ARHITECTURA CALCULATOARELOR 2003/2004
CURSUL 13

5.1.4 Microinstrucțiuni

Pentru a controla calea de date din figura 5.1 avem nevoie de 29 de semnale. Acestea pot fi împărțite în 5 grupe funcționale descrise mai jos:

- 9 semnale pentru controlul scrierii datelor de pe magistrala C în regiştri,
- 9 semnale pentru controlul activării regiştrilor pe magistrala B pentru intrarea ALU,
- 8 semnale pentru controlul funcțiilor ALU şi ale shifter-ului,
- 2 semnale (nefigurate) pentru indicarea citiri/scrierii memoriei prin MAR/MDR,
- 1 semnal (nefigurat) pentru indicarea extragerii din memorie prin PC/MBR

Valorile acestor 29 de semnale de control specifică operațiile pentru un ciclu al căii de date. Un ciclu constă în direcționarea valorilor din regiştri pe magistrala B, propagarea semnalelor prin ALU şi shifter, conducerea lor pe magistrala C şi în final scrierea rezultatelor în registrul sau regiştrii corespunzători. Daca un semnal de citire a memoriei este activat, operația cu memoria începe la sfârşitul ciclului căii de date, după ce MAR a fost încărcat. Datele memeoriei sunt disponibile la fiecare sfârşit de ciclu următor în MBR sau MDR şi pot fi folosite în ciclul de după acesta. Cu alte cuvinte, o citire a memoriei inițiată pe oricare port la sfârşitul ciclului K trimite date care nu pot fi folosite în ciclul K+1 ci numai în ciclul K+2 sau mai târziu.

Presupusul comportament care contrazice varianta intuitivă este explicat de figura diagramei de timp anterioare. Semnalele de control al memoriei nu sunt generate în ciclul 1 al ceasului decât după ce MAR şi PC sunt încărcate la frontul crescator al ceasului, către sfârşitul ciclului 1 de ceas. Vom presupune că memoria pune rezultatele pe magistrala memoriei în timpul unui ciclu deci MBR şi/sau MDR pot fi încărcate la următorul front crescator al ceasului, împreună cu alți regiştri. Cu alte cuvinte, încărcăm MAR la sfârşitul unui ciclu al căii de date şi inițializăm memoria la scurt timp după aceea. Ca urmare, nu ne putem aştepta ca rezultatele unei operațiuni de citire să fie în MDR la începutul următorului ciclu în special dacă lărgimea pulsului ceasului este scurtă. Nu este destul timp dacă memoria mai are nevoie de un ciclu de ceas. Un ciclu al căii de date trebuie să apară între începerea citirii memoriei şi folosirea rezultatului. Binențeles alte operațiuni pot avea loc în timpul ciclului, dar nu cele care au nevoie de cuvântul de memorie.

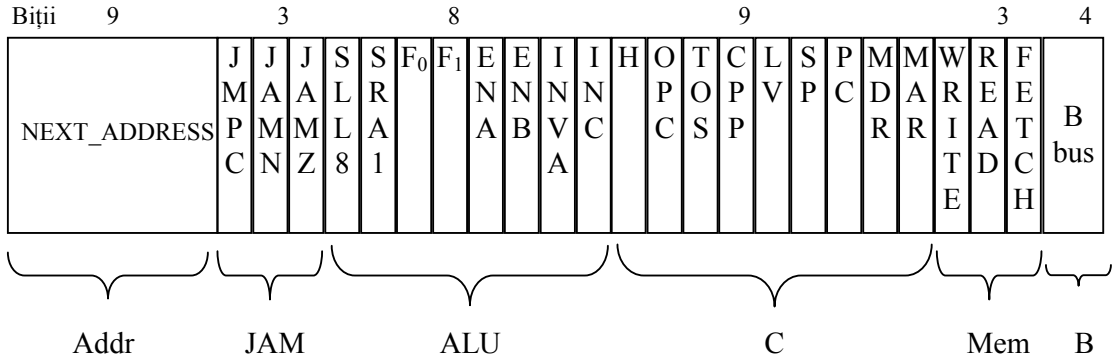
Presupunerea că memoria face un ciclu pentru a opera, este echivalent cu a presupune că rata de operare a cacheului este 100%. Această presupunere nu este adevarată niciodată, dar complexitatea introdusă de un ciclu de memorie cu lungime variabilă este mai mult decât avem nevoie aici. Din vreme ce MBR şi MDR sunt încărcate la frontul crescător al ceasului, odată cu toți ceilalți regiştri, aceştia pot fi citiți în timpul ciclurilor când este făcută o nouă citire a memoriei. Acestea returnează vechile valori din moment ce citirea nu a avut încă timp să le rescrie. Nu există nici o ambiguitate aici: până când noile valori sunt încărcate în MBR şi MDR la frontul crescător al ceasului, valorile anterioare sunt încă utilizabile. Notați că este posibil să se execute citiri "back to back" în două cicluri consecutive din moment ce o citire are nevoie de doar un ciclu. De asemenea, ambele memorii pot opera în acelaşi timp. Totuşi încercând să citim şi să scriem acelaşi octet simultan vom avea rezultate nedefinite.

