UNIVERSITATEA TEHNICĂ din CLUJ-NAPOCA
FACULTATEA de AUTOMATICĂ şi CALCULATOARE
CATEDRA de CALCULATOARE

Translatarea limbajelor de descriere a unităților hardware

Referat de doctorat

Conducător ştiinţific,
Prof. Dr. Ing. PUSZTAI Kalman

Doctorand,
ş.l. ing. BARUCH Zoltan

Cuprins

1. Introducere

1.1. Sinteza de nivel înalt

În procesul de sinteză se porneşte de la specificaţia funcţionării unui sistem numeric şi de la un set de restricţii, urmărindu-se obţinerea unei structuri care implementează specificaţia dorită şi satisface restricţiile. La nivelul algoritmic, specificaţia se prezintă sub forma unui algoritm. Aceasta implică faptul că deciziile principale de implementare au fost deja luate, dar, în comparaţie cu nivelul transferurilor între registre, detaliile de implementare trebuie stabilite de sistemul de sinteză automată.

În cadrul ierarhiei de proiectare, descrierea algoritmică specifică funcţionarea sistemului, sub forma operaţiilor şi a secvenţelor de calcule efectuate asupra intrărilor pentru a se obţine anumite ieşiri. Elementele de bază ale unei descrieri algoritmice corespund elementelor principale ale limbajelor de programare şi ale celor de descriere a unităţilor hardware. Indiferent de tipul limbajului utilizat (procedural, funcţional), o descriere algoritmică pune în evidenţă fluxul de date şi de control pentru a specifica funcţionarea unui sistem numeric.

Rezultatul *sintezei de nivel înalt* este o descriere a unui sistem numeric sincron în domeniul structural, la nivelul transferurilor între registre. Sistemul constă dintr-o parte de date, care gestionează datele de intrare pentru a obţine ieşirile cerute, şi o parte de control, care controlează secvenţa şi tipul operaţiilor cu datele. Partea de date şi cea de control comunică prin indicatori de condiţie şi semnale de control. Cele mai multe arhitecturi utilizate pentru implementare constau dintr-o singură parte de date şi un controler centralizat. Implementările tipice la nivelul transferurilor între registre pot fi caracterizate după cum urmează:

- *Partea de date* constă dintr-un set de *unităţi funcţionale*, ca sumatoare, comparatoare, unităţi aritmetice şi logice (UAL), circuite de multiplicare etc., un număr de *elemente de memorare*, ca registre, bistabile sau memorii, şi *circuite de interconectare*, ca magistrale, multiplexoare şi reţele de interconectare.
- *Controlerul* este specificat sub forma unei tabele simbolice de tranziţii între stări, pe baza căreia se realizează sinteza acestuia.
- *Conexiunile de control* validează sau adresează elementele de memorie, comută multiplexoarele sau driverele de magistrală, şi furnizează coduri de operaţie pentru unităţile multifuncţionale.

- *Conexiunile condiționale* furnizează rezultatele testării expresiilor în scopul trecerii la starea următoare şi generării semnalelor de ieşire ale controlerului.

Cea mai simplă implementare a unui sistem sincron este realizată printr-un circuit în care registrele care comută pe front sunt controlate printr-un singur ceas sistem. În acest caz, o etapă de timp corespunde unui ciclu de ceas. Dacă sistemul nu este de tip pipeline, o etapă de timp corespunde şi unei etape de control, deci unei tranziții de la o stare a controlerului la starea următoare.

Arhitecturile care implementează partea de date diferă prin tipul unităților aritmetice şi logice, prin metodele de interconectare, prin tipul unităților de memorie, ca şi prin semnalele de ceas utilizate. Cele mai utilizate arhitecturi se pot grupa în două categorii, care diferă în principal prin metoda de interconectare: arhitectura *multiplexată* şi cea cu *magistrale bidirecționale*.

Pentru *arhitectura multiplexată*, se presupun următoarele:

- Operațiile aritmetice şi logice sunt asociate cu unități funcționale combinaționale, cu excepția elementelor de memorare interne ale unităților funcționale de tip pipeline.
- Memoria este asigurată de registre distribuite.
- Interconexiunile se realizează prin intermediul multiplexoarelor.
- Un singur ceas sistem controlează toate registrele circuitului, pe acelaşi front.

Pentru *arhitectura cu magistrale bidirecționale*, se presupun următoarele:

- Unitățile funcționale au posibilități de memorare. Intrările şi ieşirile sunt bufferate, sau acestea sunt asociate cu registre.
- În locul registrelor distribuite, se utilizează grupuri de registre.
- Interconexiunile se realizează prin magistrale bidirecționale.
- Poate fi necesară utilizarea unui ceas cu două faze.

Translatarea descrierii algoritmice într-o descriere structurală nu este unică. Implementarea poate varia de la soluții secvențiale la soluții complet paralele. Compromisul principal care trebuie rezolvat în cadrul sintezei de nivel înalt este cel între modelarea serială şi cea paralelă, deci între o implementare eficientă din punct de vedere al spațiului, dar lentă, şi una costisitoare din punct de vedere al spațiului, dar rapidă.

Implementările structurale care satisfac specificațiile funcționale sunt considerate puncte în *spațiul de proiectare*. În principiu, spațiul de proiectare este măsurat prin toți parametrii fizici – spațiu, performanțe (întârzieri sau rate de transfer), consum de putere, durata ciclului de ceas, etc. – care sunt relevanți pentru satisfacerea restricțiilor hardware.

Ca şi la proiectarea manuală, şi în cazul sintezei de nivel înalt problema constă în determinarea unei structuri corespunzătoare la nivelul transferurilor între registre, astfel încât aceasta să permită o implementare finală care satisface restricțiile fizice. Satisfacerea sau nu a acestor restricții va fi cunoscută însă numai după terminarea întregului proces de proiectare, deci după sinteza la nivelul RT, sinteza logică şi proiectarea fizică. În consecință, măsurile parametrilor fizici trebuie abstractizate pentru a fi utilizate de componentele sistemului de sinteză, care, pe de altă parte, trebuie să interacționeze cu module de estimare pentru a obține o implementare care satisface restricțiile cerute.

1.2. Etapele sintezei de nivel înalt

Există mai multe etape care trebuie parcurse pentru transformarea unei descrieri algoritmice într-o structură corespunzătoare la nivelul transferurilor între registre. În **Figura 1** se prezintă etapele sintezei de nivel înalt.

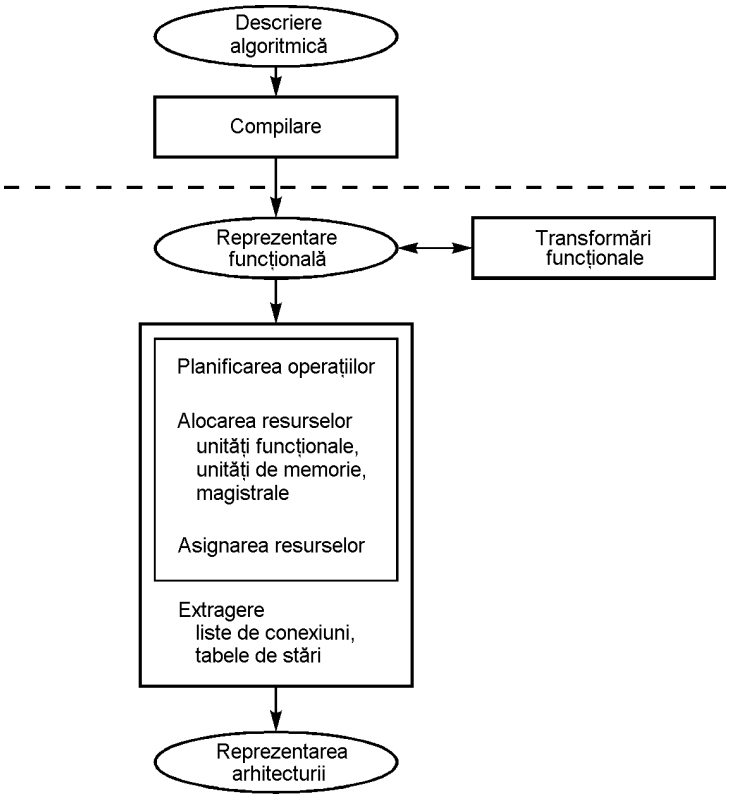


Figura 1. Etapele sintezei de nivel înalt.

Prima etapă este cea de obținere a unei *reprezentări interne* bazată pe grafuri, echivalentă cu decrierea algoritmică, atât pentru fluxul de date cât şi pentru fluxul de control. Fluxul de date se bazează pe operațiile efectuate şi dependențele de date ale acestora, iar fluxul de control provine din construcțiile de control ale descrierii algoritmice. Generarea reprezentării interne este combinată de obicei cu tehnici de optimizare a codului preluate de la compilatoarele convenționale, în scopul eliminării ineficiențelor din descrierea algoritmică. Prelucrarea descrierii algoritmice, generarea reprezentării interne şi optimizările efectuate de compilator reprezintă partea "front end" a procesului de sinteză. Reprezentarea internă a fluxului de date şi de control constituie punctul de început al sintezei structurii la nivelul transferurilor între registre.

Înaintea transformării fluxului de date şi de control într-o structură, pot fi efectuate operații de transformare a reprezentării interne, numite *transformări funcționale*. Pre-

supunând că funcționarea este definită prin fluxul de date şi de control specificat, fiecare transformare care modifică fluxul de date sau de control, modifică funcționarea sistemului numeric. Un exemplu de modificare a fluxului de date este aplicarea proprietății de asociativitate a expresiilor aritmetice. Dintr-un punct de vedere, nu este recomandată aplicarea transformărilor care nu păstrează fluxul de date şi de control specificat. Totuşi, o anumită decriere algoritmică poate fi doar una dintr-o clasă de descrieri algoritmice care satisfac interacțiunea dorită a sistemului numeric cu exteriorul. De aceea, transformările grafurilor care modifică fluxul de date sau de control sunt utile pentru a investiga variantele care rezultă dintr-o descriere algoritmică generică.

Transformarea fluxului de date şi de control într-o structură la nivelul transferurilor între registre constă din următoarele etape principale: *planificarea*, *alocarea resurselor* şi *asignarea resurselor*.

- *Planificarea* reprezintă asignarea operațiilor pentru diferitele intervale de timp, în funcție de anumite restricții, astfel încât să se minimizeze o anumită funcție obiectivă.
- *Alocarea resurselor* constă în determinarea tipului şi a numărului de resurse necesare, deci a unităților funcționale, a elementelor de memorie şi a magistralelor. În unele cazuri, se face distincția între determinarea tipului resurselor (*selecția*) şi determinarea numărului acestora (*alocarea*).
- *Asignarea resurselor* este asignarea la diferite instanțieri ale resurselor, de exemplu a operațiilor la instanțieri ale unităților funcționale, a valorilor care trebuie memorate la instanțieri ale elementelor de memorie, şi a transferurilor de date la instanțieri ale magistralelor.

Relația strânsă care există între resursele alocate şi planificare conduce la două probleme de bază ale planificării. Dacă numărul de resurse – cel mai probabil unitățile funcționale – este fix, şi scopul este de a minimiza numărul intervalelor de timp necesare, trebuie soluționată *planificarea cu restricții de resurse*. Dacă numărul intervalelor de timp este specificat şi scopul este de a minimiza resursele necesare, trebuie executată *planificarea cu restricții de timp*.

Deoarece planificarea, alocarea resurselor şi asignarea resurselor sunt interdependente, modulele de sinteză corespunzătoare acestor etape nu sunt independente, şi nu există o anumită ordine prestabilită în care trebuie efectuate aceste operații. Totuşi, există anumite relații de precedență, de exemplu o operație nu poate fi asignată unei unități funcționale fără alocarea acestei unități funcționale. Soluționarea tuturor problemelor într-un mod optim nu este, în general, posibilă. Au fost dezvoltate diferite strategii pentru a elimina dependențele între diferitele etape, şi metode euristice care permit rezultate apropiate de cele optime pentru etape separate.

Înaintea efectuării operației de planificare, se poate executa *partiționarea* sistemului, care constă în asignarea operațiilor unor grupuri de unități funcționale şi alocarea acestor grupuri pe baza distanței existente între componentele grupurilor. Un grup de unități funcționale este o cale de date multifuncțională construită de jos în sus, care fie există în bibliotecă, fie trebuie compusă din mai multe unități funcționale existente în bibliotecă. Scopul acestei etape de partiționare este de a se direcționa procesul de sinteză

într-o etapă inițială, pe baza estimării spațiului şi a timpului obținute în urma unei etape suplimentare de amplasare, în care se utilizează informații topologice bazate pe interconexiunile între grupuri. Utilizarea informațiilor topologice în această etapă inițială corespunde executării operațiilor de alocare a unităților funcționale şi de asignare a acestora ca etape anterioare a sintezei.

După planificare, alocarea resurselor şi asignarea resurselor, sunt disponibile toate informațiile necesare pentru generarea structurii la nivelul transferurilor între registre. Fluxul de date este implementat sub forma căii de date a sistemului. După generarea acesteia, semnalele de condiție şi de control pot fi codificate, obținându-se o specificație finală a controlerului sub forma unui tabel simbolic de tranziții a stărilor. În acest moment sinteza de nivel înalt este terminată, lista de conexiuni şi tabela de tranziții a stărilor reprezentând implementarea dorită la nivelul transferurilor între registre.

2. Reprezentări interne şi transformări

2.1 Introducere

Deoarece între modelele semantice ale limbajelor de descriere hardware şi arhitecturile utilizate pentru sinteză pot exista diferenţe importante, este necesară o *reprezentare canonică intermediară* care facilitează implementarea descrierilor hardware prin diferite arhitecturi, utilizând diferite sisteme de sinteză. O asemenea reprezentare canonică intermediară trebuie să păstreze nemodificată specificaţia funcţională originală a limbajului utilizat ca intrare, şi să permită adăugarea rezultatelor sintezei pe parcursul diferitelor etape ale acesteia. Deoarece rezultatele sintezei de nivel înalt constau dintr-un set de componente la nivelul RT şi o tabelă simbolică de control, reprezentarea canonică trebuie de asemenea să permită reprezentarea acestor rezultate ale sintezei. Specificaţia funcţională de intrare, structura obţinută prin sinteză şi controlerul reprezintă obiecte în diferite domenii şi diferite nivele de proiectare, acestea fiind plasate pe axa funcţională, respectiv pe cea structurală a unei diagrame *Y*. Astfel, este necesară corelarea acestor obiecte pentru a se putea executa simularea multi-nivel şi depanarea.

O reprezentare internă ideală care se utilizează pentru sinteza de nivel înalt trebuie să respecte următoarele cerinţe:

- Trebuie să păstreze informaţiile întregului proiect în timpul sintezei, cuprinzând specificaţia originală, restricţiile de proiectare, informaţiile despre stările rezultate în urma sintezei, structurile obţinute şi legăturile între diferite obiecte.
- Trebuie să aibă o formă canonică astfel încât aceasta să asigure o vedere uniformă a reprezentării de către utilizatori şi modulele de sinteză.
- Trebuie să fie independentă de limbajul de descriere hardware, pentru a permite utilizarea diferitelor limbaje de descriere.
- Trebuie să fie suficient de puternică pentru a permite utilizarea a numeroase stiluri arhitecturale pentru implementare.

Unele din aceste cerinţe pentru o reprezentare internă pot fi în contradicţie între ele. De exemplu, nu există un singur limbaj funcţional standard, limbajele de descriere utilizând diferite tipuri de primitive operaţionale şi diferite nivele de abstractizare a datelor, rezultând diferite semantici pentru reprezentare. Totuşi, în practică se poate utiliza o reprezentare internă care este utilă pentru o anumită clasă de limbaje şi permite utilizarea unei anumite clase de arhitecturi.

În cele ce urmează se vor descrie diferitele etape ale sintezei de nivel înalt pentru un exemplu simplu, pentru a se ilustra tipul informațiilor care trebuie reprezentate. Apoi se vor descrie aspecte legate de compilarea limbajelor de descriere hardware într-o reprezentare intermediară bazată pe grafuri, şi se vor pune în evidență informațiile suplimentare generate la terminarea sintezei de nivel înalt. În final se vor descrie diferite transformări utilizate pentru optimizarea reprezentării interne.

2.2. Etapele sintezei de nivel înalt: un exemplu

Prima etapă în cadrul sintezei de nivel înalt este compilarea descrierii funcționale într-o reprezentare intermediară. În continuare se execută diferite etape ale sintezei, ca planificarea, selecția unităților, asignarea unităților funcționale, a elementelor de memorie şi a interconexiunilor, şi generarea controlerului.

```
ENTITY mult IS
   PORT (a_port, b_port:
           IN  BIT_VECTOR(3 DOWNTO 0);
         p_out:
           OUT BIT_VECTOR(7 DOWNTO 0);
         clk:   IN CLOCK;
         start: IN BIT;
         term:  OUT BIT;
        );
 END mult;


 ARCHITECTURE depl_ad OF mult IS
 BEGIN
   PROCESS
     VARIABLE a, b, p: BIT_VECTOR;
     VARIABLE cont: INTEGER;
     BEGIN
       WAIT UNTIL (start = '1');
 a := a_port; cont := 0;
       b := b_port; term <= '0';
 p := B"0000";
 WHILE (cont < 4) LOOP
         IF (a(0) = '1') THEN
             p := p + b;
         END IF;
         a := SHR (a, p(0));
         p := SHR (p, '0');
         cont := cont + 1;
       END LOOP;
       p_out <= p & a;
       term := '1';
     END PROCESS;
   END depl_ad;
```

Figura 2. Descrierea funcțională a unui circuit de înmulțire.

Se consideră descrierea funcțională în limbajul *VHDL* a unui circuit de înmulțire (**Figura 2**). Descrierea entității indică porturile de intrare *a_port* şi *b_port* de 4 biți, un semnal *start* şi semnalul de ceas *clk*. Circuitul înmulțeşte valorile de la porturile *a_port* şi *b_port*, plasează rezultatul de 8 biți la portul *p_out*, şi la terminarea operației activează semnalul *term*. Descrierea funcțională, utilizând un algoritm simplu de înmulțire prin deplasare şi adunare, este specificată într-un bloc PROCESS. De menționat că instrucțiunile dintr-un bloc PROCESS sunt executate secvențial, dar operațiile care nu au dependențe de date între ele se pot executa concurent. Astfel, descrierea secvențială în limbajul *VHDL* a circuitului de înmulțire conține un paralelism implicit.

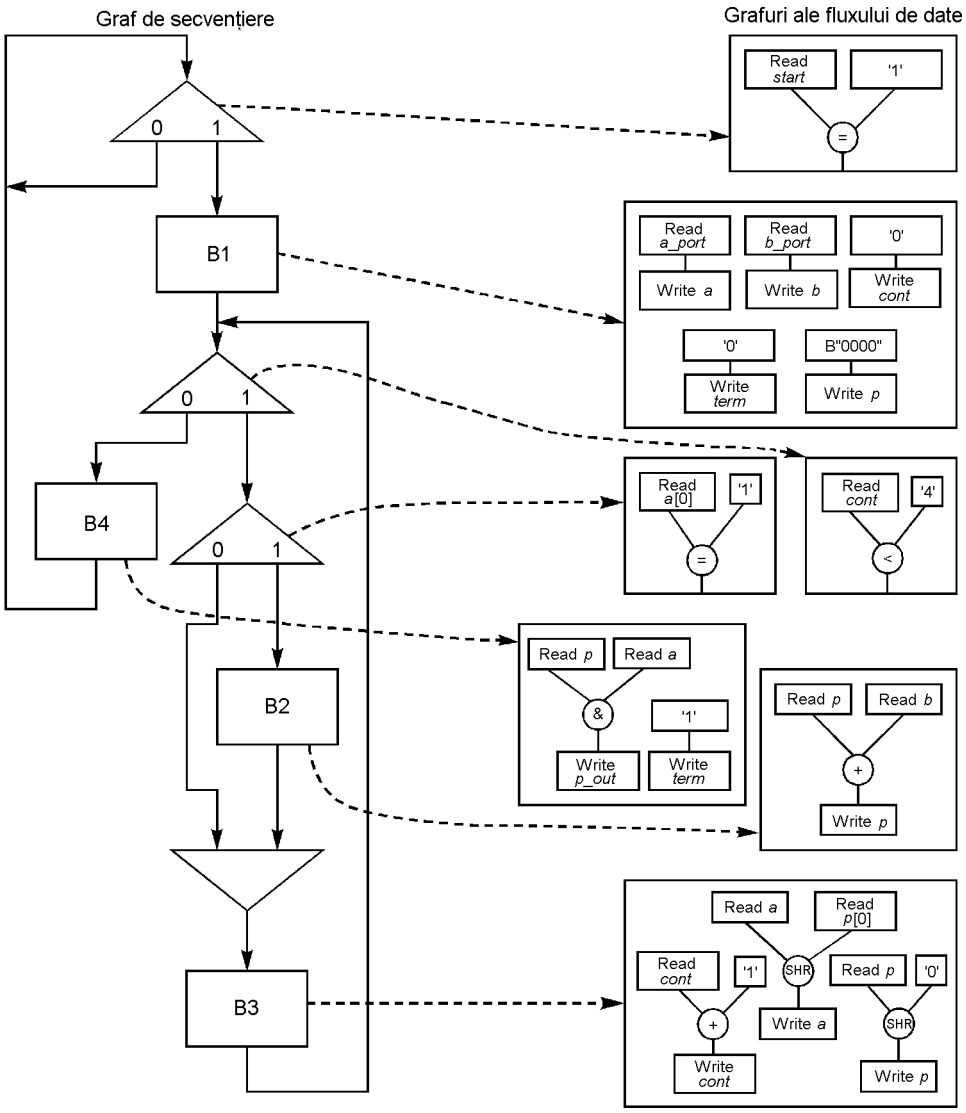


Figura 3. Graful fluxului de control şi de date pentru circuitul de înmulţire.

Descrierea circuitului de înmulţire se poate compila într-o reprezentare sub forma unui graf al fluxului de control şi de date (GFCD), ca în **Figura 3**. Graful fluxului de control al modelului GFCD reprezintă secvenţierea, ramificările condiţionate şi construcţiile de buclare ale descrierii funcţionale, iar graful fluxului de date reprezintă activităţile operaţionale indicate de instrucţiunile de asignare ale limbajului *VHDL*. Fiecare nod al grafului fluxului de control poate avea asociat un bloc al fluxului de date, care reprezintă operaţiile executate în cadrul nodului respectiv. Blocurile fluxului de date sunt similare cu blocurile de bază din limbajele de programare structurate. De notat că modelul GFCD este numai un exemplu din reprezentările intermediare posibile prin grafuri. Acest model

se utilizează pentru exemplificare deoarece reflectă în mod clar structura descrierii inițiale.

În cadrul descrierii circuitului de înmulțire, instrucțiunile de asignare dintr-un bloc de bază nu au dependențe de date între ele. De exemplu, blocul *B1* conține cinci asignări în graful fluxului de date, şi toate se pot executa în paralel. Reprezentarea fluxului de date din **Figura 3** pune în evidență paralelismul intrinsec din descrierea secvențială a unui proces în limbajul *VHDL*.

Descrierea *VHDL* originală, ca şi reprezentarea GFCD corespunzătoare, nu indică modul de implementare al circuitului de înmulțire. Variabilele, ca *a*, *b* şi *p* din **Figura 3**, nu sunt asignate elementelor de memorie. În mod similar, operațiile, ca de exemplu adunarea din blocul *B2*, nu sunt asignate unităților funcționale. Mai mult, descrierea *VHDL* şi modelul GFCD nu specifică secvențierea stărilor sau semnalele de control pentru activarea componentelor căii de date în fiecare stare.

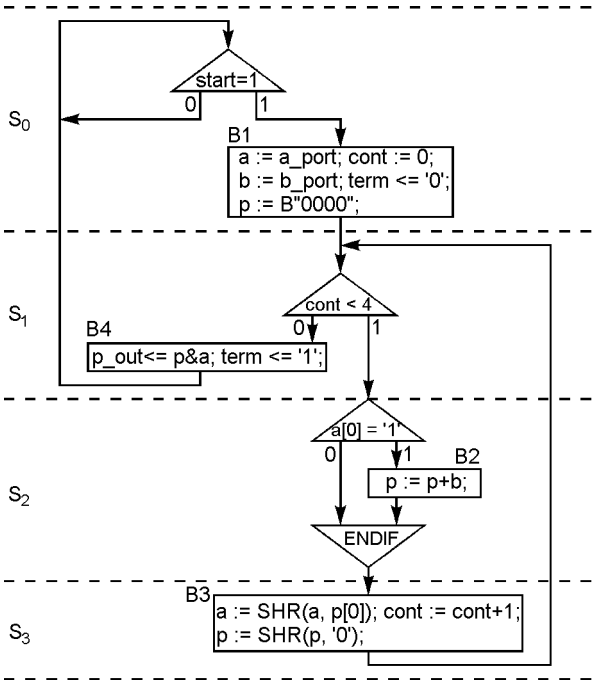


Figura 4. Planificarea operațiilor pentru circuitul de înmulțire.

Se va implementa circuitul de înmulțire utilizând modelul arhitectural al automatelor cu stări finite cu cale de date (ASFD). În primul rând, se realizează *planificarea operațiilor* pe parcursul a patru intervale sincrone de timp, notate S_0, S_1, S_2 şi S_3 (**Figura 4**). În starea S_0, circuitul aşteaptă activarea semnalului *start*. La activarea semnalului *start*, sunt citiți cei doi operanzi de la porturile *a_port* şi *b_port*, şi valoarea lor se atribuie variabilelor *a* şi *b*, celelalte variabile interne fiind inițializate. Din **Figura 3** se observă că nu

există dependențe de date între instrucțiunile de asignare din graful fluxului de date *B1*. Prin urmare, toate instrucțiunile din blocul *B1* se planifică într-o singură stare S_0. Starea S_1 este începutul buclei de înmulțire. Dacă bitul cel mai puțin semnificativ al variabilei *a* este 1, variabila *p* acumulează produsul parțial curent în starea S_2. În starea S_3, contorul este incrementat şi produsul parțial este deplasat la dreapta. La terminarea buclei, când contorul este egal cu 4 în starea S_1, înmulțirea este terminată şi rezultatul se transmite la portul *p_out*, indicând prin valoarea '1' a semnalului *term* sfârşitul operației. După planificare, se poate adnota fiecare nod al modelului GFCD cu starea în care acesta este planificat, menținând astfel o legătură între descrierea abstractă şi stările circuitului.

