarea funcționalității și paralelismul hardware. În raport cu un ASIC echivalent, FPGA-ul utilizează resurse programabile suplimentare pentru LUT-uri, rutare și memorie de configurare, ceea ce conduce în general la o eficiență mai redusă în privința suprafeței, vitezei și puterii [51].

Tabela 9.1 Compararea procesorului general, FPGA-ului și ASIC-ului

Caracteristică	Procesor	FPGA	ASIC
Flexibilitate	Foarte ridicată	Ridicată	Foarte redusă
Paralelism personalizat	Limitat de arhitectură	Ridicat	Foarte ridicat
Eficiență energetică	Moderată	Ridicată pentru sarcini adecvate	Foarte ridicată
Cost inițial	Redus	Redus sau moderat	Foarte ridicat
Cost unitar	Depinde de platformă	Relativ ridicat	Redus la volum mare
Timp de dezvoltare	Redus	Moderat	Ridicat
Actualizare	Prin software	Prin software și bitstream	În general imposibilă

Valorile din tabel sunt orientative — rezultatul concret depinde de tehnologia de fabricație, familia dispozitivului, algoritm și implementare.

9.3.3 Alegerea soluției

Selecția trebuie realizată pe baza cerințelor tehnice și economice. Costul total al unei soluții poate fi exprimat simplificat prin:

$$C_{\text{total}} = C_{\text{NRE}} + NC_{\text{unitar}}, \tag{9.12}$$

unde C_{NRE} reprezintă costurile nerecurente de dezvoltare, N este numărul de unități, iar C_{unitar} este costul unei unități. ASIC-ul are de regulă un cost nerecurent mare și un cost unitar redus, iar FPGA-ul are un cost nerecurent mai mic și un cost unitar mai mare.

Pragul aproximativ la care cele două variante au același cost este obținut din:

$$C_{\text{NRE,ASIC}} + N^{\star}C_{\text{ASIC}} = C_{\text{NRE,FPGA}} + N^{\star}C_{\text{FPGA}}. \tag{9.13}$$

Rezultă:

$$N^{\star} = \frac{C_{\text{NRE,ASIC}} - C_{\text{NRE,FPGA}}}{C_{\text{FPGA}} - C_{\text{ASIC}}}. \tag{9.14}$$

Figura 9.3 prezintă conceptual această intersecție.

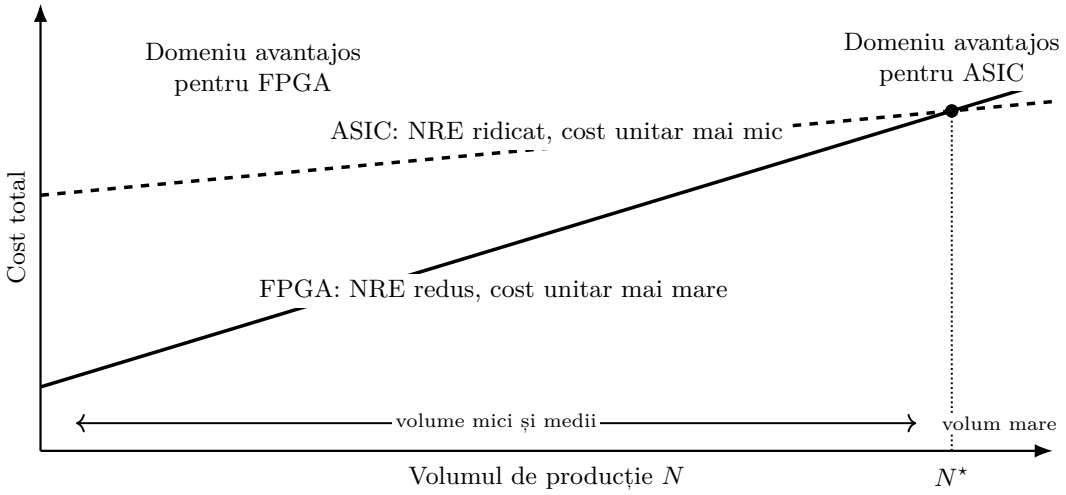


Figura 9.3 Evoluția conceptuală a costului total în funcție de volumul de producție.

Criteriile principale de selecție sunt volumul de producție, performanța și latența necesare, consumul energetic, durata de viață a produsului, probabilitatea modificării algoritmului, costul și riscul dezvoltării, cerințele de securitate și disponibilitatea competențelor. O strategie întâlnită în practică este:

prototip pe FPGA → validare → ASIC la volum ridicat.

Migrarea nu este însă automată — descrierea hardware, memoriile, blocurile DSP și interfețele specifice FPGA-ului trebuie adaptate la tehnologia ASIC. Pentru produse cu volume reduse, cerințe în schimbare sau durată mare de viață, FPGA-ul poate rămâne soluția finală. Pentru produse cu volum foarte mare și funcționalitate stabilă, investiția într-un ASIC poate deveni justificată.

9.4 Arhitectura internă a unui FPGA

Un FPGA modern poate fi privit ca o matrice de resurse hardware configurabile, conectate printr-o rețea programabilă. Configurația stabilește atât funcțiile implementate în blocurile logice, cât și traseele semnalelor dintre acestea.

Principalele categorii de resurse sunt:

- blocuri logice configurabile;
- registre;
- interconectări programabile;
- blocuri de intrare–ieșire;
- memorii interne;
- blocuri aritmetice specializate;
- resurse pentru distribuția ceasului.

Arhitectura exactă diferă între familiile de dispozitive, însă principiile generale sunt similare. Proiectantul descrie comportamentul dorit, iar instrumentele de sinteză și implementare aleg resursele și conexiunile necesare [20, 56].

9.4.1 Blocuri logice, rutare și intrări–ieșiri

Blocul logic configurabil reprezintă unitatea de bază a FPGA-ului. Acesta include de regulă:

- unul sau mai multe LUT-uri;
- registre de tip flip-flop;
- multiplexoare;
- logică rapidă pentru transportul biților;
- conexiuni către rețeaua de rutare.

Mai multe blocuri logice sunt grupate în structuri de nivel superior, denumite diferit în funcție de producător și familie. Din acest motiv, valorile raportate ca *logic element*, *logic cell* sau *slice* nu trebuie comparate direct fără analizarea definiției utilizate.

Figura 9.4 prezintă structura conceptuală a unui FPGA.

Rețeaua programabilă de interconectare permite conectarea blocurilor logice, memoriilor, unităților DSP și pinilor externi. Aceasta conține segmente de rutare locale, conexiuni între regiuni, magistrale dedicate, comutatoare programabile și rețele speciale pentru semnalele de ceas. Întârzierea totală a unei căi poate fi aproximată prin:

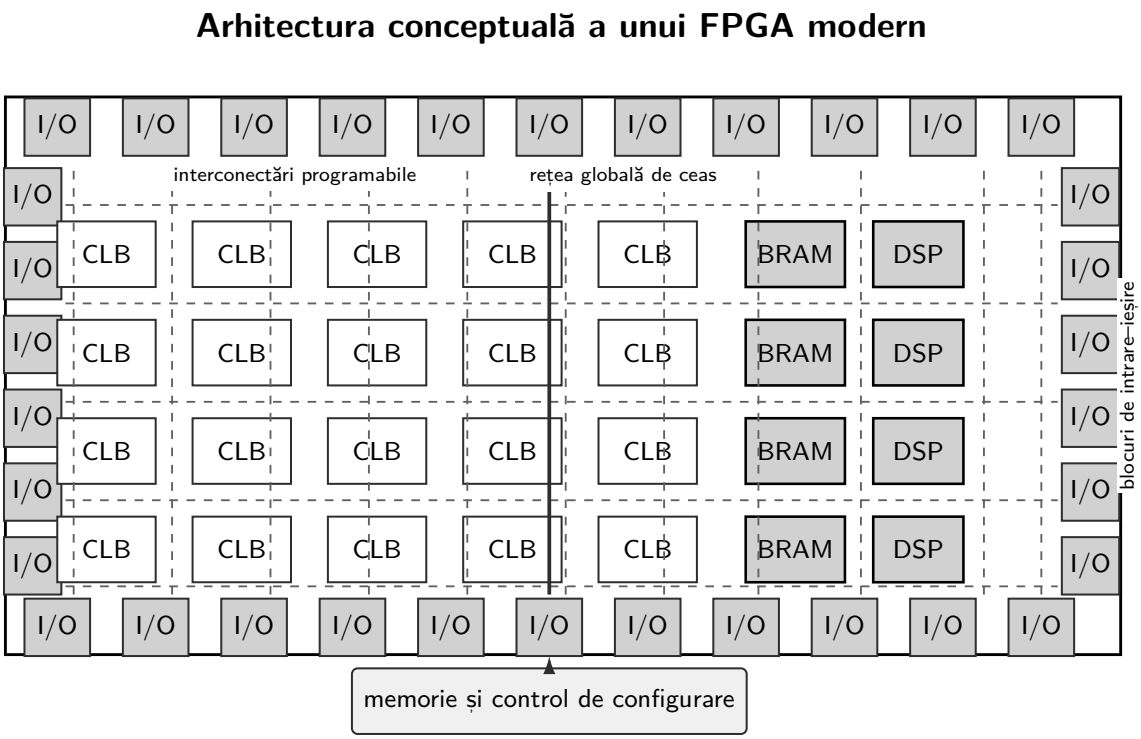


Figura 9.4 Structura conceptuală a unui FPGA modern.

$$T_{\text{cale}} = T_{\text{logic}} + T_{\text{rutare}}. \tag{9.15}$$

În multe implementări, întârzierea rutării este comparabilă cu sau chiar mai mare decât întârzierea logicii, motiv pentru care două descrieri funcțional echivalente pot conduce la frecvențe maxime diferite după plasare și rutare.

Frecvența maximă este limitată de cea mai lentă cale sincronă:

$$f_{\max} \leq \frac{1}{T_{\text{clk}\to\text{q}} + T_{\text{logic}} + T_{\text{rutare}} + T_{\text{setup}}}. \tag{9.16}$$

Blocurile de intrare–ieșire conectează logica internă la pinii circuitului. Acestea permit configurarea direcției semnalului, standardului electric, tensiunii, rezistențelor interne, vitezei de comutare, registrelor de intrare și ieșire și semnalelor diferențiale. În aplicațiile de mare viteză, plasarea registrelor aproape de pini reduce întârzierile și îmbunătățește stabilitatea temporală.

9.4.2 Memoria de configurare

Configurația FPGA-ului este descrisă printr-un fișier binar denumit frecvent *bitstream*. Acesta stabilește conținutul LUT-urilor, starea comutatoarelor de rutare, configurația blocurilor I/O, modurile de funcționare ale memoriilor și unităților DSP, precum și opțiunile resurselor de ceas. În dispozitivele bazate pe SRAM, configurația este volatilă — la pornire, bitstream-ul trebuie încărcat dintr-o memorie externă sau de către un procesor.

Fluxul simplificat este:

$$\text{reset} \rightarrow \text{ncrcare bitstream} \rightarrow \text{verificare} \rightarrow \text{activare}.$$

Există și FPGA-uri bazate pe tehnologii nevolatile, precum Flash sau antifuse. Acestea pot păstra configurația după întreruperea alimentării, dar prezintă caracteristici diferite privind reprogramarea, viteza și securitatea.

Timpul de configurare depinde de dimensiunea bitstream-ului și de rata interfeței de încărcare:

$$T_{\text{config}} \approx \frac{N_{\text{bitstream}}}{R_{\text{config}}}. \tag{9.17}$$

De exemplu, pentru un bitstream de 32 Mbit încărcat cu 100 Mbit/s:

$$T_{\text{config}} \approx \frac{32}{100} = 0{,}32\,\text{s}.$$

În practică se adaugă timpii de inițializare, verificare și pornire a resurselor interne.

Unele FPGA-uri permit reconfigurarea parțială — o regiune a dispozitivului poate fi modificată fără oprirea completă a logicii aflate în celelalte regiuni.

Conceptual:

$$\text{regiune static} + \text{regiune reconfigurabil}.$$

Această metodă poate fi utilizată pentru înlocuirea dinamică a unui accelerator, reducerea numărului total de resurse necesare, actualizarea unei funcții fără oprirea întregului sistem sau izolarea unor componente. Reconfigurarea parțială crește însă complexitatea proiectării, verificării și gestionării interfețelor dintre regiuni.

9.4.3 Structura unei celule logice

O celulă logică simplificată conține un LUT, un registru și un multiplexor care selectează ieșirea combinațională sau cea înregistrată.

Figura 9.5 prezintă această structură.

Ieșirea poate fi exprimată conceptual prin:

$$y = \begin{cases} F(x_0, \ldots, x_{k-1}), & \text{ieșire combinațională}, \\ Q, & \text{ieșire înregistrată}. \end{cases} \tag{9.18}$$

Registrul permite introducerea etapelor pipeline și realizarea circuitelor secvențiale. În funcție de arhitectură, acesta poate avea enable, reset sau set, polaritate configurabilă și selecția sursei de date. Blocurile logice includ frecvent și căi rapide de carry pentru implementarea eficientă a sumatoarelor, scăzătoarelor, contoarelor, comparatoarelor și acumulatoarelor. Fără aceste căi dedicate, propagarea transportului prin LUT-uri și rutare generală

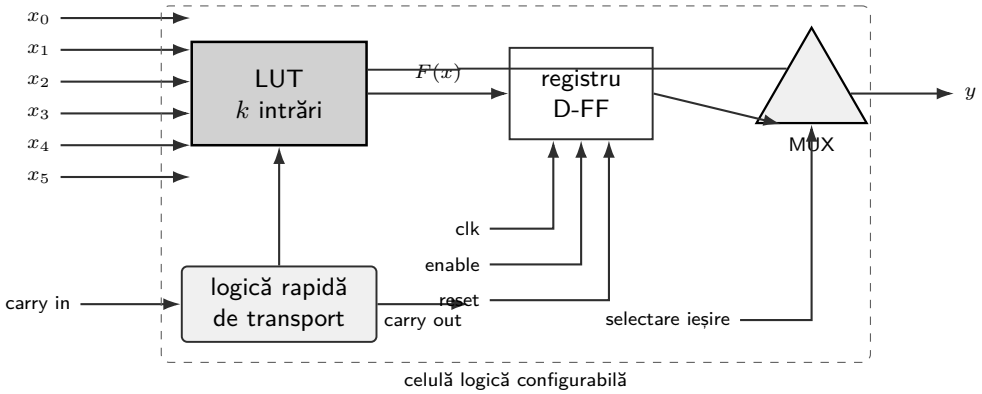


Figura 9.5 Structura simplificată a unei celule logice FPGA.

ar limita semnificativ frecvența maximă.

9.5 LUT-uri, registre și implementarea funcțiilor logice

Look-Up Table-ul este elementul principal utilizat pentru implementarea logicii combinaționale. Acesta poate fi interpretat ca o memorie de dimensiune redusă, adresată de semnalele de intrare.

Pentru un LUT cu k intrări sunt necesare:

$$N_{\text{biți}} = 2^k \tag{9.19}$$

valori de configurare.

Un LUT cu patru intrări conține 16 valori, iar un LUT cu șase intrări conține 64 de valori.

9.5.1 Principiul LUT

Considerăm funcția XOR cu două intrări:

$$F = A \oplus B. \tag{9.20}$$

Tabelul de adevăr este:

Tabela 9.2 Tabelul de adevăr al funcției XOR

A	B	F
0	0	0
0	1	1
1	0	1
1	1	0

Intrările formează adresa memoriei:

$$\text{adres} = (A, B),$$

iar ieșirea reprezintă valoarea memorată la adresa selectată.
Figura 9.6 prezintă principiul unui LUT cu patru intrări.

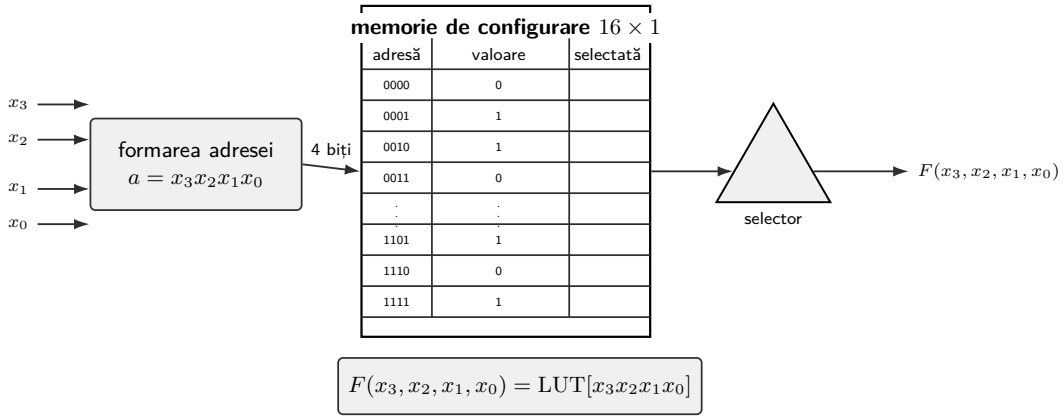


Figura 9.6 Principiul implementării unei funcții logice prin LUT.

Orice funcție booleană cu cel mult k intrări poate fi implementată direct într-un LUT cu k intrări. Dacă funcția are mai multe intrări, instrumentul de sinteză o descompune în mai multe LUT-uri conectate între ele. Această descompunere poate crește numărul de niveluri logice, întârzierea, utilizarea rutării și consumul energetic. Din acest motiv, numărul de intrări al LUT-ului influențează atât flexibilitatea, cât și eficiența arhitecturii.

9.5.2 Logică combinațională și secvențială

O descriere combinațională produce o ieșire dependentă numai de intrările curente:

$$y(t) = F\left(x_0(t), x_1(t), \ldots\right). \tag{9.21}$$

O descriere secvențială include starea anterioară:

$$q[n+1] = F\left(q[n], x[n]\right). \tag{9.22}$$

Registrele asociate LUT-urilor permit implementarea mașinilor de stare, contoarelor, registrelor pipeline, temporizatoarelor și registrelor de deplasare. Un circuit secvențial sincron poate fi descris astfel:

Exemplu 9.1 Registru sincron descris în VHDL

```
1 process(clk)
2 begin
3     if rising_edge(clk) then
4         if reset = '1' then
```

```
            q <= (others => '0');
        elsif enable = '1' then
            q <= d;
        end if;
    end if;
end process;
```

Instrumentul de sinteză mapează variabila `q` pe registrele fizice ale FPGA-ului.

Introducerea registrelor pipeline reduce lungimea căilor combinaționale. Considerăm un calcul format din trei operații cu întârzierile:

$$T_1, \quad T_2, \quad T_3.$$

Fără pipeline:

$$T_{\text{comb}} = T_1 + T_2 + T_3. \tag{9.23}$$

Cu registre între etape, perioada de ceas este limitată aproximativ de:

$$T_{\text{clk}} \geq \max(T_1, T_2, T_3) + T_{\text{registre}}. \tag{9.24}$$

Latența exprimată în cicluri crește, dar throughput-ul poate crește semnificativ.

9.5.3 Exemplu de implementare

Considerăm un multiplexor 4:1 cu intrările d_0, d_1, d_2, d_3 și semnalul de selecție s.

Funcția este:

$$y = \begin{cases} d_0, & s = 00, \\ d_1, & s = 01, \\ d_2, & s = 10, \\ d_3, & s = 11. \end{cases} \tag{9.25}$$

Descrierea VHDL poate fi:

Exemplu 9.2 Multiplexor 4:1 descris în VHDL

```
with sel select
    y <= d0 when "00",
         d1 when "01",
         d2 when "10",
         d3 when others;
```

Funcția are șase intrări logice:

$$d_0, d_1, d_2, d_3, s_0, s_1.$$

Pe o arhitectură cu LUT-uri de șase intrări, multiplexorul poate fi implementat conceptual într-un singur LUT — implementarea concretă depinde însă de regulile de mapare și de resursele arhitecturii.

Dacă ieșirea trebuie sincronizată, aceasta poate fi înregistrată:

Exemplu 9.3 Multiplexor cu ieșire înregistrată

```
process(clk)
begin
    if rising_edge(clk) then
        case sel is
            when "00"   => y_reg <= d0;
            when "01"   => y_reg <= d1;
            when "10"   => y_reg <= d2;
            when others => y_reg <= d3;
        end case;
    end if;
end process;
```

Această variantă introduce o latență de un ciclu, dar oferă o ieșire sincronă și poate facilita închiderea temporală. Unele LUT-uri pot fi configurate și ca memorie distribuită, registru de deplasare sau memorie ROM; utilizarea acestor moduri poate reduce consumul de registre și poate evita utilizarea unui bloc RAM pentru structuri foarte mici.

9.6 Resurse hardware specializate

Implementarea tuturor funcțiilor exclusiv prin LUT-uri și registre ar conduce la un consum ridicat de resurse. FPGA-urile moderne includ blocuri dedicate pentru operații întâlnite frecvent. Aceste resurse oferă performanță mai mare, consum energetic mai redus, ocuparea unei suprafețe logice mai mici și rutare simplificată.

9.6.1 Block RAM și memorii distribuite

Memoria distribuită utilizează LUT-urile ca elemente de stocare. Este adecvată pentru tabele mici, buffere scurte, registre de deplasare și memorii cu dimensiuni neregulate.

Block RAM, prescurtat BRAM, este o memorie fizică dedicată, integrată în FPGA. Aceasta poate fi configurată cu diferite:

- adâncimi;
- lățimi de cuvânt;
- moduri de citire;
- moduri single-port sau dual-port;
- mecanisme de corecție a erorilor.

Figura 9.7 compară memoria distribuită și Block RAM.

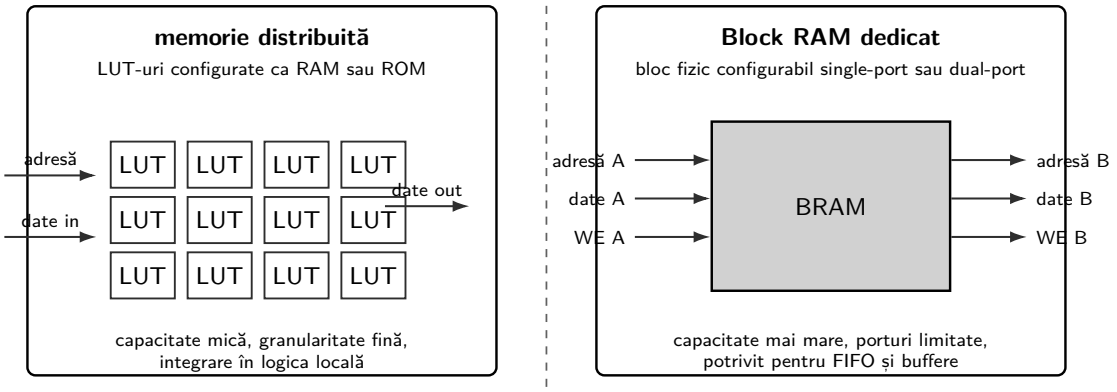


Figura 9.7 Memorie distribuită și Block RAM într-un FPGA.

Capacitatea necesară unei memorii este:

$$M = N_{\text{cuvinte}} \cdot W_{\text{cuvnt}}. \tag{9.26}$$

Pentru 1024 de cuvinte a câte 16 biți:

$$M = 1024 \cdot 16 = 16\,384\,\text{biți}.$$

Un bloc RAM dual-port permite două accesări în același ciclu, în limitele modurilor suportate. Aceasta nu înseamnă că un număr nelimitat de module poate accesa simultan aceeași memorie.

Pentru mai multe citiri paralele pot fi necesare:

- replicarea memoriei;
- împărțirea în bănci;
- arbitrarea accesului;
- memorii multiport construite din mai multe blocuri.

În aplicațiile de procesare a imaginilor, BRAM-urile sunt utilizate frecvent pentru buffere de linii, nu pentru stocarea întregului cadru.

9.6.2 Blocuri DSP și aritmetică

Blocurile DSP sunt optimizate pentru operații aritmetice precum:

$$P = A \cdot B + C. \tag{9.27}$$

Această operație de tip Multiply-Accumulate apare în filtre FIR, transformate, control numeric, convoluții, rețele neuronale și procesarea semnalelor de comunicație.

Figura 9.8 prezintă structura conceptuală a unui bloc DSP.

Un filtru FIR cu N coeficienți este descris prin:

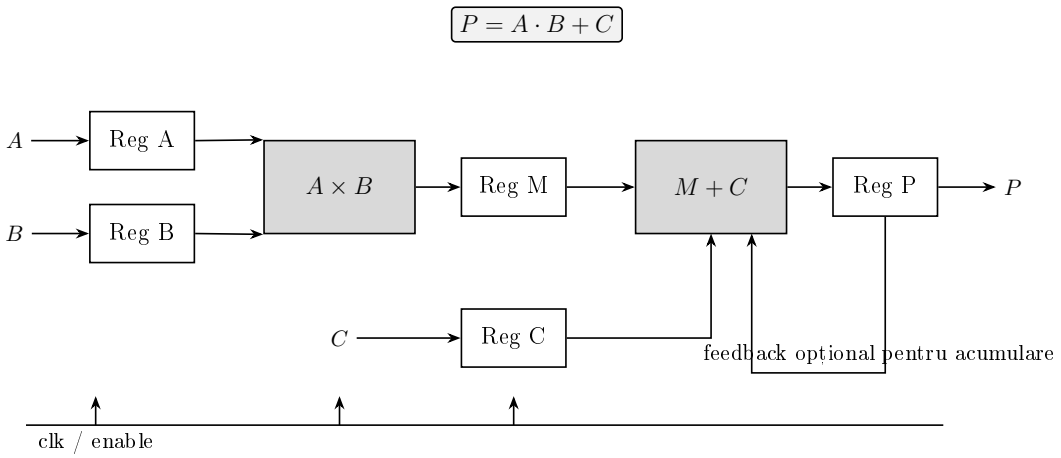


Figura 9.8 Structura conceptuală a unui bloc DSP configurabil.

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k]. \tag{9.28}$$

O implementare complet paralelă poate utiliza N multiplicatoare și o structură de adunare, iar o implementare serializată poate reutiliza un număr mai mic de blocuri DSP, dar necesită mai multe cicluri. Compromisul poate fi exprimat conceptual prin:

$$\text{resurse} \cdot \text{timp} \approx \text{volum de calcul}. \tag{9.29}$$

Această relație nu este o egalitate exactă, dar exprimă faptul că reducerea resurselor conduce frecvent la reutilizarea lor în timp. Blocurile DSP includ registre pipeline interne — activarea acestora permite frecvențe ridicate, dar crește latența exprimată în cicluri. Precizia numerică influențează utilizarea resurselor: un multiplicator pe 8 biți consumă mai puține resurse decât unul pe 32 de biți.

În aplicațiile embedded se utilizează frecvent reprezentarea în virgulă fixă:

$$x = -2^{I-1}b_{I-1} + \sum_{k=-F}^{I-2} b_k 2^k, \tag{9.30}$$

unde I reprezintă numărul biților pentru partea întreagă, iar F numărul biților fracționari. Reducerea preciziei trebuie validată pentru a evita overflow, pierderea preciziei, acumularea erorilor și degradarea algoritmului.

9.6.3 Ceasuri, transceivere și procesoare integrate

FPGA-urile includ rețele dedicate pentru distribuția semnalelor de ceas, care reduc skew-ul și permit alimentarea unui număr mare de registre. Blocurile de management al ceasului, precum PLL-urile, pot realiza multiplicarea frecvenței, divizarea frecvenței, deplasarea fazei, compensarea întârzierilor și generarea mai multor domenii de ceas.

Figura 9.9 prezintă generarea mai multor domenii de ceas.

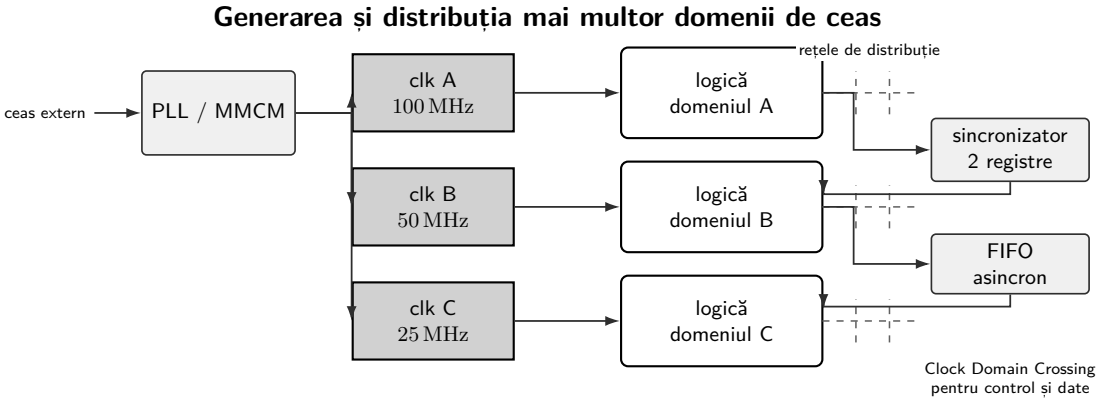


Figura 9.9 Generarea și distribuția mai multor domenii de ceas.

Semnalele transferate între domenii de ceas diferite necesită mecanisme de Clock Domain Crossing, prescurtat CDC. Pentru un semnal de control cu un singur bit se poate utiliza un sincronizator cu două registre; pentru magistrale și fluxuri de date se utilizează handshake, FIFO asincron, cod Gray sau protocoale specifice. Conectarea directă a unei magistrale între domenii asincrone poate conduce la metastabilitate și date incoerente.

Transceiverele seriale de mare viteză includ serializare și deserializare, recuperarea ceasului, egalizare, codare, detectarea erorilor și circuite analogice de transmisie și recepție. Acestea sunt utilizate pentru interfețe precum Ethernet, PCI Express sau legături seriale proprietare, suportul exact depinzând de familia FPGA.

Unele dispozitive includ procesoare fizice integrate. Arhitectura devine:

$$\text{procesor} + \text{logic programabil} + \text{memorie} + \text{periferice}.$$

Procesorul este potrivit pentru control, sisteme de operare și protocoale, iar logica FPGA este utilizată pentru accelerare și interfețe temporale precise.

Tabela 9.3 Rolul principalelor resurse dintr-un FPGA

Resursă	Rol principal	Exemplu de utilizare
LUT	Logică combinațională	Decodificare și multiplexare
Registru	Memorarea stării	Pipeline și mașini de stare
Carry chain	Aritmetică rapidă	Sumator și contor
Memorie distribuită	Stocare de capacitate mică	Tabel și registru de deplasare
Block RAM	Stocare internă dedicată	FIFO și buffer de linie
Bloc DSP	Operații aritmetice	Filtru FIR și convoluție
PLL	Managementul ceasului	Generarea mai multor frecvențe
Transceiver	Comunicație serială rapidă	Ethernet și PCI Express

9.7 Fluxul de proiectare FPGA

Dezvoltarea unei aplicații FPGA diferă fundamental de dezvoltarea software. Compilarea unui program produce instrucțiuni care vor fi executate de un procesor existent. În schimb, implementarea unei descrieri hardware produce configurația unui circuit digital.

Fluxul general cuprinde:

1. definirea cerințelor;
2. descrierea arhitecturii;
3. simularea funcțională;
4. sinteza logică;
5. plasarea și rutarea;
6. analiza temporală;
7. generarea bitstream-ului;
8. validarea pe hardware.

Figura 9.10 prezintă principalele etape.

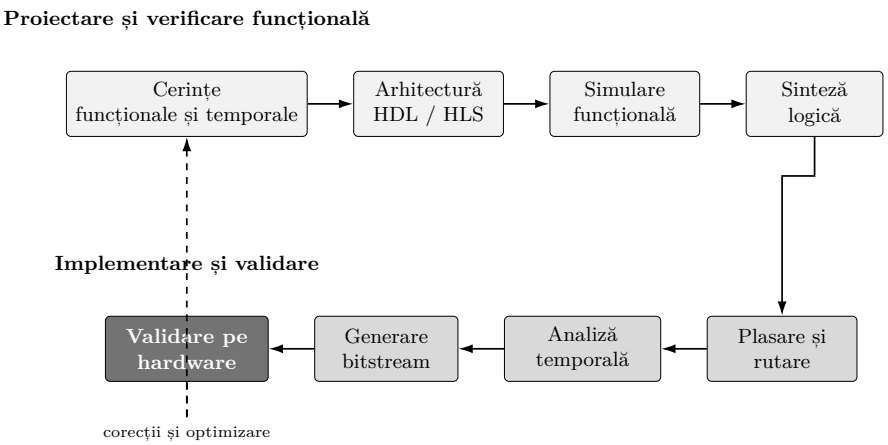


Figura 9.10 Fluxul principal de proiectare și verificare a unei aplicații FPGA.