Dacă se doreşte să se scrie ieşirea de pe magistrală C în mai mult de un registru, nu este de dorit să activăm mai mult de un registru la un moment dat pe magistrala B (practic se ard nişte circuite dacă se încearcă). Cu o mică creştere în circuite, putem reduce numărul de biţi necesari pentru a selecta între posibilele surse pentru conducerea magistralei B. Nu sunt decât 9 regiştrii de intrare posibili care pot conduce magistrala B. De aceea putem coda informaţia din magistrala B pe 4 biţi şi putem folosi un decodor pentru a genera cele 16 semnale de control dintre care de 7 nu avem nevoie. Într-un proiect comercial inginerii vor simţi nevoia copleşitoare de a scapa de unul dintre regiştri pentru ca 3 biţi să facă treaba. Ca profesori ne permitem luxul de a fi în stare să pierdem un bit pentru a prezenta un proiect simplu şi curat.

În acest punct putem controla calea de date cu 9+4+8+2+1=24 de semnale, deci 24 de biţi. Oricum aceşti 24 de biţi controlează doar calea de date pentru un ciclu. A doua parte a controlului are sarcina să determine ce va trebui făcut în ciclu următor. Pentru a include aceasta în proiectul controlerului, se va crea un format pentru descrierea operaţiilor care vor fi efectuate folosind cei 24 de biţi de control plus două câmpuri adiţionale. "NEXT_ADDRESS" şi "JAM". Conţinutul fiecăruia dintre aceste câmpuri îl vom discuta pe scurt.

Figura 5.5 arată un posibil format, împărţit în 6 grupe şi conţinând următoarele 36 de semnale:

- Addr – conţine adresa unei potenţiale următoare microinstrucţiuni,
- JAM – determină cum va fi selectată următoarea microinstrucţiune,
- ALU – funcţiile ALU şi shifter.
- C – selectează care regiştri vor fi scrişi de pe magistrala C,
- Mem – funcţiile memoriei,
- B – selectează sursa magistralei B.



Regiştrii magistralei B

0=MDR	5=LV
1=PC	6=CPP
2=MBR	7=TOS
3=MBRU	8=OPC
4=SP	9-15 none

Figura 5.5 Formatul microinstrucţiunii

Ordonarea grupurilor este în principiu arbitrară deşi defapt s-a ales foarte atent pentru a minimaliza întretăierea între linii într-o figură următoare. Întretăierea liniilor în diagrame schematice, ca în figura în discuţie, corespunde deseori cu întretăierea traseelor în cipuri, care cauzează probleme în proiectele bidimensionale şi e mai bine să fie evitate.

5.1.5 Controlul microistrucțiunii: MIC-1

Până acum am descris cum este controlată calea de date, dar încă nu am descris cum se decide care dintre semnalele de control trebuie activate în fiecare ciclu. Aceasta este determinată de un secvențiator (sequencer) care este responsabil de parcurgerea secvențelor de operații necesare pentru executarea unei singure instrucțiuni ISA.

Secvențiatorul trebuie să producă două tipuri de informații pe fiecare ciclu:

- Starea fiecărui semnal de control în sistem.
- Adresa microinstrucțiunii care urmează să fie executată.