Următoarea etapă a sintezei de nivel înalt este *alocarea unităților (selecția)*, care determină tipul şi numărul unităților de la nivelul RT care vor fi utilizate. Se alocă unități de memorie (registre) pentru variabilele care sunt utilizate pe parcursul a mai multor stări. Asemenea variabile sunt *cont*, *a*, *b* şi *p*, pentru care se selectează patru registre numite *Cont_Reg*, *A_Reg*, *B_Reg* şi *Prod*. Pe lângă acestea, sunt necesare unități funcționale pentru implementarea operațiilor specificate în fiecare stare. Se alocă un sumator (*Sum*), două circuite de deplasare (*Depl1* şi *Depl2*) şi un comparator (*Comp*), după cum se indică în **Figura 5**.

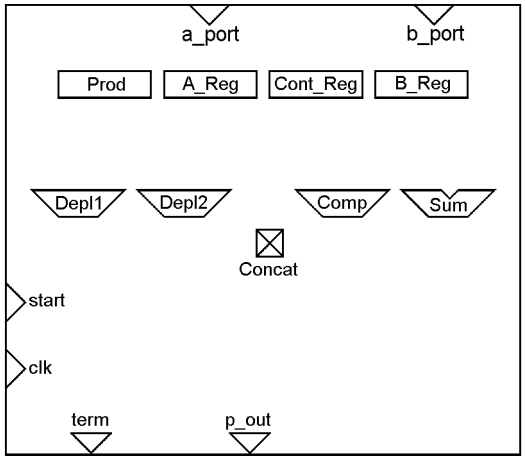


Figura 5. Selecția unităților pentru circuitul de înmulțire.

După alocarea unităților, urmează *asignarea unităților*: trebuie să se asigneze variabilele şi operatorii funcționali din reprezentarea GFCD unităților de memorie şi celor funcționale selectate. Se asignează variabilele *cont*, *a*, *b* şi *p* registrelor *Cont_Reg*, *A_Reg*, *B_Reg*, respectiv *Prod*. Operațiile de deplasare a variabilelor *p* şi *a* din starea S_3 se asignează circuitelor de deplasare *Depl1*, respectiv *Depl2*. Sumatorul *Sum* se partajează între operația de adunare din starea S_2 şi incremenatrea variabilei *cont* din starea S_3. Comparatorul *Comp* este alocat pentru testul variabilei *cont* din starea S_1.

Aceste unități funcționale şi de memorie sunt conectate în mod corespunzător pentru a se asigura transferurile de date corecte în calea de date. Se observă că există surse multiple la intrările unităților *Prod*, *A_Reg* şi *Sum*. Este necesară deci alocarea unor multiplexoare, notate cu *Mux1*, *Mux2*, respectiv *Mux3* în **Figura 6**.

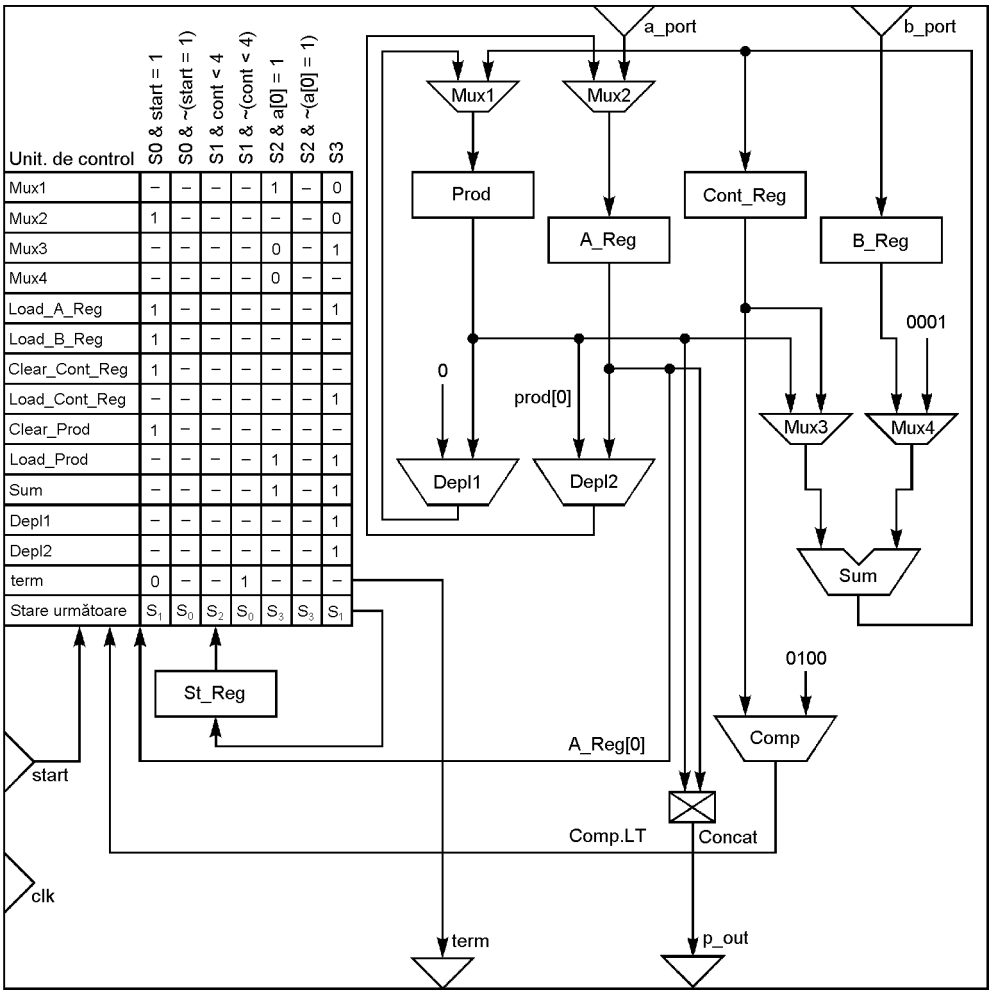


Figura 6. Rezultatul sintezei pentru circuitul de înmulțire.

În final, se determină *controlerul* care realizează secvențierea operațiilor şi controlează unitățile funcționale şi de memorie din calea de date. Tabela simbolică de control (*Unit_Control* din **Figura 6**) indică starea următoare şi semnalele de control necodificate pentru controlul unităților în fiecare stare. Fiecare coloană din această tabelă corespunde unei combinații între o stare şi o condiție, iar fiecare linie (cu excepția ultimei linii) reprezintă un semnal de control pentru o unitate din calea de date. Ultima linie a tabelei de control indică starea următoare pentru starea curentă şi valoarea condiției

respective. Pentru a exemplifica modul în care este generată această tabelă, considerăm prima coloană, care reprezintă starea S_0 şi condiția *start = 1*. Din **Figura 4** se observă că variabila *a*, care este asignată unității *A_Reg*, trebuie încărcată cu valoarea citită de la portul *a_port*. Acest transfer de date este realizat în **Figura 6** prin transferul datei din portul *a_port* prin intrarea din dreapta a multiplexorului *Mux2* şi încărcarea *A_Reg*. De aceea, semnalul de control pentru *Mux2* este setat la 1 (deci este selectată intrarea din dreapta a multiplexorului), şi semnalul *Load_A_Reg* este setat la 1 (deci *A_Reg* este încărcat). Ultima intrare din prima coloană a tabelului de control indică faptul că starea următoare este S_1. Restul tabelului de control este generat în mod similar.

Pornind de la descrierea funcțională, au fost adăugate mai multe tipuri de informații în timpul sintezei de nivel înalt: stări, unități RT şi conexiuni, informații de control a căii de date, şi informații de secvențiere a stărilor. În plus, este necesară corelarea acestor informații suplimentare cu descrierea funcțională inițială, utilizând adnotări şi/sau legături de asignare. Reprezentarea intermediară a proiectului trebuie deci să reprezinte în mod explicit asignarea stărilor, selecția şi asignarea unităților, structura RT interconectată, şi controlul simbolic. O asemenea reprezentare va permite parcurgerea tuturor fazelor sintezei de nivel înalt, de la specificația abstractă la implementarea finală, prin reprezentarea rezultatelor intermediare ale sintezei, ca şi a legăturilor între etapele intermediare ale sintezei.

2.3. Compilarea limbajelor de descriere

2.3.1. Exemplu de generare a reprezentării interne

Se va ilustra etapa de compilare utilizând o descriere *VHDL* ca intrare şi reprezentarea GFCD ca ieşire. În reprezentarea GFCD, construcțiile fluxului de control ale limbajului, ca buclele şi construcțiile condiționale (de exemplu, `IF` şi `CASE`), sunt reprezentate prin noduri ale fluxului de control, iar blocurile cu instrucțiuni de asignare dintre aceste construcții de control sunt reprezentate prin grafuri ale fluxului de date.

Se exemplifică generarea grafurilor fluxului de date pentru descrierea secvențială în limbajul *VHDL* din **Figura 7(a)**. În primul rând, analizorul sintactic al limbajului generează arbori sintactici prin translatarea instrucțiunilor individuale ale limbajului (**Figura 7(b)**). Deoarece aceşti arbori descriu instrucțiunile de asignare secvențiale în limbajul *VHDL*, se va utiliza lista de instrucțiuni (*Instr1*, *Instr2*, *Instr3*) pentru a ilustra ordinea de execuție a arborilor sintactici.

În continuare se interpretează ordinea de execuție a arborilor sintactici pe baza semanticii stilului limbajului de descriere utilizat. Dacă semantica limbajului indică faptul că instrucțiunile se execută concurent (de exemplu, într-o descriere de tipul fluxului de date în limbajul *VHDL*), arborii sintactici sunt analizați pentru a se asigura ca toate expresiile din partea dreaptă a instrucțiunilor de asignare să fie evaluate concurent, înaintea asignării valorilor variabilelor din partea stângă a instrucțiunilor. Arborii nu sunt modificați, cu excepția accesului la variabile comune din partea dreaptă a instrucțiunilor, care sunt unite.

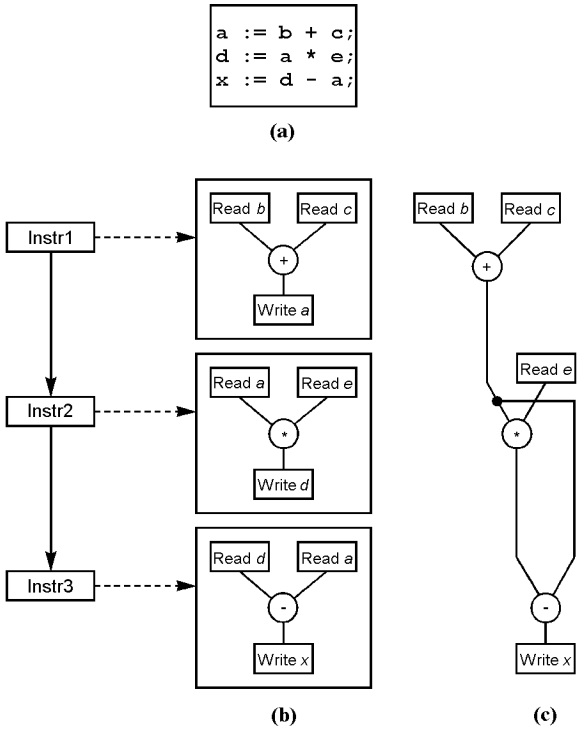


Figura 7. Generarea grafului fluxului de date (GFD): (a) instrucțiuni într-un limbaj de descriere; (b) arbori sintactici; (c) GFD.

Dacă construcțiile limbajului au o semantică de execuție secvențială (de exemplu, în cadrul unui bloc PROCESS al limbajului *VHDL*), se execută analiza fluxului de date între arborii sintactici pentru a pune în evidență concurența între instrucțiunile secvențiale ale limbajului. Această etapă este similară cu analiza fluxului de date executată de compilatoarele limbajelor de programare tradiționale. Deoarece descrierea din **Figura 7(a)** are o semantică de execuție secvențială, se execută analiza fluxului de date asupra arborilor sintactici din **Figura 7(b)**. Se observă că variabila *a* este definită în instrucțiunea *Instr1* şi este utilizată în instrucțiunile *Instr2* şi *Instr3*. Astfel, există dependențe de date pentru variabila *a* între instrucțiunea *Instr1* şi ambele instrucțiuni *Instr2* şi *Instr3*. Similar, variabila *d* este definită în instrucțiunea *Instr2* şi este utilizată în instrucțiunea *Instr3*, rezultând dependențe de date pentru variabila *d* între instrucțiunile *Instr2* şi *Instr3*.

Analiza fluxului de date se completează prin unirea tuturor arborilor sintactici într-un singur graf al fluxului de date, menținând toate dependențele de date care există între diferite instrucțiuni. **Figura 7(c)** prezintă graful fluxului de date generat pentru setul de instrucțiuni secvențiale de asignare în limbajul *VHDL* din **Figura 7(a)**.

2.3.2. Tehnici de compilare

Partea front-end a unui compilator execută analiza lexicală şi sintactică a limbajului, şi crează o formă intermediară. *Analizorul lexical* este componenta compilatorului care citeşte modelul sursă şi produce ca ieşire un set de simboluri numite atomi lexicali, care sunt utilizați pentru analiza sintactică. Un analizor lexical poate executa şi alte operații, ca eliminarea comentariilor şi expandarea macrourilor. Metavariabilele pot fi de asemenea prelucrate în această etapă.

Analizorul sintactic primeşte un set de simboluri, şi verifică dacă acestea respectă regulile sintactice ale limbajului, generând un set de arbori ai analizei sintactice. Un asemenea arbore este o reprezentare a structurii sintactice a limbajului. Erorile sintactice, ca şi unele erori semantice (ca de exemplu un operator aplicat unui operand incompatibil ca tip), sunt detectate în această etapă. Pot fi utilizate diferite pachete de programe pentru a crea analizoare lexicale şi analizoare sintactice. Asemenea programe sunt *lex* şi *yacc* din cadrul sistemului de operare *UNIX*.

În timp ce componentele front-end ale compilatoarelor limbajelor de programare şi ale limbajelor de descriere hardware sunt asemănătoare, următoarele componente sunt diferite. În particular, pentru limbajele de descriere hardware se utilizează diverse strategii în funcție de semantica lor.

Se consideră mai întâi *limbajele structurale*. O asignare într-un asemenea limbaj exprimă o relație între pini (sau module) şi conexiuni, fiind echivalentă din punct de vedere semantic cu un element al unei structuri de incidență. În mod similar, există o echivalență semantică între noțiunea de apel al unui modul şi structura ierarhică. Astfel, arborii sintactici pot fi transformați cu uşurință în liste de conexiuni (eventual ierarhice). Specificațiile sub forma listelor de conexiuni sunt preferate matricilor, deoarece ele sunt mai compacte. În general, nu se efectuează optimizări la compilarea limbajelor structurale.

Se consideră în continuare *limbaje care modelează circuite logice combinaționale*. Cazul cel mai simplu este cel al *limbajelor aplicative* care modelează un circuit printr-un set de ecuații booleene. De aceea, semantica modelului corespunde exact unei rețele logice care este o interconexiune între module, fiecare modul fiind caracterizat printr-o ecuație logică. Compilarea unei rețele logice este deci directă, ca şi în cazul modelelor structurale.

Un caz mai complex este cel în care limbajul are o *semantică procedurală*, eventual cu instrucțiuni de ramificație. Asignările multiple la o variabilă trebuie rezolvate prin anumite mecanisme. De exemplu, se poate utiliza o funcție de decizie modelată explicit. Astfel, în limbajele *VHDL* şi *Verilog*, semnalele pot avea diferite intensități, permițând modelarea, printre altele, a circuitelor cu trei stări.

Construcțiile de ramificație pot fi utilizate pentru modelarea rețelelor logice. Un mod obişnuit de utilizare a ramificațiilor este prin instrucțiunile de asignare condiționată la variabile. O construcție de ramificație poate fi înlocuită printr-o expresie logică, reprezentând disjuncția asignărilor posibile în conjuncție cu testul clauzei condiționale. Dacă o construcție de ramificație nu specifică o asignare pentru toate valorile clauzei condiționale, asignările lipsă reprezintă condiții indiferente ale variabilei respective, cu excepția

cazului când există o altă asignare la acea variabilă. Problema devine mai complexă în cazul existenței construcțiilor de ramificație imbricate.

De exemplu, considerăm următorul fragment al modelului unui circuit combinațional:

```
if (q)
{
      x = a + b;
      y = a + c;
}
else
      x = a b;
```

Presupunând că nu mai există alte asignări la variabilele `x` şi `y`, modelul poate fi expandat sub forma:

```
x = q(a + b) + q'ab
y = q(a + c)
```

unde `q'` reprezintă condiția indiferentă pentru `y`, deci `y` poate lua orice valoare dacă `q` este fals. Condițiile indiferente pot fi utilizate în diferite moduri pentru simplificarea modelului circuitului, de exemplu prin simplificarea expresiei pentru `y` prin `y = a + c`.

De multe ori, construcțiile de ramificație testează valoarea unei variabile cu un tip enumerat. Este necesară codificarea binară a variabilei pentru a se genera o descriere la nivelul logic. În anumite cazuri, este convenabilă reprezentarea valorilor unei asemenea variabile ca stări ale circuitului, codificarea fiind amânată până la etapa de sinteză logică.

Un *model secvențial* al unui automat cu stări finite este caracterizat printr-un set de acțiuni executate în funcție de stări şi de intrări. În general, starea este declarată printr-o variabilă cu un tip enumerat. Valorile posibile ale acestei variabile pot fi puse într-o corespondență de unu la unu cu stările automatului. Setul de acțiuni reprezintă componentele unei construcții de ramificare ale cărei clauze sunt legate de starea prezentă şi de valorile intrărilor. Aceste acțiuni sunt în general asignări combinaționale. Astfel, compilarea modelelor automatelor cu stări finite necesită recunoaşterea stărilor şi procesarea instrucțiunilor de asignare combinațională.

Compilarea modelelor hardware la nivel arhitectural implică o *analiză semantică* completă, care cuprinde analiza *fluxului de date* şi a *fluxului de control*, şi *verificări de tip*. Analiza semantică este executată asupra arborilor sintactici în diferite moduri, de exemplu prin transformarea arborilor într-o formă intermediară. Verificările de tip au anumite caracteristici în cazul compilării limbajelor de descriere hardware. Operațiile asupra vectorilor de variabile booleene sunt testate din punct de vedere al compatibilității operanzilor. Vectorii pot fi completați în unele cazuri cu valori de 1 sau 0 pentru a asigura compatibilitatea.

Operațiile asupra vectorilor booleeni trebuie asignate la operatori hardware care execută funcția corespunzătoare. De exemplu, suma a doi vectori booleeni trebuie interpretată ca o legătură la un circuit sumator. Similar, compararea a două valori întregi, care vor fi implementați prin vectori booleeni, trebuie translatată într-o legătură la un circuit comparator. Deoarece maparea la resursele hardware nu este întotdeauna unică, implementările hardware diferite având parametri de spațiu/performanță diferiți, în această

etapă se utilizează operatori hardware abstracți, şi asignarea la resursele hardware este amânată până la o etapă ulterioară de optimizare.

Analiza semantică a arborilor generați de analiza sintactică conduce la generarea unei forme intermediare, care reprezintă implementarea descrierii originale pe o maşină abstractă. O asemenea maşină este identificată printr-un set de operații şi dependențe, şi poate fi reprezentată printr-un graf, de exemplu un graf de secvențiere. Modelul hardware sub forma unei maşini abstracte este virtual, în sensul că nu face distincție între costurile, din punct de vedere al spațiului şi al întârzierilor, ale diferitelor operații. Astfel, se pot executa optimizări funcționale asupra unui asemenea model, abstractizând parametrii tehnologici ai circuitului.

Analiza fluxului de date şi de control determină structura ierarhică a grafului utilizat şi topologia entităților sale. Arborii sintactici pentru fiecare instrucțiune de asignare identifică operațiile corespunzătoare vârfurilor fiecărei entități a grafurilor. Muchiile sunt deduse prin considerarea dependențelor fluxului de date şi a dependențelor de secvențiere. Fiecare entitate terminală a unui graf ierarhic corespunde unui bloc de bază.

Analiza fluxului de date cuprinde mai multe operații, fiind utilizată ca o bază pentru optimizarea funcțională. Ea include extragerea duratei de viață a variabilelor. De exemplu, grafurile de secvențiere nu modelează explicit faptul că variabilele trebuie memorate pe durata de viață a acestora, ceea ce implică unele costuri suplimentare la implementare. La considerarea modelelor hardware cu semantici imperative, pot apare asignări multiple la variabile. Variabilele păstrează valorile lor până la următoarea asignare. De aceea ele pot corespunde registrelor din implementarea hardware. De asemenea, ele pot corespunde conexiunilor, dacă informația pe care o reprezintă nu trebuie memorată. Implementarea hardware a variabilelor este decisă în etapele ulterioare ale sintezei de nivel înalt.

Un aspect important al sintezei este de a propaga restricțiile hardware specificate implicit sau explicit în cadrul descrierilor. De exemplu, pot fi specificate restricții explicite de timp, prin etichetarea operațiilor şi asigurarea mijloacelor pentru exprimarea duratelor minime şi maxime între momentele de început ale unor perechi de operații. Restricțiile sunt adăugate modelului şi sunt utilizate la optimizarea arhitecturală. În asemenea cazuri, ca şi în cazurile în care nu sunt specificate restricții de timp, planificarea grafului de secvențiere poate fi optimizată prin algoritmi specializați.

Restricțiile pot asocia operații cu anumiți operatori hardware. De exemplu, modelul poate specifica utilizarea unei implementări particulare pentru o anumită adunare. Asemenea restricții pot fi considerate ca indicații care trebuie urmate de sistemele de sinteză.

2.4. Reprezentarea descrierilor hardware

Descrierile unităților hardware sunt reprezentate de obicei prin grafuri, cum este modelul GFCD. Aceste grafuri diferă între ele prin modul de reprezentare a construcțiilor de control şi reprezentarea transferurilor de date în cadrul unui graf al fluxului de date. În continuare, se vor prezenta mai întâi unele descrieri simple pentru a ilustra reprezentarea

construcțiilor fluxului de control. Se vor prezenta apoi unele opțiuni pentru reprezentarea informațiilor de secvențiere şi de temporizare. În final, se vor utiliza exemple pentru a ilustra trei clase specifice de grafuri utilizate în sistemele de sinteză de nivel înalt: grafuri disjuncte ale fluxului de date şi de control, grafuri hibride, şi reprezentări prin arbori sintactici.

2.4.1. Reprezentarea fluxului de control

Fluxul de control poate fi reprezentat în moduri diferite. Se vor prezenta două reprezentări simple ale construcțiilor de control. Fragmentul de cod *VHDL* din **Figura 8(a)** conține o instrucțiune CASE care selectează un set de asignări pe baza valorii semnalului *c*. În prima reprezentare (**Figura 8(b)**), construcțiile de control sunt asociate cu nodurile fluxului de control, care păstrează secvențierea explicită şi fluxul de control specificat în descrierea inițială. Graful fluxului de control constă din noduri de ramificație condiționată, noduri de reuniune şi blocuri de instrucțiuni de asignare, reprezentate în mod simbolic prin triunghiuri, triunghiuri inversate, respectiv dreptunghiuri. Fiecare nod al fluxului de control poate avea un bloc al fluxului de date asociat, care descrie operațiile executate de acest nod (de exemplu, asignări sau teste). De exemplu, în **Figura 8(b)** instrucțiunea CASE este asociată cu o construcție de ramificație condițională, iar asignările din cadrul fiecărei evaluări condiționale sunt asociate cu un bloc al fluxului de date. În această reprezentare a fluxului de control, fiecare cale a instrucțiunii CASE prezintă în mod explicit o excluziune mutuală, astfel încât algoritmii de sinteză vor putea partaja cu uşurință resurse între diferite ramuri ale construcției condiționale. Această reprezentare este apropiată de fluxul de control din descrierea inițială, astfel încât se poate efectua legătura cu descrierea originală. Această metodă de reprezentare este similară cu grafurile fluxului de control şi de date cu blocuri de bază utilizate de compilatoarele limbajelor de programare.

În cea de-a doua reprezentare, construcțiile de control sunt translatate într-un graf al fluxului de date prin evaluarea în paralel a tuturor ramurilor unei construcții condiționale şi alegerea valorilor corecte pentru asignare după ce toate ramurile au fost executate. Pentru aceasta, se calculează toate valorile posibile pentru o variabilă destinație şi se selectează valoarea corespunzătoare pe baza valorii variabilei de condiție. În această reprezentare, se utilizează cercuri pentru a indica operațiile, arce pentru a indica fluxul de date, dreptunghiuri pentru a indica citirea şi scrierea datelor, şi triunghiuri inversate pentru a exprima selecția datelor pe baza valorii unui semnal de control.

Figura 8(c) arată reprezentarea de tipul fluxului de date pentru instrucțiunea CASE din **Figura 8(a)**. Nodurile reprezentate prin triunghiuri inversate selectează valorile corespunzătoare pentru variabilele destinație *x* şi *a*, care apar în partea stângă din cel puțin o instrucțiune de asignare.

Ca şi reprezentarea de tipul fluxului de control, reprezentarea de tipul fluxului de date pune în evidență în mod explicit concurența datorită excluziunii mutuale a diferitelor căi condiționale. Însă, deoarece a doua reprezentare evaluează toate ramurile unui test condițional în paralel, o porțiune mai largă a grafului este disponibilă pentru rafinare şi optimizare. În consecință, reprezentarea de tipul fluxului de date este mai potrivită pentru operația de planificare a unui cod liniar.

