Universitatea “Politehnica” din București

Facultatea de Electronică, Telecomunicații și Tehnologia Informației

***Evaluarea unor caracteristici pentru un microprocesor de uz general de tip RISC***

# Proiect de diplomă

prezentat ca cerință parțială pentru obținerea titlului de

*Inginer* în domeniul *Electronică și Telecomunicații*

programul de studii de licență *Microelectronică, Optoelectronică și Nanotehnologii*

**Conducător științific:**

Prof. dr. ing. Corneliu BURILEANU

**Absolvent:**

Bianca-Alessandra BĂNICĂ

București

2019

Universitatea "Politehnica" din Bucureşti
Facultatea de Electronică, Telecomunicaţii şi Tehnologia Informaţiei
Departamentul **DCAE**

**Anexa 1**

**TEMA PROIECTULUI DE DIPLOMĂ**

a studentului **BĂNICĂ I. Bianca-Alessandra , 441E**

**1. Titlul temei:** Evaluarea unor caracteristici pentru un microprocesor de uz general de tip RISC

**2. Descrierea contribuţiei originale a studentului (în afara părţii de documentare) şi specificaţii de proiectare:**
În cadrul proiectului, propun dezvoltarea unei arhitecturi proprii de microprocesor pe 8 biți, de uz general, de tip RISC, în limbajul de descriere hardware Verilog. Structura sa se bazează pe o implementare hardware minimă, constând din elemente precum: set de 16 registre de uz general, o unitate aritmetico-logică, un contor de program (alocabil dinamic), un registru de fanioane, multiplicator și împărțitor cablate. Microprocesorul va avea la dispoziție un mecanism de răspuns la întreruperi, precum și o stivă hardware. Numărul de instrucțiuni va fi redus (maxim 32), fiind de diferite tipuri: de transfer cu memoria (load-store), aritmetico-logice (doar între registrele interne) și de control (salturi condiționate și necondiționate). Modurile de adresare vor fi adresarea indirectă prin registru, relativă (la contorul de program) și imediată.

Prin scrierea unor programe de test în assembly, folosind propriul setul de instrucțiuni al microprocesorului, voi urmări evaluarea unor aspecte precum funcționalitatea arhitecturii, sincronizările corecte între diferitele blocuri logice componente, frecvența la care operează sau timpii de execuție pentru diverse secvențe de instrucțiuni. Pe baza rezultatelor acestor teste, voi elabora niște concluzii privind performanțele unei astfel de mașini.

**3. Resurse folosite la dezvoltarea proiectului:**
Sublime Text 3 ca editor de text pentru limbajul hardware Verilog, simulatorul Incisive de la Cadence.

**4. Proiectul se bazează pe cunoştinţe dobîndite în principal la următoarele 3-4 discipline:**
Arhitectura Microprocesoarelor, Circuite Integrate Digitale, Arhitectura Sistemelor de Calcul

**5. Proprietatea intelectuală asupra proiectului aparţine:** studentului

**6. Data înregistrării temei:** 2018-11-28 23:37:58

**Conducător(i) lucrare,**
Prof. dr. ing. Corneliu BURILEANU

semnătura: .....

**Student,**

semnătura: .....

**Director departament,**
Prof. dr. ing Claudius DAN

semnătura: .....

**Decan,**
Prof. dr. ing. Cristian NEGRESCU

semnătura: .....

Cod Validare: **4d83d2fa8d**

# Declarație de onestitate academică

Prin prezenta declar că lucrarea cu titlul "Evaluarea unor caracteristici pentru un microprocesor de uz general de tip RISC", prezentată în cadrul Facultății de Electronică, Telecomunicații și Tehnologia Informației a Universității "Politehnica" din București ca cerință parțială pentru obținerea titlului de *Inginer* în domeniul *Electronică și Telecomunicații,* programul de studii *Microelectronică, Optoelectronică și Nanotehnologii*, este scrisă de mine și nu a mai fost prezentată niciodată la o facultate sau instituție de învățământ superior din țară sau străinătate.

Declar că toate sursele utilizate, inclusiv cele de pe Internet, sunt indicate în lucrare, ca referințe bibliografice. Fragmentele de text din alte surse, reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și fac referință la sursă. Reformularea în cuvinte proprii a textelor scrise de către alți autori face referință la sursă. Înțeleg că plagiatul constituie infracțiune și se sancționează conform legilor în vigoare.

Declar că toate rezultatele simulărilor, experimentelor și măsurătorilor pe care le prezint ca fiind făcute de mine, precum și metodele prin care au fost obținute, sunt reale și provin din respectivele simulări, experimente și măsurători. Înțeleg că falsificarea datelor și rezultatelor constituie fraudă și se sancționează conform regulamentelor în vigoare.

**București,** ***iulie 2019***

Absolvent Bianca-Alessandra **BĂNICĂ**

(semnătura în original)

# Cuprins

# Lista figurilor

# Lista tabelelor

# Lista acronimelor

| Acronim | Semnificație - engleză | Semnificație – română |
|---|---|---|
| ALU/UAL | Arithmetic-logic unit | Unitatea aritmetico-logică |
| ASIC | Application-specific integrated circuit | Circuit integrat specific pentru o aplicație |
| CAD | Computer-aided design | Proiectare asistată de calculator |
| CPI | Clock per instruction | Cicli pe instrucțiune |
| DEMUX | Demultiplexer | Demultiplexor |
| DM/MD | Data memory | Memorie de date |
| FPGA | Field-programmable gate array | Arie de porți programabile |
| IR/RI | Instruction register | Registru de instrucțiune |
| ISR | Interrupt service routine | Rutină de răspuns la întrerupere |
| LSR | Left shift register | Registru de deplasare la stânga |
| MUX | Multiplexer | Multiplexor |
| PC | Program counter | Numărător de program |
| PM/MP | Program memory | Memorie de program |
| PWM | Pulse-width modulation | Modulare în lățime a impulsurilor |
| RAM | Random-access memory | Memorie cu acces aleator |
| RASR | Right arithmetic shift register | Registru de deplasare aritmetică la dreapta |
| RISC | Reduced instruction set computer | Calculator cu set redus de instrucțiuni |
| ROM | Read-only memory | Memorie care poate fi doar citită |
| RTL | Register-transfer level | Transfer la nivel de registru |
| SP | Stack pointer | Indicator de stivă |
| TCL | Tool command language | Limbaj de comandă al programului |
| UCS | - | Unitatea de control și sincronizare |

# Introducere

## Motivația alegerii temei

Microprocesoarele, circuite integrate capabile de a interpreta și executa instrucțiunile unei aplicații software, sunt elementul constituent esențial al unei unități centrale de procesare, cu funcții diverse precum controlul diferitelor părți ale mașinii, transferarea de date între memorie și dispozitive de intrare sau de ieșire, sau realizarea de operații aritmetice și logice. Ele există, în ziua de astăzi, în dispozitive digitale din multe domenii, de la calculatoare și electrocasnice, la ceasuri inteligente și automobile. Se poate spune, așadar, că un astfel de circuit este partea integrantă însărcinată cu luarea deciziilor în cadrul funcționării oricărui sistem digital.

La baza alegerii temei acestui proiect a stat provocarea de a implementa, în limbajul de descriere hardware Verilog, un astfel de microprocesor, de uz general, pe 8 biți, care să poată fi folosit în diverse aplicații, și pentru care un compilator pentru limbajele de nivel înalt să fie ușor de implementat. Acest avantaj se datorează atât arhitecturii alese, precum posibilitatea alocării dinamice a numărătorului de program, diferitele tipuri de salturi și moduri de adresare, sau existența unei stive hardware care să asigure un timp mai rapid de execuție decât un potențial acces în memoria de date.

Un microprocesor cu un set instrucțiuni uzuale, care poate executa funcțiile de bază ale unui sistem digital simplu și îi conferă utilizatorului o oarecare libertate în alegerea alocării resurselor interne, prin prisma lipsei de registre dedicate unei anumite funcții în setul de registre, poate fi introdus cu ușurință în orice dispozitiv uzual menit să îndeplinească câteva cerințe elementare.

## Obiective

Scopul principal a fost reprezentat de asigurarea funcționalității arhitecturii, prin implementarea unui sistem de ceasuri cu ajutorul căruia să fie date semnalele de control menite să sincronizeze corect blocurile microprocesorului între ele pentru fiecare instrucțiune în parte. De asemenea, am dorit din start existența unei instrucțiuni prin care numărătorul de program să poată fi alocat în mod dinamic – să poată fi schimbat în timpul execuției unui program, el devenind astfel atribut de arhitectură, și fiind unul dintre cele șaisprezece registre interne generale ale microprocesorului.

Scopuri secundare au fost implementarea unor algoritmi de înmulțire și împărțire în Verilog pentru ca operațiile respective să se facă în mod cablat, posibilitatea alegerii între cel puțin două moduri de adresare pentru salturile condiționate și necondiționate, existența unei instrucțiuni prin care să se facă apel la o subrutină din care apoi să se revină, maparea dispozitivelor de intrare/ieșire în memoria de date și existența unui mod prin care microprocesorul să răspundă la o întrerupere.

## Etape parcurse

Proiectul a pornit de la zero și a constat în mai multe etape, prezentate în diagrama de mai jos:

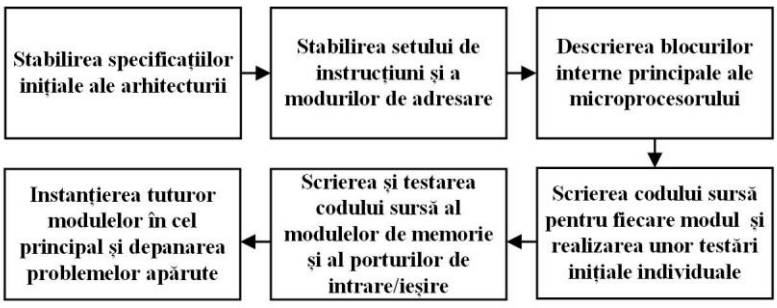


*Figura 0.1* Etape parcurse.

Am creat structura microprocesorului pe baza stabilirii inițiale a arhitecturii alături de conducătorul proiectului, iar apoi am scris codul sursă pentru fiecare bloc component din schema generală de mai jos, în conformitate cu cerințele de funcționare și setul de instrucțiuni ales. Pe lângă modulele constituente ale microprocesorului propriu-zis, am adăugat și un modul de generare a ceasurilor necesare sincronizării corecte interne, o memorie ROM (de program) și una RAM (care conține și porturile mapate în memorie), întrucât arhitectura are la bază modelul Harvard, cu magistrale și memorii separate pentru program și date.

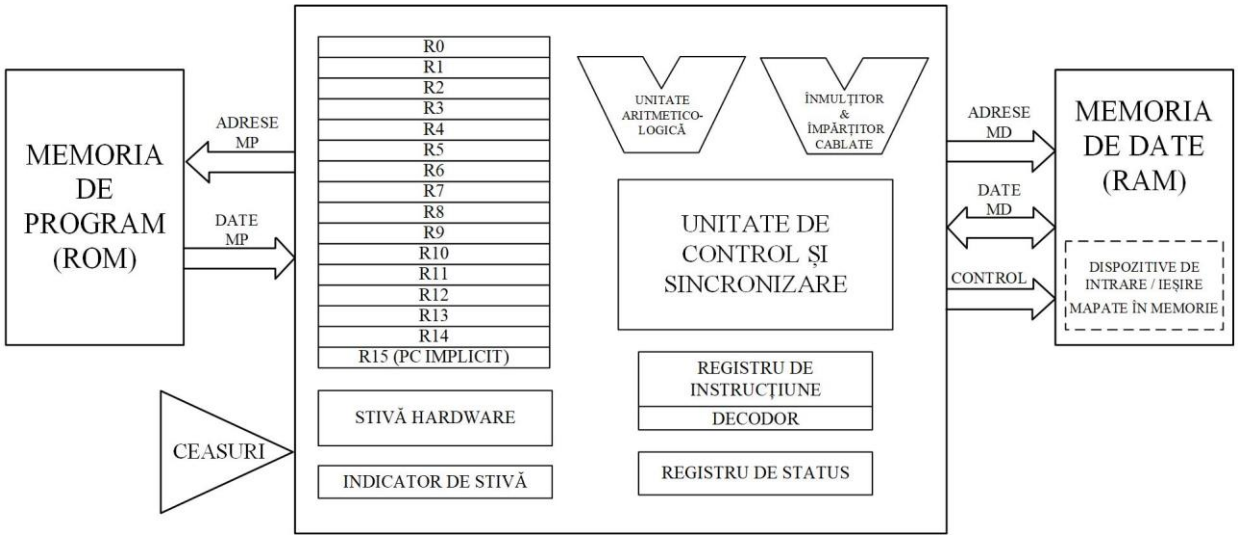


*Figura 0.2* Schema bloc generală.

În urma testării modulului principal al microprocesorului, au apărut o serie de probleme de sincronizare, constând în lucruri precum nesesizarea activării sau dezactivării anumitor semnale de control în funcție de ceasul pe care aceste schimbări aveau loc, pierderea valorilor din anumite registre, incrementarea sau decrementarea indicatorului de stivă la momente nepotrivite de timp, și altele. Depanarea a reprezentat găsirea de soluții diverse (registre ascunse, semnale în plus pentru diferite situații particulare) în scopul execuției fiecărei instrucțiuni în mod corect. Aceste probleme întâmpinate vor fi reluate în sfârșitul lucrării.

În final, am populat memoria ROM cu un program relativ scurt, dar conținând fiecare tip de instrucțiune în limbaj de asamblare din setul de instrucțiuni, pentru a putea testa în mod complet funcționalitatea arhitecturii propuse. De asemenea, am realizat simulări și pentru a verifica faptul că porturile de intrare/ieșire și memoria puteau distinge între adresele care li se adresau lor și cele care nu.

Microprocesorul implementat s-a dovedit a fi, într-adevar, preponderent RISC, prin aspecte precum o anumită uniformitate în timp a instrucțiunilor, o realizare a operațiilor aritmetico-logice strict între registrele generale, o dimensiune fixă a formatului instrucțiunii sau transferul cu memoria doar prin instrucțiuni de tip load și store.

# Capitolul 1 - Descrierea arhitecturii

## 1.1 Atribute de arhitectură

Alegerea arhitecturii microprocesorului a fost prima etapă parcursă, pe baza ei urmând a se scrie modulele Verilog. Am urmărit, în principal, apropierea caracteristicilor sistemului de cele ale unei mașini de tip RISC și, de asemenea, păstrarea unei structuri minimale astfel încât o posibilă implementare hardware ulterioară să se realizeze ușor.

Pentru început, am ales o implementare pe 8 biți, însemnând că dimensiunea operanzilor uzuali, dimensiunea magistralelor de date și de adrese și dimensiunea locațiilor de memorie sunt de 8 biți. În plus, am urmat modelul unei arhitecturi Harvard, menținând memoriile de program și de date separate, așadar și magistralele aferente sunt diferite [1]. Cele mai importante atribute de arhitectură sunt:

- 16 registre de uz general (niciunul nu este dedicat, singurul care are o funcție implicită este $R_{15}$, care este numărătorul de program în caz că nu se folosește instrucțiunea de schimbare a acestuia (PCIS);
- Organizare liniară a memoriei, microprocesorul putând să acceseze toate locațiile de memorie folosind direct adresa fizică pe 8 biți;
- Porturi mapate în memorie – primele 16 locații de memorie (de la 0 la 15) sunt destinate unor dispozitive de intrare/ieșire, astfel încât instrucțiunile prin care se realizează comunicarea cu memoria să poată fi folosite și pentru porturi;
- Singurele instrucțiuni de acces la memorie sunt de tip LOAD și STORE, aspect specific unei arhitecturi de tip RISC, astfel că instrucțiunile de procesări de date și de control al programului sunt exclusiv realizate folosind registrele interne [1];
- Numărător de program care poate fi alocat dinamic, prin instrucțiunea specială PCIS, astfel încât utilizatorului îi rămâne o mai mare libertate în manipularea resurselor interne și a salturilor;
- Un set de instrucțiuni relativ redus (sub 32), ceea ce determină un cod al operației (codul binar care specifică operația ce trebuie executată) de 5 biți;
- Un format al instrucțiunii de 4 octeți, întrucât pentru instrucțiuni de tip MUL sau DIV, sunt specificate adresele a patru registre diferite, necesitând cel puțin 16 biți, iar în plus mai apar câmpuri precum codul operației, modul de adresare sau valori imediate;
- Înmulțitor și împărțitor realizate cablat, pentru numere cu semn, și o unitate aritmetico-logică
- O stivă hardware de 8 locații (o mică memorie RAM dedicată în interiorul microprocesorului) prin care să se poată realiza apeluri și întoarceri din subrutine, sau stocări temporare de date pentru a nu necesita accesul în memorie;
- Un bit în cadrul registrului de stare care să îi semnaleze sistemului venirea unei întreruperi externe;
- Numere pe 8 biți cu semn (în registre, în locațiile de memorie, în stivă sau în înmulțitorul/împărțitorul cablate) și fără semn (în ALU, sau valoarea PC trimisă spre memoria de program).

Alte elemente de bază sunt unitatea de control și sincronizare, care activează sau dezactivează anumite semnale în funcție de instrucțiunea ce trebuie executată, registrul de instrucțiune de 32 de biți și decodorul aferent, și registrul de stare, care conține informații despre fanioane, despre numărătorul de program curent sau despre existența unei întreruperi pe un pin. În Figura 1.1, este reprezentată o schema bloc mai detaliată a microprocesorului, care pune în evidență cele 4 magistrale interne (de date și de adrese pentru memoria de date, de date și de adrese pentru memoria de program) și legăturile blocurilor interne cu ele, câteva semnale de control importante trimise de către UCS (denumite ca în codul sursă și explicate mai jos), semnalele externe de întrerupere și de resetare, ceasurile externe pe care funcționează microprocesorul și împărțirea în module Verilog pe care am ales-o.

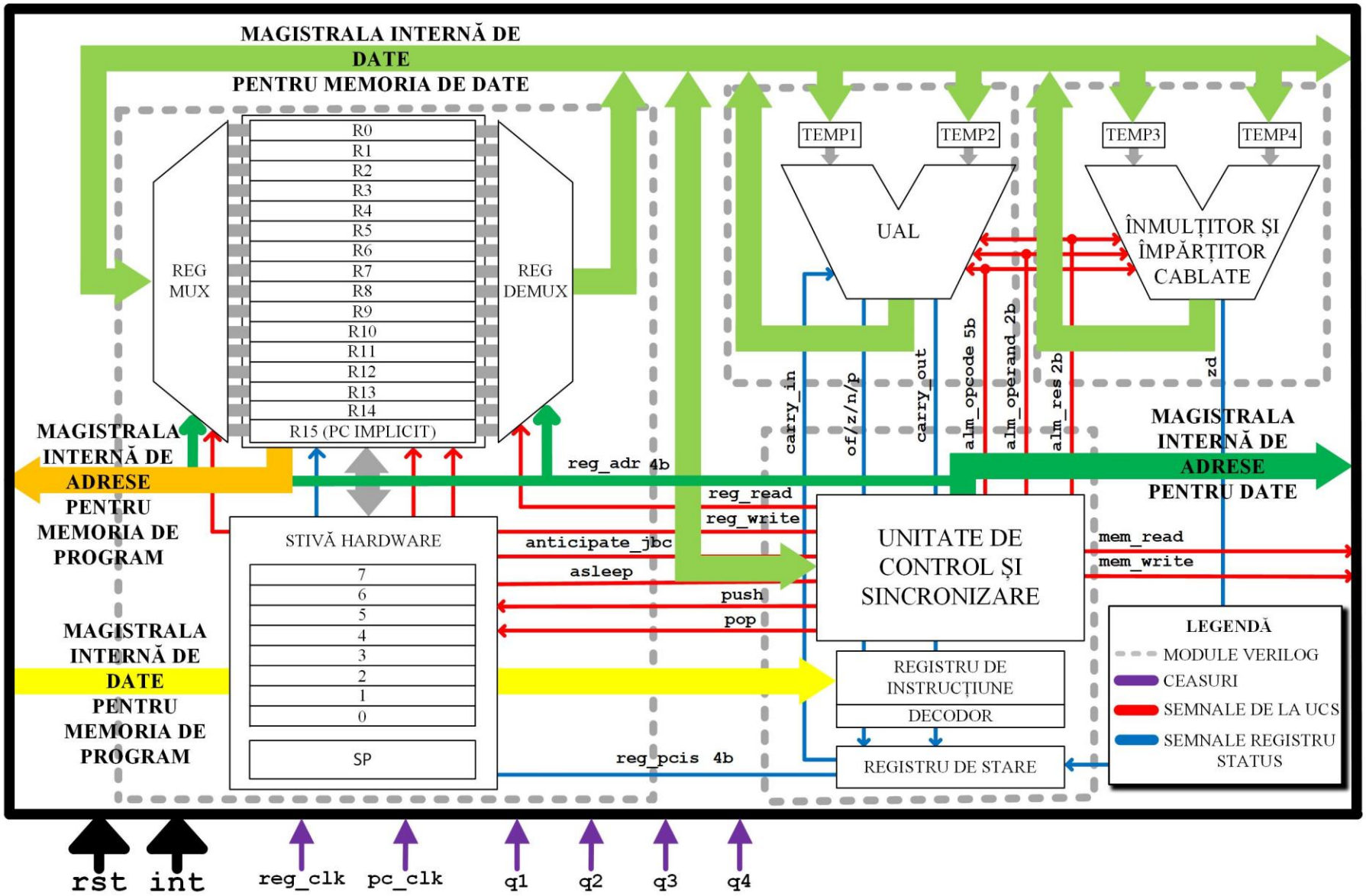


*Figura 1.1* Schema bloc detaliată.

Toate modulele descrise mai jos se folosesc de modulul cu definiții ale constantelor din Anexa 1.

## 1.1 Setul de registre, numărătorul de program și stiva hardware

În Figura 1.2, este ilustrată componența modulului cu setul de registre, numărătorul de program și stiva hardware. Codul sursă pentru acest modul se află în Anexa 2.

Setul de registre comunică atât cu magistrala internă de date, prin intermediul unui multiplexor și al unui demultiplexor comandate de semnalele de citire, scriere și adresă venite de la UCS, cât și cu stiva hardware internă, legătură necesară pentru instrucțiuni de tipul PUSH, POP, CALL sau RET.

În exteriorul modului este trimisă valoarea curentă a numărătorului de program, către memoria de program. Numărătorul de program se incrementează pe fiecare front pozitiv al ceasului de instrucțiune, astfel încât la un anumit moment de timp dat, el conține adresa instrucțiunii imediat următoare celei care se află în registrul de instrucțiune.

Vârful stivei interne este ținut în registrul SP (indicatorul de stivă), el fiind preincrementat (începe de la -1, astfel că prima poziție introdusă în stivă va fi la locația 0) atunci când este necesar să se pună o valoare în stivă, și postdecrementat scoaterii unei valori din stivă.

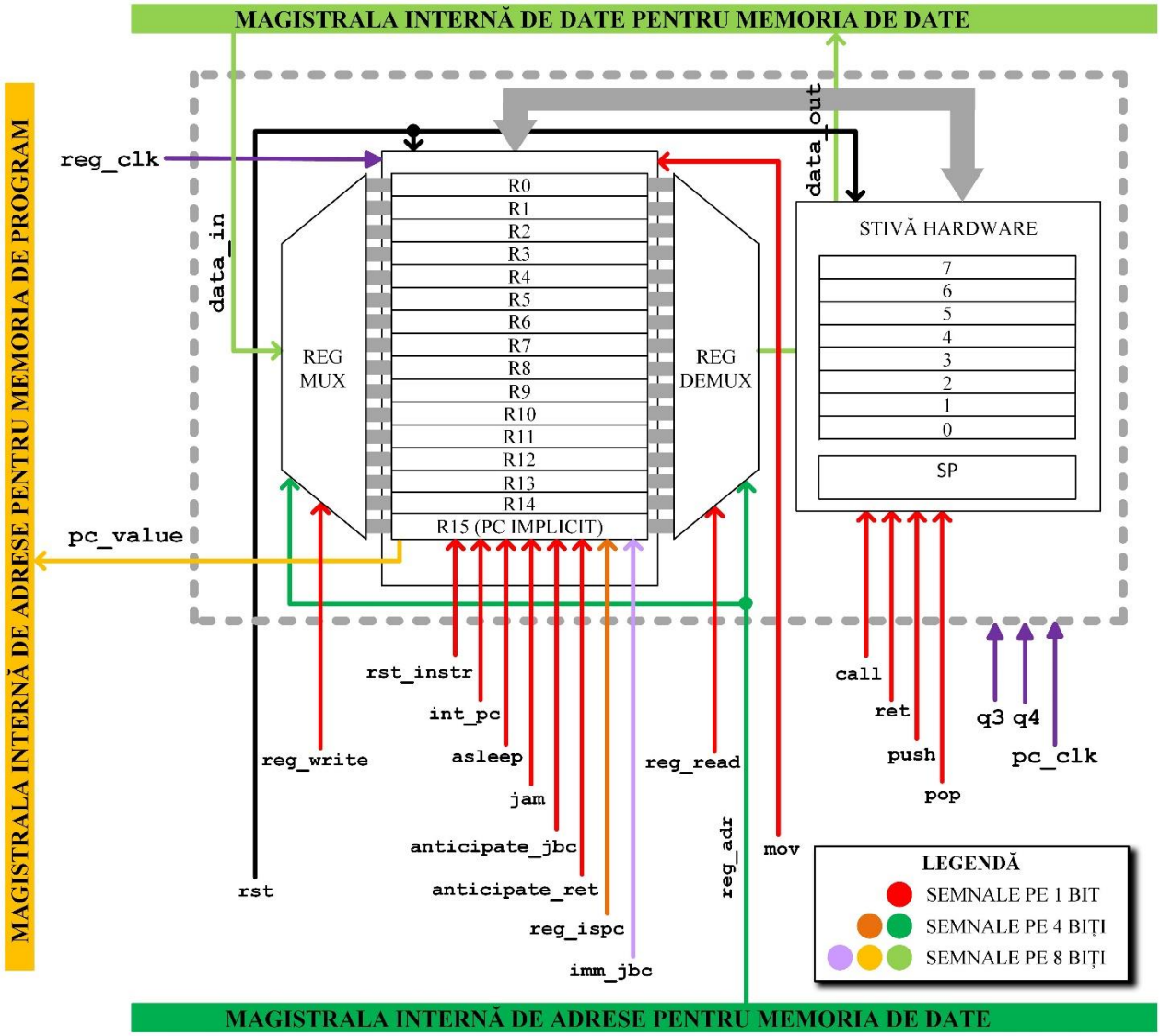


*Figura 1.2* Setul de registre, numărătorul de program și stiva hardware.

Semnalele care intră și ies din modul au fost denumite astfel încât să fie identice cu cele din cod, iar în Tabelul 1.1 este prezentată semnificația fiecăruia dintre ele.

| Semnal | Semnificatie |
|---|---|
| *anticipate_jbc* | Semnal primit de la UCS atunci când următoarea în registrul de instrucțiune este o instrucțiune de salt, astfel încât, la primul front de ceas de instrucțiune, să se sară direct la adresa de salt, pentru a nu fi necesară introducerea unei instrucțiuni NOP după o instrucțiune de salt |
| *anticipate_ret* | Semnal primit de la UCS atunci când următoarea în registrul de instrucțiune este instrucțiunea RET, astfel încât, la primul front de ceas de instrucțiune să se revină la programul principal, pentru a nu fi necesară introducerea unei instrucțiuni NOP după instrucțiunea RET |
| *asleep* | Semnal primit de la UCS care face ca numărătorul de program să se oprească din numărat atunci când este folosită instrucțiunea SLEEP |
| *call* | Semnal primit de la UCS care anunță stiva că este necesară punerea în stivă a adresei instrucțiunii imediat următoare instrucțiunii de salt |
| *data_in* | Intrarea de date legată la magistrala internă de date |
| *data_out* | Ieșirea de date legată la magistrala internă de date |
| *imm_jbc* | Semnal venit de la UCS, care conține fie valoarea imediată ce trebuie adăugată la numărătorul de program atunci când avem un salt cu adresare relativă imediată, fie adresa registrului în care se află adresa de salt atunci când avem adresare implicită în registru |
| *int_pc* | Semnal venit de la UCS, bazat pe fanionul de întrerupere din registrul de stare, care anunță numărătorul de program că trebuie să sară la rutina de deservire a întreruperii, plasată arbitrar la adresa 248 (în zecimal) |
| *jam* | Semnal venit de la UCS, necesar instrucțiunilor de salt și referitor la modul de adresare (0 pentru adresare relativă imediată și 1 pentru adresare implicită în registru), pentru ca numărătorul de program să distingă între a aduna la valoarea sa datele venite la intrare (imm_jbc) sau a lua valoarea dintr-un anumit registru |
| *mov* | Semnal venit de la UCS la instrucțiunea MOV, pentru ca registrul folosit în instrucțiune să ia datele din alt registru, nu de pe intrarea de date |
| *pc_clk* | Ceasul de instrucțiune (pe care funcționează numărătorul de program și stiva hardware) |
| *pc_value* | Valoarea numărătorului de program, trimisă la memoria de program |
| *pop* | Semnal venit de la UCS la instrucțiunea POP, pentru ca registrul folosit în instrucțiune să ia datele din vârful stivei, nu de pe intrarea de date |
| *push* | Semnal venit de la UCS la instrucțiunea PUSH, pentru ca în vârful stivei să fie pusă valoarea conținută în registrul folosit în instrucțiune |
| *q3, q4, reg_clk* | Ceasurile de stare 3 și 4 și ceasul de registru, venite extern |
| *reg_adr* | Adresa unui registru folosit în instrucțiune, necesară demultiplexorului la scriere sau multiplexorului la citire |
| *reg_ispc* | Semnal venit de la registrul de stare, reprezentând adresa registrului care este numărător de program la momentul curent |
| *reg_read* | Semnal venit de la UCS pentru a permite citirea unui registru (citire asincronă) |
| *reg_write* | Semnal venit de la UCS pentru a permite scrierea într-un registru (scriere sincronă, pe ceasul de registru) |
| *ret* | Semnal venit de la UCS pentru ca din stivă să fie scoasă valoarea următoare a numărătorului de program |
| *rst* | Semnal de resetare extern |
| *rst_instr* | Semnal de resetare pentru numărătorul de program aferent instrucțiunii RST |

*Tabelul 1.1* Semnalele blocului cu setul de registre, numărătorul de program și stiva hardware.

Diferențierea între registrele generale disponibile și registrul general care este numărătorul de program la momentul curent se face prin intermediul semnalului de 4 biți *reg_ispc* din tabelul de mai sus, după cum apare în codul ilustrat mai jos. De asemenea, prefixul *rpd* apare în fața tuturor semnalelor de mai jos deoarece semnalele care sunt comune mai multor module au primit un prefix corespunzător modului în care se află, pentru ca procesul de depanare să fie mai ușor, însă după prefix ele respectă întocmai forma din tabelul de mai sus.

```
always @ (posedge rpd_reg_clk or posedge rpd_rst) begin
   if (rpd_rst) begin
      for (i=0; i<16; i=i+1)
      reg_file[i] <= 8'd0;
   end
   else begin
      if (rpd_reg_adr !== rpd_reg_ispc)
         reg_file[rpd_reg_adr] <= rpd_reg_1;
      if (q4 && rpd_rst_instr) reg_file[rpd_reg_ispc] <= 0;
      else if (q4) reg_file[rpd_reg_ispc] <= next_pc_int;
      else reg_file[rpd_reg_ispc] <= reg_file[rpd_reg_ispc];
   end
end

//////REGISTRE//////
assign rpd_reg_1 = (rpd_mov && (rpd_reg_adr != rpd_reg_ispc) && rpd_reg_write)
?
   hidden_temp : rpd_reg_2; //instrucțiunea MOV

assign rpd_reg_2 = (rpd_pop && (rpd_reg_adr != rpd_reg_ispc)  &&
rpd_reg_write) ?
   hardware_stack[stack_pointer] : rpd_reg_3; //instrucțiunea POP

assign rpd_reg_3 = ((rpd_reg_adr != rpd_reg_ispc) &&
   rpd_reg_write) ? rpd_data_in : reg_file[rpd_reg_adr]; //scriere simplă

//////PC//////
assign next_pc_int = rpd_int_pc ? 8'd248 : next_pc; //întrerupere

assign next_pc = rpd_asleep ? reg_file[rpd_reg_ispc] : pc_jbc;
   //instrucțiunea SLEEP

assign pc_jbc = (rpd_anticipate_jbc && !rpd_jam) ?
   (reg_file[rpd_reg_ispc] + rpd_imm_jbc) : ((rpd_anticipate_jbc
   && rpd_jam) ? reg_file[rpd_imm_jbc] : pc_ret);
   //salt relativ imediat sau implicit în registru

assign pc_ret = rpd_anticipate_ret ?
   hardware_stack[stack_pointer] : reg_file[rpd_reg_ispc] + 4;
   //instrucțiunea RET
```

*Figura 1.3* Cod sursă pentru setul de registre.

În Figura 1.4 este prezentat modul de atribuire pentru indicatorul vârfului stivei, care este preincrementat la scrierea în stivă și postdecrementat la scoaterea unor valori din stivă [1].

```
always @ (posedge rpd_reg_clk or posedge rpd_rst) begin
   if (rpd_rst) stack_pointer <= -3'd1;
   else stack_pointer <= stack_pointer_value;
end
assign stack_pointer_value = stack_modify_up ? stack_pointer + 1
   : (stack_modify_down ? stack_pointer - 1 : stack_pointer);
assign stack_modify_up = (rpd_push || rpd_call || rpd_int_pc) && q3;
assign stack_modify_down = (rpd_pop || rpd_ret) && q4;
```

*Figura 1.4* Cod sursă pentru indicatorul de stivă SP.

## 1.2 Unitatea aritmetico-logică

În schema bloc de mai jos este ilustrată componența modulului cu unitatea aritmetico-logică. Codul sursă pentru acest modul se află în Anexa 3.

Comunicarea cu magistrala internă de date se realizează printr-un port de intrare și unul de iesire. În interior, există două registre temporare menite să stocheze operanzii veniți pe magistrala de date la momentul potrivit. Există 5 fanioane aritmetico-logice care sunt trimise spre registrul de stare.

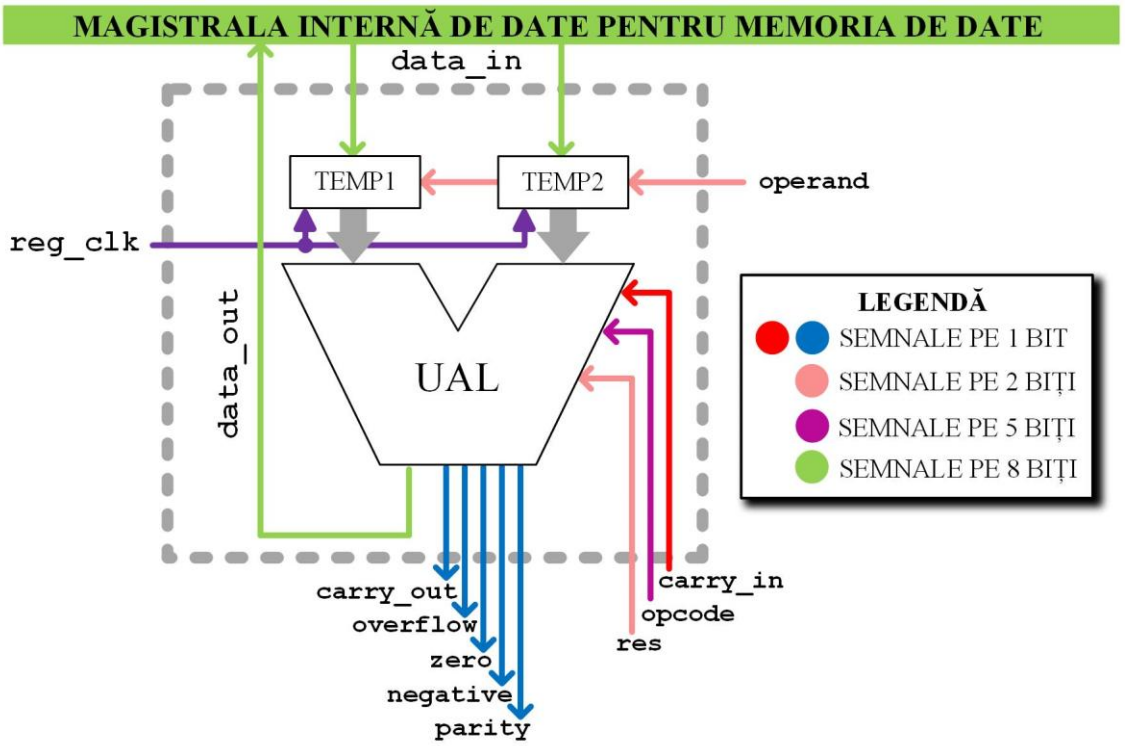


*Figura 1.5* Unitatea aritmetico-logică.