5.1.5.1 Funcția de control a secvențiatorului

Figura 5.6 este o diagramă bloc detaliată a unei microarhitecturi complete a maşinii prezentate ca exemplu, pe care o vom numi Mic-1. Cand veți înțelege pe deplin fiecare căsuță şi fiecare linie în această figură, veți fi pe cale să înțelegeți nivelul microarhitecturii. Diagrama bloc are două părți: calea de date, în stânga, pe care deja am discutat-o în detaliu, şi secțiunea de control, în dreapta la care ne vom concentra acum.

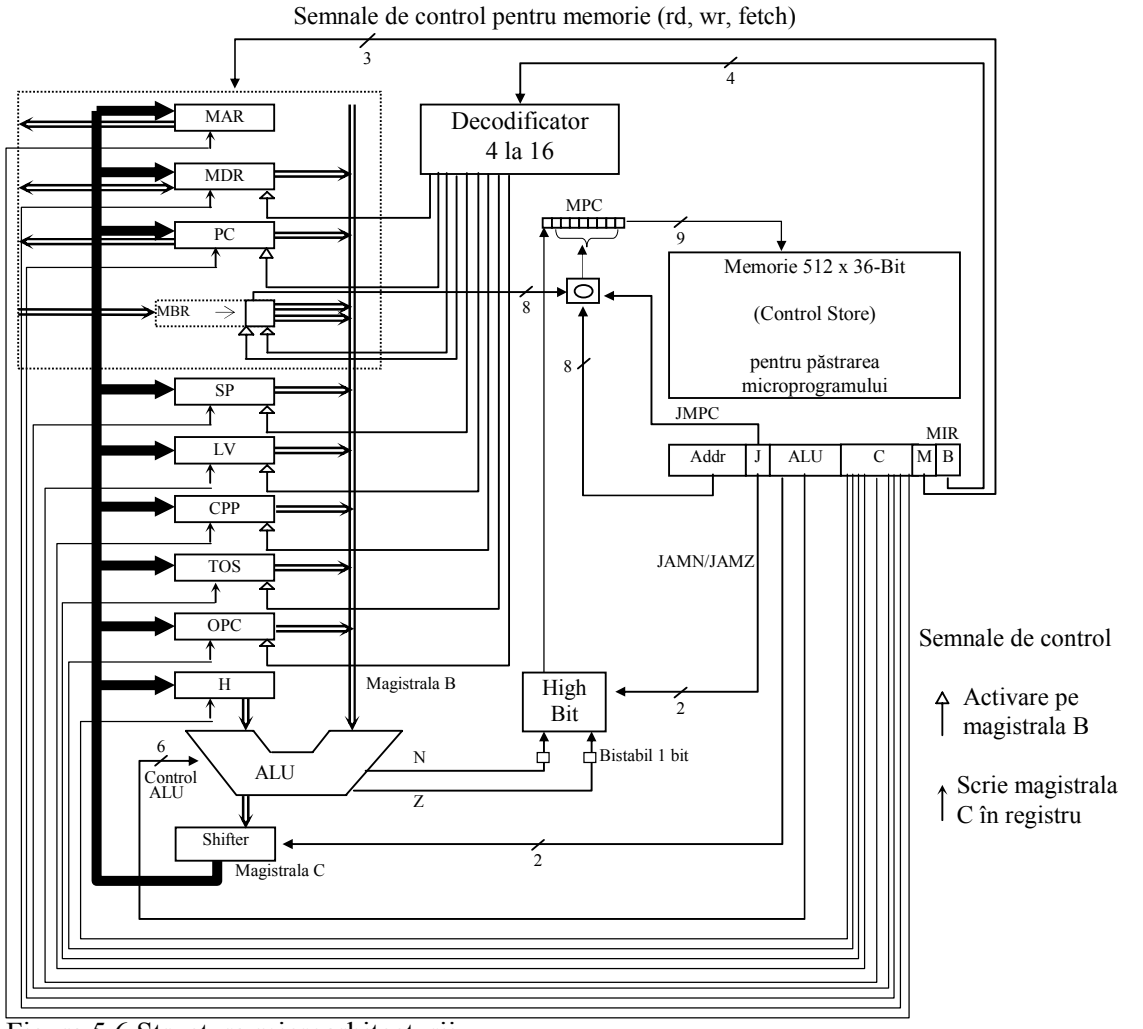


Figura 5.6 Structura microarhitecturii

Cel mai mare şi cel mai important lucru din partea de control a maşinii este memoria numită "control store". Este convenabil să o privim ca o memorie care conţine microprogramul complet, deşi uneori este implementată ca un set de porţi logice. În general, ne vom referi la aceasta cu termenul "control store", pentru a evita confuzia cu memoria principală, accesată prin MBR şi MDR. Oricum funcţional control store este o memorie care pur şi simplu conţine microinstrucţiuni înloc de instrucţiuni ISA. Pentru maşina luată ca exemplu, conţine 512 cuvinte, fiecare cuvânt constînd dintr-o microinstrucţiune de 36 de biţi ilustrată în figura 5.5.