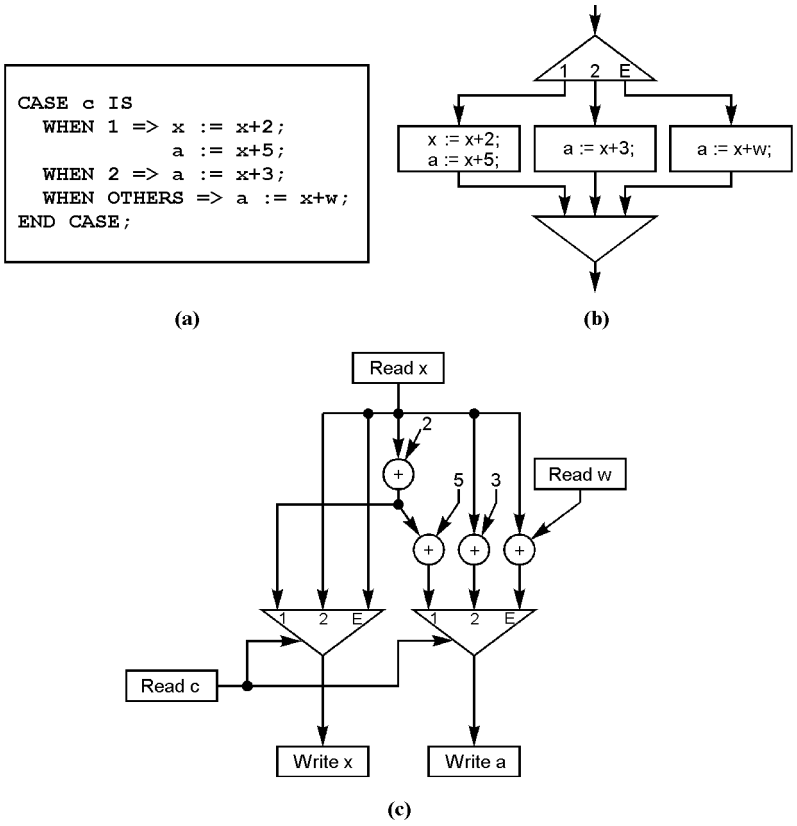


Figura 8. Exemplu de instrucțiune *CASE*: (a) descrierea în limbajul *VHDL*; (b) reprezentarea GFC; (c) reprezentarea GFD.

Pe de altă parte, reconstrucția descrierii originale din reprezentarea de tipul fluxului de date devine mai dificilă, deoarece toate informațiile fluxului de control sunt înglobate în graful fluxului de date. De asemenea, deoarece toate condițiile şi buclele imbricate sunt cuprinse într-un singur graf al fluxului de date, structurile puternic imbricate generează mai multe nivele de selectori care aleg între căile condiționale. Aceasta poate conduce la grafuri ale fluxului de date care sunt de dimensiuni mari şi greu de manipulat.

2.4.2. Reprezentarea secvențierii şi a temporizării

În plus față de dependențele fluxului de control şi de date, reprezentarea intermediară trebuie să păstreze relații de ordonare care sunt specificate implicit sau explicit în descrierea inițială. Se pot utiliza arcuri de precedență pentru a indica explicit ordonarea între nodurile unui graf. Arcurile de precedență sunt necesare pentru a impune ordonarea operațiilor externe de citire şi scriere din descrierea hardware (de exemplu, două operații succesive de citire a unui port). Arcurile de precedență pot fi utilizate şi pentru a impune ordonarea accesurilor la tablouri atunci când valorile de index nu sunt constante.

În **Figura 9(a)** se prezintă un fragment al unei descrieri concurente în limbajul *VHDL* în care semnalele *a* şi *b* sunt incrementate şi interschimbate. Deoarece aceste instrucțiuni se execută concurent, operațiile de citire trebuie să preceadă operațiile de scriere a semnalelor *a* şi *b*. Arcele cu linii îngroşate din reprezentările fluxurilor de date din **Figura 9(b)** indică aceste relații de precedență.

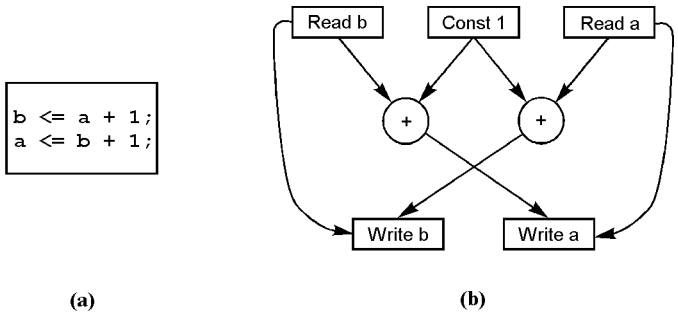


Figura 9. Flux de date cu arce de precedență: (a) descriere concurentă; (b) reprezentare GFD.

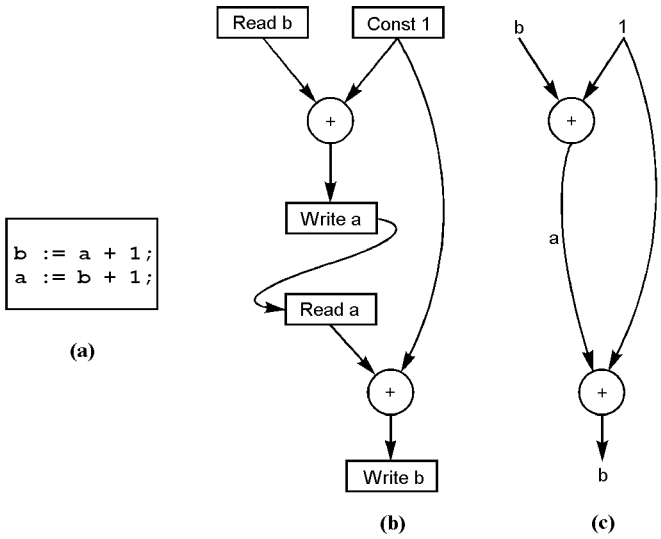


Figura 10. Reprezentarea accesului la variabile: (a) descriere *VHDL* secvențială; (b) GFD cu noduri de acces la variabile; (c) GFD cu trasee ale variabilelor.

Metodele de reprezentare prin grafuri diferă şi prin reprezentarea accesului la variabile (a operațiilor de citire şi scriere). Anumite metode reprezintă în mod explicit fiecare citire şi scriere a unei variabile printr-un nod de acces la acea variabilă. Într-un bloc secvențial, trebuie să se asigure ca valorile variabilelor să fie citite înaintea definirii unor

noi valori pentru acestea (prin scriere). Invers, un acces pentru citirea unei variabile nu trebuie executat înaintea definirii valorii sale. Se pot utiliza arce de precedență pentru a impune aceste ordonări ale nodurilor de citire şi scriere pentru aceeaşi variabilă (**Figura 10(b)**). Deoarece accesurile la variabile pot fi legate de elemente de memorie care necesită o parte de control, această metodă de reprezentare explicită permite tratarea uniformă a accesurilor la variabile şi a operatorilor limbajului, ceea ce uşurează operațiile de selecție a unităților şi de asignare a acestora.

Alte metode bazate pe grafuri reprezintă accesurile la variabile în mod implicit, ca urme (trasee) ale datelor, prin utilizarea arcelor în cadrul reprezentării fluxului de date (**Figura 10(c)**). Spre deosebire de metodele explicite, optimizările fluxului de date se pot executa mai uşor asupra traseelor valorilor, față de cazul în care aceste accesuri ar fi reprezentate în mod explicit ca noduri ale operațiilor.

De multe ori se specifică restricții de timp în descrierile funcționale pentru a se asigura funcționarea corectă sau pentru a se realiza anumite cerințe de performanță. Informațiile de temporizare reprezentate în grafurile fluxurilor de control şi de date sunt utilizate ca restricții pentru etapele de planificare, selecția unităților şi asignarea acestora. Aceste informații reprezintă de asemenea restricții asupra performanțelor unei implementări (de exemplu, ciclul de ceas). În funcție de metoda de reprezentare aleasă, aceste informații pot fi puse în evidență în mai multe moduri. Se prezintă în continuare unele exemple de reprezentare a restricțiilor de timp în grafurile fluxului de date şi de control.

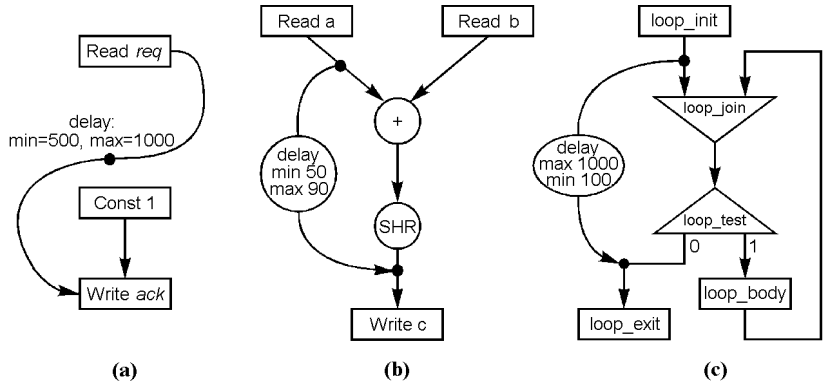


Figura 11. Reprezentarea restricțiilor de timp: (a) adnotarea arcelor de precedență ale GFD; (b) utilizarea unui nod de temporizare în GFD; (c) utilizarea unui nod de temporizare în GFC.

La nivelul unui *graf al fluxului de date*, se pot utiliza două metode simple: adnotarea informațiilor de timp pe arcele de precedență dintre două noduri ale grafului, sau crearea unor noduri de temporizare între două arce ale grafului. *Adnotarea informațiilor de timp* pe arcele de precedență dintre nodurile unui graf al fluxului de date este utilă pentru reprezentarea restricțiilor minime, maxime şi nominale de timp între execuția a două operații. O asemenea restricție de timp apare adesea atunci când mai multe operații de citire

şi scriere la porturile externe ale circuitului trebuie ordonate şi separate în timp pentru funcționarea corectă (de exemplu, în cazul protocoalelor de comunicație). De exemplu, în **Figura 11(a)** se indică o restricție de timp *min=500*, *max=1000*, adnotată pe arcul de precedență dintre operația de citire de la portul de intrare *req* şi operația de scriere la portul de ieşire *ack*.

A doua metodă, cea prin *crearea unor noduri de temporizare* între arcele grafului fluxului de control, descrie întârzierile punct la punct în cadrul grafului. Aceste întârzieri reprezintă restricții asupra timpului minim şi maxim de execuție pentru unii operatori individuali, ca şi pentru grupuri de operatori din calea de date, între începutul şi sfârşitul nodului de temporizare. De exemplu, în **Figura 11(b)** există un nod de temporizare între intrarea din stânga a operației de adunare şi ieşirea operației de deplasare *SHR*, care restricționează execuția operațiilor de adunare şi deplasare între un minim de 50 ns şi un maxim de 90 ns. Atunci când nodurile de temporizare sunt utilizate între intrările şi ieşirea unui singur nod, acestea modelează întârzierile pin la pin pentru componenta care implementează operația.

În mod similar, la nivelul unui *graf al fluxului de control*, se poate insera un nod de temporizare între două arce ale grafului. Semantica unui asemenea nod diferă în funcție de metoda de reprezentare utilizată. În cazul reprezentării prin modelul GFCD, un nod de temporizare între două arce ale fluxului de control indică o restricție asupra execuției tuturor blocurilor fluxului de date de-a lungul căii de control între începutul şi sfârşitul nodului de temporizare. De aceea, acest nod de temporizare reprezintă restricții asupra cerințelor de performanță pentru mai multe blocuri ale fluxului de date, sau chiar pentru întregul circuit. De exemplu, în **Figura 11(c)** se indică limitele minime şi maxime ale întârzierilor pentru execuția unei bucle, specificate prin inserarea unui nod de întârziere între arcul de intrare şi arcul de ieşire al buclei din graful fluxului de control.

În continuare se prezintă trei reprezentări diferite bazate pe grafuri care sunt utilizate în sinteza de nivel înalt.

2.4.3. Reprezentări disjuncte ale fluxului de control şi de date

Reprezentarea disjunctă a fluxului de control şi de date este similară cu reprezentările intermediare prin grafuri utilizate de compilatoarele standard, unde construcțiilor de control le corespund nodurile fluxului de control, iar asignărilor din blocurile de bază le corespund nodurile fluxului de date. În această reprezentare, nodurile fluxului de control sunt păstrate separat de nodurile fluxului de date, astfel că secvențierea globală a proiectului este mai uşor de urmărit.

Graful fluxului de control este generat direct din construcțiile de control ale limbajului. Se utilizează mai multe tipuri de noduri ale fluxului de control. Construcțiile limbajului ca *"if", "case"* şi *"loop"* sunt translatate în noduri de bifurcație şi de reunire. Apelurile de proceduri şi de funcții sunt reprezentate prin noduri de apel. Se utilizează noduri suplimentare de demarcație pentru a defini limitele programelor funcționale, a funcțiilor şi a procedurilor.

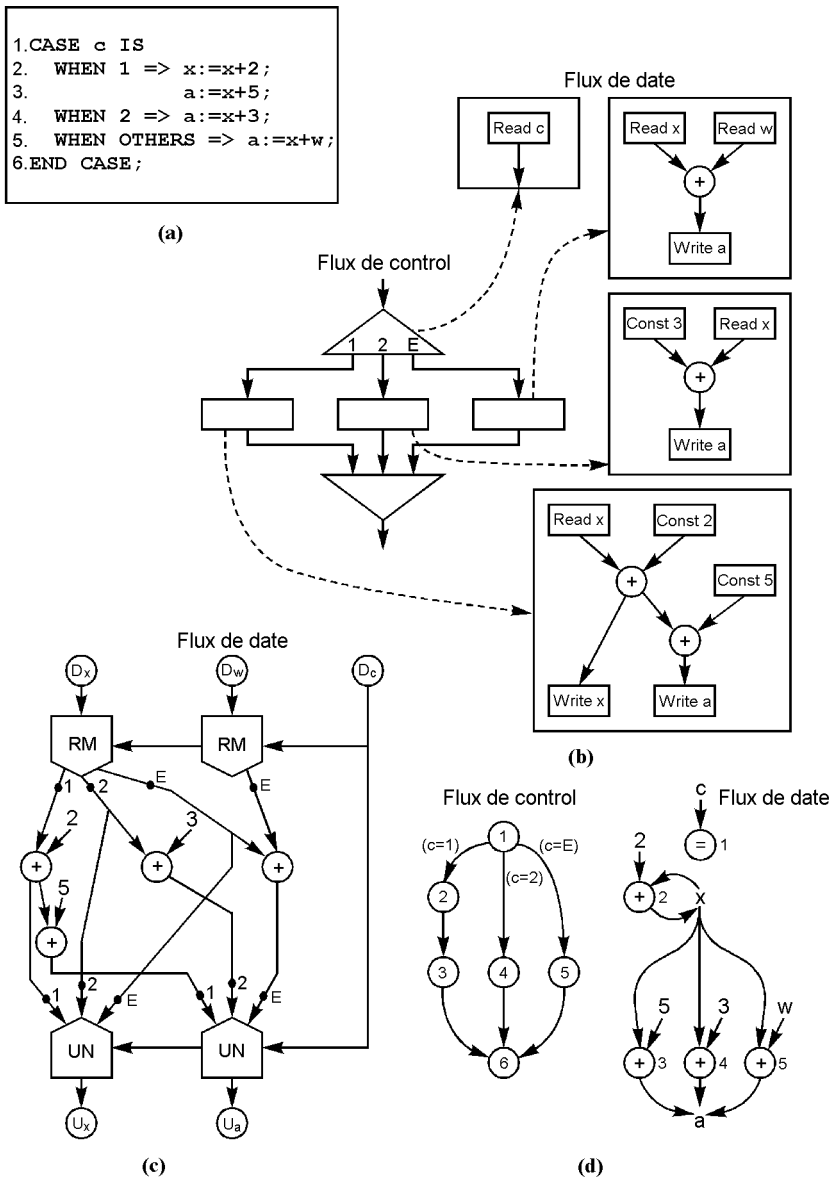


Figura 12. Reprezentări pentru graful fluxului: (a) fragment în limbajul *VHDL*; (b) GFCD partiționat; (c) graf hibrid al lui *DeJong*; (d) graf *SSIM*.

Graful fluxului de date constă de asemenea din mai multe tipuri de noduri. Nodurile de operație ale fluxului de date reprezintă operatori ai limbajului, nodurile de referire la variabile reprezintă citirea şi scrierea variabilelor, iar nodurile de index reprezintă accesurile la tablouri. Un singur nod al unui bloc de instrucțiuni de control reprezintă un graf al fluxului de date cu mai multe noduri ale fluxului de date. Se consideră instrucțiunea CASE din **Figura 12(a)**. Reprezentarea prin modelul GFCD este prezentată în **Figura 12(b)**. Acest graf reprezintă în mod explicit accesurile la variabile utilizând noduri ale fluxului de date.

Modelul *DDS* este o altă reprezentare care utilizează grafuri separate pentru fluxul de date şi fluxul de control/temporizare, cu legături între ele pentru a indica dependențele şi secvențierea.

2.4.4. Reprezentări hibride ale fluxului de control şi de date

Reprezentarea hibridă a fluxului de control şi de date înglobează toate informațiile de control şi de secvențiere în mod explicit în graful fluxului de date. Un exemplu al acestui model este reprezentarea fluxului de date din **Figura 8(c)**, în care toate căile mutual exclusive sunt executate concurent şi rezultatul final este selectat pe baza valorii condiției.

Reprezentarea lui *DeJong* este un alt model bazat pe fluxul de date, în care se execută numai căile condiționale active, şi nu toate ramurile unui test. Reprezentarea utilizează noduri de ramificație şi de unire pentru a determina calea din graful fluxului de date care va fi executată. Un exemplu de reprezentare a lui *DeJong* pentru instrucțiunea CASE din **Figura 12(a)** este prezentat în **Figura 12(c)**. În acest exemplu, D_X, D_W şi D_C reprezintă valorile variabilelor *X*, *W* şi *C* din partea dreaptă a asignărilor, în timp ce U_X şi U_A reprezintă asignările la variabilele destinație *X* şi *A*. Nodurile etichetate cu *RM* reprezintă ramificații, iar cele etichetate cu *UN* reprezintă asignări condiționate. Arcele din această reprezentare sunt adnotate cu valorile condițiilor pentru a pune în evidență calea corespunzătoare pentru execuție.

Sistemul *DSL* utilizează o variație a modelului hibrid al fluxului de control şi de date, unde nodurile grafului reprezintă operații ale limbajului şi variabile. Pe baza acestor noduri sunt construite trei grafuri, utilizând trei tipuri de arce: *arcele de secvențiere* reprezintă informațiile de control şi de secvențiere din descrierea sursă, *arcele fluxului de date* indică dependențele de date între operații, iar *arcele de temporizare* indică restricțiile de timp între diferite operații. Modelul *SSIM* este o extensie a modelului *DSL* care păstrează partea de control şi fluxul de date disjuncte, prin duplicarea nodurilor de operație şi prin indicarea arcelor de secvențiere ale fluxului de control în primul graf, şi a arcelor de dependență a fluxului de date în al doilea graf. Această relație unu la unu între nodurile fluxului de control şi de date pentru descrierea *VHDL* din **Figura 12(a)** este indicată în **Figura 12(d)**.

Modelul *DSFG* este utilizat pentru reprezentarea grafurilor de flux ale semnalelor, obținute din descrierile în limbajul *Silage* a aplicațiilor de procesare a semnalelor digitale. Modelul este o altă reprezentare hibridă, în care graful fluxului semnalelor (graful fluxului de date) este adnotat cu informații de control şi restricții de proiectare. Similar, reprezentarea *SIF* utilizată de sistemul de sinteză *Olympus* utilizează un graf de secvențiere ierarhic pentru a indica dependențele fluxului de date şi de control.

2.4.5. Reprezentări prin arbori sintactici

În **Figura 7(b)** s-au prezentat arborii sintactici generați în procesul de compilare a unei descrieri într-o reprezentare sub forma unui graf. Aceşti arbori pot fi utilizați ca reprezentare intermediară, şi pot fi adnotați cu asignări şi structuri pe măsura continuării sintezei.

Deoarece arborii sintactici sunt păstrați fără o analiză a fluxului de date între instrucțiuni, această metodă are avantajul că reprezentarea este apropiată de descrierea funcțională inițială. Totuşi, reprezentarea prin arbori sintactici nu indică în mod explicit paralelismul potențial între diferite instrucțiuni. De aceea, sistemele de sinteză trebuie să extragă aceste informații din reprezentarea internă atunci când este necesar.

Un exemplu de reprezentare intermediară prin arbori sintactici este *TREEMOLA*, care păstrează liste de arbori sintactici care provin din descrierea funcțională în limbajul *MIMOLA*.

2.5. Reprezentarea rezultatelor sintezei de nivel înalt

Rezultatul sintezei de nivel înalt este o cale de date structurală a unor componente RT interconectate, şi o tabelă simbolică de control. Componentele RT şi conexiunile lor reprezintă obiecte în domeniul structural al sintezei, iar tabela simbolică de control reprezintă automatul cu stări finite obținut, care este la un nivel inferior față de descrierea inițială. Deoarece obiectele care trebuie reprezentate sunt în domenii de proiectare diferite şi la nivele diferite, structura şi controlerul obținute prin sinteză se păstrează în general separat față de descrierea inițială, şi sunt corelate cu aceasta cu ajutorul unor legături.

Structura rezultată este memorată de obicei ca o listă de conexiuni sau de noduri compuse din trei părți: o listă de componente, o listă de semnale, şi interconexiunile dintre componente şi semnale. Pentru fiecare conexiune (semnal), o listă de conexiuni specifică porturile sursă şi destinație ale componentelor. Invers, pentru fiecare nod (componentă), o listă de noduri specifică semnalele care sunt conectate la porturile de intrare şi de ieşire ale componentei. Listele de conexiuni şi de noduri pot fi foarte variate ca sintaxă, dar ele conțin aceleaşi informații de interconectare.

Se poate menține o corelație între entitățile din descrierea funcțională (de exemplu, operații şi variabile) şi structura RT rezultată prin sinteză prin adnotarea fiecărui nod al grafului cu o legătură la componenta structurală corespunzătoare. De exemplu, nodul de adunare al blocului *B2* din **Figura 3** poate fi adnotat cu legătura la componenta *Sum*. Similar, fiecare nod de acces de citire sau scriere a variabilei *p* din grafurile fluxului de date din **Figura 3** poate fi adnotat cu legătura la registrul *Prod*.

O tabelă simbolică de control descrie stările, tranzițiile între stări şi semnalele de control activate pentru a valida acțiunile corespunzătoare din calea de date în fiecare stare. Tabela simbolică de control pentru exemplul circuitului de înmulțire din **Figura 3** este prezentată în unitatea de control din **Figura 6**. Tabela de control utilizează o reprezentare simbolică a stărilor şi a ieşirilor de control, fără a utiliza o anumită codificare particulară. Aceasta se ilustrează în **Figura 6**, unde atât stările (de exemplu S_0), cât şi ieşirile de control (de exemplu, *Load_A_Reg*) sunt reprezentate simbolic. Se pot

menține legături între descrierea originală şi tabela simbolică de control prin adnotarea fiecărui nod al grafului cu starea şi condiția în care este executat. De exemplu, deoarece blocul de instrucțiuni *B2* este planificat în starea S_2 atunci când *a(0) = '1'* (**Figura 4**), se poate adnota fiecare nod de operație din blocul fluxului de date *B2* (**Figura 3**) cu starea planificată S_2 şi condiția *a(0) = '1'*.

2.6. Transformări

Reprezentarea inițială prin grafuri ale fluxului de date şi de control este apropiată de descrierea originală, şi deci poate reprezenta construcții sintactice ale limbajului care nu sunt utile sau relevante pentru sinteză. Pot fi aplicate transformări asupra grafului inițial înaintea etapelor de planificare şi alocare ale sintezei de nivel înalt, creând un alt graf care este mai potrivit pentru aceste etape ale sintezei.

Fiecare transformare are efecte diferite. O transformare de simplificare elimină redundanțele pe baza sintaxei limbajului de descriere. O transformare de optimizare îmbunătățeşte reprezentarea prin eliminarea sau înlocuirea corespunzătoare a unor segmente ale reprezentării cu altele mai eficiente. O transformare de restructurare modifică reprezentarea pentru adaptarea acesteia unui stil specific (de exemplu, pentru prelucrarea pipeline), sau pentru a asigura diferite grade de paralelism.

Transformările pot fi aplicate asupra reprezentărilor în diferite etape ale sintezei de nivel înalt. În timpul compilării descrierii într-un graf al fluxului, se pot executa diferite optimizări de către compilator pentru eliminarea construcțiilor sintactice suplimentare şi a redundanțelor din specificarea prin limbajul de descriere hardware. Transformările grafurilor sunt utilizate pentru a converti părți ale reprezentării de la un stil (de exemplu, de tipul fluxului de control) la altul (de exemplu, de tipul fluxului de date), şi pentru a modifica gradul de paralelism. Transformările specifice unităților hardware utilizează proprietățile componentelor RT şi a celor logice pentru a efectua optimizări (de exemplu, înlocuirea unui segment al grafului fluxului de date care incrementează o variabilă cu o operație de incrementare). În continuare se vor trece în revistă pe scurt tipurile de transformări şi se vor ilustra unele transformări utilizând exemple simple.

2.6.1. Transformări efectuate de compilator

Deoarece reprezentarea sub forma unor grafuri provine de obicei dintr-o descriere într-un limbaj imperativ, se pot utiliza diferite tehnici de optimizare ale compilatoarelor standard pentru transformarea acestei reprezentări.