Trecerea prin instrumentele de sinteză și implementare nu garantează automat corectitudinea circuitului. Proiectarea trebuie verificată atât funcțional, cât și temporal.

9.7.1 Descriere, simulare și sinteză

Circuitele FPGA sunt descrise tradițional prin limbaje HDL, precum VHDL sau Verilog. Aceste limbaje permit definirea:

- logicii combinaționale;

- registrelor;
- mașinilor de stare;
- interfețelor;
- relațiilor dintre domeniile de ceas.

O descriere VHDL simplă a unui sumator înregistrat este:

Exemplu 9.4 Sumator sincron descris în VHDL

```
process(clk)
begin
    if rising_edge(clk) then
        if reset = '1' then
            result <= (others => '0');
        elsif enable = '1' then
            result <= input_a + input_b;
        end if;
    end if;
end process;
```

Descrierea nu stabilește direct poziția fizică a sumatorului sau a registrelor. Instrumentul de sinteză transformă codul într-o rețea logică alcătuită din LUT-uri, registre, blocuri DSP, memorii, multiplexoare și conexiuni.

Înaintea sintezei se realizează simularea funcțională: modulul testat este stimulat printr-un *testbench*, iar ieșirile sunt comparate cu rezultatele așteptate.

Pentru un modul M, verificarea urmărește relația:

$$y_{\text{HDL}}(x) = y_{\text{referință}}(x), \tag{9.31}$$

pentru setul de intrări și scenarii analizate.

Testbench-ul trebuie să includă valori nominale, valori limită, reset, activări consecutive, situații de overflow, protocoale incomplete și combinații neobișnuite de semnale.

O alternativă la descrierea RTL este sinteza de nivel înalt, denumită HLS (*High-Level Synthesis*). În acest caz, funcția este descrisă printr-un limbaj de nivel înalt, iar instrumentul generează arhitectura hardware [24].

Un exemplu conceptual este:

Exemplu 9.5 Funcție simplă destinată sintezei de nivel înalt

```
void vector_add(
    const int input_a[SIZE],
    const int input_b[SIZE],
    int output[SIZE])
{
    for (int index = 0; index < SIZE; ++index)
    {
```

```
8            output[index] =
9                input_a[index] + input_b[index];
10       }
11   }
```

Descrierea software nu garantează o implementare paralelă. Instrumentul trebuie să decidă dacă bucla este executată secvențial, desfășurată parțial, desfășurată complet sau transformată într-un pipeline. Directivele HLS modifică raportul dintre resurse, latență și throughput — codul trebuie analizat ca descriere a unei arhitecturi, nu doar ca program C++.

9.7.2 Plasare, rutare și analiză temporală

După sinteză, instrumentul realizează plasarea resurselor fizice și rutarea conexiunilor. Plasarea stabilește poziția fiecărui element:

$$\text{instanț logic} \rightarrow \text{resurs fizic}.$$

Rutarea selectează traseele dintre resurse, iar rezultatul influențează direct întârzierea circuitului. Pentru o cale sincronă, timpul de sosire al datelor poate fi aproximat prin:

$$T_{\text{arrival}} = T_{\text{clk}\rightarrow\text{q}} + T_{\text{logic}} + T_{\text{rutare}}. \tag{9.32}$$

Timpul cerut este:

$$T_{\text{required}} = T_{\text{clk}} - T_{\text{setup}} - T_{\text{incertitudine}}. \tag{9.33}$$

Marja temporală, denumită *slack*, este:

$$S_{\text{timing}} = T_{\text{required}} - T_{\text{arrival}}. \tag{9.34}$$

Cerința este satisfăcută dacă:

$$S_{\text{timing}} \geq 0. \tag{9.35}$$

Un slack negativ indică o încălcare temporală — circuitul poate produce rezultate corecte în simularea funcțională, dar poate eșua pe hardware la frecvența solicitată. Metodele uzuale de corectare includ introducerea registrelor pipeline, reducerea nivelurilor de logică, utilizarea blocurilor DSP, replicarea unor semnale cu fan-out mare, modificarea plasării, reducerea frecvenței și corectarea constrângerilor temporale. Constrângerile incorecte sunt periculoase — dacă o cale reală nu este analizată, instrumentul poate raporta în mod eronat că implementarea satisface cerințele.

9.7.3 Utilizarea resurselor și validarea pe hardware

După implementare sunt generate rapoarte privind utilizarea resurselor: LUT-uri, registre, Block RAM, blocuri DSP, pini, resurse de ceas și transceivere. Gradul de utilizare al resursei R este:

$$U_R = \frac{N_{R,\text{utilizat}}}{N_{R,\text{disponibil}}}. \tag{9.36}$$

O utilizare apropiată de 100% poate conduce la dificultăți de rutare și la reducerea frecvenței maxime — simpla încadrare în capacitatea dispozitivului nu este întotdeauna suficientă.

După generarea bitstream-ului, validarea pe hardware urmărește corectitudinea rezultatelor, frecvența stabilă de funcționare, latența, throughput-ul, consumul energetic, comportamentul la reset, funcționarea interfețelor și temperaturile și tensiunile. Semnalele interne pot fi observate printr-un analizor logic integrat în FPGA, care memorează eșantioane într-o zonă de memorie internă, citită ulterior prin interfața de depanare.

Instrumentarea utilizează resurse și poate modifica plasarea sau rutarea. Rezultatele temporale trebuie reverificate după introducerea logicii de diagnostic.

9.8 Sisteme heterogene CPU–FPGA

O arhitectură heterogenă combină un procesor general cu logică reconfigurabilă. Procesorul execută software-ul de control, iar FPGA-ul implementează acceleratoare și interfețe cu cerințe temporale stricte.

Cele două componente pot fi:

- circuite separate;
- integrate în același pachet;
- integrate pe același circuit.

Integrarea reduce latența comunicației și permite utilizarea unor magistrale interne de mare viteză.

9.8.1 Arhitectura generală

Un sistem CPU–FPGA include, în mod obișnuit:

- unul sau mai multe nuclee CPU;
- memorie externă;
- logică programabilă;
- magistrale de control;
- căi de transfer pentru date;
- mecanisme de întrerupere.

Figura 9.11 prezintă organizarea conceptuală.

Comunicația poate fi împărțită în două categorii:

- cale de control;
- cale de date.

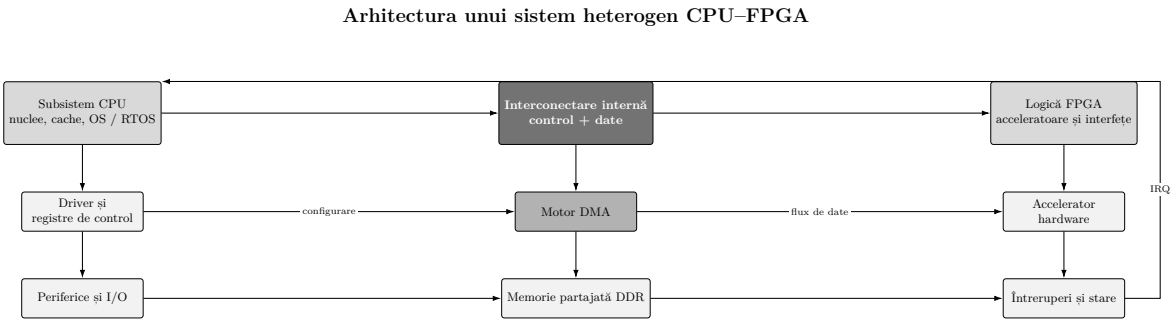


Figura 9.11 Arhitectura generală a unui sistem heterogen CPU–FPGA.

Calea de control este utilizată pentru:

- configurarea acceleratorului;
- scrierea parametrilor;
- pornirea execuției;
- citirea stării;
- confirmarea întreruperilor.

Calea de date transportă volumele mari de informații procesate de accelerator.

Registrele de control pot fi mapate în spațiul de adrese al procesorului:

Exemplu 9.6 Controlul conceptual al unui accelerator mapat în memorie

```
accelerator->source_address =
    input_buffer_physical_address;

accelerator->destination_address =
    output_buffer_physical_address;

accelerator->data_length =
    number_of_elements;

accelerator->control =
    start_command;

while ((accelerator->status & completed_flag) == 0U)
{
    wait_for_interrupt_or_poll();
}
```

Într-un sistem real trebuie respectate tipurile registrelor, barierele de memorie și regulile driverului.

9.8.2 DMA și transferul datelor

Transferul fiecărui element prin instrucțiuni CPU poate consuma mai mult timp decât procesarea propriu-zisă. Pentru volume mari se utilizează DMA (*Direct Memory Access*).

Fluxul este:

$$\text{memorie} \rightarrow \text{DMA} \rightarrow \text{accelerator} \rightarrow \text{DMA} \rightarrow \text{memorie}.$$

Procesorul configurează transferul, dar nu copiază fiecare cuvânt. Timpul aproximativ pentru transferul unei cantități D este:

$$T_{\text{transfer}} = T_{\text{inițializare}} + \frac{D}{B_{\text{efectiv}}}, \tag{9.37}$$

unde B_{efectiv} este lățimea de bandă efectivă.

Aceasta este mai mică decât valoarea teoretică din cauza arbitrajului, latenței memoriei, fragmentării transferurilor, accesului altor componente, limitărilor magistralei și alinierii datelor. Pentru transferuri mici, timpul de inițializare poate domina; pentru transferuri mari, lățimea de bandă devine factorul principal.

Double buffering permite suprapunerea transferului cu procesarea:

$$B_0 : \text{procesare},$$
$$B_1 : \text{transfer}.$$

După finalizare, rolurile bufferelor sunt inversate. Dacă timpii sunt echilibrați, durata unei iterații tinde spre:

$$T_{\text{iterație}} \approx \max\left(T_{\text{transfer}}, T_{\text{accelerator}}\right), \tag{9.38}$$

în locul sumei celor două durate.

9.8.3 Memorie partajată și coerența datelor

În multe sisteme, CPU-ul și acceleratorul accesează aceeași memorie externă. Această arhitectură reduce necesitatea copiilor explicite, dar introduce probleme de sincronizare.

Figura 9.12 prezintă fluxul principal.

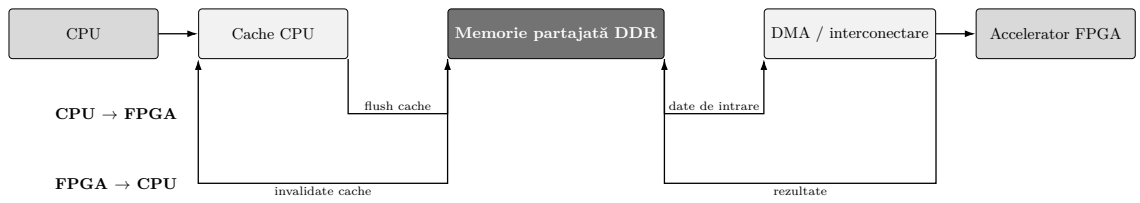


Figura 9.12 Accesul CPU și FPGA la o memorie partajată.

Dacă procesorul utilizează memorie cache, valoarea modificată poate să nu fie scrisă ime-

diat în memoria vizibilă acceleratorului. Înaintea transferului CPU–FPGA poate fi necesară:

$$\text{flush}\,(\text{cache})\,. \tag{9.39}$$

După ce acceleratorul scrie rezultatul, poate fi necesară:

$$\text{invalidate}\,(\text{cache})\,. \tag{9.40}$$

Unele platforme oferă interfețe coerente hardware, dar utilizarea lor trebuie verificată pentru sistemul concret. Sincronizarea dintre CPU și accelerator poate utiliza polling, întreruperi, semafoare hardware, cozi partajate sau descriptori DMA. Polling-ul este simplu, dar consumă timp de procesor; întreruperea permite procesorului să execute alte activități până la finalizarea acceleratorului.

9.9 Partiționarea hardware–software

Partiționarea stabilește ce funcții rămân în software și ce funcții sunt implementate în FPGA. O alegere incorectă poate produce un accelerator rapid, dar un sistem mai lent din cauza transferurilor și a controlului suplimentar.

Figura 9.13 prezintă principalele criterii.

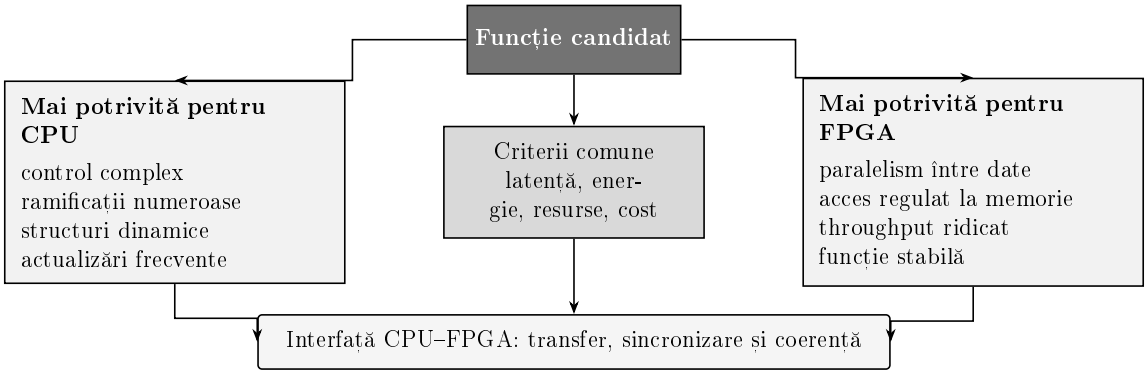


Figura 9.13 Criterii pentru partiționarea unei aplicații între CPU și FPGA.

9.9.1 Selectarea funcțiilor accelerate

O funcție este un candidat bun pentru hardware dacă prezintă contribuție mare la timpul total, paralelism între date, acces regulat la memorie, număr redus de ramificații, reutilizare frecventă și cerințe stricte de latență sau throughput. Funcțiile mai potrivite pentru software prezintă control complex, structuri dinamice, modificări frecvente, apeluri de sistem și protocoale și gestionarea erorilor. Un indicator util este intensitatea calculului:

$$I_C = \frac{N_{\text{operații}}}{N_{\text{bytes transferați}}}. \tag{9.41}$$

O funcție cu intensitate mare efectuează multe operații pentru fiecare byte transferat și poate beneficia mai mult de accelerare. O funcție cu intensitate redusă poate fi limitată de

memoria externă, indiferent de numărul unităților aritmetice implementate.

9.9.2 Costul transferului și pragul de eficiență

Timpul total al execuției accelerate este:

$$T_{\text{FPGA,total}} = T_{\text{setup}} + T_{\text{input}} + T_{\text{calcul}} + T_{\text{output}}. \tag{9.42}$$

Accelerarea este utilă dacă:

$$T_{\text{FPGA,total}} < T_{\text{CPU}}. \tag{9.43}$$

Figura 9.14 prezintă aceste componente.

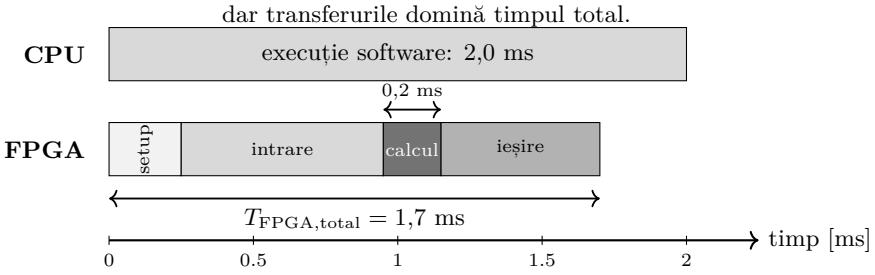


Figura 9.14 Componentele timpului total al unei funcții accelerate.

Considerăm o operație executată de CPU în 2 ms. Acceleratorul calculează rezultatul în 0,2 ms, dar transferurile necesită 1,5 ms.

Timpul total este:

$$T_{\text{FPGA,total}} = 0{,}2 + 1{,}5 = 1{,}7\,\text{ms}.$$

Accelerația este numai:

$$S = \frac{2}{1{,}7} \approx 1{,}18,$$

deși nucleul hardware este de zece ori mai rapid decât CPU-ul. Reducerea costului transferurilor poate necesita procesarea mai multor date într-un singur apel, păstrarea datelor în FPGA între operații, combinarea mai multor etape într-un accelerator, utilizarea fluxurilor de date, precum și DMA și double buffering.

9.9.3 Limitele accelerației globale

Accelerarea unei funcții nu produce aceeași accelerare pentru întreaga aplicație. Dacă proporția accelerată este p, iar accelerația sa este S_A, accelerația globală este descrisă de legea lui Amdahl [3]:

$$S_{\text{total}} = \frac{1}{(1-p) + \frac{p}{S_A}}. \tag{9.44}$$

Considerăm că 60% din timpul aplicației poate fi accelerat de zece ori:

$$p = 0{,}6, \qquad S_A = 10.$$

Rezultă:

$$S_{\text{total}} = \frac{1}{0{,}4 + 0{,}06} \approx 2{,}17.$$

Deși funcția este accelerată de zece ori, aplicația completă este accelerată de aproximativ 2,17 ori. Chiar dacă acceleratorul ar fi infinit de rapid:

$$\lim_{S_A \to \infty} S_{\text{total}} = \frac{1}{1-p}. \tag{9.45}$$

Pentru $p = 0{,}6$, limita maximă este:

$$S_{\max} = 2{,}5.$$

Rezultatul arată că optimizarea trebuie să urmărească întregul flux al aplicației.

9.10 Performanță, latență și utilizarea resurselor

Evaluarea unui accelerator FPGA trebuie să distingă între latență, throughput, interval de inițiere, frecvență, utilizarea resurselor, lățimea de bandă și consumul energetic. O singură valoare de speedup nu descrie complet comportamentul sistemului.

9.10.1 Latență, throughput și interval de inițiere

Pentru un pipeline cu N_P etape și frecvența f:

$$L_{\text{pipeline}} = \frac{N_P}{f}. \tag{9.46}$$

Dacă intervalul de inițiere este II, throughput-ul maxim este:

$$\Theta_{\text{pipeline}} = \frac{f}{II}. \tag{9.47}$$

Figura 9.15 prezintă diferența dintre latență și throughput.

Pentru un pipeline cu zece etape la 200 MHz:

$$L_{\text{pipeline}} = \frac{10}{200 \cdot 10^6} = 50\,\text{ns}.$$

Dacă:

$$II = 1,$$

poate fi acceptat un nou element la fiecare:

$$5\,\text{ns}.$$

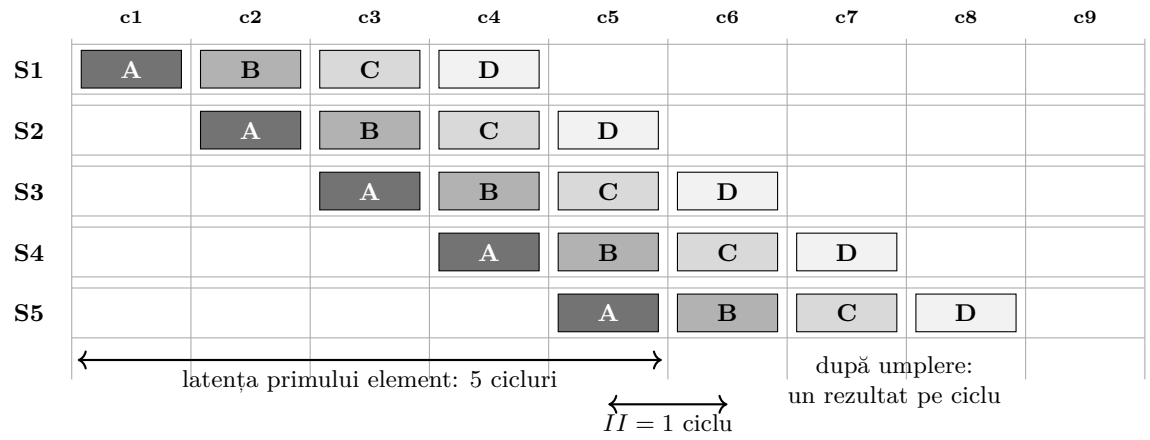


Figura 9.15 Latența și throughput-ul unei arhitecturi pipeline.

Prima ieșire apare după 50 ns, dar următoarele ieșiri pot apărea la fiecare 5 ns.

9.10.2 Paralelism, resurse și lățime de bandă

Creșterea paralelismului necesită mai multe resurse. Pentru P unități identice, utilizarea poate fi aproximată prin:

$$R(P) \approx R_{\text{control}} + PR_{\text{unitate}}. \tag{9.48}$$

Throughput-ul nu crește nelimitat — dacă fiecare rezultat necesită D bytes, lățimea de bandă necesară este:

$$B_{\text{necesar}} = \Theta D. \tag{9.49}$$

Cerința de funcționare este:

$$B_{\text{necesar}} \leq B_{\text{memorie}}. \tag{9.50}$$

Dacă acceleratorul produce 100 milioane de rezultate pe secundă, iar fiecare rezultat necesită 8 bytes:

$$B_{\text{necesar}} = 100 \cdot 10^6 \cdot 8 = 800\,\text{MB/s}.$$

O memorie care oferă efectiv numai 500 MB/s nu poate alimenta acceleratorul la rata proiectată. Soluțiile includ reutilizarea locală a datelor, Block RAM, memorii împărțite în bănci, transferuri în rafală, reducerea preciziei și procesarea în flux.

9.10.3 Evaluarea completă a acceleratorului

O comparație corectă între CPU și FPGA trebuie să utilizeze:

- aceleași date de intrare;
- aceeași precizie numerică;

- același algoritm sau o eroare echivalentă;
- toate transferurile;
- costurile driverului;
- aceeași definiție a timpului măsurat.

Accelerația este:

$$S = \frac{T_{\text{CPU}}}{T_{\text{FPGA,total}}}. \tag{9.51}$$

Energia consumată este:

$$E = \int_0^T P(t)\,dt. \tag{9.52}$$

Pentru o putere aproximativ constantă:

$$E \approx P_{\text{medie}} T. \tag{9.53}$$

Indicatorii recomandați sunt:

Tabela 9.4 Indicatori pentru evaluarea unui accelerator FPGA

Indicator	Semnificație	Observație
Latență	Durata unei operații	Include transferurile
Throughput	Rezultate pe unitatea de timp	Relevant pentru fluxuri continue
Speedup	Raportul timpilor CPU și FPGA	Trebuie măsurat end-to-end
Energie	Energia pentru sarcina completă	Nu doar puterea instantanee
LUT și registre	Logică generală utilizată	Influențează rutarea
BRAM și DSP	Resurse specializate utilizate	Pot limita paralelizarea
Lățime de bandă	Rata transferului de date	Poate limita acceleratorul

Rezultatele trebuie raportate împreună cu dispozitivul FPGA, frecvența, dimensiunea datelor, precizia numerică, configurația memoriei și instrumentele și opțiunile de implementare.

9.11 Configurarea și securitatea sistemelor reconfigurabile

Funcționalitatea unui FPGA este definită printr-un fișier de configurare, denumit în mod uzual *bitstream*. Acesta stabilește conținutul LUT-urilor, conexiunile programabile și modurile de funcționare ale resurselor interne.

Posibilitatea reconfigurării reprezintă un avantaj important, dar introduce și riscuri specifice — un atacator care poate înlocui sau modifica bitstream-ul poate schimba chiar structura hardware a sistemului. Obiectivele principale ale securității sunt autenticitatea configurației, integritatea bitstream-ului, confidențialitatea proprietății intelectuale, prevenirea revenirii la versiuni vulnerabile, recuperarea după o actualizare eșuată și protejarea cheilor criptografice.

9.11.1 Bitstream, autenticitate și confidențialitate

Bitstream-ul trebuie considerat o componentă executabilă critică. Încărcarea unei configurații neautorizate poate modifica interfețele, logica de control, mecanismele de securitate sau procesarea datelor. Autenticitatea și integritatea pot fi asigurate printr-o semnătură digitală:

$$\sigma = \mathrm{Sign}\left(K_{\mathrm{privat}}, H(B)\right), \tag{9.54}$$

unde B este bitstream-ul, H este o funcție hash criptografică, K_{privat} este cheia privată a producătorului, iar σ este semnătura digitală. Dispozitivul verifică:

$$\mathrm{Verify}\left(K_{\mathrm{public}}, H(B), \sigma\right) = \mathrm{valid}. \tag{9.55}$$

Semnătura demonstrează că configurația provine de la o sursă autorizată și că nu a fost modificată. Confidențialitatea este un obiectiv diferit — pentru protejarea proprietății intelectuale, bitstream-ul poate fi criptat:

$$B_{\mathrm{criptat}} = \mathrm{Enc}\left(K_{\mathrm{dispozitiv}}, B\right). \tag{9.56}$$

Criptarea împiedică interpretarea directă a configurației, dar nu demonstrează singură că aceasta este autentică — într-un sistem sigur sunt utilizate atât verificarea autenticității, cât și criptarea.

Figura 9.16 prezintă fluxul recomandat.

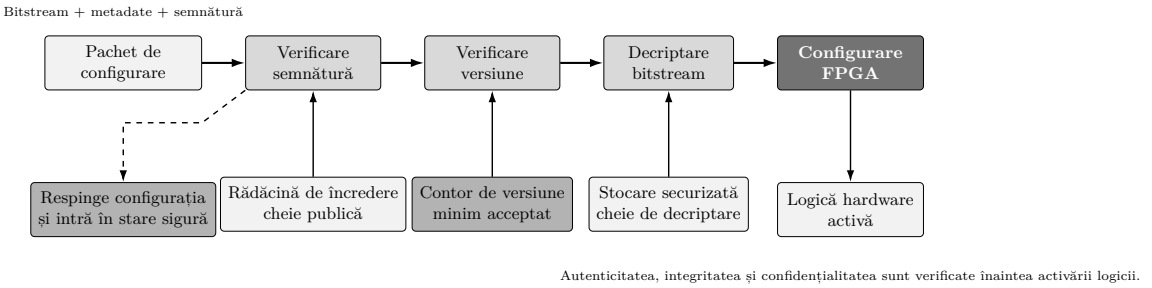


Figura 9.16 Lanțul de verificare și încărcare sigură a unui bitstream.

Cheile criptografice pot fi păstrate în memorie internă nevolatilă, siguranțe electronice programabile, memorie alimentată permanent, modul hardware de securitate sau element securizat extern. Protejarea cheilor este critică — dacă o cheie este extrasă, criptarea bitstream-ului nu mai oferă protecție efectivă.

9.11.2 Secure Boot și actualizarea sigură

Secure Boot definește un lanț de încredere în care fiecare etapă verifică următoarea componentă înainte de execuție sau configurare.

Fluxul conceptual este:

$$\text{ROM de pornire} \rightarrow \text{bootloader} \rightarrow \text{bitstream} \rightarrow \text{software}.$$

Rădăcina de încredere trebuie să fie protejată împotriva modificării. Aceasta poate conține cheia publică utilizată pentru verificarea componentelor următoare.

O actualizare sigură trebuie să parcurgă etapele:

1. descărcarea pachetului;
2. verificarea semnăturii;
3. verificarea versiunii;
4. stocarea într-o zonă temporară;
5. activarea controlată;
6. confirmarea funcționării;
7. păstrarea sau ștergerea versiunii anterioare.

Numărul versiunii poate fi verificat prin:

$$V_{\text{nou}} > V_{\text{minim acceptat}}. \tag{9.57}$$

Această condiție previne atacurile de tip rollback, în care este instalată o configurație veche, semnată corect, dar cunoscută ca vulnerabilă.

Figura 9.17 prezintă o arhitectură cu două imagini de configurare.

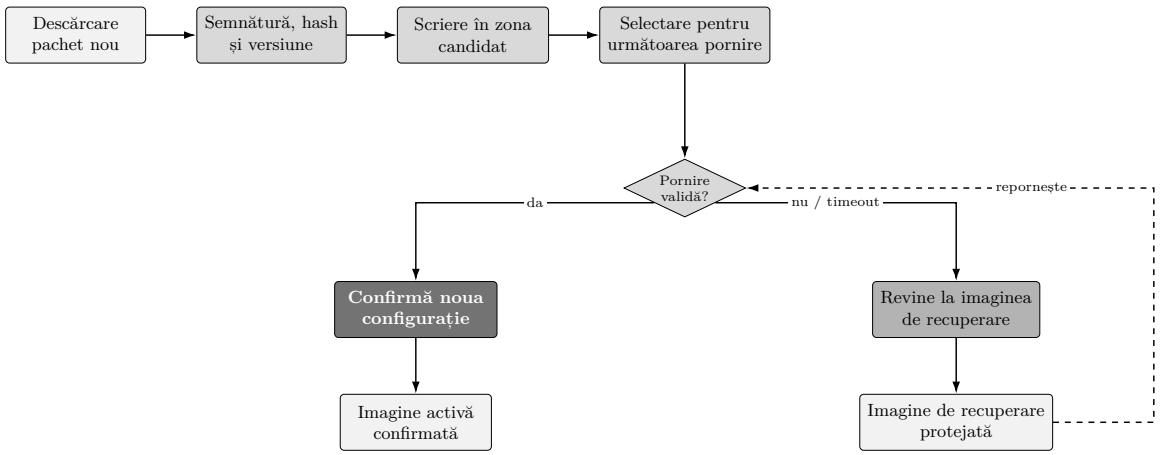


Figura 9.17 Actualizarea sigură și recuperarea configurației FPGA.

Sistemul poate păstra o imagine principală, o imagine de recuperare, un contor al pornirilor eșuate și un indicator de confirmare a noii versiuni. Dacă noua configurație nu finalizează pornirea într-un timp stabilit, sistemul revine automat la imaginea de recuperare.

9.11.3 Atacuri fizice și protecția interfețelor

Securitatea configurației nu elimină atacurile fizice — un adversar cu acces la dispozitiv poate analiza consumul energetic, emisiile electromagnetice, durata operațiilor, semnalele de depanare sau comportamentul la variații de tensiune sau ceas. Atacurile de tip fault injection încearcă să producă erori controlate prin modificarea tensiunii, perturbarea ceasului, impulsuri electromagnetice, laser sau temperaturi în afara domeniului normal; o eroare produsă în timpul verificării poate determina acceptarea unei configurații invalide sau divulgarea unor informații.

Măsurile de protecție includ detectarea tensiunii și frecvenței anormale, verificări redundante, compararea rezultatelor, monitorizarea temperaturii, ștergerea cheilor la detectarea unei manipulări și limitarea accesului la interfețele de depanare. Interfețe precum JTAG sunt utile în dezvoltare, dar trebuie dezactivate, autentificate sau restricționate în produsul final.

Securitatea sistemului trebuie analizată pe întregul lanț:

$$\text{actualizare} \rightarrow \text{stocare} \rightarrow \text{configurare} \rightarrow \text{execuție} \rightarrow \text{diagnostic}.$$

Protejarea exclusivă a bitstream-ului nu este suficientă dacă bootloader-ul, memoria externă sau aplicația software pot fi compromise [87].

9.12 Studiu de caz: accelerarea procesării imaginilor

Se consideră un sistem embedded care procesează imagini provenite de la o cameră cu rezoluția:

$$1920 \times 1080$$

la o rată de:

$$60\,\text{cadre/s}.$$

Fluxul de procesare include:

1. conversie la niveluri de gri;
2. filtrare spațială 3×3;
3. calculul gradientului Sobel;
4. aplicarea unui prag;
5. transmiterea imaginii binare către procesor.

9.12.1 Cerințele și arhitectura acceleratorului

Numărul de pixeli procesați într-o secundă este:

$$R_{\text{pixeli}} = 1920 \cdot 1080 \cdot 60 \\ = 124\,416\,000 \text{ pixeli/s}. \tag{9.58}$$

Acceleratorul trebuie să susțină cel puțin:

$$124{,}416\,\text{Mpixeli/s}.$$

Figura 9.18 prezintă arhitectura propusă.