Într-un mod important, "control store" este diferit de memoria principală: instrucţiunile din memoria principală au proprietatea de a fi executate în ordinea adreselor (cu excepţia salturilor), iar microinstrucţiunile **nu**. Incrementarea contorul programului exprimă faptul că instrucţiunea implicită care va fi executată după aceea curentă este instrucţiunea care urmează instrucţiunii curente în memorie. Microprogramele au nevoie de mai multă flexibilitate (pentru că secvenţele de microinstrucţiuni au tendinţa de a fi scurte), deci de obicei nu au această proprietate. În schimb fiecare microinstrucţiune îşi specifică explicit succesorul. Din vreme ce "control store" este funcţional o memorie (read only), are nevoie de propriul registru de adrese de memorie şi de propriul registru de date de memorie. Nu are nevoie de semnale de citire şi de scriere, pentru că este continuu citită. Vom numi registrul de adrese de memorie al "control store" MPC (microprogram counter). Acest nume este ironic din moment ce locatiile din el sunt explicit neordonate, deci conceptul de counting nu este util (dar cine suntem noi să punem sub semnul întrebării tradiţia?). Registrul de date de memorie se numeşte MIR, (microinstruction register). Funcţia sa este să păstreze microinstrucţiunea curentă ai căror biţi conduc (drive) semnalele de control care operează calea de date. Registrul MIR din figura 5.6 conţine aceleaşi 6 grupuri ca în figura 5.5. Grupurile Addr şi J (de la JAM) controlează selecţia următoarei microinstrucţiuni. Grupul ALU contine 8 biţi care selectează funcţia ALU şi conduc shifterul. Biţii C determină regiştriii individuali să încarce ieşirile ALU din magistrala C. Biţii M controlează operaţiile de memorie.

Ultimii 4 biţi conduc decodorul care determină ceea ce merge pe magistrala B. În acest caz s-a ales să se folosească un decodificator standard 4 la16, deşi doar 9 posibilităţi sunt necesare. Într-un proiect mai detaliat poate fi folosit un decodificator 4 la 9. Varianta comercială este să se folosească un circuit standard luat din biblioteca de circuite pentru a nu construi unul nou. Folosind circuitul standard este simplu şi evităm să introducem erori. Folosind propriul cip, cu arie mai mica de utilizare, ia mai mult timp pentru a fi proiectat şi s-ar putea să-l greşiţi.

Modul de funcţionare pentru figura 5.6 este prezentat mai jos. La începutul fiecărui ciclu de ceas (frontul descrescător al ceasului din figura 5.3), MIR este încărcat din cuvântul din "control store" indicat de MPC. Timpul de încărcare MIR este indicat în figura 5.3 de Dw. Dacă judecăm în termeni de subcicluri, MIR este încărcat în timpul primului subciclu. Odată ce microinstrucţiunea este încărcată în MIR, variate semnalele sunt propagate pe calea de date. Un registru este scos pe magistrala B, ALU ştie ce operaţie să execute, şi va fi multă treabă de făcut acolo. Acesta este al doilea subciclu. După un interval Dw+ Dx de la începutul ciclului intrările ALU sunt stabile. Încă cu Dy mai târziu totul a fost aranjat şi ALU, N, Z şi ieşirile shifter-ului sunt stabile. Valorile N şi Z sunt atunci salvate într-o pereche de bistabili pe un bit. Aceşti biţi, ca şi restul de regiştri care sunt încărcaţi de pe magistrala C şi din memorie, sunt salvaţi pe frontul crescător al ceasului aproape de sfărşitul ciclului căii de date. Ieşirea ALU este trasmisă în shifter. Activitatea ALU şi a shifterului au loc în timpul subciclului 3. După un interval adiţional Dz ieşirea shifterului a ajuns în regiştri prin magistrala C. Acum regiştrii pot fi încărcaţi spre sfârşitul ciclului (la frontul crescător al pulsului ceasului).