Propagarea constantelor şi a variabilelor. *Propagarea constantelor* (**Figura 13(a)**) constă din detectarea operanzilor de tipul constantelor şi calculul valorii operației cu aceşti operanzi. Astfel se înlocuieşte un segment al grafului care conține un operator aritmetic cu operanzi constanți cu rezultatul operației (o singură constantă). Constanta rezultată se poate propaga în continuare pe arcele fluxului de date.

Propagarea variabilelor constă în detectarea copiilor variabilelor, deci a asignărilor de forma $x = y$, şi utilizarea părții din dreapta a asignării în următoarele referiri, în

locul părții din stânga. Analiza fluxului de date permite identificarea instrucțiunilor pentru care transformarea poate fi efectuată. De exemplu, propagarea variabilei *y* nu poate fi efectuată după o nouă asignare la variabila *x*. Propagarea variabilelor permite eliminarea asignării copiilor. De observat că aceste asignări pot fi introduse de alte transformări.

Considerăm următorul fragment:

```
a = x; b = a+1; c = 2*a;
```

Acest fragment poate fi înlocuit prin:

```
a = x; b = x+1; c = 2*x;
```

Instrucțiunea `a = x;` poate fi apoi eliminată, dacă nu mai există alte referiri la variabila `a`.

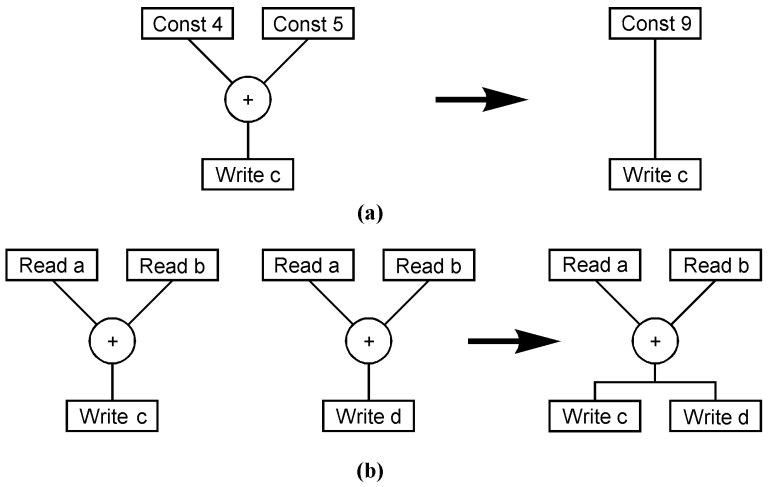


Figura 13. Optimizări efectuate de compilator: (a) propagarea constantelor; (b) eliminarea operatorilor redundanți.

Eliminarea operatorilor redundanți. Aceasta este o altă optimizare care elimină un operator dintr-un graf dacă există un alt operator cu intrări identice în cadrul grafului (**Figura 13(b)**). Această transformare se poate generaliza pentru căutarea şi **eliminarea subexpresiilor comune** care apar la ieşirile tuturor operatorilor din graf.

Căutarea subexpresiilor aritmetice comune este simplificată dacă expresiile aritmetice sunt reduse la expresii cu doi termeni. În acest caz, transformarea constă din selectarea unei operații aritmetice destinație şi căutarea unei operații precedente de acelaşi tip şi cu aceiaşi operanzi. Poate fi utilizată comutativitatea operatorilor. Prin analiza fluxului de date trebuie să se asigure ca în oricare expresie găsită operanzii să aibă întotdeauna aceleaşi valori. Dacă se găseşte o expresie precedentă identică, expresia destinație este înlocuită cu o copie a variabilei care reprezintă rezultatul expresiei precedente identice.

Considerăm fragmentul următor:

```
a = x+y; b = a+1; c = x+y;
```

Acest fragment poate fi înlocuit prin:

```
a = x+y; b = a+1; c = a;
```

De observat că s-a introdus o copie pentru variabila `a`, care poate fi propagată în continuare.

Reducerea numărului de accesuri la tablouri. Asupra variabilelor de tip tablou pot fi efectuate diferite optimizări de către compilator pentru sinteza de nivel înalt. Deoarece tablourile din descrierea funcțională vor fi implementate prin memorii, reducerea numărului de accesuri la tablouri va reduce întârzierile de acces la datele din memorie. Referirile la tablouri prin indici variabili determină şiruri lungi de dependențe între accesurile consecutive la acestea. Se pot utiliza transformări de optimizare pentru a reduce aceste şiruri de dependențe. Dacă un şir este pe o cale critică, o asemenea optimizare reduce lungimea căii critice şi îmbunătățește performanțele.

Reducerea complexității operatorilor. Această optimizare constă din reducerea costului de implementare a unui operator prin utilizarea unui operator mai simplu. Deşi în principiu sunt necesare informații despre implementarea hardware, adesea sunt valabile anumite considerații generale. De exemplu, o înmulțire cu 2 (sau cu o putere a acestuia) poate fi înlocuită cu o deplasare. Circuitele de deplasare sunt mai rapide şi mai simple decât circuitele de înmulțire în cele mai multe implementări.

Considerând următorul fragment:

```
a = x^2; b = 3*x;
```

acesta poate fi înlocuit prin:

```
a = x*x; t = x<<1; b = x+t;
```

Transformarea codului. Această optimizare se poate aplica adesea asupra invarianților buclelor, deci a cantităților care sunt calculate în interiorul unei construcții iterative, dar a căror valoare nu se modifică de la o iterație la alta. Scopul este de a se evita evaluarea repetitivă a aceleiaşi expresii.

Considerăm următoarea construcție iterativă:

```
for (i = 1; i <= a*b) {...}
```

unde variabilele `a` şi `b` nu sunt actualizate în cadrul buclei. Construcția se poate transforma în:

```
t = a*b; for (i = 1; i <= t;) {...}
```

Înlocuirea construcțiilor sintactice specifice. Anumite transformări efectuate de compilator sunt specifice limbajului utilizat pentru descrierea proiectului. De exemplu, dacă se utilizează limbajul *VHDL*, se pot identifica construcții sintactice specifice, care se pot înlocui cu atribute ale semnalelor şi conexiunilor pentru a indica funcțiile lor. **Figura 14(a)** prezintă o instrucțiune care testează apariția frontului crescător al semnalului *x*. Compilatorul generează inițial un graf care conține noduri pentru citirea constantelor *1* şi *stable*, ca şi noduri pentru operatorii logici *AND* şi *NOT* pentru evaluarea condiției de apariție a frontului crescător (**Figura 14(b)**). Cercurile care conțin cifre în grafurile fluxului de date indică semnale. Deoarece trebuie să se indice un test al apariției unui front crescător al semnalului *x*, transformarea grafului colectează atributele pentru modificarea

semnalului (de exemplu, *sensitivity* şi *signal change*) şi ataşează aceste atribute arcului de ieşire al operației *READ* (semnalul 7 din **Figura 14(c)**).

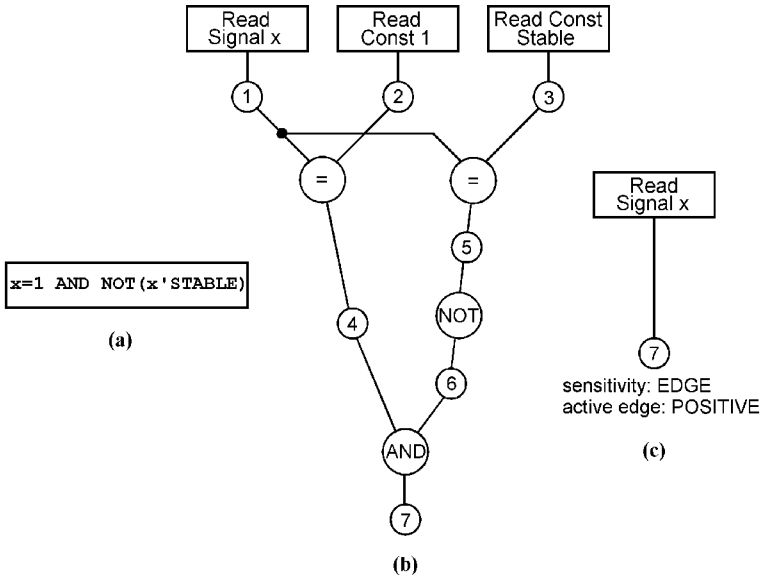


Figura 14. Înlocuirea construcțiilor sintactice specifice: (a) instrucțiune pentru detectarea frontului crescător al semnalului *x*; (b) graf inițial; (c) graf transformat.

2.6.2. Transformări ale grafurilor

Este posibilă efectuarea unor transformări la nivelul grafurilor fluxului de date şi de control pentru a îmbunătăți paralelismul implementării. Anumite transformări ale grafurilor sunt independente de metoda de reprezentare, în timp ce altele sunt aplicabile unor metode de reprezentare specifice. Se prezintă mai multe tipuri de transformări ale grafurilor: reducerea înălțimii arborilor, transformarea fluxului de control într-un flux de date, aplatizarea grafurilor fluxului de control şi de date, expandarea buclelor, expandarea construcțiilor condiționale.

Reducerea înălțimii arborilor. Această transformare utilizează proprietățile de comutativitate şi distributivitate a operatorilor limbajului în cazul unor expresii lungi, şi pune în evidență paralelismul potențial dintr-un graf complex al fluxului de date. În **Figura 15** se prezintă un exemplu de reducere a înălțimii arborilor, aplicat unei instrucțiuni de asignare cu 6 operatori. Arborele sintactic generat inițial de compilator are înălțimea 6, şi nu indică paralelismul potențial din cadrul acestei instrucțiuni de asignare. După reducerea înălțimii arborelui, rezultă un arbore cu înălțimea 3 în care mai multe operații pot fi executate în paralel.

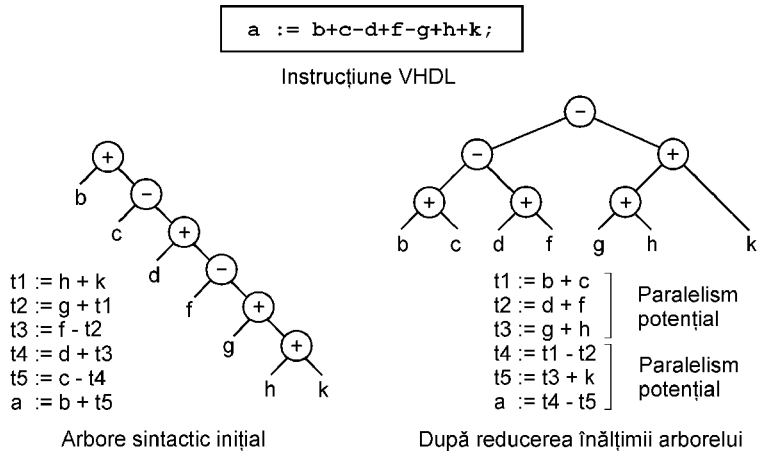


Figura 15. Reducerea înălțimii arborilor sintactici ai expresiilor.

Transformarea fluxului de control într-un flux de date. Această transformare este specifică modelului de reprezentare GFCD, şi este utilă atunci când se testează variabile booleene în cadrul fluxului de control. Deoarece fiecare test condițional din fluxul de control determină crearea unei noi stări în timpul sintezei, rezultă mai multe stări false corespunzătoare testării variabilelor booleene. Aceste stări pot fi eliminate prin transformarea testelor booleene din fluxul de control într-o reprezentare sub forma unui flux de date. **Figura 16** ilustrează această transformare pentru o instrucțiune `if`. **Figura 16(b)** prezintă reprezentarea sub forma fluxului de control pentru descrierea din **Figura 16(a)**. Un avantaj suplimentar al acestei transformări este creşterea paralelismului explicit în grafurile fluxului de date. Deoarece blocurile fluxului de date din **Figura 16(b)** sunt disjuncte, sistemele de sinteză nu pot exploata în totalitate paralelismul între blocurile de instrucțiuni mutual exclusive *B1* şi *B2*. Conversia acestei reprezentări a fluxului de control în reprezentarea echivalentă a fluxului de date, ilustrată în **Figura 16(c)**, pune în evidență în mod explicit paralelismul din cadrul unui singur graf al fluxului de date.

Aplatizarea grafurilor fluxului de control şi de date. Dacă descrierea inițială este ierarhică, reprezentarea prin modelul GFCD poate fi aplatizată în mod recursiv într-un graf al fluxului de date. Această transformare ierarhică este utilă în special atunci când toate testele fluxului de control se execută asupra variabilelor booleene; în final se pot elimina mai multe stări fictive. **Figura 17(a)** prezintă un model GFCD ierarhic. Se poate aplica în mod recursiv transformarea fluxului de control într-un flux de date pentru acest graf, pentru a se obține un singur graf al fluxului de date, după cum se ilustrează în **Figura 17(b)** şi **(c)**.

Această transformare corespunde expandării modelelor. Utilizarea modelelor structurate, care conțin subrutine şi funcții, este utilă deoarece conduce la modularitate şi la posibilitatea reutilizării modelelor. Modularitatea permite punerea în evidență a unui anumit task. De multe ori, modelele sunt apelate o singură dată. Prin expandare se aplati-

zează ierarhia de apel a modelului. Un avantaj este că domeniul de aplicare al unor tehnici de optimizare (la diferite nivele) este lărgit, permițând obținerea unui circuit final mai performant. În cazul unor apeluri multiple ale modelului, o expandare completă conduce la o creştere a dimensiunii codului intermediar şi la pierderea probabilă a posibilității de partajare a unităților hardware.

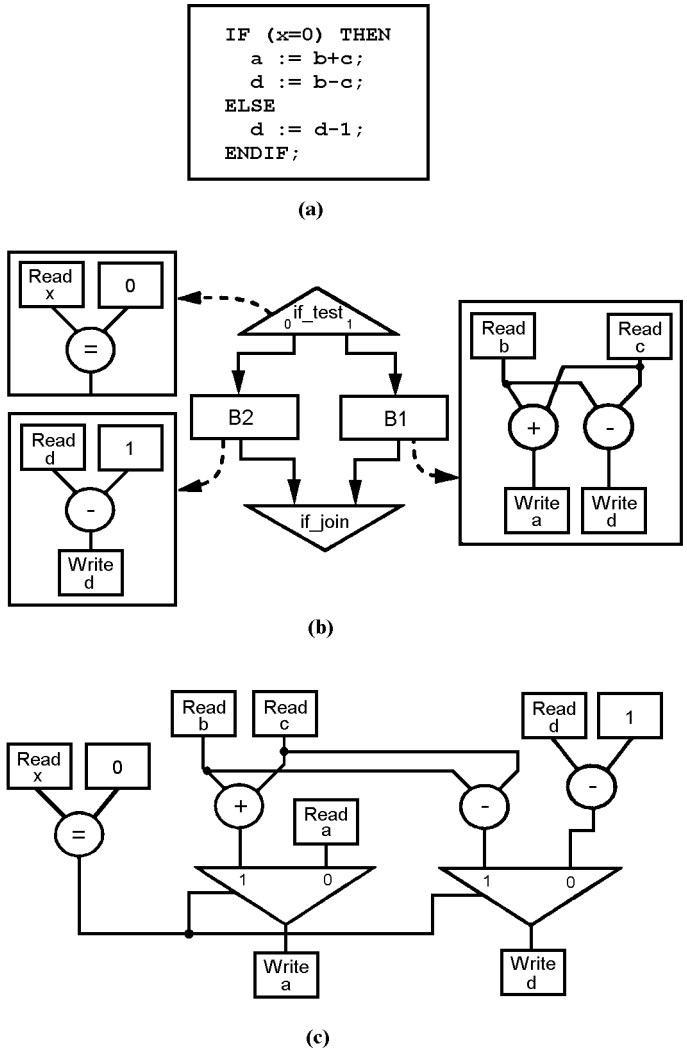


Figura 16. Transformarea fluxului de control în flux de date: (a) instrucțiune *IF*; (b) reprezentare inițială a fluxului de control; (c) reprezentare transformată a fluxului de date.

Expandarea buclelor. În cadrul modelului GFCD, construcțiile de buclare sunt reprezentate în graful fluxului de control. Buclele cu un număr fix de iterații pot fi expandate pentru a se genera grafuri de dimensiuni mai mari ale fluxului de date. Avantajul constă în extinderea domeniului pentru alte transformări. Dacă însă numărul de iterații al unei bucle nu este fixat la compilare, bucla nu poate fi expandată complet. În asemenea

cazuri, nu se poate transforma întregul graf într-un singur bloc al fluxului de date, ca în **Figura 17**. Totuşi, pot fi utilizate tehnici de expandare parțială a buclelor pentru a se creşte gradul de paralelism.

Considerăm următorul fragment de cod:

```
x = 0; for (i = 1; i <= 3; i++) {x = x+a[i];}
```

Bucla poate fi expandată astfel:

```
x = 0; x = x+a[1]; x = x+a[2]; x = x+a[3];
```

şi apoi poate fi transformată prin propagare astfel:

```
x = a[1]+a[2]+a[3]
```

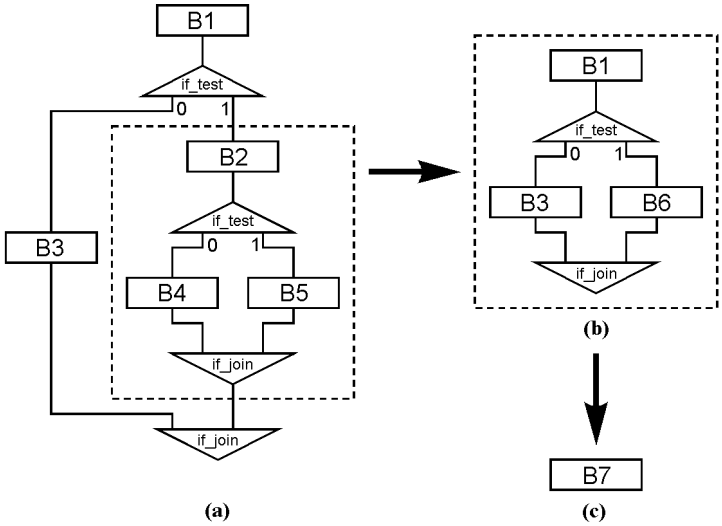


Figura 17. Aplatizarea grafului fluxului de control: graful inițial; (b) după aplatizarea grafului instrucțiunii *IF* interioare; (c) blocul final generat.

Expandarea construcțiilor condiționale. O construcție condițională poate fi transformată întotdeauna într-o construcție paralelă cu un test la sfârşit. În unele cazuri această transformare poate creşte performanțele circuitului, de exemplu atunci când clauza condițională depinde de anumite semnale care vor fi generate cu o anumită întârziere. Această transformare exclude însă anumite posibilități pentru partajarea unităților hardware, deoarece trebuie executate operațiile din toate ramurile construcției condiționale.

Un caz special este cel al construcțiilor condiționale ale căror clauze şi ramuri reprezintă evaluarea unor funcții logice. În acest caz, expandarea este avantajoasă deoarece permite extinderea domeniului de aplicare al optimizării logice.

Considerăm următorul fragment:

```
y = ab; if (a) {x = b+d;} else {x = bd;}
```

Instrucțiunea condițională poate fi transformată în:

```
x = a(b+d) + a'bd;
```

şi poate fi rescrisă sub forma:

```
y = ab; x = y + d(a+b);
```

2.6.3. Transformări specifice unităților hardware

Asupra reprezentării intermediare pot fi aplicate transformări hardware la nivelul logic, RT şi sistem. În general, acestea sunt transformări locale care utilizează proprietățile unităților hardware la diferite nivele de proiectare pentru a optimiza reprezentarea intermediară.

La *nivelul logic*, se pot aplica tehnici de optimizare booleană asupra reprezentării intermediare. Aceste transformări utilizează proprietățile algebrei booleene pentru a optimiza local părți ale unui graf prin potrivirea modelelor şi înlocuire. **Figura 18** prezintă un graf care poate fi redus la un simplu transfer de date, deoarece funcția booleană echivalentă se reduce la o singură variabilă booleană. Operațiile care compară anumite valori cu zero sau cu alte constante pot fi de asemenea optimizate în acest mod.

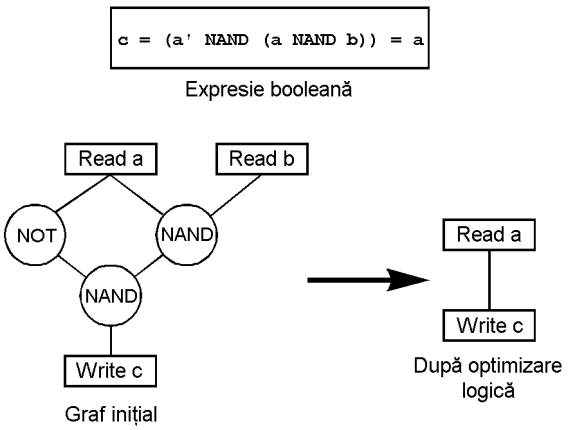


Figura 18. Transformări ale grafului la nivel logic.

La *nivelul RT*, se pot înlocui porțiuni ale unui graf cu segmente mai simple ale grafului. Aceste transformări de potrivire a modelelor se bazează pe semantica RT a componentelor hardware corespunzătoare operatorilor grafului. Transformările pot fi extinse pentru a recunoaşte segmente întregi ale grafului care corespund unei unități funcționale de la nivelul RT. În **Figura 19(a)** se prezintă un exemplu simplu în care semnalele *a* şi *b* sunt adunate şi rezultatul este incrementat. Dacă biblioteca de module RT conține o unitate funcțională care combină o adunare cu o incrementare, se poate înlocui segmentul grafului cu operația *add_inc* indicată în **Figura 19(b)**. Similar, se pot efectua transformări pentru conversia înmulțirii sau împărțirii cu puteri ale lui 2 în operații de deplasare.

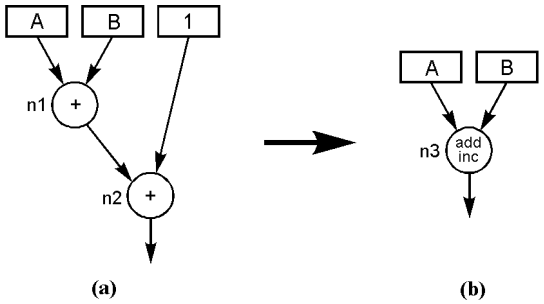


Figura 19. Transformare a unui graf simplu la nivel RT: (a) graf inițial; (b) nod funcțional specific unei biblioteci.

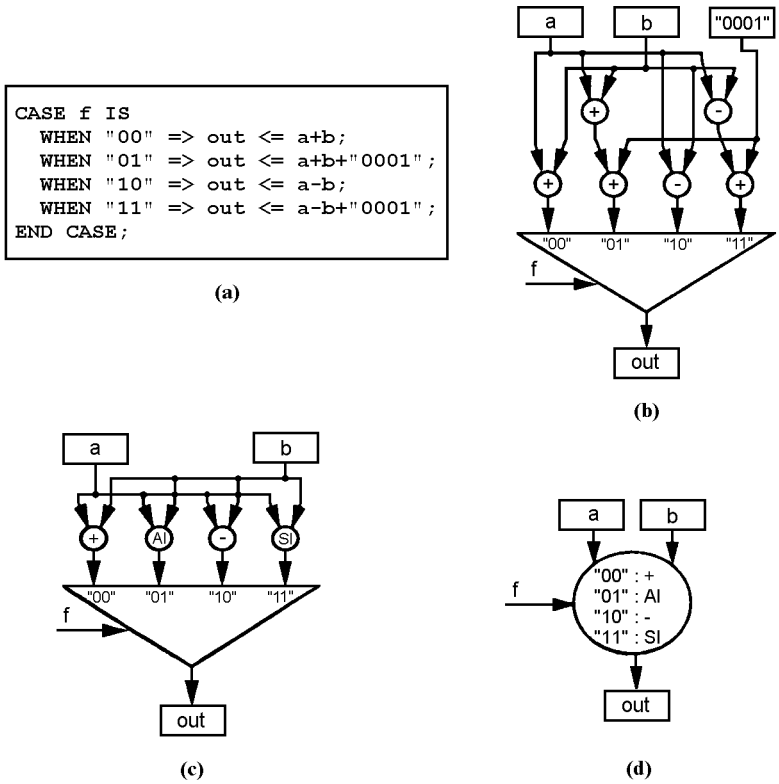


Figura 20. Recunoaşterea unei funcții complexe: (a) descriere *VHDL*; (b) reprezentare inițială a fluxului de date; (c) reprezentare simplificată cu operatorii "*Add_Inc*" şi "*Sub_Inc*"; (d) reprezentare finală cu un operator UAL.

Se pot utiliza o serie de astfel de transformări pentru a reuni nodurile unui graf în unități funcționale tipice la nivelul RT, ca unitățile aritmetice şi logice. **Figura 20(a)** prezintă un fragment de cod în limbajul *VHDL*, în care semnalului *out* i se asignează o valoare diferită pe baza valorii semnalului *f*. **Figurile 20(b)**, **(c)** şi **(d)** indică modul în care reprezentarea codului *VHDL* sub forma unui graf al fluxului de date este transformată progresiv într-un nod multifuncțional care corespunde unei unități funcționale de tip sumator-scăzător.

Aceste optimizări sunt utile în mod special atunci când biblioteca de unități funcționale RT conține unități multifuncționale complexe ca unitățile aritmetice şi logice. În asemenea cazuri, această serie de transformări poate fi considerată ca o fază locală de preprocesare pentru alocare, unde grupuri de operații corespunzătoare unei componente RT complexe sunt transformate într-un nod funcțional complex; prin alocare se poate efectua maparea directă a nodului funcțional la componenta RT.

La *nivelul sistem*, transformările pot fi utilizate pentru a diviza părți ale grafului în procese separate care se execută concurent sau în mod pipeline. De asemenea, se pot partiționa părți ale grafului în blocuri funcționale care vor fi implementate în cipuri separate sau în partiții fizice.