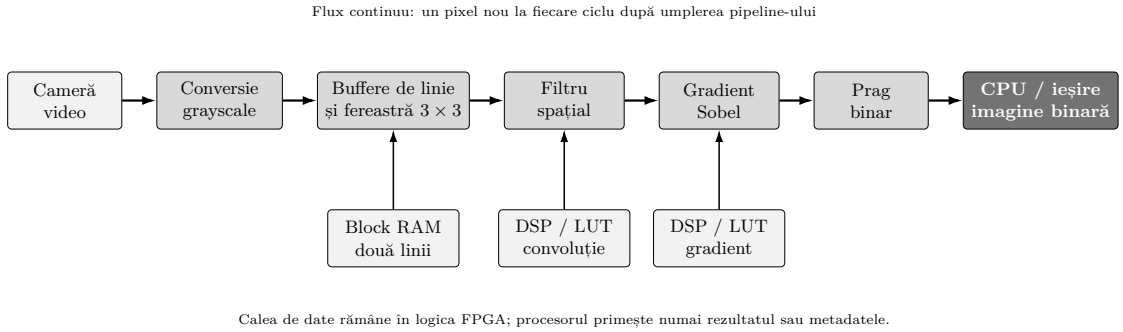


Figura 9.18 Pipeline hardware pentru filtrarea și detectarea muchiilor.

Etapele sunt conectate printr-un flux continuu de pixeli. Fiecare modul primește un pixel și transmite rezultatul către etapa următoare.

Pentru operațiile 3×3 sunt necesare trei linii ale imaginii. Stocarea întregului cadru în memoria internă ar necesita:

$$M_{\text{cadru}} = 1920 \cdot 1080 \cdot 8 = 16\,588\,800 \text{ biți}. \tag{9.59}$$

O soluție mai eficientă utilizează două buffere de linie și registre pentru fereastra curentă:

$$M_{\text{linii}} = 2 \cdot 1920 \cdot 8 = 30\,720 \text{ biți}. \tag{9.60}$$

Figura 9.19 prezintă acest mecanism.

Reducerea memoriei este semnificativă:

$$\frac{M_{\text{cadru}}}{M_{\text{linii}}} = 540.$$

Prin urmare, pentru procesarea în flux nu este necesară stocarea completă a imaginii în memoria internă.

9.12.2 Latență și throughput

Se consideră o frecvență de funcționare de:

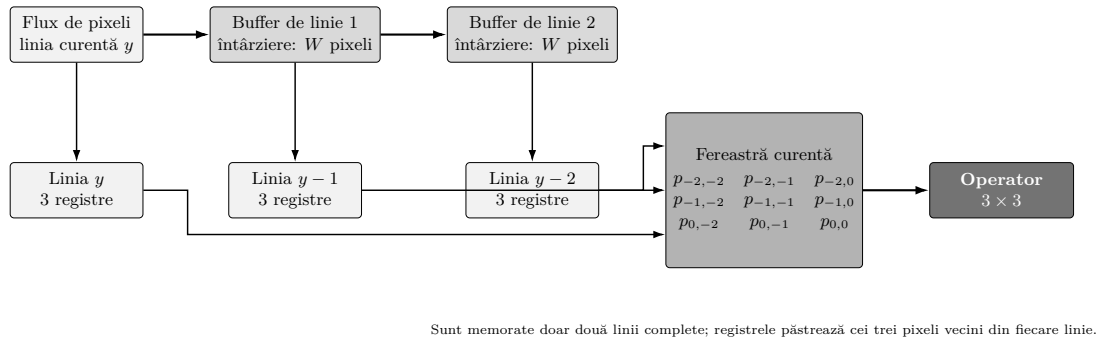


Figura 9.19 Formarea unei ferestre 3×3 cu ajutorul bufferelor de linie.

$$f = 150\,\text{MHz}.$$

Pipeline-ul este proiectat cu:

$$II = 1.$$

Throughput-ul teoretic este:

$$\Theta = \frac{f}{II} = 150\,\text{Mpixeli/s}. \tag{9.61}$$

Deoarece:

$$150 > 124{,}416,$$

acceleratorul poate procesa fluxul video de 60 cadre/s.
Timpul necesar procesării numărului de pixeli dintr-un cadru este:

$$\begin{aligned} T_{\text{cadru}} &= \frac{1920 \cdot 1080}{150 \cdot 10^6} \\ &\approx 13{,}82\,\text{ms}. \end{aligned} \tag{9.62}$$

Perioada disponibilă pentru un cadru la 60 cadre/s este:

$$T_{\text{disponibil}} = \frac{1}{60} \approx 16{,}67\,\text{ms}. \tag{9.63}$$

Rezultă o marjă de:

$$16{,}67 - 13{,}82 = 2{,}85\,\text{ms}.$$

Prima ieșire validă apare numai după umplerea bufferelor de linie și a pipeline-ului. Pentru două linii, doi pixeli și 12 etape interne, latența aproximativă este:

$$L_{\text{inițial}} \approx \frac{2 \cdot 1920 + 2 + 12}{150 \cdot 10^6} \approx 25{,}7\,\mu\text{s}. \tag{9.64}$$

Această latență inițială nu trebuie confundată cu timpul necesar procesării întregului cadru.

9.12.3 Evaluarea accelerației end-to-end

Se presupune că implementarea software necesită:

$$T_{\text{CPU}} = 35\,\text{ms/cadru}.$$

Pentru un flux direct cameră–FPGA, timpul acceleratorului este:

$$T_{\text{FPGA}} \approx 13{,}82\,\text{ms/cadru}.$$

Accelerația este:

$$S_{\text{streaming}} = \frac{35}{13{,}82} \approx 2{,}53. \tag{9.65}$$

Dacă imaginile sunt stocate mai întâi în memoria externă, trebuie incluse transferurile. Presupunem:

$$T_{\text{transfer}} = 3\,\text{ms}.$$

Timpul total devine:

$$T_{\text{total}} = 13{,}82 + 3 = 16{,}82\,\text{ms}. \tag{9.66}$$

Accelerația end-to-end scade la:

$$S_{\text{total}} = \frac{35}{16{,}82} \approx 2{,}08. \tag{9.67}$$

În plus:

$$16{,}82\,\text{ms} > 16{,}67\,\text{ms},$$

ceea ce înseamnă că cerința de 60 cadre/s nu mai este satisfăcută.

Figura 9.20 compară cele trei variante.

Rezultatul arată că integrarea datelor este la fel de importantă ca performanța nucleului hardware.

Pentru îmbunătățirea sistemului pot fi utilizate:

- conectarea directă a camerei la pipeline;

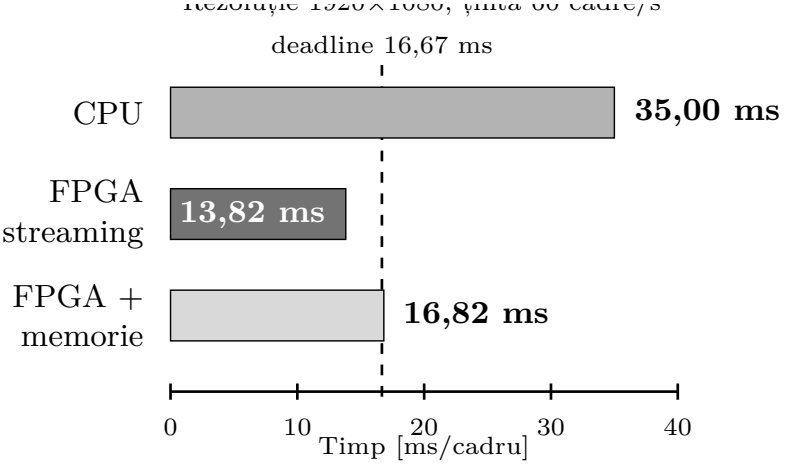


Figura 9.20 Compararea implementării CPU, FPGA streaming și FPGA cu transfer prin memorie.

- transferuri DMA în rafală;
- double buffering;
- procesarea mai multor etape în același accelerator;
- reducerea preciziei datelor;
- evitarea copierilor intermediare.

Evaluarea finală trebuie să includă:

- LUT-uri și registre;
- Block RAM;
- blocuri DSP;
- frecvența maximă;
- latența;
- throughput-ul;
- consumul energetic;
- precizia rezultatului.

Accelerarea este justificată numai dacă sistemul complet respectă cerințele de performanță, resurse și energie.

9.13 Întrebări și probleme propuse

9.13.1 Întrebări de verificare

1. Ce informații sunt conținute într-un bitstream?
2. Care este diferența dintre semnarea și criptarea unui bitstream?
3. Ce reprezintă rădăcina de încredere într-un proces Secure Boot?
4. De ce este necesară protecția împotriva revenirii la versiuni vechi?
5. Care este rolul imaginii de recuperare?

6. De ce interfața JTAG poate reprezenta un risc de securitate?
7. Ce este un atac de tip fault injection?
8. De ce procesarea unei imagini în flux necesită numai buffere de linie?
9. Care este diferența dintre latența inițială și timpul de procesare al unui cadru?
10. De ce un accelerator rapid poate să nu satisfacă rata necesară de cadre?
11. Ce efect are intervalul de inițiere asupra throughput-ului?
12. De ce timpul de transfer trebuie inclus în calculul accelerației?

9.13.2 Probleme de calcul și proiectare

1. Un bitstream are dimensiunea de 48 Mbit și este încărcat printr-o interfață cu rata efectivă de 120 Mbit/s. Calculați timpul minim de configurare.
2. O cameră produce imagini de 1280×720 pixeli la 90 cadre/s. Calculați rata minimă de procesare în pixeli pe secundă.
3. Un accelerator funcționează la 100 MHz cu $II = 2$. Calculați throughput-ul maxim și verificați dacă poate procesa fluxul din problema precedentă.
4. Pentru un filtru 5×5 aplicat unei imagini cu lățimea de 2048 pixeli și 8 biți/pixel, determinați memoria minimă necesară bufferelor de linie.
5. O funcție necesită 24 ms pe CPU. Acceleratorul execută calculul în 4 ms, iar transferurile necesită în total 5 ms. Calculați accelerația nucleului și accelerația end-to-end.
6. Proiectați arhitectura de actualizare pentru un sistem care trebuie să păstreze o configurație principală și una de recuperare. Precizați etapele de verificare și condițiile de revenire.
7. Propuneți partiționarea CPU–FPGA pentru un sistem de inspecție vizuală care realizează achiziție, filtrare, clasificare, comunicație și stocarea rezultatelor.
8. Definiți un plan experimental pentru măsurarea latenței, throughput-ului, consumului și utilizării resurselor unui accelerator de procesare a imaginilor.

10. Energy Harvesting și Sisteme Embedded Autonome Energetic

Introducere în Energy Harvesting
Autonomie și neutralitate energetică
Surse de energie pentru sisteme embedded
Conversia fotovoltaică
Conversia termoelectrică și mecanică
Recoltarea energiei RF
Conversia și managementul puterii
Modelarea consumului și a balanței energetice
Stocarea energiei
Noduri senzoriale autonome energetic
Studiu de caz: nod IoT alimentat fotovoltaic
Întrebări și probleme propuse

10.1 Introducere în Energy Harvesting

Numeroase sisteme embedded trebuie să funcționeze timp îndelungat în locații în care alimentarea de la rețea nu este disponibilă, iar înlocuirea bateriilor este dificilă sau costisitoare. Exemple reprezentative sunt nodurile IoT distribuite, senzorii industriali instalați pe utilaje, sistemele de monitorizare structurală și dispozitivele medicale purtabile.

Reducerea consumului microcontrolerelor, senzorilor și transceiverelor a făcut posibilă alimentarea unor astfel de sisteme prin energia disponibilă în mediul înconjurător. Procesul de captare, conversie și utilizare a acestei energii este denumit *Energy Harvesting* [64, 83]. Sursele utilizate pot include radiația luminoasă, diferențele de temperatură, vibrațiile și mișcarea, câmpurile electromagnetice sau procesele biomecanice.

Energy Harvesting nu implică în mod obligatoriu eliminarea bateriei. În multe aplicații, energia recoltată completează energia stocată și reduce frecvența operațiilor de mentenanță. Numai în condiții bine definite poate fi obținută funcționarea autonomă pe termen foarte lung.

10.1.1 Principiul general

Un sistem de Energy Harvesting transformă o formă de energie disponibilă în mediu într-o tensiune și un curent utilizabile de circuitul electronic.

Lanțul energetic general este:

$$\text{surs} \rightarrow \text{convertor} \rightarrow \text{management energetic} \rightarrow \text{stocare} \rightarrow \text{sistem embedded}.$$

Figura 10.1 prezintă această arhitectură.

Arhitectura generală a unui sistem de Energy Harvesting

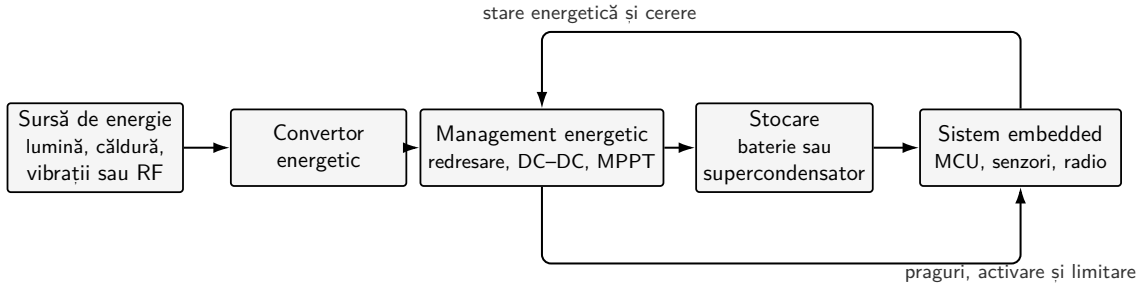


Figura 10.1 Arhitectura generală a unui sistem de Energy Harvesting.

Sursa furnizează o putere dependentă de mediul exterior. Convertorul realizează transformarea fizică — de exemplu lumină în energie electrică, diferență de temperatură în tensiune, deformare mecanică în sarcină electrică sau radiație electromagnetică în tensiune continuă. Circuitul de management energetic adaptează tensiunea, urmărește punctul optim de funcționare al sursei și controlează încărcarea elementului de stocare.

Puterea utilă disponibilă la consumator poate fi exprimată simplificat prin:

$$P_{\text{util}} = \eta_{\text{conv}} \eta_{\text{PMIC}} P_{\text{sursă}}, \tag{10.1}$$

unde $P_{\text{sursă}}$ este puterea captată din mediu, η_{conv} este randamentul convertorului energetic, iar η_{PMIC} este randamentul circuitului de management. Randamentul global nu este constant. Acesta poate varia cu tensiunea de intrare, curentul de sarcină, temperatura și punctul de funcționare al sursei.

10.1.2 Rolul stocării și al managementului energetic

Energia recoltată și energia consumată nu sunt, de regulă, disponibile în același moment.

De exemplu, un panou fotovoltaic produce energie în timpul zilei, în timp ce nodul senzorial trebuie să funcționeze și noaptea. În mod similar, un transceiver poate solicita pentru câteva milisecunde o putere mult mai mare decât cea furnizată instantaneu de sursă.

Elementul de stocare îndeplinește două funcții: acumulează energia produsă în exces și furnizează vârfurile de putere cerute de consumator.

Energia stocată evoluează conform relației:

$$E_s(t_2) = E_s(t_1) + \int_{t_1}^{t_2} [P_h(t) - P_c(t) - P_l(t)]\ dt, \tag{10.2}$$

unde E_s este energia stocată, P_h este puterea utilă recoltată, P_c este puterea consumată, iar P_l reprezintă pierderile și autodescărcarea.

Energia trebuie să rămână în limitele:

$$0 \leq E_s(t) \leq E_{\max}. \tag{10.3}$$

Depășirea limitei superioare conduce la pierderea energiei care nu mai poate fi stocată. Atingerea limitei inferioare poate determina oprirea sistemului.

Din acest motiv, proiectarea nu trebuie să urmărească numai energia medie. Trebuie analizate și tensiunea minimă de pornire, curentul maxim solicitat, durata perioadelor fără recoltare, pierderile de conversie, capacitatea efectiv utilizabilă și comportamentul la descărcarea stocării.

10.1.3 Domenii de aplicare

Energy Harvesting este atractiv atunci când reducerea mentenanței justifică dimensiunea și costul suplimentar al sursei și al circuitului de management.

Aplicațiile reprezentative includ monitorizarea utilajelor industriale, monitorizarea podurilor, clădirilor și căilor ferate, agricultura inteligentă, stațiile meteorologice distribuite, senzorii pentru infrastructuri energetice, electronica purtabilă, dispozitivele IoT cu funcționare intermitentă și etichetele și senzorii fără baterie.

Un sistem nu devine autonom numai prin conectarea unui convertor energetic. Trebuie adaptate simultan arhitectura hardware, frecvența măsurătorilor, strategia de comunicație, algoritmii software, capacitatea de stocare și comportamentul în condiții energetice nefavorabile.

10.2 Autonomie și neutralitate energetică

Autonomia reprezintă perioada în care sistemul poate funcționa fără alimentare externă sau intervenție de mentenanță.

Pentru un sistem alimentat exclusiv din baterie, autonomia este limitată de energia inițială disponibilă. Pentru un sistem cu Energy Harvesting, durata de funcționare depinde de balanța dintre energia recoltată, energia consumată și capacitatea de stocare.

10.2.1 Limitările alimentării exclusiv din baterii

Autonomia ideală a unei baterii poate fi estimată prin:

$$T_{\text{ideal}} = \frac{C_B}{I_{\text{mediu}}}, \tag{10.4}$$

unde C_B este capacitatea exprimată în amper-oră, iar I_{mediu} este curentul mediu.

Pentru o baterie de:

$$C_B = 2000\,\text{mAh}$$

și un curent mediu de:

$$I_{\text{mediu}} = 0{,}1\,\text{mA},$$

rezultă:

$$\begin{aligned} T_{\text{ideal}} &= \frac{2000}{0{,}1} \\ &= 20\,000\,\text{h} \approx 2{,}28\,\text{ani}. \end{aligned} \tag{10.5}$$

Aceasta este o valoare teoretică. Autonomia reală este influențată de autodescărcare, temperatură, îmbătrânire, vârfurile de curent, tensiunea minimă acceptată de sistem și capacitatea care poate fi utilizată în siguranță.

O estimare mai realistă poate introduce un factor de utilizare k_u:

$$T_{\text{real}} \approx k_u \frac{C_B}{I_{\text{mediu}}}, \qquad 0 < k_u < 1. \tag{10.6}$$

Pentru $k_u = 0{,}75$, autonomia devine aproximativ:

$$T_{\text{real}} \approx 1{,}71\,\text{ani}.$$

Într-o rețea cu mii de noduri, înlocuirea periodică a bateriilor poate deveni mai costisitoare decât echipamentele monitorizate.

10.2.2 Neutralitatea energetică

Un sistem este neutru energetic pe intervalul $[t_1, t_2]$ dacă energia consumată nu depășește energia recoltată și energia disponibilă inițial:

$$\int_{t_1}^{t_2} P_c(t)\,dt \leq E_s(t_1) + \int_{t_1}^{t_2} P_h(t)\,dt. \tag{10.7}$$

Pe termen lung, o condiție necesară este:

$$\overline{P_h} \geq \overline{P_c} + \overline{P_l}, \tag{10.8}$$

unde bara indică valoarea medie pe intervalul analizat.

Această relație nu este suficientă dacă energia recoltată și consumul sunt distribuite diferit în timp. Un sistem poate produce suficientă energie într-o lună și totuși se poate opri într-o perioadă consecutivă de mai multe zile fără sursă.

Figura 10.2 prezintă un exemplu în care stocarea compensează variația dintre producție și consum.

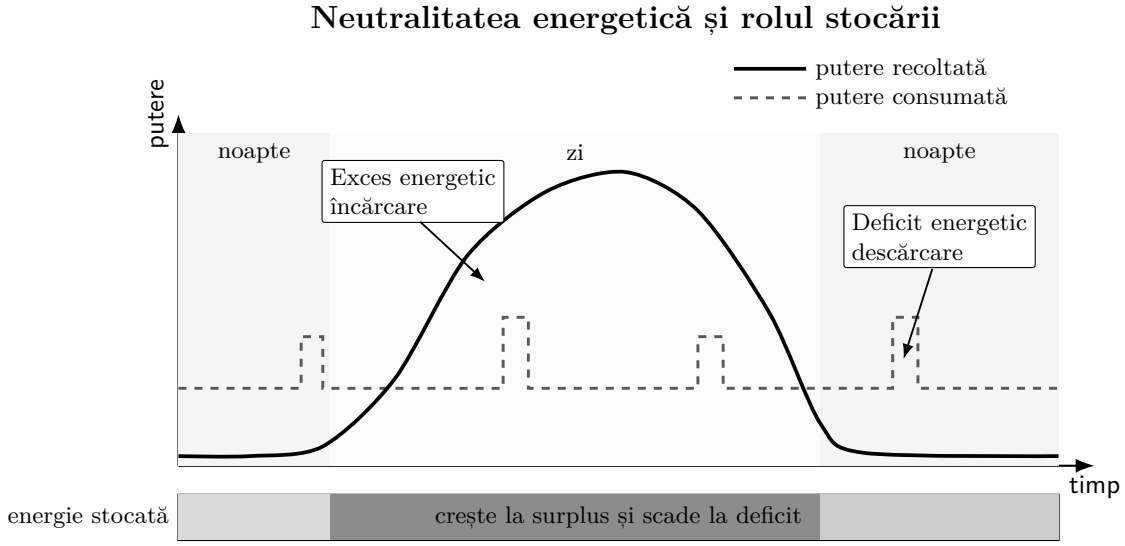


Figura 10.2 Evoluția energiei recoltate, consumate și stocate.

Pentru o proiectare robustă trebuie utilizat un scenariu nefavorabil, nu numai media anuală. De exemplu, pentru un sistem solar pot fi analizate luna cu producția minimă, mai multe zile consecutive cu iluminare redusă, temperaturile care reduc capacitatea bateriei, orientarea și umbrirea panoului și degradarea componentelor.

10.2.3 Adaptarea funcționării la energia disponibilă

Dacă energia disponibilă scade, sistemul poate reduce nivelul serviciului furnizat înainte de a ajunge la oprire.

Parametrii care pot fi adaptați includ perioada de eșantionare, numărul transmisiilor, puterea radio, precizia măsurătorilor, complexitatea procesării și numărul senzorilor activi.

O politică simplificată poate utiliza nivelul energiei stocate:

$$\text{mod} = \begin{cases} \text{normal}, & E_s \geq E_H, \\ \text{economic}, & E_L < E_s < E_H, \\ \text{supraviețuire}, & E_s \leq E_L. \end{cases} \tag{10.9}$$

În modul economic, sistemul poate transmite mai rar. În modul de supraviețuire, poate păstra numai măsurătorile esențiale și poate opri comunicațiile necritice.

Această strategie este denumită frecvent management energetic adaptiv [48].

10.3 Surse de energie pentru sisteme embedded

Alegerea sursei începe prin măsurarea condițiilor reale din mediul de funcționare. O tehnologie performantă în laborator poate furniza foarte puțină energie dacă nu este adaptată aplicației.

Principalele criterii sunt puterea medie disponibilă, puterea minimă în scenariul nefavo-

rabil, variația în timp, dimensiunea convertorului, tensiunea și impedanța sursei, condițiile de mediu și durata de viață.

10.3.1 Clasificarea surselor

Sursele utilizate în sistemele embedded pot fi clasificate după forma energiei primare.

Tabela 10.1 Principalele surse utilizate în sistemele de Energy Harvesting

Sursă	Convertor	Avantaj principal	Limitare principală
Lumină	Celulă fotovoltaică	Putere relativ ridicată	Dependență de iluminare
Gradient termic	Generator termoelectric	Fără componente mobile	Necesită diferență de temperatură
Vibrații	Piezoelectric, electromagnetic sau capacitiv	Potrivit pentru utilaje	Sensibil la frecvență și amplitudine
Radiofrecvență	Antenă și redresor	Posibilitate de funcționare fără contact	Putere ambientală redusă
Mișcare umană	Convertor mecanic	Integrare în dispozitive purtabile	Disponibilitate dependentă de utilizator

Valorile efective diferă cu mai multe ordine de mărime în funcție de suprafață, amplasare și condițiile mediului. Din acest motiv, comparațiile bazate pe o singură valoare de densitate de putere trebuie utilizate cu atenție.

10.3.2 Disponibilitate și predictibilitate

O sursă trebuie evaluată atât prin putere, cât și prin disponibilitate.

Energia solară prezintă variații zilnice și sezoniere, dar acestea pot fi estimate. Vibrațiile industriale pot fi disponibile numai în timpul funcționării utilajului. Energia termoelectrică dispare dacă cele două fețe ale convertorului ajung la aceeași temperatură.

Pentru sursa j, energia disponibilă într-un interval este:

$$E_{h,j} = \int_{t_1}^{t_2} \eta_j(t) P_j(t)\, dt. \tag{10.10}$$

Evaluarea trebuie realizată pe o perioadă reprezentativă pentru aplicație: pentru un sistem agricol, cel puțin variația sezonieră; pentru un utilaj, mai multe regimuri de turație și sarcină; pentru un dispozitiv purtabil, perioade active și inactive; pentru RF ambiental, mai multe poziții și momente ale zilei.

O sursă cu putere medie mare, dar cu întreruperi lungi, poate necesita o stocare mai mare decât o sursă modestă, dar disponibilă continuu.

10.3.3 Alegerea și combinarea surselor

Figura 10.3 prezintă un flux simplificat pentru alegerea sursei.

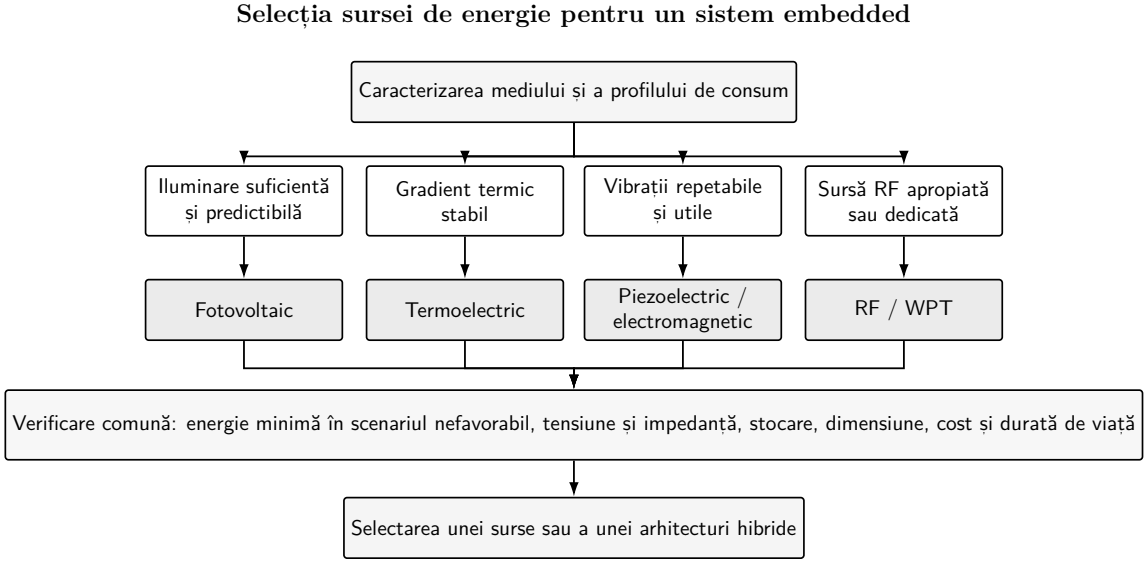


Figura 10.3 Selecția sursei de energie în funcție de mediul aplicației.

O sursă este adecvată dacă:

$$E_{h,\min} \geq E_{c,\max} + E_{\text{rezervă}}, \tag{10.11}$$

pentru intervalul de proiectare și scenariul nefavorabil considerat.

În practică, această condiție poate fi satisfăcută și prin combinarea mai multor surse:

$$P_h(t) = \sum_{j=1}^{N} P_{h,j}(t). \tag{10.12}$$

Exemple de arhitecturi hibride sunt fotovoltaic și termoelectric, fotovoltaic și vibrații, vibrații și diferență de temperatură, sau RF și fotovoltaic interior.

Utilizarea mai multor surse crește disponibilitatea, dar introduce mai multe convertoare, circuite suplimentare de adaptare, control mai complex și cost și dimensiune mai mari.

Alegerea trebuie realizată prin compararea energiei suplimentare cu resursele necesare integrării.

În blocul următor sunt analizate conversia fotovoltaică, generatoarele termoelectrice, recoltarea energiei din vibrații și sistemele RF.

10.4 Conversia fotovoltaică

Conversia fotovoltaică reprezintă una dintre cele mai utilizate metode de alimentare a sistemelor embedded autonome. Celulele fotovoltaice transformă direct energia luminoasă în energie electrică, fără componente mecanice în mișcare.

În aplicațiile embedded sunt utilizate atât celule destinate iluminării exterioare, cât și celule optimizate pentru iluminare artificială. Alegerea trebuie realizată în funcție de spectrul sursei luminoase, nivelul de iluminare, suprafața disponibilă și tensiunea necesară sistemului.

Puterea electrică furnizată poate fi estimată inițial prin:

$$P_{\mathrm{PV}} = GA\eta_{\mathrm{PV}}, \tag{10.13}$$

unde G este iradierea incidentă, A este suprafața activă, iar η_{PV} este randamentul conversiei fotovoltaice.

Relația este utilă pentru o dimensionare preliminară, dar randamentul nu este constant. Acesta depinde de temperatură, spectrul luminii, punctul de funcționare și tehnologia celulei.

10.4.1 Modelul celulei și caracteristica tensiune–curent

O celulă fotovoltaică poate fi reprezentată printr-o sursă de curent, o diodă și elemente parazite. Modelul simplificat evidențiază faptul că sursa nu se comportă ca o sursă ideală de tensiune.

Parametrii principali sunt tensiunea în gol V_{OC}, curentul de scurtcircuit I_{SC}, tensiunea la putere maximă V_{MPP} și curentul la putere maximă I_{MPP}.

Puterea furnizată la un punct de funcționare este:

$$P_{\mathrm{PV}} = V_{\mathrm{PV}} I_{\mathrm{PV}}. \tag{10.14}$$

La borne deschise:

$$I_{\mathrm{PV}} = 0, \qquad V_{\mathrm{PV}} = V_{\mathrm{OC}},$$

iar în scurtcircuit:

$$V_{\mathrm{PV}} = 0, \qquad I_{\mathrm{PV}} = I_{\mathrm{SC}}.$$

În ambele situații, puterea transmisă sarcinii este nulă. Puterea maximă este obținută într-un punct intermediar al caracteristicii.

Figura 10.4 prezintă caracteristicile I–V și P–V.

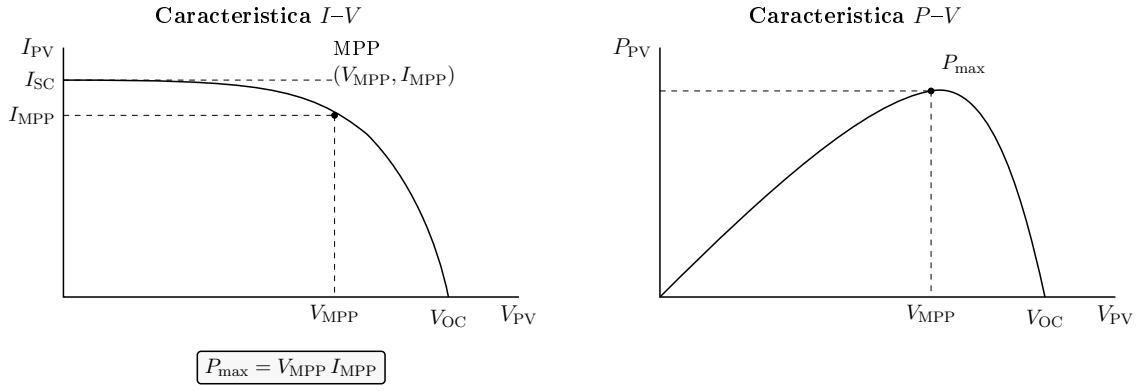


Figura 10.4 Caracteristicile unei celule fotovoltaice și punctul de putere maximă.

Factorul de umplere poate fi definit prin:

$$FF = \frac{V_{\mathrm{MPP}} I_{\mathrm{MPP}}}{V_{\mathrm{OC}} I_{\mathrm{SC}}}. \tag{10.15}$$

Acesta oferă o indicație privind forma caracteristicii și performanța electrică a celulei.