Subciclul 4 constă în încărcarea regiştrilor şi a bistabilelor N şi Z. Se termină puţin după frontul crescător al ceasului, când toate rezultatele au fost salvate şi rezultatele anterioarei operaţii de memorie sunt disponibile şi MPC a fost încărcat. Acest proces continuă şi continuă până cineva se plictiseşte şi opreşte maşina.

5.1.5.2 Determinarea adresei microinstrucţiunii care urmează să fie executată

În paralel cu conducerea căii de date, microprogramul trebuie să determine care instrucţiune trebuie executată în continuare pentru că nu sunt în ordinea în care apar în memoria de comenzi. Calcularea adresei următoarei microinstrucţiuni începe după ce MIR a fost încărcat şi este stabil. Câmpul de 9 biţi al adresei următoare NEXT_ADDRESS este copiat în MPC. În timp ce copierea are loc JAM este inspectat. Dacă are valoarea 000 nu se întâmplă nimic altceva; când copierea adresei următoare NEXT_ADDRESS este completată, MPC va indica următoarea instrucţiune.

Dacă unul sau mai mulţi dintre biţii JAM sunt 1 atunci este nevoie de mai multă muncă. Dacă JAMN este setat, bistabilul pe 1 bit N este facut OR în bitul de rang superior, High Bit, al MPC. Asemănător dacă JAMZ este setat, atunci este făcut OR bistabilul Z pe 1 bit, in High Bit. Dacă amândouă sunt setate atunci ambele sunt făcute OR acolo. Motivul pentru care N şi Z sunt necesare este că după trecerea frontului crescator al ceasului (când ceasul este High), magistrala B nu mai este condusă şi ieşirile ALU nu mai pot fi considerate corecte. Salvând indicatoarele (flags) ALU în N şi Z, se fac disponibile valorile corecte şi stabile pentru calculul lui MPC, indiferent de ce se întâmplă în ALU.

În figura 5.6, logica care efectuează aceste calcule este etichetată “HIGH BIT”. Funcţia booleana realizata este: F = (JAMZ AND Z) OR (JANM AND N) OR NEXT ADDRESS [8]. În toate cazurile, MPC poate lua numai una dintre cele două valori posibile.

- Valoarea NEXT_ADDRESS.
- Valoarea NEXT_ADDRESS cu HIGH BIT făcut OR cu 1.

Nu există altă posibilitate. Dacă cu bitul high al NEXT_ADDRESS a fost deja facut 1 atunci folosirea lui JAMN sau JAMZ nu are nici un sens.

Când toţi biţi lui JAM sunt 0, adresa următoarei microinstrucţiuni ce va fi executată este pur şi simplu numărul pe 9 biţi din câmpul adresei următoare NEXT_ADDRESS. Când ori JAMN ori JAMZ sunt 1, sunt doi potenţiali succesori: NEXT_ADDRESS şi NEXT_ADDRESS făcut OR cu 0x100 (presupunând că NEXT_ADDRESS ≤ 0xFF). (0x arată că următorul număr este în sistem hexazecimal). Acest punct este ilustrat în figura 5.7. Microinstrucţiunea curentă la locatia 0x75 are NEXT ADDRESS = 0x92 şi JAMZ setat la 1. În consecinţă următoarea adresă a microinstrucţiunii depinde de bitul Z memorat la operaţia ALU anterioară. Dacă bitul Z este 0, următoare microinstrucţiune vine din 0x92. Dacă bitul Z este 1, următoare microinstrucţiune vine din 0x192.

Al treilea bit în câmpul JAM este JMPC. Dacă el este setat, cei 8 biţi MBR sunt făcuţi OR cu cei 8 biţi de pondere inferioară ai NEXT_ADDRESS din microinstrucţiunea curentă. Rezultatul este trimis MPC-ului. Căsuţa cu eticheta “O” din figura structurii generale face OR din MBR cu NEXT_ADDRESS dacă JMPC este 1, dar decât trece NEXT_ADDRESS spre MPC dacă JMPC este 0. Când JMPC este 1, cei 8 biţi de pondere inferioară din NEXT_ADDRESS sunt normal zero. Biţii de pondere superioară pot fi 1 sau 0, deci valoarea NEXT_ADDRESS folosită cu JMPC este în mod normal 0x000 sau 0x100.