În Tabelul 1.2 se află toate semnalele de intrare și de iesire aferente acestui modul, denumite ca în codul sursă, și semnificațiile lor.

| Semnal | Semnificatie |
|---|---|
| *carry_in* | Fanionul de transport de intrare, venit de la registrul de stare, pentru ADC și SBC |
| *carry_out* | Fanionul de transport de ieșire |
| *data_in* | Intrarea de date legată la magistrala internă de date |
| *data_out* | Ieșirea de date legată la magistrala internă de date |
| *negative* | Fanionul negativ, care se setează dacă rezultatul operației este un număr negativ |
| *opcode* | Codul operației ce trebuie executată |
| *operand* | Semnal de la UCS prin care se pun operanzii în registrele temporare |
| *overflow* | Fanionul de depășire, care marchează (pentru numerele cu semn) depășirea domeniului de reprezentare |
| *parity* | Fanionul de paritate, care este setat dacă în rezultat avem un număr par de biți de 1 și 0 în caz contrar |
| *reg_clk* | Ceasul de registru, necesar pentru TEMP1 și TEMP2, deși ALU este combinațională |
| *res* | Semnal de la UCS prin care se pune rezultatul pe magistrala internă de date la momentul potrivit |
| *zero* | Fanionul de zero, modificat de instrucțiunile SUB și CP, când rezultatul scăderii e 0 |

*Tabelul 1.2* Semnalele blocului ALU.

Figura 1.6 ilustrează modul în care cele două registre temporare din schema bloc sunt atribuite în funcție de instrucțiunea ce trebuie executată și valoarea semnalului de control *operand*, prefixul *alu* datorându-se faptului că semnalele fac parte din acest bloc.

```
always @(posedge reg_clk) begin
   case (alu_opcode)
      `ADD, `ADC, `SUB, `SBC, `INC, `DEC, `OR, `AND, `XOR, `CP: //diadice
         begin
            if (alu_operand == 2'b01) alu_op_1 <= alu_data_in;
            else if (alu_operand == 2'b10) alu_op_2 <= alu_data_in;
            else begin alu_op_1 <= 8'd0; alu_op_2 <= 8'd0; end
         end
      `SHR, `SHL, `ASR: //monadice
         begin
            if (alu_operand == 2'b01) alu_op_1 <= alu_data_in;
            else alu_op_1 <= 8'd0;
         end
      default: alu_op_1 <= 8'd0; alu_op_2 <= 8'd0;
   endcase
end
```

*Figura 1.6* Cod sursă pentru registrele temporare din ALU.

Valoarea 1 pentru *operand* semnifică preluarea primului operand de pe magistrala de date, valoarea 2 preluarea celui de-al doilea operand, iar valoarea 0 înseamnă că ALU nu trebuie să mai scrie noi valori în registrele temporare.

## 1.3 Înmulțitorul și împărțitorul cablate

Figura 1.7 este ilustrează componența modulului cu înmulțitorul/împărțitorul cablate, al cărui cod sursă este în Anexa 4.

Comunicarea cu magistrala internă de date se realizează printr-un port de intrare și unul de iesire. Exista două registre temporare menite să preia valorile operanzilor de pe magistrală la momentul potrivit, pe ceasul de registru. Blocul în sine este unul combinațional, ce are rezultatul gata odată ce intrările de date sunt salvate ca operanzi. Aferent operației de împărțire este și fanionul de împărțire la zero, trimis mai departe spre registrul de stare.

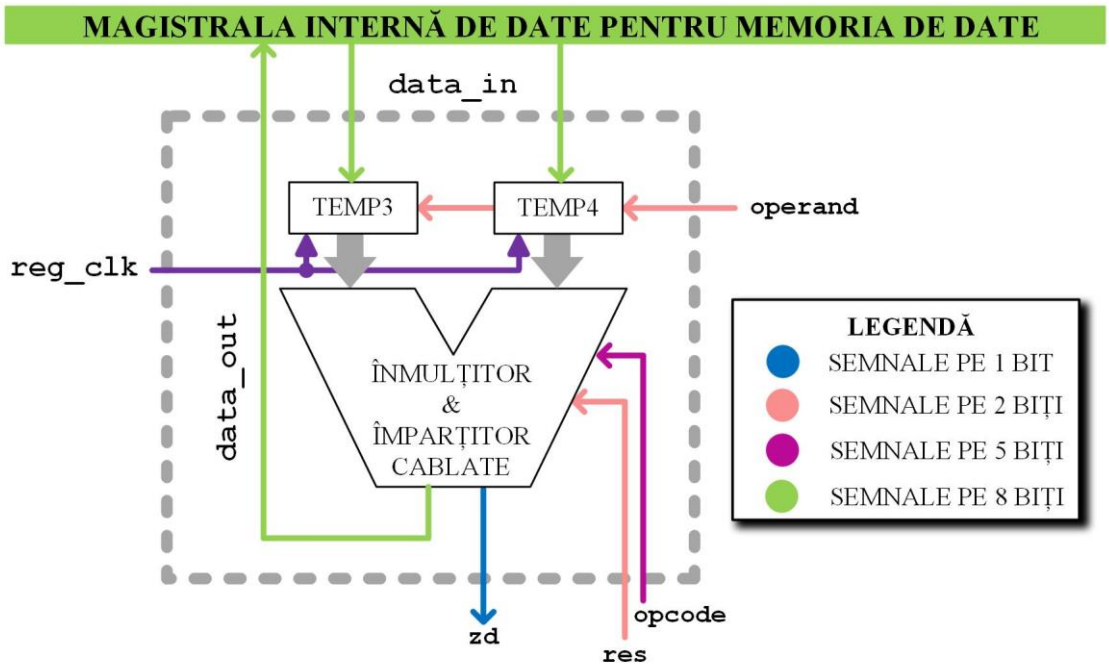


*Figura 1.7* Înmulțitorul și împărțitorul cablate.

Tabelul 1.3 prezintă toate semnalele de intrare și de ieșire aferente acestui modul, denumite ca în codul sursă, și semnificațiile lor.

| Semnal | Semnificație |
|---|---|
| *data_in* | Intrarea de date legată la magistrala internă de date |
| *data_out* | Ieșirea de date legată la magistrala internă de date |
| *opcode* | Codul operației ce trebuie executată |
| *operand* | Semnal de la UCS prin care se pun operanzii în registrele temporare |
| *reg_clk* | Ceasul de registru |
| *res* | Semnal de la UCS prin care se pune rezultatul potrivit pe magistrala internă de date la momentul potrivit |
| *zd* | Fanionul de împărțire la zero |

*Tabelul 1.3* Semnalele blocului cu înmulțitorul și împărțitorul cablate.

Blocurile care realizează înmulțirea și împărțirea numerelor cu semn au fost alese de tip cablat, caracteristică specifică RISC, și implementare care, deși este dedicată unui anumit număr de biți și poate realiza o singură operație, neputând fi schimbată la nivel software, poate fi mult mai rapidă, timpul de execuție fiind dat exclusiv de timpul de propagare al semnalelor prin porți [1].

Mai jos este ilustrat modul în care acest bloc pune pe magistrala internă de date rezultatul operației, în funcție de semnalul de control *res*.

```
assign mul_div_data_out = (mul_div_opcode == `MUL) ?
   ((mul_div_res == 2'b01) ? product[15:8] :
   ((mul_div_res == 2'b10) ? product[7:0] : 8'bz)) :
   ((mul_div_opcode == `DIV) ? ((mul_div_res == 2'b01)? quotient :
   ((mul_div_res == 2'b10)? remainder : 8'bz)) : 8'bz);
```

*Figura 1.8* Cod sursă pentru trimiterea rezultatelor instrucțiunilor MUL și DIV spre registre.

Așadar, pentru operația de înmulțire, pe *res* 1 punem partea superioară a rezultatului, iar pe *res* 2 punem partea inferioară, iar pentru împărțire, pe *res* 1 avem câtul trimis pe magistrala de date, iar pe *res* 2 restul împărțirii.

### 1.3.1 Înmulțitorul

Pentru înmulțire, am implementat algoritmul lui Booth, întrucât acesta reduce, în cele mai multe cazuri, numărul de adunări parțiale necesare. El se bazează pe faptul că orice secvență de biți de 1 dintr-un număr binar poate fi reprezentată ca diferența a două numere binare [2]. Acest lucru este exemplificat în Figura 1.9.

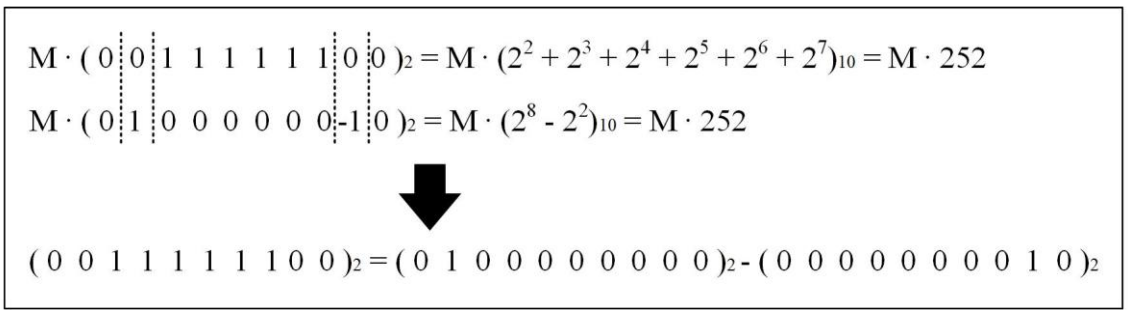


*Figura 1.9* Blocuri de 1 în codurile binare.

Așadar, orice bloc de biți de 1 înconjurat de biți de 0 va putea fi echivalat cu diferența următoarelor numere binare: primul cu un bit de 1 pe poziția ultimului bit de 0 dinaintea șirului de 1 din codul binar inițial, iar al doilea cu un bit de 1 pe poziția ultimului bit de 1 din șir. În zecimal, analog acestei afirmații ar fi faptul că înmulțirea cu 63 este același lucru cu înmulțirea cu diferența (64 - 1) [2].

Întrucât microprocesorul realizat folosește, în principal, numere cu semn, putem spune că, în funcție de frecvența apariției numerelor negative, care, dacă sunt relativ mici, au mulți biți de 1 pe pozițiile dinspre cel mai semnificativ bit, algoritmul lui Booth este de preferat, fiind foarte eficient când vine vorba de blocuri de 1 înconjurate de biți de 0.

În Figura 1.10, este prezentată diagrama pașilor algoritmului. Pe baza celor discutate mai sus, trebuie evaluate perechi de câte 2 biți din înmulțitor, pentru a putea distinge unde încep șiruri de 1 și unde se termină. În funcție de asta, la produs se adaugă sau se scade (mai bine spus, se adaugă cu semn opus) deînmulțitul [3]. Această evaluare are loc de un număr de ori egal cu numărul de biți ai factorilor, în cazul nostru 8. Mărimile cu care operează algoritmul diferă printr-un bit în cazul în care deînmulțitul este valoarea maximă negativă reprezentabilă pe 8 biți, pentru ca rezultatul operației să fie corect [4]. La sfârșit, din valoarea de final se elimină bitul cel mai puțin semnificativ, ceilalți 16 biti reprezentând valoarea produsului celor 2 operanzi.

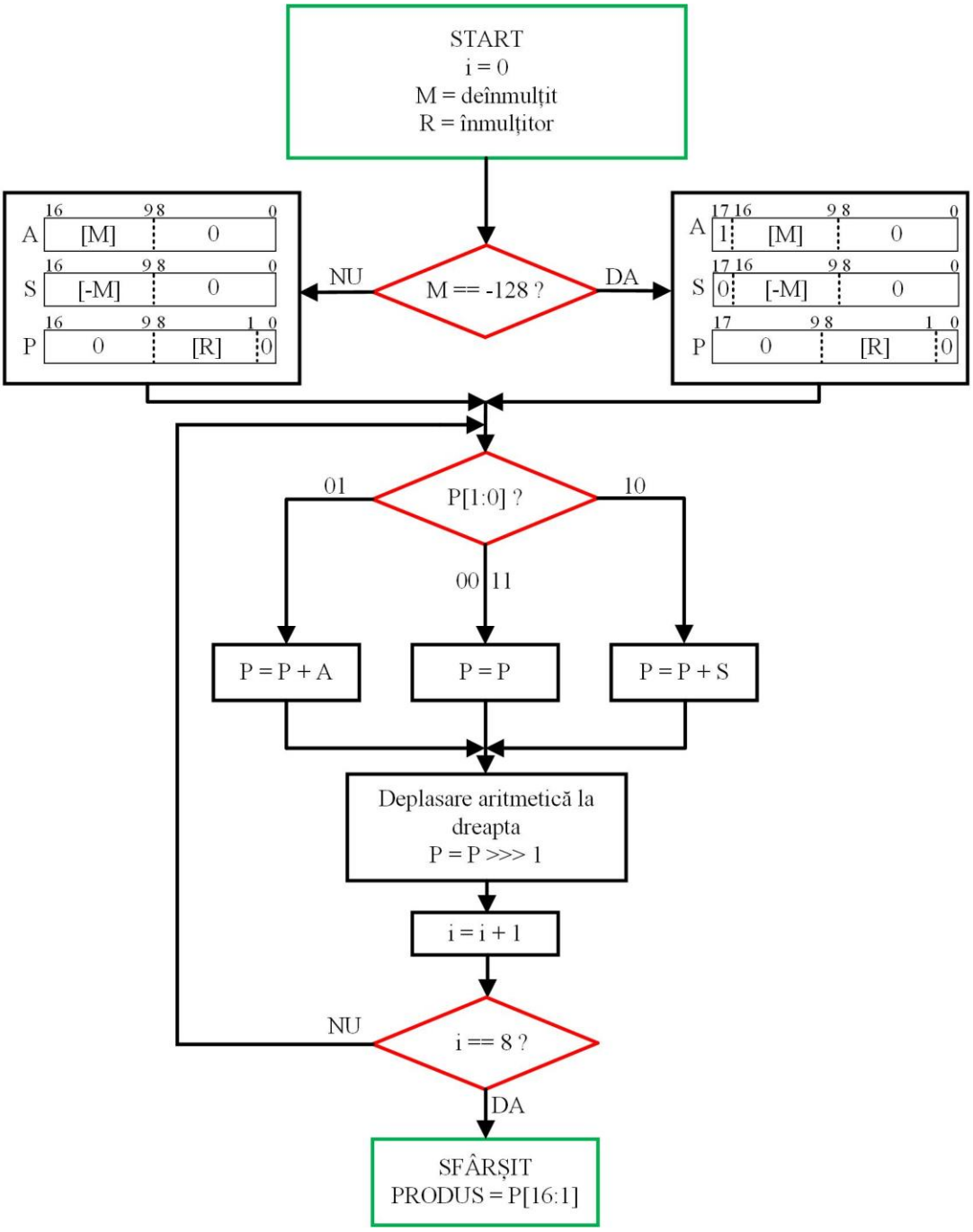


*Figura 1.10* Algoritmul de înmulțire Booth.

Mai jos este redat un exemplu concret al înmulțirii folosind algoritmul lui Booth.

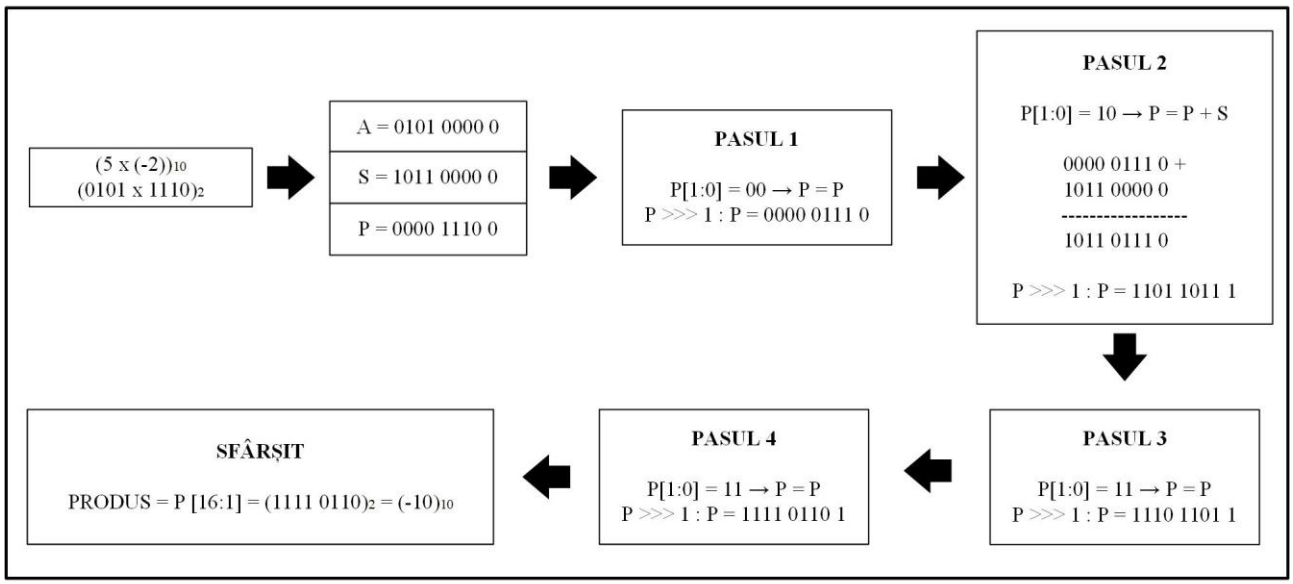


*Figura 1.11* Exemplu de înmulțire Booth.

În continuare sunt exemplificate 2 cazuri, unul în care algoritmul este foarte eficient, și anume când avem o secvență semnificativă de biți de 1 în înmulțitor, și altul în care avem scenariul cel mai pesimist pentru algoritm: înmulțitorul are biți de 1 și de 0 care alternează.

| | **$(5 \times (-4))_{10}$**<br>**$(00000101 \times 11111100)_2$** | **$(6 \times 85)_{10}$**<br>**$(00000110 \times 01010101)_2$** |
|---|---|---|
| **A** | **00000101 00000000 0** | **00000110 00000000 0** |
| **S** | **11111011 00000000 0** | **11111010 00000000 0** |
| **P** | **00000000 11111100 0** | **00000000 01010101 0** |
| Pas 1 | P[1:0] = 00 → P = P, deplasare | P[1:0] = 10 → P = P + S, deplasare |
| Pas 2 | P[1:0] = 00 → P = P, deplasare | P[1:0] = 01 → P = P + A, deplasare |
| Pas 3 | P[1:0] = 10 → P = P + S, deplasare | P[1:0] = 10 → P = P + S, deplasare |
| Pas 4 | P[1:0] = 11 → P = P, deplasare | P[1:0] = 01 → P = P + A, deplasare |
| Pas 5 | P[1:0] = 11 → P = P, deplasare | P[1:0] = 10 → P = P + S, deplasare |
| Pas 6 | P[1:0] = 11 → P = P, deplasare | P[1:0] = 01 → P = P + A, deplasare |
| Pas 7 | P[1:0] = 11 → P = P, deplasare | P[1:0] = 10 → P = P + S, deplasare |
| Pas 8 | P[1:0] = 11 → P = P, deplasare | P[1:0] = 01 → P = P + A, deplasare |
| Total | O adunare | Opt adunări |

*Tabelul 1.4* Comparație pe exemple concrete de înmulțire Booth.

Pentru cazul pesimist, există o variantă modificată a algoritmului Booth, bazată pe recodarea perechilor de biți. Ea constă în introducerea unui 0 în partea mai puțin semnificativă a înmulțitorului și extinderea semnului acestuia cu o poziție, codarea Booth, iar apoi gruparea fiecăror 2 biți de la dreapta la stânga pentu a obține valorile cu care trebuie înmulțit deînmulțitul, după cum apare în Figura 1.12. După obținerea acestor valori, rezultatele parțiale se însumează, iar rezultatul va fi obținut în forma complement față de 2 [5].

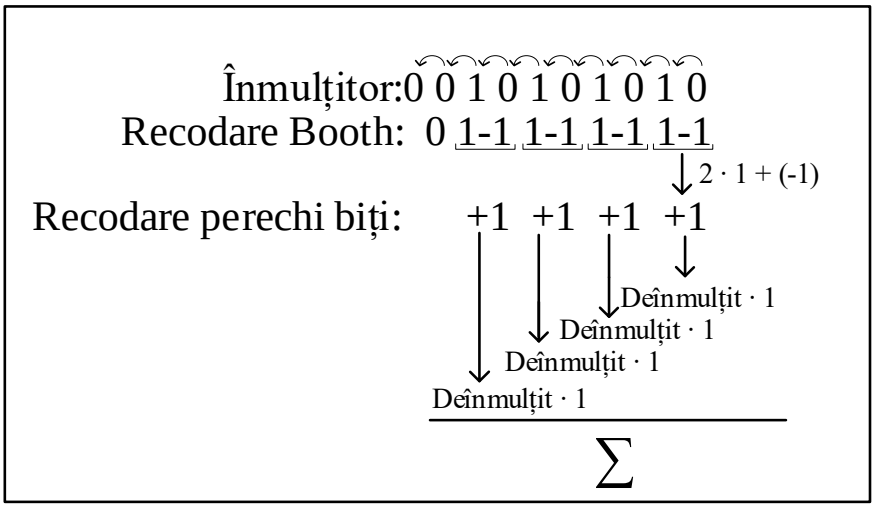


*Figura 1.12* Recodarea perechilor de biți pentru algoritmul Booth.

Pentru exemplul concret de mai sus, acest algoritm modificat va duce, așadar, la 4 adunări, față de 8 adunări pentru algoritmul Booth clasic. Al doilea scenariu negativ este pentru valoarea zecimală 170 (10101010 în binar), adică cealaltă alternanță a biților de 0 și de 1.

Microprocesorul implementează varianta clasică a algoritmului, întrucât am considerat că situațiile cele mai pesimiste (de alternare continuă a biților de 0 și de 1 în înmulțitor) sunt doar două, așadar le întâlnim rar, în rest cei doi algoritmi având performanțe similare. În plus, varianta modificată necesită o logică suplimentară, ce poate fi implementată la o posibilă variantă ulterioară a microprocesorului, menținând deocamdată o structură minimală, deoarece scopul principal în această lucrare este strict apropierea arhitecturii de una de tip RISC.

Figura 1.13 ilustrează schița unei posibile implementări hardware cablate, conținând sumatoarele de care este nevoie, multiplexoare cu selecția dată de biții cei mai puțin semnificativi ai lui P și registrele de deplasare aritmetică la dreapta. Multiplexoarele au 4 intrări de date (dintre care două legate la masă) deoarece selecția lor este pe 2 biți – un multiplexor cu selecția pe n biți poate alege între $2^n$ intrări [6].

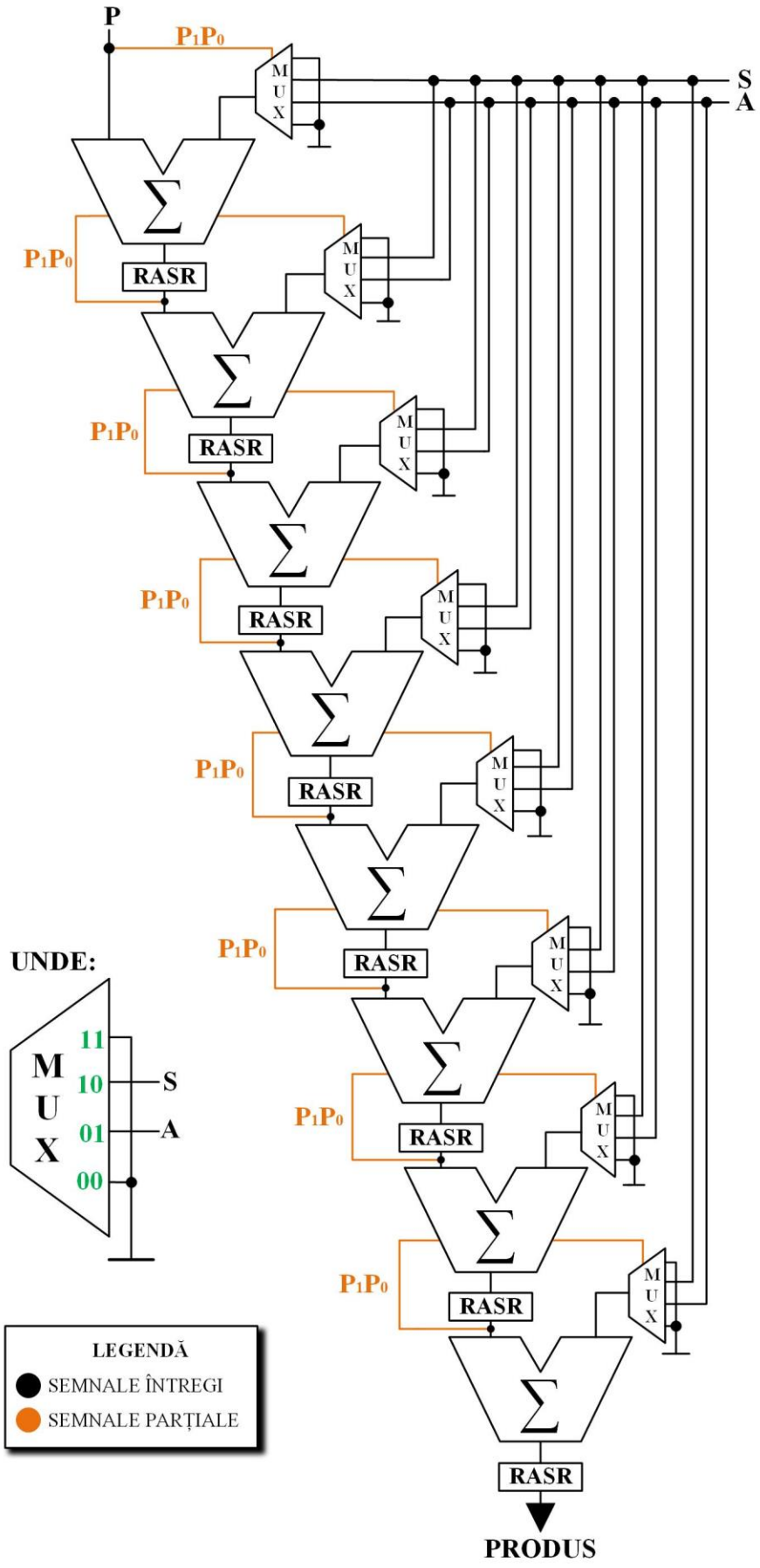


*Figura 1.13* Schemă hardware pentru înmulțitorul Booth cablat.

### 1.3.2 Împărțitorul

Împărțitorul cablat are la bază algoritmul clasic de împarțire, analog celui din zecimal, în care se ia pe rând fiecare bit din deîmpărțit, iar valoarea obținută se împarte la împărțitor [3]. Bitul curent al câtului va fi 0 dacă valoarea obținută după coborârea bitului corespunzător din deîmpărțit este mai mică decât împărțitorul și 1 în caz contrar, după cum este ilustrat în exemplul de mai jos, în care se împart echivalentele binare ale numerelor 146 și 73.

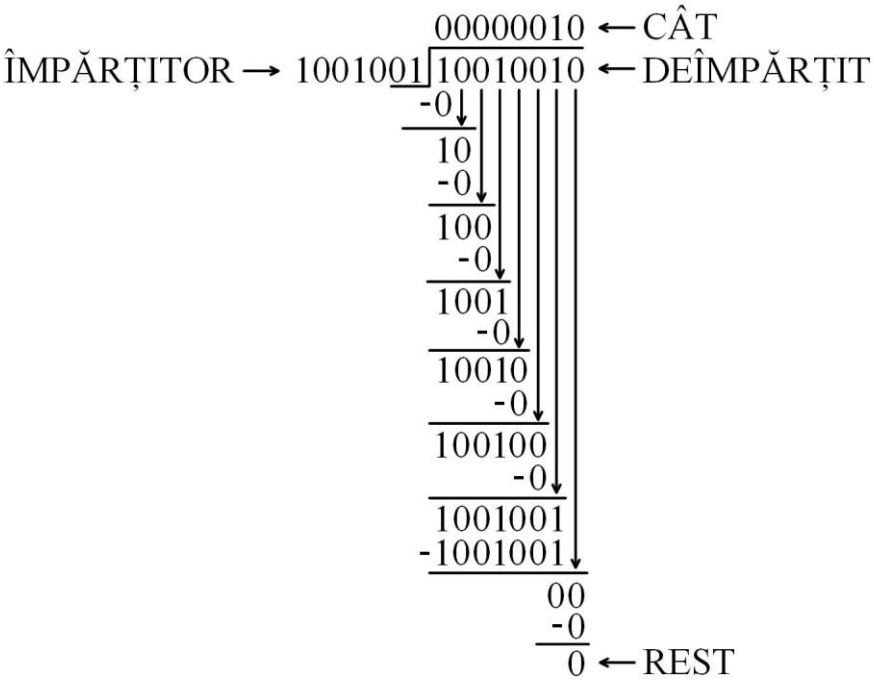


*Figura 1.14* Exemplu de împărțire binară.

Mai jos sunt codul sursă care implementează algoritmul de împărțire binar (în Figura 1.14), și diagrama pașilor aferentă acestuia (Figura 1.15). Se folosește variabila A (*div_a*) inițializată cu 0 care, concatenată cu deîmpărțitul, va fi deplasată la stânga la începutul fiecărui pas, echivalent cu a lua fiecare bit din deîmpărțit, iar din ea se va scădea valoarea împărțitorului pentru a evalua dacă această diferență este sau nu mai mică decât 0, adică dacă bitul curent al câtului va fi 0 sau 1. Algoritmul folosit este pentru numere pozitive, motiv pentru care primul pas este ca variabilele M (*div_m*) și Q (*div_q*) să ia valoarea modulului deîmpărțitului și împărțitorului, iar, în final, semnele câtului și restului se ajustează în funcție de semnele acestora.

```
div_a = 0; //A inițializat cu 0
div_q = (mul_div_op_1 < 0) ? -mul_div_op_1 : mul_div_op_1;//modul
div_m = (mul_div_op_2 < 0) ? -mul_div_op_2 : mul_div_op_2;//modul
for (i=0; i<8; i=i+1) begin
   {div_a, div_q} = {div_a, div_q} << 1; //deplasarea la stânga

   div_a = div_a - div_m; //scăderea împărțitorului din A

   div_q[0] = (div_a < 0) ? 0 : 1; //atribuirea biților din cât
   div_a = (div_a < 0) ? div_a + div_m : div_a; //restaurarea A
end
quotient = (mul_div_op_1[7] != mul_div_op_2[7]) ? -div_q : div_q;
remainder = (mul_div_op_1 < 0) ? -div_a : div_a;
```

*Figura 1.15* Cod sursă pentru algoritmul de împărțire binară.

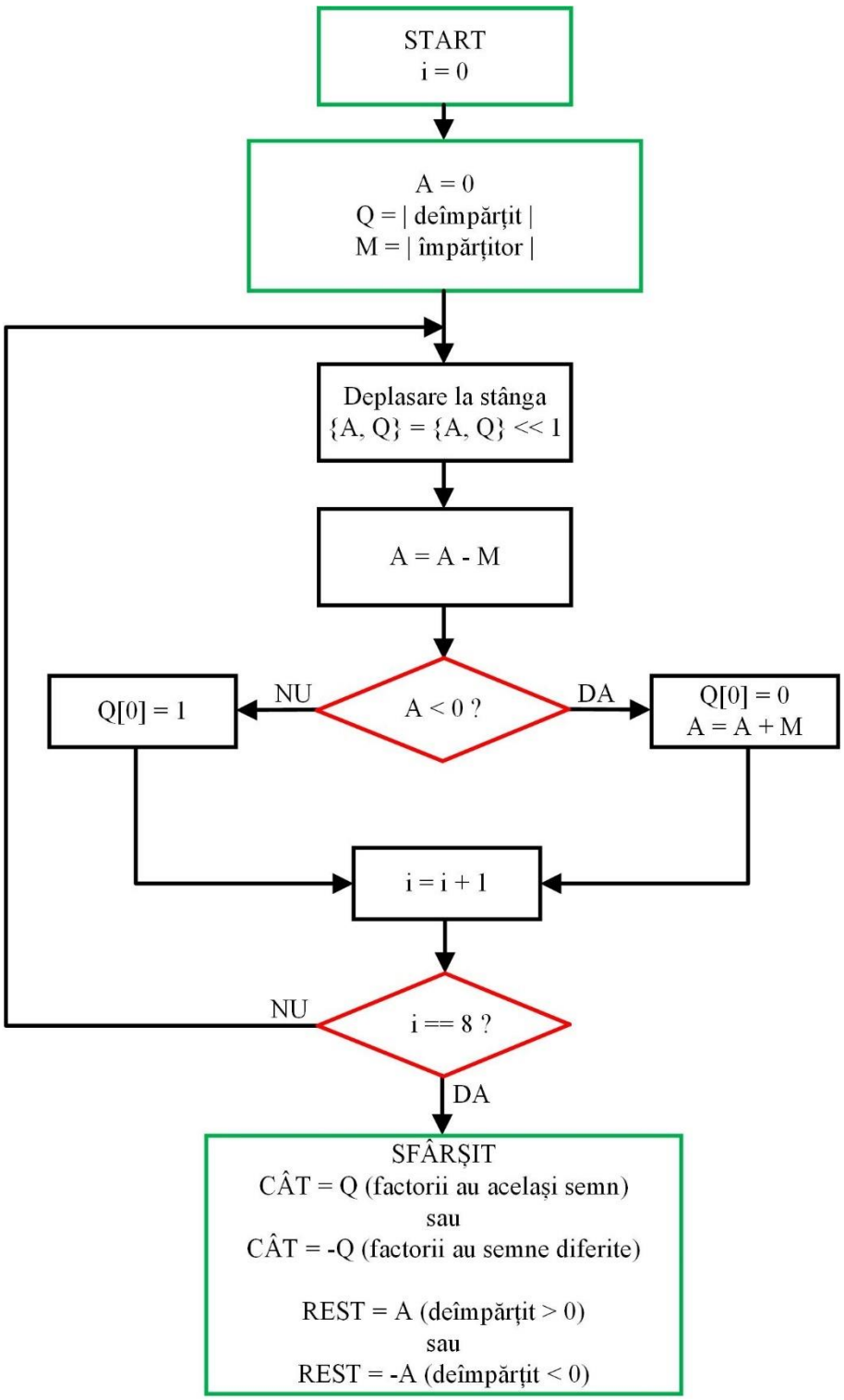


*Figura 1.16* Algoritmul de împărțire binară.