10.4.2 Punctul de putere maximă

Punctul de putere maximă se modifică odată cu iluminarea și temperatura. Conectarea directă a panoului la baterie sau la sarcină nu garantează funcționarea în acest punct.

Un circuit MPPT, denumit *Maximum Power Point Tracker*, modifică impedanța aparentă văzută de panou astfel încât acesta să funcționeze în apropierea punctului optim.

Condiția matematică pentru punctul de putere maximă este:

$$\frac{dP_{\mathrm{PV}}}{dV_{\mathrm{PV}}} = 0. \tag{10.16}$$

Deoarece:

$$P_{\mathrm{PV}} = V_{\mathrm{PV}} I_{\mathrm{PV}},$$

rezultă:

$$I_{\mathrm{PV}} + V_{\mathrm{PV}} \frac{dI_{\mathrm{PV}}}{dV_{\mathrm{PV}}} = 0. \tag{10.17}$$

Metodele MPPT uzuale includ perturbare și observare, conductanță incrementală, menținerea unei fracții din V_{OC} sau control bazat pe un model al sursei.

Pentru nodurile cu puteri foarte mici, consumul algoritmului și al circuitului MPPT trebuie să fie mai mic decât energia suplimentară obținută. În unele aplicații, o metodă simplificată poate fi mai eficientă energetic decât un algoritm complex.

10.4.3 Dimensionarea sursei fotovoltaice

Dimensionarea trebuie realizată pentru intervalul cu producția minimă, nu numai pentru condițiile nominale.

Energia zilnică utilă poate fi estimată prin:

$$E_{\mathrm{PV,zi}} = P_{\mathrm{PV,nom}} H_{\mathrm{echiv}} \eta_{\mathrm{sistem}}, \tag{10.18}$$

unde $P_{\mathrm{PV,nom}}$ este puterea nominală a panoului, H_{echiv} este numărul echivalent de ore la putere nominală, iar η_{sistem} include pierderile de conversie și stocare.

Puterea minimă necesară a panoului este:

$$P_{\mathrm{PV,min}} = \frac{E_{\mathrm{consum,zi}}}{H_{\mathrm{echiv,min}} \eta_{\mathrm{sistem}}}. \tag{10.19}$$

Se consideră un nod care consumă:

$$E_{\text{consum,zi}} = 24\,\text{mWh}.$$

Pentru:

$$H_{\text{echiv,min}} = 2\,\text{h}, \qquad \eta_{\text{sistem}} = 0{,}70,$$

rezultă:

$$\begin{aligned} P_{\text{PV,min}} &= \frac{24\,\text{mWh}}{2\,\text{h} \cdot 0{,}70} \\ &\approx 17{,}1\,\text{mW}. \end{aligned} \tag{10.20}$$

În practică se introduce o rezervă pentru umbrire, murdărire, îmbătrânire și variațiile meteorologice. Alegerea unui panou de exemplu de două sau trei ori mai mare decât valoarea minimă poate fi justificată de scenariul aplicației, dar nu trebuie realizată fără o analiză energetică.

10.5 Conversia termoelectrică și mecanică

Energia termică și energia mecanică sunt surse importante în mediile industriale. Conductele, motoarele, compresoarele și structurile aflate în mișcare pot furniza energie locală pentru senzori și sisteme de monitorizare.

Ambele tehnologii necesită o caracterizare atentă a mediului. Un generator termoelectric produce energie numai dacă se menține o diferență de temperatură, iar un convertor de vibrații este eficient numai într-un domeniu adecvat de frecvență și amplitudine [68, 74].

10.5.1 Generatorul termoelectric

Conversia termoelectrică utilizează efectul Seebeck. O diferență de temperatură aplicată unui ansamblu de materiale produce o tensiune electrică.

Pentru un generator termoelectric complet:

$$V_{\text{OC}} = S_{\text{echiv}} \Delta T, \tag{10.21}$$

unde S_{echiv} este coeficientul Seebeck echivalent al modulului, iar $\Delta T = T_H - T_C$ este diferența dintre fețele caldă și rece.

Coeficientul S_{echiv} depinde de materialele, numărul și conexiunea cuplurilor termoelectrice. El nu trebuie confundat cu valoarea unui singur material semiconductor.

Figura 10.5 prezintă structura energetică a unui generator termoelectric.

Dacă generatorul este modelat prin tensiunea V_{OC} și rezistența internă R_i, puterea transmisă unei sarcini R_L este:

$$P_L = \frac{V_{\text{OC}}^2 R_L}{(R_i + R_L)^2}. \tag{10.22}$$

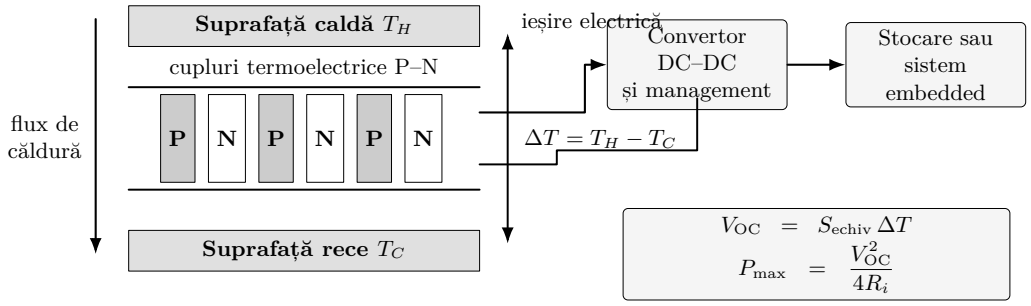


Figura 10.5 Conversia unei diferențe de temperatură printr-un generator termoelectric.

Puterea maximă este obținută în modelul simplificat pentru:

$$R_L = R_i.$$

Rezultă:

$$P_{\max} = \frac{V_{\mathrm{OC}}^2}{4R_i}. \tag{10.23}$$

Puterea reală depinde și de rezistențele termice. Dacă fața rece nu poate evacua căldura, diferența de temperatură la bornele modulului scade, chiar dacă temperaturile mediului par favorabile.

În aplicațiile cu gradienți mici sunt necesare convertoare capabile să pornească de la tensiuni foarte reduse și să acumuleze inițial energia necesară pornirii.

10.5.2 Conversia vibrațiilor

Un convertor de vibrații este reprezentat frecvent printr-un sistem masă–arc–amortizor:

$$m\ddot{x} + c\dot{x} + kx = -m\ddot{y}, \tag{10.24}$$

unde:

- x este deplasarea relativă a masei;
- y este deplasarea suportului;
- m este masa mobilă;
- k este rigiditatea;
- c include amortizarea mecanică și electrică.

Frecvența proprie neamortizată este:

$$f_0 = \frac{1}{2\pi}\sqrt{\frac{k}{m}}. \tag{10.25}$$

În apropierea rezonanței, deplasarea relativă și energia convertită pot crește. Totuși, o rezonanță foarte îngustă poate fi dezavantajoasă dacă frecvența sursei se modifică.

Figura 10.6 prezintă modelul și principalele mecanisme de conversie.

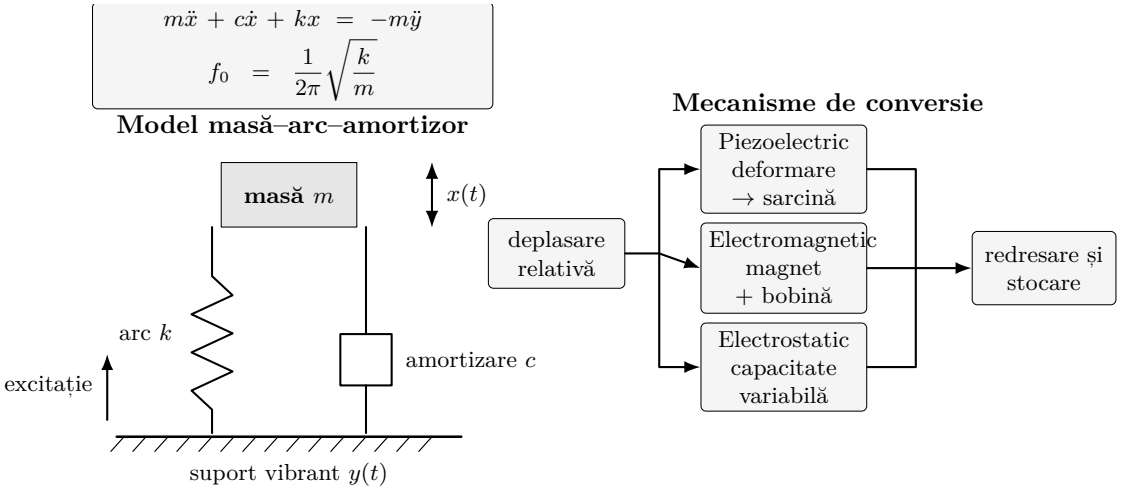


Figura 10.6 Modelul unui convertor de vibrații și mecanismele de conversie.

Principalele mecanisme sunt piezoelectric — deformarea materialului produce sarcină electrică —, electromagnetic — mișcarea relativă dintre magnet și bobină induce o tensiune — și electrostatic — vibrația modifică valoarea unei capacități.

Pentru conversia electromagnetică, tensiunea indusă este:

$$v(t) = -N\frac{d\Phi(t)}{dt}, \tag{10.26}$$

iar pentru o structură capacitivă idealizată:

$$C(x) = \frac{\varepsilon A}{d(x)}. \tag{10.27}$$

Conversia electrostatică necesită în general o polarizare inițială sau un material electret.

10.5.3 Adaptarea convertorului la sursă

Înaintea proiectării trebuie măsurată vibrația în condiții reale. Parametrii relevanți sunt spectrul de frecvență, accelerația, variația cu regimul utilajului, durata funcționării și direcția vibrației.

Pentru un motor care funcționează la:

$$n = 1500\,\text{rot/min},$$

frecvența fundamentală de rotație este:

$$f_{\text{rot}} = \frac{n}{60} = 25\,\text{Hz}. \tag{10.28}$$

Aceasta nu este obligatoriu singura componentă semnificativă. Spectrul poate conține armonici, frecvențe asociate rulmenților și componente structurale.

Metodele de creștere a domeniului util includ mai multe rezonatoare, reglarea mecanică a rigidității, structuri neliniare, conversie de frecvență și adaptarea electronică a sarcinii.

Creșterea lățimii de bandă poate reduce amplitudinea maximă obținută la o singură frecvență. Proiectarea reprezintă astfel un compromis între puterea maximă și robustețea la variația sursei.

10.6 Recoltarea energiei RF

Un sistem RF Energy Harvesting captează energia electromagnetică printr-o antenă și o transformă într-o tensiune continuă.

Trebuie diferențiate două situații: recoltarea energiei RF ambientale și transferul intenționat de energie de la un emițător dedicat.

În primul caz, puterea disponibilă este dependentă de emițătoarele existente și poate varia considerabil. În al doilea caz, emițătorul, antenele și distanța sunt proiectate pentru alimentarea receptorului.

10.6.1 Puterea recepționată

În condiții ideale de propagare în spațiu liber, în zona de câmp îndepărtat, puterea recepționată poate fi estimată prin ecuația Friis:

$$P_R = P_T G_T G_R \left(\frac{\lambda}{4\pi d}\right)^2, \tag{10.29}$$

unde P_T este puterea transmisă, G_T și G_R sunt câștigurile antenelor, λ este lungimea de undă, iar d este distanța.

În formă logaritmică:

$$P_R[\text{dBm}] = P_T[\text{dBm}] + G_T[\text{dBi}] + G_R[\text{dBi}] - L_{\text{cale}}[\text{dB}]. \tag{10.30}$$

Ecuația Friis nu descrie complet propagarea în interiorul clădirilor, în apropierea obiectelor sau în zona de câmp apropiat. Reflexiile, absorbția, polarizarea și orientarea antenelor pot modifica semnificativ puterea recepționată.

10.6.2 Rectenna și conversia RF–DC

Ansamblul format din antenă și redresor este denumit *rectenna*. Arhitectura include, în ordine, antena, rețeaua de adaptare a impedanței, redresorul, filtrul și circuitul de stocare sau convertorul DC–DC.

Figura 10.7 prezintă lanțul de conversie.

Puterea continuă obținută este:

$$P_{\text{DC}} = \eta_{\text{RF–DC}} P_R. \tag{10.31}$$

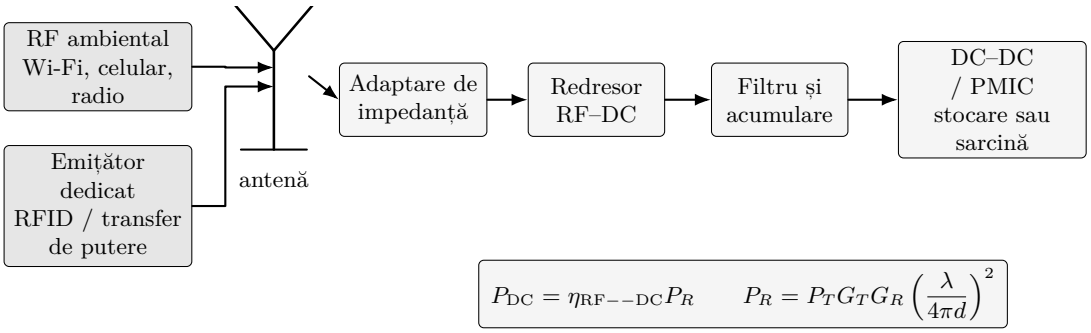


Figura 10.7 Arhitectura unui sistem de conversie RF–DC.

Randamentul $\eta_{\mathrm{RF–DC}}$ depinde de nivelul puterii recepționate, frecvență, impedanța sarcinii, pragurile dispozitivelor de redresare, adaptarea antenei și tensiunea de ieșire.

La niveluri foarte mici de putere, randamentul redresorului poate scădea semnificativ. Din acest motiv, eficiența măsurată la o putere ridicată nu poate fi extrapolată automat pentru energia RF ambientală.

10.6.3 Energie ambientală și transfer dedicat

Energia RF ambientală este adecvată în special pentru acumularea lentă a energiei, sisteme cu funcționare intermitentă, senzori cu consum foarte redus și aplicații cu backscatter.

Transferul dedicat este utilizat în RFID pasiv, NFC, senzori interogați de un cititor, unele dispozitive implantabile și alimentarea controlată a unor noduri fără baterie.

Un dispozitiv RFID pasiv nu transmite prin generarea activă a unei unde radio proprii. El modifică impedanța antenei și modulează semnalul reflectat către cititor, reducând energia necesară comunicației.

Pentru sisteme ambientale, intervalul dintre două operații poate fi estimat prin:

$$T_{\text{încărcare}} \geq \frac{E_{\text{operație}}}{P_{\mathrm{DC,mediu}}}. \tag{10.32}$$

Dacă o măsurare și o transmisie necesită:

$$E_{\text{operație}} = 1\,\mathrm{mJ},$$

iar puterea medie obținută este:

$$P_{\mathrm{DC,mediu}} = 10\,\mu\mathrm{W},$$

timpul minim ideal este:

$$T_{\text{încărcare}} \geq 100\,\mathrm{s}.$$

În practică trebuie incluse pierderile, consumul în repaus și energia minimă necesară pornirii.

Tabela 10.2 Compararea tehnologiilor analizate

Tehnologie	Parametru critic	Avantaj	Limitare
Fotovoltaică	Iluminare și suprafață	Putere relativ ridicată	Producție variabilă
Termoelectrică	Diferență de temperatură	Fără elemente mobile	Putere redusă la gradient mic
Vibrații	Frecvență și accelerație	Potrivită pentru utilaje	Sensibilitate la regimul mecanic
RF ambiental	Densitatea câmpului	Fără contact cu sursa	Putere recepționată redusă
RF dedicat	Distanță și orientare	Funcționare fără baterie	Necesită emițător dedicat

Alegerea tehnologiei nu trebuie realizată pe baza puterii nominale a convertorului, ci pe baza energiei utile obținute în condițiile reale ale aplicației.

În blocul următor vor fi analizate circuitele de management al puterii, pornirea de la tensiuni reduse, modelarea profilului de consum și dimensionarea stocării.

10.7 Conversia și managementul puterii

Sursele utilizate pentru Energy Harvesting nu pot alimenta, în general, direct un sistem embedded. Tensiunea și curentul furnizate variază cu iluminarea, temperatura, vibrațiile sau intensitatea câmpului electromagnetic, în timp ce microcontrolerul, senzorii și transceiverul necesită o tensiune stabilă.

Între convertorul energetic și consumator este introdus un circuit de management al puterii, denumit frecvent PMIC (*Power Management Integrated Circuit*). Acesta poate îndeplini mai multe funcții: redresarea tensiunii produse de sursă, adaptarea punctului de funcționare, conversia DC–DC, controlul încărcării, protejarea elementului de stocare, furnizarea unei tensiuni stabile către consumator și monitorizarea energiei disponibile.

Figura 10.8 prezintă lanțul energetic complet.

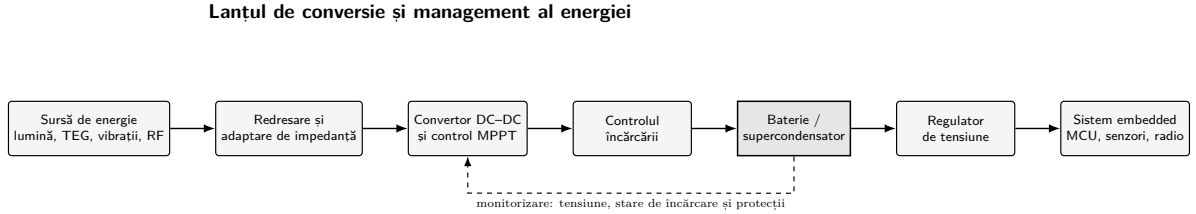


Figura 10.8 Lanțul de conversie și management al energiei într-un sistem autonom.

Randamentul global al lanțului este:

$$\eta_{\text{global}} = \eta_{\text{redresare}}\eta_{\text{DC–DC}}\eta_{\text{încărcare}}\eta_{\text{regulator}}. \tag{10.33}$$

Chiar dacă fiecare etapă are un randament ridicat, produsul acestora poate reduce semnificativ energia disponibilă consumatorului.

10.7.1 Redresare, conversie DC–DC și pornire la tensiune redusă

Sursele piezoelectrice, electromagnetice și RF produc tensiuni alternative. Acestea trebuie redresate înaintea încărcării bateriei sau supercondensatorului.

Redresarea poate fi realizată prin punte de diode, redresor sincron, multiplicator de tensiune sau circuit activ cu pierderi reduse.

La puteri foarte mici, căderea de tensiune pe diode poate reprezenta o parte importantă din tensiunea disponibilă. Redresoarele sincrone utilizează tranzistoare controlate pentru reducerea acestor pierderi, dar necesită circuite suplimentare de comandă.

După redresare, tensiunea este adaptată printr-un convertor *boost*, dacă tensiunea trebuie ridicată, *buck*, dacă tensiunea trebuie redusă, sau *buck–boost*, dacă tensiunea sursei poate fi mai mică sau mai mare decât tensiunea de ieșire.

Pentru un convertor ideal:

$$P_{\text{intrare}} = P_{\text{ieșire}}. \tag{10.34}$$

Într-un sistem real:

$$P_{\text{ieșire}} = \eta_{\text{DC–DC}} P_{\text{intrare}}. \tag{10.35}$$

La puteri de ordinul microwaților, consumul propriu al circuitului devine important. Puterea netă disponibilă este:

$$P_{\text{net}} = \eta_{\text{DC–DC}} P_{\text{sursă}} - P_{\text{PMIC}}. \tag{10.36}$$

Recoltarea este utilă numai dacă:

$$P_{\text{net}} > 0. \tag{10.37}$$

O dificultate specifică este pornirea la rece, denumită *cold start*. La început, elementul de stocare este descărcat, iar PMIC-ul trebuie să pornească folosind exclusiv energia furnizată instantaneu de sursă.

Figura 10.9 prezintă diferența dintre pornirea la rece și funcționarea normală.

Circuitul poate utiliza două căi: o cale de pornire, capabilă să funcționeze de la tensiuni foarte mici, și o cale principală, cu randament mai ridicat după încărcarea stocării.

Parametrii importanți ai unui PMIC sunt tensiunea minimă de pornire, tensiunea minimă de funcționare după pornire, curentul propriu, randamentul la sarcini reduse, curentul maxim de încărcare și compatibilitatea cu elementul de stocare.

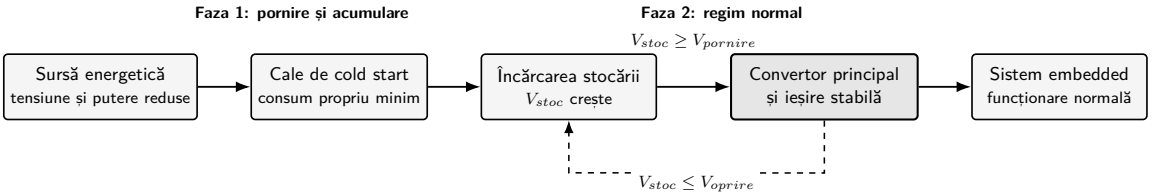


Figura 10.9 Pornirea la rece și funcționarea normală a circuitului de management energetic.

10.7.2 MPPT și adaptarea impedanței

Puterea extrasă dintr-o sursă depinde de punctul său de funcționare. Sarcina conectată prin PMIC trebuie adaptată astfel încât sursa să furnizeze o putere apropiată de valoarea maximă.

Pentru o sursă caracterizată prin tensiunea V_s și curentul I_s:

$$P_s = V_s I_s. \tag{10.38}$$

Circuitul MPPT modifică impedanța aparentă prezentată sursei:

$$Z_{\text{in}} = \frac{V_s}{I_s}. \tag{10.39}$$

Pentru un generator termoelectric modelat printr-o sursă Thévenin, transferul maxim de putere este obținut, în modelul simplificat, atunci când:

$$R_{\text{sarcină}} = R_{\text{internă}}. \tag{10.40}$$

În cazul celulelor fotovoltaice, punctul optim variază cu iluminarea și temperatura. Pentru sursele piezoelectrice și RF, sarcina optimă depinde și de frecvență, redresor și elementele reactive.

Adaptarea poate fi realizată prin:

- controlul factorului de umplere al convertorului;
- comutarea între mai multe impedanțe;
- măsurarea periodică a tensiunii în gol;
- algoritmi de perturbare și observare;
- control bazat pe model.

MPPT-ul nu trebuie evaluat numai prin randamentul conversiei. Trebuie inclus și consumul circuitului de măsurare și control:

$$P_{\text{câștig}} = P_{\text{MPPT}} - P_{\text{fără MPPT}} - P_{\text{control}}. \tag{10.41}$$

Pentru surse foarte slabe, un algoritm simplificat poate produce mai multă energie netă decât un control complex.

10.7.3 Praguri energetice și funcționare intermitentă

Unele sisteme nu recoltează suficientă putere pentru funcționare continuă. Energia este acumulată până la atingerea unui prag, după care sistemul execută o operație și revine în starea de încărcare.

Secvența este:

$$\text{ncrcare} \rightarrow \text{pornire} \rightarrow \text{msurare} \rightarrow \text{transmisie} \rightarrow \text{oprire}.$$

Pentru evitarea comutărilor repetate se utilizează două praguri:

$$V_{\text{pornire}} > V_{\text{oprire}}. \tag{10.42}$$

Energia utilizabilă dintr-un condensator între cele două praguri este:

$$E_{\text{util}} = \frac{1}{2} C \left(V_{\text{pornire}}^2 - V_{\text{oprire}}^2 \right). \tag{10.43}$$

Operația poate fi executată dacă:

$$E_{\text{util}} \geq E_{\text{operație}} + E_{\text{rezervă}}. \tag{10.44}$$

Funcționarea intermitentă necesită ca software-ul să tolereze întreruperea alimentării. Datele importante trebuie salvate înainte de oprire, iar operațiile asupra memoriei nevolatile trebuie proiectate astfel încât să nu rămână într-o stare inconsistentă.

10.8 Modelarea consumului și a balanței energetice

Un sistem autonom trebuie analizat pe baza energiei consumate de fiecare stare de funcționare. Puterea maximă este importantă pentru dimensionarea regulatorului și a stocării, în timp ce puterea medie determină energia necesară pe termen lung.

Modelarea trebuie să includă microcontrolerul, senzorii, memoria, transceiverul, convertoarele, curenții de pierdere și autodescărcarea stocării.

10.8.1 Profilul stărilor de funcționare

Un nod senzorial funcționează de regulă în mai multe stări:

1. sleep;
2. trezire;
3. măsurare;
4. procesare;
5. transmisie;
6. revenire în sleep.

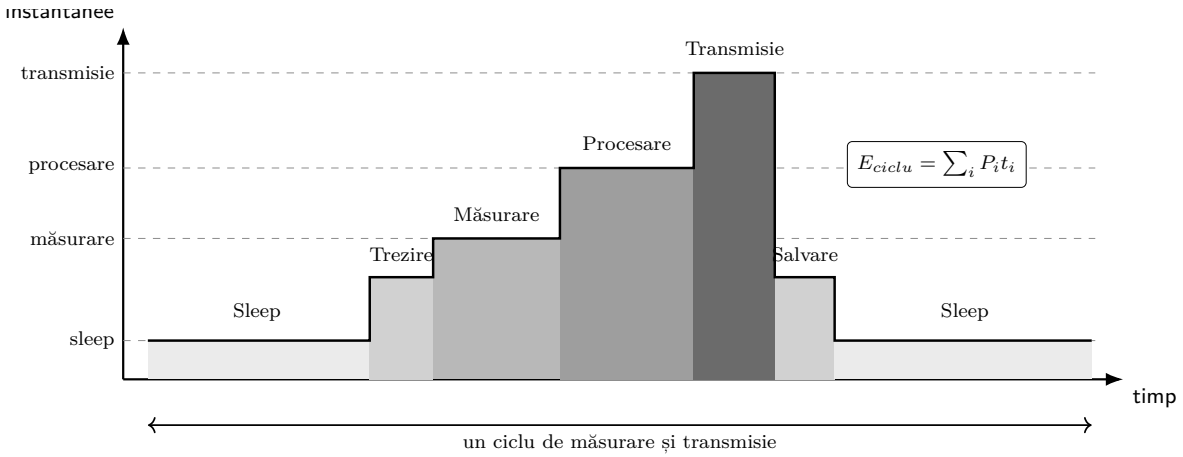


Figura 10.10 Profilul de putere al unui ciclu de măsurare și transmisie.

Figura 10.10 prezintă un profil tipic.

Energia consumată în starea i este:

$$E_i = P_i t_i. \tag{10.45}$$

Energia unui ciclu complet este:

$$E_{\text{ciclu}} = \sum_{i=1}^{N} P_i t_i. \tag{10.46}$$

Chiar dacă transmisia are o durată redusă, aceasta poate domina consumul ciclului din cauza puterii ridicate.

10.8.2 Puterea și energia medie

Puterea medie a unui ciclu cu perioada T_C este:

$$P_{\text{medie}} = \frac{\sum_{i=1}^{N} P_i t_i}{T_C}. \tag{10.47}$$

Se consideră un nod cu perioada de 600 s și profilul din Tabelul 10.3.

Tabela 10.3 Exemplu de profil energetic pentru un nod senzorial

Stare	Putere	Durată	Energie
Sleep	8 μW	598 s	4,784 mJ
Măsurare	5 mW	1 s	5 mJ
Procesare	12 mW	0,5 s	6 mJ
Transmisie	120 mW	0,5 s	60 mJ

Energia totală este:

$$E_{\text{ciclu}} = 75{,}784\,\text{mJ}. \tag{10.48}$$

Puterea medie rezultă:

$$\begin{aligned} P_{\text{medie}} &= \frac{75{,}784\,\text{mJ}}{600\,\text{s}} \\ &\approx 126{,}3\,\mu\text{W}. \end{aligned} \tag{10.49}$$

Energia zilnică este:

$$\begin{aligned} E_{\text{zi}} &= P_{\text{medie}} \cdot 24\,\text{h} \\ &\approx 3{,}03\,\text{mWh}. \end{aligned} \tag{10.50}$$

Deși nodul petrece 99,7% din timp în sleep, transmisia reprezintă cea mai mare parte a energiei active. Reducerea numărului de transmisii poate fi mai eficientă decât optimizarea suplimentară a procesării.

10.8.3 Neutralitatea energetică în timp

Compararea energiilor medii este necesară, dar nu suficientă. Evoluția stocării trebuie analizată pe intervale discrete.

Pentru intervalul k:

$$E_s[k+1] = E_s[k] + \eta_c E_h[k] - \frac{E_c[k]}{\eta_d} - E_l[k], \tag{10.51}$$

unde η_c este randamentul încărcării, η_d este randamentul descărcării, $E_h[k]$ este energia recoltată, $E_c[k]$ este energia consumată, iar $E_l[k]$ reprezintă pierderile.

Valoarea trebuie limitată la intervalul fizic:

$$0 \leq E_s[k] \leq E_{\max}. \tag{10.52}$$

Figura 10.11 prezintă evoluția energiei pe mai multe intervale de producție și consum.

Pentru verificarea proiectului trebuie analizate nivelul minim al energiei stocate, numărul zilelor fără producție suficientă, energia pierdută când stocarea este plină, perioadele în care serviciul trebuie redus și timpul necesar recuperării după descărcare.

Managementul energetic adaptiv poate modifica perioada de măsurare astfel încât consumul să urmeze energia disponibilă [48].

10.9 Stocarea energiei

Elementul de stocare separă temporal producerea energiei de consum. Acesta trebuie ales pe baza energiei totale necesare, a vârfurilor de putere, a numărului de cicluri și a condițiilor

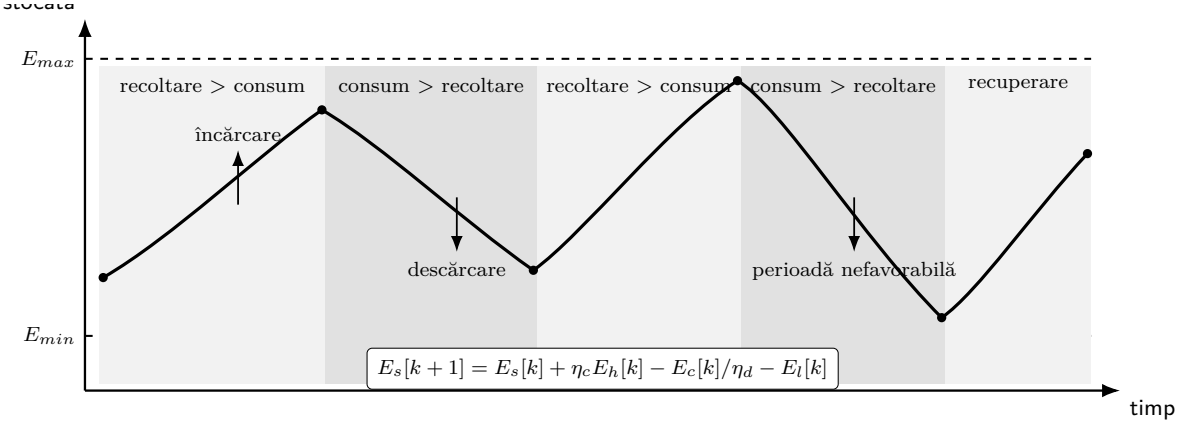


Figura 10.11 Evoluția energiei stocate în condiții variabile de recoltare și consum.

de mediu.

Principalele opțiuni sunt bateriile reîncărcabile, supercondensatorii și combinațiile hibride.

10.9.1 Baterii și supercondensatori

Energia nominală a unei baterii poate fi estimată prin:

$$E_B \approx V_{\text{nom}} C_B, \tag{10.53}$$

unde capacitatea C_B este exprimată în amper-oră.

Energia efectiv utilizabilă este mai redusă:

$$E_{B,\text{util}} = V_{\text{nom}} C_B DOD \eta_d, \tag{10.54}$$

unde DOD este adâncimea permisă de descărcare.

Pentru un supercondensator, energia utilizabilă între două tensiuni este:

$$E_{SC,\text{util}} = \frac{1}{2} C \left(V_{\text{max}}^2 - V_{\text{min}}^2 \right). \tag{10.55}$$

Alegerea trebuie să includă temperatura, îmbătrânirea și curentul solicitat de transceiver.