Abilitatea de a face OR din MBR împreună cu NEXT_ADDRESS şi de a stoca rezultatul în MPC permite o implementare eficientă a salturilor ramificate multiplu. De notat că pot fi specificate oricare dintre cele 256 de adrese, determinate numai de biţii prezenţi în MBR. Într-o folosire tipică MBR conţine codul de operaţie astfel că folosirea JMPC va determina o selecţie unică pentru următoarea microinstrucţie pentru orice posibil cod de operaţie. Această metodă este folosită pentru ramificarea rapidă direct la funcţia care corespunde codului operaţiei tocmai extras. Înţelegerea secvenţierii maşinii este foarte importantă ceea ce face utilă repetarea ei. O vom face făcând referire la subciclii, pentru ca aceasta să fie uşor de vizualizat, dar singurul ceas real al evenimentelor sunt fronturile descrescătoare care încep ciclul şi fronturile crescătoare care încarcă registrele şi bistabilele N şi Z.

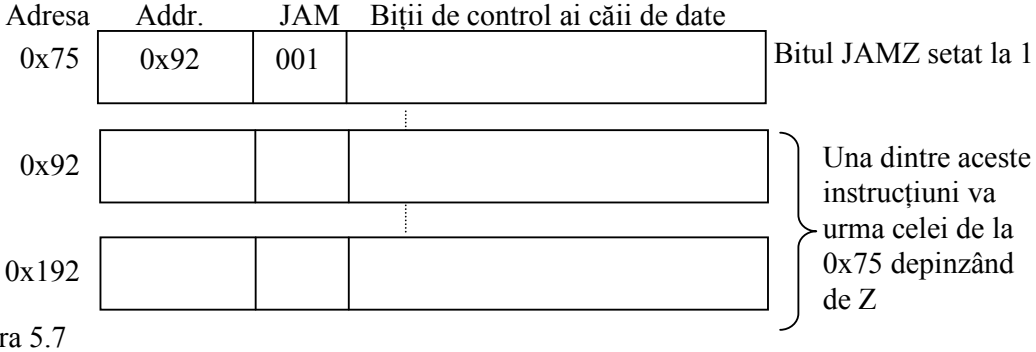


Figura 5.7

În timpul subciclului 1, iniţiat de frontul căzător al ceasului, MIR este încărcat din adresa curentă conţinută în MPC. În timpul subciclului 2, semnalele primite de la MIR sunt emise şi magistrala B este încărcată din registrul selectat. În timpul subciclului 3, ALU şi shifterul operează şi se produce un rezultat stabil. În timpul subciclului 4, magistrala C, magistralele de memorie, şi valorile lui ALU devin stabile. La frontul crescător, registrele sunt încărcate din magistrala C, bistabilele N şi Z sunt încărcate, iar MBR şi MDR primesc rezultatele de la operaţia pornită la sfârşitul ciclului de date anterior (dacă a existat). Imediat ce MBR devine disponibil, MPC este încărcat pentru a pregăti următoarea microinstrucţiune. Astfel MPC işi primeşte valoarea undeva în mijlocul intervalului în care ceasul e high, dar după MBR/MDR sunt gata. Poate fi folosită declanşarea pe nivel (înloc de declanşarea pe front) sau declanşarea la un interval de timp fix faţă de frontul crescător al ceasului. Tot ceea ce contează este ca MPC să nu fie încărcat până ce registrele de care depinde (MBR, N şi Z) nu sunt gata. Imediat ce ceasul cade, MPC poate adresa control store şi un nou ciclu poate începe. Remarcaţi că fiecare ciclu se autoconţine. Acesta specifică ce se duce la magistrala B, ce au de facut ALU şi shifterul, unde se stochează magistrala C şi care poate să fie următoarea valoare a MPC.