Figura 1.16 prezintă schița unei posibile implementări hardware a algoritmului, cu sumatoarele și circuitele de diferență necesare, multiplexoarele pentru stabilirea bitului curent al câtului și cele pentru decizia de a restaura valoarea lui A sau nu, cât și o parte de sfârșit în care este pusă în evidență ajustarea finală a semnelor câtului și restului în funcție de semnele deîmpărțitului și împărțitorului. Nu au mai fost desenate cele 6 blocuri intermediare din mijloc, întrucât ele sunt identice cu primul și cu ultimul.

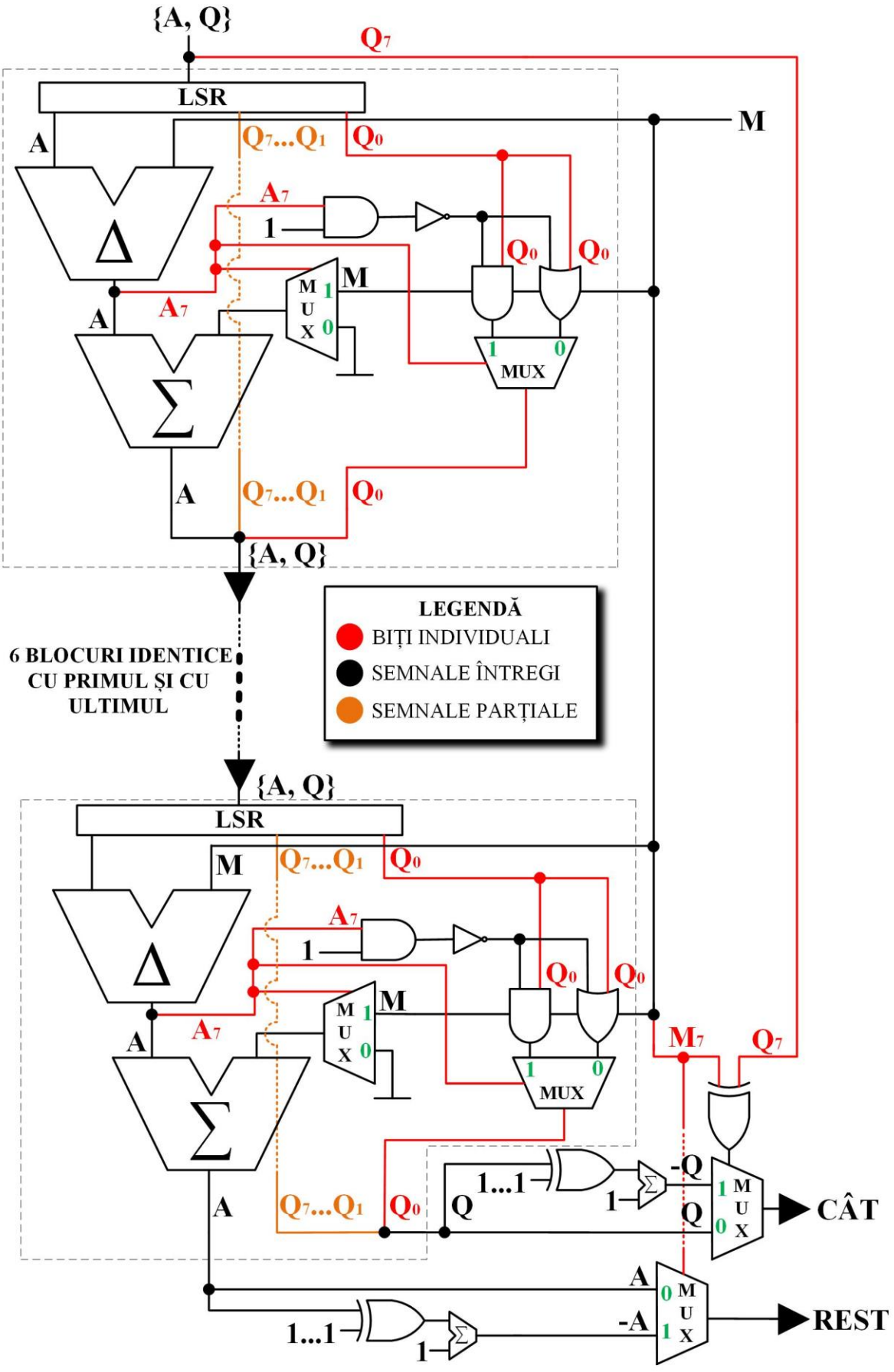


*Figura 1.17* Schemă hardware pentru împărțitorul cablat.

## 1.4 Unitatea de control și sincronizare și registrul de stare

În schema bloc din Figura 1.17 este ilustrată componența modulului cu unitatea de control și sincronizare, registrul de instrucțiune și decodorul, și registrul de stare, pentru care codul sursă este în Anexa 5.

Unitatea de control și sincronizare are rolul de a asigura executarea corectă a tuturor instrucțiunilor prin semnalele de control pe care le trimite spre diferitele blocuri. Ea are acces la magistrala internă de date (pentru adresarea indirectă a instrucțiunilor LD, ST și pentru instrucțiunea LDI), la magistrala internă de adrese pentru date pentru a comanda demultiplexorul și multiplexorul setului de registre și portul memoriei, și la magistrala internă de control pentru menținerea funcționării corecte a microprocesorului. Există o legătură a modulului și cu magistrala internă de date pentru memoria de program, pe care ajung în registrul de instrucțiune cei 4 octeți ai acesteia.

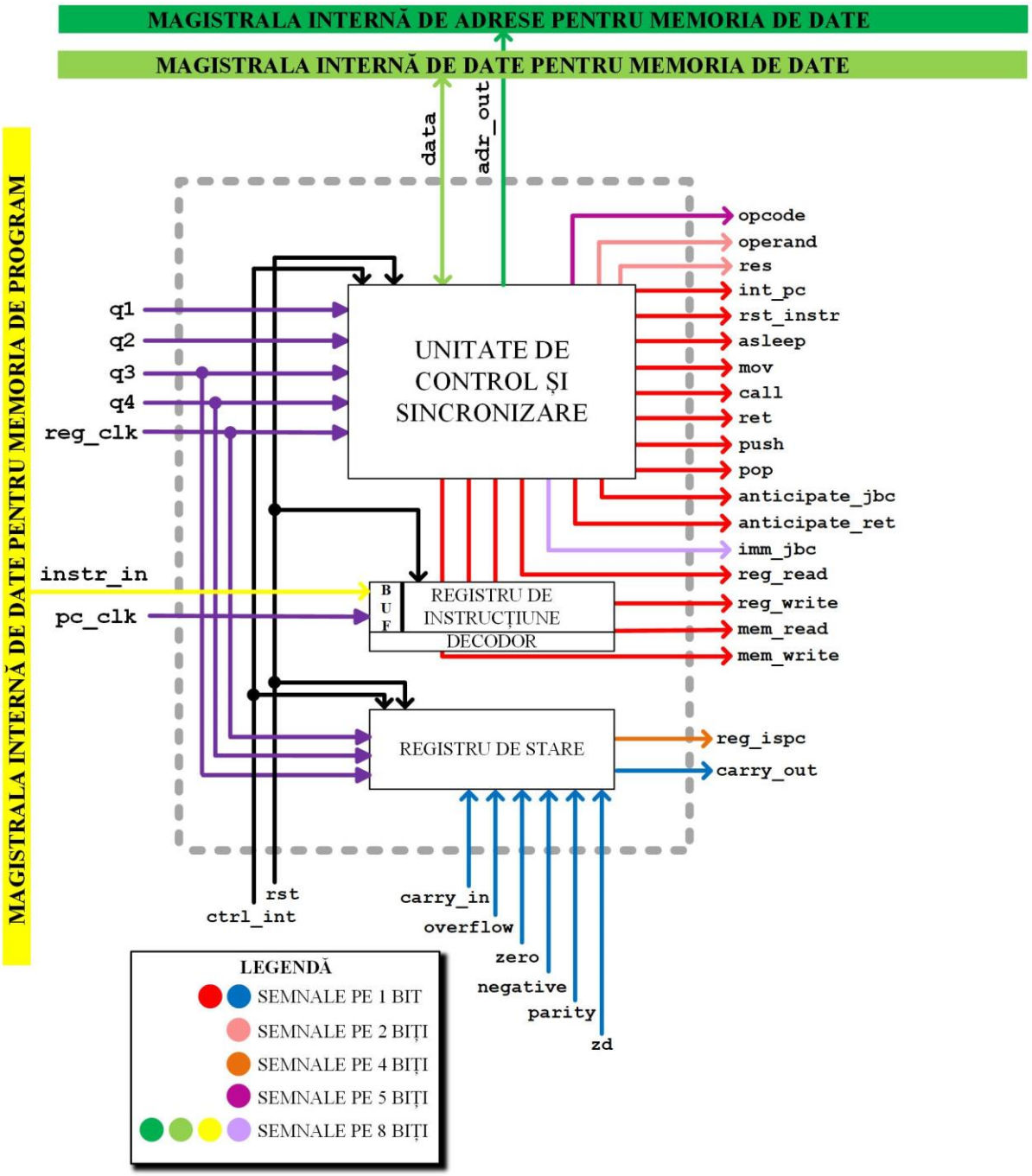


*Figura 1.18* Unitatea de control și sincronizare, registrele de instrucțiune și de stare.

Tabelul 1.5 prezintă toate semnalele de intrare și de ieșire aferente acestui modul, denumite ca în codul sursă, și semnificațiile lor. Semnalele care sunt explicate în tabelele celorlalte module nu sunt explicate și aici, doar este menționat modulul spre care sunt trimise.

| **Semnal** | **Semnificație** |
|---|---|
| *adr_out* | Ieșirea spre magistrala internă de adrese pentru registre și pentru memoria de date |
| *anticipate_jbc* | Semnal trimis spre numărătorul de program din setul de registre la un salt |
| *anticipate_ret* | Semal trimis spre numărătorul de program din setul de registre la RET |
| *asleep* | Semnal trimis spre numărătorul de program din setul de registre la SLEEP |
| *call* | Semnal trimis spre numărătorul de program din setul de registre și stivă la CALL |
| *carry_in* | Fanionul de transport trimis de ALU |
| *carry_out* | Fanionul de transport din registrul de stare trimis spre ALU |
| *ctrl_int* | Întrerupere externă care vine asincron la microprocesor |
| *data* | Port bidirecțional cu magistrala internă de date |
| *imm_jbc* | Semnal trimis spre numărătorul de program din setul de registre la un salt |
| *instr_in* | Intrarea de date de la memoria de program |
| *int_pc* | Semnalul de întrerupere trimis din registrul de stare spre numărătorul de program din setul de registre |
| *mem_read* | Semnal de validare a citirii pentru memoria de date |
| *mem_write* | Semnal de validare a scrierii pentru memoria de date |
| *mov* | Semnal trimis spre setul de registre la MOV |
| *negative* | Fanionul negativ trimis de ALU |
| *opcode* | Codul operației de executat |
| *operand* | Semnal trimis spre ALU și înmulțitor/împărțitor |
| *overflow* | Fanionul de depășire a domeniului pentru numerele cu semn trimis de ALU |
| *parity* | Fanionul de paritate trimis de ALU |
| *pc_clk* | Ceasul de instrucțiune – pentru buffer-ul de intrare al instrucțiunii |
| *pop* | Semnal trimis spre setul de registre și stivă la POP |
| *push* | Semnal trimis spre setul de registre și stivă la PUSH |
| *q1, q2, q3, q4* | Ceasurile de stare 1, 2, 3 și 4 |
| *reg_clk* | Ceasul de registru |
| *reg_ispc* | Semnal trimis de la registrul de stare la setul de registre care conține adresa numărătorului de program curent |
| *reg_read* | Semnal de validare a citirii pentru setul de registre |
| *reg_write* | Semnal de validare a scrierii pentru setul de registre |
| *res* | Semnal trimis spre ALU și înmulțitor/împărțitor |
| *ret* | Semnal triims spre numărătorul de program din setul de registre la RET |
| *rst_instr* | Semnal trimis spre numărătorul de program din setul de registre la RST |
| *zd* | Fanionul de împărțire la zero trimis de  împărțitor |
| *zero* | Fanionul de zero trimis de ALU |

*Tabelul 1.5* Semnalele blocului cu UCS, RI și registrul de stare.

Mai jos este ilustrat modul în care se dau semnalele de anticipare a salturilor (introduse pentru a nu fi necesară instrucțiunea NOP imediat după salt), trimise spre numărătorul de program înainte ca instrucțiunea de salt să fie pusă în registrul de instrucțiune. Registrul *ctrl_instr* este buffer-ul de intrare

pentru registrul de instrucțiune, iar funcția lui va deveni mai clară în capitolul cu desfășurarea în timp a instrucțiunilor. Prefixul *ctrl* al semnalelor se datorează faptului că ele fac parte din modulul cu unitatea de control a microprocesorului.

```
//pentru BREQ, BRNE
always @(negedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) branch_anticipation <= 0;
   else if (check_zero && ((ctrl_instr[31:27] == `BREQ &&
   status_register[2] == 1) || (ctrl_instr[31:27] == `BRNE &&
   status_register[2] == 0)))
      branch_anticipation <= 1;
   else
      branch_anticipation <= 0;
end

//pentru JMP, CALL
always @(posedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) jc_anticipation_value <= 0;
   else if (ctrl_instr[31:27] == `JMP || ctrl_instr[31:27] ==
   `CALL) jc_anticipation_value <= 1;
   else jc_anticipation_value <= 0;
end

//pentru RET
always @ (posedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) ctrl_anticipate_ret <= 0;
   else if (ctrl_instr[31:27] == `RET) ctrl_anticipate_ret <= 1;
   else ctrl_anticipate_ret <= 0;
end

assign ctrl_anticipate_jbc = jc_anticipation_value ||
   branch_anticipation;
```

*Figura 1.19* Cod sursă pentru semnalele de control în instrucțiunile de salt.

Registrul de stare are 16 biți și conține următoarele câmpuri:

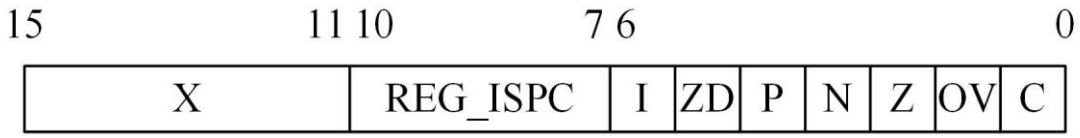


*Figura 1.20* Componența registrului de stare.

Unde:

- C – fanionul de transport (carry);
- OV – fanionul de depășire a domeniului (overflow);
- Z – fanionul de zero;
- N – fanionul de negativ;
- P – fanionul de paritate;
- ZD – fanionul de împărțire la zero;
- I – fanionul de întrerupere;
- REG_ISPC – biții de adresă ai numărătorului de program curent;
- X – biți neimplementați, rezervați pentru posibile dezvoltări ulterioare.

Fanionul de întrerupere se setează asincron la venirea unei întreruperi pe pin, însă el este trimis spre numărătorul de program în mod sincron, pe frontul negativ al ceasului de stare *q1*.

Celelalte fanioane sunt modificate de următoarele instrucțiuni:

| Fanion | Instrucțiuni |
|---|---|
| C | ADC, ADD, DEC, INC, SBC, SUB |
| OV | ADD, DEC, INC, SHL |
| Z | CP, SUB |
| N | Toate instrucțiunile ALU |
| P | Toate instrucțiunile ALU |
| ZD | DIV |
| REG_ISPC | PCIS |

*Tabelul 1.6* Instrucțiunile care modifică fanioanele din registrul de stare.

În Figura 1.20 este ilustrată secțiunea din codul sursă în care cei 4 biți din registrul de stare care țin adresa numărătorului de program curent sunt schimbați de instrucțiunea PCIS (și care au implicit valoarea 15 dacă nu a fost specificat altfel).

```
always @(negedge q3 or posedge ctrl_rst) begin
   if (ctrl_rst) {status_register[10:7], status_register[4:0]} <= 9'b0;
   else if (modify_status) {status_register[10:7],
   status_register[4:0]} <= {status_pc, status_value};
   else {status_register[10:7], status_register[4:0]} <=
   {status_pc, status_register[4:0]};
end

always @ (negedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) pc_change <= 0;
   else if (q3 && ctrl_instr[31:27] == `PCIS) pc_change <= 1;
   else pc_change <= 0;
end

always @ (negedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) status_pc <= 4'd15;
   else if (q4 && pc_change) status_pc <= ctrl_instr[23:20];
   else status_pc <= status_pc;
end

assign ctrl_reg_ispc = status_pc;
```

*Figura 1.21* Cod sursă pentru semnalele de control la instrucțiunea PCIS.

Bitul 5 al registrului de stare este fanionul de împărțire la 0, care se actualizează după încă o perioadă de ceas de registru față de ceilalți biți din registru, întrucât abia atunci este gata rezultatul împărțirii. Bitul 6 este fanionul de întrerupere, care se seteaza asincron la venirea unei întreruperi externe. Valoarea biților care păstrează adresa PC-ului curent se schimbă atunci când instrucțiunea PCIS activează semnalul *pc_change*, și anume atunci când se face fetch la ea. Semnalul *modify_status* este activ atunci când se execută o instrucțiune din cele care afectează primele 5 fanioane din registrul de stare.

## 1.5 Memoriile de program și de date și porturile de intrare/ieșire

Întrucât am stabilit inițial că arhitectura implementată va fi de tip Harvard, memoriile de program și de date sunt separate, cu magistrale pentru date și adrese separate, toate de 8 biți. Ambele au aceeași dimensiune (256 de octeți), întrucât ambele sunt adresabile cu valori aflate în registrele generale interne ale microprocesorului, pe 8 biți.

În timp ce memoria de program este de tip ROM – poate fi doar citită, nu și scrisă, și este nevolatilă (își menține informația la decuplarea de la alimentare), memoria de date este de tip RAM – putând fi și citită, și scrisă, la orice adresă, atunci când avem semnalele de validare respective în 1 [7].

În Figura 1.21 este ilustrată schema bloc a memoriei de program, al cărei cod sursă este în Anexa 6.

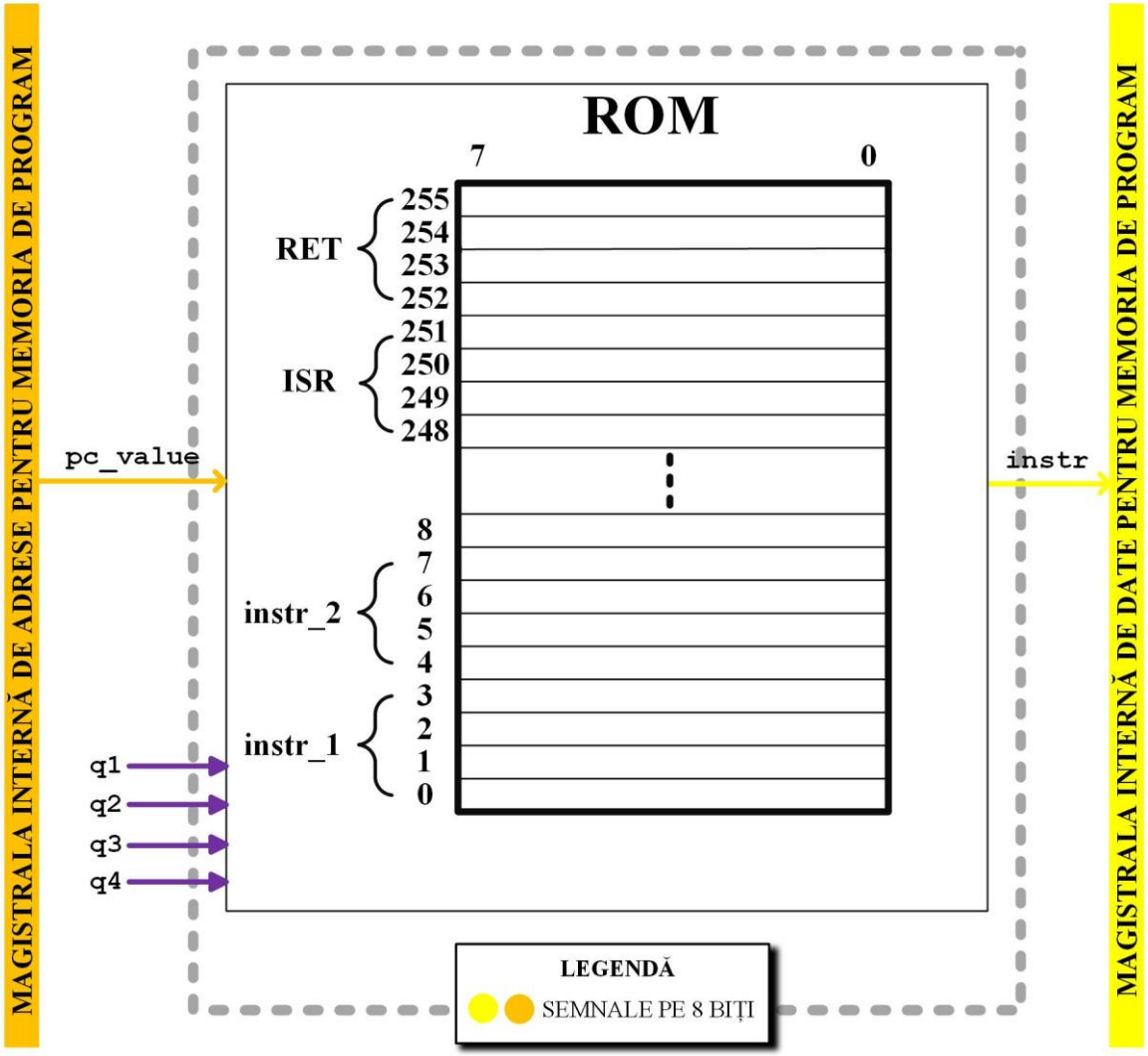


*Figura 1.22* Memoria de program.

Memoria de program trimite câte un octet pentru o instrucțiune pe fiecare ceas de stare (*q1*, *q2*, *q3*, *q4*), întrucât magistrala internă de date aferentă este de 8 biți, iar formatul instrucțiunii are 32 de biți. Singurele poziții pe care le-am ales a fi dedicate sunt ultimele două, de la adresele 248 și 252, reprezentând, în mod simbolic, adresa rutinei de deservire a întreruperii la care sare numărătorul de program atunci când în registrul de stare se seteaza fanionul corespunzător întreruperii externe. În codul sursă, cele 2 instrucțiuni nu reprezintă decât o încărcare a valorii imediate 1 în registrul $R_0$ și o întoarcere la programul curent.

Mai jos este un tabel cu semnalele de intrare și de ieșire aferente memoriei ROM, denumite ca în codul sursă și explicate.

| Semnal | Semnificație |
|---|---|
| *instr* | Ieșirea spre magistrala de date pentru memoria de program, pe care se trimite instrucțiunea microprocesorului |
| *pc_value* | Valoarea numărătorului de program primită de la microprocesor |
| *q1, q2, q3, q4* | Ceasurile de stare 1, 2, 3 și 4 |

*Tabelul 1.7* Semnalele blocului ROM.

În continuare este prezentată o parte din codul prin care am populat memoria ROM cu câteva instrucțiuni, trimise spre registrul de instrucțiune în 4 pași (*q1*, *q2*, *q3* și *q4*).

```
wire [`INSTRUCTION_SIZE-1:0] instr_1;
wire [`INSTRUCTION_SIZE-1:0] instr_2;
wire [`INSTRUCTION_SIZE-1:0] instr_3;
wire [`INSTRUCTION_SIZE-1:0] instr_4;

assign instr_1 = {`NOP, 27'bx}; //NOP
assign instr_2 = {`LDI, 3'bx, `R0, 12'bx, 8'd48}; //LDI R0, #48
assign instr_3 = {`LDI, 3'bx, `R1, 12'bx, 8'd40}; //LDI R1, #40
assign instr_4 = {`SUB, 3'bx, `R2, 4'bx, `R0, `R1, 8'bx}; //SUB R2, R0, R1
assign instr_default = {`SLEEP, 27'bx}; //SLEEP

always @(*) begin
   case (pm_pc_value)
      8'd0: pm_instr = q1 ? instr_1[31:24] : q2 ? instr_1[23:16]
                     : q3 ? instr_1[15:8] : instr_1[7:0];
      8'd4: pm_instr = q1 ? instr_2[31:24] : q2 ? instr_2[23:16]
                     : q3 ? instr_2[15:8] : instr_2[7:0];
      8'd8: pm_instr = q1 ? instr_3[31:24] : q2 ? instr_3[23:16]
                     : q3 ? instr_3[15:8] : instr_3[7:0];
      8'd12: pm_instr = q1 ? instr_4[31:24] : q2 ? instr_4[23:16]
                      : q3 ? instr_4[15:8] : instr_4[7:0];

      default: pm_instr = q1 ? instr_default[31:24] : q2 ?
                              instr_default[23:16] : q3 ?
                              instr_default[15:8] :
                              instr_default[7:0];
   endcase
end
```

*Figura 1.23* Cod sursă pentru memoria de program.

Instrucțiunile sunt atribuite în mod continuu, sub o formă care respectă formatul instrucțiunii și pe care registrul de instrucțiune știe să o decodeze. Biții care au valoarea x sunt câmpuri care nu au nicio semnificație pentru instrucțiunea în cauză, ca atare ei pot fi orice. Pentru a distinge între octeții care trebuie trimiși spre microprocesor la un moment dat de timp, se verifică starea care este activă (dintre *q1*, *q2*, *q3* sau *q4*). Instrucțiunea implicită, pe care începe să o execute mașina atunci când numărătorul de program nu mai arată spre o locație ROM implementată, este SLEEP, astfel încât microprocesorul să înceteze execuția.

În Figura 1.23 ilustrată memoria de date, în care sunt marcate zona disponibilă microprocesorului (locațiile 16 - 255) și zonele dedicate dispozitivelor de intrare și ieșire care sunt mapate în memorie (locațiile 0 - 15). Se observă în figură că memoria și porturile interoghează aceleași magistrale de date, adrese și control, astfel că, din punct de vedere al microprocesorului, memoria propriu-zisă și porturile sunt văzute și tratate în mod identic, rămânând în sarcina celor 2 să distingă care sunt semnalele adresate fiecărei părți. Scrierea în aceste două entități se face sincron, pe ceasul de registru, însă citirea este asincronă, fiind activă atunci când semnalul de validare al citirii este setat de UCS la instrucțiunea LD.

În timp ce includerea hărții porturilor în harta memoriei vine cu avantajul de a le accesa cu aceleași instrucțiuni folosite pentru accesarea memoriei (LD și ST), nefiind necesare instrucțiuni suplimentare dedicate, un dezavantaj ar fi faptul că se reduce numărul de locații RAM disponibile propriu-zis microprocesorului.

Codul sursă pentru memoria de date și pentru cele 4 dispozitive de intrare/ieșire se află în Anexa 7.

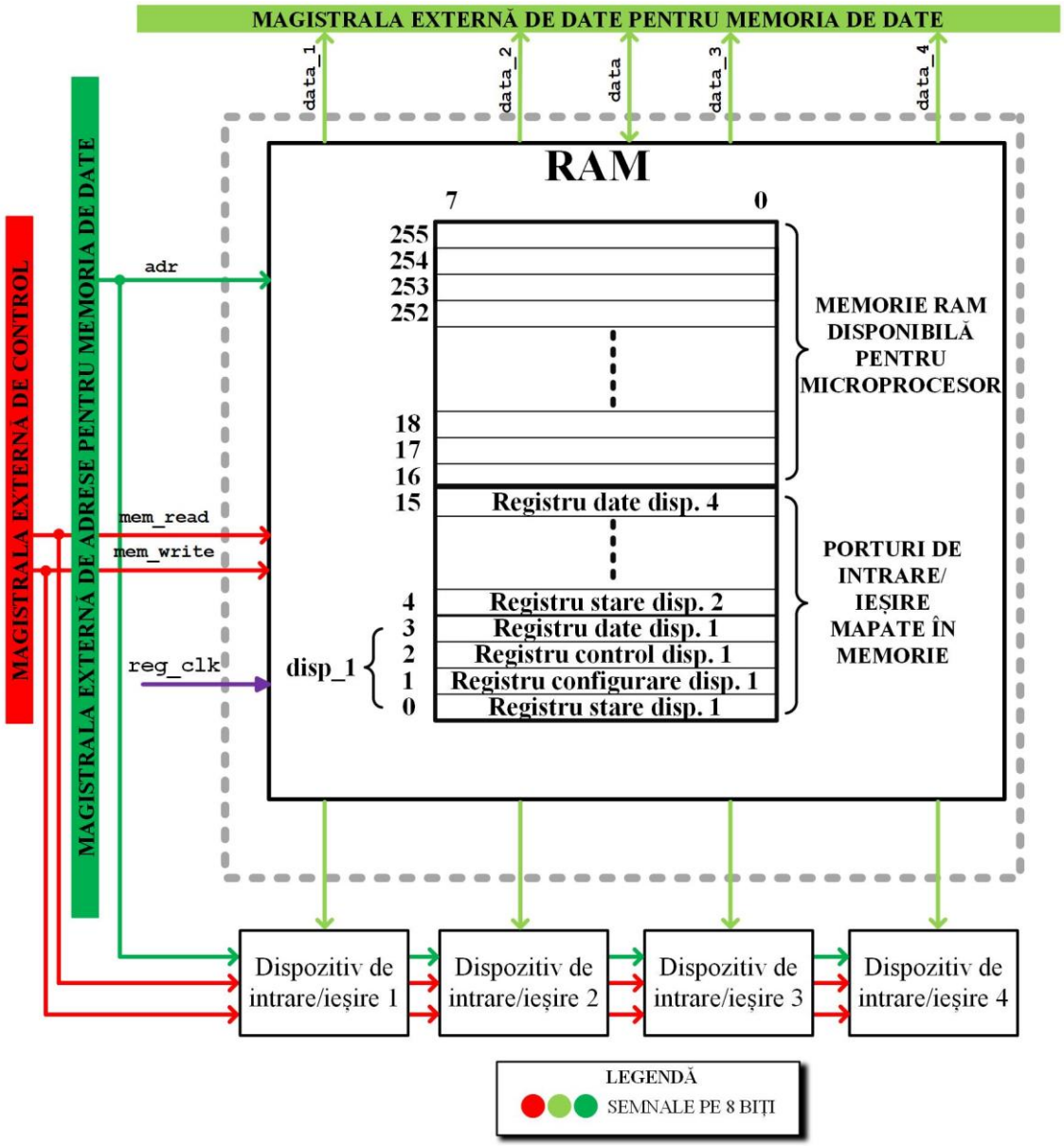


*Figura 1.24* Memoria de date și dispozitivele de intrare/ieșire.

Tabelul 1.8 conține semnalele de intrare și de ieșire din memoria RAM (aceleași și pentru porturi, așadar), denumite ca în codul sursă, și explicate.

| Semnal | Semnificație |
|---|---|
| *adr* | Adresa la care se scrie sau de la care se citește, trimisă de UCS |
| *data* | Portul bidirecțional de date care realizează legătura cu magistrala externă de date a memoriei propriu-zise |
| *data_i* | Portul bidirecțional de date care realizează legătura cu magistrala externă de date a dispozitivului de intrare/ieșire i, cu i de la 1 la 4 |
| *mem_read* | Semnalul de validare a citirii trimis de UCS |
| *mem_write* | Semnalul de validare a scrierii trimis de UCS |
| *reg_clk* | Ceasul de registru (scrierea în memorie este sincronă, iar citirea asincronă) |

*Tabelul 1.8* Semnalele blocului RAM.

În continuare sunt ilustrate scrierea sincronă și citirea asincronă din memorie. Prefixul *dm* se datorează faptului că semnalele fac parte din modulul cu memoria de date.

```
reg signed [`OPERAND_SIZE-1:0] data_memory [16:255];

//scriere sincronă
always @ (posedge dm_reg_clk) begin
   if (dm_write && dm_adr > 15)
   data_memory[dm_adr] <= dm_data;
end

//citire asincronă
assign dm_data = (dm_read && dm_adr > 15) ? data_memory[dm_adr] : 8'bz;
```

*Figura 1.25* Cod sursă pentru memoria de date.

Absolut similar se întâmplă și pentru cele 4 porturi de intrare/ieșire, cu diferența că adresele lor sunt diferite.

Am presupus că cele 4 dispozitive de intrare/ieșire sunt identice, cu 4 registre fiecare:

- Un registru de stare care se poate doar citi (accesibil doar prin instrucțiunea LD);
- Un registru de configurare care se poate doar scrie (accesibil doar prin instrucțiunea ST);
- Un registru de control care poate fi și citit, și scris;
- Un registru de date care poate fi și citit, și scris.

Un astfel de dispozitiv ar putea fi un numărător, care dă următoarea semnificație registrelor de mai sus:

- Primul bit al registrului de stare este 1 dacă valoarea numărătorului este sub jumătate din valoarea maximă (de resetare);
- Primii 4 biți din registrul de configurare reprezintă pasul cu care numărătorul trebuie să numere;
- Primul bit din registrul de control activează sau dezactivează numărătorul;
- În registrul de date se aduce valoarea maximă până la care să se facă numărarea, urmând resetarea valorii din numărător.

În Figura 1.25 este o parte din codul sursă care implementează o astfel de funcționare.

```
wire enable;
wire [3:0] step;
wire [7:0] timer_max_value;
reg change;

assign enable = io_control_reg[0];
assign step = io_config_reg[3:0];
assign timer_max_value = io_data_reg;

always @ (negedge io_reg_clk or posedge io_rst) begin
   if (io_rst) change <= 0;
   else change <= (io_output == timer_max_value || (io_output + step) >
   timer_max_value);
end
always @ (posedge io_reg_clk or posedge io_rst) begin
   if (io_rst || change) io_output <= 0;
   else if (enable) io_output <= io_output + step;
   else io_output <= io_output;
end
assign io_status_reg[0] = (io_output < timer_max_value/2) ? 1:0;
```

*Figura 1.26* Cod sursă pentru un dispozitiv de intrare/ieșire arbitrar (numărător).

Luând un exemplu concret în care valoarea maximă de numărare este 50 și pasul 4, forma de undă a ieșirii va arăta ca în Figura 1.26, în care se observă schimbarea primului bit din registrul de stare din 1 în 0 când valoarea ieșirii depășește jumătatea valorii maxime, și resetarea numărătorului la atingerea (sau, mai degraba, înainte de atingerea) acesteia.

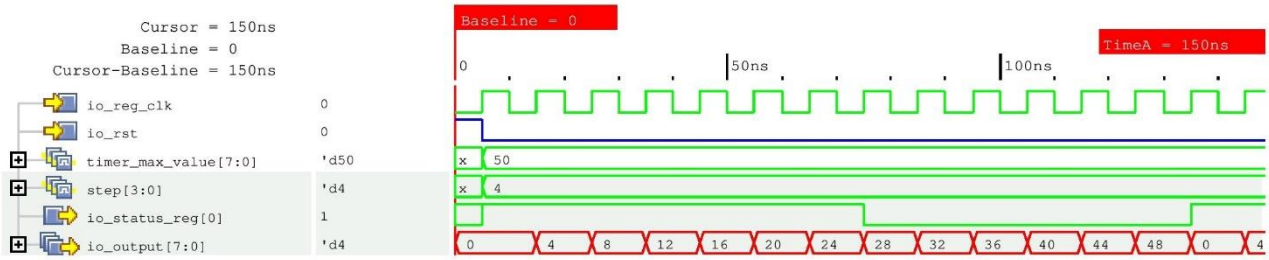


*Figura 1.27* Forme de undă pentru un numărător arbitrar.