10.9.2 Dimensionarea rezervei energetice

Dacă sistemul trebuie să funcționeze N_A zile fără recoltare, energia minimă necesară este:

$$E_{\text{rezervă}} = N_A E_{\text{consum,zi}}. \tag{10.56}$$

Capacitatea minimă a bateriei este:

$$C_{B,\text{min}} = \frac{N_A E_{\text{consum,zi}}}{V_{\text{nom}} DOD \eta_d}. \tag{10.57}$$

Pentru:

Tabela 10.4 Compararea bateriilor și supercondensatorilor

Caracteristică	Baterie	Supercondensator
Densitate energetică	Ridicată	Redusă
Densitate de putere	Moderată	Ridicată
Număr de cicluri	Limitat	Foarte ridicat
Autodescărcare	Relativ redusă	Mai ridicată
Tensiune la descărcare	Relativ stabilă	Scade continuu
Utilizare tipică	Stocare pe termen lung	Vârfuri de putere și cicluri dese

$$E_{\text{consum,zi}} = 24\,\text{mWh},$$

$$N_A = 5, \qquad V_{\text{nom}} = 3{,}7\,\text{V},$$

$$DOD = 0{,}8, \qquad \eta_d = 0{,}9,$$

rezultă:

$$\begin{aligned} C_{B,\min} &= \frac{5 \cdot 24\,\text{mWh}}{3{,}7\,\text{V} \cdot 0{,}8 \cdot 0{,}9} \\ &\approx 45\,\text{mAh}. \end{aligned} \tag{10.58}$$

În practică se selectează o capacitate mai mare pentru a include îmbătrânirea bateriei, reducerea capacității la temperaturi joase, autodescărcarea, creșterea consumului comunicației și incertitudinea modelului energetic.

Pentru un supercondensator, capacitatea minimă este:

$$C_{\min} = \frac{2E_{\text{necesar}}}{\eta_d \left(V_{\max}^2 - V_{\min}^2\right)}. \tag{10.59}$$

10.9.3 Arhitecturi hibride

O arhitectură hibridă utilizează bateria pentru energia pe termen lung și supercondensatorul pentru vârfurile de curent.

Figura 10.12 prezintă această structură.

În timpul transmisiei:

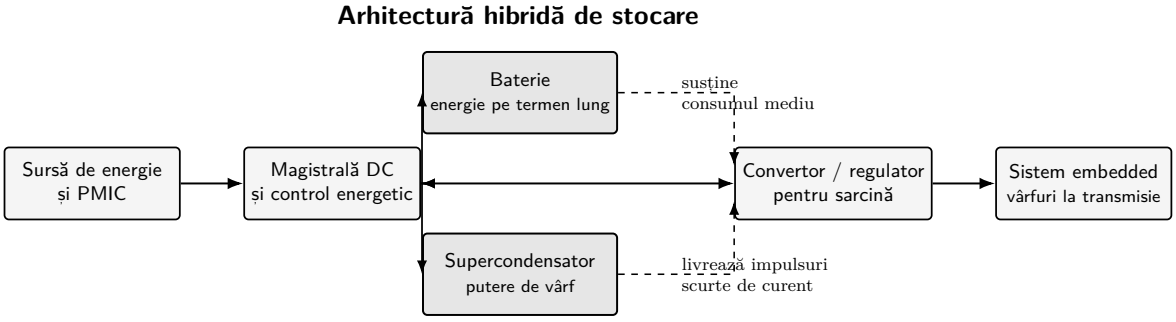


Figura 10.12 Arhitectură hibridă cu baterie și supercondensator.

$$P_{\text{sarcină}} = P_{\text{baterie}} + P_{\text{supercondensator}}.$$

Supercondensatorul reduce curentul de vârf furnizat de baterie și poate diminua căderea tensiunii, încălzirea internă, solicitarea electrochimică și degradarea produsă de impulsurile repetate.

Circuitul de control trebuie să stabilească prioritatea încărcării, tensiunea maximă a fiecărui element, momentul utilizării supercondensatorului, protecția la descărcare profundă și separarea curenților dintre elemente.

Arhitectura hibridă este justificată atunci când consumatorul are o putere medie redusă, dar solicită impulsuri scurte și repetate de curent.

În blocul următor vor fi integrate sursa, stocarea, senzorii și comunicația într-un nod IoT autonom energetic, urmate de dimensionarea completă a unui sistem alimentat fotovoltaic.

10.10 Noduri senzoriale autonome energetic

Un nod senzorial autonom energetic combină o sursă de Energy Harvesting, un circuit de management al puterii, un element de stocare și o platformă embedded cu consum redus.

Obiectivul nu este numai reducerea consumului, ci menținerea serviciului aplicației în limitele energiei disponibile. În funcție de condițiile de mediu, nodul poate modifica frecvența măsurătorilor, procesarea locală și strategia de comunicație.

Principalele componente sunt convertorul energetic, circuitul PMIC, bateria sau supercondensatorul, microcontrolerul, senzorii, transceiverul wireless și software-ul de management energetic.

10.10.1 Arhitectura nodului

Figura 10.13 prezintă arhitectura generală a unui nod senzorial alimentat prin Energy Harvesting.

Lanțul energetic poate fi descris prin:

$$\text{surs} \rightarrow \text{PMIC} \rightarrow \text{stocare} \rightarrow \text{sarcin}.$$

Lanțul informațional este:

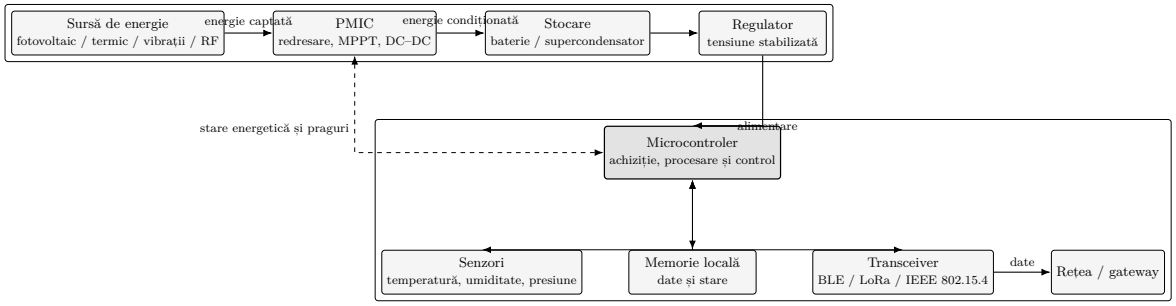


Figura 10.13 Arhitectura unui nod senzorial autonom energetic.

$$\text{senzori} \rightarrow \text{microcontroler} \rightarrow \text{transceiver} \rightarrow \text{rețea}.$$

Cele două lanțuri sunt interdependente. O decizie software, precum creșterea frecvenței transmisiilor, modifică direct profilul energetic al sistemului.

Circuitul PMIC poate furniza și informații către microcontroler, precum tensiunea bateriei, starea încărcării, prezența sursei, atingerea unui prag energetic sau detectarea unei tensiuni insuficiente.

Pe baza acestor informații, software-ul poate selecta modul de funcționare.

10.10.2 Funcționarea intermitentă și factorul de activitate

Majoritatea nodurilor autonome nu funcționează permanent în regim activ. Un ciclu tipic conține:

1. stare sleep;
2. trezire;
3. activarea senzorilor;
4. măsurare;
5. procesare;
6. transmisie;
7. revenire în sleep.

Factorul de activitate este:

$$D = \frac{T_{\text{activ}}}{T_{\text{ciclu}}}. \tag{10.60}$$

Pentru un nod activ timp de 5 s la fiecare 15 minute:

$$D = \frac{5}{900} \approx 0{,}00556 = 0{,}556\%. \tag{10.61}$$

Deși factorul de activitate este redus, consumul mediu nu poate fi estimat numai din această valoare. Transceiverul poate consuma de sute sau mii de ori mai mult decât microcontrolerul în sleep.

Energia unui ciclu este:

$$E_{\text{ciclu}} = E_{\text{sleep}} + E_{\text{senzori}} + E_{\text{procesare}} + E_{\text{radio}}. \tag{10.62}$$

Numărul ciclurilor zilnice pentru o perioadă T_{ciclu} este:

$$N_{\text{cicluri,zi}} = \frac{24 \cdot 3600}{T_{\text{ciclu}}}. \tag{10.63}$$

Energia zilnică devine:

$$E_{\text{zi}} = N_{\text{cicluri,zi}} E_{\text{ciclu}} + E_{\text{pierderi,zi}}. \tag{10.64}$$

10.10.3 Adaptarea serviciului la energia disponibilă

Un sistem robust nu trebuie să se oprească imediat atunci când producția energetică scade. El poate reduce gradual nivelul serviciului.

Figura 10.14 prezintă trei moduri posibile.

Modul poate fi selectat în funcție de energia stocată:

$$M(E_s) = \begin{cases} M_{\text{normal}}, & E_s \geq E_H, \\ M_{\text{economic}}, & E_L < E_s < E_H, \\ M_{\text{supraviețuire}}, & E_s \leq E_L. \end{cases} \tag{10.65}$$

Un exemplu de strategie este prezentat în Tabelul 10.5.

Tabela 10.5 Adaptarea funcționării nodului la nivelul energetic

Mod	Măsurare	Comunicație
Normal	La fiecare 15 minute	Transmiterea fiecărei măsurări
Economic	La fiecare 60 de minute	Agregarea mai multor măsurări
Supraviețuire	La fiecare 6 ore	Numai alarme sau fără transmisie

Reducerea transmisiilor este frecvent cea mai eficientă măsură. Datele pot fi stocate local și trimise ulterior, după refacerea rezervei energetice.

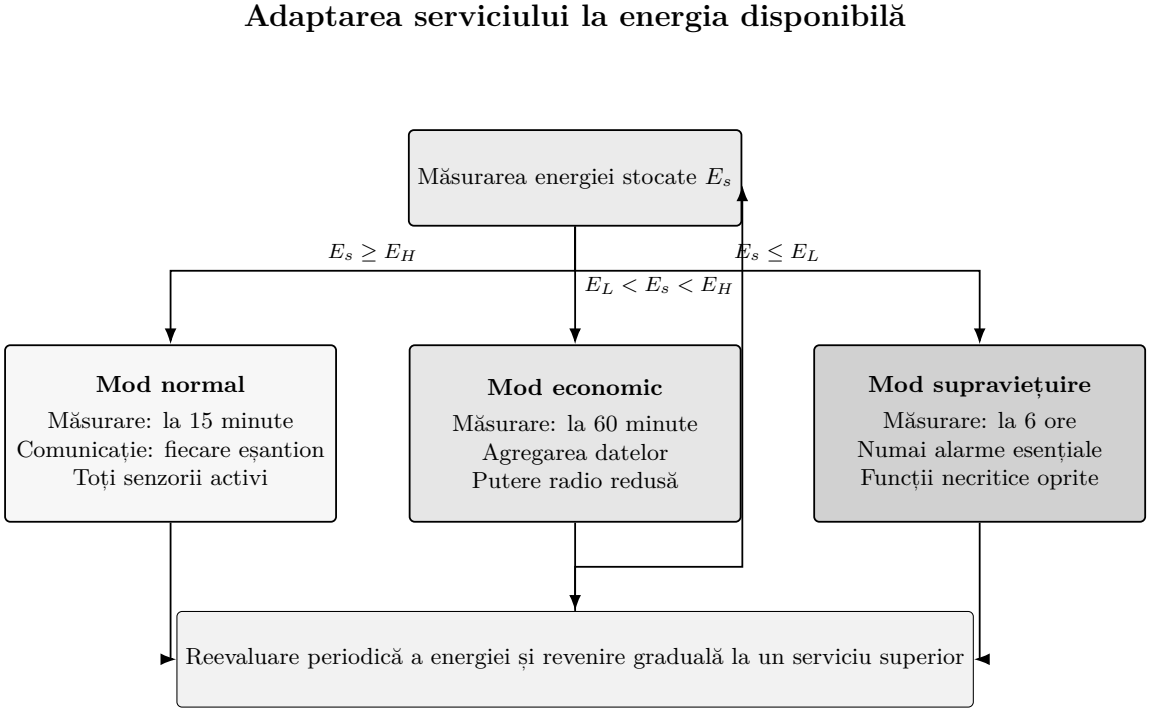


Figura 10.14 Adaptarea serviciului la energia disponibilă.

10.11 Studiu de caz: nod IoT alimentat fotovoltaic

Se proiectează un nod IoT pentru monitorizarea unei culturi agricole. Sistemul măsoară:

- temperatura și umiditatea aerului;
- umiditatea solului;
- nivelul iluminării.

Datele sunt transmise printr-o legătură radio cu rază lungă la fiecare 15 minute.

Cerințele principale sunt:

- funcționare continuă pe tot parcursul anului;
- minimum cinci zile de autonomie fără recoltare;
- alimentare fotovoltaică;
- adaptarea serviciului la nivelul bateriei;
- temperatură de funcționare compatibilă cu mediul exterior.

10.11.1 Arhitectura și profilul de consum

Figura 10.15 prezintă arhitectura nodului.

Configurația include:

- panou fotovoltaic;
- PMIC cu funcție MPPT;
- baterie reîncărcabilă;

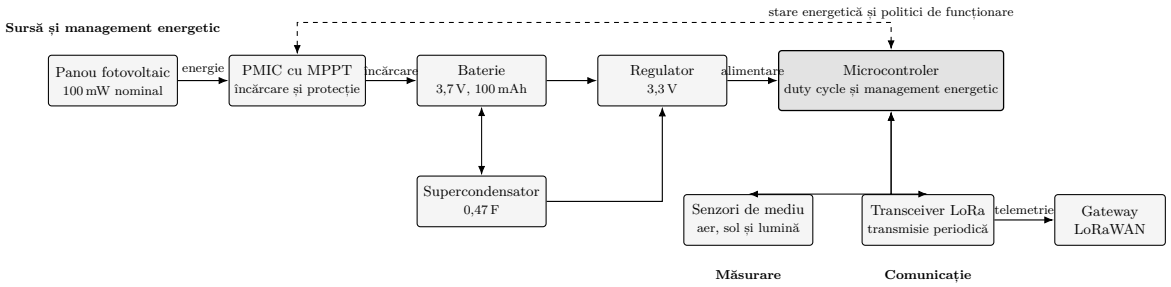


Figura 10.15 Arhitectura nodului IoT alimentat fotovoltaic.

- supercondensator pentru vârful de transmisie;
- microcontroler cu mod deep sleep;
- senzori de mediu;
- transceiver radio.

Profilul energetic al unui ciclu de 900 s este prezentat în Tabelul 10.6.

Tabela 10.6 Profilul energetic estimat al nodului IoT

Stare	Putere	Durată	Energie
Sleep	$12\,\mu$W	895 s	10,74 mJ
Măsurare	15 mW	1,5 s	22,5 mJ
Procesare	20 mW	0,5 s	10 mJ
Transmisie	165 mW	2 s	330 mJ
Control auxiliar	10 mW	1 s	10 mJ

Energia totală a unui ciclu este:

$$E_{\text{ciclu}} = 383{,}24\,\text{mJ}. \tag{10.66}$$

Numărul ciclurilor zilnice este:

$$N_{\text{cicluri,zi}} = \frac{86400}{900} = 96. \tag{10.67}$$

Energia consumată de sarcina principală este:

$$\begin{aligned} E_{\text{sarcină,zi}} &= 96 \cdot 383{,}24\,\text{mJ} \\ &\approx 36{,}79\,\text{J} \\ &\approx 10{,}22\,\text{mWh}. \end{aligned} \tag{10.68}$$

Introducând pierderile PMIC-ului, ale stocării și o rezervă de proiectare, considerăm:

$$E_{\text{proiectare,zi}} = 13{,}5\,\text{mWh}. \tag{10.69}$$

Figura 10.16 prezintă distribuția energetică.

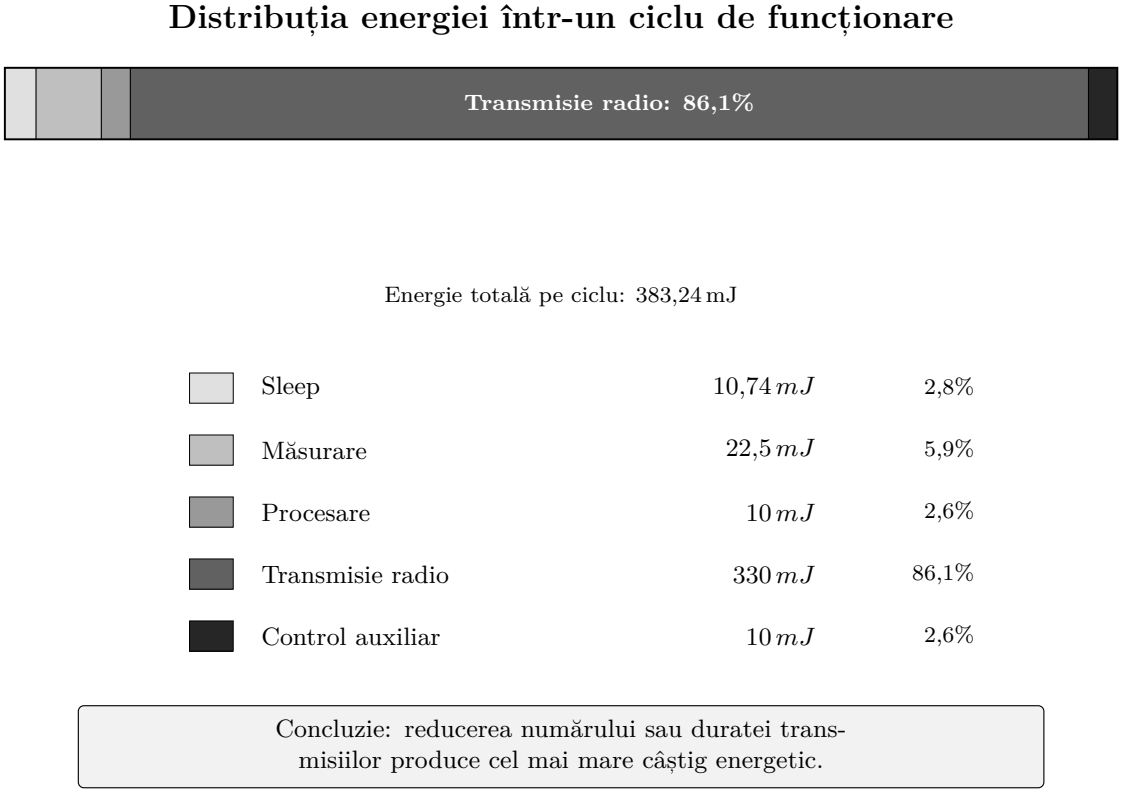


Figura 10.16 Distribuția energiei consumate într-un ciclu al nodului IoT.

Transmisia reprezintă componenta dominantă:

$$\frac{E_{\text{radio}}}{E_{\text{ciclu}}} = \frac{330}{383{,}24} \approx 86{,}1\%. \tag{10.70}$$

Prin urmare, optimizarea comunicației are un impact mai mare decât reducerea suplimentară a duratei procesării.

10.11.2 Dimensionarea panoului și a stocării

Pentru intervalul nefavorabil se consideră:

$$H_{\text{echiv,min}} = 1{,}5\,\text{h/zi}$$

și:

$$\eta_{\text{sistem}} = 0{,}65.$$

Se introduce un factor de rezervă:

$$k_R = 1{,}5.$$

Puterea minimă a panoului este:

$$P_{\mathrm{PV,min}} = \frac{k_R E_{\mathrm{proiectare,zi}}}{H_{\mathrm{echiv,min}} \eta_{\mathrm{sistem}}}. \tag{10.71}$$

Rezultă:

$$\begin{aligned} P_{\mathrm{PV,min}} &= \frac{1{,}5 \cdot 13{,}5\,\mathrm{mWh}}{1{,}5\,\mathrm{h} \cdot 0{,}65} \\ &\approx 20{,}8\,\mathrm{mW}. \end{aligned} \tag{10.72}$$

Aceasta este valoarea energetică minimă rezultată din model. Pentru a include umbrirea, murdărirea, variația orientării, îmbătrânirea și condițiile meteorologice, se poate selecta un panou nominal de:

$$P_{\mathrm{PV,selectat}} = 100\,\mathrm{mW}.$$

Energia zilnică furnizată de acesta în scenariul considerat este:

$$\begin{aligned} E_{\mathrm{PV,zi}} &= 100\,\mathrm{mW} \cdot 1{,}5\,\mathrm{h} \cdot 0{,}65 \\ &= 97{,}5\,\mathrm{mWh}. \end{aligned} \tag{10.73}$$

Aceasta depășește consumul zilnic de proiectare și oferă rezervă pentru condițiile nefavorabile.

Pentru cinci zile fără recoltare:

$$E_{\mathrm{rezervă}} = 5 \cdot 13{,}5\,\mathrm{mWh} = 67{,}5\,\mathrm{mWh}. \tag{10.74}$$

Pentru o baterie de 3,7 V, cu:

$$DOD = 0{,}7, \qquad \eta_d = 0{,}9,$$

capacitatea minimă este:

$$\begin{aligned} C_{B,\mathrm{min}} &= \frac{67{,}5\,\mathrm{mWh}}{3{,}7\,\mathrm{V} \cdot 0{,}7 \cdot 0{,}9} \\ &\approx 29\,\mathrm{mAh}. \end{aligned} \tag{10.75}$$

După includerea temperaturii, îmbătrânirii și a incertitudinilor, se poate selecta o baterie de:

$$C_{B,\text{selectat}} = 100\,\text{mAh}.$$

Această alegere este mult mai apropiată de necesarul energetic decât o baterie de ordinul amper-oră.

10.11.3 Vârful de transmisie și scenariul nefavorabil

În timpul transmisiei, curentul aproximativ la 3,3 V este:

$$I_{\text{TX}} = \frac{165\,\text{mW}}{3{,}3\,\text{V}} = 50\,\text{mA}. \tag{10.76}$$

Pentru susținerea curentului timp de 2 s, cu o scădere maximă de tensiune de 0,3 V, capacitatea necesară este:

$$C = \frac{I_{\text{TX}} T_{\text{TX}}}{\Delta V}. \tag{10.77}$$

Rezultă:

$$\begin{aligned} C &= \frac{0{,}05 \cdot 2}{0{,}3} \\ &\approx 0{,}333\,\text{F}. \end{aligned} \tag{10.78}$$

Se poate selecta un supercondensator de:

$$C_{\text{selectat}} = 0{,}47\,\text{F},$$

cu respectarea tensiunii maxime și a rezistenței serie echivalente.

În scenariul fără producție solară, energia stocată trebuie să acopere cinci zile. Dacă nivelul bateriei scade sub pragul economic, perioada de transmitere poate fi crescută de la 15 minute la o oră.

Numărul transmisiilor se reduce de la:

$$96$$

la:

$$24$$

pe zi.

Energia radio zilnică devine aproximativ un sfert din valoarea inițială. Acest mecanism prelungește autonomia și reduce probabilitatea opririi complete.

Sistemul poate utiliza următoarele praguri:

$$\text{mod} = \begin{cases} \text{normal}, & SOC \geq 60\%, \\ \text{economic}, & 30\% < SOC < 60\%, \\ \text{supraviețuire}, & SOC \leq 30\%. \end{cases} \tag{10.79}$$

În modul de supraviețuire, măsurătorile pot fi păstrate local, iar transmisia este realizată numai la apariția unei alarme sau după refacerea energiei.

Studiul de caz evidențiază următoarele principii:

- dimensionarea pornește de la energia unui ciclu;
- transmisia radio domină frecvent consumul;
- panoul se dimensionează pentru perioada nefavorabilă;
- bateria se dimensionează pentru numărul dorit de zile de autonomie;
- vârfurile de putere pot necesita un supercondensator;
- serviciul trebuie adaptat la energia disponibilă.

10.12 Întrebări și probleme propuse

10.12.1 Întrebări de verificare

1. Care sunt componentele principale ale unui nod senzorial autonom energetic?
2. Care este diferența dintre lanțul energetic și lanțul informațional al nodului?
3. Ce reprezintă factorul de activitate?
4. De ce un factor de activitate redus nu garantează automat un consum energetic redus?
5. Ce componente trebuie incluse în energia unui ciclu de funcționare?
6. Cum poate fi adaptat serviciul aplicației la energia disponibilă?
7. De ce transmisia wireless domină frecvent bugetul energetic?
8. De ce dimensionarea panoului trebuie realizată pentru o perioadă nefavorabilă?
9. Ce reprezintă numărul echivalent de ore de producție nominală?
10. Cum se dimensionează rezerva energetică pentru mai multe zile fără recoltare?
11. Care este rolul unui supercondensator conectat împreună cu bateria?
12. Ce funcții trebuie păstrate în modul energetic de supraviețuire?

10.12.2 Probleme de calcul și proiectare

1. Un nod este activ 4 s la fiecare 20 de minute. Calculați factorul de activitate.
2. Un dispozitiv consumă $20\,\mu$W timp de 595 s și 100 mW timp de 5 s. Determinați energia unui ciclu și puterea medie.

3. Un nod consumă 18 mWh/zi. Pentru $H_{\text{echiv,min}} = 2\,\text{h}$ și $\eta_{\text{sistem}} = 0{,}6$, determinați puterea minimă a panoului. Introduceți apoi o rezervă de 100%.

4. Dimensionați o baterie de 3,7 V pentru un consum de 25 mWh/zi și șapte zile fără recoltare. Considerați $DOD = 0{,}75$ și $\eta_d = 0{,}9$.

5. Un transceiver solicită 80 mA timp de 1,5 s. Calculați capacitatea necesară pentru limitarea scăderii tensiunii la 0,2 V.

6. Comparați energetic două strategii: transmiterea unei măsurări la fiecare 10 minute și agregarea a șase măsurări într-o singură transmisie pe oră.

7. Proiectați trei moduri de funcționare pentru un senzor de monitorizare industrială, în funcție de tensiunea elementului de stocare.

8. Realizați bugetul energetic pentru un nod IoT ales de student, incluzând microcontrolerul, senzorii, comunicația, PMIC-ul și pierderile de stocare.

11. Internet of Things și Rețele de Dispozitive Inteligente

11.1 Introducere în Internet of Things

Internet of Things, prescurtat IoT, descrie integrarea obiectelor fizice, senzorilor, actuatoarelor și sistemelor embedded în infrastructuri de comunicație și servicii digitale. Dispozitivele IoT colectează informații din mediul fizic, le procesează local sau la distanță și pot genera acțiuni automate asupra procesului monitorizat.

Un sistem IoT realizează astfel o legătură bidirecțională între lumea fizică și lumea digitală:

$$\text{mediu fizic} \rightarrow \text{senzori} \rightarrow \text{procesare} \rightarrow \text{rețea} \rightarrow \text{aplicație} \rightarrow \text{actuatoare}.$$

Spre deosebire de o rețea informatică tradițională, în care datele sunt introduse în principal de utilizatori, un sistem IoT colectează automat și continuu informații despre obiecte și procese reale [12, 34].

Figura 11.1 prezintă principiul general.

Un dispozitiv conectat nu reprezintă în mod automat un sistem IoT complet. Pentru a participa la un ecosistem IoT, dispozitivul trebuie să fie integrat într-un serviciu care per-

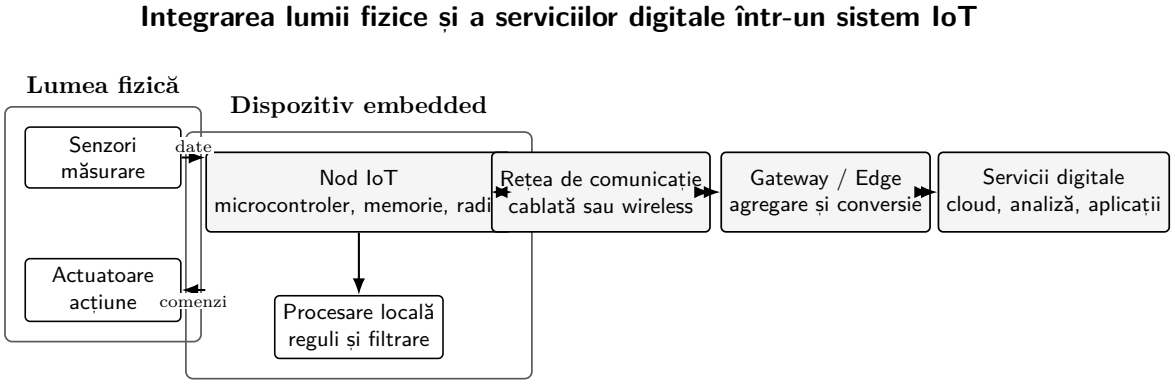


Figura 11.1 Integrarea lumii fizice și a serviciilor digitale într-un sistem IoT.

mite cel puțin identificarea dispozitivului, colectarea sau modificarea unor date, comunicarea printr-o rețea, procesarea informațiilor și administrarea pe durata exploatării.

11.1.1 De la sisteme embedded la obiecte conectate

Un sistem embedded tradițional execută o funcție bine definită și poate funcționa complet izolat. Exemple sunt un controler de temperatură, un modul de comandă pentru un motor sau un aparat electrocasnic.

Prin adăugarea conectivității, sistemul poate transmite măsurători, primi configurații, raporta erori, coopera cu alte dispozitive și fi actualizat și diagnosticat de la distanță.

Un termostat local, de exemplu, măsoară temperatura și comandă sistemul de încălzire. Un termostat IoT poate suplimentar să primească programări de la distanță, să transmită istoricul consumului și să participe la optimizarea energetică a unei clădiri.

Trecerea de la un sistem embedded izolat la un nod IoT poate fi exprimată conceptual prin:

$$\text{sistem embedded} + \text{identitate} + \text{conectivitate} + \text{servicii} = \text{nod IoT}.$$

Conectivitatea introduce însă responsabilități suplimentare: protejarea comunicațiilor, gestionarea erorilor de rețea, sincronizarea datelor, actualizarea firmware-ului și administrarea identității dispozitivului.

11.1.2 Interacțiunea dintre lumea fizică și cea digitală

Senzorii transformă mărimi fizice în date digitale. Actuatoarele realizează procesul invers, transformând comenzile digitale în acțiuni asupra mediului.

Pentru o mărime fizică $x(t)$, sistemul de achiziție generează o succesiune de eșantioane:

$$x[k] = x(kT_s), \tag{11.1}$$

unde T_s este perioada de eșantionare.

Datele pot fi filtrate și interpretate local:

$$z[k] = F\left(x[k], x[k-1], \ldots\right), \tag{11.2}$$

iar rezultatul poate genera o comandă:

$$u[k] = G\left(z[k], r[k]\right), \tag{11.3}$$

unde $r[k]$ reprezintă o referință sau o regulă stabilită de aplicație.

Într-un sistem de irigare, de exemplu:

$$\text{umiditate sol} \rightarrow \text{decizie} \rightarrow \text{deschidere electrovalv.}$$

Decizia poate fi luată direct de nod, de un gateway local, de o aplicație edge sau de un serviciu cloud.

Locul în care este executată decizia influențează latența, consumul energetic, dependența de rețea și confidențialitatea datelor.

11.1.3 Domenii și obiective

Sistemele IoT sunt utilizate acolo unde monitorizarea distribuită și automatizarea produc beneficii operaționale.

Domeniile reprezentative includ automatizări industriale, mentenanță predictivă, agricultură inteligentă, monitorizarea mediului, infrastructuri urbane, clădiri inteligente, sănătate digitală, transport și logistică și management energetic.

Obiectivele nu se limitează la colectarea unui volum mare de date. Un sistem IoT trebuie să producă informații sau acțiuni utile, precum detectarea unei anomalii, reducerea consumului energetic, optimizarea unui proces, prevenirea unei defecțiuni, creșterea siguranței sau automatizarea unei decizii repetitive.

11.2 Caracteristicile și cerințele sistemelor IoT

Sistemele IoT combină dispozitive cu resurse limitate, rețele eterogene și servicii distribuite. Proiectarea lor necesită evaluarea simultană a performanței, consumului energetic, securității și costului.

O arhitectură trebuie să funcționeze nu numai pentru un prototip cu câteva noduri, ci și în condițiile reale de exploatare, unde pot apărea:

- pierderi de pachete;
- întreruperi ale alimentării;
- variații ale calității semnalului;
- dispozitive neactualizate;
- defectarea gateway-urilor;
- creșterea numărului de noduri.