O remarcă utila despre structura prezentată. Se trateaza MPC ca un registru propriu-zis cu o capacitate de memorare de 9 biţi care este încărcat în timp ce ceasul este high. În realitate nu este nevoie de a avea un registru. Toate intrările lui pot fi incluse direct în control store. Cât timp aceste intrări sunt prezente în control store la fronturile descrescătoare ale ceasului, când MIR este selectat şi citeşte, înseamnă că este suficient. Nu este nevoie sa se stocheze aceste valori în MPC. Pentru acest motiv, MPC poate fi foarte bine implementat ca un registru virtual, care este doar un loc de plasare a semnalelor, mai mult decât un dispozitiv electronic, decât un registru real. A face MPC un registru virtual simplifică secvenţierea: acum evenimentele se petrec numai la fronturile descrescătoare şi crescătoare ale ceasuluisi nicăieri în altă parte. Dar dacă aceasta este mai uşor de imaginat atunci puteţi imagina MPC ca un registru real, şi aceasta fiind un punct de vedere valid.

5.2 Exemplu ISA: IJVM

Să continuăm exemplele noastre prin introducerea nivelului ISA al maşinii care trebuie interpretat de microprogramul care rulează pe microarhitectura din paragraful precedent. Pentru usurință, ne vom referi uneori la Instruction Set Architecture ca la **macroarhitectură,** în contrast cu microarhitectura.

5.2.1 Stive

Virtual toate limbajele de programare suportă conceptul de procedură care are variabile locale. Aceste variabile pot fi accesate din interiorul procedurii, dar încetează să poată fi accesibile odată ce procedura e terminată. Întrebarea ce trebuie pusă este: "Unde trebuie stocate în memorie aceste variabile?"

Cea mai simplă soluție, să se dea oricărei variabile o adresă absolută de memorie, nu este bună. Problema este că procedura se poate autoapela singură. Pentru moment este suficient să spunem că dacă o procedură este apelată de două ori, este imposibil de memorat variabilele ei în locații de memorie absolute pentru că a doua apelare va interfera cu prima. Alternativ, este folosită o strategie diferită. O zona a memoriei, numită stivă, este rezervată variabilelor, iar variabilele individuale nu conțin adrese absolute. Un registru, să zicem LV, este setat să indice baza stivei pentru variabilele locale pentru procedura curentă.

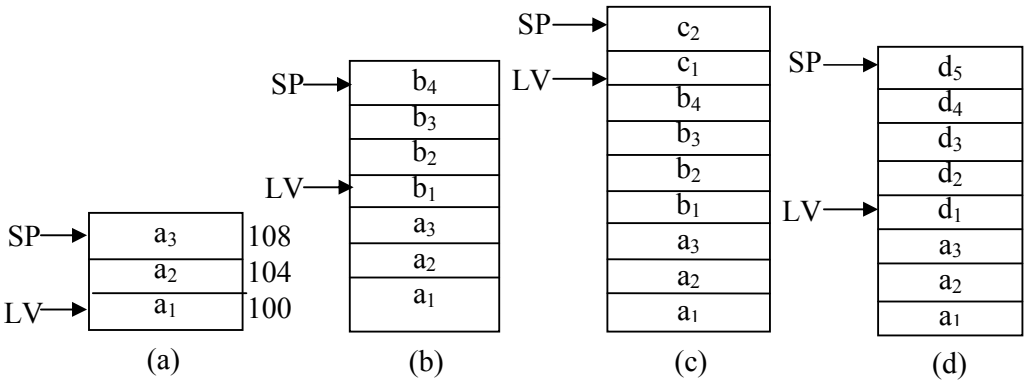


Figura 5.8 Folosirea stivei pentru memorarea variabilelor locale. (a) Când A este activ. (b) După ce A apelează B. (c) După ce B apelează C. (d) După ce C şi B s-au terminat şi A apelează D.

În figura 5.8 (a) de mai sus, procedura A care are ca variabile locale a_1, a_2 şi a_3, a fost apelată deci stocarea variabilele locale a fost făcută în zona de memorie rezervată la adresa indicată de LV. Alt registru, SP, indică cea mai mare adresă de cuvânt folosită de variabilele locale ale procedurii A. Dacă LV este 100 şi cuvintele au 4 octeți, atunci SP va fi 108. Variabilele sunt referite dându-se adresa lor relativă la baza indicată de LV. Structura de date dintre LV şi SP, inclusiv ambele cuvinte indicate de cei doi regiştrii, se numeşte **structura variabilelor locale** a procedurii A.