# Capitolul 2 - Setul de instrucțiuni

## 2.1 Categorii de instrucțiuni

Microprocesorul poate executa un număr total de trezeci de instrucțiuni, pe care le-am ales să fie conforme cu un model de arhitectură de tip RISC [8]. Ele sunt de trei tipuri:

- De transfer de date (între registre, între registre și stivă sau între registre și memoria de date);
- De prelucrări de date (exclusiv cu registrele interne generale);
- De control al programului (salturi condiționate și necondiționate, apel de subprograme, resetare, adormire, nicio operație sau instrucțiunea de modificare a numărătorului de program).

Formatul instrucțiunii conține 7 câmpuri diferite, după cum urmează:

- Codul operației (cei mai semnificativi 5 biți);
- Modul de adresare (următorii 3 biți);
- Patru câmpuri pentru adresele registrelor implicate;
- Valoarea imediată (pentru adresarea relativă imediată la salturi și pentru instrucțiunea LDI).

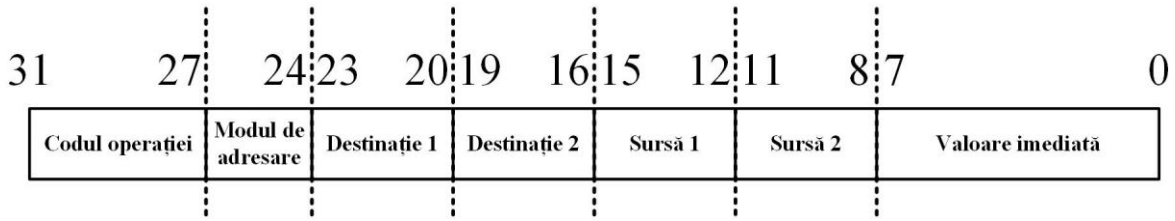


*Figura 2.1* Formatul instrucțiunii.

Formatul și dimensiunea instrucțiunii sunt fixe pentru întregul set de instrucțiuni, aspect specific RISC, care ajută la o decodare într-o singură stare [9].

În Tabelele 2.1, 2.2 și 2.3 de mai jos sunt trecute:

- Mnemonicele instrucțiunilor (cu forma echivalentă a instrucțiunii în limbaj de asamblare);
- Codul operației;
- Conținutul formatului instrucțiunii;
- Pseudocodul instrucțiunii;
- O descriere literală;
- Modurile de adresare folosite.

De precizat următoarele 2 semnificații de notații:

- ($R_X$) reprezintă valoarea din registrul $R_X$;
- (($R_X$)) reprezintă valoarea din memoria de date (sau din stivă, daca $R_X$ este SP), aflată la adresa specificată în $R_X$.

Modurile de adresare folosite sunt:

- Adresare implicită în registru (specificarea unui registru/mai multor registre într-unul din câmpurile codului instrucțiunii) [1];
- Adresare indirectă prin registru (într-unul din câmpurile instrucțiunii este specificat registrul în care se află adresa din memorie la care sunt datele necesare) [1];

- Adresare relativă imediată pentru salturi (în formatul instrucțiunii este specificată valoarea imediată care trebuie adunată la PC) [1].

Instrucțiuni de transfer de date:

| **Instrucțiune** | **Cod operație (zecimal)** | **Formatul instrucțiunii și pseudocodul** | **Descriere** | **Mod de adresare** |
|---|---|---|---|---|
| LD $R_A$, $R_B$ | 0 | [5'd0, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'bx, 8'bx]<br><br>($R_A$) ← (($R_B$)) | În $R_A$ este adusă valoarea aflată în memoria de date la adresa cuprinsă în $R_B$. | Indirectă în registru ($R_B$) și implicită în registru ($R_A$). |
| LDI $R_A$, #imm | 1 | [5'd1, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'b imm]<br><br>($R_A$) ← imm | În $R_A$ este pusă valoarea codată imediat imm. | Implicită în registru (pentru destinația $R_A$) și imediată. |
| MOV $R_A$, $R_B$ | 2 | [5'd2, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'bx, 8'bx]<br><br>($R_A$) ← ($R_B$) | În $R_A$ se scrie valoarea din $R_B$. | Implicită în registru. |
| POP $R_A$ | 3 | [5'd3, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>($R_A$) ← ((SP))<br>(SP) ← (SP) - 1 | În registrul $R_A$ se pune valoarea aflată în vârful stivei (SP), după care SP e decrementat. | Implicită în registru. |
| PUSH $R_A$ | 4 | [5'd4, 3'bx, 4'bx, 4'bx, 4'b $R_A$, 4'bx, 8'bx]<br><br>(SP) ← (SP) + 1<br>((SP)) ← ($R_A$) | În stiva hardware din interiorul microprocesorului, după incrementarea lui SP, se pune valoarea din $R_A$. | Implicită în registru. |
| ST $R_A$, $R_B$ | 5 | [5'd5, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'bx, 8'bx]<br><br>(($R_A$)) ← ($R_B$) | În memoria de date, la adresa aflată în $R_A$ se pune valoarea din $R_B$. | Implicită în registru ($R_B$) și indirectă în registru ($R_A$). |

*Tabelul 2.1* Instrucțiuni de transfer de date.

Instrucțiuni de prelucrări de date:

| **Instrucțiune** | **Cod operație (zecimal)** | **Formatul instrucțiunii și pseudocodul** | **Descriere** | **Mod de adresare** |
|---|---|---|---|---|
| ADC $R_A$, $R_B$, $R_C$ | 6 | [5'd6, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_c$, 8'bx]<br><br>($R_A$) ← ($R_B$) + ($R_C$) + (C) | În $R_A$ este pusă suma valorilor din registrele $R_B$ si $R_C$, la care se adaugă și fan-ionul de | Implicită în registru. |
| ADD $R_A$, $R_B$, $R_C$ | 7 | [5'd7, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_c$, 8'bx]<br><br>($R_A$) ← ($R_B$) + ($R_C$) | În $R_A$ este pusă suma valorilor din registrele $R_B$ și $R_C$. | Implicită în registru. |
| AND $R_A$, $R_B$, $R_C$ | 8 | [5'd8, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_C$, 8'bx]<br><br>($R_A$) ← ($R_B$) & ($R_C$) | Se face ȘI (bit cu bit) între valorile din registrele $R_B$ și $R_C$, iar re-zultatul se scrie în $R_A$. | Implicită în registru. |
| ASR $R_A$ | 9 | [5'd9, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>($R_A$) ← ($R_A$) >>> 1 | Se deplasează ar-itmetic la dreapta valoarea din $R_A$. | Implicită în registru. |
| CP $R_A$, $R_B$ | 10 | [5'd10, 3'bx, 4'bx, 4'bx, 4'b $R_A$, 4'b $R_B$, 8'bx]<br><br>($R_A$) - ($R_B$)<br>if ($R_A$) = ($R_B$) then (Z) ← 1<br>else (Z) ← 0 | Se face scăderea între valorile din registrele $R_A$ și $R_B$, rezultatul nu este scris nicăieri, însă fanionul zero se setează daca cele două sunt egale. | Implicită în registru. |
| DEC $R_A$ | 11 | [5'd11, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>($R_A$) ← ($R_A$) - 1 | Se decrementează valoarea din $R_A$. | Implicită în registru. |
| DIV $R_A$, $R_B$, $R_C$, $R_D$ | 12 | [5'd12, 3'bx, 4'b $R_A$, 4'b $R_B$, 4'b $R_c$, 4'b $R_D$, 8'bx]<br><br>($R_A$) ← ($R_C$) DIV ($R_D$)<br>($R_B$) ← ($R_C$) MOD ($R_D$) | Se face împărțirea dintre valorile din registrele $R_C$ și $R_D$, iar câtul se pune în $R_A$, restul în $R_B$. | Implicită în registru. |

| | | | | |
|---|---|---|---|---|
| INC $R_A$ | 13 | [5'd13, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>$(R_A) \leftarrow (R_A) + 1$ | Se incrementează valoarea din $R_A$. | Implicită în registru. |
| MUL $R_A$, $R_B$, $R_C$, $R_D$ | 14 | [5'd14, 3'bx, 4'b $R_A$, 4'b $R_B$, 4'b $R_c$, 4'b $R_D$, 8'bx]<br><br>$(R_A, R_B) \leftarrow (R_C) * (R_D)$ | Se face înmulțirea între valorile din registrele $R_C$ și $R_D$, iar rezultatul pe 16 biți se pune în $R_A$ (partea superioară) și $R_B$ (partea inferioară). | Implicită în registru. |
| OR $R_A$, $R_B$, $R_C$ | 15 | [5'd15, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_C$, 8'bx]<br><br>$(R_A) \leftarrow (R_B) \mid (R_C)$ | Se face SAU (bit cu bit între valorile din registrele $R_B$ și $R_C$, iar rezultatul se scrie în $R_A$. | Implicită în registru. |
| SBC $R_A$, $R_B$, $R_C$ | 16 | [5'd16, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_c$, 8'bx]<br><br>$(R_A) \leftarrow (R_B) - (R_C) - (C)$ | În $R_A$ este pusă diferența valorilor din registrele $R_B$ și $R_C$, din care se scade și fanionul de transport. | Implicită în registru. |
| SHL $R_A$ | 17 | [5'd17, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>$(R_A) \leftarrow (R_A) << 1$ | Se deplasează logic la stânga valoarea din $R_A$. | Implicită în registru. |
| SHR $R_A$ | 18 | [5'd18, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>$(R_A) \leftarrow (R_A) >> 1$ | Se deplasează logic la dreapta valoarea din $R_A$. | Implicită în registru. |
| SUB $R_A$, $R_B$, $R_C$ | 19 | [5'd19, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_c$, 8'bx]<br><br>$(R_A) \leftarrow (R_B) - (R_C)$ | În $R_A$ este pusă diferența valorilor din registrele $R_B$ și $R_C$. | Implicită în registru. |
| XOR $R_A$, $R_B$, $R_C$ | 20 | [5'd20, 3'bx, 4'b $R_A$, 4'bx, 4'b $R_B$, 4'b $R_C$, 8'bx]<br><br>$(R_A) \leftarrow (R_B) \wedge (R_C)$ | Se face SAU EXCLUSIV (bit cu bit) între valorile din registrele $R_B$ și $R_C$, iar rezultatul se scrie în $R_A$. | Implicită în registru. |

*Tabelul 2.2* Instrucțiuni de prelucrări de date.

Instrucțiuni de control al programului:

| **Instrucțiune** | **Cod operație (zecimal)** | **Formatul instrucțiunii și pseudocodul** | **Descriere** | **Mod de adresare** |
|---|---|---|---|---|
| BREQ #imm | 21 | [5'd21, 3'b0, 4'bx, 4'bx, 4'bx, 4'bx, 8'b imm]<br><br>dacă (Z) = 1, atunci (PC) ← (PC) + imm<br>altfel (PC) ← (PC) + 4 | Se face un salt condiționat (dacă valoarea fanionului zero este 1), relativ la PC (la care se adaugă valoarea imm). De regulă, se folosește după instrucțiunile SUB sau CP. | Relativă imediată. |
| BREQ $R_A$ | | [5'd21, 3'b1, 4'bx, 4'bx, 4'b $R_A$, 4'bx, 8'bx]<br><br>dacă (Z) = 1, atunci (PC) ← ($R_A$)<br>altfel (PC) ← (PC) + 4 | Se face un salt condiționat (dacă valoarea fanionului zero este 1), indirect (noua valoare a PC este în $R_A$). De regulă, se folosește după instrucțiunile SUB sau CP. | Implicită în registru. |
| BRNE #imm | 22 | [5'd22, 3'b0, 4'bx, 4'bx, 4'bx, 4'bx, 8'b imm]<br><br>dacă (Z) = 0, atunci (PC) ← (PC) + imm<br>altfel (PC) ← (PC) + 4 | Se face un salt condiționat (dacă valoarea fanionului zero este 0), relativ la PC (la care se adaugă valoarea imm). De regulă, se folosește dupa instrucțiunile SUB sau CP. | Relativă imediată. |
| BRNE $R_A$ | | [5'd22, 3'b1, 4'bx, 4'bx, 4'b $R_A$, 4'bx, 8'bx]<br><br>dacă (Z) = 0, atunci (PC) ← ($R_A$)<br>altfel (PC) ← (PC) + 4 | Se face un salt condiționat (dacă valoarea fanionului zero este 0), indirect (noua valoare a PC este în $R_A$). De regulă, se folosește după instrucțiunile SUB sau CP. | Implicită în registru. |

| | | | | |
|---|---|---|---|---|
| CALL #imm | 23 | [5'd23, 3'b0, 4'bx, 4'bx, 4'bx, 4'bx, 8'b imm]<br><br>(SP) ← (SP) + 1<br>((SP)) ← (PC) + 4<br>(PC) ← (PC) + imm | Se face apel la o subrutină aflată relativ de PC la imm locații, stocându-se în stiva hardware adresa instrucțiunii următoare dinaintea apelului. | Relativă imediată. |
| CALL $R_A$ | | [5'd23, 3'b1, 4'bx, 4'bx, 4'b $R_A$, 4'bx, 8'bx]<br><br>(SP) ← (SP) + 1<br>((SP)) ← (PC) + 4<br>(PC) ← ($R_A$) | Se face apel la o subrutină a cărei adresă este în $R_A$, stocându-se în stiva hardware adresa instrucțiunii următoare dinaintea apelului. | Implicită în registru. |
| JMP #imm | 24 | [5'd24, 3'b0, 4'bx, 4'bx, 4'bx, 4'bx, 8'b imm]<br><br>(PC) ← (PC) + imm | Se face un salt necondiționat, relativ la PC (la care se adaugă valoarea imm). | Relativă imediată. |
| JMP $R_A$ | | [5'd24, 3'b1, 4'bx, 4'bx, 4'b $R_A$, 4'bx, 8'bx]<br><br>(PC) ← ($R_A$) | Se face un salt necondiționat, adresa de salt fiind în registrul $R_A$. | Implicită în registru. |
| NOP | 25 | [5'd25, 3'bx, 4'bx, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>(PC) ← (PC) | Nu se efectuează nicio operație. | - |
| PCIS $R_A$ | 26 | [5'd26, 3'bx, 4'b $R_A$, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>(PC) ← ($R_A$), $R_A$ devine PC | $R_A$ devine noul PC (altfel $R_{15}$ este implicit PC). | Implicită în registru. |
| RET | 27 | [5'd27, 3'bx, 4'bx, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>(PC) ← ((SP))<br>(SP) ← (SP) - 1 | Se revine din subrutina apelată prin CALL, la instrucțiunea imediat următoare. | - |
| RST | 28 | [5'd28, 3'bx, 4'bx, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>(PC) ← 0 | Se activează semnalul de reset, implicit resetându-se valoarea PC-ului. | - |
| SLEEP | 29 | [5'd29, 3'bx, 4'bx, 4'bx, 4'bx, 4'bx, 8'bx]<br><br>(PC) ← (PC) | PC oprit pe loc, ieșiri (ale magistralei de date și adrese pentru memoria de date) în HiZ. | - |

*Tabelul 2.3* Instrucțiuni de control al programului.

## 2.2 Desfășurarea în timp a instrucțiunilor

Pentru a realiza sincronizarile corecte între toate blocurile microprocesorului, am implementat 6 ceasuri diferite, după cum apar în Figura 2.2 de mai jos. Codul sursă pentru ele se află în Anexa 8. Primul este ceasul de registru (*reg_clk*), cel pe care se face scrierea în registre și, de asemenea, cel pe care se verifică starea în care se află mașina. Prin stare, înțelegem fiecare perioadă consecutivă de ceas de registru, echivalentă cu pulsul unuia dintre cele 4 ceasuri de stare *q1*, *q2*, *q3* și *q4*, care se succed. Astfel, o perioadă a ceasului de instrucțiune/ceasului numărătorului programului (*pc_clk*) conține 4 perioade de ceas de registru, sau, altfel spus, succesiunea de pulsuri *q1*-*q4*. Semnalele din figură au fost denumite ca în codul sursă, și, în plus, avem semnificațiile:

- FETCH – etapa în care se aduce instrucțiunea în interiorul microprocesorului (în BUF)
- DECODE – etapa de decodare a instrucțiunii de către UCS
- EXECUTE – execuția propriu-zisă a operației cerute

Aducerea instrucțiunii din memoria de program în interiorul microprocesorului durează patru astfel de stări, întrucât sunt necesari patru pași pentru a aduce cuvântul de 32 de biți pe o magistrala de 8 biți, încărcarea buffer-ului (BUF) de intrare făcându-se pe frontul negativ al ceasului de registru, astfel că la primul ceas de instrucțiune următor, toți cei 4 octeți ai acesteia au fost aduși deja în buffer. Urmează apoi o stare de decodare a instrucțiunii, și anume *q1*, în care se interpretează câmpurile instrucțiunii, în special codul operației. În urma acestei etape, microprocesorul știe ce pași trebuie executați, iar această execuție durează încă 4 stări – *q2*, *q3*, *q4*, *q1*. Acest număr de 4 stări a fost ales luând în considerare cele 2 instrucțiuni care necesită 4 pași distincți – MUL și DIV, întrucât în formatul acestor instrucțiuni sunt specificate patru registre diferite care trebuie citite sau scrise.

Se poate vedea în figură că, deși dacă luăm o instrucțiune anume, de exemplu $I_2$ (cu albastru), ea pare să dureze aproximativ 9 stări (9 perioade de ceas de registru), în realitate, la fiecare front negativ de *q1*, microprocesorul are rezultatul instrucțiunii ce tocmai a părăsit registrul de instrucțiune, astfel că putem vorbi despre o tehnică pipeline în implementare. Prin tehnică pipeline, înțelegem divizarea unei instrucțiuni într-un număr de etape diferite, care necesită resurse hardware diferite și care, deci, pot avea loc în același timp pentru instrucțiuni diferite [6, 10]. Așadar, timpul de execuție efectiv este de patru stări (patru perioade de ceas de registru), întrucât acesta este timpul necesar microprocesorului pentru a livra câte un rezultat.

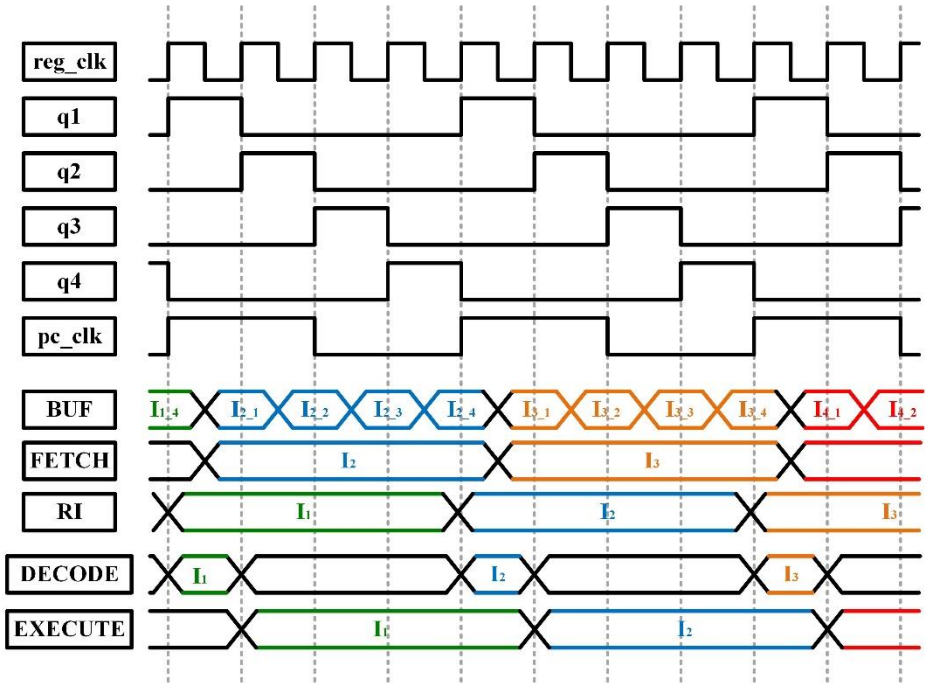


*Figura 2.2* Desfășurarea în timp a instrucțiunilor.

Important de menționat este faptul că semnalele de control se dau pe frontul pozitiv al ceasului de registru *reg_clk*, moment la care se verifică dacă *q1* este activ, sau *q2*, sau *q3* sau *q4*, ceea ce va face ca activarea acestor semnale sa aibă loc abia pe frontul negativ al ceasului de stare respectiv, întrucât frontul pozitiv nu este văzut în același timp cu frontul pozitiv la ceasului de registrul, ele reprezentând, de fapt, același moment de timp, deci timpul de set-up necesar unui ceas de stare nu se respectă.

În Figura 2.3, este exemplificată instrucțiunea de înmulțire, cu semnalele de control menționate în Tabelul 1.3, astfel încât să fie mai clar modul de trimitere al acestora spre diferitele blocuri din microprocesor, aici, în particular, spre înmulțitorul cablat. Înmulțirea dintre valorile din $R_C$ și $R_D$ trebuie pusă în $R_A$, partea superioară, și în $R_B$, partea inferioară.

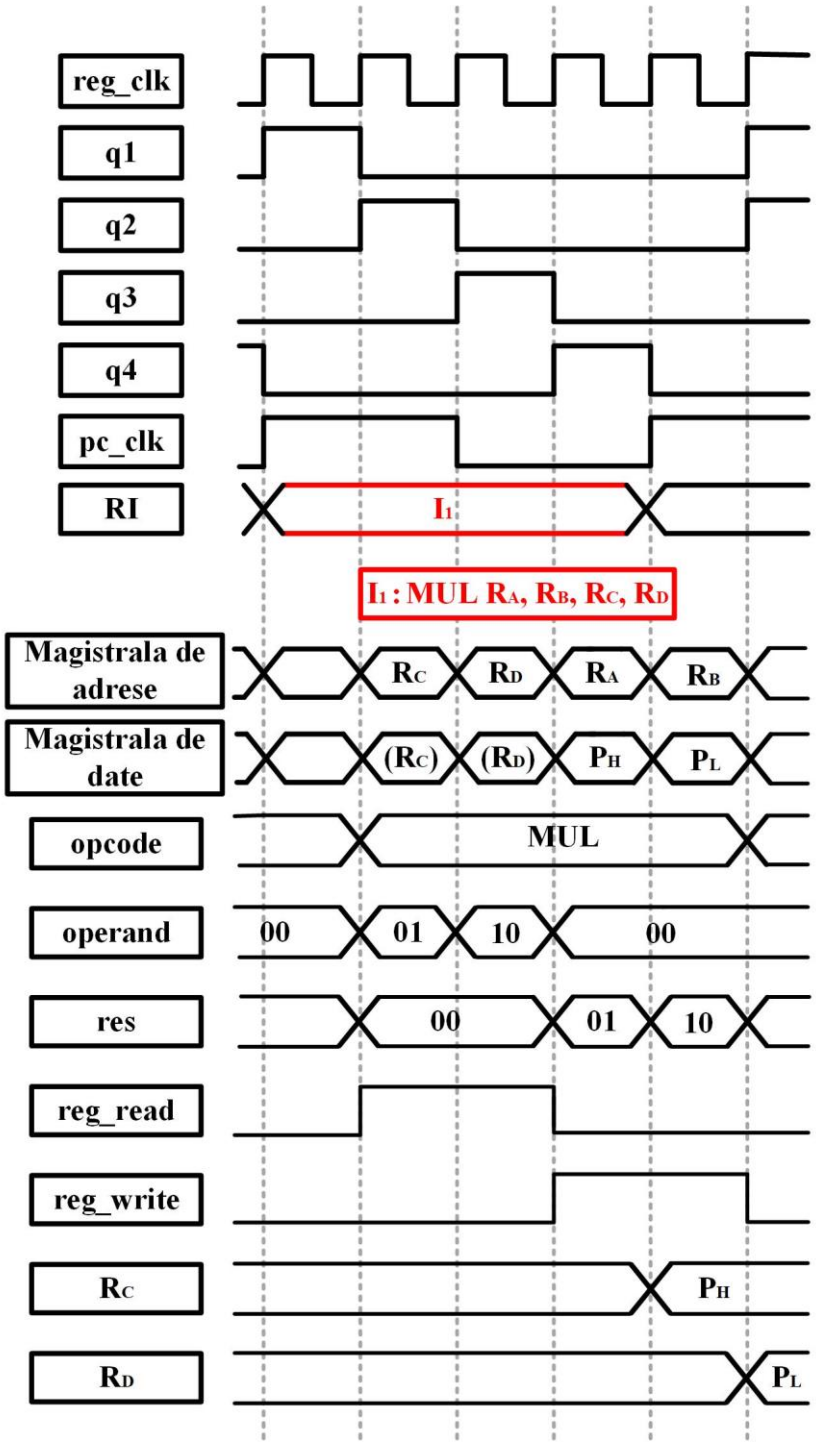


*Figura 2.3* Desfășurarea în timp a instrucțiunii MUL.

Pe primul front pozitiv al ceasului de stare *q1,* instrucțiunea de înmulțire ajunge în registrul de instrucțiune, urmând să fie decodată. Pe frontul negativ al aceluiași ceas de stare, decodarea a avut loc, și încep să se dea primele semnale de control:

- Semnalul de cod al operației (*opcode*) devine codul operației de înmulțire și este trimis spre înmulțitor
- Semnalul care specifică ce operand vine pe magistrala de date (*operand*) este dus în 1
- Semnalul care specifică ce parte din rezultat trebuie pus pe magistrala de date (*res*) este pus în 0, din două motive – întrucât la o instrucțiune de înmulțire sau împărțire sunt folosite toate cele 4 stări aferente execuției propriu-zise, o astfel de instrucțiune nu mai apucă sa facă acest semnal 0 la sfârșit, astfel încât e necesară inițializarea lui cu 0 aici, pentru a nu se pune niciun rezultat pe magistrala de date până când nu îl avem pe cel corect, și registrul corespunzător pe magistrala de adrese
- Semnalul de validare a citirii registrelor (*reg_read*) este dus în 1, întrucât urmează a se citi cele 2 registre în care se află operanzii necesari
- Pe magistrala de adrese este pusă adresa primul registru care trebuie citit, registrul sursă 1, adică $R_C$, care va fi pus în TEMP3 din schema bloc de la înmulțitor

Urmează apoi al doilea pas, reprezentat de frontul negativ al ceasului de stare *q2,* în care:

- Valoarea semnalului *operand* devine 2, pentru a anunța înmulțitorul că va ajunge al doilea operand
- Pe magistrala de adrese este pusă adresa celui de-al doilea registru ce trebuie citit, registrul sursă 2, reprezentat de $R_D$
- Semnalul de validare al citirii registrelor (*reg_read*) este menținut în 1, pentru ca cel de al doilea operand să fie adus în TEMP4

La al treilea pas, care începe pe frontul negativ al ceasului de stare *q3*, rezultatul înmulțitorului este gata, așadar semnalele vor fi după cum urmează:

- Valoarea semnalului *operand* devine 0, pentru ca înmulțitorul să nu mai preia date de pe magistrala de date și să le pună în registrele temporare
- Valoarea semnalului de validare a citirii (*reg_read*) devine 0, întrucât nu se mai citește din registre, având deja operanzii
- Valoarea semnalului de validare a scrierii în registre (*reg_write*) devine 1, întrucât se începe scrierea rezultatului în primul registru destinație $R_A$
- Semnalul privitor la rezultat *(res*) este dus în 1 pentru a specifica înmulțitorului că pe magistrala internă de date trebuie pusă partea superioara a rezultatului
- Pe magistrala de adrese este pusă adresa registrului destinație 1 $R_A$

Ultimul pas, care începe pe frontul negativ al ceasului de stare *q4* și se termina pe următorul front negativ al ceasului de stare *q1,* constă în punerea ultimei părți din rezultat în registrul corespunzător:

- Semnalul de validare a scrierii (*reg_write*) rămâne în 1
- Semnalul privitor la rezultat (*res*) este dus în 2 pentru a specifica înmulțitorului că pe magistrala internă de date trebuie pusă partea inferioară a rezultatului
- Pe magistrala de adrese este pusă adresa registrului destinație 2 $R_B$

Întrucât scrierea în registre se face pe frontul pozitiv al ceasului de registru, al doilea registru destinație va fi scris la sfârșitul stării 1, adică pe frontul negativ al ceasului de stare *q1*. În acest moment, operația este completă.

În continuare este prezentată partea din codul sursă aferentă trimiterii semnalelor de control pentru instrucțiunea de înmulțire, executată conform cu pașii discutați mai sus.

```
always @ (posedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) ctrl_alm_opcode <= 5'd0;
   else ctrl_alm_opcode <= instruction;
end

always @ (posedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) begin ... end
   else begin
      case (instruction)
         ...
         `MUL: begin
               if (q1) begin
                  ctrl_reg_write <= 0;
                  ctrl_alm_res <= 2'b00;
                  ctrl_alm_operand <= 2'b01;
                  ctrl_adr_out <= {4'b0, src1};
                  ctrl_reg_read <= 1;
               end
               else if (q2) begin
                  ctrl_adr_out <= {4'b0, src2};
                  ctrl_alm_operand <= 2'b10;
               end
               else if (q3) begin
                  ctrl_alm_operand <= 2'b0;
                  ctrl_adr_out <= {4'b0, dest1};
                  ctrl_reg_read <= 0;
                  ctrl_reg_write <= 1;
                  ctrl_alm_res <= 2'b01;
               end
               else begin
                  ctrl_adr_out <= {4'b0, dest2};
                  ctrl_alm_res <= 2'b10;
               end
         ...
      endcase
   end
end
```

*Figura 2.4* Cod sursă pentru semnalele de control la instrucțiunea MUL.

Semnalul *opcode* este pus într-un bloc separat de celelalte semnale de control deoarece el primește același câmp la fiecare instrucțiune, și nu ar fi avut rost să fie scris în cazul fiecăreia din cele 30 de instrucțiuni ale microprocesorului. Semnalele *instruction*, *src1*, *src2*, *dest1* și *dest2* sunt niște valori atribuite direct din câmpul instrucțiunii, după cum urmează:

```
assign instruction = instruction_register[31:27];
assign dest1 = instruction_register[23:20];
assign dest2 = instruction_register[19:16];
assign src1 = instruction_register[15:12];
assign src2 = instruction_register[11:8];
```

*Figura 2.5* Cod sursă pentru formatul instrucțiunii.

În Figurile 2.6 - 2.10, sunt ilustrați pasul de decodare a instrucțiunii de înmulțire și cei 4 pași de execuție discutați mai sus, cu blocurile interne și semnalele implicate. Aici, registrele sursă 1 și sursă 2 sunt $R_0$, respectiv $R_1$, iar registrele destinație 1 și destinație 2 sunt $R_4$, respectiv $R_5$. Cu negru sunt reprezentate semnalele sau magistralele neactive în pasul respectiv, iar colorate sunt semnalele sau magistralele care participă la operație în acea stare de ceas.

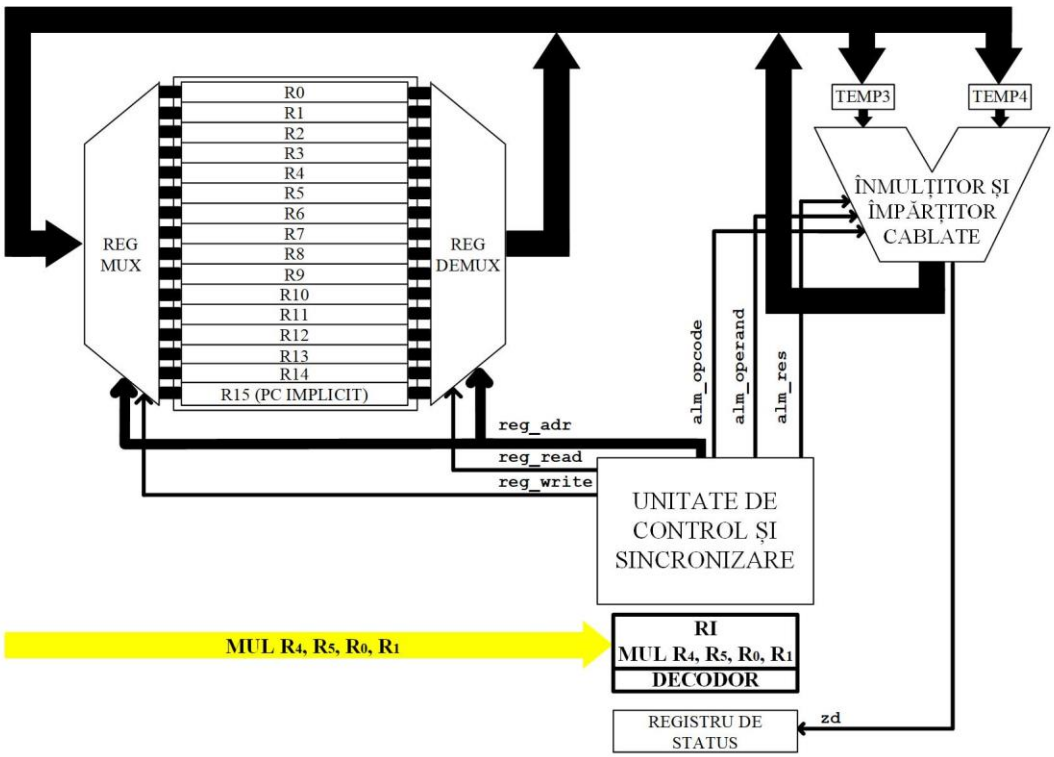


*Figura 2.6* Pasul 1 al instrucțiunii MUL.

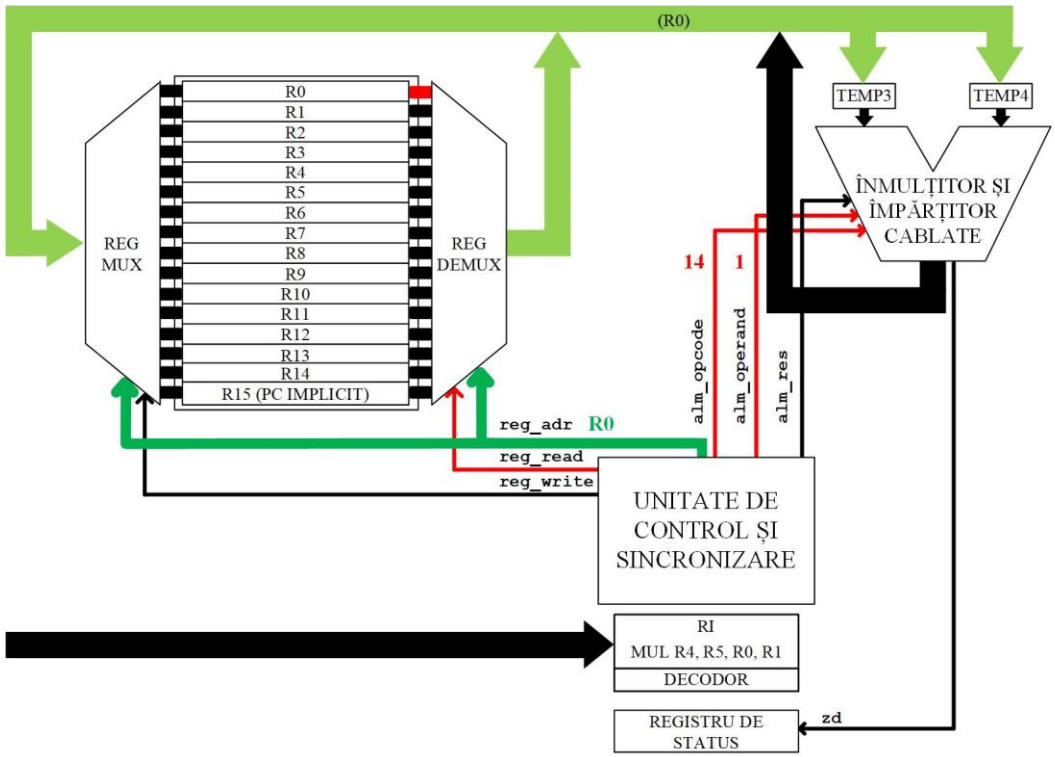


*Figura 2.7* Pasul 2 al instrucțiunii MUL.