11.2.1 Scalabilitate, heterogenitate și conectivitate

Scalabilitatea reprezintă capacitatea sistemului de a suporta creșterea numărului de dispozitive, mesaje și utilizatori fără degradarea necontrolată a serviciului.

Pentru N noduri, fiecare transmițând f_m mesaje într-o unitate de timp, numărul total de mesaje este:

$$N_{\text{mesaje}} = N f_m. \tag{11.4}$$

Dacă lungimea medie a unui mesaj este L_m, volumul brut de date este:

$$D = N f_m L_m. \tag{11.5}$$

Această valoare nu include antetele protocoalelor, confirmările, retransmisiile, mesajele de administrare sau criptarea și autentificarea.

Un sistem cu mesaje mici poate genera un trafic semnificativ dacă numărul dispozitivelor este foarte mare.

Heterogenitatea rezultă din utilizarea unor dispozitive diferite: microcontrolere cu resurse limitate, gateway-uri cu sisteme de operare, servere edge, servicii cloud și aplicații web și mobile.

Aceste componente pot utiliza protocoale și formate de date diferite. Interoperabilitatea necesită interfețe bine definite și modele comune ale informațiilor.

11.2.2 Energie, latență și fiabilitate

Numeroase noduri IoT funcționează din baterii sau prin Energy Harvesting. Comunicarea radio reprezintă frecvent una dintre cele mai costisitoare operații din punct de vedere energetic.

Energia necesară transmiterii unui mesaj poate fi exprimată prin:

$$E_{\text{mesaj}} = E_{\text{trezire}} + E_{\text{procesare}} + E_{\text{TX}} + E_{\text{confirmare}}. \tag{11.6}$$

Reducerea numărului de mesaje, agregarea datelor și funcționarea ciclică pot prelungi semnificativ autonomia.

Latența totală dintre producerea unui eveniment și reacția sistemului este:

$$L_{\text{total}} = L_{\text{nod}} + L_{\text{acces}} + L_{\text{rețea}} + L_{\text{procesare}} + L_{\text{actuare}}. \tag{11.7}$$

Pentru monitorizarea lentă a temperaturii, o latență de câteva secunde poate fi acceptabilă. Pentru oprirea unui utilaj în urma detectării unei condiții periculoase, decizia trebuie realizată local sau foarte aproape de proces.

Fiabilitatea include probabilitatea livrării mesajelor, disponibilitatea serviciului, toleranța la defectarea unui nod, recuperarea după întreruperea comunicației și păstrarea datelor în absența rețelei.

Figura 11.2 prezintă principalele compromisuri.

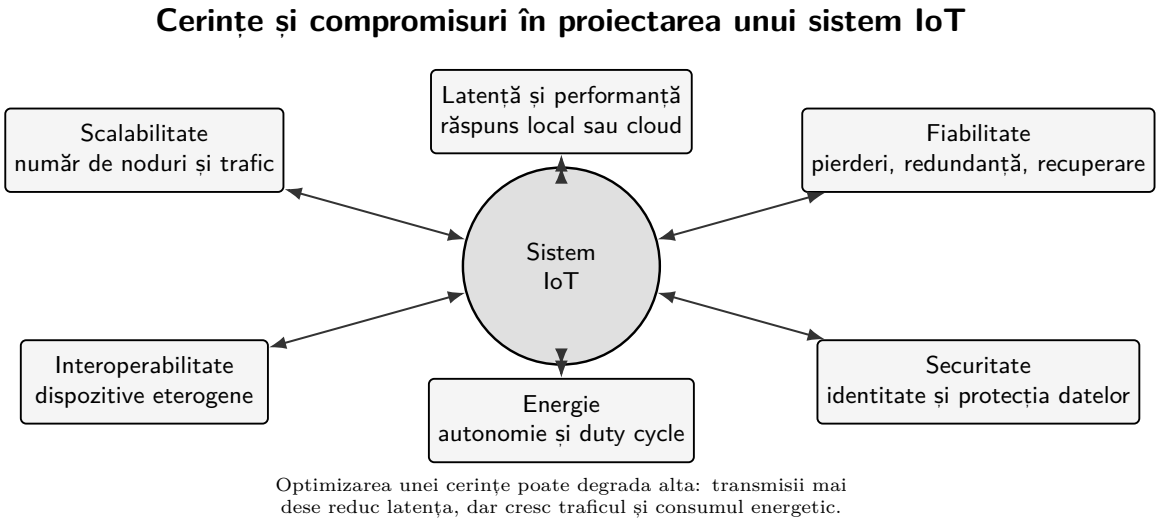


Figura 11.2 Cerințe și compromisuri în proiectarea unui sistem IoT.

11.2.3 Inteligență locală și autonomie

Procesarea locală permite reducerea volumului de date transmis și scăderea latenței. În loc să transmită toate eșantioanele, nodul poate trimite numai valori medii, minime și maxime, evenimente, alarme, caracteristici extrase sau rezultate ale clasificării.

Dacă un accelerometru generează 1000 de eșantioane pe secundă, transmiterea completă a datelor poate fi costisitoare. Nodul poate calcula local indicatori precum valoarea RMS sau componentele spectrale și poate transmite numai rezultatul.

Un criteriu simplificat pentru procesarea locală este:

$$E_{\text{procesare}} + E_{\text{mesaj redus}} < E_{\text{mesaj complet}}. \tag{11.8}$$

Autonomia funcțională presupune și capacitatea de a continua serviciile esențiale în absența conexiunii la cloud. Nodul sau gateway-ul poate stoca temporar măsurătorile, aplica reguli locale, comanda actuatoare și retransmite datele după restabilirea conexiunii.

11.3 Arhitectura unui sistem IoT

Arhitectura IoT descrie distribuirea funcțiilor între dispozitive, rețele, gateway-uri, infrastructuri edge și servicii cloud.

Un model practic include patru niveluri: dispozitive și procese fizice, comunicație, edge și gateway, servicii și aplicații.

Această împărțire nu reprezintă un standard unic, ci un model util pentru organizarea responsabilităților [2].

11.3.1 Dispozitiv, rețea, edge și cloud

Figura 11.3 prezintă arhitectura generală.

Nivelul dispozitivelor conține senzori, actuatoare, microcontrolere, interfețe de comunicație

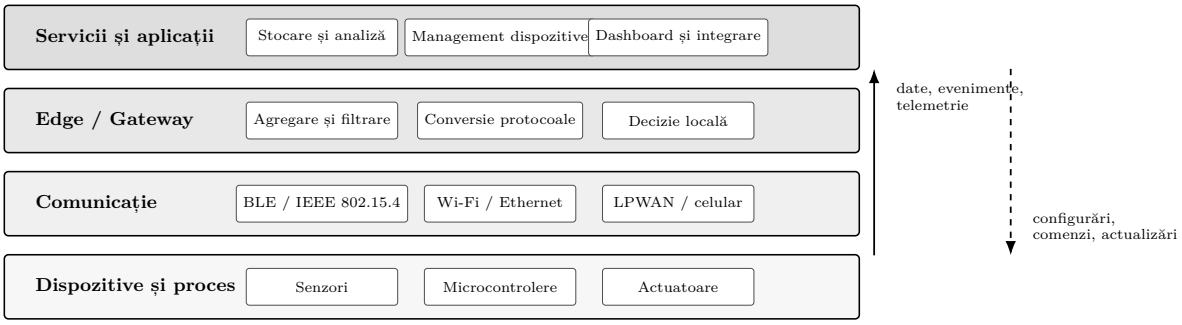


Figura 11.3 Arhitectura distribuită a unui sistem IoT.

și surse de alimentare.

Nivelul rețelei asigură transportul datelor prin tehnologii locale, LPWAN, rețele celulare sau conexiuni cablate.

Nivelul edge execută funcții apropiate de proces: filtrare, agregare, conversia protocoalelor, detecția locală a evenimentelor și stocare temporară.

Infrastructura cloud oferă stocare pe termen lung, analiză la scară mare, managementul dispozitivelor, servicii pentru utilizatori și integrare cu alte sisteme informatice.

Distribuirea procesării între dispozitiv, edge și cloud trebuie realizată în funcție de latență, cost, energie și confidențialitate [80].

11.3.2 Fluxul datelor și al comenzilor

Fluxul ascendent transportă datele de la dispozitive către aplicație:

$$\text{senzor} \rightarrow \text{nod} \rightarrow \text{gateway} \rightarrow \text{serviciu} \rightarrow \text{aplicație}.$$

Fluxul descendent transportă:

- configurații;
- praguri;
- comenzi;
- actualizări;
- confirmări.

Într-un sistem de control, cele două fluxuri formează o buclă:

$$\text{msurare} \rightarrow \text{decizie} \rightarrow \text{comand} \rightarrow \text{proces}.$$

Comenzile trebuie validate înainte de execuție. Un actuator nu trebuie să accepte automat orice mesaj primit din rețea.

Pentru fiecare mesaj sunt importante:

- sursa;
- destinația;
- momentul producerii;

- versiunea formatului;
- autenticitatea;
- validitatea valorii.

11.3.3 Rolul gateway-ului

Gateway-ul conectează rețeaua locală de dispozitive la infrastructura IP sau cloud. Figura 11.4 prezintă principalele sale funcții.

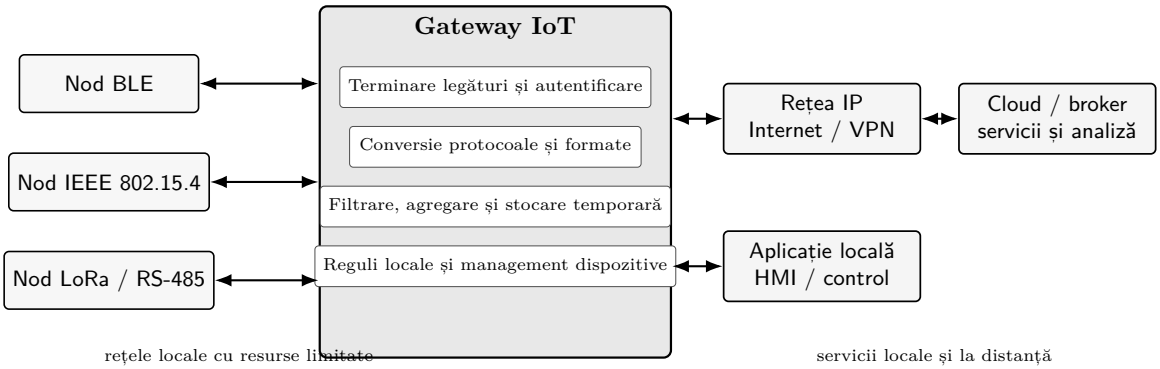


Figura 11.4 Fluxurile de date și funcțiile unui gateway IoT.

Gateway-ul poate realiza conversia dintre protocoale, agregarea mesajelor, eliminarea datelor redundante, stocarea temporară, autentificarea nodurilor, terminarea conexiunilor securizate, procesarea locală și gestionarea actualizărilor.

Gateway-ul nu este obligatoriu în toate sistemele. Un dispozitiv Wi-Fi sau celular poate comunica direct cu un serviciu cloud. Totuși, gateway-ul este util atunci când nodurile folosesc protocoale locale, au resurse limitate sau trebuie să funcționeze în absența Internetului.

Gateway-ul poate deveni un punct critic. Proiectarea trebuie să considere defectarea sa, pierderea alimentării, indisponibilitatea conexiunii externe, compromiterea securității și depășirea capacității de procesare.

11.4 Noduri senzoriale inteligente

Nodul senzorial reprezintă interfața directă dintre sistemul IoT și mediul fizic. Acesta măsoară parametri, execută procesări locale și transmite informațiile relevante.

Arhitectura nodului depinde de aplicație, însă principalele componente sunt similare [94].

11.4.1 Structura hardware

Figura 11.5 prezintă structura generală.

Componentele principale sunt senzori și circuite de condiționare, convertoare analog–digitale, microcontroler, memorie, transceiver, circuit de management energetic și interfețe de diagnostic.

Senzorii pot furniza semnale analogice sau date digitale. Pentru semnalele analogice trebuie analizate domeniul tensiunii, rezoluția convertorului, frecvența de eșantionare, filtrarea

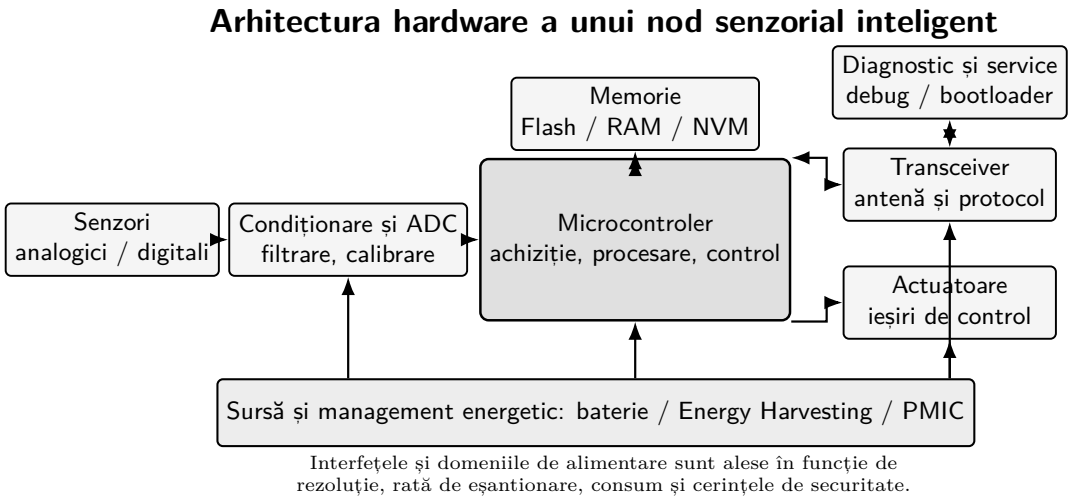


Figura 11.5 Arhitectura hardware a unui nod senzorial inteligent.

și calibrarea.

Microcontrolerul trebuie ales în funcție de resursele de calcul, memoria necesară, periferice, modurile de consum redus, suportul criptografic și durata de menținere a produsului.

11.4.2 Achiziție, procesare și comunicație

Fluxul intern al nodului este:

$$\text{achiziție} \rightarrow \text{validare} \rightarrow \text{procesare} \rightarrow \text{formatare} \rightarrow \text{transmisie}.$$

Validarea poate detecta valori în afara domeniului, erori de comunicație cu senzorul, date incomplete, calibrare invalidă sau variații fizic imposibile.

Un mesaj nu trebuie să conțină numai valoarea măsurată. Pot fi necesare identificatorul nodului, marca temporală, unitatea de măsură, starea bateriei, indicatori de calitate și numărul secvenței.

Exemplu

Exemplu: raportarea temperaturii.
Un nod poate transmite temperatura numai dacă variația depășește un prag:

$$|T[k] - T_{\text{transmis}}| \geq \Delta T_{\min}.$$

Pentru evitarea unei perioade foarte lungi fără mesaje, se poate impune și o transmisie periodică de confirmare.

Această strategie reduce traficul, dar trebuie proiectată astfel încât aplicația să poată distinge între o valoare constantă și un nod defect.

11.4.3 Funcționare ciclică și procesare locală

Pentru reducerea consumului energetic, nodul alternează între stări active și moduri sleep:

$$\text{sleep} \rightarrow \text{trezire} \rightarrow \text{msurare} \rightarrow \text{procesare} \rightarrow \text{transmisie} \rightarrow \text{sleep}.$$

Factorul de activitate este:

$$D = \frac{T_{\text{activ}}}{T_{\text{ciclu}}}. \tag{11.9}$$

Energia ciclului este:

$$E_{\text{ciclu}} = \sum_{i=1}^{N} P_i t_i. \tag{11.10}$$

Procesarea locală poate include:

- filtrare;
- agregare;
- compresie;
- detectarea evenimentelor;
- extragerea caracteristicilor;
- inferență cu modele compacte.

Rezultatul trebuie evaluat prin economia netă:

$$E_{\text{economie}} = E_{\text{comunicație evitată}} - E_{\text{procesare locală}}. \tag{11.11}$$

Procesarea locală este avantajoasă energetic dacă:

$$E_{\text{economie}} > 0. \tag{11.12}$$

În plus față de reducerea consumului, procesarea locală poate îmbunătăți latența și poate limita transmiterea datelor sensibile.

În blocul următor vor fi analizate topologiile rețelelor IoT și principalele tehnologii de comunicație, de la BLE și IEEE 802.15.4 până la LoRaWAN și rețele celulare.

11.5 Topologii și organizarea rețelelor IoT

Topologia descrie modul în care dispozitivele sunt conectate logic și traseele prin care circulă informațiile. Aceasta influențează consumul energetic, latența, toleranța la defecte și capacitatea rețelei de a integra noduri noi.

Topologia trebuie diferențiată de tehnologia de comunicație. De exemplu, o rețea bazată pe IEEE 802.15.4 poate fi organizată în stea, arbore sau mesh, în funcție de protocolul utilizat și de cerințele aplicației.

Principalele criterii de alegere sunt numărul și distribuția nodurilor, distanțele dintre dispozitive, posibilitatea comunicației directe, consumul energetic permis, toleranța necesară

la defecte și volumul și frecvența mesajelor.

Figura 11.6 prezintă topologiile utilizate frecvent în sistemele IoT.

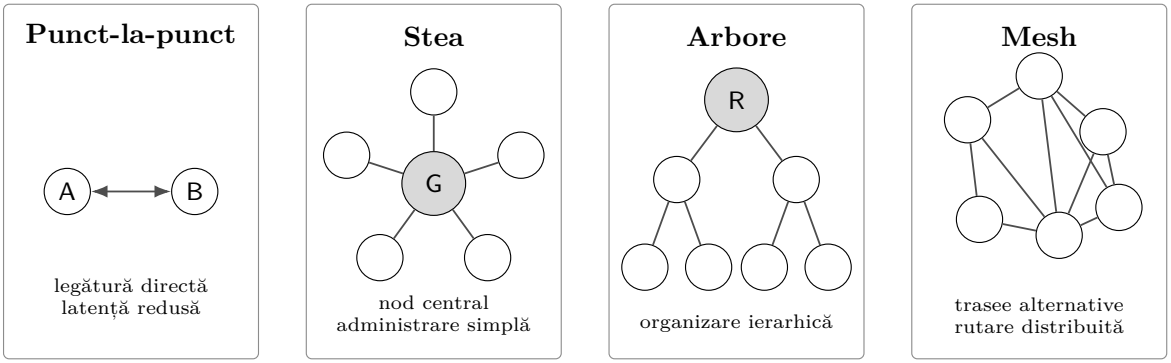


Figura 11.6 Topologii utilizate în rețelele de dispozitive IoT.

11.5.1 Comunicația punct-la-punct și topologia stea

Comunicația punct-la-punct conectează direct două dispozitive. Este utilizată în configurații simple, precum un senzor și un controler, un dispozitiv purtabil și un telefon, sau un echipament industrial și un terminal de configurare.

Avantajele principale sunt simplitatea și latența redusă. Totuși, extinderea la un număr mare de noduri necesită o altă organizare.

În topologia stea, toate nodurile comunică direct cu un element central:

$$\text{noduri} \rightarrow \text{coordonator sau gateway.}$$

Nodurile periferice nu retransmit mesajele altor dispozitive. Din acest motiv, ele pot rămâne majoritatea timpului în moduri de consum redus.

Topologia stea este utilizată frecvent în rețele Wi-Fi, rețele Bluetooth Low Energy cu un dispozitiv central, sisteme LoRaWAN și rețele celulare IoT.

Avantajele sale sunt implementarea simplă, administrarea centralizată, consumul redus la nivelul nodurilor și rutarea simplificată.

Limitarea principală este dependența de elementul central. Defectarea gateway-ului sau pierderea alimentării acestuia poate afecta întreaga zonă deservită.

Capacitatea unei rețele stea nu este determinată numai de numărul nodurilor, ci și de traficul generat. Pentru N noduri care transmit fiecare cu frecvența f_m, rata totală a mesajelor este:

$$R_{\text{mesaje}} = N f_m. \tag{11.13}$$

Dacă nodurile transmit simultan sau prea frecvent, pot apărea coliziuni și întârzieri chiar dacă mesajele individuale sunt scurte.

11.5.2 Topologiile arbore și mesh

Topologia arbore organizează nodurile ierarhic. Un nod superior poate agrega datele mai multor noduri inferioare și le poate transmite către nivelul următor.

Această structură este potrivită pentru clădiri cu mai multe niveluri, instalații industriale organizate pe zone, infrastructuri cu gateway-uri intermediare și rețele în care datele sunt agregate ierarhic.

Principalul avantaj este scalabilitatea. Totuși, defectarea unui nod intermediar poate izola întreaga ramură aflată sub acesta.

Într-o topologie mesh, nodurile pot comunica cu mai mulți vecini, iar mesajele pot urma mai multe trasee până la destinație. Rețeaua poate adapta traseul atunci când o legătură devine indisponibilă.

Pentru un traseu cu H legături, probabilitatea de livrare poate fi estimată, în ipoteza independenței legăturilor, prin:

$$P_{\text{livrare}} = \prod_{i=1}^{H} p_i, \tag{11.14}$$

unde p_i este probabilitatea de succes a legăturii i.

Dacă toate legăturile au aceeași probabilitate p:

$$P_{\text{livrare}} = p^H. \tag{11.15}$$

Relația arată că introducerea mai multor salturi nu garantează automat o fiabilitate mai mare. Traseele alternative și retransmisiile trebuie gestionate de protocolul de rutare.

Avantajele topologiei mesh sunt acoperirea extinsă, traseele alternative, toleranța mai bună la defecte și posibilitatea auto-organizării.

Dezavantajele includ complexitatea mai mare, traficul suplimentar pentru rutare, latența dependentă de numărul de salturi și consumul crescut pentru nodurile care retransmit.

Energia unui nod cu funcție de rutare poate fi aproximată prin:

$$E_{\text{nod}} = E_{\text{propriu}} + E_{\text{recepție}} + E_{\text{retransmisie}}. \tag{11.16}$$

Nodurile amplasate aproape de gateway pot retransmite o parte importantă din traficul rețelei și se pot descărca mai repede decât nodurile periferice.

11.5.3 Alegerea topologiei

Nu există o topologie optimă pentru toate aplicațiile.

Topologia stea este potrivită atunci când toate nodurile pot comunica direct cu gateway-ul, consumul energetic trebuie minimizat, infrastructura centrală este suficient de fiabilă și traficul total poate fi gestionat de canal.

Topologia mesh este justificată atunci când comunicația directă nu este posibilă, mediul produce umbriri și obstacole, sunt necesare trasee alternative, iar nodurile dispun de

suficientă energie pentru retransmisii.

O arhitectură hibridă poate combina:

$$\text{mesh local} \rightarrow \text{gateway} \rightarrow \text{rețea IP} \rightarrow \text{cloud}.$$

Magistralele precum CAN, RS-485 sau Modbus sunt întâlnite frecvent în sisteme industriale, însă acestea descriu o infrastructură locală cablată. Un gateway industrial poate integra o astfel de magistrală într-un sistem IoT mai larg.

Exemplu

Exemplu: monitorizarea unei hale industriale.
Senzorii amplasați în aceeași hală pot forma o rețea mesh de putere redusă. Un gateway colectează datele și le transmite prin Ethernet către o platformă locală sau cloud.
Dacă toate nodurile au vizibilitate radio directă către gateway, o topologie stea poate fi mai simplă și mai eficientă energetic.

11.6 Tehnologii de comunicație pentru IoT

Tehnologia de comunicație trebuie aleasă pe baza aplicației, nu numai pe baza razei maxime declarate. Parametrii reali depind de frecvență, puterea de emisie, antenă, obstacole, interferențe și reglementările spectrului radio.

Un buget simplificat al legăturii poate fi exprimat în decibeli prin:

$$P_R = P_T + G_T + G_R - L_{\text{cale}} - L_{\text{suplimentare}}, \tag{11.17}$$

unde P_T este puterea transmisă, G_T și G_R sunt câștigurile antenelor, L_{cale} reprezintă pierderea de propagare, iar $L_{\text{suplimentare}}$ include cabluri, neadaptări și obstacole.

Legătura este posibilă dacă puterea recepționată depășește sensibilitatea receptorului cu o marjă suficientă.

Figura 11.7 prezintă compromisurile generale dintre rază, rată de transfer și consum.

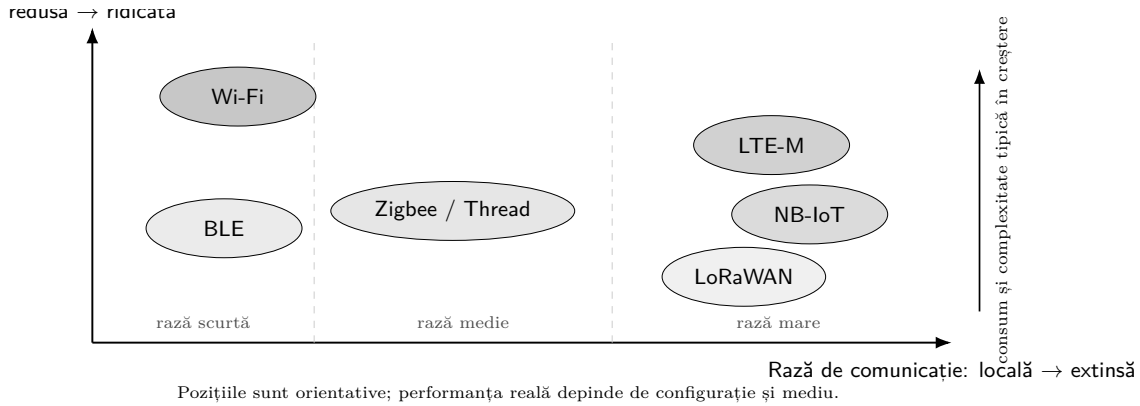


Figura 11.7 Compromisuri între raza de comunicație, rata de transfer și consum.

11.6.1 Comunicații de rază scurtă

Bluetooth Low Energy este optimizat pentru dispozitive care transmit cantități reduse de date și trebuie să funcționeze cu un consum redus.

Este utilizat frecvent în dispozitive purtabile, senzori medicali, balize de proximitate, configurarea locală a echipamentelor și senzori conectați la un smartphone.

Un avantaj important este integrarea sa în telefoane, tablete și calculatoare. Un dispozitiv poate comunica direct cu aplicația utilizatorului fără un gateway specializat.

Consumul real depinde de intervalul de publicitate, intervalul conexiunii, puterea de emisie, lungimea mesajelor și numărul retransmisiilor.

Wi-Fi oferă o rată de transfer mai mare și integrare directă cu rețelele IP. Este potrivit pentru camere video, echipamente conectate la rețea, gateway-uri și aplicații cu volume mari de date.

Consumul mai ridicat și procedura de asociere pot reprezenta dezavantaje pentru nodurile alimentate din baterii. Pentru mesaje rare, energia necesară inițierii conexiunii poate depăși energia efectivă a transmisiei.

NFC și RFID sunt utilizate pentru comunicații la distanță mică, identificare și, în anumite configurații, alimentarea dispozitivelor fără baterie. Acestea nu trebuie comparate direct cu tehnologiile destinate unei conexiuni permanente la Internet.

11.6.2 IEEE 802.15.4, Zigbee și Thread

IEEE 802.15.4 definește mecanisme pentru nivelul fizic și controlul accesului la mediu în rețele wireless cu rată redusă și consum scăzut [42].

Standardul nu definește singur întreaga stivă IoT. Protocoale precum Zigbee, Thread și 6LoWPAN adaugă funcții de rețea și aplicație.

Relația poate fi reprezentată astfel:

$$\text{IEEE 802.15.4} \rightarrow \text{legtur radio},$$

$$\text{Zigbee sau Thread} \rightarrow \text{rețea},\ \text{rutare},\ \text{servicii}.$$

Zigbee oferă mecanisme pentru formarea rețelelor, adresarea nodurilor, rutarea mesh și definirea profilurilor de aplicație.

Thread utilizează de asemenea IEEE 802.15.4, dar este orientat către comunicație IPv6. Un dispozitiv Thread poate participa la o rețea IP printr-un *border router*.

Matter este un protocol de aplicație pentru interoperabilitatea dispozitivelor. El poate utiliza rețele IP bazate, de exemplu, pe Thread, Wi-Fi sau Ethernet. Matter nu reprezintă o nouă tehnologie radio.

Figura 11.8 prezintă poziția acestor tehnologii în stivă.

Rețelele mesh bazate pe aceste tehnologii sunt potrivite pentru clădiri, automatizări și sisteme distribuite în care distanțele dintre noduri sunt moderate.

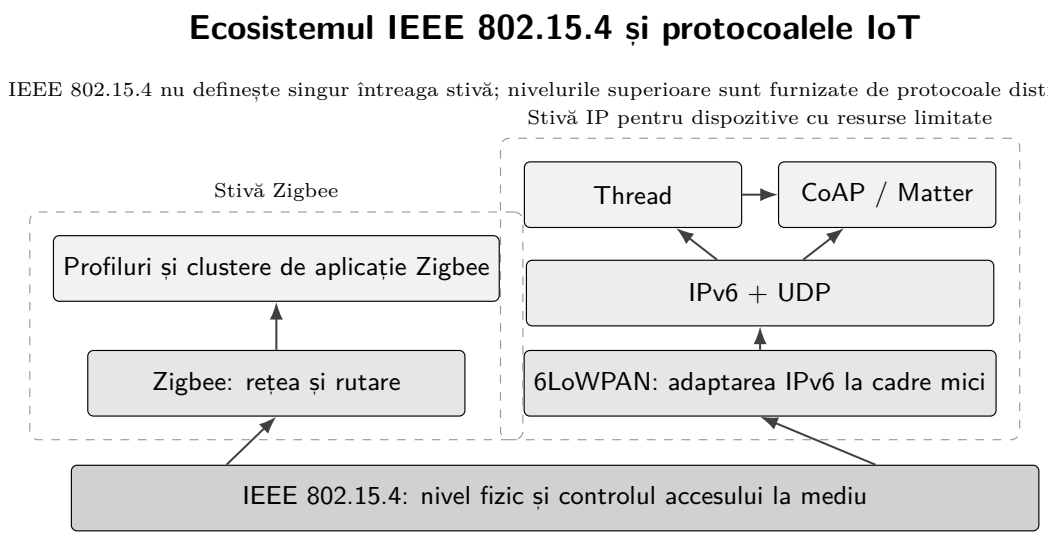


Figura 11.8 Relația dintre IEEE 802.15.4, 6LoWPAN, Thread și protocoalele de aplicație.

11.6.3 LPWAN și comunicații celulare

LPWAN, *Low-Power Wide-Area Network*, desemnează tehnologii proiectate pentru transmiterea unor mesaje mici pe distanțe mari, cu un consum energetic redus.

Aceste tehnologii sunt adecvate pentru agricultură, contoare inteligente, monitorizarea mediului, localizarea activelor și infrastructuri urbane.

LoRa definește tehnologia de modulație a stratului fizic. LoRaWAN definește arhitectura rețelei, accesul la canal și comunicarea dintre dispozitive, gateway-uri și serverele de rețea.

Arhitectura tipică este:

$$\text{noduri} \rightarrow \text{gateway} - \text{uri} \rightarrow \text{server de rețea} \rightarrow \text{aplicație}.$$

Figura 11.9 prezintă această structură.

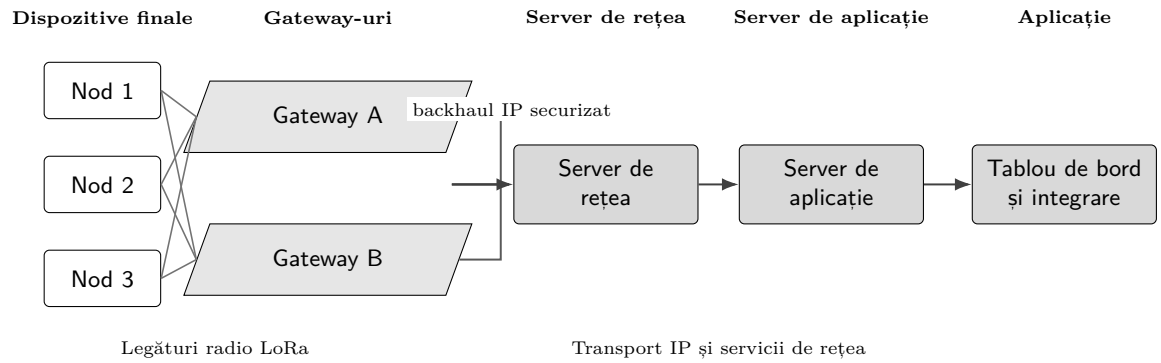


Figura 11.9 Arhitectura unei rețele LoRaWAN.