3. Planificarea operațiilor

3.1. Introducere

O descriere funcțională specifică secvența de operații executate de circuitul care trebuie implementat. Această descriere se compilează de obicei într-o reprezentare internă, de exemplu sub forma unui graf al fluxului de control şi de date (GFCD), care conține toate dependențele fluxului de control şi de date ale descrierii funcționale date. Algoritmii de planificare partiționează apoi această reprezentare în subgrafuri, astfel încât fiecare subgraf să fie executat într-un pas de control. Fiecare pas de control corespunde unei stări a automatului cu stări finite controlat. În reprezentarea GFCD a circuitului de înmulțire (**Figura 4**) liniile întrerupte definesc limitele paşilor de control, iar operațiile dintre două linii adiacente sunt executate în acelaşi pas de control.

În cadrul unui pas de control, este necesară o unitate funcțională separată pentru a executa fiecare operație asignată pasului respectiv. Astfel, numărul total de unități funcționale necesare într-un pas de control corespunde cu numărul de operații planificate în acest pas. Dacă sunt planificate mai multe operații în fiecare pas de control, sunt necesare mai multe unități funcționale, ceea ce conduce la un număr mai redus de paşi de control necesari pentru implementare. Pe de altă parte, dacă sunt planificate mai puține operații în fiecare pas de control, sunt suficiente mai puține unități funcționale, dar este necesar un număr mai mare de paşi de control.

Planificarea este o operație importantă în sinteza de nivel înalt deoarece influențează compromisul între performanță şi cost. Algoritmii de planificare trebuie adaptați în funcție de diferitele arhitecturi utilizate pentru implementare. De exemplu, un algoritm de planificare trebuie reformulat dacă arhitectura utilizată este de tip pipeline. Tipul unităților funcționale şi de memorie utilizate şi topologiile de interconectare influențează de asemenea formularea algoritmilor de planificare.

Diferitele construcții de limbaj influențează de asemenea algoritmii de planificare. Descrierile funcționale care conțin construcții condiționale şi de buclare necesită tehnici mai complexe de planificare. Similar, trebuie utilizate tehnici sofisticate de planificare dacă descrierea conține tablouri multidimensionale.

Operațiile de planificare şi de alocare a unităților sunt interdependente. Caracterizarea calității unui algoritm de planificare dat este dificilă fără considerarea algoritmilor care execută alocarea. Două operații diferite de planificare cu acelaşi număr de paşi de control şi care necesită acelaşi număr de unități funcționale pot avea ca rezultat proiecte cu măsuri calitative substanțial diferite după executarea alocării.

În continuare se va introduce problema planificării prin prezentarea unor algoritmi fundamentali de planificare pentru o arhitectură simplificată, utilizând o descriere simplă a proiectului. Apoi se vor descrie extensii ale algoritmilor fundamentali pentru modele de descriere mai complexe şi biblioteci realiste de unități funcționale.

3.2. Algoritmi fundamentali de planificare

Pentru a simplifica discutarea algoritmilor fundamentali de planificare, se vor presupune următoarele restricții:

- descrierile funcționale nu conțin construcții condiționale sau bucle;
- fiecare operație necesită pentru execuție un singur pas de control;
- fiecare tip de operație poate fi executat printr-un singur tip de unitate funcțională.

Aceste restricții vor fi eliminate atunci când se vor considera modele mai realiste.

Se pot defini două scopuri diferite ale problemei de planificare, fiind dată o bibliotecă de unități funcționale cu caracteristici cunoscute (de exemplu, dimensiuni, întârzieri, putere consumată), şi lungimea unui pas de control. În primul rând, se poate minimiza numărul de unități funcționale pentru un număr fix al paşilor de control; aceasta reprezintă *planificarea cu restricții de timp*. În al doilea rând, se poate minimiza numărul paşilor de control pentru un cost de proiectare dat; acest cost poate fi măsurat sub forma numărului de unități funcționale şi de memorie, numărul de porți ŞI-NU cu două intrări, sau spațiul necesar pe cip. Această abordare reprezintă *planificarea cu restricții de resurse*.

Pentru prezentarea algoritmilor se introduc următoarele definiții. Un graf al fluxului de date (GFD) este un graf direcționat aciclic $G(V, E)$, unde V este un set de noduri şi E un set de muchii. Fiecare nod $v_i \in V$ reprezintă o operație o_i din descrierea funcțională. În setul E există o muchie direcționată $e_{i\,j}$ de la nodul $v_i \in V$ la nodul $v_j \in V$ dacă data produsă de operația o_i (reprezentată prin v_i) este consumată de operația o_j (reprezentată prin v_j). În acest caz, v_i este un predecesor imediat al nodului v_j. Setul tuturor predecesorilor imediați ai nodului v_i este notat cu $Pred_{vi}$. Similar, v_j este un succesor imediat al nodului v_i. Setul tuturor succesorilor imediați ai nodului v_i este notat cu $Succ_{vi}$. Fiecare operație o_i poate fi executată în d_i paşi de control. Deoarece s-a presupus că fiecare operație necesită un singur pas de control, d_i are valoarea 1 pentru fiecare operație o_i.

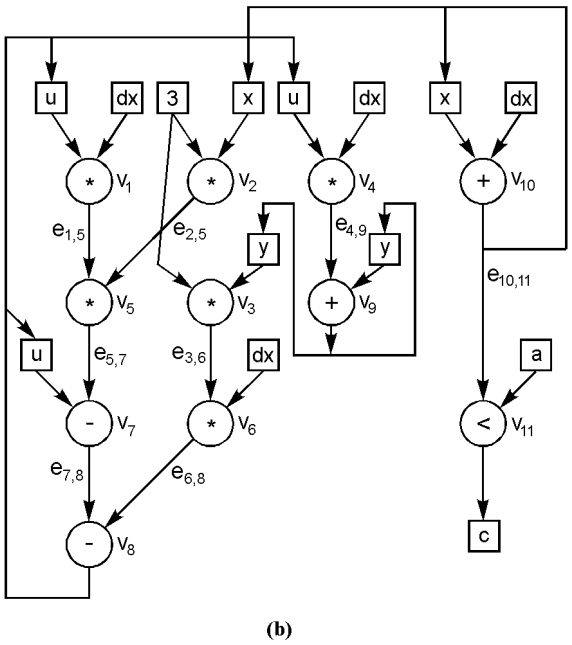
Aceste definiții se ilustrează printr-un exemplu de descriere al unui circuit pentru determinarea soluțiilor ecuației diferențiale:

$$y'' + 3xy' + 3y = 0$$

prin metoda Euler, în intervalul $[0, a]$, cu pasul dx şi valorile inițiale $x(0) = x$; $y(0) = y$; $y'(0) = u$. **Figura 21(a)** prezintă descrierea textuală, iar **Figura 21(b)** prezintă graful fluxului de date, care este format din 11 noduri, $V = \{v_1, v_2, ..., v_{11}\}$ şi 8 muchii, $E = \{e_{1,5}, e_{2,5}, e_{5,7}, e_{7,8}, e_{3,6}, e_{6,8}, e_{4,9}, e_{10,11}\}$.

```
WHILE (x < a) DO
  x1 := x + dx;
  u1 := u - (3*x*u*dx) - (3*y*dx);
  y1 := y + (u*dx);
  x := x1; u := u1; y := y1;
ENDWHILE
```

(a)



(b)

Figura 21. Exemplu pentru rezolvarea unei ecuații diferențiale: (a) descriere textuală; (b) GFD.

Graful fluxului de date pune în evidență paralelismul din cadrul proiectului. Există o anumită flexibilitate în privința stării în care poate fi planificat fiecare nod al grafului. Algoritmii de planificare necesită de obicei specificarea limitelor în timp în care operațiile trebuie planificate. Prima stare căreia i se poate asigna un nod este numită valoarea *ASAP* a acestuia. Această valoare este determinată prin algoritmul de planificare *ASAP* prezentat în **Figura 22**.

Algoritmul de planificare *ASAP* asignează o etichetă E_i (un index al pasului de control) fiecărui nod v_i al unui graf al fluxului de date, planificând operația o_i în pasul de control cel mai apropiat în timp s_{Ei}. Funcția *NODURI_PLANIF*($Pred_{vi}$, E) returnează valoarea *TRUE* dacă toate nodurile din setul $Pred_{vi}$ sunt planificate (deci toți predecesorii imediați ai nodului v_i au o etichetă E diferită de zero). Funcția *MAX*($Pred_{vi}$, E) returnează indexul nodului cu valoarea E maximă din setul de predecesori ai nodului v_i.

```
for fiecare nod v_i ∈ V do
    if Pred_vi = φ then
        E_i = 1;
        V = V - {v_i};
    else
        E_i = 0;
    endif
endfor
while V ≠ φ do
    for fiecare nod v_i ∈ V do
        if NODURI_PLANIF(Pred_vi, E) then
            E_i = MAX(Pred_vi, E) + 1;
            V = V - {v_i};
        endif
    endfor
endwhile
```

Figura 22. Algoritmul de planificare *ASAP*.

Bucla **for** a algoritmului inițializează valoarea *ASAP* a tuturor nodurilor grafului. Nodurile care nu au nici un predecesor sunt asignate la starea s_1, iar celelalte sunt asignate la starea s_0. În fiecare iterație, bucla **while** determină nodurile care au toți predecesorii planificați, şi le asignează la starea cea mai apropiată în timp care este posibilă. Deoarece s-a presupus că întârzierea introdusă de fiecare operație este de un pas de control, starea cea mai apropiată în timp este calculată prin ecuația $E_i = MAX(Pred_{vi}, E) + 1$.

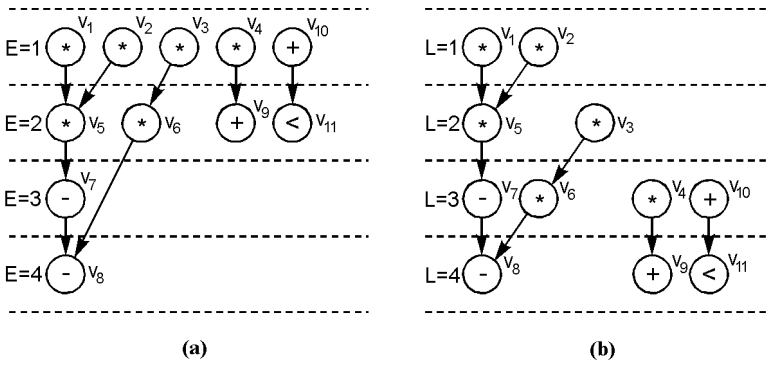


Figura 23. Planificarea pentru exemplul circuitului de rezolvare a ecuației diferențiale: (a) planificarea *ASAP*; (b) planificarea *ALAP*.

Figura 23(a) prezintă rezultatele algoritmului de planificare *ASAP* pentru exemplul din **Figura 21**. Operațiile o_1, o_2, o_3, o_4 şi o_{10} sunt asignate pasului de control s_1, deoarece nu au nici un predecesor. Operațiile o_5, o_6, o_9 şi o_{11} sunt asignate pasului de control s_2, iar operațiile o_7 şi o_8 sunt asignate paşilor de control s_3, respectiv s_4.

Valoarea *ALAP* a unui nod defineşte starea cea mai depărtată în timp în care poate fi planificat nodul respectiv. Această valoare poate fi determinată prin algoritmul de planificare din **Figura 24**. Fiind dată o restricţie de timp de T paşi de control, algoritmul determină pasul de control cel mai depărtat în timp în care trebuie să înceapă execuţia unei operaţii. Algoritmul asignează o etichetă *ALAP* L_i fiecărui nod v_i din graful fluxului de date, planificând astfel operaţia o_i în pasul de control cel mai depărtat în timp s_{Li}. Funcţia *NODURI _PLANIF(*$Succ_{vi}$*, L)* returnează valoarea *TRUE* dacă toate nodurile notate cu $Succ_{vi}$ sunt planificate (deci dacă toţi succesorii imediaţi ai nodului v_i au o etichetă L diferită de zero). Funcţia *MIN(*$Succ_{vi}$*, L)* returnează indexul nodului cu valoarea L minimă din setul de noduri succesoare ale nodului v_i.

```
for fiecare nod v_i ∈ V do
    if Succ_vi = φ then
        L_i = T;
        V = V - {v_i};
    else
        L_i = 0;
    endif
endfor
while V ≠ φ do
        for fiecare nod v_i ∈ V do
                if NODURI_PLANIF(Succ_vi, L) then
                        L_i = MIN(Succ_vi, L) - 1;
                        V = V - {v_i};
                endif
        endfor
endwhile
```

Figura 24. Algoritmul de planificare *ALAP*.

Bucla **for** a algoritmului iniţializează valoarea *ALAP* a tuturor nodurilor din graful fluxului de date. Nodurile care nu au nici un succesor sunt asignate la ultima stare posibilă, iar celelalte sunt asignate la starea s_0. În fiecare iteraţie, bucla **while** determină nodurile care au toţi succesorii planificaţi şi le asignează pe acestea la ultima stare posibilă.

Figura 23(b) prezintă rezultatele algoritmului de planificare *ALAP* (unde $T = 4$), pentru exemplul din **Figura 21**. Operaţiile o_8, o_9 şi o_{11} sunt asignate la ultimul pas de control s_4, deoarece acestea nu au nici un succesor. Operaţiile o_4, o_6, o_7 şi o_{10} sunt asignate pasului s_3, iar operaţiile o_3 şi o_5 sunt asignate pasului s_2. Celelalte operaţii o_1 şi o_2 sunt asignate pasului de control s_1.

Pe baza planificării finale, se poate calcula numărul de unităţi funcţionale care sunt necesare pentru implementare. Numărul maxim de operaţii din fiecare stare indică numărul de unităţi funcţionale de un anumit tip. În cazul planificării *ASAP* din **Figura 23(a)** numărul maxim de operaţii de înmulţire planificate în oricare pas de control este 4 (în starea s_1), deci sunt necesare 4 circuite de înmulţire. În plus, pentru această planificare mai este necesar un sumator/scăzător şi un comparator. În cazul planificării *ALAP* din **Fi-**

gura 23(b), numărul maxim de operații de înmulțire planificate în oricare pas de control este 2 stările s_1, s_2 şi s_3), fiind suficiente două circuite de înmulțire. În plus, mai este necesar un sumator, un scăzător şi un comparator.

3.3. Planificarea cu restricții de timp

Planificarea cu restricții de timp este importantă pentru sistemele destinate aplicațiilor de timp real. De exemplu, în cazul sistemelor de procesare a semnalelor digitale (DSP), rata de eşantionare a şirului datelor de intrare determină timpul maxim disponibil pentru execuția unui algoritm DSP asupra eşantionului curent al datelor, înaintea preluării eşantionului următor. Deoarece rata de eşantionare este fixă, scopul principal este de a minimiza costul unităților hardware. Fiind dată durata pasului de control, rata de eşantionare poate fi exprimată în funcție de numărul paşilor de control necesari pentru execuția algoritmului DSP.

Algoritmii de planificare cu restricții de timp poate utiliza trei tehnici diferite: programarea matematică, euristici constructive şi rafinarea iterativă. Se va prezenta metoda programării liniare (ca exemplu de programare matematică), o metodă euristică constructivă şi o tehnică de planificare iterativă.

3.3.1. Metoda programării liniare

Această metodă determină o planificare optimă utilizând un algoritm de căutare de tip "branch-and-bound" care implică revenirea, unele decizii luate în etapele inițiale ale algoritmului fiind reevaluate pe măsură ce căutarea avansează. Fie s_{E_k} şi s_{L_k} paşii de control în care este planificată operația o_k prin algoritmul *ASAP*, respectiv *ALAP*. În cazul unei planificări fezabile, execuția operației o_k trebuie să înceapă într-un pas de control care nu este mai apropiat în timp decât s_{E_k} şi nu este mai târziu decât s_{L_k}, deoarece $E_k \le L_k$.

Numărul paşilor de control între s_{E_k} şi s_{L_k} reprezintă domeniul de mobilitate al operației o_k:

$$dom\_mob(o_k) = \{s_j \mid E_k \le j \le L_k\}$$

Figura 25(a) indică domeniul fiecărei operații din GFD pentru exemplul circuitului de rezolvare a ecuației diferențiale, calculat pe baza etichetelor *ASAP* şi *ALAP* din **Figura 23**. De exemplu, domeniul operației o_4 este $\{s_1, s_2, s_3\}$, deoarece etichetele sale *ASAP* şi *ALAP* sunt $E_4 = 1$ şi $L_4 = 3$.

Se pot utiliza valorile *ASAP*, *ALAP* şi domeniul de mobilitate al operațiilor pentru a formula problema planificării prin metoda programării liniare. În continuare se descriu notațiile utilizate pentru formularea problemei.

Fie $OP = \{o_i \mid 1 \le i \le n\}$ setul operațiilor din graf şi $t_i = tip(o_i)$ tipul fiecărei operații o_i. Fie $T = \{t_k \mid 1 \le k \le m\}$ setul tipurilor posibile de operații. Setul OP_{t_k} constă din operațiile din setul OP care au tipul t_k, deci $OP_{t_k} = \{o_i \mid o_i \in OP \wedge tip(o_i) = t_k\}$. Fie $INDEX_{t_k}$ setul indicilor operațiilor din OP_{t_k} : $INDEX_{t_k} = \{i \mid o_i \in OP_{t_k}\}$. Fie N_{t_k} numărul

de unități care execută operația t_k şi fie C_{t_k} costul unei asemenea unități. Fie $S = \{sj \mid 1 \le j \le r\}$ setul paşilor de control disponibili pentru planificarea operațiilor. Fie $x_{i,j}$ variabile întregi, care au valoarea 1 dacă operația o_i este planificată în pasul s_j, şi 0 în caz contrar.

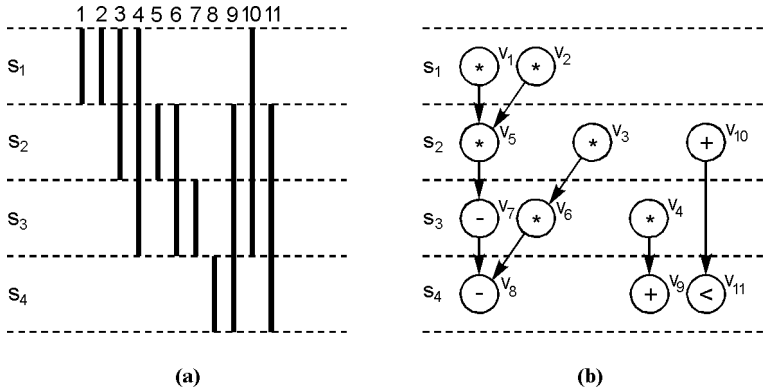


Figura 25. Exemplu de planificare prin metoda programării liniare: (a) domeniul operațiilor; (b) planificarea finală.

Problema planificării poate fi formulată după cum urmează. Să se minimizeze expresia:

$$\sum_{k=1}^{m}(C_{t_k} \times N_{t_k}) \tag{3.1}$$

cu următoarele ipoteze:

$$\forall i,\ 1 \le i \le n,\ (\sum_{E_i \le j \le L_i} x_{i,j} = 1), \tag{3.2}$$

$$\forall j,\ 1 \le j \le r,\ \forall k,\ 1 \le k \le m,\ (\sum_{i \in INDEX_{tk}} x_{i,j} \le N_{t_k}), \tag{3.3}$$

$$\forall i, j, o_i \in Pred_{o_j},\ (\sum_{E_i \le k \le L_i}(k \times x_{i,k}) - \sum_{E_j \le l \le L_j}(l \times x_{j,l}) \le -1\ ). \tag{3.4}$$

Funcția (3.1) minimizează costul total al unităților funcționale necesare. Condiția (3.2) necesită ca fiecare operație o_i să fie planificată într-un singur pas de control, nu mai repede decât E_i şi nu mai târziu decât L_i. Condiția (3.3) asigură ca nici un pas de control să nu conțină operații de tipul t_k într-un număr mai mare decât N_{t_k}. În sfârşit, condiția (3.4) garantează că pentru o operație o_j, toți predecesorii acestuia ($Pred_{o_j}$) sunt planificați într-un pas de control anterior. Cu alte cuvinte, dacă $x_{i,k} = x_{j,l} = 1$, atunci $k < l$.

Pentru ilustrarea formulării metodei, se va utiliza graful fluxului de date din **Figura 21(b)**, realizând planificarea în 4 paşi de control. Deoarece graful conține patru tipuri diferite de operații (înmulțire, adunare, scădere şi comparație), sunt necesare patru tipuri de unități funcționale din bibliotecă. Fie C_m, C_a, C_s şi C_c costurile unui circuit de înmulțire, unui sumator, unui scăzător, respectiv a unui comparator. Fie N_m, N_a, N_s şi N_c numărul circuitelor de înmulțire, a sumatoarelor, a scăzătoarelor, respectiv a comparatoarelor necesare pentru planificarea finală. Formularea problemei de planificare în patru paşi de

control prin metoda programării liniare pentru graful din **Figura 21(b)** este: Să se minimizeze expresia:

$$C_m \times N_m + C_a \times N_a + C_s \times N_s + C_c \times N_c \quad (3.5)$$

cu condițiile următoare:

$x_{1,1} = 1$
$x_{2,1} = 1$
$x_{3,1} + x_{3,2} = 1$
$x_{4,1} + x_{4,2} + x_{4,3} = 1$
$x_{5,2} = 1$
$x_{6,2} + x_{6,3} = 1$
$x_{7,3} = 1$
$x_{8,4} = 1$
$x_{9,2} + x_{9,3} + x_{9,4} = 1$
$x_{10,1} + x_{10,2} + x_{10,3} = 1$
$x_{11,2} + x_{11,3} + x_{11,4} = 1$
$x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} \leq N_m$
$x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} \leq N_m$
$x_{4,3} + x_{6,3} \leq N_m$
$x_{7,3} \leq N_s$
$x_{8,4} \leq N_s$
$x_{10,1} \leq N_a$
$x_{9,2} + x_{10,2} \leq N_a$
$x_{9,3} + x_{10,3} \leq N_a$
$x_{9,4} \leq N_a$
$x_{11,2} \leq N_c$
$x_{11,3} \leq N_c$
$x_{11,4} \leq N_c$
$1x_{3,1} + 2x_{3,2} - 2x_{6,2} - 3x_{6,3} \leq -1$
$1x_{4,1} + 2x_{4,2} + 3x_{4,3} - 2x_{9,2} - 3x_{9,3} - 4x_{9,4} \leq -1$
$1x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} \leq -1$

Dacă se presupune că $C_m = 2$ şi $C_a = C_s = C_c = 1$, funcția de cost este minimizată şi toate inegalitățile sunt satisfăcute dacă valorile variabilelor sunt următoarele:

$N_m = 2$,
$N_a = N_s = N_c = 1$,
$x_{1,1} = x_{2,1} = x_{3,2} = x_{4,3} = x_{5,2} = x_{6,3} = x_{7,3} = x_{8,4} = x_{9,4} = x_{10,2} = x_{11,4} = 1$

celelalte valori ale variabilelor x fiind egale cu 0. Această soluție a problemei de planificare este prezentată în **Figura 25(b)**. În figură, o operație o_i este planificată în pasul de control s_j, dacă şi numai dacă $x_{i,j}$ are valoarea 1.

Numărul de inegalități necesare pentru formularea problemei prin această metodă creşte rapid cu numărul paşilor de control. De exemplu, dacă se creşte numărul paşilor de control cu 1, vor exista n variabile x suplimentare în cadrul inegalităților, deoarece trebuie considerat un pas de control suplimentar pentru fiecare operație. În plus, numărul inega-

lităților rezultate din condiția (3.3) va creşte cu o valoare care depinde de structura grafului respectiv. Deoarece timpul de execuție al algoritmului creşte rapid cu numărul de variabile şi numărul de inegalități, în practică metoda este aplicabilă numai pentru probleme de dimensiuni reduse.

Deoarece metoda nu este practică pentru descrieri de dimensiuni mari, au fost dezvoltate metode euristice care pot fi executate în mod eficient. Eficiența crescută a metodelor euristice este obținută prin eliminarea revenirii din metoda programării liniare. Revenirea poate fi eliminată prin planificarea operațiilor una câte una, pe baza unei criterii care determină decizii corecte în cele mai multe cazuri. În cazul acestor metode euristice, costul circuitului rezultat depinde în mare măsură de selecția următoarei operații care va fi planificată şi de asignarea acestei operații pasului de control cel mai potrivit.

3.3.2. Metoda euristică constructivă

Se va prezenta o versiune simplificată a algoritmului de planificare prin metoda euristică constructivă. Scopul principal al algoritmului este de a reduce numărul total de unități funcționale utilizate pentru implementare. Acest obiectiv este realizat prin distribuirea uniformă a operațiilor de acelaşi tip în toate stările disponibile. Această distribuire uniformă asigură faptul că unitățile funcționale alocate pentru execuția operațiilor într-un pas de control sunt utilizate în mod eficient în toți ceilalți paşi de control, ceea ce conduce la o rată ridicată de utilizare a unităților.

Ca şi metoda programării liniare, şi acest algoritm se bazează pe algoritmii de planificare *ASAP* şi *ALAP* pentru a determina domeniul paşilor de control pentru fiecare operație (deci, $dom\_mob(o_i)$). Algoritmul presupune de asemenea că fiecare operație o_i are o probabilitate uniformă de a fi planificată în oricare pas de control al domeniului, şi probabilitatea zero de a fi planificată în oricare alt pas de control. Astfel, pentru o stare dată s_j, astfel încât $E_i \le j \le L_i$, probabilitatea că operația o_i va fi planificată în această stare este dată de $p_j(o_i) = 1/(L_i - E_i + 1)$.