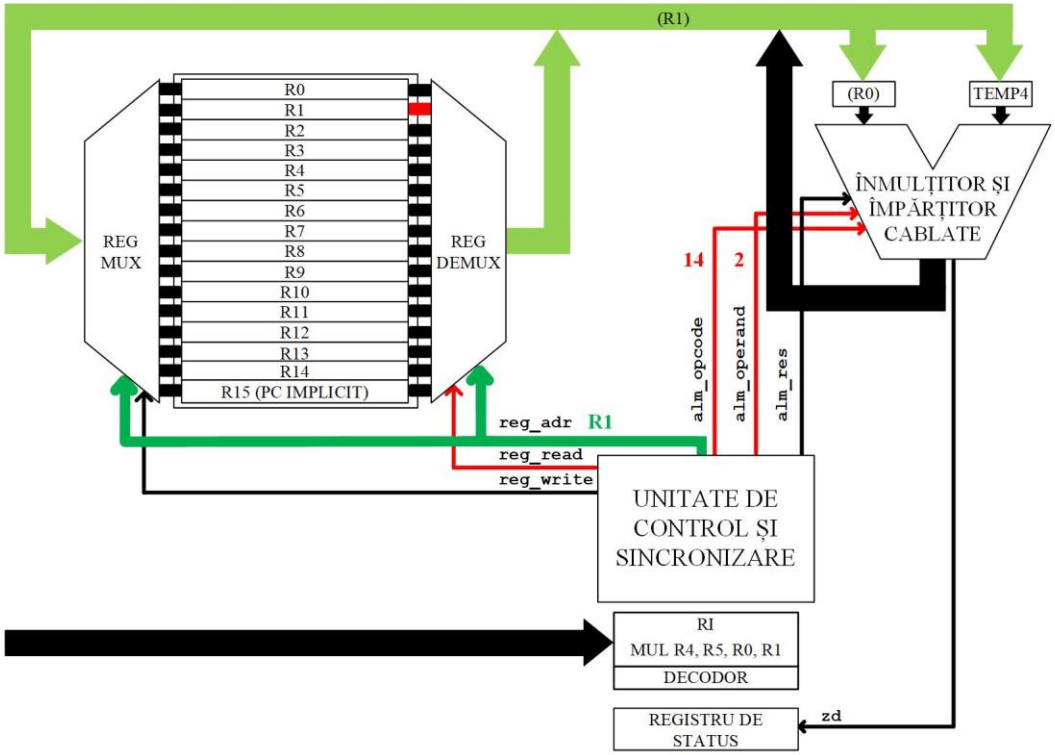


*Figura 2.8* Pasul 3 al instrucțiunii MUL.

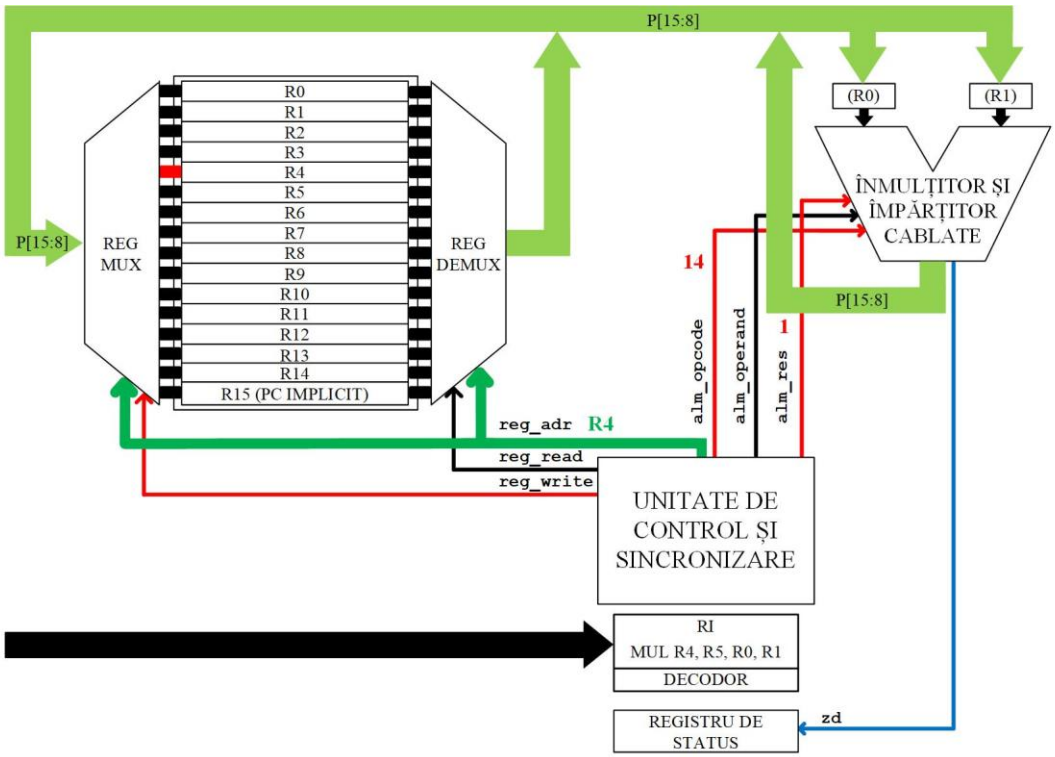


*Figura 2.9* Pasul 4 al instrucțiunii MUL.

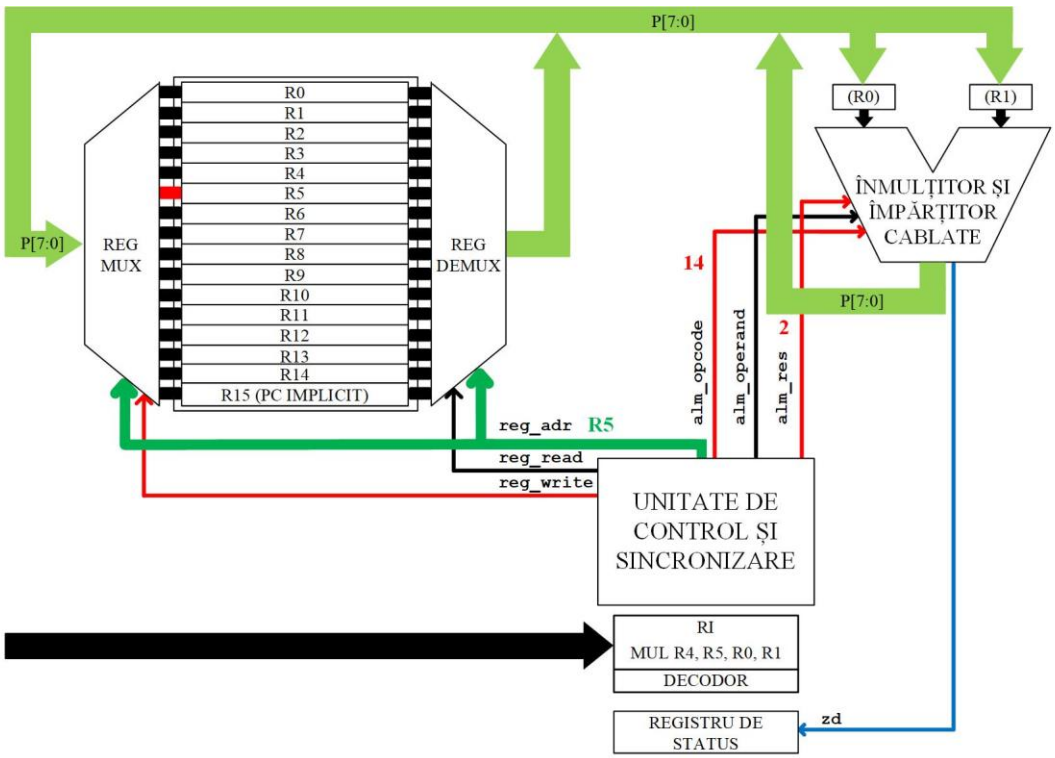


*Figura 2.10* Pasul 5 al instrucțiunii MUL.

## 2.3 Semnale de control pentru fiecare instrucțiune

Pe pagina ce urmează, în Figura 2.11, sunt trecute toate semnalele trimise de către UCS în restul microprocesorului sau spre memoria de date, pentru fiecare instrucțiune în parte.

Capetele de tabel sunt puse în secvența *q2*, *q3*, *q4* și *q1* întrucât, având în vedere cele discutate mai sus, acestea sunt cele 4 stări efective de execuție a unei operații. Astfel că dacă un semnal este activ pe o coloană, înseamnă că el este activ pe starea dată de ceasul de stare respectiv. Sub rândurile cu semnalele de control, se află și magistralele interne de date și de adrese pentru memoria de date, astfel încât să poată fi observat ce registre se citesc sau se scriu, sau ce locații de memorie se citesc sau se scriu, la un anumit moment de timp dat.

La fel ca la tabelele cu setul de instrucțiuni, pentru cele 2 magistrale avem următoarele notații:

- *(src)/(dest)* reprezintă valoarea aflată la adresa src/dest (registrul sursă/destinație);
- *((src))/((dest))* reprezintă valoarea aflată la adresa stocată în src/dest.

În josul figurii, în partea dreaptă, sunt reprezentate câteva semnale speciale care sunt activate doar în anumite situații. Ele sunt reprezentate de:

- Semnalul de întrerupere trimis spre numărătorul de program, *int_pc*
- Patru semnale pentru instrucțiunile de salt (*jam*, *anticipate_jbc*, *anticipate_ret*, *imm_jbc*) care au fost discutate mai sus
- Semnalul intern UCS *pc_change* activat de instrucțiunea PCIS care duce la schimbarea celor 4 biți de stare (*reg_ispc*) care țin adresa numărătorului de program curent
- Fanionul de transport *carry_out* care este trimis spre ALU pentru operațiile ADC și SBC
- Un semnal intern UCS *check_zero*, care este activat exclusiv de instrucțiunile CP și SUB, menit să asigure faptul că semnalul de anticipare pentru BREQ sau BRNE se dă numai după aceste instrucțiuni, și că, în plus, dacă vine o întrerupere sau o instrucțiune NOP între CP/SUB și BREQ/BRNE, saltul se va executa odată ce întreruperea/NOP a avut loc

| | LD | | | | LDI | | | | MOV | | | | POP | | | | PUSH | | | | ST | | | | ADC | | | | ADD | | | | AND | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SEMNALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| rst_instr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| asleep | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mov | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| call | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ret | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| push | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| alm_opcode (d) | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 6 | 6 | 6 | 6 | 7 | 7 | 7 | 7 | 8 | 8 | 8 | 8 |
| alm_operand(d) | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 1 | 2 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 0 | 0 |
| alm_res (d) | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| reg_read | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| reg_write | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| mem_read | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MAGISTRALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| data | (src1) | ((src1)) | ((src1)) | z | z | z | z | z | (src1) | z | z | z | z | z | z | z | z | z | z | z | (src1) | (dest1) | (src1) | z | (src1) | (src2) | result | z | (src1) | (src2) | result | z | (src1) | (src2) | result | z |
| adr | src1 | (src1) | dest1 | z | dest1 | z | z | z | src1 | dest1 | z | z | dest1 | dest1 | dest1 | z | src1 | src1 | src1 | z | src1 | dest1 | (dest1) | z | src1 | src2 | dest1 | z | src1 | src2 | dest1 | z | src1 | src2 | dest1 | z |

| | ASR | | | | CP | | | | DEC | | | | DIV | | | | INC | | | | MUL | | | | OR | | | | SBC | | | | SHL | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SEMNALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| rst_instr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| asleep | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mov | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| call | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ret | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| push | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| alm_opcode (d) | 9 | 9 | 9 | 9 | 10 | 10 | 10 | 10 | 11 | 11 | 11 | 11 | 12 | 12 | 12 | 12 | 13 | 13 | 13 | 13 | 14 | 14 | 14 | 14 | 15 | 15 | 15 | 15 | 16 | 16 | 16 | 16 | 17 | 17 | 17 | 17 |
| alm_operand(d) | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 1 | 0 | 0 |
| alm_res (d) | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| reg_read | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |
| reg_write | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| mem_read | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MAGISTRALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| data | (dest1) | (dest1) | result | z | (src1) | (src2) | z | z | (dest1) | (dest1) | result | z | (src1) | (src2) | result | result | (dest1) | (dest1) | result | z | (src1) | (src2) | result | result | (src1) | (src2) | result | z | (src1) | (src2) | result | z | (dest1) | (dest1) | result | z |
| adr | dest1 | dest1 | dest1 | z | src1 | src2 | z | z | dest1 | dest1 | dest1 | z | src1 | src2 | dest1 | dest2 | dest1 | dest1 | dest1 | z | src1 | src2 | dest1 | dest2 | src1 | src2 | dest1 | z | src1 | src2 | dest1 | z | dest1 | dest1 | dest1 | z |

| | SHR | | | | SUB | | | | XOR | | | | BREQ | | | | BRNE | | | | CALL | | | | JMP | | | | NOP | | | | PCIS | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SEMNALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| rst_instr | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| asleep | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mov | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| call | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ret | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| push | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| alm_opcode (d) | 18 | 18 | 18 | 18 | 19 | 19 | 19 | 19 | 20 | 20 | 20 | 20 | 21 | 21 | 21 | 21 | 22 | 22 | 22 | 22 | 23 | 23 | 23 | 23 | 24 | 24 | 24 | 24 | 25 | 25 | 25 | 25 | 26 | 26 | 26 | 26 |
| alm_operand | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 | 0 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| alm_res | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x | x |
| reg_read | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| reg_write | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_read | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MAGISTRALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| data | (dest1) | (dest1) | result | z | (src1) | (src2) | result | z | (src1) | (src2) | result | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z |
| adr | dest1 | dest1 | dest1 | z | src1 | src2 | dest1 | z | src1 | src2 | dest1 | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z | z |

| | RET | | | | RST | | | | SLEEP | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| SEMNALE | q2 | q3 | q4 | q1 | q2 | q2 | q4 | q1 | q2 | q3 | q4 | q1 |
| rst_instr | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| asleep | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| mov | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| call | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ret | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| push | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| pop | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| alm_opcode (d) | 27 | 27 | 27 | 27 | 28 | 28 | 28 | 28 | 29 | 29 | 29 | 29 |
| alm_operand | x | x | x | x | x | x | x | x | x | x | x | x |
| alm_res | x | x | x | x | x | x | x | x | x | x | x | x |
| reg_read | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| reg_write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_read | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| mem_write | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| MAGISTRALE | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 | q2 | q3 | q4 | q1 |
| data | z | z | z | z | z | z | z | z | z | z | z | z |
| adr | z | z | z | z | z | z | z | z | z | z | z | z |

| SEMNALE SPECIALE / TRIGGER | ÎNTRERUPERE EXTERNĂ | SALT | PCIS | ADD, ADC, SUB, SBC, INC, DEC | CP, SUB |
|---|---|---|---|---|---|
| int_pc | Trimis pe q2 după întrerupere. | x | x | x | x |
| jam | x | 0 adresare imediată relativă<br>1 adresare implicită în registru | x | x | x |
| anticipate_jbc | x | 1 pe frontul de pc_clk aferent saltului | x | x | x |
| anticipate_ret | x | 1 pe frontul de pc_clk aferent întoarcerii din apel | x | x | x |
| imm_jbc | x | valoare imediată pt. jam 0 sau adresă registru pentru jam 1 | x | x | x |
| pc_change | x | x | schimbă biții de status pt. PC | x | x |
| reg_ispc | x | x | trimite adresa PC-ului | x | x |
| carry_out | x | x | x | trimis pe negedge q4 spre ALU | x |
| check_zero | x | x | x | x | setat pentru anticipare branch |

*Figura 2.11* Semnalele de control trimise de UCS.

# Capitolul 3 - Resurse software

## 3.1 Limbajul de descriere hardware Verilog

Codul sursă pentru toate modulele realizate a fost scris în limbajul de descriere hardware Verilog, care va fi prezentat pe scurt în continuare.

Folosit în general pentru a modela partea hardware a sistemele electronice, limbajul este deseori este întâlnit în cazul proiectării, verificării sau testării de circuite digitale în RTL (deși Verilog poate fi folosit și pentru a descrie circuite la nivel comportamental sau la nivel de porți), fiind o formă literală de descriere a structurii unui circuit logic [11]. Codul sursă din acest proiect a fost scris la nivel de transfer în registre, întrucât am încercat, pe cât posibil, să scriu într-o manieră sintetizabilă – care să poată fi tradusă în circuite fizice reale (scop pentru care am evitat blocurile initial – folosit doar pentru testbench, întârzierile, forfecările, variabilele de tip real, sau atribuirea de valori acelorași registre în blocuri always diferite).

Limbajul s-a căutat să fie asemănătorul cu limbajul C, care era foarte des folosit la momentul apariției Verilog, motiv pentru care avem și aici structuri de tip condițional (if-else, case), de care am avut nevoie foarte mare în toate modulele scrise, sau bucle (while, for, repeat), din care am folosit doar bucla for, în câteva cazuri izolate (pentru calcularea fanionului de paritate sau pentru o populare inițiala a memoriei de date RAM).

Verilog folosește o logică cu 4 valori, spre deosebire de logica booleană care folosește doar 1 și 0, întrucât dispune și de valoarea x, care este o stare necunoscută pe care limbajul nu știe să o interpreteze, și care apare atunci când mai multe semnale diferite sunt trimise spre același port, sau niciun semnal nu este trimis), și încă o valoare z, de impedanță înaltă, care lasă alte semnale să dicteze valoarea portului curent [11].

Am folosit două tipuri de semnale, wire (fir) și reg. În timp ce primul are nevoie de atribuire continuă, și nu are o valoare cunoscută atunci când nu este conectat, cel de-al doilea poate reține o valoare până la următoarea atribuire, chiar și fără conexiune [11]. Legăturile între module s-au făcut exclusiv prin semnale de tip wire în modulul principal (top) din Anexa 9.

Există posibilitatea de a atribui valori semnalelor în mod combinațional, folosind un bloc de tip assign, însemnând că semnalul primește o valoare în mod continuu [11]. Acest bloc apare în multe părți ale codului propriu, și a fost în special util pentru semnalele de ieșire din module care erau legate la aceleași magistralele de date sau de adrese, întrucât în acest caz, era necesar ca aceste porturi sa acționeze precum niște buffer-e cu trei stări: țineau date pe ele dacă partea respectivă din modul se dorea citită, iar în rest erau trecute în starea de impedanță înaltă z (high impedance), pentru a evita apariția conflictelor de magistrală. Tot combinațional, mai putem atribui valori semnalelor și în interiorul unor blocuri always, care se comportă combinațional atunci când avem o listă de senzitivități pe palier, adică blocul va fi executat atunci când un semnal din lista precizată este activ sau inactiv, după caz. Important de menționat faptul ca, dacă se folosește un bloc always, variabilele atribuite în interiorul lui trebuie să fie exclusiv de tip reg. De asemenea, este de bună credință ca același semnal de tip reg să nu fie atribuit în mai multe blocuri always diferite, întrucât, dacă se dorește o sinteză la un moment dat, o astfel de implementare nu își găsește echivalentul fizic, fiind, conceptual, un registru care funcționează pe mai multe ceasuri sau semnale în locuri diferite.

Mai există și cazul în care avem un bloc always secvențial, caz în care lista de sensitivități funcționează pe fronturile semnalelor din ea (nu pe palier, ca mai sus), bloc pe care l-am folosit în cazul atribuirii semnalelor sau registrelor pe diferite fronturi de ceasuri. De asemenea, de cele mai multe ori (excepție făcând resetarea numărătorului de program), semnalul de resetare a fost folosit asincron (adică semnalele sau registrele erau puse în 0 pe frontul pozitiv al semnalului de resetare, nu pe frontul

ceasului în caz ca semnalul de resetare era activ atunci – cum ar fi fost cazul unui reset sincron). Merită menționat și faptul că atribuirea unui semnal se poate face blocant sau non-blocant, prima variantă însemnând o execuție secvențială (una după alta) a liniilor dintr-un bloc, iar a doua o execuție în paralel – de exemplu, într-un bloc always care funcționează pe un front de ceas, vrem ca toate liniile scrise acolo să se execute în același timp.

Limbajul pune la dispoziție și o serie de operatori diverși: aritmetici, relaționali, de egalitate, logici, bit cu bit și alții [11]. Operatorii aritmetici sunt cei de tip adunare, scădere, înmulțire etc. Operatorii relaționali se referă la cei prezenți și în C, de tip comparație, rezultatul acestei operații putând fi 0 (când relația este falsă), 1 (când relația este adevărată) sau x (stare necunoscută, datorată existenței unor biți necunoscuți în operanzi). Operatorii de egalitate sunt și ei de două tipuri: logici, care pot avea drept rezultat și pe x, dacă unul dintre operanzi conține biți x sau z, sau de caz, care testează și potrivirea biților de x sau de z din operanzi, astfel că rezultatul va fi 0 sau 1. Operatorii logici pot avea drept rezultat 0 (relație falsă), 1 (relație adevărată) sau x dacă în operanzi există biți necunoscuți. Ultimul tip de operatori sunt cei care execută operațiile pe operanzi bit cu bit, însemnând că fiecare bit dintr-un operand este luat și trecut prin operație cu bitul corespunzător din celălalt operand, care poate conține și biți x (cu relații precum 0 & x = 0, 1 & x = x, 0 | x = x, 1 | x = 1) [11].

În acest limbaj, numerele negative sunt implicit reprezentate în complement față de doi (care se obține complementând biții unui număr binar și adăugând 1 la valoarea obținută) [11].

Modulele scrise în Verilog au fost simulate prin intermediul testelor (testbenches), în care intrările unui modul se declară de tip reg, ieșirile de tip wire, modulul este instanțiat, iar apoi este introdus un bloc de tipul initial, pentru a da stimulul necesar simulării, care poate conține ceasuri, semnale de resetare sau alte semnale ce pornesc circuitul [12, 13].

De asemenea, menționez faptul că am folosit două editoare de text pentru scrierea codului, unul în sistemul de operare Windows, Sublime Text, și gedit în Linux.

## 3.2 Suita de instrumente software Incisive de la Cadence Design Systems

Incisive este numele pe care îl poartă o serie de instrumente software de la Cadence Design Systems, folosite pentru proiectarea și verificarea circuitelor digitale de tipul plăcilor FPGA sau ASIC-urilor, făcând așadar parte din tehnologia CAD. Programele sunt de mai multe tipuri:

- Compilatoare (NCVerilog, NCVHDL, NCSystemC)
- Linker/elaborator pentru limbajele Verilog, VHDL și SystemC (NCElaborator), menit să genereze un fișier obiect pentru simulare, denumit, în general, snapshot (instantaneu)
- Motor de simulare pentru limbajele menționate mai sus, care încarcă fișierele generate de NC Elaborator, care, rulat prin intermediul unei interfețe grafice, este similar cu depanatorulModelSim
- Simulatorul SimVision, folosit în principal pentru vizualizarea formelor de undă și urmărirea atribuirii diferitelor semnale în codul sursă la anumite momente de timp

Am folosit interfața grafică NCLaunch pentru a putea realiza comunicarea cu instrumentele software folosite, întrucât ea conține tot ce este necesar în proiectul în cauză (fișierele Verilog care conțin codul sursă al microprocesorului, librăriile cds.lib în care sunt compilate aceste fișiere și legătura cu programele precum compilatorul NCVerilog, linker-ul NCElaborator și simulatorul SimVision) [14].

Accesarea NCLaunch s-a făcut de la distanță, pe contul personal de pe serverul flash.dcae.pub.ro, creat în timpul perioadei de practică de vară din anul 3 la compania Microchip Technology.

Prima fereastră care apare la deschiderea interfeței grafice este cea din Figura 3.1, în care există 3 subferestre diferite: cea cu directorul curent, în care se află fișierele Verilog și din care am rulat programul (în stanga sus), cea cu librăriile disponibile și librăriile create pentru proiect (în dreapta sus)

și o linie de comandă care nu va fi direct folosită (ci accesibilă prin icoanele marcate pe figură), însă în care se vor afișa erorile apărute la compilare sau elaborare.


*Figura 3.1* Fereastra de pornire a interfeței grafice NCLaunch.

În urma etapelor de compilare a codului sursă și elaborare a modulului aferent testbench-ului, fereastra de comandă arată ca în figura de mai jos, pe care sunt marcate comenzile de compilare pentru Verilog (ncvlog) și pentru elaborare (ncelab), în urma cărora obținem snapshot-ul necesar simulării.

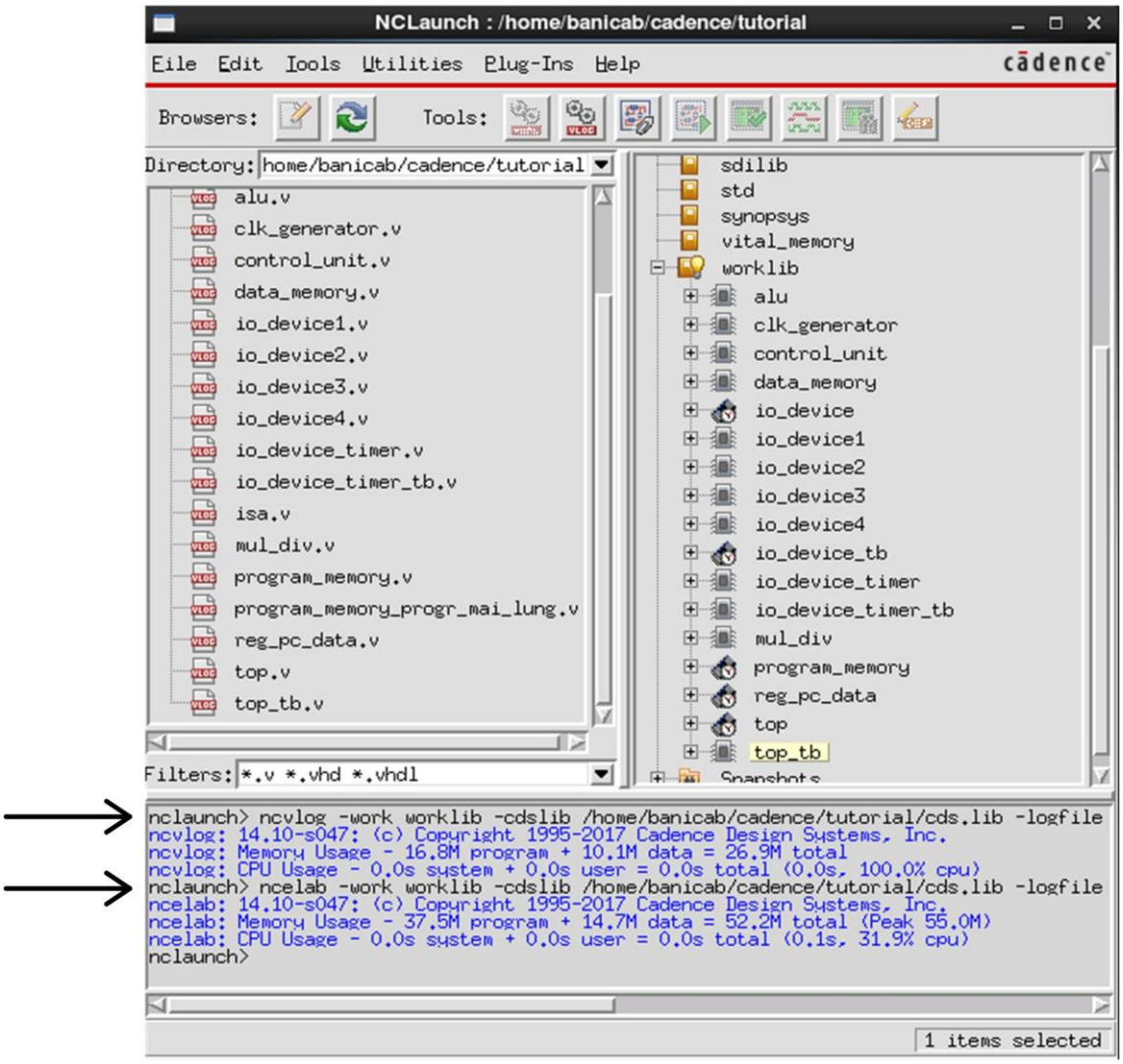


*Figura 3.2* Compilatorul NCVerilog și elaboratorul NCElaborator.

Odată cu încărcarea snapshotului în simulatorul SimVision, sunt deschise două ferestre diferite, ilustrate mai jos. Una dintre ele, numită Design Browser, conține ierarhia modulelor și a semnalelor din ele, și este folosită pentru a deschide o nouă fereastră cu formele de undă pentru anumite semnale sau module alese, iar cealaltă este o consolă, în care sunt afișate diverse mesaje referitoare la simulare sau posibile afișări de mesaje cerute de codul sursă (de exemplu, în Verilog, am putea folosi *$display*). Pentru a vizualiza formele de undă putem fie să folosim comanda Send to Waveform Window, pentru modulul sau semnalele care ne interesează, sau, mai ales în etapa de depanare, în care o simulare poate fi rulată de multe ori până se ajunge la soluționarea eventualelor probleme, ne putem folosi de posibilitatea de a salva un fișier de tip .tcl, adică un fișier care salvează un script de comenzi folosit pentru a ne readuce în punctul în care suntem, cu semnalele curente afișate [15]. Comanda pentru această opțiune este Save Command Script, iar pentru a încărca acest fișier la următoarea simulare, ne folosim de comanda Source Command Script, vizibilă în Figura 3.3.

În figură se observă ierarhia modulelor, iar în partea dreapta semnalele din modulul selectat, precum și valorile lor în caz ca există o simulare curentă și avem cursorul plasat pe un anumit moment de timp. De asemenea, avem posibilitatea de a căuta anumite semnale, aspect util mai ales în cadrul proiectelor foarte mari, cu module numeroase.

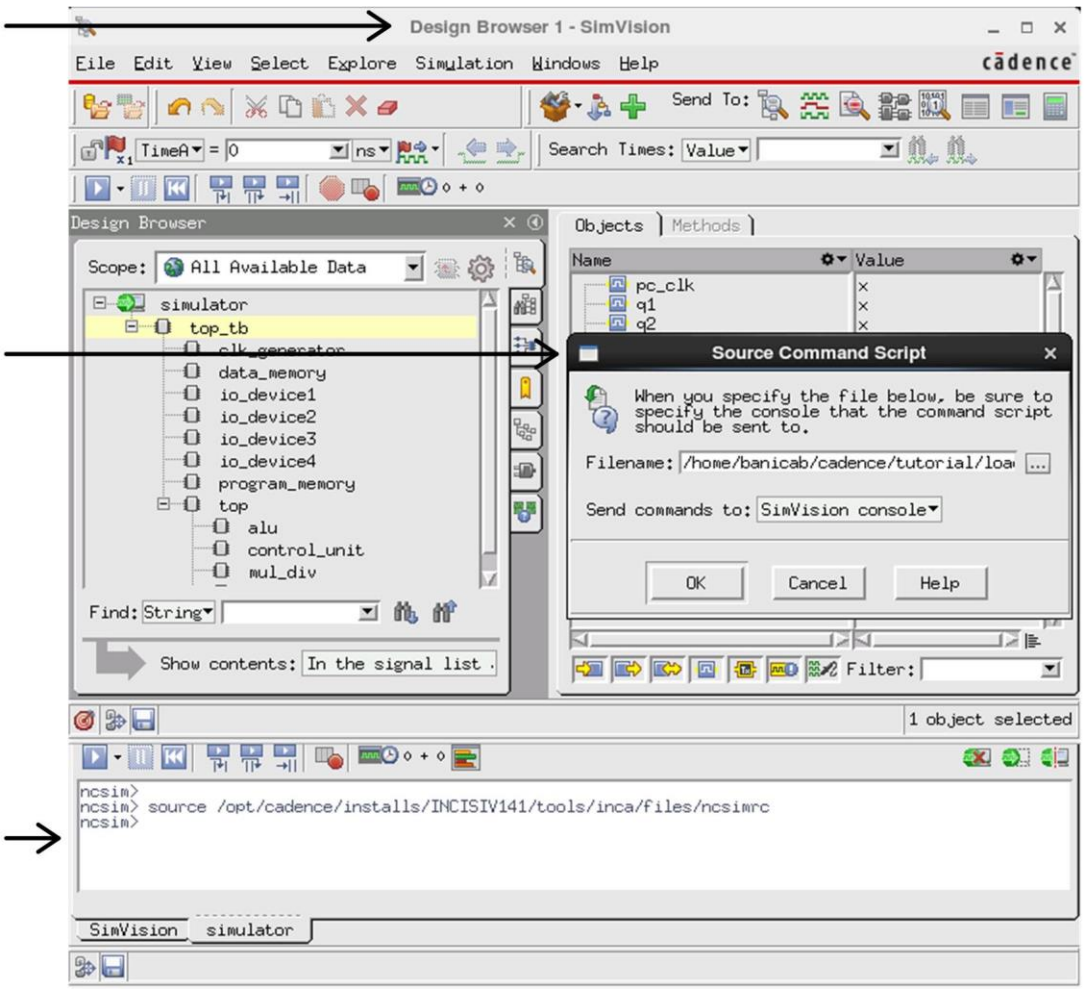


*Figura 3.3* Ferestrele simulatorului SimVision.

În final, se poate deschide o fereastră ca cea din Figura 3.4, cu formele de undă dorite, foarte asemănătoare celei din ModelSim, care vine cu instrumentele uzuale pentru o astfel de simulare (cursori de timp, posibilitatea de afișare a semnalelor în zecimal/binar/ASCII/hexazecimal, puncte de oprire sau trimiterea anumitor semnale în codul sursă și urmărirea atribuirii lor pas cu pas).

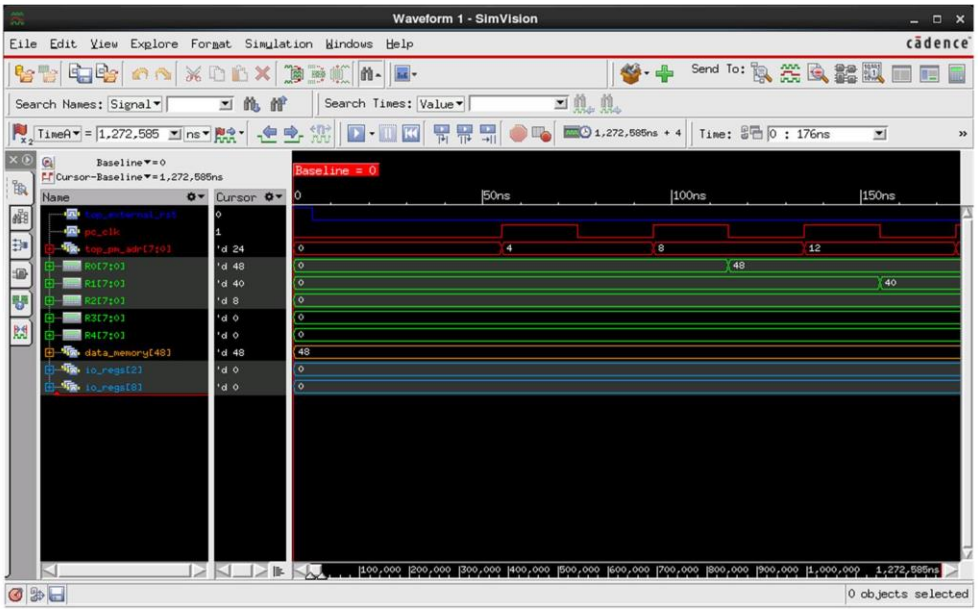


*Figura 3.4* Forme de undă în SimVision.