Gateway-ul LoRaWAN retransmite mesajele radio către infrastructura IP. El nu îndeplinește, în mod obișnuit, rolul unui router mesh între nodurile finale.

Avantajele LoRaWAN includ acoperirea extinsă, consumul redus pentru mesaje rare, posibilitatea instalării unei infrastructuri private și suportul pentru un număr mare de noduri cu trafic redus.

Limitările includ rata redusă de transfer, restricțiile privind timpul de ocupare a canalului, latența variabilă și capacitatea limitată pentru mesaje frecvente.

NB-IoT și LTE-M utilizează infrastructura operatorilor celulari. Aceste tehnologii oferă acoperire administrată de operator, autentificare specifică rețelelor celulare, integrare directă cu rețele IP și suport pentru implementări geografice extinse.

LTE-M este mai potrivit pentru aplicații care necesită mobilitate, latență mai redusă sau o rată de transfer mai mare. NB-IoT este orientat către dispozitive cu trafic redus și funcționare îndelungată.

Utilizarea unei rețele celulare introduce costuri de conectivitate și dependență de serviciile operatorului.

11.6.4 Compararea și alegerea tehnologiei

Tabelul 11.1 prezintă o comparație calitativă. Valorile concrete trebuie determinate pentru versiunea standardului, configurația radio și mediul aplicației.

Tabela 11.1 Compararea calitativă a tehnologiilor de comunicație IoT

Tehnologie	Acoperire și infrastructură	Caracteristică dominantă	Aplicații reprezentative
BLE	Locală, direct sau prin gateway	Consum redus și integrare mobilă	Wearables și senzori personali
Wi-Fi	Locală, cu punct de acces	Rată mare de transfer	Multimedia și echipamente alimentate
Zigbee / Thread	Locală, stea sau mesh	Rețele distribuite de putere redusă	Clădiri și automatizări
LoRaWAN	Extinsă, prin gateway-uri	Mesaje mici și rare	Agricultură și Smart City
NB-IoT	Extinsă, prin operator	Trafic redus și acoperire celulară	Contorizare și monitorizare
LTE-M	Extinsă, prin operator	Mobilitate și latență mai redusă	Logistică și dispozitive mobile

Timpul minim ideal necesar transmiterii unui cadru de L biți la rata efectivă R_b este:

$$T_{\text{cadru}} = \frac{L}{R_b}. \tag{11.18}$$

În practică se adaugă preambulul, antetele, confirmările, timpii de acces la canal și retransmisiile.

Energia unei încercări de transmisie este:

$$E_{\text{încercare}} = P_{\text{TX}} T_{\text{TX}} + P_{\text{RX}} T_{\text{RX}} + E_{\text{procesare}}. \tag{11.19}$$

Dacă probabilitatea de succes a unei încercări este p_s, energia medie idealizată pentru livrarea unui mesaj este:

$$E_{\text{livrare}} \approx \frac{E_{\text{încercare}}}{p_s}. \tag{11.20}$$

Această relație evidențiază efectul unei legături radio slabe. Creșterea retransmisiilor poate elimina avantajul aparent al unei tehnologii cu putere redusă.

Selecția trebuie realizată prin parcurgerea următorilor pași:

1. stabilirea distanței și a mediului de propagare;
2. estimarea dimensiunii și frecvenței mesajelor;
3. determinarea latenței acceptabile;
4. evaluarea sursei de alimentare;
5. stabilirea infrastructurii disponibile;
6. calcularea bugetului legăturii și al energiei;
7. verificarea restricțiilor de spectru și securitate.

Figura 11.10 prezintă un proces simplificat de selecție.

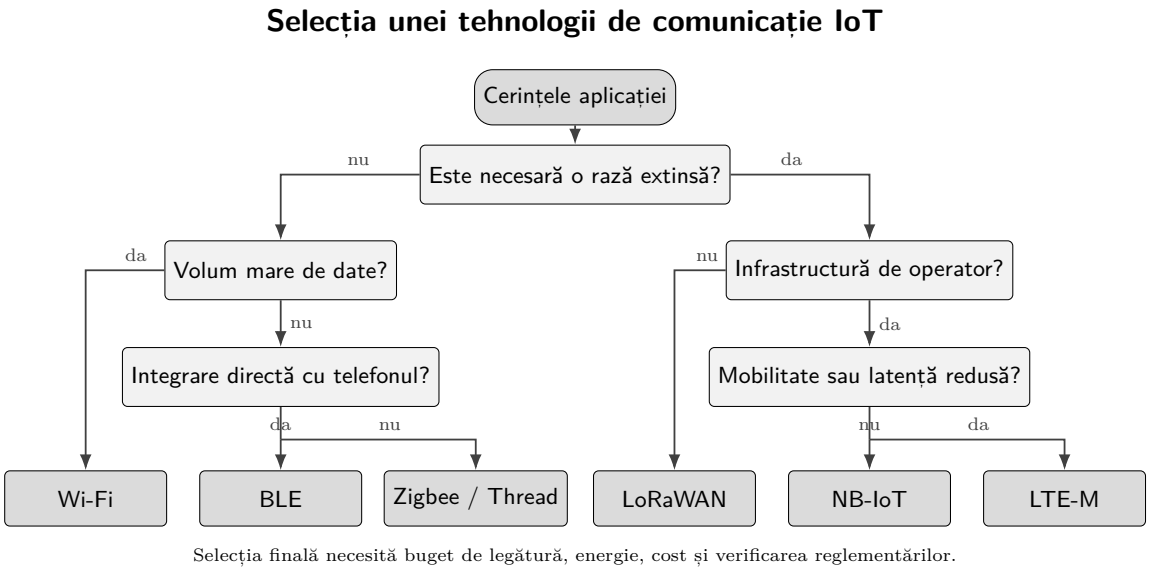


Figura 11.10 Selecția tehnologiei de comunicație pentru o aplicație IoT.

În blocul următor vor fi analizate protocoalele IPv6, 6LoWPAN, MQTT și CoAP, distribuirea procesării între edge și cloud, precum și managementul și securizarea dispozitivelor.

11.7 Protocoale și stive software IoT

Tehnologia radio stabilește modul în care biții sunt transmiși prin mediul fizic. Pentru realizarea unui sistem IoT complet sunt însă necesare protocoale suplimentare pentru adresare, rutare, transport, securitate și schimbul informațiilor dintre aplicații.

O stivă IoT poate include următoarele niveluri: nivel fizic și acces la mediu, nivel de adaptare și rețea, nivel de transport, nivel de securitate și nivel de aplicație.

Figura 11.11 prezintă o stivă software generică.

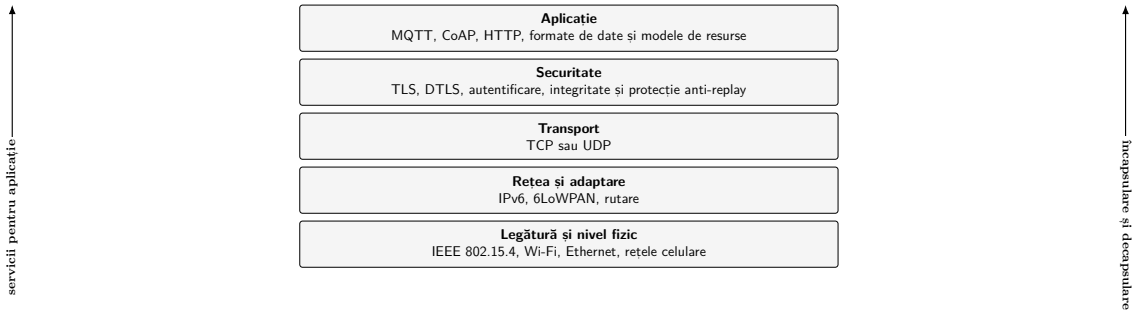


Figura 11.11 Organizarea stratificată a protocoalelor într-un sistem IoT.

Separarea pe niveluri permite înlocuirea unei tehnologii fără modificarea completă a aplicației. De exemplu, aceeași aplicație MQTT poate utiliza o conexiune Wi-Fi, Ethernet sau celulară, dacă nivelurile inferioare oferă transport IP.

11.7.1 IPv6 și adaptarea pentru dispozitive cu resurse limitate

Protocolul IPv6 oferă adresare și rutare pentru rețele IP și utilizează adrese pe 128 de biți [25].

Numărul teoretic de adrese posibile este:

$$N_{\mathrm{IPv6}} = 2^{128}. \tag{11.21}$$

Dimensiunea mare a spațiului de adresare permite identificarea unui număr foarte mare de interfețe, dar nu elimină necesitatea administrării identității și a politicilor de acces.

În rețelele cu cadre mici și resurse limitate, transmiterea antetelor IPv6 complete poate produce un overhead semnificativ. 6LoWPAN introduce un nivel de adaptare pentru transportul pachetelor IPv6 peste rețele de putere redusă, incluzând comprimarea antetelor, fragmentarea și reasamblarea, adaptarea la dimensiunea redusă a cadrelor și suport pentru transmiterea în rețele constrânse.

Mecanismele de bază sunt definite în RFC 4944 [58].

Dacă un pachet IP de lungime L_{IP} trebuie fragmentat în cadre cu sarcină utilă maximă L_F, numărul minim ideal de fragmente este:

$$N_F = \left\lceil \frac{L_{\mathrm{IP}}}{L_F} \right\rceil. \tag{11.22}$$

Pierderea unui singur fragment poate impune retransmiterea unei părți sau a întregului pachet, în funcție de protocolul utilizat. Din acest motiv, mesajele IoT trebuie păstrate cât mai compacte.

11.7.2 MQTT și modelul publish–subscribe

MQTT este un protocol de aplicație bazat pe modelul *publish–subscribe*. Comunicația este intermediată de un broker, iar producătorul datelor nu trebuie să cunoască direct destinatarii [62].

Elementele principale sunt *publisher-ul*, care publică mesaje, *broker-ul*, care primește și distribuie mesajele, *subscriber-ul*, care se abonează la anumite subiecte, și *topic-ul*, care identifică logic fluxul de informații.

Figura 11.12 prezintă această arhitectură.

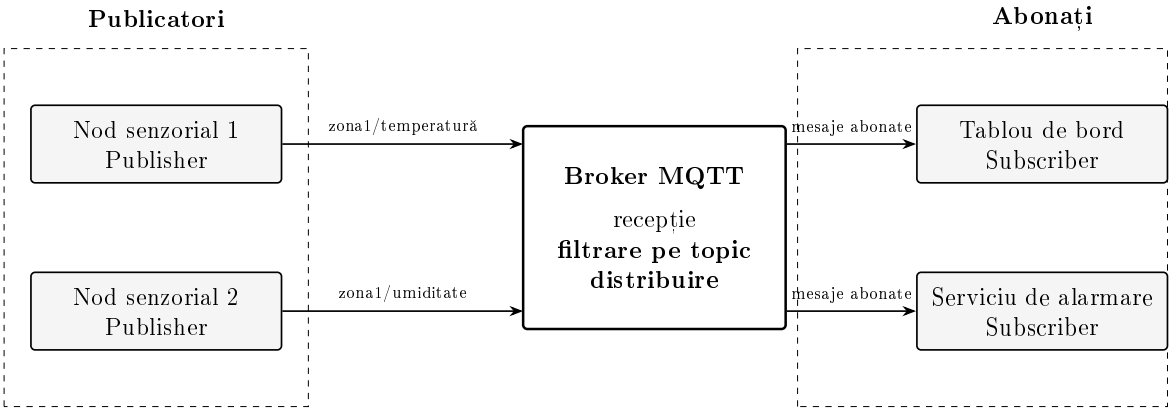


Figura 11.12 Modelul publish–subscribe utilizat de MQTT.

Un nod poate publica temperatura pe subiectul:

```
1 agricultura/zona1/nod12/temperatura
```

iar o aplicație se poate abona la:

```
1 agricultura/zona1/+/temperatura
```

MQTT definește trei niveluri de calitate a serviciului: QoS 0, cu livrare de tip *at most once*, QoS 1, cu livrare de tip *at least once*, și QoS 2, cu livrare de tip *exactly once* la nivelul protocolului.

QoS 1 poate conduce la mesaje duplicate. Aplicația trebuie să poată recunoaște și procesa în siguranță retransmisiile.

Numărul total de octeți transferați pentru N mesaje poate fi aproximat prin:

$$D_{\mathrm{MQTT}} = N\left(L_{\mathrm{payload}} + L_{\mathrm{antet}} + L_{\mathrm{transport}}\right). \tag{11.23}$$

La mesaje foarte mici, antetele și confirmările pot reprezenta o parte importantă din traficul total.

11.7.3 CoAP și modelul bazat pe resurse

CoAP, *Constrained Application Protocol*, este proiectat pentru dispozitive și rețele cu resurse limitate. Protocolul utilizează un model bazat pe resurse, apropiat conceptual de REST, și definește operații precum GET, POST, PUT și DELETE.

O resursă poate fi adresată printr-un URI:

```
coap://nod12.local/senzori/temperatura
```

CoAP utilizează mesaje compacte și este definit în RFC 7252 [79].

Sunt disponibile mesaje confirmabile, neconfirmabile, de confirmare și de resetare.

MQTT și CoAP deservesc modele diferite: MQTT este potrivit pentru distribuirea fluxurilor de evenimente prin intermediul unui broker, iar CoAP este potrivit pentru accesarea și modificarea directă a unor resurse ale dispozitivului.

Exemplu

Exemplu: MQTT sau CoAP.
Un nod care transmite periodic măsurători către mai multe aplicații poate utiliza MQTT. Un sistem în care un controler citește starea unui actuator și îi modifică configurația poate utiliza CoAP.
Alegerea depinde de arhitectura aplicației, nu numai de dimensiunea mesajului.

11.7.4 Formate de date și interoperabilitate

Protocolul de aplicație stabilește schimbul mesajelor, dar nu definește întotdeauna semnificația datelor.

Un mesaj trebuie să specifice cel puțin identificatorul mărimii, valoarea, unitatea de măsură, marca temporală, starea sau calitatea măsurătorii și versiunea formatului.

Un exemplu JSON este:

```
{
  "device_id": "nod12",
  "timestamp": "2026-07-30T10:00:00Z",
  "temperature": {
    "value": 24.6,
    "unit": "degC"
  },
  "battery_mv": 3710
}
```

Formatele text sunt ușor de analizat, dar pot introduce un overhead ridicat. Formatele binare reduc traficul, însă necesită o schemă comună și gestionarea versiunilor.

Interoperabilitatea presupune compatibilitate la mai multe niveluri: conectivitate, protocoale, structură a mesajelor, unități și semnificație, și comportament al dispozitivelor.

11.8 Edge, gateway și infrastructuri cloud

Procesarea unui sistem IoT poate fi distribuită între noduri, gateway-uri, servere edge și infrastructuri cloud.

Nu toate datele trebuie transmise către cloud. Alegerea locului de procesare depinde de latența maximă admisă, volumul datelor, conectivitatea disponibilă, confidențialitatea informațiilor, resursele hardware și costul comunicației.

11.8.1 Distribuirea procesării

Figura 11.13 prezintă distribuirea funcțiilor.

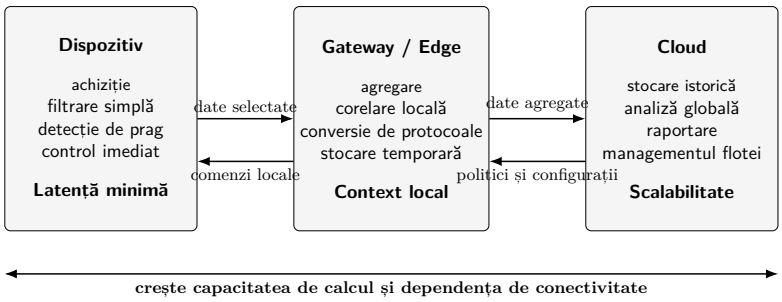


Figura 11.13 Distribuirea procesării între dispozitiv, edge și cloud.

La nivelul nodului pot fi executate filtrarea semnalului, detectarea pragurilor, compresia, clasificări simple și controlul imediat al actuatoarelor.

Gateway-ul sau serverul edge poate realiza agregarea datelor, corelarea mai multor senzori, detectarea anomaliilor locale, conversia protocoalelor și controlul unei zone.

Cloud-ul este potrivit pentru stocare pe termen lung, analiză globală, antrenarea modelelor, raportare și managementul unei flote mari.

Latența totală poate fi exprimată prin:

$$L_{\text{total}} = L_{\text{dispozitiv}} + L_{\text{acces}} + L_{\text{gateway}} + L_{\text{transport}} + L_{\text{serviciu}}. \tag{11.24}$$

O funcție critică nu trebuie transferată către cloud dacă latența sau indisponibilitatea rețelei poate produce o stare periculoasă.

11.8.2 Agregarea, stocarea temporară și funcționarea offline

Gateway-ul poate reduce traficul prin agregarea măsurătorilor.

Pentru N valori individuale, valoarea medie este:

$$\overline{x} = \frac{1}{N} \sum_{i=1}^{N} x_i. \tag{11.25}$$

În locul tuturor eșantioanelor pot fi transmise:

- media;
- minimul;
- maximul;
- abaterea standard;
- numărul evenimentelor;
- valorile care depășesc un prag.

Dacă legătura cu serviciul cloud este indisponibilă, gateway-ul trebuie să poată stoca temporar datele.

Capacitatea minimă a bufferului poate fi estimată prin:

$$C_{\text{buffer}} \geq R_D T_{\text{offline}}, \tag{11.26}$$

unde R_D este rata generării datelor, iar T_{offline} este durata maximă estimată a întreruperii.

După restabilirea comunicației, mesajele trebuie retransmise fără pierderea ordinii și fără generarea unor duplicări necontrolate.

Pentru fiecare mesaj sunt utile:

- marca temporală;
- numărul secvenței;
- identificatorul sursei;
- indicatorul de retransmisie.

11.8.3 Alegerea locului de procesare

O funcție poate fi amplasată la nivelul dispozitivului dacă necesită latență foarte redusă, poate fi executată cu resursele locale, reduce semnificativ traficul sau utilizează date sensibile.

Procesarea edge este potrivită când sunt necesare corelarea mai multor noduri, controlul unei zone, independența față de Internet sau resurse mai mari decât cele ale nodului.

Cloud-ul este preferat când aplicația necesită date din mai multe locații, stocare extinsă, analiză intensivă sau integrare cu sisteme organizaționale.

O arhitectură robustă distribuie responsabilitățile în loc să depindă exclusiv de un singur nivel.

11.9 Managementul dispozitivelor IoT

Administrarea unei flote IoT include toate operațiile necesare de la fabricarea dispozitivului până la retragerea sa din exploatare.

Un ciclu de viață tipic conține:

1. fabricare și personalizare;
2. înrolare;
3. configurare;
4. exploatare și monitorizare;

5. actualizare;

6. retragere și ștergere sigură.

Figura 11.14 prezintă aceste etape.

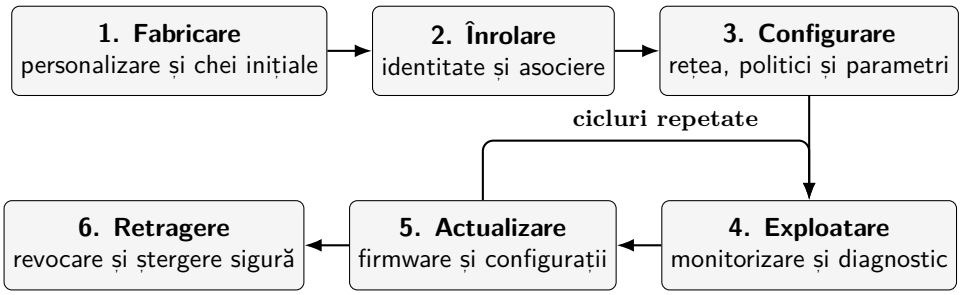


Figura 11.14 Ciclul de viață al unui dispozitiv IoT.

11.9.1 Înrolare și identitate

Înrolarea introduce dispozitivul în infrastructura administrată și îi atribuie o identitate verificabilă.

Identitatea poate fi bazată pe chei simetrice, perechi de chei asimetrice, certificate digitale sau identități stocate într-un element securizat.

Fiecare dispozitiv trebuie să aibă o identitate unică. Utilizarea aceleiași parole sau chei pentru o întreagă familie de produse mărește semnificativ impactul compromiterii unui singur exemplar.

Înrolarea poate include verificarea originii, asocierea cu proprietarul, configurarea rețelei, instalarea certificatelor și stabilirea politicilor de acces.

11.9.2 Configurare, monitorizare și diagnostic

În timpul exploatării trebuie monitorizate versiunea firmware-ului, nivelul bateriei, calitatea comunicației, resetările, erorile senzorilor, memoria disponibilă și timpul de funcționare.

Un mesaj de diagnostic poate conține:

```
{
  "device_id": "nod12",
  "fw_version": "2.3.1",
  "uptime_s": 864230,
  "battery_mv": 3680,
  "rssi_dbm": -104,
  "reset_reason": "watchdog"
}
```

Configurarea la distanță trebuie validată. Un parametru eronat poate crește traficul, poate descărca bateria sau poate afecta procesul controlat.

Sunt recomandate verificarea limitelor, versionarea configurației, păstrarea ultimei configurații valide și revenirea automată după o actualizare eșuată.

11.9.3 Actualizarea firmware-ului și retragerea

Actualizarea OTA permite remedierea vulnerabilităților și corectarea erorilor fără acces fizic la dispozitiv.

Procesul include:

1. descărcarea imaginii;
2. verificarea autenticității;
3. verificarea integrității;
4. instalarea;
5. repornirea;
6. confirmarea funcționării.

Arhitectura actualizărilor firmware pentru dispozitive IoT este tratată și în RFC 9019 [59].

Pentru o imagine de lungime L_{FW} transmisă în blocuri de L_B, numărul minim de blocuri este:

$$N_B = \left\lceil \frac{L_{\mathrm{FW}}}{L_B} \right\rceil . \tag{11.27}$$

Actualizarea trebuie să tolereze întreruperea alimentării și pierderea conexiunii.

Sunt utilizate frecvent două partiții firmware, imagine de rezervă, confirmare după pornire, revenire automată la versiunea anterioară și protecție împotriva revenirii la o versiune vulnerabilă.

La retragerea dispozitivului trebuie revocate identitățile și șterse cheile și datele sensibile.

11.10 Securitatea sistemelor IoT

Securitatea IoT trebuie proiectată pentru întregul sistem și întregul ciclu de viață. Protejarea exclusivă a comunicației nu este suficientă dacă firmware-ul poate fi modificat sau dacă toate dispozitivele folosesc aceeași cheie.

Obiectivele clasice sunt confidențialitatea, integritatea și disponibilitatea. Pentru IoT se adaugă autenticitatea dispozitivelor, siguranța actualizărilor, protejarea procesului fizic și trasabilitatea operațiilor.

11.10.1 Modelul amenințărilor

Un atacator poate acționa asupra:

- dispozitivului;
- legăturii radio;
- gateway-ului;
- serviciilor cloud;
- aplicației utilizator;
- procesului de actualizare.

Amenințările includ:

- interceptarea;
- modificarea mesajelor;
- retransmiterea unor mesaje vechi;
- impersonarea unui nod;
- extragerea cheilor;
- instalarea unui firmware malițios;
- blocarea serviciului;
- manipularea senzorilor.

Figura 11.15 prezintă suprafețele principale de protecție.

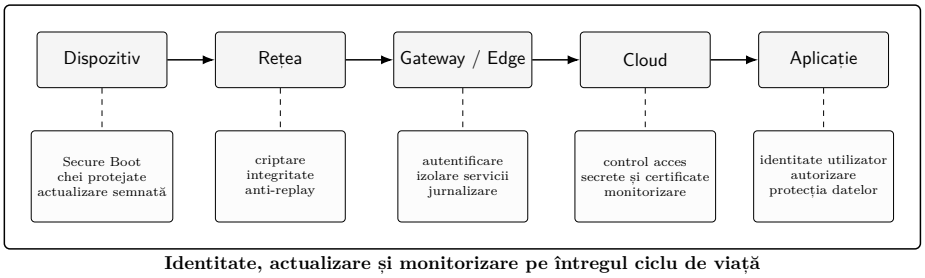


Figura 11.15 Mecanisme de securitate de-a lungul lanțului IoT.

Modelul amenințărilor trebuie să precizeze:

- activele protejate;
- capabilitățile atacatorului;
- punctele de acces;
- impactul posibil;
- măsurile de reducere a riscului.

11.10.2 Protecția dispozitivului și a comunicațiilor

Secure Boot verifică autenticitatea și integritatea firmware-ului înainte de execuție.

Un lanț de încredere poate fi reprezentat prin:

$$\text{ROM de ncredere} \rightarrow \text{bootloader} \rightarrow \text{firmware} \rightarrow \text{aplicație}.$$

Fiecare etapă verifică semnătura sau valoarea criptografică a etapei următoare.

Cheile trebuie protejate prin memorie cu acces restricționat, periferice criptografice, elemente securizate sau mecanisme anti-debug pentru producție.

Comunicațiile pot fi protejate prin TLS, DTLS sau mecanismele de securitate ale tehnologiei radio.

Criptarea singură nu garantează autenticitatea. Un canal securizat trebuie să ofere autentificarea entităților, integritatea mesajelor, protecție anti-replay și confidențialitate, dacă este necesară.

Pentru prevenirea retransmiterii unui mesaj vechi pot fi utilizate numere de secvență, contoare monotone, nonce-uri sau mărci temporale.

11.10.3 Actualizare sigură și protecția datelor

O imagine firmware trebuie acceptată numai dacă:

$$\text{Valid} = \text{SemnturCorect} \wedge \text{VersiunePermis} \wedge \text{PlatformCompatibil}. \tag{11.28}$$

Verificarea integrității printr-un hash neprotejat nu este suficientă. Un atacator poate modifica atât imaginea, cât și hash-ul. Este necesară o semnătură digitală sau un mecanism autentificat echivalent.

Protecția datelor trebuie să includă minimizarea datelor colectate, limitarea perioadei de păstrare, controlul accesului, criptarea datelor sensibile, jurnalizarea operațiilor și ștergerea sigură.

Capabilitățile minime de securitate pentru dispozitive IoT sunt discutate și în ghidurile NIST dedicate bazei de securitate a dispozitivului [28].

Principiul privilegiului minim impune ca fiecare componentă să primească numai drepturile necesare funcției sale.

Un nod care publică temperatura nu trebuie să poată modifica configurația altor dispozitive sau administra brokerul.

În blocul următor vor fi analizate aplicațiile IoT reprezentative și proiectarea completă a unei rețele pentru monitorizarea unei exploatații agricole.

11.11 Aplicații IoT reprezentative

Internet of Things este utilizat în aplicații în care un număr mare de dispozitive trebuie să observe procese fizice, să comunice și să contribuie la luarea unor decizii.

Deși domeniile de utilizare sunt foarte diverse, majoritatea sistemelor IoT urmăresc unul sau mai multe dintre următoarele obiective: monitorizarea continuă a unui proces, detectarea timpurie a anomaliilor, reducerea consumului de resurse, automatizarea unor operații, îmbunătățirea trasabilității și sprijinirea deciziilor prin date măsurate.

Figura 11.16 prezintă câteva domenii reprezentative.

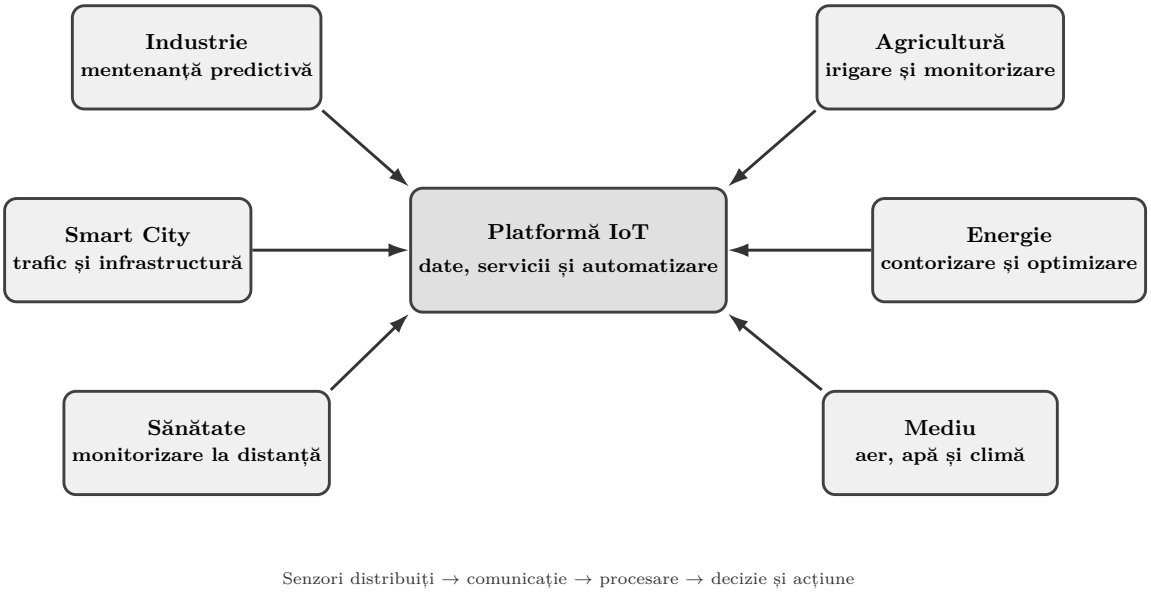


Figura 11.16 Domenii reprezentative de aplicare a tehnologiilor IoT.

11.11.1 Industrie și mentenanță predictivă

În mediul industrial, sistemele IoT sunt utilizate pentru monitorizarea utilajelor și a proceselor de producție.

Parametrii măsurați pot include vibrații, temperatură, presiune, curent electric, turație, zgomot și calitatea produsului.

Un sistem de mentenanță predictivă nu se limitează la detectarea unei defecțiuni deja produse. El urmărește identificarea degradării înainte ca aceasta să determine oprirea echipamentului.

Fluxul general este:

msurare → extragere caracteristici → detectare anomalie → recomandare mentenanț.

Pentru un semnal de vibrații $x[k]$, un indicator simplu este valoarea RMS:

$$x_{\mathrm{RMS}} = \sqrt{\frac{1}{N}\sum_{k=1}^{N} x^2[k]}. \tag{11.29}$$

Nodul poate transmite numai indicatorii calculați și evenimentele, în locul întregului semnal. Totuși, pentru diagnostic avansat poate fi necesară păstrarea temporară a datelor brute.

În aplicațiile industriale trebuie analizate suplimentar latența maximă permisă, interferențele electromagnetice, disponibilitatea rețelei, sincronizarea măsurătorilor și integrarea cu siste-

mele de automatizare existente.

Deciziile de siguranță nu trebuie să depindă exclusiv de un serviciu cloud. Oprirea de urgență trebuie realizată local, prin sisteme proiectate pentru funcția respectivă.

11.11.2 Agricultură și monitorizarea mediului

Agricultura inteligentă utilizează senzori distribuiți pentru monitorizarea condițiilor de dezvoltare a culturilor și pentru utilizarea mai eficientă a apei, energiei și fertilizanților.

Parametrii frecvent monitorizați sunt umiditatea solului, temperatura și umiditatea aerului, iluminarea, precipitațiile, viteza vântului și conductivitatea sau pH-ul solului.

Decizia de irigare nu trebuie bazată exclusiv pe o singură măsurare. Pot fi utilizate mai multe adâncimi de măsurare, filtrarea variațiilor rapide, prognoza meteorologică, tipul culturii și limitele sistemului de irigare.

Un criteriu simplificat poate utiliza două praguri:

$$u_{\text{irigare}}[k] = \begin{cases} 1, & H_s[k] \leq H_L, \\ 0, & H_s[k] \geq H_H, \\ u_{\text{irigare}}[k-1], & H_L < H_s[k] < H_H, \end{cases} \tag{11.30}$$

unde H_s este umiditatea solului, iar H_L și H_H definesc o zonă de histerezis.