Aceste calcule de probabilitate se pot ilustra pentru exemplul din **Figura 21**, utilizând valorile *ASAP* (E_i) şi *ALAP* (L_i) din **Figura 23**. Probabilitățile operațiilor sunt indicate în **Figura 26(a)**. Operațiile o_1, o_2, o_5, o_7 şi o_8 au probabilitatea 1 pentru a fi planificate în paşii s_1, s_1, s_2, s_3, respectiv s_4, deoarece pentru aceste operații valoarea s_{E_i} este egală cu valoarea s_{L_i}. Lățimea unui dreptunghi din această figură reprezintă probabilitatea $(1/(L_i - E_i + 1))$ unei operații de a fi planificată în pasul de control respectiv. De exemplu, operația o_3 are o probabilitate de 0.5 de a fi planificată în pasul s_1 sau s_2. Deci, $p_1(o_3) = p_2(o_3) = 0.5$.

Pe baza valorilor probabilităților pentru fiecare operație, se crează un set de grafuri de distribuție a probabilității (grafuri de bare), fiind construit câte un graf de bare separat pentru fiecare tip de operație. Un graf de bare pentru un anumit tip de operație reprezintă costul probabil al operatorului (*CPO*) în fiecare stare. Costul probabil al operatorului în starea s_j pentru operația de tip k este dat de:

$$CPO_{j,k} = c_k * \sum_{i, s_j \in dom\_mob(o_i)} p_j(o_i)$$

unde o_i este o operație de tip k, iar c_k este costul unității funcționale care execută operația de tip k.

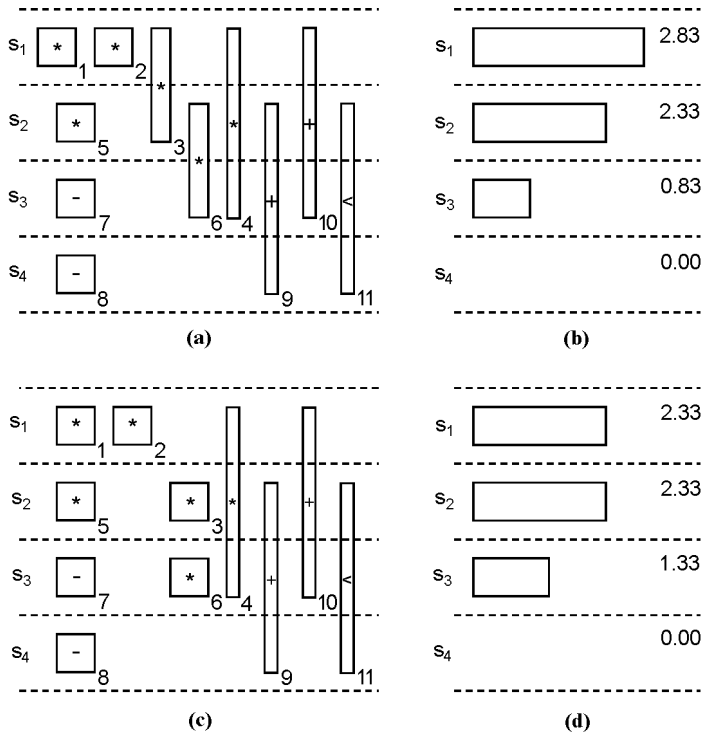


Figura 26. Exemplu de planificare prin metoda euristică constructivă: (a) probabilitatea de planificare a operațiilor în paşi de control; (b) costul operatorilor pentru operațiile de înmulțire din fig. (a); (c) probabilitatea de planificare a operațiilor în paşi de control după planificarea operației o_3 în pasul s_2; (d) costul operatorilor pentru operațiile de înmulțire din fig. (c).

Figura 26(b) prezintă grafurile de bare ale costurilor probabile pentru operația de înmulțire în fiecare pas de control. Valoarea pentru $CPO_{1,m}$ se poate calcula din relația $CPO_{1,m} = c_m \times (p_1(o_1) + p_1(o_2) + p_1(o_3) + p_1(o_4))$, care este $c_m \times (1.0 + 1.0 + 0.5 + 0.33)$, sau $2.83 \times c_m$.

Graful de bare din **Figura 26(b)** indică valorile *CPO* pentru operația de înmulțire în cele patru stări: 2.83, 2.33, 0.83, respectiv 0. Deoarece unitățile funcționale pot fi partajate între stări, maximul costurilor probabile ale unui operator pentru toate stările reprezintă o măsură a costului total al implementării tuturor operațiilor de acel tip. Pentru toate celelalte tipuri de operații se construiesc grafuri de bare similare celor din **Figura 26(b)**.

Deoarece scopul principal al algoritmului este partajarea eficientă a unităților funcționale între toate stările, se încearcă echilibrarea valorilor *CPO* pentru fiecare tip de operație. Algoritmul din **Figura 27** descrie o metodă pentru obținerea valorilor uniforme

ale costurilor probabile ale operatorilor. P_{curent} indică cea mai recentă planificare parțială. P_{temp} este o copie a planificării curente, asupra căreia se încearcă efectuarea unor asignări temporare de planificare. În fiecare iterație, variabilele *OpOpt* şi *PasOpt* păstrează operația optimă pentru care se poate realiza planificarea şi pasul de control optim pentru planificarea operației. Atunci când aceste variabile sunt determinate pentru o iterație dată, planificarea P_{curent} este modificată în mod corespunzător utilizând funcția *PLANIF_OP*(P_{curent}, o_i, s_j), care returnează o nouă planificare, după planificarea operației o_i în starea s_j din P_{curent}. Planificarea unei anumite operații într-un pas de control afectează valorile probabilităților celorlalte operații, din cauza dependenței datelor. Funcția *AJUST_DISTRIB* parcurge setul de noduri şi ajustează distribuția probabilităților pentru nodurile succesoare şi predecesoare ale grafului.

```
ASAP (V);
ALAP (V);
while există operații oi astfel încât Ei ≠ Li do
    CâştigMax = – ∞;
    /* Încearcă planificarea tuturor operațiilor */
    /* neplanificate în fiecare pas din domeniu */
    for fiecare oi , Ei ≠ Li do
        for fiecare j, Ei ≤ j ≤ Li do
            Ptemp = PLANIF_OP (Pcurent, oi, sj);
            AJUST_DISTRIB (Ptemp, oi, sj);
            if COST (Pcurent) - COST (Ptemp) > CâştigMax then
                CâştigMax = COST (Pcurent) - COST (Ptemp);
                OpOpt = oi; PasOpt = sj;
            endif
        endfor
    endfor
    Pcurent = PLANIF_OP (Pcurent, OpOpt, PasOpt);
    AJUST_DISTRIB (Pcurent, OpOpt, PasOpt);
endwhile
```

Figura 27. Algoritmul de planificare prin metoda euristică constructivă.

Funcția *COST* (*P*) evaluează costul implementării unei planificări parțiale *P* pe baza unei funcții de cost date. O funcție de cost simplă poate aduna valorile *CPO* pentru fiecare tip de operație, după cum se indică în ecuația (3.6).

$$COST\,(P) = \sum_{1 \le k \le m} \max_{1 \le j \le r} CPO_{j,k} \qquad (3.6)$$

Acest cost este calculat utilizând valorile *ASAP* (E_i) şi *ALAP* (L_i) pentru toate nodurile.

În timpul fiecărei iterații, este calculat costul asignării fiecărei operații neplanificate unor stări posibile din domeniul *dom_mob*(o_i), utilizând planificarea P_{temp}. Asignarea care conduce la costul minimal va fi acceptată, planificarea P_{curent} fiind actualizată. Astfel, în fiecare iterație este asignată o operație o_i pasului de control s_k, pentru $E_i \le k \le L_i$. Distribuția probabilităților pentru operația o_i este modificată în $p_k(o_i) = 1$ şi $p_j(o_i) =$

0 pentru fiecare j diferit de k. Operația o_i rămâne fixată şi nu este mutată în iterațiile următoare.

Pentru exemplul prezentat, sunt calculate costurile pentru asignarea fiecărei operații neplanificate paşilor de control posibili. Asignarea operației o_3 pasului de control s_2 are ca rezultat un cost probabil minim pentru înmulțire, deoarece max(p_j) scade de la 2.83 la 2.33. Această asignare este acceptată. La asignarea operației o_3 pasului de control s_2, valoarea probabilității pentru operația o_6 se modifică de asemenea, după cum se prezintă în **Figura 26(c)**. Operația o_3 nu va mai fi mutată în timpul continuării iterațiilor pentru planificarea altor operații neplanificate.

În fiecare iterație a algoritmului, este asignată o operație unui pas de control pe baza costului probabil minim al operatorului. Dacă există două asignări posibile cu costuri apropiate sau identice, algoritmul nu poate estima cu acuratețe alegerea cea mai potrivită.

Metoda se numeşte constructivă deoarece se construieşte o soluție fără a efectua nici o revenire. Decizia de a planifica o operație într-un pas de control este luată pe baza unui graf al fluxului de date parțial planificat; nu se ține cont de asignările ulterioare ale operatorilor la acelaşi pas de control. Foarte probabil, soluția rezultată nu va fi optimă, datorită lipsei unei strategii de anticipare şi a lipsei compromisurilor între deciziile inițiale şi cele întârziate.

3.3.3. Metoda de rafinare iterativă

Performanțe mai ridicate se pot obține prin replanificarea unor operații în cadrul unei planificări date. Aspectele esențiale legate de replanificare sunt: alegerea unui candidat pentru replanificare, procedura de replanificare şi controlul procesului de îmbunătățire. Se va descrie o metodă bazată pe paradigma propusă inițial pentru problema de bisecție a grafurilor de *Kernighan* şi *Lin*.

Pentru a determina o planificare inițială se poate utiliza oricare algoritm de planificare. Pot fi obținute noi planificări prin simpla replanificare a câte unei operații. O operație poate fi replanificată într-un pas mai timpuriu sau mai întârziat, cât timp nu se violează restricțiile legate de dependența datelor.

De exemplu, există cinci mutări posibile în cazul planificării inițiale din **Figura 28(a)**, indicate prin arcele cu linii întrerupte (o_6 în s_3, o_9 în s_3, o_9 în s_4, o_{11} în s_3 şi o_{11} în s_4). Fiecare mutare conduce la o nouă planificare. De exemplu, prin mutarea operației o_6 în starea s_3 rezultă noua planificare din **Figura 28(b)**. Mutările ulterioare posibile sunt indicate prin arce cu linii întrerupte în **Figura 28(b)**. O operație care a fost mutată (de exemplu, o_6) este blocată, şi nu mai poate fi mutată din nou.

După o secvență de $m \leq n$ mutări şi blocări, toate operațiile devin blocate. Se poate determina o subsecvență de $k \leq m$ mutări şi blocări care determină un câştig cumulativ maxim, câştigul fiind definit ca scăderea costului de implementare a noii planificări. Dacă câştigul nu este negativ (deci costul scade), se modifică planificarea inițială pe baza primelor k mutări, se blochează toate operațiile şi se reiterează procesul de rafinare. Dacă nu se poate obține un câştig cumulativ pozitiv, procesul se termină.

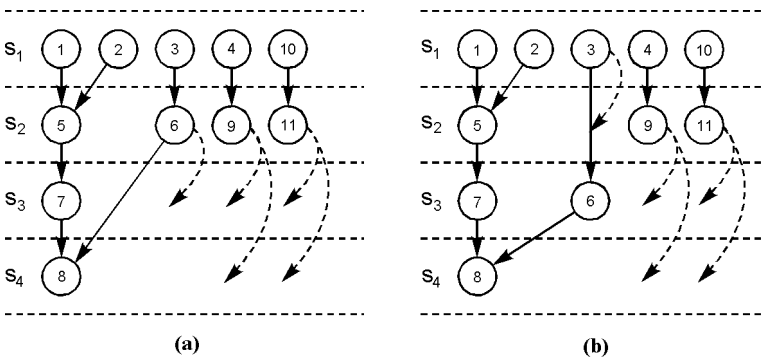


Figura 28. Replanificarea iterativă: (a) planificare inițială în care trei operații pot fi mutate în cinci paşi de control (săgeţile cu linii întrerupte); (b) după mutarea şi blocarea operației 6.

În algoritmul de replanificare iterativă (**Figura 29**), variabila P_{curent} păstrează copia actualizată cel mai recent a planificării. P_{temp} şi P_m sunt versiuni temporare ale planificării, utilizate pentru evaluarea diferitelor opțiuni de replanificare. Funcția *PLANIF_OP* (P_{temp}, o_i, s_j) returnează o nouă planificare, după planificarea operației o_i în starea s_j din P_{temp}. Funcția *COST* (*P*) calculează costul unei anumite planificări. Acest cost poate fi determinat din ecuația (3.6). Setul *OpDebloc* este un set de noduri care rămân deblocate într-o anumită iterație şi astfel pot fi replanificate în iterațiile viitoare. Tabloul *O*[*m*] păstrează secvența de operații care sunt mutate în fiecare iterație, iar tabloul *S*[*m*] păstrează stările în care va fi mutată fiecare operație din *O*[*m*]. Funcția *CÂŞTIG_CUM_MAX* parcurge tabloul *C*[*m*] şi returnează câştigul cumulativ maxim posibil. Variabila *CâştigMaxIndex* memorează numărul de mutări necesare pentru a obține câştigul cumulativ maxim.

Bucla **while** a algoritmului determină secvența mutărilor celor mai bune până când toate operațiile devin blocate. Câştigul maxim pentru fiecare mutare este memorat în tabloul *C*[*m*]. Funcțiile *CÂŞTIG_CUM_MAX* şi *CÂŞTIG_CUM_MAX_INDEX* parcurg tabloul *C*[*m*] şi extrag sub-secvența de mutări care determină un câştig cumulativ maxim. Planificarea P_{curent} este actualizată pentru a reflecta modificările produse de aceste mutări.

Algoritmul *Kernighan-Lin* este util în domeniul proiectării fizice. Utilitatea sa pentru planificare poate fi crescută prin încorporarea unor îmbunătățiri propuse pentru problema de bipartiționare a grafurilor. Două dintre acestea sunt prezentate în continuare.

1. Prima îmbunătățire este de natură probabilistică şi ține cont de faptul că rezultatele obținute prin metoda *Kernighan-Lin* depind în mare măsură de soluția inițială. Deoarece algoritmul este eficient din punct de vedere computațional, poate fi rulat de mai multe ori, de fiecare dată cu o soluție inițială diferită, alegându-se soluția cea mai bună.

```
repeat
    P_temp = P_curent; m = 0;
    OpDebloc = {Toate operaţiile din P_temp};
    while OpDebloc ≠ ϕ do
        m = m + 1; C[m] = –∞;
        for fiecare operaţie o_i ∈ P_temp do
            for fiecare destinaţie posibilă s_j a operaţiei o_i do
                P_m = PLANIF_OP (P_temp, o_i, s_j);
                Câştig = COST(P_temp) - COST(P_m);
                if Câştig > C[m] then
                    O[m] = i; S[m] = j; C[m] = Câştig;
                endif
            endfor
        endfor
        P_temp = PLANIF_OP (P_temp, o_O[m], s_S[m]);
        OpDebloc = OpDebloc - {o_O[m]};
    endwhile
    CâştigMax = CÂŞTIG_CUM_MAX(C);
    CâştigMaxIndex = CÂŞTIG_CUM_MAX_INDEX(C);
    if CâştigMax ≥ 0 then
        for j = 1 to CâştigMaxIndex do
            P_curent = PLANIF_OP (P_curent, o_O[j], s_S[j]);
        endfor
    endif
until CâştigMax < 0.
```

Figura 29. Algoritmul de replanificare iterativă.

2. A doua îmbunătăţire se bazează pe o strategie mai sofisticată de selecţie a mutărilor. În loc să se evalueze doar câştigul unei mutări, algoritmul efectuează o anticipare şi evaluează mutarea prezentă pe baza mutărilor viitoare posibile. Gradul de anticipare determină compromisul dintre timpul de calcul şi calitatea proiectului.

3.4. Planificarea cu restricţii de resurse

Problema de planificare cu restricţii de resurse este întâlnită în numeroase aplicaţii unde există o limitare dată de spaţiul de pe cip. Restricţia este specificată de obicei sub forma numărului de unităţi funcţionale sau a spaţiului disponibil. Dacă restricţia este dată sub forma spaţiului total disponibil, algoritmul de planificare determină tipul unităţilor funcţionale utilizate. Scopul unui asemenea algoritm este de a se obţine performanţele cele mai bune, cu satisfacerea restricţiei date.

Restricţiile de resurse sunt satisfăcute dacă numărul total de operaţii planificate într-un pas de control dat nu depăşeşte restricţiile impuse. Restricţiile de spaţiu sunt satisfăcute dacă suprafaţa tuturor unităţilor nu depăşeşte restricţiile. O restricţie pentru o uni-

tate poate fi testată în fiecare pas de control, dar restricția spațiului total poate fi testată numai pentru întregul proiect. Restricțiile de dependențe ale datelor pot fi satisfăcute dacă se asigură ca toți predecesorii unui nod să fie planificați înaintea planificării nodului. Astfel, la planificarea operației o_i în pasul de control s_j, trebuie să se asigure ca cerințele hardware ale operației o_i și ale altor operații deja planificate în pasul s_j să nu depășească restricția dată, și ca toți predecesorii nodului o_i să fie deja planificați.

Se vor descrie doi algoritmi de planificare cu restricții de resurse: metoda bazată pe liste și metoda de planificare cu liste statice.

3.4.1. Metoda de planificare bazată pe liste

Planificarea bazată pe liste este una din cele mai utilizate metode pentru planificarea operațiilor cu restricții de resurse. În principiu, ea este o generalizare a tehnicii de planificare *ASAP*, deoarece produce același rezultat în absența restricțiilor de resurse.

Algoritmul de planificare bazată pe liste păstrează o listă de priorități a nodurilor pregătite. Un nod pregătit reprezintă o operație ale cărei predecesori au fost deja planificați. În fiecare iterație, operațiile de la începutul listei nodurilor pregătite sunt planificate până când sunt utilizate toate resursele în acea stare. Lista de priorități este sortată după o funcție de prioritate. Funcția de prioritate rezolvă deci conflictul la resurse între operații. Dacă există conflicte de utilizare a resurselor între operațiile pregătite (de exemplu, sunt pregătite trei operații de adunare, dar se specifică o restricție de numai două sumatoare), va fi planificată operația cu prioritatea mai înaltă. Operațiile cu prioritate mai redusă vor fi amânate până la pașii de control următori. Planificarea unei operații poate determina ca unele operații să devină pregătite. Aceste operații sunt inserate în listă conform funcției de prioritate. Calitatea rezultatelor obținute de un algoritm de planificare bazată pe resurse depinde în principal de funcția sa de prioritate.

Figura 30 prezintă metoda de planificare bazată pe liste. Se utilizează o listă de priorități *Plist* pentru fiecare tip de operație ($t_k \in T$). Aceste liste sunt reprezentate de variabilele $PList_{t_1}$, $PList_{t_2}$, ..., $PList_{t_m}$. Operațiile din aceste liste sunt planificate în pașii de control pe baza valorii N_{t_k}, care este numărul de unități funcționale cu tipul t_k. Funcția *INS_OP_PREG* parcurge setul de noduri, *V*, determină dacă există o operație din set care este pregătită (deci toți predecesorii acestuia sunt planificați), șterge fiecare nod pregătit din setul *V* și îl adaugă la una din listele de priorități pe baza tipului acestuia. Funcția *PLANIF_OP* (P_{curent}, o_i, s_j) returnează o nouă planificare după ce operația o_i a fost planificată în pasul de control s_j. Funcția *DELETE* ($PList_{t_k}$, o_i) șterge operația indicată o_i din lista specificată.

Inițial, toate nodurile care nu au nici un predecesor sunt inserate în lista de priorități corespunzătoare de către funcția *INS_OP_PREG*, pe baza funcției de prioritate. Bucla **while** extrage operații din fiecare listă de priorități și le planifică în pasul curent, până când toate resursele sunt epuizate. Planificarea unui operații în pasul curent determină ca alte operații succesoare să devină pregătite. Aceste operații sunt planificate în timpul iterațiilor următoare ale buclei. Iterațiile continuă până când toate listele de priorități devin vide.

```
INS_OP_PREG (V, PList_t1 , PList_t2 , ..., PList_tm);
Pas_c = 0;
while ((PList_t1 ≠ φ ) sau ... sau (PList_tm ≠ φ )) do
    Pas_c = Pas_c + 1;
    for k = 1 to m do
        for unit_func = 1 to N_k do
            if PList_tk ≠ φ then
                PLANIF_OP (P_curent, FIRST (PList_tk ), Pas_c);
                PList_tk = DELETE (PList_tk , FIRST (PList_tk ));
            endif
        endfor
    endfor
    INS_OP_PREG (V, PList_t1 , PList_t2 , ..., PList_tm);
endwhile
```

Figura 30. Algoritmul metodei de planificare bazată pe liste.

Se va ilustra procesul de planificare printr-un exemplu (**Figura 31**). Se presupune că resursele disponibile sunt două circuite de înmulţire, un sumator, un scăzător şi un comparator (**Figura 31(c)**). Fiecare operaţie o_i din graful fluxului de date (**Figura 31(a)**) este etichetată cu domeniul său de mobilitate (*dom_mob*(o_i)). Nodurile cu o mobilitate mai redusă trebuie planificate mai repede, deoarece întârzierea asignării lor unui pas de control creşte probabilitatea extinderii planificării. În consecinţă, valoarea mobilităţii reprezintă o funcţie de prioritate adecvată. Pentru fiecare tip de operaţie, se construieşte o listă de priorităţi (**Figura 31(b)**), în care nodurile pregătite cu o mobilitate mai redusă sunt mai prioritare. Dacă două operaţii au aceeaşi mobilitate, cea cu un index mai mic va fi mai prioritară.

În timpul primei iteraţii, când operaţiile pregătite sunt planificate în pasul de control s_1, există cinci operaţii pregătite: o_1, o_2, o_3, o_4 şi o_{10}. Operatorul o_{10}, care este singurul operator de adunare, este planificat în pasul de control s_1, fără considerarea altor factori. Deoarece sunt disponibile numai două circuite de înmulţire, numai două din cele patru operaţii de înmulţire pregătite pot fi planificate în pasul de control s_1; se aleg operaţiile o_1 şi o_2 pentru planificare, deoarece acestea au mobilităţi mai reduse decât o_3 sau o_4.

După prima iteraţie, operaţiile o_1, o_2 şi o_{10} sunt planificate în pasul s_1. În iteraţia a doua, operaţiile o_5 şi o_{11} sunt adăugate listei de priorităţi, deoarece aceste operaţii au noduri de intrare care au fost planificate. În iteraţia a doua, lista de operaţii pregătite constă din o_5, o_{11}, o_3 şi o_4. Această listă este sortată în funcţie de mobilităţi şi întregul proces se repetă. După patru iteraţii, toate operaţiile sunt planificate în paşii de control corespunzători (**Figura 31(d)**).

Gradul de mobilitate reprezintă doar una din funcţiile de prioritate care au fost propuse pentru planificarea bazată pe liste. O altă funcţie de prioritate utilizează lungimea căii celei mai lungi de la nodul de operaţie la un nod care nu are un succesor imediat. Această cale este proporţională cu numărul de paşi suplimentari necesari pentru terminarea planificării dacă operaţia nu este planificată în pasul curent.

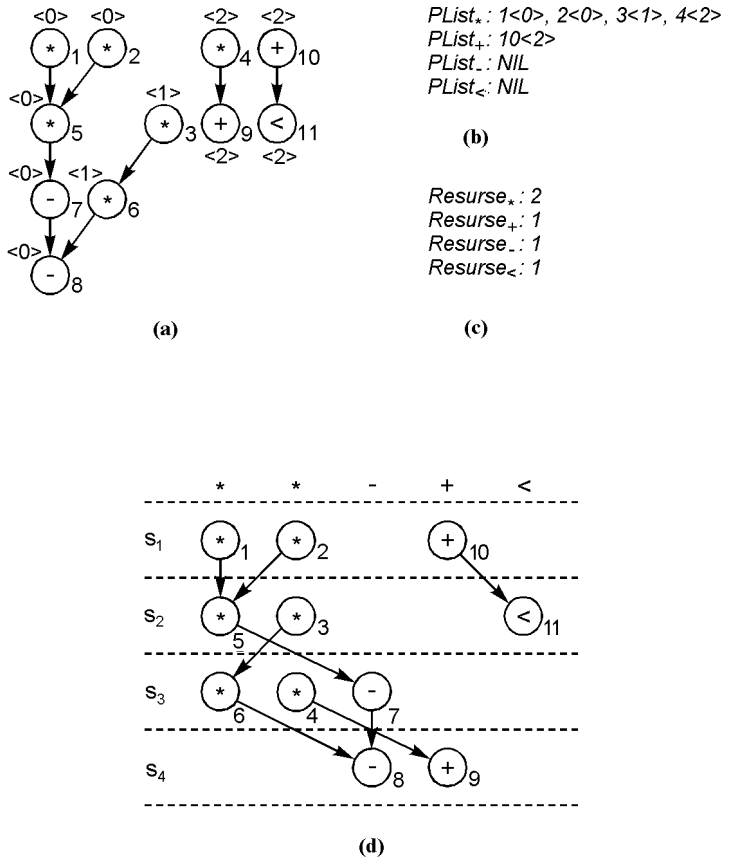


Figura 31. Planificarea bazată pe liste: (a) GFD etichetat cu gradele de mobilitate ; (b) lista operațiilor pregătite pentru starea s_1; (c) restricțiile de resurse; (d) GFD planificat.

O altă metodă utilizează numărul de noduri succesoare imediate ale unei operații ca funcție de prioritate: un nod de operație cu mai mulți succesori imediați este planificat mai repede, deoarece astfel devin pregătite mai multe operații decât în cazul unui nod cu mai puțini succesori.

3.4.2. Metoda de planificare cu liste statice

În locul construirii unei liste de priorități în mod dinamic în timpul planificării fiecărui pas de control, se poate crea o singură listă de dimensiuni mari înaintea începerii planificării. Această metodă este diferită de planificarea obişnuită bazată pe liste nu numai prin selecția operațiilor candidate, dar şi prin asignarea paşilor de control la aceste operații. Algoritmul sortează toate operațiile utilizând etichetele cu valorile *ALAP* în ordine crescătoare ca şi cheie primară şi etichetele *ASAP* în ordine descrescătoare ca şi cheie secundară. Dacă ambele chei au aceeaşi valoare, se utilizează o ordonare arbitrară. Această listă sortată este păstrată ca listă de priorități care determină ordinea în care operațiile sunt planificate.