# Capitolul 4 - Rularea programelor din ROM pe microprocesor

## 4.1 Simulări

Pentru testarea codului sursă scris, am realizat atât module de test individuale pentru fiecare bloc în parte, cât și un modul de test (din Anexa 10) pentru întregul microprocesor, în care am instanțiat modulul principal cuprinzând setul de registre (deci și PC) și stiva, ALU, înmulțitorul/împărțitorul cablate și unitatea de control și sincronizare, împreună cu registrele de instrucțiune și de stare. În modulul de test aferent acestuia, am adăugat modulul de generare a ceasurilor, memoria de program, memoria propriu-zisă de date și patru dispozitive de intrare ieșire identice. Folosind mai multe secțiuni diferite în cadrul modulului ROM, am testat câteva programe diferite menite să testeze funcționalitatea tuturor tipurilor de instrucțiuni. Am ales să ilustrez 3 dintre ele, cu echivalentul în limbaj de asamblare al instrucțiunilor din memorie (pentru o mai bună înțelegere a ceea ce trebuie să facă respectiva parte din ROM) și ferestrele formele de undă pentru registrele sau locațiile de memorie implicate în respectivul program.

```
PC = 0: NOP                    //nicio operație
PC = 4: LDI R3, #16            //(R3) ← 16
PC = 8: LDI R4, #15            //(R4) ← 15
PC = 12: SUB R5, R3, R4        //(R5) ← (R3) – (R4)
PC = 16: BRNE #16              //dacă Z = 0, (PC) ← (PC) + 16
PC = 32: LDI R6, #3            //(R6) ← 3
PC = 36: LDI R7, #4            //(R7) ← 4
PC = 40: MUL R10, R11, R6, R7 //(R10, R11) ← (R6)*(R7)
PC = 44: DIV R1, R7, R11, R6  //(R1) ← (R11) DIV (R6), (R7) ← (R11) MOD (R6)
PC = 48: PUSH R1               //(SP) ← (SP) + 1; ((SP)) ← (R1)
PC = 52: POP R2                //(R2) ← ((SP)); (SP) ← (SP) - 1
PC = 56: ST R6, R4             //((R6)) ← (R4)
PC = 60: LD R0, R6             //(R0) ← ((R6))
PC = 64: MOV R8, R0            //(R8) ← (R0)
PC = 68: LDI R9, #255          //(R9) ← 255
PC = 72: INC R9                //(R9) ← (R9) - 1
PC = 76: ADC R12, R8, R5       //(R12) ← (R8) + (R5) + C
PC = 80: LDI R10, #96          //(R10) ← 96
PC = 84: CALL R10              //(PC) ← (R10)
PC = 88: MOV R9, R14           //(R9) ← R14
PC = 96: LDI R13, #13          //(R13) ← 13
PC = 100: ADD R14, R13, R6     //(R14) ← (R13) + (R6)
PC = 104: RET                  //(PC) ← ((SP)); (SP) ← (SP) - 1
PC = 92: SLEEP                 //(PC) ← (PC)
```

*Figura 4.1* Limbaj de asamblare pentru programul de test general.

În programul din Figura 4.1, am ales, pe cât posibil, ca instrucțiunile să depindă unele de altele secvențial, astfel încât să poată fi urmărită ușor execuția corectă a acestora. Instrucțiunea cea mai la îndemână pentru a porni un astfel de program de test este LDI, întrucât fără a încărca anumite valori în registre, este greu să urmărim funcționalitatea arhitecturii – execuția corectă a programului. Alternativa este LD, adică aducerea datelor din memoria de date, care oricum este populată inițial cu niște valori arbitrare. Se testează, în programul de mai sus, operațiile ALU monadice și diadice, operațiile de înmulțire și împărțire, accesul în memorie, operațiile de transfer cu stiva, saltul condiționat și apelul de subrutină, astfel încât se trece printr-o mare parte a setului de instrucțiuni.

În Figura 4.2 este prezentat programul prin care am ales să testez funcționalitatea porturilor, în conformitate cu cele discutate în subcapitolul 1.6, în care am menționat faptul că prima locație a fiecărui dispozitiv de intrare/ieșire nu poate fi scrisă, fiind registrul de stare, iar cea de-a doua nu poate fi citită, fiind registrul de configurare.

```
PC = 0: NOP               //nicio operație
PC = 4: LDI R0, #48       //(R0) ← 48
PC = 8: LDI R1, #40       //(R1) ← 40
PC = 12: SUB R2, R0, R1   //(R2) ← (R0) – (R1)
PC = 16: BREQ #16         //dacă Z = 0, (PC) ← (PC) + 16
PC = 20: ST R2, R1        //((R2)) ← (R1)
                          //nu pune, fiind registrul de stare al unui port
PC = 24: ST R0, R1        //((R0)) ← (R1)
PC = 28: INC R2           //(R2) ← (R2) + 1
PC = 32: LD R1, R2        //(R1) ← ((R2))
                          //(9 – este un registru de configurare care nu
                            poate fi citit, deci în R1 pune z)
PC = 36: LD R1, R0        //(R1) ← ((R0))
PC = 40: LDI R3, #2       //(R3) ← 2
PC = 44: ST R3, R1        //((R3)) ← (R1)
PC = 48: LD R4, R3        //(R4) ← ((R3))
PC = 52: SLEEP            //(PC) ← (PC)
```

*Figura 4.2* Limbaj de asamblare pentru programul de test al memoriei și al porturilor.

În Figura 4.3 este ultimul program de test pe care am ales să îl prezint în acest capitol, care verifică, în principal, execuția instrucțiunii de schimbare a numărătorului de program PC, și modul în care ea afectează instrucțiunea precedentă sau pe cea următoare ei. Am folosit de trei ori instrucțiunea NOP, deoarece aceasta este necesară în fiecare din următoarele cazuri:

- Instrucțiunea de înmulțire (MUL) termină scrierea rezultatelor pe frontul negativ al ceasului de stare *q1*, așadar la o stare după frontul pozitiv al ceasului de instrucțiune, moment în care $R_{10}$ a devenit deja numărător de program, fapt care împiedică scrierea în el în afara instrucțiunilor de salt; NOP-ul plasat înaintea înmulțirii are, deci, rolul de a da răgaz registrului să se actualizeze pentru ca adresa de program la care dorim să ajungem să fie cea corectă (64 în cazul de față), el nefiind necesar dacă înainte de PCIS avem o altă instrucțiune de scriere a registrului în afara de MUL sau DIV
- NOP-ul aflat la adresa din $R_{10}$ este necesar deoarece, dacă am pune o instrucțiune utilă acolo, ea nu ar fi executată, întrucât valoarea numărătorului de program trimisă spre memoria de program este egală cu ce este în $R_{10}$, într-adevăr, însă odată ce $R_{10}$ devine PC, el începe să numere, incrementându-și valoarea cu 4 la primul front pozitiv al ceasului de instrucțiune următor, după cum apare în Figura 4.4; se poate vedea acolo că nu se face fetch la instrucțiunea aflată la adresa 64
- NOP-ul plasat înaintea saltului necondiționat la adresa din $R_2$ își are scopul în faptul că valoarea din $R_2$ este înmulțită cu 2 la finalul stării *q4*, adică fix pe frontul pozitiv al ceasului de instrucțiune, după ce deja s-a trimis spre PC valoarea de salt, care, deci, nu ar fi cea pe care ne-o dorim, ci una neactualizată încă

```
PC = 0: LDI R8, #8              //(R8) ← 8
PC = 4: LDI R9, #10             //(R9) ← 10
PC = 8: ADD R10, R8, R9         //(R10) ← (R8) + (R9)
PC = 12: MOV R15, R10           //(R15) ← (R10); nu se poate, este PC
PC = 16: ASR R10                //(R10) ← (R10) >>> 1
PC = 20: DEC R10                //(R10) ← (R10) - 1
PC = 24: MUL R11, R10, R10, R8 //(R11, R10) ← (R10)*(R8)
PC = 28: NOP                    //nicio operație
PC = 32: PCIS R10               //(PC) ← (R10), R10 e PC
PC = 64: NOP                    //nicio operație
PC = 68: LDI R2, #6             //(R2) ← 6
PC = 72: SHL R2                 //(R2) ← (R2) << 1
PC = 76: NOP                    //nicio operație
PC = 80: JMP R2                 //(PC) ← (R2)
PC = 84: SLEEP                  //(PC) ← (PC)
```

*Figura 4.3* Limbaj de asamblare pentru programul de test al instrucțiunii PCIS.

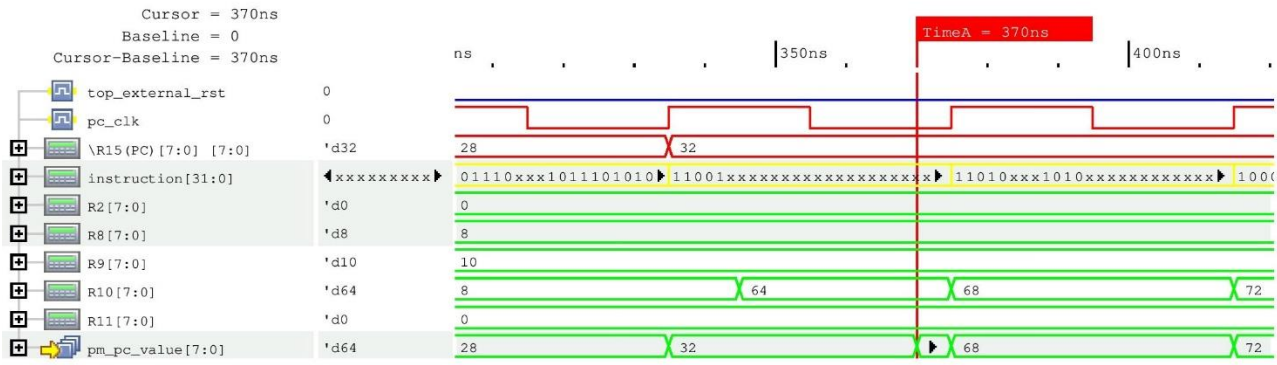


*Figura 4.4* Necesitatea instrucțiunii NOP la instrucțiunea PCIS.

În continuare sunt ilustrate rezultatele acestor simulări, prin formele de undă ale semnalelor și registrelor implicate.

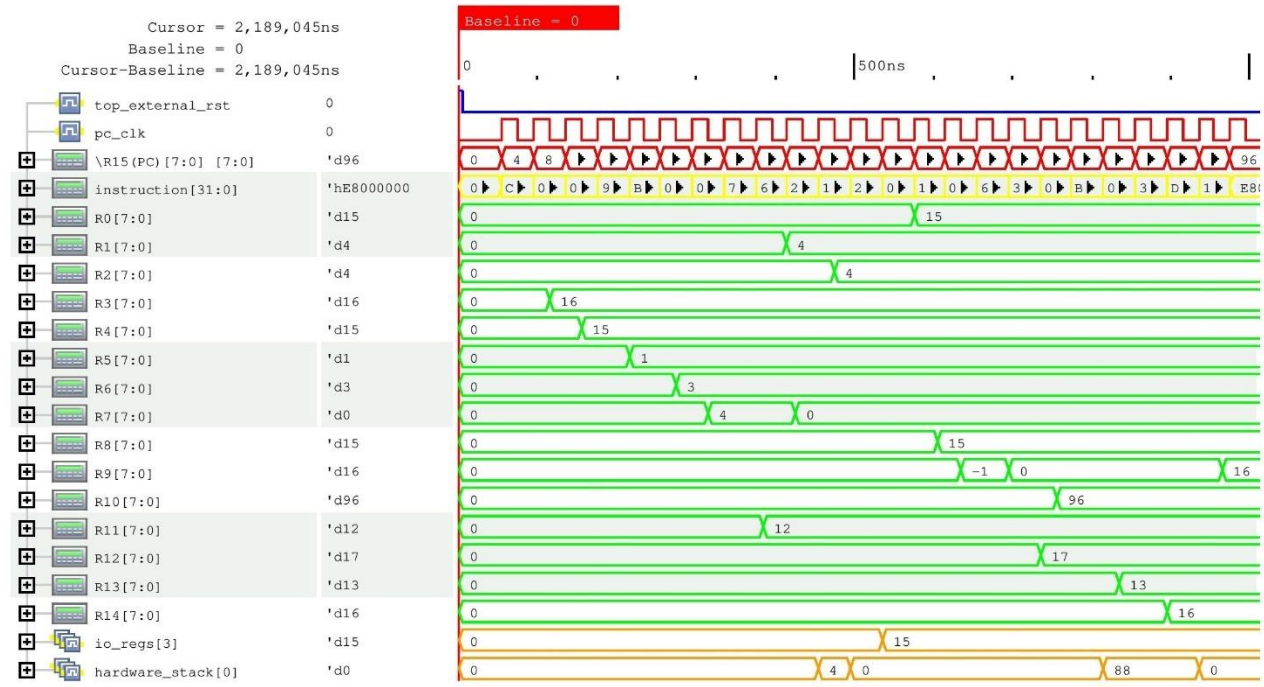


*Figura 4.5* Forme de undă pentru programul de test general.

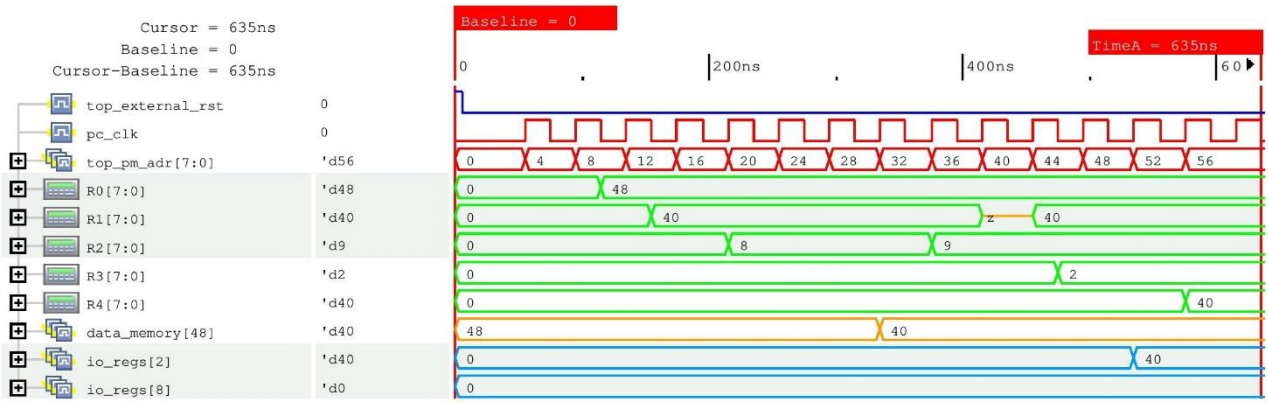


*Figura 4.6* Forme de undă pentru programul de test al memoriei și al porturilor.

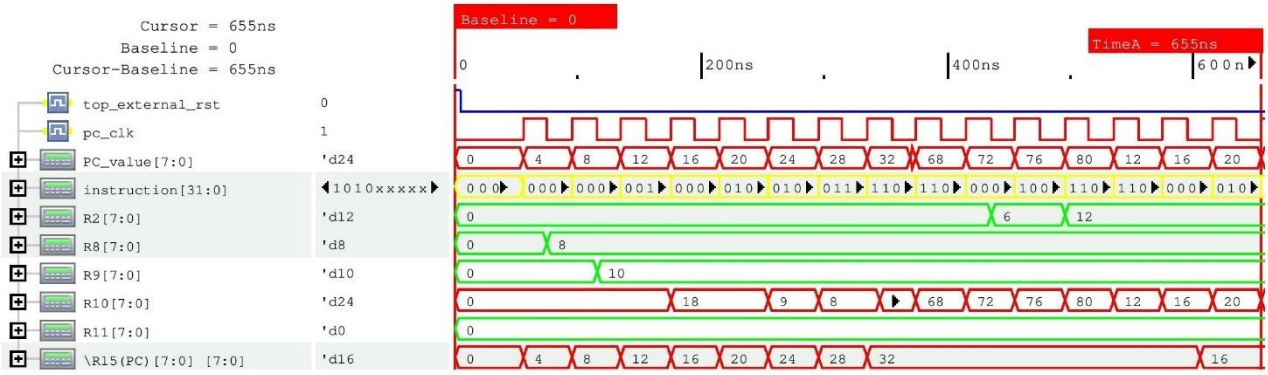


*Figura 4.7* Forme de undă pentru programul de test al instrucțiunii PCIS.

Programele descrise în paragrafele precedente nu s-au executat corect încă de la prima simulare, apărând, pe parcurs, o serie de probleme de diferite tipuri. Cele mai multe au fost de sincronizare, însă am întâlnit și câteva datorate unor semnale necunoscute x sau unor conflicte pe magistralele microprocesorului. Am ales să vorbesc despre cele pe care le-am considerat mai importante în subcapitolul următor.

## 4.2 Probleme întâmpinate și soluțiile găsite

### 4.2.1 Fronturile de ceas

Una dintre primele probleme întâmpinate a fost legată de modul în care se generau cele 6 ceasuri necesare microprocesorului. Am descris în subcapitolul 2.2, cu desfășurarea în timp a instrucțiunilor, faptul că o perioadă de ceas de instrucțiune este echivalentă cu 4 perioade de ceas de registru și cu succesiunea de pulsuri *q1*, *q2*, *q3*, *q4*. Modul în care voiam să funcționeze aceste ceasuri era următorul: trebuia ca 4 fronturi diferite să reprezinte exact același moment de timp – frontul negativ al ceasului de stare *q4*, frontul pozitiv al ceasului de stare *q1*, frontul pozitiv aferent al ceasului de registru și frontul pozitiv al ceasului de instrucțiune. Pe baza acestei funcționări se baza așteptarea ca, dacă un semnal oarecare devenea activ sau dacă un registru își schimba valoarea pe unul dintre cele 4 fronturi, celelalte 3 să nu îl vadă, totuși, activ, decât la următorul front pozitiv de ceas de registru. Mai concret, dacă presupunem că vorbim despre semnalul de control *res*, din Figura 2.3, el devine 2 pe frontul negativ al lui *q4*, condiția fiind, după cum apare în Figura 2.4, să avem instrucțiunea MUL în registrul de instrucțiune. Aici a survenit problema: la acest front negativ al lui *q4*, echivalent cu frontul pozitiv al ceasului de program, registrul de instrucțiune primea altă valoare, fapt care era observat de simulator pe același front, deci *res* nu mai apuca să devină 2. Același lucru se întâmpla și cu fronturile pozitive al ceasului de instrucțiune și cel aferent al ceasului de registru. Soluționarea a stat în schimbarea codului îngroșat din Figura 4.8 în cel din Figura 4.9, astfel încât ambele să depindă de un ceas intermediar *basic_clk*.

```
initial begin
   reg_clk = 0;
   forever #5 reg_clk = ~reg_clk;
end

always @(posedge q1 or posedge q3 or posedge rst) begin
   if (rst || q3) pc_clk <= 0;
   else pc_clk <= 1;
end

always @(posedge reg_clk or posedge rst) begin
   if (rst) q <= 4'b0;
   else if (q == 0) q <= 4'b0001;
   else q <= {q[2:0], q[3]};
end

assign q1 = q[0];
assign q2 = q[1];
assign q3 = q[2];
assign q4 = q[3];
```

*Figura 4.8* Cod sursă inițial pentru ceasurile de instrucțiune și de registru.

```
initial begin
   basic_clk = 0;
   forever #5 basic_clk = ~basic_clk;
end

always @ (posedge basic_clk or negedge basic_clk or posedge rst) begin
   if (rst) reg_clk <= 0;
   else reg_clk <= basic_clk;
end

always @(posedge basic_clk or posedge rst) begin
   if (rst || q2) pc_clk <= 0;
   else if (q4) pc_clk <= 1;
end
```

*Figura 4.9* Cod sursă final pentru ceasurile de instrucțiune și de registru.

### 4.2.2 Instrucțiunile de salt

O altă problemă a constat în faptul că, la instrucțiunile de salt, în registrul de instrucțiune se punea și instrucțiunea imediat următoare saltului înainte ca saltul propriu-zis să se realizeze, lucru pentru care existau 2 soluții:

- Fie utilizatorul trebuia sa introducă un NOP după orice instrucțiune de salt, astfel încât să nu se execute nimic propriu-zis între cererea de salt și instrucțiunea aflată la adresa de salt;
- Fie se anticipa faptul că urmează un salt, decodând primul octet din buffer-ul registrului de instrucțiune.

Am ales a doua variantă, prin care, monitorizând mai degrabă intrarea de instrucțiuni în modulul cu UCS, după cum apare în Figura 1.18, se trimit spre numărătorul de program valoarea imediată (pentru adresarea imediată relativă) sau adresa registrului (pentru adresarea implicită în registru) înainte ca instrucțiunea de salt să fie pusă în registrul de instrucțiune. Astfel, numărătorul de program poate să se actualizeze la primul front pozitiv al ceasului de program de după intrarea instrucțiunii de salt în buffer. Apare întrebarea: dacă semnalul de salt se dă înainte ca instrucțiunea de salt să fie propriu-zis în registrul de instrucțiune, atunci ce acțiuni au loc în timp ce ea este în registru? Pentru CALL, se pune în stivă valoarea de întoarcere din subrutină (instrucțiunea imediat următoare instrucțiunii de salt), însă pentru JMP sau BREQ/BRNE, într-adevăr nu se mai dau și semnale de control în plus, doar se dezactivează anumite semnale care nu au apucat să fie dezactivate de instrucțiunile care le-au folosit (de exemplu, *reg_write*, care nu este dezactivat de DIV sau MUL).

De asemenea, o altă problemă legată de execuția salturilor a fost, în urma soluționării necesității NOP-ului, faptul că trimiterea anticipată a unei valori imediate sau a adresei unui registru se făcea pe aceleași magistrale pe care instrucțiunea aflată în acel moment în registrul de instrucțiune își trimitea operanzii sau rezultatele, apărând astfel un conflict pe magistrală ce ducea la o execuție incorectă. Soluția a fost ca valorile imediate pentru salt sau adresele registrelor în cauză să fie trimise pe „magistrala" dedicată *imm_jbc* din Tabelul 1.1. Este bine cunoscut faptul că, mai ales în cazul implementării unei tehnici pipeline, salturile pot cauza diverse probleme, de aici și necesitatea introducerii unei instrucțiuni precum NOP [15].

Tot la instrucțiunile de salt, în momentul testării comportamentului programului când pe pinul de întrerupere apare o astfel de cerere, am observat că dacă întreruperea venea în timpul unei comparații (CP) sau scăderi (SUB) după care urma un salt condiționat (de fanionul Z) sau între ele, odată ce microprocesorul executa rutina de întrerupere și se întorcea la programul principal, indiferent de rezultatul scăderii, saltul condiționat nu se mai executa. Acest lucru se întâmpla deoarece UCS forțează microprocesorul să realizeze un astfel de salt doar după o instrucțiune CP sau SUB, astfel că întreruperea făcea să se piardă informația prezenței CP sau SUB în registrul de instrucțiune. Soluția a constat în crearea unui semnal în interiorul UCS care să fie activat de CP și SUB și dezactivat doar de BREQ și BRNE, semnal care condiționează trimiterea comenzii de salt spre numărătorul de program.

### 4.2.3 Alte dificultăți

Pe parcursul realizării proiectului, am mai întâmpinat o serie de probleme mai puțin semnificative, dar care au necesitat, oricum, numeroase rescrieri ale anumitor părți din cod și, deci, foarte multe simulări. Printre acestea, se numără:

- Găsirea unui mod optim de sincronizare între valoarea numărătorului de program trimisă spre memoria de program și modul în care se aduc instrucțiunile în RI, având în vedere faptul că magistrala internă de instrucțiuni este de doar 8 biți, iar formatul unei instrucțiuni are 32 de biți;
- Conflictele pe magistrala internă datorate multiplelor blocuri care puneau date pe ea în același timp, necomandate corect de UCS;
- Implementarea algoritmului Booth și pentru cazul în care înmulțitorul era valoarea maximă (în modul) negativă reprezentabilă pe 8 biți;
- Preincrementarea indicatorului de stivă la CALL și PUSH sau postdecrementarea lui la RET și POP în momentele corecte de timp astfel încât în datele din stivă să fie corecte și în pozițiile potrivite;
- Trimiterea fanioanelor de la ALU la registrul de stare și actualizarea corectă a acestuia odată ce instrucțiunile în cauză își terminau execuția.

# Concluzii

## Concluzii generale

Proiectul a pornit cu scopul de a implementa în totalitate arhitectura propusă inițial în limbajul de descriere hardware Verilog și asigurarea funcționalității ei prin testarea unor secvențe de instrucțiuni menite să treacă prin întregul set de instrucțiuni stabilit la început. De asemenea, am avut în vedere încă de la început evaluarea modului în care aceste programe se execută în timp și apropierea acestei dimensiuni temporale de cele specifice mașinilor RISC.

Toate elementele de arhitectură din specificațiile inițiale au fost traduse cu succes în module Verilog, care au fost testate întâi individual pentru a mă asigura că, odată ajunsă la pasul de integrare a lor într-un modul principal de top al microprocesorului, probleme apărute se vor datora exclusiv sincronizărilor sau legăturilor dintre blocuri, și nu hibelor specifice unui anumit modul. În continuare, am evaluat executarea tuturor tipurilor de instrucțiuni și am soluționat problemele apărute, astfel încât, în final, microprocesorul funcționează corect de-a lungul întregului set de instrucțiuni. Am obținut o mașină pentru care se poate crea ușor un compilator de limbaje de nivel înalt, prin flexibilitatea de folosire a resurselor interne pe care o oferă utilizatorului.

În ceea ce ține de dimensiunea temporală a arhitecturii, anume timpul de execuție pentru o instrucțiune, microprocesorul reușește să se apropie de unul de tip RISC, prin faptul că el are un rezultat la fiecare ceas de instrucțiune, sau, altfel spus, la fiecare patru stări (întrucât am ales ca magistralele interne de date și de instrucțiune să fie de 8 biți, la fel ca operanzii, necesitând mai mulți pași de transfer), fapt datorat unei oarecare forme de pipeline, în cadrul căreia aducerea unei instrucțiuni din memoria de program are loc în același timp cu decodarea și execuția instrucțunii din registrul de instrucțiune. Se poate spune, deci, dacă o perioadă a ceasului de registru este considerată un ciclu, că avem un CPI 4 (patru astfel de perioade pentru execuția unei instrucțiuni). Toate instrucțiunile desfășurându-se în aceste patru stări, putem vorbi despre o uniformitate în timp a arhitecturii.

Alte caracteristici care fac ca mașina implementată să fie, într-o bună proporție, un model RISC, sunt numărul mic de instrucțiuni disponibile, numărul mic de moduri de adresare folosite, arhitectura de tip load-store (transferurile cu memoria sau cu porturile făcându-se doar prin aceste instrucțiuni), formatul fix al instrucțiunii, operațiile ALU sau de înmulțire și împărțire doar între registrele generale.

Există, cu siguranță, și abateri de la modelul RISC – un format al instrucțiunii de 32 de biți în timp ce operanzii și magistralele sunt pe 8 biți sau durata execuției propriu-zise de 4 stări în loc de una singură (datorată unei unități de control cablate) în ciuda uniformității instrucțiunilor în timp, însă, în ziua de astăzi, nicio mașină nu mai este pur RISC sau pur CISC, granița dintre cele 2 arhitecturi devenind din ce în ce mai subțire.

Proiectul poate fi concluzionat prin a spune că obiectivele propuse inițial au fost atinse, arhitectura aleasă fiind complet transpusă în Verilog, iar microprocesorul se apropie foarte mult de o arhitectură de tip RISC.

Ca remarci referitoare la lucrurile pe care le-am aprofundat în timpul realizării acestei teze, pot afirma că implementarea unei astfel de mașini pas cu pas m-a ajutat să înțeleg foarte bine cum se întâmplă la acest nivel de detaliere, de la modul în care toate blocurile funcționale trebuie să comunice între ele și să se sincronizeze corect, până la diferitele variante de optimizare a execuției în timp printr-o tehnică pipeline, ce la rândul ei implică gestionarea corectă a instrucțiunilor de salt.

## Contribuții personale

În cadrul proiectului, am contribuit cu următoarele:

- Alegerea arhitecturii și a setului de instrucțiuni ce urmau a fi implementate în Verilog
- Scrierea întregului cod sursă pentru toate modulele prezentate în lucrare, pas ce a implicat alegerea în prealabil a semnalelor de control ale UCS și gândirea sincronizărilor între blocuri pentru fiecare instrucțiune în parte
- Verificarea individuală a tuturor modulelor realizate și rezolvarea problemelor apărute
- Crearea programelor de test și testarea microprocesorului la nivel de top, soluționarea problemelor de sincronizare întâmpinate (depanarea) și obținerea unei execuții corecte în final
- Realizarea tuturor schemelor bloc și diagramelor prezentate în lucrare

## Dezvoltări ulterioare

Microprocesorul are o arhitectură simplă cu o structură minimală, însă i se pot adăuga un număr mare de atribute în plus. Printre cele pe care mi-ar face plăcere să le dezvolt, ar fi:

- Implementarea modului de adresare indirectă prin registru pentru salturi
- Reducerea folosirii instrucțiunii NOP în cazurile particulare prezentate anterior
- Optimizarea pipeline-ului astfel încât să existe o suprapunere perfectă între etapele de fetch și decode-execute
- Mărirea setului de registre generale și a stivei hardware, pentru a necesita și mai putin acces în memoria de date
- Optimizarea algoritmului de împărțire binară

De asemenea, am în gând și adăugarea unui periferic de tip PWM realizat pe perioada practicii de vară din anul trei, cu o funcționare relativ complexă, dar flexibilă. În plus, aș vrea să introduc posibilitatea de a opera cu date în virgulă mobilă, întrucât am avut ocazia, în semestrul I al anului IV, la disciplina *Arhitectura Sistemelor de Calcul*, să realizez un bloc de preprocesare pentru un bloc de virgulă mobilă, menit să preia operanzii și să ofere mai departe exponenții, mantisele și semnul așteptat al rezultatului.

Proiectul este unul deschis, fără un final bine definit, care anticipez că poate fi dezvoltat sau îmbunătățiț infinit, ajungând poate, la un moment dat, să aibă o funcționalitate apropiată de microprocesoarele simple, de uz general, din ziua de astăzi. Cred că pasul cel din urmă, după șlefuirea tuturor atributelor sistemului, ar fi concretizarea – sinteza lui și implementarea pe o placă FPGA.

# Bibliografie

[1] Burileanu, C., *Curs de Arhitectura Microprocesoarelor*, UPB, București, 2017.
[2] Booth, A. D., "A Signed Binary Multiplication Technique", în *The Quarterly Journal of Mechanics and Applied Mathematics*, IV (1951), pp. 236-240.
[3] *Boolean Multplication and Division*, https://www.techglads.com/cse/sem3/boolean-multiplication-division/, accesat la data: 23.06.2019.
[4] *Booth's multiplication algorithm*, https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm, accesat la data: 23.06.2019.
[5] *Fast multiplication*, http://web.cecs.pdx.edu/~zeshan/ece341_lecture7a.pdf, accesat la data: 23.06.2019.
[6] Maican, S., *Curs de Circuite Integrate Digitale*, UPB, Bucuresti, 2017.
[7] Abd-El-Barr, M., El-Rewini, H., *Fundamentals of Computer Organization and Architecture*, Editura John Wiley & Sons, New Jersey, 2005.
[8] Patterson, D. A., Sequin, C. H., *RISC I: A Reduced Instruction Set VLSI Computer*, în ISCA 1998 - *25 Years of the International Symposia on Computer Architecture*, pp. 216-230.
[9] Grohosky, G. F., *Machine Organization of the IBM RISC System/6000 processor*, în *IBM Journal of Research and Development*, Vol. 34/1990, pp. 37.
[10] Shen, J., Lipasti, M. H., *Modern Processor Design: Fundamentals of Superscalar Processors*, Editura Waveland Press, 2013.
[11] *Introduction to Verilog*, http://www.lsi.upc.edu/~jordicf/Teaching/secretsofhardware/VerilogIntroduction_Nyasulu.pdf, accesat la data: 23.06.2019.
[12] *Art of Writing Testbenches, Part II*, http://www.asic-world.com/verilog/art_testbench_writing2.html, accesat la data: 23.06.2019.
[13] *Laborator de Circuite Integrate Digitale*, https://wiki.dcae.pub.ro/index.php/Circuite_Integrate_Digitale_(laborator), accesat la data: 23.06.2019.
[14] *Cadence Design Systems, NCLaunch User Guide*, http://www.ee.virginia.edu/~mrs8n/soc/SynthesisTutorials/nclaunch.pdf, accesat la data: 23.06.2019.
[15] *Cadence Design Systems, SimVision User Guide*, http://citeseerx.ist.psu.edu/viewdoc/download;jsesssionid=5E37BD77BD0CA5292440487483BABF7F?doi=10.1.1.433.382&rep=rep1&type=pdf, accesat la data: 23.06.2019.