Monitorizarea mediului utilizează arhitecturi similare pentru măsurarea calității aerului, calității apei, zgomotului, radiației și condițiilor meteorologice.

În ambele domenii, nodurile sunt amplasate frecvent în exterior, unde trebuie protejate împotriva apei, prafului, condensului și variațiilor de temperatură [13, 93].

11.11.3 Smart City, energie și sănătate

Aplicațiile Smart City integrează informații provenite din mai multe infrastructuri urbane.

Exemplele includ iluminat stradal adaptiv, monitorizarea parcărilor, măsurarea calității aerului, managementul deșeurilor, monitorizarea traficului și contorizarea utilităților.

Scalabilitatea și interoperabilitatea sunt esențiale deoarece infrastructura poate include dispozitive instalate în perioade diferite și administrate de organizații diferite.

În domeniul energetic, sistemele IoT permit citirea contoarelor, monitorizarea consumului, identificarea pierderilor, controlul sarcinilor, integrarea producției distribuite și monitorizarea stațiilor de încărcare.

În domeniul medical, dispozitivele conectate pot măsura parametri fiziologici și pot transmite informații către personalul autorizat. Exemplele includ monitorizarea ritmului cardiac, măsurarea glicemiei, monitorizarea tensiunii arteriale, detectarea căderilor și administrarea echipamentelor medicale.

Datele medicale necesită măsuri stricte privind confidențialitatea, autentificarea, integritatea, controlul accesului și păstrarea și ștergerea datelor.

Un dispozitiv de monitorizare nu trebuie confundat automat cu un dispozitiv medical

destinat luării unor decizii clinice. Cerințele de validare depind de scopul declarat și de reglementările aplicabile.

11.12 Studiu de caz: monitorizarea inteligentă a unei exploatații agricole

Se proiectează un sistem IoT destinat monitorizării unei exploatații agricole cu suprafața de aproximativ 40 de hectare.

Sistemul include 60 de noduri distribuite, un gateway LoRaWAN și o platformă software pentru stocarea și vizualizarea informațiilor.

Fiecare nod măsoară:

- temperatura aerului;
- umiditatea aerului;
- umiditatea solului;
- nivelul iluminării;
- tensiunea sursei de alimentare.

Obiectivele proiectului sunt:

- colectarea periodică a măsurătorilor;
- detectarea valorilor anormale;
- funcționarea în absența temporară a Internetului;
- administrarea individuală a nodurilor;
- protejarea comunicațiilor;
- posibilitatea extinderii rețelei.

11.12.1 Cerințe și arhitectură

Fiecare nod transmite un mesaj la fiecare 15 minute. În cazul detectării unui eveniment important, nodul poate genera suplimentar un mesaj de alarmă.

Figura 11.17 prezintă arhitectura sistemului.

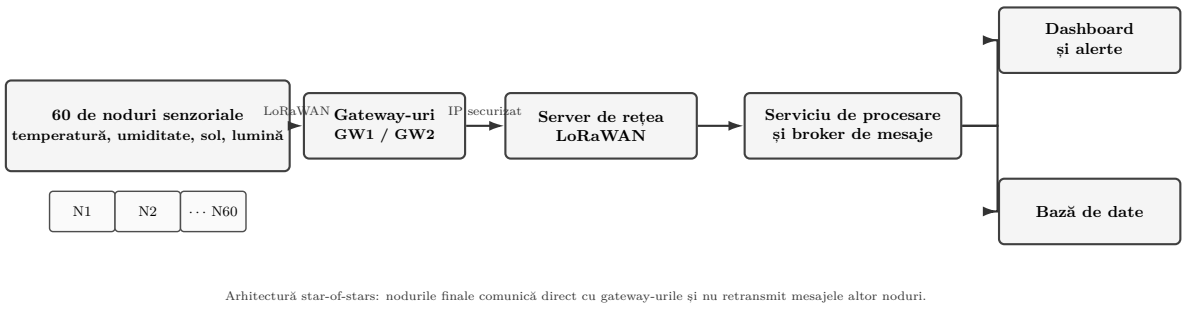


Figura 11.17 Arhitectura sistemului IoT pentru monitorizarea exploatației agricole.

Arhitectura include:

1. noduri senzoriale LoRaWAN;
2. unul sau mai multe gateway-uri;

3. serverul de rețea LoRaWAN;
4. serviciul de procesare;
5. baza de date;
6. aplicația de vizualizare;
7. serviciul de notificare.

Înaintea instalării trebuie realizate:
- măsurători radio în teren;
- verificarea obstacolelor;
- evaluarea poziției gateway-ului;
- analiza sursei de alimentare;
- verificarea acoperirii în zonele marginale.

Raza declarată a tehnologiei nu înlocuiește măsurarea bugetului real al legăturii.

Gateway-ul este amplasat într-un punct înalt și dispune de conexiune la Internet. Pentru creșterea disponibilității poate fi instalat un al doilea gateway cu acoperire parțial suprapusă.

LoRaWAN utilizează o arhitectură de tip *star-of-stars*. Nodurile nu retransmit mesajele altor noduri.

11.12.2 Fluxul datelor și formatul mesajului

Fluxul unui mesaj este prezentat în Figura 11.18.

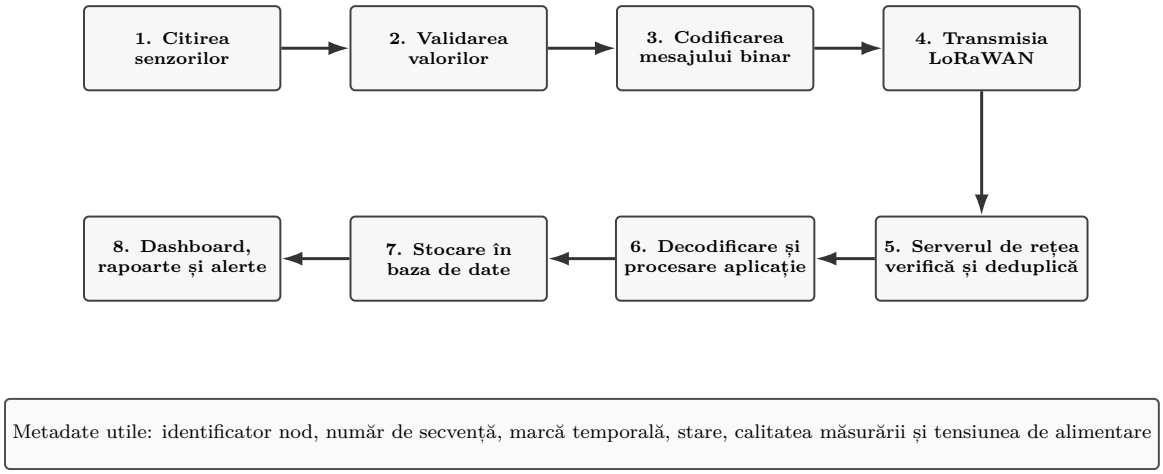


Figura 11.18 Fluxul datelor de la nodul agricol către aplicația utilizatorului.

Secvența de funcționare este:

1. nodul citește senzorii;

2. valorile sunt validate;
3. datele sunt codificate într-un mesaj binar;
4. mesajul este transmis prin LoRaWAN;
5. gateway-ul înaintează cadrul către serverul de rețea;
6. datele aplicației sunt decodificate;
7. mesajul este publicat către serviciul de procesare;
8. valorile sunt stocate și afișate.

Un posibil format al sarcinii utile este prezentat în Tabelul 11.2.

Tabela 11.2 Exemplu de structură binară a mesajului aplicației

Câmp	Lungime	Semnificație
Versiune format	1 octet	Versiunea structurii mesajului
Număr secvență	2 octeți	Detectarea pierderilor și duplicatelor
Marcă temporală	4 octeți	Momentul măsurării
Temperatură	2 octeți	Valoare scalată
Umiditate aer	2 octeți	Valoare scalată
Umiditate sol	2 octeți	Valoare calibrată
Iluminare	2 octeți	Valoare scalată
Tensiune alimentare	2 octeți	Valoare exprimată în milivolți
Stare	1 octet	Alarme și erori locale
Total	18 octeți	Sarcină utilă a aplicației

Identitatea radio și mecanismele de integritate ale cadrului sunt gestionate de stiva LoRaWAN și nu trebuie duplicate fără justificare în fiecare sarcină utilă.

Numărul secvenței permite aplicației să identifice:

- mesaje lipsă;
- retransmisii;
- resetarea neașteptată a nodului;
- ordonarea mesajelor întârziate.

11.12.3 Traficul generat și capacitatea sistemului

Numărul transmisiilor periodice ale unui nod într-o zi este:

$$N_{\text{nod,zi}} = \frac{24 \cdot 60}{15} = 96. \tag{11.31}$$

Pentru 60 de noduri:

$$N_{\text{total,zi}} = 60 \cdot 96 = 5760. \tag{11.32}$$

Volumul zilnic al sarcinilor utile este:

$$\begin{aligned} D_{\text{payload,zi}} &= 5760 \cdot 18\,\text{octeți} \\ &= 103\,680\,\text{octeți} \\ &\approx 101{,}25\,\text{KiB}. \end{aligned} \tag{11.33}$$

Această valoare nu include:

- antetele LoRaWAN;
- timpii de ocupare a canalului;
- retransmisiile;
- mesajele de asociere;
- confirmările;
- traficul dintre gateway și server.

Rata medie a mesajelor este:

$$\begin{aligned} R_{\text{mediu}} &= \frac{5760}{86400\,\text{s}} \\ &\approx 0{,}0667\,\text{mesaje/s}. \end{aligned} \tag{11.34}$$

Media este redusă, dar nodurile nu trebuie programate să transmită exact în același moment. Se introduce o întârziere aleatoare în jurul intervalului nominal:

$$T_{\text{TX}} = T_{\text{nominal}} + \Delta T_{\text{aleator}}, \tag{11.35}$$

pentru reducerea aglomerării canalului.

Figura 11.19 prezintă distribuirea transmisiilor și procesarea mesajelor.

Dacă probabilitatea medie de livrare este p_L, numărul așteptat de mesaje recepționate este:

$$N_{\text{recepționate}} = p_L N_{\text{total}}. \tag{11.36}$$

Pentru:

$$p_L = 0{,}95,$$

rezultă:

$$N_{\text{recepționate,zi}} = 0{,}95 \cdot 5760 = 5472.$$

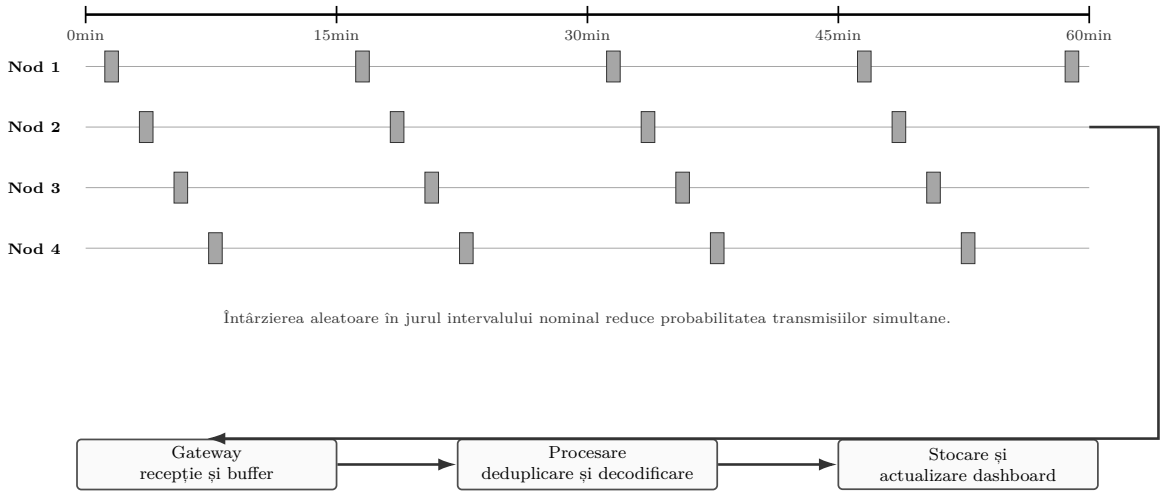


Figura 11.19 Distribuirea în timp a transmisiilor și procesarea datelor.

Aplicația trebuie să tolereze pierderea unor măsurători și să evite utilizarea necontrolată a confirmărilor pentru fiecare mesaj. Confirmările pot crește traficul descendent și consumul nodurilor.

Pentru datele periodice pot fi utilizate mesaje neconfirmate, iar mesajele critice pot utiliza mecanisme suplimentare de confirmare și repetare.

11.12.4 Funcționarea offline, securitatea și mentenanța

Dacă gateway-ul pierde conexiunea la Internet, mesajele pot fi stocate local și transmise după restabilirea legăturii.

Pentru o perioadă offline de T_O zile, capacitatea minimă pentru sarcinile utile este:

$$C_{\text{offline}} \geq T_O D_{\text{payload,zi}}. \tag{11.37}$$

Pentru șapte zile:

$$\begin{aligned} C_{\text{offline}} &\geq 7 \cdot 101{,}25\,\text{KiB} \\ &\approx 709\,\text{KiB}. \end{aligned} \tag{11.38}$$

În practică se rezervă mai mult spațiu pentru metadate, jurnale și formatele utilizate de gateway.

Figura 11.20 prezintă comportamentul sistemului în condiții normale și în absența conexiunii cloud.

Securitatea sistemului include:

- identități și chei unice pentru fiecare nod;
- activare securizată în rețeaua LoRaWAN;

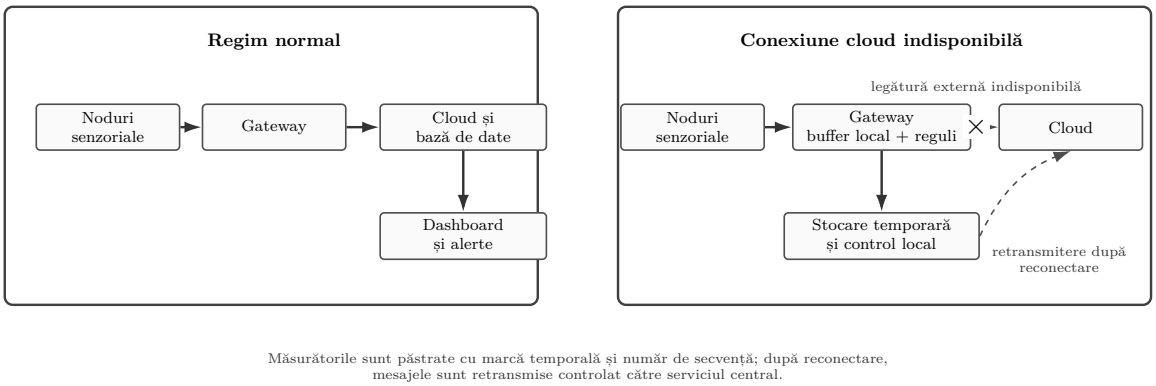


Figura 11.20 Funcționarea sistemului în regim normal și offline.

- protejarea cheilor în memoria dispozitivului;
- autentificarea gateway-ului și a serviciilor;
- comunicație securizată între componentele IP;
- controlul accesului la aplicație;
- jurnalizarea operațiilor.

Resetarea contoarelor de cadre sau reutilizarea necontrolată a cheilor poate afecta protecția anti-replay. Starea necesară trebuie păstrată în siguranță.

Managementul dispozitivelor trebuie să permită:

- observarea nivelului bateriei;
- monitorizarea calității legăturii;
- identificarea nodurilor inactive;
- modificarea intervalului de raportare;
- actualizarea configurației;
- retragerea unui nod compromis.

Actualizarea completă a firmware-ului printr-o rețea cu rată redusă poate consuma timp și energie semnificative. Actualizările trebuie planificate și pot fi realizate:

- local, în timpul operațiilor de mentenanță;
- printr-un canal alternativ;
- prin mecanisme specializate de actualizare la distanță.

Indiferent de metodă, imaginea firmware trebuie semnată și verificată înainte de instalare.

Scalarea sistemului de la 60 la N noduri trebuie să includă:

- traficul radio;
- capacitatea gateway-urilor;
- încărcarea serverului;
- volumul bazei de date;
- numărul alarmelor;
- operațiile de mentenanță.

Un prototip funcțional cu un număr mic de noduri nu garantează automat funcționarea

unei rețele extinse.

11.13 Întrebări și probleme propuse

11.13.1 Întrebări de verificare

1. Care sunt obiectivele principale urmărite prin implementarea unui sistem IoT?
2. Cum poate fi utilizat IoT în mentenanța predictivă?
3. De ce deciziile critice de siguranță nu trebuie să depindă exclusiv de un serviciu cloud?
4. Ce parametri pot fi monitorizați într-o aplicație agricolă?
5. Care este rolul histerezisului într-un sistem automat de irigare?
6. Care sunt principalele componente ale unei arhitecturi LoRaWAN?
7. De ce LoRaWAN nu reprezintă o rețea mesh între nodurile finale?
8. Ce informații trebuie incluse într-un mesaj provenit de la un nod senzorial?
9. Care este rolul numărului de secvență?
10. De ce volumul sarcinii utile este mai mic decât traficul radio real?
11. Cum poate funcționa sistemul atunci când conexiunea la cloud este indisponibilă?
12. Ce mecanisme sunt necesare pentru administrarea sigură a nodurilor pe durata exploatării?

11.13.2 Probleme de calcul și proiectare

1. O rețea conține 80 de noduri care transmit la fiecare 10 minute. Calculați numărul total de mesaje generate într-o zi.
2. Fiecare mesaj conține o sarcină utilă de 24 de octeți. Determinați volumul zilnic al sarcinilor utile pentru rețeaua de la problema precedentă.
3. Un gateway trebuie să funcționeze offline timp de 14 zile. Dacă sistemul generează 250 KiB/zi, determinați capacitatea minimă a bufferului. Introduceți o rezervă de 50%.
4. O rețea generează 10 000 de mesaje pe zi, iar probabilitatea medie de livrare este 92%. Determinați numărul așteptat de mesaje recepționate și numărul așteptat de mesaje pierdute.
5. Proiectați o structură binară de maximum 20 de octeți pentru transmiterea temperaturii, umidității, presiunii, tensiunii bateriei și a două stări de alarmă.

6. Analizați avantajele și dezavantajele utilizării unui singur gateway, respectiv a două gateway-uri cu acoperire suprapusă.

7. Propuneți o strategie de transmitere pentru date periodice, alarme și mesaje de diagnostic. Precizați care mesaje necesită confirmare.

8. Proiectați arhitectura software și de comunicație pentru o fermă cu 200 de noduri, două gateway-uri și o aplicație care trebuie să funcționeze timp de 48 de ore fără conexiune la Internet.

Bibliografie

[1] *AMD64 Architecture Programmer's Manual*, Advanced Micro Devices, 2023.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," *IEEE Communications Surveys & Tutorials*, Vol. 17, No. 4, pp. 2347–2376, 2015.

[3] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 18–20, 1967, Spring Joint Computer Conference*, 1967, pp. 483–485.

[4] G. Anastasi, M. Conti, M. D. Francesco and A. Passarella, "Energy conservation in wireless sensor networks: A survey," *Ad Hoc Networks*, Vol. 7, No. 3, pp. 537–568, 2009.

[5] Apache NuttX Project, *Apache NuttX Documentation*, Apache Software Foundation, 2026, official RTOS and POSIX compatibility documentation.

[6] *AMBA AHB Protocol Specification*, Arm Limited.

[7] *AMBA APB Protocol Specification*, Arm Limited.

[8] *AMBA AXI and ACE Protocol Specification*, Arm Limited.

[9] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, Arm Limited.

[10] *Arm CPU Architecture and Processor Profiles*, Arm Limited, accessed 2026. [Online]. Available: https://www.arm.com/architecture/cpu

[11] *ARMv7-M Architecture Reference Manual*, Arm Limited. [Online]. Available: https://developer.arm.com/documentation/ddi0403/latest/

[12] L. Atzori, A. Iera and G. Morabito, "The internet of things: A survey," *Computer Networks*, Vol. 54, No. 15, pp. 2787–2805, 2010.

[13] M. Ayaz, M. Ammad-Uddin, Z. Sharif, A. Mansour and E.-H. M. Aggoune, "Internet-of-things (IoT)-based smart agriculture: Toward making the fields talk," *IEEE Access*, Vol. 7, pp. 129 551–129 583, 2019.

[14] J. Backus, "Can programming be liberated from the von neumann style? a functional style and its algebra of programs," *Communications of the ACM*, Vol. 21, No. 8, pp. 613–641, 1978.

[15] C. R. Banbury, V. J. Reddi, M. Lam, W. Fu, A. Fazel, J. Holleman, X. Huang, R. Hurtado, D. Kanter, A. Lokhmotov, D. Patterson, D. Pau, J. sun Seo, J. Sieracki, U. Thakker, M. Verhelst and P. Yadav, "Benchmarking tinyml systems," *Proceedings of MLSys*, 2021.

[16] A. Barre, B. Deguilhem, S. Grolleau, M. Gerard, F. Suard and D. Riu, "A review on lithium-ion battery ageing mechanisms and estimations for automotive applications," *Journal of Power Sources*, Vol. 241, pp. 680–689, 2013.

[17] L. Benini, A. Bogliolo and G. D. Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 8, No. 3, pp. 299–316, 2000.

[18] L. Benini and G. D. Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, 1998.

[19] L. Benini and G. D. Micheli, "System-level power optimization: Techniques and tools," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, No. 2, pp. 115–192, 2000.

[20] S. Brown and Z. Vranesic, *Fundamentals of Digital Logic with VHDL Design*, 3rd Ed. McGraw-Hill, 2014.

[21] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd Ed. Springer, 2011.

[22] A. P. Chandrakasan, S. Sheng and R. W. Brodersen, "Low-power cmos digital design," *IEEE Journal of Solid-State Circuits*, Vol. 27, No. 4, pp. 473–484, 1992.

[23] M. Chen and G. A. Rincon-Mora, "Accurate electrical battery model capable of predicting runtime and i–v performance," *IEEE Transactions on Energy Conversion*, Vol. 21, No. 2, pp. 504–511, 2006.

[24] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer, 2009.

[25] S. E. Deering and R. M. Hinden, "Internet protocol, version 6 (IPv6) specification," Internet Engineering Task Force, RFC 8200, 2017.

[26] S. Deng, H. Zhao, W. Fang, J. Yin, S. Dustdar and A. Y. Zomaya, "Edge intelligence: The confluence of edge computing and artificial intelligence," *IEEE Internet of Things Journal*, Vol. 7, No. 8, pp. 7457–7469, 2020.

[27] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam and D. Burger, "Dark silicon and the end of multicore scaling," *Proceedings of the 38th Annual International Symposium on Computer Architecture*, 2011, pp. 365–376.

[28] M. Fagan, K. Megas, K. Scarfone and M. Smith, "IoT device cybersecurity capability core baseline," National Institute of Standards and Technology, NISTIR 8259A, 2020.

[29] J. A. Fisher, P. Faraboschi and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2005.

[30] FreeRTOS, *The FreeRTOS Kernel and Scheduling Documentation*, Amazon Web Services, 2026, official kernel documentation.

[31] FreeRTOS, *FreeRTOS Long Term Support Libraries*, Amazon Web Services, 2026, official LTS documentation.

[32] S. Furber, *ARM System-on-Chip Architecture*, 2nd Ed. Addison-Wesley, 2000.

[33] J. B. Goodenough and K.-S. Park, "The li-ion rechargeable battery: A perspective," *Journal of the American Chemical Society*, Vol. 135, No. 4, pp. 1167–1176, 2013.

[34] J. Gubbi, R. Buyya, S. Marusic and M. Palaniswami, "Internet of things (IoT): A vision, architectural elements, and future directions," *Future Generation Computer Systems*, Vol. 29, No. 7, pp. 1645–1660, 2013.

[35] E. C. Hall, *Journey to the Moon: The History of the Apollo Guidance Computer*. Reston, VA: AIAA, 1996.

[36] S. Harris and D. Harris, *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2015.

[37] S. Hauck and A. DeHon, Eds., *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.

[38] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th Ed. Morgan Kaufmann, 2019.

[39] M. Horowitz, "Computing's energy problem (and what we can do about it)," *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, 2014, pp. 10–14.

[40] X. Hu, S. Li and H. Peng, "A comparative study of equivalent circuit models for li-ion batteries," *Journal of Power Sources*, Vol. 198, pp. 359–367, 2012.

[41] A. Humayed, J. Lin, F. Li and B. Luo, "Cyber-physical systems security–a survey," *IEEE Internet of Things Journal*, Vol. 4, No. 6, pp. 1802–1831, 2017.

[42] *IEEE Standard for Low-Rate Wireless Networks*, IEEE Std. IEEE Std 802.15.4-2020, 2020.

[43] *Intel® 64 and IA-32 Architectures Software Developer's Manual*, Intel Corporation, 2024.

[44] *ISO 26262: Road vehicles – Functional safety*, International Organization for Standardization Std., 2018.

[45] *ISO/SAE 21434: Road vehicles – Cybersecurity engineering*, International Organization for Standardization and SAE International Std., 2021.

[46] J. Jaguemont, L. Boulon and Y. Dube, "A comprehensive review of lithium-ion batteries used in hybrid and electric vehicles at cold temperatures," *Applied Energy*, Vol. 164, pp. 99–114, 2016.

[47] M. B. Jones, "What really happened on mars?" The Risks Digest, Volume 19, Issue 49, 1997, technical account of the Mars Pathfinder priority inversion incident.

[48] A. Kansal, J. Hsu, S. Zahedi and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Transactions on Embedded Computing Systems*, Vol. 6, No. 4, 2007.

[49] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch and S. Winwood, "sel4: Formal verification of an os kernel," *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009, pp. 207–220.

[50] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd Ed. Springer, 2011.

[51] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2007, pp. 21–30.

[52] Linux Kernel Community, *Real-Time Preemption*, The Linux Kernel Documentation, 2026, official PREEMPT_RT architecture documentation.

[53] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, Vol. 20, No. 1, pp. 46–61, 1973.

[54] P. Marwedel, *Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems, and the Internet of Things*, 3rd Ed. Springer, 2018.

[55] P. Marwedel, *Embedded System Design*, 4th Ed. Springer, 2021.

[56] C. Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows.* Newnes, 2004.

[57] D. A. Mindell, *Digital Apollo: Human and Machine in Spaceflight.* MIT Press, 2008.

[58] G. Montenegro, N. Kushalnagar, J. Hui and D. Culler, "Transmission of IPv6 packets over IEEE 802.15.4 networks," Internet Engineering Task Force, RFC 4944, 2007.

[59] B. Moran, H. Tschofenig, D. Brown and M. Meriac, "A firmware update architecture for internet of things," Internet Engineering Task Force, RFC 9019, 2021.

[60] N. Nitta, F. Wu, J. T. Lee and G. Yushin, "Li-ion battery materials: Present and future," *Materials Today*, Vol. 18, No. 5, pp. 252–264, 2015.

[61] T. Noergaard, *Embedded Systems Architecture*, 2nd Ed. Newnes, 2013.

[62] *MQTT Version 5.0*, OASIS Std., 2019, oASIS Standard.

[63] *The Java Virtual Machine Specification, Java SE 24 Edition*, Oracle, 2025. [Online]. Available: https://docs.oracle.com/javase/specs/

[64] J. A. Paradiso and T. Starner, "Energy scavenging for mobile and wireless electronics," *IEEE Pervasive Computing*, Vol. 4, No. 1, pp. 18–27, 2005.

[65] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition*, 2nd Ed. Morgan Kaufmann, 2021.

[66] R. Pawson, "The myth of the harvard architecture," *IEEE Annals of the History of Computing*, 2022.

[67] G. L. Plett, *Battery Management Systems, Volume I: Battery Modeling.* Artech House, 2015.

[68] S. Priya and D. J. Inman, Eds., *Energy Harvesting Technologies.* New York: Springer, 2009.

[69] J. M. Rabaey, A. Chandrakasan and B. Nikolic, *Digital Integrated Circuits: A Design Perspective*, 2nd Ed. Prentice Hall, 2003.

[70] V. Raghunathan, C. Schurgers, S. Park and M. B. Srivastava, “Energy-aware wireless microsensor networks,” *IEEE Signal Processing Magazine*, Vol. 19, No. 2, pp. 40–50, 2002.

[71] P. P. Ray, “A survey on internet of things architectures,” *Journal of King Saud University – Computer and Information Sciences*, Vol. 30, No. 3, pp. 291–319, 2018.

[72] U. Raza, P. Kulkarni and M. Sooriyabandara, “Low power wide area networks: An overview,” *IEEE Communications Surveys & Tutorials*, Vol. 19, No. 2, pp. 855–873, 2017.

[73] T. B. Reddy, Ed., *Linden’s Handbook of Batteries*, 4th Ed. McGraw-Hill, 2011.

[74] S. Roundy, P. K. Wright and J. M. Rabaey, “A study of low level vibrations as a power source for wireless sensor nodes,” *Computer Communications*, Vol. 26, No. 11, pp. 1131–1144, 2003.

[75] T. Sakurai and A. R. Newton, “Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas,” *IEEE Journal of Solid-State Circuits*, Vol. 25, No. 2, pp. 584–594, 1990.

[76] M. Satyanarayanan, “The emergence of edge computing,” *Computer*, Vol. 50, No. 1, pp. 30–39, 2017.

[77] J. Schmalstieg, S. Kabitz, M. Ecker and D. U. Sauer, “A holistic aging model for li(nimnco)o2 based 18650 lithium-ion batteries,” *Journal of Power Sources*, Vol. 257, pp. 325–334, 2014.

[78] seL4 Foundation, *The seL4 Microkernel: Architecture and Assurance Overview*, seL4 Foundation, 2026, official technical overview.

[79] Z. Shelby, K. Hartke and C. Bormann, “The constrained application protocol (CoAP),” Internet Engineering Task Force, RFC 7252, 2014.

[80] W. Shi, J. Cao, Q. Zhang, Y. Li and L. Xu, “Edge computing: Vision and challenges,” *IEEE Internet of Things Journal*, Vol. 3, No. 5, pp. 637–646, 2016.

[81] A. N. Sloss, D. Symes and C. Wright, *ARM System Developer’s Guide: Designing and Optimizing System Software*. Morgan Kaufmann, 2004.

[82] W. Stallings, *Computer Organization and Architecture*, 11th Ed. Pearson, 2019.

[83] S. Sudevalayam and P. Kulkarni, “Energy harvesting sensor nodes: Survey and implications,” *IEEE Communications Surveys & Tutorials*, Vol. 13, No. 3, pp. 443–461, 2011.

[84] V. Sze, Y.-H. Chen, T.-J. Yang and J. S. Emer, “Efficient processing of deep neural networks: A tutorial and survey,” *Proceedings of the IEEE*, Vol. 105, No. 12, pp. 2295–2329, 2017.

[85] F. Tao, H. Zhang, A. Liu and A. Y. C. Nee, “Digital twin in industry: State-of-the-art,” *IEEE Transactions on Industrial Informatics*, Vol. 15, No. 4, pp. 2405–2415, 2019.

[86] J.-M. Tarascon and M. Armand, “Issues and challenges facing rechargeable lithium batteries,” *Nature*, Vol. 414, pp. 359–367, 2001.

[87] M. Tehranipoor and C. Wang, Eds., *Introduction to Hardware Security and Trust*. Springer, 2011.

[88] V. Tiwari, S. Malik and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 2, No. 4, pp. 437–445, 1994.

[89] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*, 2nd Ed. Wiley, 2010.

[90] J. von Neumann, "First draft of a report on the edvac," Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, Pennsylvania, Tech. Rep. Contract No. W-670-ORD-4926, 1945.

[91] P. Warden and D. Situnayake, *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low-Power Microcontrollers*. O'Reilly Media, 2019.

[92] W. Wolf, *Computers as Components: Principles of Embedded Computing System Design*, 3rd Ed. Morgan Kaufmann, 2012.

[93] S. Wolfert, L. Ge, C. Verdouw and M.-J. Bogaardt, "Big data in smart farming: A review," *Agricultural Systems*, Vol. 153, pp. 69–80, 2017.

[94] J. Yick, B. Mukherjee and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, Vol. 52, No. 12, pp. 2292–2330, 2008.

[95] J. Yiu, *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors*, 3rd Ed. Newnes, 2013.

[96] Zephyr Project, *Zephyr Project Documentation*, Linux Foundation, 2026, kernel, architecture, security and platform documentation.