Figura 32(a) prezintă graful fluxului de date pentru exemplul circuitului de rezolvare a ecuației diferențiale, iar **Figura 32(b)** indică valorile *ASAP* şi *ALAP* pentru fiecare nod şi lista completă de priorități pentru acest exemplu. Operațiile o_8, o_9 şi o_{11} au valoarea *ALAP* minimă (1) şi astfel au prioritatea cea mai redusă din listă. Dintre aceste operații, o_8 are o etichetă *ASAP* cu o valoare mai mare decât o_9 şi o_{11}, având deci prioritatea minimă. Operațiile o_9 şi o_{11} au aceleaşi valori *ASAP* şi *ALAP*, operația o_9 fiind selectată în mod arbitrar pentru a precede operația o_{11}. Restul listei este format în mod similar.

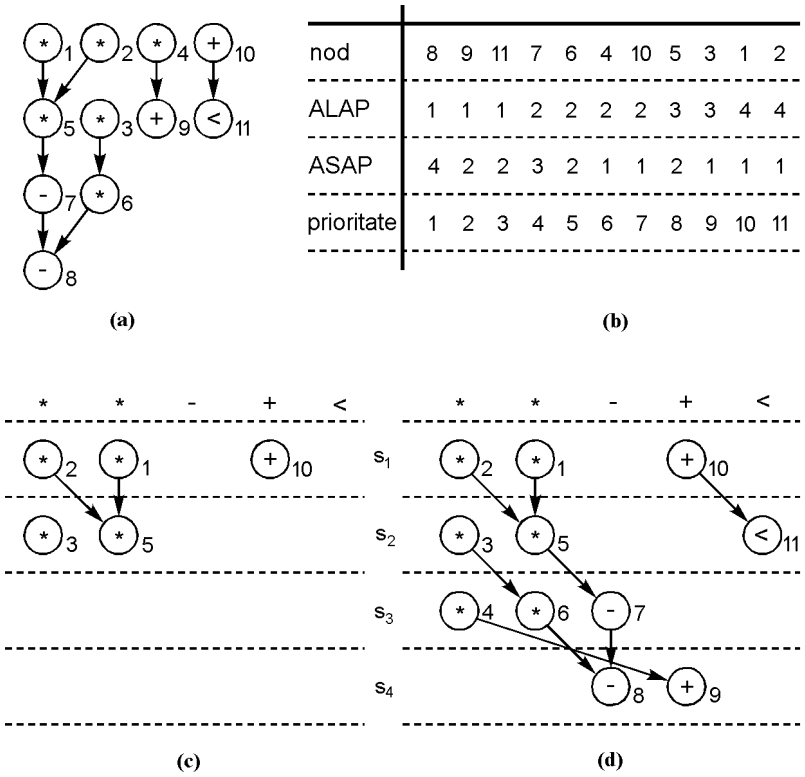


Figura 32. Planificarea cu liste statice: (a) GFD; (b) lista de priorități; (c) planificarea parțială a cinci noduri; (d) planificarea finală.

După crearea listei de priorități, operațiile sunt planificate secvențial începând cu ultima operație (cu prioritatea cea mai mare) din listă. O operație este planificată cât mai repede posibil, în funcție numai de resursele disponibile şi dependențele între operații. Astfel, operația o_2 este prima care va fi planificată. Aceasta este planificată în primul pas de control, deoarece sunt disponibile ambele circuite de înmulțire. Următoarea operație planificată este o_1 şi ea este asignată celui de-al doilea circuit de înmulțire în acelaşi pas de control. Operația o_3 nu poate fi planificată în starea s_1 deoarece nu mai sunt disponibile circuite de înmulțire, astfel încât este planificată în starea s_2. Operația o_5 nu poate fi planificată în starea s_1 deoarece datele sale de intrare sunt disponibile numai în starea s_2, astfel că ea este planificată în starea s_2. Deşi operația o_{10} are o prioritate mai redusă decât operațiile o_3 şi o_5, ea este planificată în pasul de control s_1 deoarece este disponibil un suma-

tor în starea s_1 şi nu depinde de nici o altă operație planificată anterior. Planificarea finală pentru acest exemplu este prezentată în **Figura 31(d)**.

3.5. Planificarea cu eliminarea ipotezelor simplificatoare

În secțiunile precedente au fost prezentați algoritmi de planificare care utilizează un set de ipoteze simplificatoare. În continuare se vor extinde algoritmii de planificare pentru cazul unor modele de proiectare mai realiste, ca de exemplu unități funcționale cu timpi de execuție variabili, unități multifuncționale, şi descrieri funcționale care conțin construcții condiționale şi bucle.

3.5.1. Unități funcționale cu întârzieri variabile

Unitățile funcționale reale au întârzieri de propagare diferite în funcție de proiectarea lor. De exemplu, un circuit de înmulțire în virgulă mobilă de 32 de biți este mai lent decât un sumator în virgulă fixă cu aceeaşi lungime a cuvântului. Astfel, nu se va putea presupune că fiecare operație se termină într-un pas de control. Această presupunere ar conduce la un ciclu de ceas cu o durată mai mare, pentru a se adapta la unitatea cea mai lentă, după cum se indică în **Figura 33(a)**, unde un circuit de înmulțire necesită un timp de aproximativ trei ori mai mare decât un sumator sau un scăzător. Ca rezultat, unitățile cu întârzieri mai mici decât ciclul de ceas rămân inactive în timpul unei părți a ciclului. Dacă se permite utilizarea unităților funcționale cu întârzieri arbitrare, se va îmbunătăți gradul de utilizare al unităților funcționale. Deoarece unitățile funcționale cu întârzieri arbitrare pot necesita pentru execuție mai mult decât un ciclu de ceas, trebuie să se generalizeze modelul de execuție al operației, pentru a se permite *cicluri multiple*, *înlănțuirea* şi *execuția de tip pipeline*.

Dacă perioada ciclului de ceas este redusă pentru a se permite ca operațiile mai rapide să fie executate într-un ciclu de ceas, atunci operațiile mai lente necesită cicluri de ceas multiple pentru terminarea execuției. Aceste operații mai lente, numite *operații cu cicluri multiple*, trebuie planificate de-a lungul a doi sau mai mulți paşi de control (**Figura 33(b)**). Aceasta creşte gradul de utilizare al unităților mai rapide, deoarece pentru acestea pot fi planificate două sau mai multe operații în timpul unei singure operații executate de unitățile cu cicluri multiple. Totuşi, sunt necesare circuite latch de intrare înaintea unităților funcționale cu cicluri multiple, pentru a păstra operanzii acestora până când rezultatul este disponibil. De asemenea, un ciclu de ceas cu durată mai redusă are ca rezultat un număr mai mare de paşi de control, ceea ce determină creşterea complexității logicii de control.

O altă metodă de creştere a gradului de utilizare a unităților funcționale este de a se permite execuția serială a două sau mai multe operații într-un pas de control, deci *înlănțuirea*. Rezultatul unei unități funcționale poate fi utilizat direct ca intrare pentru o altă unitate funcțională (**Figura 33(c)**). Această metodă necesită conexiuni directe între unitățile funcționale, în plus față de conexiunile dintre unitățile funcționale şi cele de memorie.

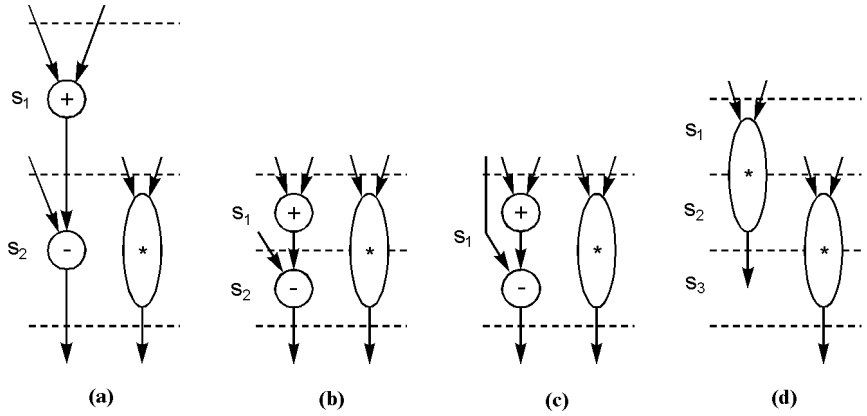


Figura 33. Planificarea pentru unităţi cu întârzieri variabile: (a) planificare în care fiecare operaţie este executată într-un pas; (b) planificare cu un multiplicator cu 2 cicluri; (c) planificare cu un sumator şi scăzător înlănţuit; (d) planificare cu un multiplicator de tip pipeline cu două etaje.

Execuţia de tip *pipeline* este o tehnică simplă şi eficientă pentru creşterea gradului de paralelism. Dacă se utilizează o unitate funcţională de tip pipeline, planificatorul trebuie să calculeze cerinţele de resurse în mod diferit. De exemplu, în **Figura 33(d)** cele două operaţii de înmulţire pot partaja acelaşi circuit de înmulţire cu două etaje, deşi cele două operaţii sunt executate concurent. Această partajare este posibilă deoarece fiecare operaţie de înmulţire utilizează un etaj diferit al circuitului de înmulţire de tip pipeline. Astfel, este necesar un singur circuit de înmulţire care utilizează tehnica pipeline, în locul a două circuite care nu utilizează această tehnică.

3.5.2. Unităţi multifuncţionale

În secţiunea precedentă, s-a extins problema de planificare la unităţile funcţionale cu întârzieri de propagare neuniforme şi un număr diferit de etaje pipeline. Totuşi, s-a presupus în continuare că fiecare unitate funcţională execută o singură operaţie. Această presupunere nu este realistă, deoarece în practică o unitate multifuncţională are un cost mai redus decât un set de unităţi unifuncţionale care execută aceleaşi operaţii. De exemplu, deşi costul unui sumator/scăzător este cu 20% mai ridicat decât cel al unui sumator sau scăzător separat, este mai avantajoasă utilizarea primului faţă de utilizarea unui sumator şi a unui scăzător. Astfel, este de aşteptat că proiectanţii vor utiliza un număr cât mai mare posibil de unităţi multifuncţionale.

S-au presupus de asemenea implementări fizice unice pentru fiecare unitate funcţională. În realitate, biblioteca de componente are implementări multiple ale aceleiaşi componente, fiecare implementare având o caracteristică spaţiu / întârziere diferită. De exemplu, o adunare poate fi executată fie rapid, utilizând un sumator cu anticiparea transportului de dimensiuni mari, fie mai lent, utilizând un sumator cu transport succesiv

de dimensiuni mici. Fiind dată o bibliotecă cu implementări multiple ale aceleiaşi unități, algoritmii de planificare trebuie să execute simultan două operații importante: planificarea operațiilor, în care operațiile sunt asignate paşilor de control, şi selecția componentelor, în care algoritmul selectează implementarea cea mai eficientă a unității funcționale, pentru fiecare operator din bibliotecă.

Poate fi utilizat un algoritm de planificare bazat pe tehnologie. Fiind dată o bibliotecă cu implementări multiple ale unităților funcționale, scopul principal al algoritmului este de a realiza implementarea în cadrul restricției de timp specificate, cu costuri minime. Pentru fiecare operație din graful fluxului de date, este selectată o componentă eficientă, astfel încât operațiile din calea critică sunt implementate cu componente mai rapide, iar operațiile care nu se află în calea critică sunt implementate cu componente mai lente. Simultan, algoritmul asigură ca operațiile să fie planificate în paşii de control corespunzători care permit partajarea unităților funcționale pe parcursul diferitelor stări.

3.5.3. Descrieri care utilizează construcții condiționale şi bucle

Pe lângă blocurile de instrucțiuni secvențiale, descrierile funcționale conțin în general atât construcții condiționale, cât şi construcții de buclare. Astfel, un algoritm de planificare realist trebuie să țină cont de aceste construcții.

3.5.3.1. Construcții condiționale

O construcție condițională este similară cu o instrucțiune *if* sau *case* a unui limbaj de programare. Aceasta are ca rezultat o serie de ramuri care sunt mutual exclusive. **Figura 34** prezintă un fragment al unei descrieri funcționale care conține o instrucțiune condițională *if*. Această descriere nu poate fi reprezentată numai printr-un graf al fluxului de date, fiind necesar un graf cu dependențe de control şi de date (de exemplu, GFCD). În acest exemplu, fluxul de control determină execuția celor patru grafuri ale fluxului de date reprezentate prin pătrate.

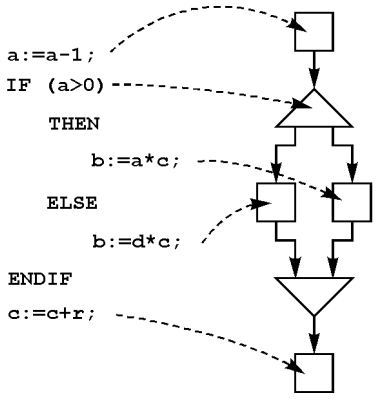


Figura 34. GFCD pentru o descriere funcțională cu o construcție condițională.

Un algoritm eficient de planificare partajează resursele între operațiile mutual exclusive. De exemplu, va fi executată numai una din cele două operații de înmulțire din **Figura 34** pe durata fiecărei execuții a modelului. Astfel, algoritmul poate realiza planificarea ambelor operații de înmulțire în acelaşi pas de control, chiar dacă este disponibil un singur circuit de înmulțire în fiecare pas.

3.5.3.2. Construcții de buclare

O descriere funcțională poate conține construcții de buclare. De exemplu, într-un filtru pentru aplicații DSP, un set de acțiuni este executat în mod repetat pentru fiecare eşantion al şirului datelor de intrare. Această acțiune repetată este modelată printr-o construcție de buclare. În aceste descrieri, optimizarea corpului buclei poate îmbunătăți performanțele modelului.

Construcțiile de buclare prezintă un paralelism potențial între diferitele iterații ale buclei. Ca urmare, pot fi executate mai multe iterații ale aceleiaşi bucle în mod concurent. Planificarea unui corp al buclei diferă de planificarea unui graf al fluxului de date, prin faptul că trebuie să se considere paralelismul potențial între diferitele iterații ale buclei.

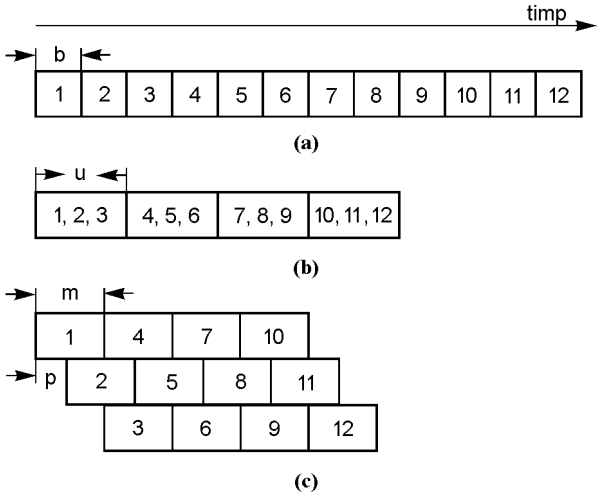


Figura 35. Planificarea buclelor: (a) execuție secvențială; (b) expandarea parțială a buclei; (c) suprapunerea iterațiilor succesive ale buclei.

Se utilizează **Figura 35** pentru a ilustra trei metode diferite de planificare a unei bucle. Se presupune că bucla este finită şi constă din n iterații (n = 12 în **Figura 35**). Prima şi cea mai simplă metodă presupune *execuția secvențială* a buclei şi planifică fiecare iterație a buclei în b paşi de control. Dacă cele 12 iterații sunt executate secvențial, timpul total de execuție este de 12b paşi de control, după cum se indică în **Figura 35(a)**. Celelalte două metode țin cont de paralelismul între iterațiile buclei.

A doua tehnică este numită *expandarea buclelor*, deoarece un anumit număr de iterații ale unei bucle sunt desfăşurate. Această acțiune are ca rezultat o buclă cu un corp al buclei de dimensiuni mai mari, dar cu un număr mai redus de iterații. Corpul buclei de dimensiuni mai mari permite o mai mare flexibilitate pentru compactarea planificării. În **Figura 35(b)** sunt desfăşurate trei iterații într-o super-iterație, rezultând patru super-iterații. Fiecare super-iterație este planificată pe durata a u paşi de control. Astfel, dacă $u < 3b$, timpul total de execuție este mai mic decât 12b paşi de control.

A treia metodă ține cont de paralelismul din interiorul buclelor, realizând *suprapunerea de tip pipeline a unor iterații succesive* ale unei bucle. **Figura 35(c)** prezintă o buclă cu un corp care necesită *m* paşi de control pentru execuție. În cadrul suprapunerii iterațiilor, se inițiază o nouă iterație după fiecare *p* paşi de control, unde $p < m$, deci se suprapun iterații succesive. Timpul total de execuție este de $m + (n - 1) \times p$ paşi de control (**Figura 35(c)**). Expandarea buclelor este aplicabilă numai dacă contorul buclei este cunoscut în avans, dar suprapunerea iterațiilor poate fi aplicată atât pentru bucle cu un număr fix de iterații, cât şi pentru cele cu un număr de iterații variabil.

Pentru ilustrarea acestor metode de execuție a buclelor, se alege un exemplu de graf al fluxului de date care conține iterații multiple. **Figura 36(a)** prezintă graful corpului unei bucle care conține 17 operații identice, fiecare necesitând un pas de control pentru execuție. Arcele cu linii întrerupte indică dependențe ale datelor peste limitele buclei. De exemplu, arcul de la nodul *P* la nodul *J* indică faptul că rezultatul generat de operația *P* în timpul unei iterații este utilizat de operația *J* în următoarea iterație. Se va realiza planificarea acestui graf al fluxului de date utilizând cele trei tehnici de planificare a buclelor. În scopul evaluării calității planificărilor rezultate, se utilizează două măsuri de performanță: rata de utilizare a unităților funcționale, deci procentul de stări în care unitățile funcționale sunt utilizate pentru a executa o anumită operație, şi costul părții de control, care este măsurat prin numărul cuvintelor de control unice din unitatea de control.

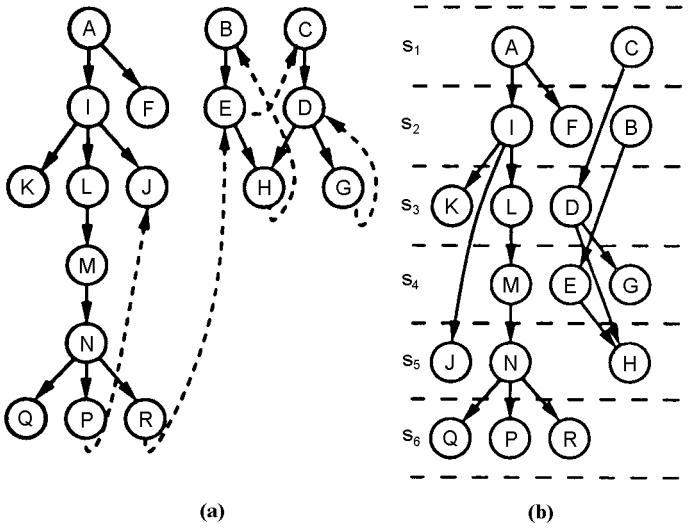


Figura 36. Planificarea standard a unei bucle: (a) GFD cu dependențe între iterații; (b) planificare secvențială utilizând trei unități funcționale.

În **Figura 36(b)** se prezintă o planificare cu 6 paşi de control utilizând 3 unități funcționale. Deoarece o cale de date cu 3 unități funcționale necesită cel puțin 6 paşi de control pentru a executa 17 operații, iar lungimea căii critice a grafului este de 6, planificarea este optimă. Rata de utilizare a unităților funcționale este $17 / (3 \times 6) = 17 / 18$.

Costul părții de control este de 6 cuvinte, presupunând că este necesar un cuvânt în unitatea de control pentru fiecare pas de control planificat. Performanțele obținute pot fi îmbunătățite numai dacă se ține cont de paralelismul existent.

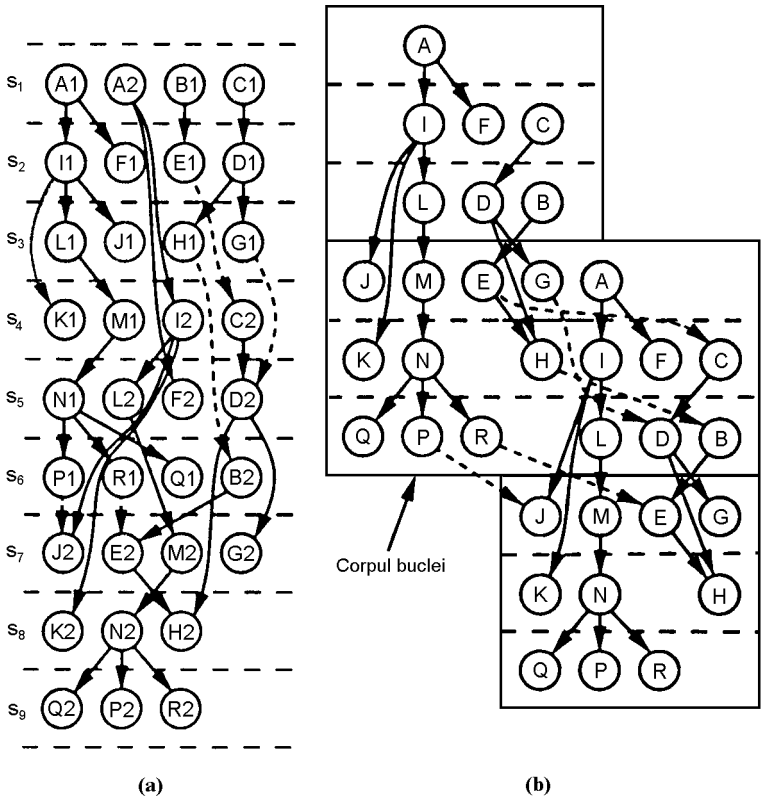


Figura 37. Planificarea prin utilizarea paralelismului: (a) expandarea buclei; (b) suprapunerea iterațiilor.

Figura 37(a) prezintă o planificare pentru exemplul din **Figura 36** utilizând expandarea buclei. Două iterații, deci două copii ale grafului, sunt planificate în 9 paşi de control, utilizând 4 unități funcționale. Rata de utilizare a unităților rămâne aceeaşi: (17×2) / (4×9) = 17 / 18. Însă, timpul mediu necesar pentru execuția unei iterații este redus de la 6 la 9/2 = 4.5 paşi de control. Costul părții de control creşte de la 6 la 9 cuvinte de control.

Figura 37(b) prezintă o planificare pentru acelaşi exemplu utilizând suprapunerea ciclurilor. Iterațiile succesive îşi încep execuția la intervale de 3 paşi de control. Se utilizează 6 unități funcționale, rata de utilizare a lor fiind de (5+6+6) / (6×3) = 17 / 18, aceeaşi ca şi în cazul planificărilor precedente. Timpul mediu de execuție al unei iterații este însă de aproximativ 3 paşi de control, dacă numărul de iterații este foarte mare. Costul părții de control este de 9 cuvinte (câte 3 pentru începutul, corpul şi sfârşitul buclei).

În contextul sintezei de nivel înalt, trebuie să se considere modificarea costului părții de control atunci când se efectuează aceste optimizări. Atât expandarea buclelor, cât şi suprapunerea ciclurilor determină creşterea costului părții de control. Trebuie efectuat un compromis între viteza de execuție şi costul părții de control.

Planificarea unei bucle expandate este relativ simplă, deoarece algoritmii de planificare descrişi pot fi aplicați grafului expandat al fluxului de date. Singura problemă constă în gradul de expandare al buclei. Performanțele unor algoritmi cu o complexitate ridicată a calculelor (de exemplu, metoda programării liniare) vor fi reduse în cazul unui grad de expandare ridicat al buclei. Pe de altă parte, planificarea unei bucle cu suprapunerea ciclurilor este mai complexă. Pentru un număr fixat al paşilor de control între iterațiile succesive, k, se poate extinde algoritmul de planificare bazat pe liste pentru a planifica operații cu restricții de resurse. Într-un pas de control q, astfel încât $q > k$, operațiile din diferitele iterații ale buclei sunt executate concurent. În consecință, trebuie prevăzute resursele necesare pentru execuția diferitelor operații în toate iterațiile concurente.

4. Alocarea căii de date

4.1. Introducere

Procedura de planificare asignează operațiile care trebuie executate unor paşi de control, şi astfel converteşte o descriere funcțională într-un set de transferuri între registre care pot fi descrise printr-o tabelă de stări. O arhitectură care poate fi utilizată pentru implementarea unei asemenea descrieri este automatul cu stări finite şi cu o cale de date (ASFD). Unitatea de control pentru această arhitectură poate fi generată pe baza transferurilor între registre asignate fiecărui pas de control; această operație se numeşte *alocarea căii de date* sau *sinteza căii de date*.

O cale de date în cadrul modelului ASFD este o listă de conexiuni formată din trei tipuri de componente sau unități la nivelul transferurilor între registre (RT): funcționale, de memorie şi de interconectare. Unitățile funcționale, ca sumatoare, circuite de deplasare, circuite de înmulțire, unități aritmetice şi logice, execută operațiile specificate în descrierea funcțională. Unitățile de memorie, ca registre, tablouri de registre, memorii RAM sau ROM, păstrează valorile variabilelor generate şi consumate în timpul execuției. Unitățile de interconectare, de exemplu magistrale şi multiplexoare, transportă datele între unitățile funcționale şi cele de memorie.