# Anexe

## Anexa 1: Modulul cu definirea constantelor

```
`define OPERAND_SIZE 8
`define OPCODE_SIZE 5
`define INSTRUCTION_SIZE 32
`define MAX_SIGNED_POS 127
`define MAX_SIGNED_NEG -128

`define SECTION_0 //programul mare cu toate tipurile de instrucţiuni
//`define SECTION_1 //pentru porturi
//`define SECTION_2 //cu PCIS

//registrul de instrucţiune are 32 biţi; cei mai semnificativi 5 reprezintă codul instructiunii
//instrucţiuni de transfer de date
`define LD 5'd0
`define LDI 5'd1
`define MOV 5'd2
`define POP 5'd3
`define PUSH 5'd4
`define ST 5'd5

//instrucţiuni de prelucrări de date
`define ADC 5'd6
`define ADD 5'd7
`define AND 5'd8
`define ASR 5'd9
`define CP 5'd10
`define DEC 5'd11
`define DIV 5'd12
`define INC 5'd13
`define MUL 5'd14
`define OR 5'd15
`define SBC 5'd16
`define SHL 5'd17
`define SHR 5'd18
`define SUB 5'd19
`define XOR 5'd20

//instrucţiuni de control al programului
`define BREQ 5'd21
`define BRNE 5'd22
`define CALL 5'd23
`define JMP 5'd24
`define NOP 5'd25
`define PCIS 5'd26
`define RET 5'd27
`define RST 5'd28
`define SLEEP 5'd29

`define R0 4'd0
`define R1 4'd1
`define R2 4'd2
`define R3 4'd3
`define R4 4'd4
`define R5 4'd5
`define R6 4'd6
`define R7 4'd7
`define R8 4'd8
`define R9 4'd9
`define R10 4'd10
`define R11 4'd11
`define R12 4'd12
`define R13 4'd13
`define R14 4'd14
`define R15 4'd15
```

# Anexa 2: Setul de registre, numărătorul de program și stiva hardware

```
`include "isa.v"

module reg_pc_data (
   input rpd_rst, //reset extern
         rpd_rst_instr, //reset de la
instrucțiunea RST
         rpd_int_pc,

         //ceasuri
         rpd_reg_clk,
         rpd_pc_clk,
         q3,
         q4,

         //semnale control PC
         rpd_asleep,
         rpd_jam,
         rpd_anticipate_jbc,
         rpd_anticipate_ret,

         //semnale control stivă
         rpd_call,
         rpd_ret,
         rpd_push,
         rpd_pop,

         //semnale control registre
         rpd_mov,
         rpd_reg_read,
         rpd_reg_write,

   //adresa PC
   input [3:0] rpd_reg_ispc,

   //valoare imediată pentru salt ime-
diat relativ sau adresă registru pen-
tru salt implicit în registru
   input  signed  [`OPERAND_SIZE-1:0]
rpd_imm_jbc,

   //adresa registrului în care se
scrie/din care se citește; aici se vor
lega 4 biți LSB ai ctrl_adr_out
   input [3:0] rpd_reg_adr,


//magistrala de date
 input   signed   [`OPERAND_SIZE-1:0]
rpd_data_in,

output   signed   [`OPERAND_SIZE-1:0]
rpd_data_out,

   //magistrala de adrese pentru me-
moria de program
   output         [`OPERAND_SIZE-1:0]
rpd_pc_value
   );

//declarații registre
reg     signed     [`OPERAND_SIZE-1:0]
reg_file [0:15];
reg  signed  [`OPERAND_SIZE-1:0]  hid-
den_temp; //pentru MOV
wire signed [`OPERAND_SIZE-1:0] hid-
den_temp_value; //pentru MOV
wire    signed    [`OPERAND_SIZE-1:0]
rpd_reg_1;
wire    signed    [`OPERAND_SIZE-1:0]
rpd_reg_2;
wire    signed    [`OPERAND_SIZE-1:0]
rpd_reg_3;
integer i;

//declarații PC
wire [`OPERAND_SIZE-1:0] next_pc_int;
//pentru  intreruperile  hardware  ex-
terne
wire   [`OPERAND_SIZE-1:0]   next_pc;
//PENTRU ASLEEP/WAKE
wire    [`OPERAND_SIZE-1:0]    pc_jbc;
//pentru JMP, CALL, BREQ si BRNE
wire    [`OPERAND_SIZE-1:0]    pc_ret;
//pentru RET

//declarații stivă
reg     signed     [`OPERAND_SIZE-1:0]
hardware_stack [0:7];
wire    signed    [`OPERAND_SIZE-1:0]
stack_value;
reg signed [2:0] stack_pointer;
wire  signed  [2:0]  stack_pointer_va-
lue;
wire stack_modify_up;
wire stack_modify_down;
reg     signed     [`OPERAND_SIZE-1:0]
save_jbc_pc;

//comportament registre, și deci și PC
//scrierea în registre (sincronă)
always @ (posedge rpd_reg_clk or po-
sedge rpd_rst) begin
   if (rpd_rst) begin
      for (i=0; i<16; i=i+1)
      reg_file[i] <= 8'd0;
   end
   else begin
      if       (rpd_reg_adr       !==
rpd_reg_ispc)
         reg_file[rpd_reg_adr]     <=
rpd_reg_1;
      if   (q4   &&   rpd_rst_instr)
reg_file[rpd_reg_ispc] <= 0;
      else          if          (q4)
reg_file[rpd_reg_ispc]            <=
next_pc_int;
      else  reg_file[rpd_reg_ispc]  <=
reg_file[rpd_reg_ispc];
```

```
   end
end

assign rpd_reg_1 = (rpd_mov && rpd_reg_write) ? hidden_temp : rpd_reg_2 ; //dacă am instrucţiunea MOV, destinaţia va lua valoarea registrului ascuns

assign rpd_reg_2 = (rpd_pop && rpd_reg_write) ? hardware_stack[stack_pointer] : rpd_reg_3; //dacă am instrucţiunea POP, destinaţia va lua valoarea din vârful stivei hardware

assign rpd_reg_3 = (rpd_reg_write) ? rpd_data_in : reg_file[rpd_reg_adr]; //dacă am alte instrucţiuni de scriere şi registrul nu e PC, pot scrie în el datele venite pe magistrala de date

//instrucţiunea MOV
always @ (posedge rpd_reg_clk or posedge rpd_rst) begin
   if (rpd_rst) hidden_temp <= 8'd0;
   else hidden_temp <= hidden_temp_value;
end

assign hidden_temp_value = (rpd_mov && rpd_reg_read) ? reg_file[rpd_reg_adr] : hidden_temp;

//citirea din registre (asincronă)
assign rpd_data_out = rpd_reg_read ? reg_file[rpd_reg_adr] : 8'bz;

assign next_pc_int = rpd_int_pc ? 8'd248 : next_pc;

assign next_pc = rpd_asleep ? reg_file[rpd_reg_ispc] : pc_jbc; //pc stă pe loc dacă am instr. SLEEP

assign pc_jbc = (rpd_anticipate_jbc && !rpd_jam) ? (reg_file[rpd_reg_ispc] + rpd_imm_jbc) : ((rpd_anticipate_jbc && rpd_jam) ? reg_file[rpd_imm_jbc] : pc_ret); //salt relativ la PC

assign pc_ret = rpd_anticipate_ret ? hardware_stack[stack_pointer] : reg_file[rpd_reg_ispc] + 4; //la RET ia valoarea din vârful stivei, altfel creşte cu 4

//trimiterea valorii lui spre memoria de program
assign rpd_pc_value = reg_file[rpd_reg_ispc];

//stiva
//scrierea în stivă
always @ (posedge rpd_pc_clk or posedge rpd_rst) begin
   if (rpd_rst) begin
      for (i=0; i<8; i=i+1) //stack is initialized with 0 at reset
      hardware_stack[i] <= 8'd0;
   end
   else begin
      hardware_stack[stack_pointer] <= stack_value;
   end
end

always @(posedge q3 or posedge rpd_rst) begin
   if (rpd_rst) save_jbc_pc <= 8'd0;
   else if (rpd_anticipate_jbc || rpd_int_pc) save_jbc_pc <= reg_file[rpd_reg_ispc] + 4;
   else save_jbc_pc <= save_jbc_pc;
end

assign stack_value = (rpd_push) ? reg_file[rpd_reg_adr] : ((rpd_pop) ? 8'd0 : ((rpd_call || rpd_int_pc) ? save_jbc_pc : (rpd_ret ? 8'd0 : hardware_stack[stack_pointer])));

//stack_pointer - arată spre ultima locaţie scrisă în stivă (începe de jos de la -1, la prima scriere devine 0 etc.)
always @ (posedge rpd_reg_clk or posedge rpd_rst) begin
   if (rpd_rst) stack_pointer <= -3'd1;
   else stack_pointer <= stack_pointer_value;
end

//se modifică pe PUSH, POP, CALL şi RET
assign stack_modify_up = (rpd_push || rpd_call || rpd_int_pc) && q3;
assign stack_modify_down = (rpd_pop || rpd_ret) && q4;

assign stack_pointer_value = stack_modify_up ? stack_pointer + 1 : (stack_modify_down ? stack_pointer - 1 : stack_pointer);

endmodule
```

# Anexa 3: Unitatea aritmetico-logică

```
`include "isa.v"

module alu (
   input reg_clk,
   input          [`OPERAND_SIZE-1:0] alu_data_in,  //alu doar calculează pentru numerele binare; interpretarea lor ca numere cu semn sau fără semn se face în registre/în alte părți
   input  [`OPCODE_SIZE-1:0]  alu_opcode, //instrucțiunea de executat
   input  [1:0]  alu_operand,  //îmi spune care dintre cei 2 operanzi a ajuns la alu
   input [1:0] alu_res,
   input alu_carry_in,

   output         [`OPERAND_SIZE-1:0] alu_data_out,
   output reg alu_overflow,
   output reg alu_negative,
   output reg alu_carry_out,
   output reg alu_zero,
   output reg alu_parity
);

reg [`OPERAND_SIZE-1:0] alu_op_1;
reg [`OPERAND_SIZE-1:0] alu_op_2;

integer i;
reg [3:0] ones; //pentru parity
reg alu_carry;
reg [`OPERAND_SIZE-1:0] alu_result;

//punerea operanzilor în registre
always @(posedge reg_clk) begin
   case (alu_opcode)
      `ADD, `ADC, `SUB, `SBC, `INC, `DEC, `OR, `AND, `XOR, `CP:
         begin
            if (alu_operand == 2'b01) alu_op_1 <= alu_data_in;
            else if (alu_operand == 2'b10) alu_op_2 <= alu_data_in;
            else begin
               alu_op_1 <= 8'd0;
               alu_op_2 <= 8'd0;
            end
         end
      `SHR, `SHL, `ASR:
         begin
            if (alu_operand == 2'b01) alu_op_1 <= alu_data_in;
            else alu_op_1 <= 8'd0;
         end
      default:
         begin
            alu_op_1 <= 8'd0;
            alu_op_2 <= 8'd0;
         end
   endcase
end

//punerea rezultatului pe magistrala de date
assign alu_data_out = ((alu_opcode == `ADD || alu_opcode == `ADC || alu_opcode == `SUB || alu_opcode == `SBC || alu_opcode == `INC || alu_opcode == `DEC || alu_opcode == `OR || alu_opcode == `AND || alu_opcode == `XOR || alu_opcode == `SHR || alu_opcode == `SHL || alu_opcode == `ASR) && (alu_res == 2'b01)) ? alu_result : 8'bz;

//efectuarea operației propriu-zise
always @(*) begin
   case (alu_opcode)
      `ADD: {alu_carry, alu_result} = alu_op_1 + alu_op_2;
      `ADC: {alu_carry, alu_result} = alu_op_1 + alu_op_2 + alu_carry_in;
      `SUB: {alu_carry, alu_result} = {1'b0, alu_op_1} - {1'b0, alu_op_2};
      `SBC: {alu_carry, alu_result} = {1'b0, alu_op_1} - {1'b0, alu_op_2} - alu_carry_in;
      `INC: {alu_carry, alu_result} = alu_op_1 + 1;
      `DEC: {alu_carry, alu_result} = {1'b0, alu_op_1} - 1;
      `OR:   alu_result = alu_op_1 | alu_op_2;
      `AND:  alu_result = alu_op_1 & alu_op_2;
      `XOR:  alu_result = alu_op_1 ^ alu_op_2;
      `SHR:  alu_result = alu_op_1 >> 1;
      `SHL:  alu_result = alu_op_1 << 1;
      `ASR:  alu_result = alu_op_1 >>> 1;
      `CP:
         begin
            alu_carry = (alu_op_1 < alu_op_2);
            alu_result = alu_op_1 - alu_op_2;
         end
      default: alu_result = 8'bz;
   endcase
end

//setarea fanioanelor
always @(*) begin
   //am overflow când adun 2 nr. negative și obtin un nr. pozitiv
   //sau 2 nr. pozitive și obțin un nr. negativ
```

```
   alu_overflow  =  ((alu_opcode  ==
`ADD)     &&     ((~alu_op_1[7]     &
~alu_op_2[`OPERAND_SIZE-1]  &  alu_re-
sult[`OPERAND_SIZE-1])              |
(alu_op_1[`OPERAND_SIZE-1]          &
alu_op_2[`OPERAND_SIZE-1]  &  ~alu_re-
sult[`OPERAND_SIZE-1]))) ||
   //incrementez peste valoarea max.
pozitivă reprezentabilă pe 8 biţi în
C2
   (alu_opcode == `INC && alu_op_1 ==
`MAX_SIGNED_POS) ||
   //decrementez sub valoarea min. ne-
gativă reprezentabilă pe 8 biţi în C2
   (alu_opcode == `DEC && alu_op_1 ==
`MAX_SIGNED_NEG) ||
   //rezultatul înmulţirii cu 2 are
semn diferit de deînmulţit
   (alu_opcode     ==     `SHL     &&
(alu_op_1[`OPERAND_SIZE-1] != alu_re-
sult [`OPERAND_SIZE-1]));

   //1 pt. numere negative, 0 pt. nu-
mere pozitive
   alu_negative  =  alu_result[`OPE-
RAND_SIZE-1];

   //parity e 1 pentru număr par de
biţi de 1 în rezultat şi 0 altfel
   ones = 0;
   for (i=0; i<8; i=i+1)
      ones = ones + alu_result[i];
   alu_parity = ~ones[0];

   alu_carry_out = alu_carry;

   //setat când rezultatul unei scă-
deri este 0 (operanzi egali)
   alu_zero = ((alu_opcode == `CP ||
alu_opcode == `SUB) && alu_result ==
0) ? 1 : 0;
end

endmodule
```

# Anexa 4: Înmulțitorul și împărțitorul cablate

```
`include "isa.v"

module mul_div(
   input reg_clk,
   input  signed  [`OPERAND_SIZE-1:0]
mul_div_data_in,
   input [1:0] mul_div_operand,
   input [1:0] mul_div_result,
   input           [`OPCODE_SIZE-1:0]
mul_div_opcode,

   output  signed  [`OPERAND_SIZE-1:0]
mul_div_data_out,
   output reg mul_div_zd
);

//registrele pentru operanzi
reg     signed     [`OPERAND_SIZE-1:0]
mul_div_op_1;
reg     signed     [`OPERAND_SIZE-1:0]
mul_div_op_2;

//rezultate finale
reg signed [15:0] product;
reg signed [`OPERAND_SIZE-1:0] quoti-
ent;
reg signed [`OPERAND_SIZE-1:0] rema-
inder;

//pentru înmulțire
reg signed [16:0] mul_a;
reg signed [16:0] mul_s;
reg signed [16:0] product_alg;
//pentru înmulțire, în cazul în care
deînmulțitul este -128
reg signed [17:0] mul_a_min;
reg signed [17:0] mul_s_min;
reg signed [17:0] product_alg_min;

//pentru împărțire
reg signed [`OPERAND_SIZE-1:0] div_a;
reg signed [`OPERAND_SIZE-1:0] div_q;
reg signed [`OPERAND_SIZE-1:0] div_m;
reg sign;
reg sub;

integer i;

//punerea operanzilor în cele 2 regis-
tre
always @(posedge reg_clk) begin
   case (mul_div_opcode)
      `MUL, `DIV:
         begin
            if   (mul_div_operand   ==
2'b01)         mul_div_op_1         <=
mul_div_data_in;
            else  if  (mul_div_operand
==     2'b10)     mul_div_op_2     <=
mul_div_data_in;
         end
      default:
         begin
            mul_div_op_1 <= 8'd0;
            mul_div_op_2 <= 8'd0;
         end
   endcase
end

//doar pe q4 și q1, în rest mag. date
are altceva pe ea
//punerea rezultatelor pe magistrala
de date
assign       mul_div_data_out       =
(mul_div_opcode == `MUL) ?
((mul_div_result  ==  2'b01)  ?  pro-
duct[15:8]   :   ((mul_div_result   ==
2'b10) ? product[7:0] : 8'bz)) :
((mul_div_opcode    ==    `DIV)    ?
((mul_div_result == 2'b01)? quotient :
((mul_div_result == 2'b10)? remainder
: 8'bz)) : 8'bz);

//algoritmii efectivi de înmulțire și
împărțire
always @(*) begin
   case (mul_div_opcode)
      `MUL: begin //înmulțire

         if    (    mul_div_op_1    ===
`MAX_SIGNED_NEG) begin

            mul_a_min     =     {1'b1,
mul_div_op_1, 9'b0};
            mul_s_min   =   {1'b0,   -
mul_div_op_1, 9'b0};
            product_alg_min  =  {9'b0,
mul_div_op_2, 1'b0};

            for (i=0; i<8; i=i+1) be-
gin

               case             (pro-
duct_alg_min[1:0])
                  2'b01:          pro-
duct_alg_min   =   product_alg_min   +
mul_a_min;
                  2'b10:          pro-
duct_alg_min   =   product_alg_min   +
mul_s_min;

                  default:        pro-
duct_alg_min = product_alg_min;
               endcase

            product_alg_min   =   pro-
duct_alg_min >>> 1;

            end
```

```
            product      =      pro-
duct_alg_min[16:1];

         end

         else begin
            mul_a   =   {mul_div_op_1,
9'b0};
            mul_s   =   {-mul_div_op_1,
9'b0};
            product_alg    =    {8'b0,
mul_div_op_2, 1'b0};

            for (i=0; i<8; i=i+1) be-
gin

               case             (pro-
duct_alg[1:0])
                  2'b01:  product_alg
= product_alg + mul_a;
                  2'b10:  product_alg
= product_alg + mul_s;

                  default:       pro-
duct_alg = product_alg;
               endcase

            product_alg = product_alg
>>> 1;

            end

            product      =      pro-
duct_alg[16:1];

         end
      end

      `DIV: begin //împărțire

         div_a = 0;
         div_q = (mul_div_op_1 < 0) ?
-mul_div_op_1 : mul_div_op_1;
         div_m = (mul_div_op_2 < 0) ?
-mul_div_op_2 : mul_div_op_2;

         for (i=0; i<8; i=i+1) begin
            {div_a, div_q} = {div_a,
div_q} << 1;

            div_a = div_a - div_m;

            div_q[0] = (div_a < 0) ?
0 : 1;
            div_a  =  (div_a  <  0)  ?
div_a + div_m : div_a;
         end

         quotient  =  (mul_div_op_1[7]
!= mul_div_op_2[7]) ? -div_q : div_q;
         remainder  =  (mul_div_op_1  <
0) ? -div_a : div_a;
         //câtul e negativ dacă ope-
ranzii au semne diferite
         //restul are același semn ca
deîmpărțitul
      end

      default: begin
      product = 8'b0;
      quotient = 8'b0;
      remainder = 8'b0;
      end

   endcase

end

//fanionul de împărțire la 0
always @ (*) begin
   mul_div_zd  =  (mul_div_opcode  ==
`DIV) && (mul_div_op_2 == 0);
end

endmodule
```

# Anexa 5: UCS, RI și registrul de stare

```
//UCS trimite semnalele de control și sincronizare în microprocesor
//modulul include registrul de stare și registrul de instrucțiune RI

`include "isa.v"

module control_unit (

   input ctrl_rst, //reset extern, nu de la instrucțiune

   input ctrl_int, //întrerupere externă

   input ctrl_reg_clk, ctrl_pc_clk, q1, q2, q3, q4,

   input [`OPERAND_SIZE-1:0] ctrl_instr_in, //magistrala de date pentru instrucțiuni

   //fanioane pentru registrul de stare
   input ctrl_carry_in,
   input ctrl_overflow,
   input ctrl_zero,
   input ctrl_negative,
   input ctrl_parity,
   input ctrl_zd, //fanionul de împărțire la 0

   inout signed [`OPERAND_SIZE-1:0] ctrl_data,//magistrala date pentru date

   output reg ctrl_int_pc,
   //semnale necesare pentru diferite instrucțiuni (rst pentru a reseta PC-ul la RST, carry_out pentru ADC și SBC, asleep pentru a ține PC-ul pe loc, mov pentru a salva într-un registru ascuns o valoare)
   output reg ctrl_rst_instr, // de la instrucțiunea RST; menit doar să reseteze PC-ul
   output reg ctrl_carry_out, //pentru ADC și SBC
   output reg ctrl_asleep, //pentru a ține PC-ul pe loc
   output reg ctrl_mov, //pentru a memora o valoare temporară într-un registru ascuns
   output reg ctrl_call,
   output reg ctrl_ret, //pentru a pune ultima valoare scrisă în stiva în PC
   output reg ctrl_push, //stack_write signal - pentru a pune valori în stivă
   output reg ctrl_pop, //stack_read signal - a scoate valori din stivă și a le pune în registre

   //semnale necesare pentru comportamentul corect al lui PC (după o instrucțiune de salt (din cele 5 de mai jos) să sară direct acolo, ca să nu fie necesar NOP)
   output ctrl_anticipate_jbc, //semnalul care anticipează JMP, BREQ, BRNE și CALL (pe perioada de ceas anterioară punerii lor în RI)
   output reg ctrl_anticipate_ret, //semnalul care anticipează RET (pe perioada de ceas anterioară punerii ei în RI)
   output signed [`OPERAND_SIZE-1:0] ctrl_imm_jbc, //val. imediată pentru salturile cu adresare relativă sau adresa registrului în care am adresa de salt dacă e adresare implicită în registru
   output ctrl_jam, //mod de adresare pentru salt (0 - relativă imediată sau 1 – implicită în registru)
   output [3:0] ctrl_reg_ispc, //cei 4 biți din reg. de stare care îmi spun cine este PC-ul

   //semnale pentru registre
   output reg ctrl_reg_read,
   output reg ctrl_reg_write,

   //semnale pentru memoria de date
   output reg ctrl_mem_read,
   output reg ctrl_mem_write,

   //ieșire magistrala de adrese pentru date
   output reg [`OPERAND_SIZE-1:0] ctrl_adr_out,

   //semnale control pentru alu/mul_div
   output reg [1:0] ctrl_alm_operand, //01 pentru primul operand, 10 pentru al doilea operand
   output reg [1:0] ctrl_alm_res, //01 pentru dest1, 10 pentru dest2
   output reg [`OPCODE_SIZE-1:0] ctrl_alm_opcode //instrucțiunea de executat
```

```
);

//formatul instrucțiunii
wire [`OPCODE_SIZE-1:0] instruction;
wire [3:0] src1;
wire [3:0] src2;
wire [3:0] dest1;
wire [3:0] dest2;
wire signed [`OPERAND_SIZE-1:0] immediate;
wire [2:0] addressing_mode;

//registrul de instrucțiune
reg [`INSTRUCTION_SIZE-1:0] instruction_register;
reg [`INSTRUCTION_SIZE-1:0] ctrl_instr;

//registrul ascuns pentru LD și ST
reg signed [`OPERAND_SIZE-1:0] ls_temp;

//semnale anticipative salturi
reg check_zero; //pentru BREQ, BRNE (se activeaza pe SUB sau CP)
reg branch_anticipation; //pentru BREQ, BRNE
reg jc_anticipation_value; //pentru JMP, CALL
wire send_jump_address;

//registrul de stare
reg [15:0] status_register; //registrul de stare
reg pc_change; //activat când avem instrucțiunea PCIS
wire [4:0] status_value;
reg [3:0] status_pc;
wire modify_status; //bit care îmi spune dacă am o instrucțiune care modifică fanioanele

//registrul de stare
always @ (negedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) pc_change <= 0;
   else if (q3 && ctrl_instr[31:27] == `PCIS) pc_change <= 1;
   else pc_change <= 0;
end

always @(negedge q1 or posedge ctrl_rst) begin
   if (ctrl_rst) ctrl_int_pc <= 0;
   else ctrl_int_pc <= status_register[6];
end

always @ (*) begin
   status_register[6] = ctrl_int;
end //aici se setează combinațional, dar spre PC trimit abia pe neg q1, deci există riscul să se mai execute încă o instrucțiune dacă întreruperea vine după neg q1

always @(negedge q3 or posedge ctrl_rst) begin
   if (ctrl_rst) {status_register[10:7], status_register[4:0]} <= 9'b0; //external reset
   else if (modify_status) {status_register[10:7], status_register[4:0]} <= {status_pc, status_value};
   else {status_register[10:7], status_register[4:0]} <= {status_pc, status_register[4:0]};
end
//pe neg q3 e necesar ca să îmi poată lua carry-ul în caz că instr. următoare e ADC sau SBC
//pe neg q4 e necesar pentru că operația DIV îmi dă ctrl_zd mai tarziu și trebuie să fie vazut aici
always @(negedge q4 or posedge ctrl_rst) begin
   if (ctrl_rst) status_register[5] <= 0;
   else if (instruction == `DIV) status_register[5] <= ctrl_zd;
end

always @(posedge ctrl_reg_clk)
ctrl_carry_out <= status_register[0];

assign modify_status = (instruction == `ADD || instruction == `ADC || instruction == `SUB || instruction == `SBC || instruction == `INC || instruction == `DEC || instruction == `OR || instruction == `AND || instruction == `XOR || instruction == `SHL || instruction == `SHR || instruction == `ASR || instruction == `CP) ? 1 : 0;

assign status_value = {ctrl_parity, ctrl_negative, ctrl_zero, ctrl_overflow, ctrl_carry_in};

always @ (negedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) status_pc <= 4'd15;
   else if (q4 && pc_change) status_pc <= ctrl_instr[23:20];
   else status_pc <= status_pc;
end

assign ctrl_reg_ispc = status_pc;


//registrul de instrucțiune
always @ (negedge ctrl_reg_clk or posedge ctrl_rst) begin
   if (ctrl_rst) ctrl_instr <= 32'b0;
```

```
   else if (q1) ctrl_instr[31:24] <=
ctrl_instr_in; //instrucțiunea vine
de la MSB la LSB
   else if (q2) ctrl_instr[23:16] <=
ctrl_instr_in;
   else if (q3) ctrl_instr[15:8] <=
ctrl_instr_in;
   else      ctrl_instr[7:0]      <=
ctrl_instr_in;
end

//RI tine PC-8 (ca sunt din 4 in 4,
instrucțiunea având 32 de biți și o
locație de memorie de program doar 8
biți)
always @ (posedge ctrl_pc_clk or po-
sedge ctrl_rst) begin
   if (ctrl_rst) instruction_register
<= 32'b0;
   else if (!ctrl_asleep) instruc-
tion_register <= ctrl_instr;
   else instruction_register <= in-
struction_register;
end

//formatul instrucțiunii
assign instruction = instruction_re-
gister[31:27];
assign addressing_mode = instruc-
tion_register[26:24]; //pentru dez-
voltari ulterioare
assign dest1 = instruction_regis-
ter[23:20];
assign dest2 = instruction_regis-
ter[19:16];
assign src1 = instruction_regis-
ter[15:12];
assign src2 = instruction_regis-
ter[11:8];
assign immediate = instruction_regis-
ter[7:0];

always @ (posedge ctrl_reg_clk or po-
sedge ctrl_rst) begin //ca să nu mai
stau să îl scriu la fiecare din cele
15 instrucțiuni care îl folosesc
   if (ctrl_rst) ctrl_alm_opcode <=
5'd0;
   else ctrl_alm_opcode <= instruc-
tion;
end

//semnale anticipative salturi
//pentru BREQ, BRNE
always @(negedge ctrl_reg_clk or po-
sedge ctrl_rst) begin
   if (ctrl_rst) branch_anticipation
<= 0;
   else     if     (check_zero     &&
((ctrl_instr[31:27] == `BREQ && sta-
tus_register[2]     ==     1)     ||
(ctrl_instr[31:27]   ==   `BRNE   &&
status_register[2] == 0))) branch_an-
ticipation <= 1;
   else branch_anticipation <= 0;
end

//pentru JMP, CALL
always @(posedge ctrl_reg_clk or po-
sedge ctrl_rst) begin
   if (ctrl_rst) jc_anticipation_va-
lue <= 0;
   else if (ctrl_instr[31:27] == `JMP
|| ctrl_instr[31:27] == `CALL) jc_an-
ticipation_value <= 1;
   else jc_anticipation_value <= 0;
end

//pentru RET
always @ (posedge ctrl_reg_clk or po-
sedge ctrl_rst) begin
   if (ctrl_rst) ctrl_anticipate_ret
<= 0;
   else if (ctrl_instr[31:27] == `RET)
ctrl_anticipate_ret <= 1;
   else ctrl_anticipate_ret <= 0;
end

assign ctrl_anticipate_jbc = jc_anti-
cipation_value   ||   branch_anticipa-
tion;

//ctrl_instr[24] e 0 pentru adresare
relativă imediată și 1 pentru adresare
implicită în registru
assign ctrl_imm_jbc = (send_jump_ad-
dress && !ctrl_jam) ? ctrl_instr[7:0]
: ((send_jump_address && ctrl_jam) ?
{4'b0, ctrl_instr[15:12]} : 8'bz) ;

assign send_jump_address = (ctrl_an-
ticipate_jbc || ctrl_instr[31:27] ==
`BREQ || ctrl_instr[31:27] == `BRNE)
&& q4;

assign ctrl_jam = ctrl_instr[24];

//semnale de control pentru fiecare
instrucțiune
always @ (posedge ctrl_reg_clk or po-
sedge ctrl_rst) begin
   if (ctrl_rst) begin
      ctrl_rst_instr <= 0;
      ctrl_asleep <= 0; //și resetul
extern îl scoate din sleep
      ctrl_mov <= 0;
      ctrl_call <= 0;
      ctrl_ret <= 0;
      ctrl_push <= 0;
      ctrl_pop <= 0;
      ctrl_reg_read <= 0;
      ctrl_reg_write <= 0;
      ctrl_mem_read <= 0;
      ctrl_mem_write <= 0;
      ctrl_adr_out <= 8'bz;
```

```
      ctrl_alm_operand <= 2'b0;
      ctrl_alm_res <= 2'b0;
   end
   else begin
      case (instruction)
         `NOP:
           begin
                ctrl_reg_write   <=
0;
                ctrl_adr_out     <=
8'bz; //toate celelalte semnale de
control care sunt 0 sau x la NOP se
fac 0 singure când e cazul
           end
         `SLEEP:
           begin
              ctrl_reg_write <= 0;
              ctrl_asleep <= 1;
              ctrl_adr_out <= 8'bz;
           end
         `PCIS:
                ctrl_reg_write   <=
0;
         `LD:
           begin
              if (q1) begin
                 ctrl_reg_write   <=
0;
                 ctrl_adr_out     <=
{4'b0, src1}; //pun adresa (Rb) în
care am adresa din memorie
                 ctrl_reg_read <= 1;
              end
              else if (q2) begin
                 ctrl_adr_out     <=
ctrl_data;
                 //îmi vine adresa
de la care trebuie să citesc în memo-
rie
                 ctrl_reg_read <= 0;
                 if (ctrl_data ==
8'd1 || ctrl_data == 8'd5 || ctrl_data
== 8'd9 || ctrl_data == 8'd13)
ctrl_mem_read <= 0; //nu pot citi din
registrul de configurare al disp. de
intrare/ieșire
                 else ctrl_mem_read
<= 1;
              end
              else if (q3) begin
              //mi-au venit datele
pe magistrala de date
                 ls_temp          <=
ctrl_data;
                 ctrl_adr_out     <=
{4'b0, dest1}; //pun adresa în care
tb. scrise (Ra)
                 ctrl_reg_write   <=
1;
                 ctrl_mem_read <= 0;
              end
              else begin
                 ctrl_reg_write = 0;
              end
           end
         `LDI:
           begin
              if (q1) begin
                 ctrl_adr_out     <=
{4'b0, dest1};
                 ctrl_reg_write   <=
1;
              end
              else if (q2 || q3 ||
q4) begin
                 ctrl_reg_write   <=
0;
              end
           end
         `ST:
           begin
              if (q1) begin
                 ctrl_reg_write   <=
0;
                 ctrl_adr_out     <=
{4'b0, src1}; //citesc din Rb val.
care tb. scrisă în mem
                 ctrl_reg_read <= 1;
              end
              else if (q2) begin
                 ls_temp          <=
ctrl_data; //salvez valoarea in ls
temp
                 ctrl_adr_out     <=
{4'b0, dest1}; //pun adresa (Ra) unde
găsesc adresa din memorie
              end
              else if (q3) begin
                 ctrl_adr_out     <=
ctrl_data; //îmi vine adresa pe mag.
de date și o pun pe mag. de adrese
                 ctrl_reg_read <= 0;
                 if (ctrl_data ==
8'd0 || ctrl_data == 8'd4 || ctrl_data
== 8'd8 || ctrl_data == 8'd12)
ctrl_mem_write <= 0; //nu pot scrie în
registrul de stare al disp. de in-
trare/ieșire
                 else ctrl_mem_write
<= 1;
              end
              else begin
                 ls_temp <= 8'd0;
                 ctrl_mem_write   <=
0;
              end
           end
         `PUSH:
           begin
              if (q1 || q2 || q3) be-
gin
                 ctrl_reg_write   <=
0;
                 ctrl_push <= 1;
                 ctrl_adr_out     <=
{4'b0, src1};
```

```
                end
                else begin
                   ctrl_push <= 0;
                end
             end
          `POP:
             begin
                if (q1) begin
                   ctrl_pop <= 1;
                   ctrl_adr_out     <=
{4'b0, dest1};
                   ctrl_reg_write   <=
1;
                end
                else if (q2 || q3) be-
gin
                   ctrl_reg_write   <=
0;
                end
                else begin
                   ctrl_pop <= 0;
                end
             end
          `ADD, `ADC, `SBC, `OR, `AND,
`XOR:
             begin
                if (q1) begin
                   ctrl_reg_write   <=
0;
                   ctrl_alm_res     <=
2'b0;
                   ctrl_adr_out     <=
{4'b0, src1};
                   ctrl_reg_read <= 1;
                   ctrl_alm_operand <=
2'b01;
                end
                else if (q2) begin
                   ctrl_adr_out     <=
{4'b0, src2};
                   ctrl_alm_operand <=
2'b10;
                end
                else if (q3) begin
                   ctrl_alm_operand <=
2'b0;
                   ctrl_adr_out     <=
{4'b0, dest1};
                   ctrl_reg_read <= 0;
                   ctrl_reg_write   <=
1;
                   ctrl_alm_res     <=
2'b01;
                end
                else begin
                   ctrl_reg_write   <=
0;
                   ctrl_alm_res     <=
2'b0;
                end
             end
          `SUB:
             begin
                if (q1) begin
                   ctrl_reg_write   <=
0;
                   ctrl_alm_res     <=
2'b00;
                   ctrl_adr_out     <=
{4'b0, src1};
                   ctrl_reg_read <= 1;
                   ctrl_alm_operand <=
2'b01;
                end
                else if (q2) begin
                   ctrl_adr_out     <=
{4'b0, src2};
                   ctrl_alm_operand <=
2'b10;
                end
                else if (q3) begin
                   ctrl_alm_operand <=
2'b0;
                   ctrl_adr_out     <=
{4'b0, dest1};
                   ctrl_reg_read <= 0;
                   ctrl_reg_write   <=
1;
                   check_zero <= 1;
                   ctrl_alm_res     <=
2'b01;
                end
                else begin
                   ctrl_reg_write   <=
0;
                   ctrl_alm_res     <=
2'b00;
                end
             end
          `MUL, `DIV:
             begin
                if (q1) begin
                   ctrl_reg_write   <=
0;
                   ctrl_alm_res     <=
2'b00;
                   ctrl_alm_operand <=
2'b01;
                   ctrl_adr_out     <=
{4'b0, src1};
                   ctrl_reg_read <= 1;
                end
                else if (q2) begin
                   ctrl_adr_out     <=
{4'b0, src2};
                   ctrl_alm_operand <=
2'b10;
                end
                else if (q3) begin
                   ctrl_alm_operand <=
2'b0;
                   ctrl_adr_out     <=
{4'b0, dest1};
                   ctrl_reg_read <= 0;
                   ctrl_reg_write   <=
1;
```

```
                ctrl_alm_res     <=
2'b01; //aici leg câtul (sau partea
superioara din produs) la data out
mul_div
             end
             else begin

                ctrl_adr_out     <=
{4'b0, dest2};
                ctrl_alm_res     <=
2'b10; //aici leg restul (sau partea
inferioară din produs) la data out
mul_div
             end
          end
          //MUL și DIV nu mai apucă
să facă reg_write 0, de-aia trebuie
făcut pe primul ceas al fiecărei in-
strucțiuni
       `INC, `DEC, `SHR, `SHL, `ASR:
          begin
             if (q1 || q2) begin
                ctrl_reg_write   <=
0;
                ctrl_alm_res     <=
2'b00;
                ctrl_alm_operand <=
2'b01;
                ctrl_adr_out     <=
{4'b0, dest1};
                ctrl_reg_read <= 1;
             end
             else if (q3) begin
                ctrl_alm_operand <=
2'b0;
                ctrl_reg_read <= 0;
                ctrl_reg_write   <=
1;
                ctrl_alm_res     <=
2'b01;
             end
             else begin
                ctrl_reg_write   <=
0;
                ctrl_alm_res     <=
2'b00;
             end
          end
       `CP:
          begin
             if (q1) begin
                ctrl_reg_write   <=
0;
                ctrl_alm_operand <=
2'b01;
                ctrl_adr_out     <=
{4'b0, src1};
                ctrl_reg_read <= 1;
            ctrl_alm_res = 2'b00;
             end
             else if (q2) begin
                ctrl_alm_operand <=
2'b10;
                ctrl_adr_out     <=
{4'b0, src2};
             end
             else if (q3) begin
check_zero <= 1; ctrl_alm_operand <=
2'b0; end
             else begin
                ctrl_reg_read <= 0;
             end
          end
       `MOV:
          begin
             if (q1) begin
                ctrl_reg_write   <=
0;
                ctrl_mov <= 1;
                ctrl_adr_out     <=
{4'b0, src1}; //citesc valoarea pe
care vreau să o scriu altundeva
                ctrl_reg_read <= 1;
             end
             else if (q2) begin
                ctrl_adr_out     <=
{4'b0, dest1};
                ctrl_reg_read <= 0;
                ctrl_reg_write   <=
1;
             end
             else begin
                ctrl_reg_write   <=
0;
                ctrl_mov <= 0;
             end
          end
       `JMP, `BREQ, `BRNE:
          begin
             check_zero <= 0;
          end
       `CALL:
          begin
             if (q1 || q2 || q3) be-
gin
                ctrl_reg_write   <=
0;
                ctrl_call <= 1;
             end
             else
                ctrl_call <= 0;
          end
       `RET:
          begin
             if (q1 || q2 || q3) be-
gin
                ctrl_reg_write   <=
0;
                ctrl_ret <= 1;
             end
             else
                ctrl_ret <= 0;
          end
       `RST:
          begin
```

```
               if (q1 || q2 || q3) be-
gin
                  ctrl_reg_write   <=
0;
                  ctrl_rst_instr   <=
1;
               end
               else
                  ctrl_rst_instr   <=
0;
            end
      endcase
   end
end

assign ctrl_data = (instruction ==
`LDI && q2) ? immediate : (((instruc-
tion == `ST || instruction == `LD) &&
q4) ? ls_temp : 8'bz);

//trimiterea valorii codată imediat
pentru saltul relativ la PC înainte,
astfel încât PC să o aibă pregătită
când trebuie să facă saltul
//pentru BREQ și BRNE trimit oricum,
căci saltul se efectuează doar dacă am
semnalul branch_anticipation activ

endmodule
```