Acum să considerăm ce s-ar întâmpla dacă A ar apela altă procedură, şi anume B. Unde ar trebui B să memoreze 4 variabile locale (b_1, b_2, b_3, b_4)? Răspuns: în stivă, în continuare după A, cum arată figura 5.8 (b). Notați că LV a fost ajustat de către procedura apelată astfel încât

să indice către variabilele locale ale procedurii B în locul acelora ale procedurii A. Variabilele locale ale proceduri B pot fi referite dânduli-se distanța față de LV. Similar, dacă B apelează C, LV şi SP sunt ajustate din nou pentru a indica spațiul de memorie pentru cele două variabile locale ale lui C, cum este ilustrat în figura 5.8 (c).

Când C se termina (returns) B devine din nou activ, şi stiva este ajustată la loc ca în figura 5.8 (b) deci acum LV indică din nou adresa variabilelor locale ale procedurii B. După ce se termină şi B, LV indică din nou adresa variabilelor locale ale procedurii A, ca în figura 5.8 (a). În orice caz LV indică adresa stivei pentru procedurile active la momentul curent şi SP indică vârful structurii stivei. Acum presupunem că A apelează D care are cinci variabile locale. Avem situația din figura 5.8 (d) în care variabilele locale ale procedurii D folosesc aceeaşi memorie pe care o foloseşte şi procedura B, ca şi procedura C. Cu această organizare a memoriei, memoria este alocată numai pentru procedurile care sunt active curent. Când procedura se termină, memoria folosită de variabilele locale este eliberată.

Stivele au şi o altă folosire, în plus față de păstrarea variabilelor locale. Pot fi folosite pentru păstrarea operanzilor în timpul calculării expresiilor aritmetice. Când o stivă este folosită în acest fel se spune că este o stivă de operanzi.

Presupunând, de exemplu, că înainte de a se apela B, A are de calculat: $a_1 = a_2 + a_3$; O cale de a face această sumă este de a pune (push) a_2 în stivă ca în figura 5.9 (a). Aici SP a fost incrementat cu numărul de octeți din cuvânt, să zicem 4 şi primul operand este memorat la adresa indicată acum de SP. Apoi a_3 este plasat (push) în stivă ca în figura 5.9 (b).Numele variabilelor şi ale procedurilor sunt alese de utilizator, codul operației şi numele registrilor sunt prestabilite.Calculul poate fi executat acum lansând o instrucțiune care extrage (pop) două cuvinte din stivă, le adună şi pune (push) rezultatul în stivă ca în figura 5.9 (c). În final, cuvântul din vârful stivei poate fi extras (pop) şi rememorat la loc în variabila locală a_1 ca în figura 5.9 (d).

Stivele pentru variabile locale şi acelea pentru operanzi pot fi amestecate între ele. De exemplu când calculam expresia $x^2+f(x)$, o parte a expresiei (de exemplu x^2) poate fi stocată în stiva de operanzi când funcția f este apelata. Rezultatul funcției rămâne în stivă, deasupra lui x^2, deci următoarea instructiune poate aduna la această valoare.

În timp ce toate maşinile folosesc stiva pentru memorarea variabilelor locale, nu toate aceste maşini folosesc stivele pentru calcule aritmetice. De fapt cele mai multe dintre ele nu le folosesc, dar IVM şi IJVM funcționează astfel ceea ce justifica prezentarea acestor operații aici.

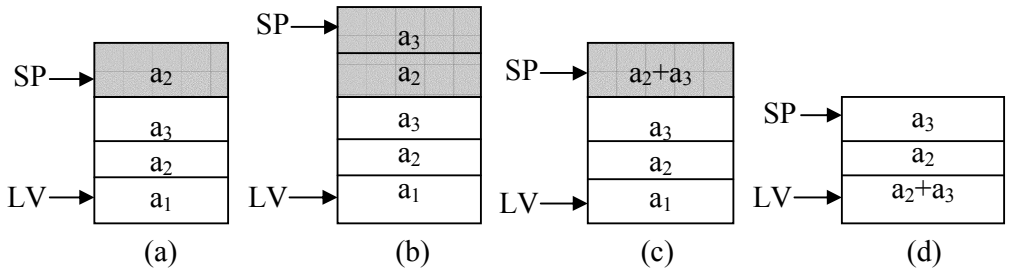


Figura 5.9 Folosirea stivele de operanzi pentru realizarea operațiilor aritmetice.