# Anexa 6: Memoria de program

```
`include "isa.v"

module program_memory (
   input q1, q2, q3, q4,
   input          [`OPERAND_SIZE-1:0]
pm_pc_value,

   output   reg   [`OPERAND_SIZE-1:0]
pm_instr
);

//popularea memoriei ROM
wire [`INSTRUCTION_SIZE-1:0] instr_1;
wire [`INSTRUCTION_SIZE-1:0] instr_2;
wire [`INSTRUCTION_SIZE-1:0] instr_3;
wire [`INSTRUCTION_SIZE-1:0] instr_4;
wire [`INSTRUCTION_SIZE-1:0] instr_5;
wire [`INSTRUCTION_SIZE-1:0] instr_6;
wire [`INSTRUCTION_SIZE-1:0] instr_7;
wire [`INSTRUCTION_SIZE-1:0] instr_8;
wire [`INSTRUCTION_SIZE-1:0] instr_9;
wire          [`INSTRUCTION_SIZE-1:0]
instr_10;
wire          [`INSTRUCTION_SIZE-1:0]
instr_11;
wire          [`INSTRUCTION_SIZE-1:0]
instr_12;
wire          [`INSTRUCTION_SIZE-1:0]
instr_13;
wire          [`INSTRUCTION_SIZE-1:0]
instr_14;
wire          [`INSTRUCTION_SIZE-1:0]
instr_15;
wire          [`INSTRUCTION_SIZE-1:0]
instr_16;
wire          [`INSTRUCTION_SIZE-1:0]
instr_17;
wire          [`INSTRUCTION_SIZE-1:0]
instr_17_ind;
wire          [`INSTRUCTION_SIZE-1:0]
instr_18;
wire          [`INSTRUCTION_SIZE-1:0]
instr_19;
wire          [`INSTRUCTION_SIZE-1:0]
instr_20;
wire          [`INSTRUCTION_SIZE-1:0]
instr_21;
wire          [`INSTRUCTION_SIZE-1:0]
instr_22;
wire          [`INSTRUCTION_SIZE-1:0]
instr_int;

wire          [`INSTRUCTION_SIZE-1:0]
instr_default;

`ifdef SECTION_0
   assign instr_1 = {`NOP, 27'bx};
assign instr_2 = {`LDI, 3'bx, `R3,
12'bx, 8'd16}; //încarcă valoarea 16
în registrul R3
   assign instr_3 = {`LDI, 3'bx, `R4,
12'bx, 8'd15}; //încarcă valoarea 10
în registrul R4
   assign instr_4 = {`SUB, 3'bx, `R5,
4'bx, `R3, `R4, 8'bx}; //scrie dife-
rența dintre R3 și R4 în R5
   assign instr_5 = {`BRNE, 3'b0,
16'bx, 8'd16}; //dacă cele 2 nu sunt
egale, salt condiționat la PC + 16
   assign instr_6 = {`LDI, 3'bx, `R6,
12'bx, 8'd3}; //încarcă valoarea 3 în
registrul R6
   assign instr_7 = {`LDI, 3'bx, `R7,
12'bx, 8'd4}; //încarcă valoarea 4 în
registrul R7
   assign instr_8 = {`MUL, 3'bx, `R10,
`R11, `R6, `R7, 8'bx}; //pune R6*R7 în
{R10, R11}
   assign instr_9 = {`DIV, 3'bx, `R1,
`R7, `R11, `R6, 8'bx}; //pune R11 DIV
R6 (4) în R1 și MOD în R7 (0)
   assign instr_10 = {`PUSH, 3'bx,
4'bx, 4'bx, `R1, 4'bx, 8'bx}; //pune
R1 (4) în stiva
   assign instr_11 = {`POP, 3'bx, `R2,
4'bx, 4'bx, 4'bx, 8'bx}; //dă pop în
R2 (pune 4 în R2)
   assign instr_12 = {`ST, 3'bx, `R6,
4'bx, `R4, 4'bx, 8'bx}; //pune în me-
moria de date, la adresa din R6 (3),
valoarea din R4 (15)
   assign instr_13 = {`LD, 3'bx, `R0,
4'bx, `R6, 4'bx, 8'bx}; //în R0 pune
ce găsești în mem. de date la adresa
din R6 (găsești valoarea 15 pusă an-
terior)
   assign instr_14 = {`MOV, 3'bx, `R8,
4'bx, `R0, 4'bx, 8'bx}; //pune în R8
valoarea din R0 (15)
   assign instr_15 = {`LDI, 3'bx, `R9,
12'bx, 8'd255}; //pune în R9 val. po-
zitivă maximă reprezentabilă pe 8
biți: 255; pentru numere fără semn,
ALU tb să dea carry aici
   assign instr_16 = {`INC, 3'bx, `R9,
4'bx, 4'bx, 4'bx, 8'bx}; //incremen-
tează valoarea din R9; mă aștept să am
carry
   assign instr_17 = {`ADC, 3'bx,
`R12, 4'bx, `R8, `R5, 8'bx}; //pune în
R12 suma cu carry dintre R8 (15) și R5
(0); mă aștept să am 16 în R12 căci
carry era 1
   assign instr_17_ind = {`LDI, 3'bx,
`R10, 12'bx, 8'd96}; //pune în R10
```

```
valoarea 96 - testează adresarea inmplicită pentru salturi
   assign instr_18 = {`CALL, 3'b1, 8'bx, `R10, 4'bx, 8'bx}; //sari la adresa din R10
   assign instr_19 = {`LDI, 3'bx, `R13, 12'bx, 8'd13}; //pune în R13 valoarea R13
   assign instr_20 = {`ADD, 3'bx, `R14, 4'bx, `R13, `R6 , 8'bx}; //pune R13(13) + R6 (3) în R14 = 16
   assign instr_21 = {`RET, 27'bx}; //se întoarce la PC = 84
   assign instr_22 = {`MOV, 3'bx, `R9, 4'bx, `R14, 4'bx, 8'bx}; //pune în R8 valoarea din R14 (16)

   assign instr_int = {`LDI, 3'bx, `R0, 12'bx, 8'd1}; //pune în R0 valoarea 1

   assign instr_default = {`SLEEP, 27'b0};

   //trimiterea instrucțiunii în 4 pași, câte 8 biți, de la MSB la LSB
   always @(*) begin
      case (pm_pc_value)
         8'd0:   pm_instr   =   q1   ?   instr_1[31:24] : q2 ? instr_1[23:16] : q3 ? instr_1[15:8] : instr_1[7:0];
       8'd4:   pm_instr   =   q1   ?   instr_2[31:24] : q2 ? instr_2[23:16] : q3 ? instr_2[15:8] : instr_2[7:0];
       8'd8:   pm_instr   =   q1   ?   instr_3[31:24] : q2 ? instr_3[23:16] : q3 ? instr_3[15:8] : instr_3[7:0];
       8'd12:   pm_instr   =   q1   ?   instr_4[31:24] : q2 ? instr_4[23:16] : q3 ? instr_4[15:8] : instr_4[7:0];
       8'd16:   pm_instr   =   q1   ?   instr_5[31:24] : q2 ? instr_5[23:16] : q3 ? instr_5[15:8] : instr_5[7:0];
       8'd32:   pm_instr   =   q1   ?   instr_6[31:24] : q2 ? instr_6[23:16] : q3 ? instr_6[15:8] : instr_6[7:0];
       8'd36:   pm_instr   =   q1   ?   instr_7[31:24] : q2 ? instr_7[23:16] : q3 ? instr_7[15:8] : instr_7[7:0];
       8'd40:   pm_instr   =   q1   ?   instr_8[31:24] : q2 ? instr_8[23:16] : q3 ? instr_8[15:8] : instr_8[7:0];
       8'd44:   pm_instr   =   q1   ?   instr_9[31:24] : q2 ? instr_9[23:16] : q3 ? instr_9[15:8] : instr_9[7:0];
       8'd48:   pm_instr   =   q1   ?   instr_10[31:24] : q2 ? instr_10[23:16] : q3 ? instr_10[15:8] : instr_10[7:0];
       8'd52:   pm_instr   =   q1   ?   instr_11[31:24] : q2 ? instr_11[23:16] : q3 ? instr_11[15:8] : instr_11[7:0];
       8'd56:   pm_instr   =   q1   ?   instr_12[31:24] : q2 ? instr_12[23:16] : q3 ? instr_12[15:8] : instr_12[7:0];
       8'd60:   pm_instr   =   q1   ?   instr_13[31:24] : q2 ? instr_13[23:16] : q3 ? instr_13[15:8] : instr_13[7:0];
       8'd64:   pm_instr   =   q1   ?   instr_14[31:24] : q2 ? instr_14[23:16] : q3 ? instr_14[15:8] : instr_14[7:0];
       8'd68:   pm_instr   =   q1   ?   instr_15[31:24] : q2 ? instr_15[23:16] : q3 ? instr_15[15:8] : instr_15[7:0];
       8'd72:   pm_instr   =   q1   ?   instr_16[31:24] : q2 ? instr_16[23:16] : q3 ? instr_16[15:8] : instr_16[7:0];
       8'd76:   pm_instr   =   q1   ?   instr_17[31:24] : q2 ? instr_17[23:16] : q3 ? instr_17[15:8] : instr_17[7:0];
       8'd80:   pm_instr   =   q1   ?   instr_17_ind[31:24]   :   q2   ?   instr_17_ind[23:16]   :   q3   ?   instr_17_ind[15:8]   :   instr_17_ind[7:0];
       8'd84:   pm_instr   =   q1   ?   instr_18[31:24] : q2 ? instr_18[23:16] : q3 ? instr_18[15:8] : instr_18[7:0];
       8'd88:   pm_instr   =   q1   ?   instr_22[31:24] : q2 ? instr_22[23:16] : q3 ? instr_22[15:8] : instr_22[7:0];
       8'd96:   pm_instr   =   q1   ?   instr_19[31:24] : q2 ? instr_19[23:16] : q3 ? instr_19[15:8] : instr_19[7:0];
       8'd100:   pm_instr   =   q1   ?   instr_20[31:24] : q2 ? instr_20[23:16] : q3 ? instr_20[15:8] : instr_20[7:0];
       8'd104:   pm_instr   =   q1   ?   instr_21[31:24] : q2 ? instr_21[23:16] : q3 ? instr_21[15:8] : instr_21[7:0];


       8'd248:   pm_instr   =   q1   ?   instr_int[31:24]   :   q2   ?   instr_int[23:16]   :   q3   ?   instr_int[15:8]   :   instr_int[7:0];
//rutina de deservire a întreruperii...
       8'd252:   pm_instr   =   q1   ?   instr_21[31:24] : q2 ? instr_21[23:16] : q3 ? instr_21[15:8] : instr_21[7:0];
//RET din ISR

       default:   pm_instr   =   q1   ?   instr_default[31:24] : q2 ? instr_default[23:16]   :   q3   ?   instr_default[15:8] : instr_default[7:0];
      endcase
   end
`endif


`ifdef SECTION_1
   assign instr_1 = {`NOP, 27'bx};
```

```
   assign instr_2 = {`LDI, 3'bx, `R0, 12'bx, 8'd48}; //încarca valoarea 48 în registrul R0
   assign instr_3 = {`LDI, 3'bx, `R1, 12'bx, 8'd40}; //încarcă valoarea 40 în registrul R1
   assign instr_4 = {`SUB, 3'bx, `R2, 4'bx, `R0, `R1, 8'bx}; //scrie diferența dintre R0 și R1 în R2
   assign instr_5 = {`BREQ, 3'b0, 16'bx, 8'd16}; //dacă cele 2 sunt egale, salt condiționat la PC + 16
   assign instr_6 = {`ST, 3'bx, `R2, 4'bx, `R1, 4'bx, 8'bx}; //pune în memorie, la adresa din R2 (8), valoarea din R1 (40) //mă aștept să nu pună căci nu poți scrie în reg. de stare al I/O
   assign instr_7 = {`ST, 3'bx, `R0, 4'bx, `R1, 4'bx, 8'bx}; //pune în memoria de date, la adresa din R0 (48), valoarea din R1 (40) //mă aștept să pună
   /*assign instr_7 = {`RST, 27'bx};*/
   assign instr_8 = {`INC, 3'bx, `R2, 4'bx, 4'bx, 4'bx, 8'bx}; //incrementează valoarea din R2; mă duc pe registrul de configurare pe care mă aștept să nu îl pot citi //adică să nu pună 0 în R1 la instrucțiunea următoare
   assign instr_9 = {`LD, 3'bx, `R1, 4'bx, `R2, 4'bx, 8'bx}; //în R1 pune ce găsești în mem. de date la adresa din R2 (va pune z pentru că nu pot citi registrele de configurare)
   assign instr_10 = {`LD, 3'bx, `R1, 4'bx, `R0, 4'bx, 8'bx}; //în R1 pune ce găsești în mem. de date la adresa din R0 (găsești 40)
   assign instr_11 = {`LDI, 3'bx, `R3, 12'bx, 8'd2}; //pun în R3 2, urmând să accesez memoria la adresa 2, adică registrul de control pentru un disp. de intrare/ieșire, care poate fi și scris, și citit
   assign instr_12 = {`ST, 3'bx, `R3, 4'bx, `R1, 4'bx, 8'bx}; //pun în acesta val. din R1 (40)
   assign instr_13 = {`LD, 3'bx, `R4, 4'bx, `R3, 4'bx, 8'bx}; //pun în R4 ce citesc în registrul de control al primului disp. de intrare/ieșire scris mai sus
   assign instr_default = {`SLEEP, 27'b0};

//trimiterea instrucțiunii în 4 pași, câte 8 biți, de la MSB la LSB
   always @(*) begin
      case (pm_pc_value)
         8'd0:  pm_instr = q1 ? instr_1[31:24] : q2 ? instr_1[23:16] : q3 ? instr_1[15:8] : instr_1[7:0];
         8'd4:  pm_instr = q1 ? instr_2[31:24] : q2 ? instr_2[23:16] : q3 ? instr_2[15:8] : instr_2[7:0];
         8'd8:  pm_instr = q1 ? instr_3[31:24] : q2 ? instr_3[23:16] : q3 ? instr_3[15:8] : instr_3[7:0];
         8'd12: pm_instr = q1 ? instr_4[31:24] : q2 ? instr_4[23:16] : q3 ? instr_4[15:8] : instr_4[7:0];
         8'd16: pm_instr = q1 ? instr_5[31:24] : q2 ? instr_5[23:16] : q3 ? instr_5[15:8] : instr_5[7:0];
         8'd20: pm_instr = q1 ? instr_6[31:24] : q2 ? instr_6[23:16] : q3 ? instr_6[15:8] : instr_6[7:0];
         8'd24: pm_instr = q1 ? instr_7[31:24] : q2 ? instr_7[23:16] : q3 ? instr_7[15:8] : instr_7[7:0];
         8'd28: pm_instr = q1 ? instr_8[31:24] : q2 ? instr_8[23:16] : q3 ? instr_8[15:8] : instr_8[7:0];
         8'd32: pm_instr = q1 ? instr_9[31:24] : q2 ? instr_9[23:16] : q3 ? instr_9[15:8] : instr_9[7:0];
         8'd36: pm_instr = q1 ? instr_10[31:24] : q2 ? instr_10[23:16] : q3 ? instr_10[15:8] : instr_10[7:0];
         8'd40: pm_instr = q1 ? instr_11[31:24] : q2 ? instr_11[23:16] : q3 ? instr_11[15:8] : instr_11[7:0];
         8'd44: pm_instr = q1 ? instr_12[31:24] : q2 ? instr_12[23:16] : q3 ? instr_12[15:8] : instr_12[7:0];
         8'd48: pm_instr = q1 ? instr_13[31:24] : q2 ? instr_13[23:16] : q3 ? instr_13[15:8] : instr_13[7:0];

         default: pm_instr = q1 ? instr_default[31:24] : q2 ? instr_default[23:16] : q3 ? instr_default[15:8] : instr_default[7:0];
      endcase
   end
`endif

`ifdef SECTION_2
   assign instr_1 = {`LDI, 3'bx, `R8, 12'bx, 8'd8}; //pune 8 în R8
   assign instr_2 = {`LDI, 3'bx, `R9, 12'bx, 8'd10}; //pune 8 în R9
   assign instr_3 = {`ADD, 3'bx, `R10, 4'bx, `R8, `R9, 8'bx}; //pune 17 în R10
   assign instr_4 = {`MOV, 3'bx, `R15, 4'bx, `R10, 4'bx, 8'bx};//încearcă să scrii în PC, nu se poate
   assign instr_5 = {`ASR, 3'bx, `R10, 12'bx, 8'bx}; //deplasare arit. la dreapta R10 - devine 9
   assign instr_6 = {`DEC, 3'bx, `R10, 12'bx, 8'bx}; //decrementează R10 - devine 8
```

```
   assign instr_7 = {`MUL, 3'bx, `R11,
`R10, `R10, `R8, 8'bx}; //pune 64 în
R10
   assign instr_8 = {`PCIS, 3'bx,
`R10, 12'bx, 8'bx}; //R10 devine PC,
adresa 64
   assign instr_9 = {`NOP, 27'bx};
//NOP
   assign instr_10 = {`LDI, 3'bx, `R2,
12'bx, 8'd6}; //pune 6 în R2
   assign instr_11 = {`SHL, 3'bx, `R2,
12'bx, 8'bx}; //pune 12 în R2
   assign instr_12 = {`JMP, 3'b1,
8'bx, `R2, 4'bx, 8'bx}; //sari la PC
12

   assign instr_default = {`SLEEP,
27'b0};

//trimiterea instrucțiunii în 4 pași,
câte 8 biți, de la MSB la LSB
   always @(*) begin
      case (pm_pc_value)
         8'd0:  pm_instr  =  q1  ?
instr_1[31:24] : q2 ? instr_1[23:16]
: q3 ? instr_1[15:8] : instr_1[7:0];
         8'd4:  pm_instr  =  q1  ?
instr_2[31:24] : q2 ? instr_2[23:16]
: q3 ? instr_2[15:8] : instr_2[7:0];
         8'd8:  pm_instr  =  q1  ?
instr_3[31:24] : q2 ? instr_3[23:16]
: q3 ? instr_3[15:8] : instr_3[7:0];
         8'd12:  pm_instr  =  q1  ?
instr_4[31:24] : q2 ? instr_4[23:16]
: q3 ? instr_4[15:8] : instr_4[7:0];
         8'd16:  pm_instr  =  q1  ?
instr_5[31:24] : q2 ? instr_5[23:16]
: q3 ? instr_5[15:8] : instr_5[7:0];
         8'd20:  pm_instr  =  q1  ?
instr_6[31:24] : q2 ? instr_6[23:16]
: q3 ? instr_6[15:8] : instr_6[7:0];
         8'd24:  pm_instr  =  q1  ?
instr_7[31:24] : q2 ? instr_7[23:16]
: q3 ? instr_7[15:8] : instr_7[7:0];
         8'd28:  pm_instr  =  q1  ?
instr_9[31:24] : q2 ? instr_9[23:16]
: q3 ? instr_9[15:8] : instr_9[7:0];
         8'd32:  pm_instr  =  q1  ?
instr_8[31:24] : q2 ? instr_8[23:16]
: q3 ? instr_8[15:8] : instr_8[7:0];
         8'd64:  pm_instr  =  q1  ?
instr_9[31:24] : q2 ? instr_9[23:16]
: q3 ? instr_9[15:8] : instr_9[7:0];
         8'd68:  pm_instr  =  q1  ?
instr_10[31:24] : q2 ? instr_10[23:16]
: q3 ? instr_10[15:8] : instr_10[7:0];
         8'd72:  pm_instr  =  q1  ?
instr_11[31:24] : q2 ? instr_11[23:16]
: q3 ? instr_11[15:8] : instr_11[7:0];
         8'd76:  pm_instr  =  q1  ?
instr_9[31:24] : q2 ? instr_9[23:16]
: q3 ? instr_9[15:8] : instr_9[7:0];
         8'd80:  pm_instr  =  q1  ?
instr_12[31:24] : q2 ? instr_12[23:16]
: q3 ? instr_12[15:8] : instr_12[7:0];

         default:  pm_instr  =  q1  ?
instr_default[31:24] : q2 ? instr_de-
fault[23:16]  :  q3  ?  instr_defa-
ult[15:8] : instr_default[7:0];
      endcase
   end
`endif

endmodule
```

# Anexa 7: Memoria de date și porturile

```
`include "isa.v"

module data_memory (
   input dm_reg_clk,
   input [`OPERAND_SIZE-1:0] dm_adr,
   input dm_read,
   input dm_write,
   inout  signed  [`OPERAND_SIZE-1:0]
dm_data
);

reg     signed     [`OPERAND_SIZE-1:0]
data_memory [16:255];
integer i;

//memoria  RAM  pentru  microprocesor,
populare inițială
initial begin
      for (i=16; i<256; i=i+1)
      data_memory[i] <= i;
end

//scriere în memorie (sincronă)
always @ (posedge dm_reg_clk) begin
   if (dm_write && dm_adr > 15)
   data_memory[dm_adr] <= dm_data;
end

//citire din memorie (asincronă)
assign dm_data = (dm_read && dm_adr >
15) ? data_memory[dm_adr] : 8'bz;

endmodule

`include "isa.v"

module io_device1 (
   input io_rst,
   input io_reg_clk,
   input [`OPERAND_SIZE-1:0] io_adr,
   input io_read,
   input io_write,
   inout  signed  [`OPERAND_SIZE-1:0]
io_data
);

reg     signed     [`OPERAND_SIZE-1:0]
io_regs [0:3];
wire valid;
integer i;

assign valid = (io_adr >= 0 && io_adr
<= 3 ) ? 1 : 0;

//scriere  în  registrele  care  pot  fi
scrise din disp. de intrare/ieșire

always @(posedge io_reg_clk or posedge
io_rst) begin

if (io_rst) begin
      for (i=0; i<4; i=i+1)
         io_regs[i] <= 0;
   end
   else if (io_write && valid)
      io_regs[io_adr] <= io_data;
end

//citire din porturi
assign io_data = (io_read && valid) ?
io_regs[io_adr] : 8'bz;

endmodule



`include "isa.v"

module io_device2 (

   input io_rst,

   input io_reg_clk,

   input [`OPERAND_SIZE-1:0] io_adr,

   input io_read,

   input io_write,

   inout  signed  [`OPERAND_SIZE-1:0]
io_data

);



reg     signed     [`OPERAND_SIZE-1:0]
io_regs [4:7];

wire valid;

integer i;

assign valid = (io_adr >= 4 && io_adr
<= 7 ) ? 1 : 0;

//scriere  în  registrele  care  pot  fi
scrise din disp. de intrare/ieșire

always @(posedge io_reg_clk or posedge
io_rst) begin

   if (io_rst) begin

      for (i=4; i<8; i=i+1)
```

```
         io_regs[i] <= 0;
   end
   else if (io_write && valid)
      io_regs[io_adr] <= io_data;
end
//citire din porturi
assign io_data = (io_read && valid) ?
io_regs[io_adr] : 8'bz;
endmodule
`include "isa.v"
module io_device3 (
   input io_rst,
   input io_reg_clk,
   input [`OPERAND_SIZE-1:0] io_adr,
   input io_read,
   input io_write,
   inout  signed  [`OPERAND_SIZE-1:0]
io_data
);
reg    signed    [`OPERAND_SIZE-1:0]
io_regs [8:11];
wire valid;
integer i;
assign valid = (io_adr >= 8 && io_adr
<= 11 ) ? 1 : 0;
//scriere în registrele care pot fi
scrise din disp. de intrare/ieșire
always @(posedge io_reg_clk or posedge
io_rst) begin
   if (io_rst) begin
      for (i=8; i<12; i=i+1)
         io_regs[i] <= 0;
   end
   else if (io_write && valid)
      io_regs[io_adr] <= io_data;
end
//citire din porturi
assign io_data = (io_read && valid) ?
io_regs[io_adr] : 8'bz;
endmodule
`include "isa.v"
module io_device4 (
   input io_rst,
   input io_reg_clk,
   input [`OPERAND_SIZE-1:0] io_adr,
   input io_read,
   input io_write,
   inout  signed  [`OPERAND_SIZE-1:0]
io_data
);
reg    signed    [`OPERAND_SIZE-1:0]
io_regs [12:15];
wire valid;
integer i;
assign valid = (io_adr >= 12 && io_adr
<= 15 ) ? 1 : 0;
//scriere în registrele care pot fi
scrise din disp. de intrare/ieșire
always @(posedge io_reg_clk or posedge
io_rst) begin
   if (io_rst) begin
      for (i=12; i<16; i=i+1)
         io_regs[i] <= 0;
   end
   else if (io_write && valid)
      io_regs[io_adr] <= io_data;
end
//citire din porturi
assign io_data = (io_read && valid) ?
io_regs[io_adr] : 8'bz;
endmodule
```

# Anexa 8: Modulul de generare a ceasurilor

```
module clk_generator (
   input rst,
   output reg reg_clk,
   output reg pc_clk,
   output q1,
   output q2,
   output q3,
   output q4
);

reg [3:0] q;
reg basic_clk;

initial begin
   basic_clk = 0;
   forever #5 basic_clk = ~basic_clk;
end

always @ (posedge basic_clk or negedge basic_clk or posedge rst) begin
if (rst) reg_clk <= 0;
else reg_clk <= basic_clk;
end

always @(posedge basic_clk or posedge rst) begin
if (rst || q2) pc_clk <= 0;
else if (q4) pc_clk <= 1;
end

always @(posedge reg_clk or posedge rst) begin
      if (rst) q <= 4'b0;
      else if (q == 0) q <= 4'b0001;
      else q <= {q[2:0], q[3]};
end

assign q1 = q[0];
assign q2 = q[1];
assign q3 = q[2];
assign q4 = q[3];
endmodule
```

# Anexa 9: Modulul principal al microprocesorului

```
`include "isa.v"

module top (
   input top_external_rst,
   input top_external_int,
   input   top_reg_clk,   top_pc_clk,
top_q1, top_q2, top_q3, top_q4,
   input          [`OPERAND_SIZE-1:0]
top_pm_data, //intrarea instrucțiuni-
lor în microprocesor

   inout          [`OPERAND_SIZE-1:0]
top_data_bus, //magistrala de date
   output         [`OPERAND_SIZE-1:0]
top_adr_bus, //trimiterea unei adrese
spre memorie
   output         [`OPERAND_SIZE-1:0]
top_pm_adr, //trimiterea valorii PC la
memoria de program
   output top_mem_read,
   output top_mem_write
);

wire top_carry_ctrl_to_alu;
wire top_carry_alu_to_ctrl;
wire top_overflow;
wire top_zero;
wire top_negative;
wire top_parity;
wire top_zd;
wire top_rst_instr;
wire top_int_pc;
wire top_asleep;
wire top_mov;
wire top_call;
wire top_ret;
wire top_push;
wire top_pop;
wire top_anticipate_jbc;
wire top_anticipate_ret;
wire [`OPERAND_SIZE-1:0] top_imm_jbc;
wire top_jam;
wire [3:0] top_reg_ispc;
wire top_reg_read;
wire top_reg_write;
wire [1:0] top_alm_operand;
wire [1:0] top_alm_res;
wire  [`OPCODE_SIZE-1:0]  top_alm_op-
code;


control_unit control_unit (
.ctrl_rst(top_external_rst),
.ctrl_int(top_external_int),
.ctrl_reg_clk(top_reg_clk),
.ctrl_pc_clk(top_pc_clk),
.q1(top_q1),
.q2(top_q2),
.q3(top_q3),
.q4(top_q4),
.ctrl_instr_in(top_pm_data),
.ctrl_carry_in(top_carry_alu_to_ctrl)
,
.ctrl_overflow(top_overflow),
.ctrl_zero(top_zero),
.ctrl_negative(top_negative),
.ctrl_parity(top_parity),
.ctrl_zd(top_zd),
.ctrl_data(top_data_bus),
.ctrl_int_pc(top_int_pc),
.ctrl_rst_instr(top_rst_instr),
.ctrl_carry_out(top_carry_ctrl_to_alu
),
.ctrl_asleep(top_asleep),
.ctrl_mov(top_mov),
.ctrl_call(top_call),
.ctrl_ret(top_ret),
.ctrl_push(top_push),
.ctrl_pop(top_pop),
.ctrl_anticipate_jbc(top_antici-
pate_jbc),
.ctrl_anticipate_ret(top_antici-
pate_ret),
.ctrl_imm_jbc(top_imm_jbc),
.ctrl_jam(top_jam),
.ctrl_reg_ispc(top_reg_ispc),
.ctrl_reg_read(top_reg_read),
.ctrl_reg_write(top_reg_write),
.ctrl_mem_read(top_mem_read),
.ctrl_mem_write(top_mem_write),
.ctrl_adr_out(top_adr_bus),
.ctrl_alm_operand(top_alm_operand),
.ctrl_alm_res(top_alm_res),
.ctrl_alm_opcode(top_alm_opcode)
);

reg_pc_data reg_pc_data (
.rpd_rst(top_external_rst),
.rpd_rst_instr(top_rst_instr),
.rpd_int_pc(top_int_pc),
.rpd_reg_clk(top_reg_clk),
.rpd_pc_clk(top_pc_clk),
.q3(top_q3),
.q4(top_q4),
.rpd_asleep(top_asleep),
.rpd_jam(top_jam),
.rpd_anticipate_jbc(top_antici-
pate_jbc),
.rpd_anticipate_ret(top_antici-
pate_ret),
.rpd_call(top_call),
.rpd_ret(top_ret),
.rpd_push(top_push),
.rpd_pop(top_pop),
.rpd_mov(top_mov),
.rpd_reg_read(top_reg_read),
```

```
.rpd_reg_write(top_reg_write),
.rpd_reg_ispc(top_reg_ispc),
.rpd_imm_jbc(top_imm_jbc),
.rpd_reg_adr(top_adr_bus[3:0]),
.rpd_data_in(top_data_bus),
.rpd_data_out(top_data_bus),
.rpd_pc_value(top_pm_adr)
);

alu alu (
.reg_clk(top_reg_clk),
.alu_data_in(top_data_bus),
.alu_opcode(top_alm_opcode),
.alu_operand(top_alm_operand),
.alu_res(top_alm_res),
.alu_carry_in(top_carry_ctrl_to_alu),
.alu_data_out(top_data_bus),
.alu_overflow(top_overflow),
.alu_negative(top_negative),
.alu_carry_out(top_carry_alu_to_ctrl)
,
.alu_zero(top_zero),
.alu_parity(top_parity)
);

mul_div mul_div (
.reg_clk(top_reg_clk),
.mul_div_data_in(top_data_bus),
.mul_div_operand(top_alm_operand),
.mul_div_result(top_alm_res),
.mul_div_opcode(top_alm_opcode),
.mul_div_data_out(top_data_bus),
.mul_div_zd(top_zd)
);

endmodule
```

# Anexa 10: Modulul de test pentru modulul principal al microprocesorului

```
`include "isa.v"

module top_tb ();

reg top_external_rst;
reg top_external_int;

wire reg_clk;
wire pc_clk;
wire q1;
wire q2;
wire q3;
wire q4;

wire [`OPERAND_SIZE-1:0] top_pm_data;
wire              [`OPERAND_SIZE-1:0]
top_data_bus;
wire [`OPERAND_SIZE-1:0] top_adr_bus;
wire [`OPERAND_SIZE-1:0] top_pm_adr;

wire top_mem_read;
wire top_mem_write;

clk_generator clk_generator (
.rst(top_external_rst),
.reg_clk(reg_clk),
.pc_clk(pc_clk),
.q1(q1),
.q2(q2),
.q3(q3),
.q4(q4)
);

top top (
.top_external_rst(top_external_rst),
.top_external_int(top_external_int),
.top_reg_clk(reg_clk),
.top_pc_clk(pc_clk),
.top_q1(q1),
.top_q2(q2),
.top_q3(q3),
.top_q4(q4),
.top_pm_data(top_pm_data),
.top_data_bus(top_data_bus),
.top_adr_bus(top_adr_bus),
.top_pm_adr(top_pm_adr),
.top_mem_read(top_mem_read),
.top_mem_write(top_mem_write)
);

program_memory program_memory (
.q1(q1),
.q2(q2),
.q3(q3),
.q4(q4),
.pm_pc_value(top_pm_adr),
.pm_instr(top_pm_data)
);

data_memory data_memory (
.dm_reg_clk(reg_clk),
.dm_adr(top_adr_bus),
.dm_read(top_mem_read),
.dm_write(top_mem_write),
.dm_data(top_data_bus)
);

io_device1 io_device1 (
.io_rst(top_external_rst),
.io_reg_clk(reg_clk),
.io_adr(top_adr_bus),
.io_read(top_mem_read),
.io_write(top_mem_write),
.io_data(top_data_bus)
);
io_device2 io_device2 (
.io_rst(top_external_rst),
.io_reg_clk(reg_clk),
.io_adr(top_adr_bus),
.io_read(top_mem_read),
.io_write(top_mem_write),
.io_data(top_data_bus)
);
io_device3 io_device3 (
.io_rst(top_external_rst),
.io_reg_clk(reg_clk),
.io_adr(top_adr_bus),
.io_read(top_mem_read),
.io_write(top_mem_write),
.io_data(top_data_bus)
);
io_device4 io_device4 (
.io_rst(top_external_rst),
.io_reg_clk(reg_clk),
.io_adr(top_adr_bus),
.io_read(top_mem_read),
.io_write(top_mem_write),
.io_data(top_data_bus)
);

initial begin
   top_external_rst = 1;
   top_external_int = 0;
   #5 top_external_rst = 0;

   //#110
   //@(negedge q4) top_external_int =
1;
   //@(negedge q2) top_external_int =
0;
end
endmodule
```