Alocarea căii de date constă din două operații principale: selecția unităților şi asignarea unităților. Selecția unităților determină numărul şi tipul componentelor RT care vor fi utilizate. Asignarea unităților implică asocierea variabilelor şi operațiilor din graful planificat al fluxului de control şi de date (GFCD) cu unitățile funcționale, de memorie şi de interconectare, asigurând funcționarea corectă a implementării pentru setul selectat al componentelor. Pentru fiecare operație din GFCD, este necesară o unitate funcțională care poate executa acea operație. Pentru fiecare variabilă care este utilizată pe parcursul a mai multor paşi de control, este necesară o unitate de memorie pentru păstrarea valorilor datelor pe durata de viață a variabilei. În sfârşit, pentru fiecare transfer al datelor din GFCD, este necesar un set de unități de interconectare pentru efectuarea transferului.

Pe lângă restricțiile de proiectare impuse în cadrul descrierii funcționale originale şi reprezentate în GFCD, sunt impuse restricții suplimentare pentru procesul de asignare de tipul unităților hardware selectate. De exemplu, o unitate funcțională poate executa o singură operație în fiecare pas de control dat. Similar, numărul de accesuri multiple la o unitate de memorie pe durata unui pas de control este limitat de numărul porturilor paralele ale unității.

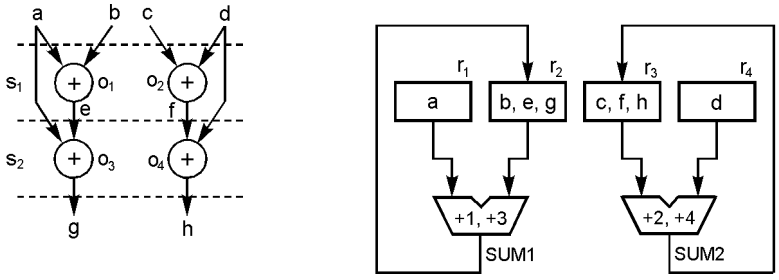


Figura 38. Asignarea obiectelor funcționale la componentele RT.

Se ilustrează asignarea variabilelor şi a operațiilor grafului fluxului de date din **Figura 38** la componente RT. Presupunem că se selectează două sumatoare, *SUM1* şi *SUM2*, şi patru registre, r_1, r_2, r_3 şi r_4. Operațiile o_1 şi o_2 nu pot fi asignate la acelaşi sumator deoarece ele trebuie executate în acelaşi pas de control s_1. Pe de altă parte, operația o_1 poate partaja un sumator cu operația o_3, deoarece ele sunt executate în timpul unor paşi de control diferiți. Astfel, operațiile o_1 şi o_3 sunt asignate la sumatorul *SUM1*. Variabilele *a* şi *e* trebuie memorate separat deoarece valorile lor sunt necesare simultan în pasul de control s_2. Registrele r_1 şi r_2, în care se memorează variabilele *a* şi *e*, trebuie conectate la porturile de intrare ale sumatorului *SUM1*; în caz contrar, operația o_3 nu va putea fi executată de sumatorul *SUM1*. Similar, operațiile o_2 şi o_4 sunt asignate sumatorului *SUM2*. De observat că există mai multe posibilități pentru efectuarea asignării. De exemplu, se pot asigna operațiile o_2 şi o_3 la sumatorul *SUM1*, iar operațiile o_1 şi o_4 la sumatorul *SUM2*.

Pe lângă asigurarea funcționării corecte, calea de date alocată trebuie să satisfacă restricțiile globale de proiectare, specificate sub forma unor indicatori ca spațiul ocupat, puterea disipată, perioada ciclului de ceas, etc. Pentru simplificarea problemei alocării, se vor utiliza doi indicatori de calitate: dimensiunea totală a implementării (suprafața ocupată) şi întârzierea introdusă de un registru în cazul cel mai defavorabil (durata ciclului de ceas).

Problema alocării se poate soluționa în trei moduri: prin metode de tip "greedy", care construiesc o implementare în mod progresiv prin traversarea GFCD; prin descompunerea problemei alocării în părțile sale constituente şi soluționarea separată a acestora; prin metode iterative, care încearcă combinarea soluțiilor subproblemelor alocării.

În continuare se vor prezenta caracteristicile tipice ale arhitecturilor cu căi de date şi efectele acestora asupra problemei alocării.

4.2. Arhitecturi cu căi de date

O arhitectură cu cale de date defineşte caracteristicile unităților din calea de date şi topologia de interconectare a acestora. O arhitectură simplă poate reduce semnificativ complexitatea problemelor de sinteză deoarece numărul variantelor alternative este mult

mai redus. Pe de altă parte, pentru o arhitectură cu mai puține restricții, deşi implementarea acesteia este mai dificilă, calitatea rezultată poate fi mai ridicată. Deşi o arhitectură mult simplificată conduce la algoritmi eleganți de sinteză, de obicei rezultatele obținute sunt inacceptabile.

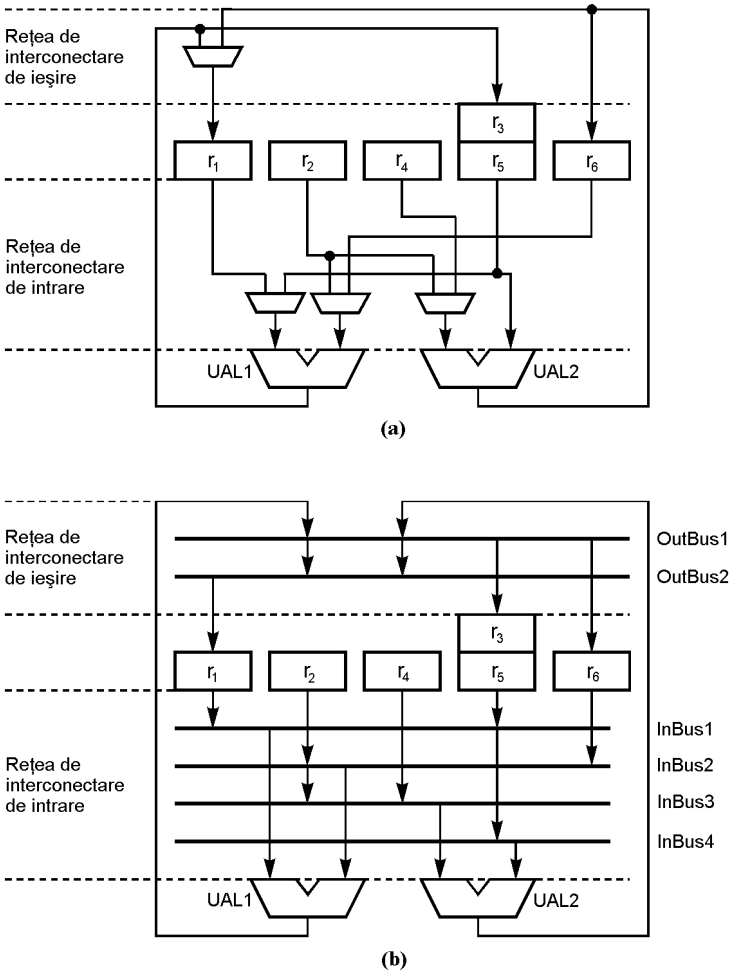


Figura 39. Interconexiuni ale căii de date: (a) prin multiplexoare; (b) prin magistrale.

Topologia de interconectare care permite realizarea transferurilor de date între unitățile funcționale şi cele de memorie este unul din factorii care au o influență semnificativă asupra performanțelor căii de date. Complexitatea topologiei de interconectare este definită de numărul maxim al unităților de interconectare între oricare două porturi ale unităților funcționale sau de memorie. Fiecare unitate de interconectare poate fi implementată cu un multiplexor sau o magistrală. De exemplu, **Figura 39** prezintă două căi de date, care utilizează ca unități de interconectare multiplexoare, respectiv magistrale, şi care implementează următoarele transferuri între registre:

s_1: $r_3 \Leftarrow UAL1\ (r_1, r_2)$; $r_1 \Leftarrow UAL2\ (r_3, r_4)$;
s_2: $r_1 \Leftarrow UAL1\ (r_5, r_6)$; $r_6 \Leftarrow UAL2\ (r_2, r_5)$;
s_3: $r_3 \Leftarrow UAL1\ (r_1, r_6)$;

Topologia de interconectare este de tip punct la punct dacă există o singură unitate de interconectare între oricare două porturi ale unităților funcționale şi/sau de memorie. Această topologie este cea mai utilizată pentru sinteza de nivel înalt, deoarece simplifică algoritmii de alocare. În cazul acestei topologii, se crează o conexiune între oricare două unități funcționale sau de memorie, dacă este necesar. Dacă este asignată mai mult de o conexiune la intrarea unei unități, se utilizează un multiplexor sau o magistrală. În scopul minimizării numărului de interconexiuni, se pot combina registrele în tablouri de registre cu porturi multiple. Fiecare port poate permite accesuri pentru citirea şi/sau scrierea datelor. Anumite tablouri de registre permit accesuri simultane de citire şi scriere prin porturi diferite. Deşi tablourile de registre reduc costul unităților de interconectare, fiecare port necesită circuite dedicate pentru decodificare în cadrul tabloului de registre, ceea ce creşte costul unităților de memorie şi întârzierea de propagare.

Pentru simplificarea problemei de asignare a unităților funcționale, se presupune că toate transferurile între registre se execută prin intermediul unităților funcționale, interconexiunile directe între două unități funcționale nu sunt permise. Astfel, sunt necesare unități de interconectare numai pentru conectarea porturilor de ieşire ale unităților de memorie la porturile de intrare ale unităților funcționale (care formează rețeaua de interconectare de intrare), şi pentru conectarea porturilor de ieşire ale unităților funcționale la porturile de intrare ale unităților de memorie (care formează rețeaua de interconectare de ieşire).

Complexitatea rețelelor de interconectare de intrare şi de ieşire nu trebuie să fie aceeaşi, fiind posibilă simplificarea uneia dintre ele cu prețul creşterii complexității celeilalte. De exemplu, este posibil ca circuitele de selecție să nu fie admise înaintea porturilor de intrare ale unităților de memorie. Aceasta are ca rezultat o rețea de interconectare de intrare mai complexă, şi deci un dezechilibru între timpii de citire şi de scriere ai unităților de memorie. De asemenea, poate exista o restricție în ceea ce priveşte numărul de magistrale de la ieşirea diferitelor tablouri de registre. Dacă se alocă o magistrală unică pentru fiecare ieşire a tablourilor de registre, nu sunt necesare drivere de magistrală cu trei stări între registre şi magistrale. Această restricție asupra ieşirilor tablourilor de registre necesită introducerea mai multor multiplexoare la porturile de intrare ale unităților funcționale. Mai mult, poate fi necesară duplicarea anumitor variabile în diferite tablouri de registre în scopul simplificării circuitelor de selecție dintre magistrale şi unitățile funcționale.

O altă metodă de interconectare, utilizată în mod frecvent pentru procesoare, este cea în care magistralele sunt partiționate în segmente, astfel încât fiecare segment este utilizat de o singură unitate funcțională la un moment dat. Unitățile funcționale şi de memorie sunt aranjate într-o singură linie, existând segmente de magistrale la intrările şi ieşirile acestora. Un transfer de date între segmente este realizat prin intermediul comutatoarelor între segmente. Astfel, alocarea interconexiunilor se realizează prin controlul comutatoarelor dintre segmentele magistralelor.

În continuare se vor analiza întârzierile implicate în transferurile între registre pentru exemplul din **Figura 39**. Secvențierea microoperațiilor de citire, execuție şi scriere în primele două cicluri de ceas este prezentată în **Figura 40**. Fie t_r întârzierea pentru citirea datelor din registre şi propagarea lor prin rețeaua de interconectare de intrare; t_e, întârzierea de propagare printr-o unitate funcțională; şi t_w, întârzierea pentru propagarea datelor de la unitățile funcționale prin rețeaua de interconectare de ieşire, şi scrierea lor în registre. Se presupune că toate componentele de la porturile de ieşire ale registrelor şi până la porturile de intrare ale acestora (deci, rețeaua de interconectare de intrare, unitățile aritmetice şi logice, şi rețeaua de interconectare de ieşire), sunt combinaționale. Astfel, ciclul de ceas va fi egal cu sau mai mare decât $t_r + t_e + t_w$.

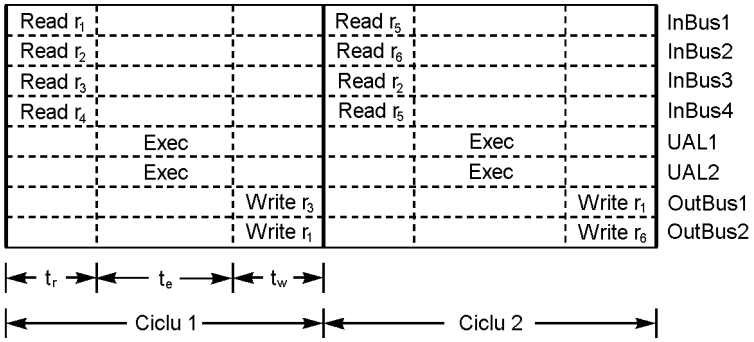


Figura 40. Execuția secvențială a trei microoperații în aceeaşi perioadă de ceas.

Pentru îmbunătățirea performanțelor căii de date, pot fi inserate latch-uri la porturile de intrare şi/sau de ieşire ale unităților funcționale. Dacă acestea sunt inserate numai la ieşirile unităților funcționale (**Figura 41**), accesurile pentru citire şi execuția operațiilor planificate în pasul de control curent pot fi realizate în acelaşi timp cu accesurile pentru scriere ale operațiilor planificate în pasul de control anterior (**Figura 42**). Perioada ciclului este redusă la $max(t_r + t_e, t_w)$, însă transferurile între registre nu sunt echilibrate: citirea registrelor şi execuția unei operații într-o unitate aritmetică şi logică sunt efectuate în primul ciclu, în timpul ciclului al doilea executându-se doar înscrierea rezultatului în registre.

Similar, dacă se introduc latch-uri numai la intrările unităților funcționale, accesurile pentru citire ale operațiilor planificate în următorul pas de control pot fi efectuate în acelaşi timp cu execuția şi accesurile pentru scriere ale operațiilor planificate în pasul de control curent. În acest caz perioada ciclului va fi $max(t_r, t_e + t_w)$. În ambele cazuri, tablourile de registre şi latch-urile sunt controlate printr-un ceas cu o singură fază.

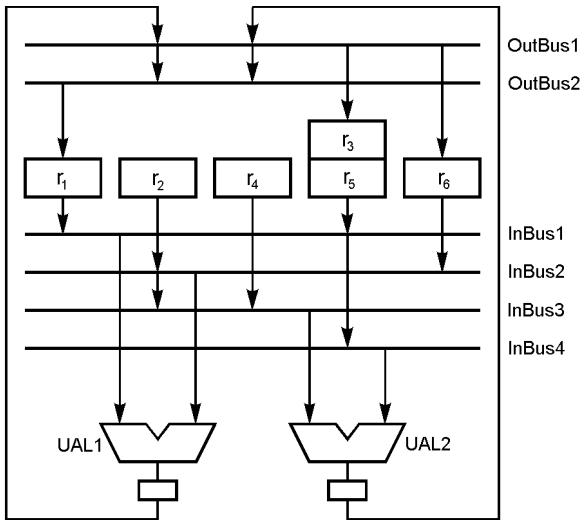


Figura 41. Inserarea latch-urilor la porturile de ieşire ale unităţilor funcţionale.

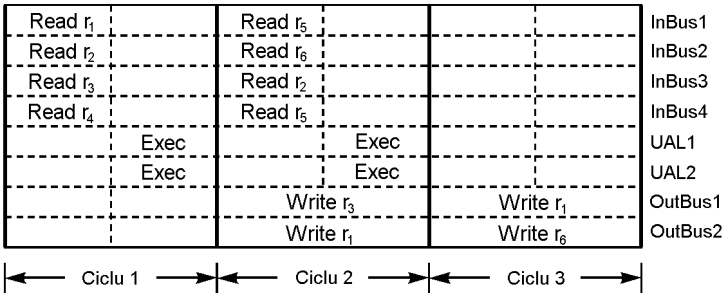


Figura 42. Suprapunerea transferurilor de date pentru citire şi scriere.

Execuţia unei operaţii şi citirea/scrierea datelor pot fi efectuate în mod concurent dacă se introduc latch-uri atât la intrările cât şi la ieşirile unităţilor funcţionale (**Figura 43**). Cele trei componente combinaţionale, şi anume unităţile de interconectare de intrare, unităţile funcţionale şi unităţile de interconectare de ieşire, pot fi active în mod concurent. **Figura 44** prezintă această execuţie de tip pipeline. Ciclul de ceas constă din două cicluri minore. Execuţia unei operaţii are loc pe parcursul a trei cicluri consecutive de ceas. Operanzii unei operaţii sunt transferaţi din tablourile de registre în latch-urile de intrare ale unităţilor funcţionale în timpul celui de-al doilea ciclu minor al primului ciclu. În ciclul al

doilea, unitatea funcțională execută operația şi scrie rezultatul în latch-ul de ieşire la sfârşitul ciclului. Rezultatul este transferat în tabloul de registre în timpul primului ciclu minor al celui de-al treilea ciclu. Este necesar un ceas cu două faze, fără suprapunere. Atât latch-ul de intrare, cât şi cel de ieşire sunt controlate de aceeaşi fază, deoarece accesul pentru citire şi execuția operației se termină simultan. Cealaltă fază este utilizată pentru controlul accesurilor pentru scriere în registre.

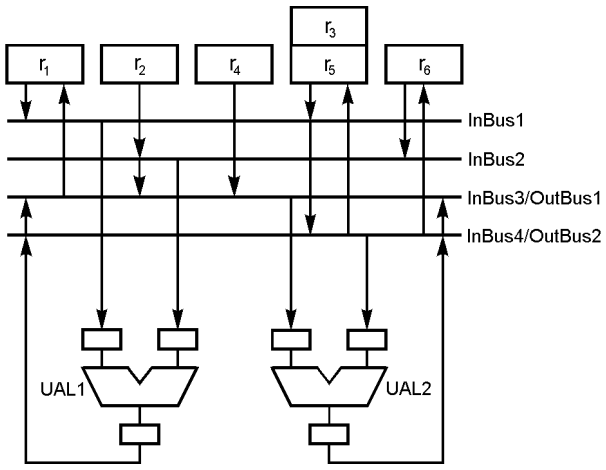


Figura 43. Inserarea latch-urilor la porturile de intrare şi de ieşire ale unităților funcționale.

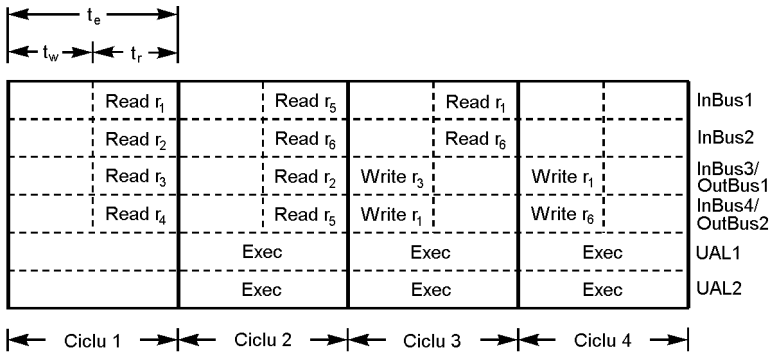


Figura 44. Suprapunerea transferurilor de date cu execuția operațiilor de către unitățile funcționale.

Prin suprapunerea execuției operațiilor în paşi de control succesivi, se poate creşte gradul de utilizare al unităților hardware. Perioada ciclului de ceas este redusă la $max(t_e, t_r + t_w)$. În plus, rețelele de intrare şi de ieşire pot partaja anumite unități de interconectare.

De exemplu, pentru *OutBus1* şi *InBus3*, respectiv *OutBus2* şi *InBus4*, se utilizează câte o singură magistrală în **Figura 43**.

Astfel, prin inserarea latch-urilor de intrare şi de ieşire se pot fuziona mai multe unități de interconectare, ceea ce poate simplifica proiectarea căii de date. Prin divizarea transferurilor între registre în microoperații executate în cicluri diferite de ceas, se obține o utilizare mai eficientă a resurselor hardware. Această metodă necesită însă ca algoritmii de asignare a unităților să evalueze un număr mai mare de alternative de proiectare.

Pentru a se permite înlănțuirea operațiilor, deci execuția a două sau mai multor operații în serie în cadrul aceluiaşi pas de control, sunt necesare legături directe de la porturile de ieşire ale unor unități funcționale la porturile de intrare ale altor unități funcționale. Într-o arhitectură cu magistrale partajate (de exemplu, în **Figura 43**), această legătură poate fi realizată simplu prin utilizarea căii de la portul de ieşire al unei unități funcționale printr-o magistrală la portul de intrare al unei alte unități funcționale. Deoarece o asemenea cale trebuie să fie combinațională, trebuie adăugate legături pentru ocolirea latch-urilor în calea de înlănțuire.

4.3. Operații pentru alocarea căii de date

Sinteza căii de date constă din următoarele operații diferite, dar interdependente: selecția modulelor, alocarea unităților funcționale, alocarea unităților de memorie şi alocarea interconexiunilor.

4.3.1. Selecția unităților

Un model simplu de proiectare poate presupune că este disponibil un singur tip de unitate funcțională pentru fiecare operație. O bibliotecă reală de componente RT conține însă tipuri multiple de unități funcționale, fiecare cu caracteristici diferite, şi fiecare implementând una sau mai multe operații diferite din descrierea la nivelul transferurilor între registre. De exemplu, o adunare poate fi executată fie printr-un sumator cu transport succesiv, de dimensiuni mici dar lent, fie printr-un sumator cu transport anticipat, de dimensiuni mai mari dar rapid. De asemenea, se pot utiliza mai multe tipuri de componente, ca de exemplu un sumator, un sumator/scăzător, sau o unitate aritmetică şi logică, pentru execuția unei operații de adunare.

Selecția unităților determină numărul şi tipul diferitelor unități funcționale şi de memorie din biblioteca de componente. O cerință de bază pentru selecția unităților este că numărul de unități care execută un anumit tip de operație trebuie să fie egal cu sau mai mare decât numărul maxim de operații de acel tip care trebuie executate în fiecare pas de control. Selecția unităților este combinată în mod frecvent cu asignarea unităților într-o operație numită alocare.

4.3.2. Asignarea unităților funcționale

După selecția tuturor unităților funcționale, operațiile din descrierea funcțională trebuie implementate prin setul de unități funcționale selectate. În toate cazurile când există operații care pot fi implementate prin mai multe tipuri de unități funcționale, este necesar un algoritm de asignare a unităților funcționale pentru determinarea asocierii exacte a operațiilor cu aceste unități.

De exemplu, operațiile o_1 şi o_3 din **Figura 38** au fost asignate sumatorului *SUM1*, iar operațiile o_2 şi o_4 au fost asignate sumatorului *SUM2*.

4.3.3. Asignarea unităților de memorie

Prin această operație se asociază constantele, variabilele şi structurile de date din descrierea funcțională cu elemente de memorie din calea de date. Constantele, ca de exemplu coeficienții dintr-un algoritm DSP, sunt memorate de obicei într-o memorie de tip ROM. Variabilele sunt memorate în registre sau memorii. Variabilele ale căror durate de viață nu se suprapun între ele pot partaja acelaşi registru sau locație de memorie. Durata de viață a unei variabile este intervalul de timp între prima asignare a unei valori la acea variabilă (prima apariție a variabilei în partea stângă a unei instrucțiuni de asignare) şi ultima utilizare a variabilei (ultima apariție a variabilei în partea dreaptă a unei instrucțiuni de asignare).

După asignarea variabilelor la registre, registrele pot fi reunite într-un tablou de registre cu un singur port de acces dacă registrele nu sunt accesate simultan. Similar, registrele pot fi reunite într-un tablou multiport, dacă numărul de registre accesate în fiecare pas de control nu depăşeşte numărul de porturi.

4.3.4. Asignarea interconexiunilor

Fiecare transfer de date necesită o cale de interconectare de la sursă la destinație. Două transferuri de date pot partaja în întregime sau parțial o cale de interconectare dacă ele nu au loc simultan. De exemplu, în **Figura 38**, citirea variabilei *b* în pasul de control s_1 şi a variabilei *e* în pasul de control s_2 poate fi realizată prin utilizarea aceleiaşi unități de interconectare. Însă, scrierea variabilelor *e* şi *f*, care se execută simultan în pasul de control s_1, trebuie realizată prin utilizarea unor căi diferite.

Obiectivul asignării interconexiunilor este de a se maximiza partajarea unităților de interconectare şi deci de a se minimiza costul interconexiunilor, asigurând în acelaşi timp realizarea fără conflicte a transferurilor de date cerute de descrierea la nivelul transferurilor între registre.

4.3.5. Interdependența operațiilor

Toate operațiile legate de sinteza căii de date (planificarea, selecția unităților, asignarea unităților funcționale, a elementelor de memorie şi a interconexiunilor) sunt interdependente. În particular, asignarea unităților funcționale, a elementelor de memorie şi a interconexiunilor sunt strâns legate între ele. De exemplu, **Figura 45** ilustrează modul în care asignarea unităților funcționale şi a elementelor de memorie afectează alocarea interconexiunilor. Presupunem că cele opt variabile (*a* .. *g*) din graful fluxului de date (**Figura 45(a)**) au fost partiționate în patru registre după cum urmează: $r_1 \leftarrow \{a\}$, $r_2 \leftarrow \{b, e, g\}$, $r_3 \leftarrow \{c, f, h\}$, $r_4 \leftarrow \{d\}$. Fiind date două sumatoare, *SUM1* şi *SUM2*, există două moduri de grupare a celor patru operații de adunare, o_1, o_2, o_3 şi o_4, astfel ca fiecare grup să fie asignat unui sumator:

(1) $SUM1 \leftarrow \{o_1, o_4\}$, $SUM2 \leftarrow \{o_2, o_3\}$, sau

(2) $SUM1 \leftarrow \{o_1, o_3\}$, $SUM2 \leftarrow \{o_2, o_4\}$.

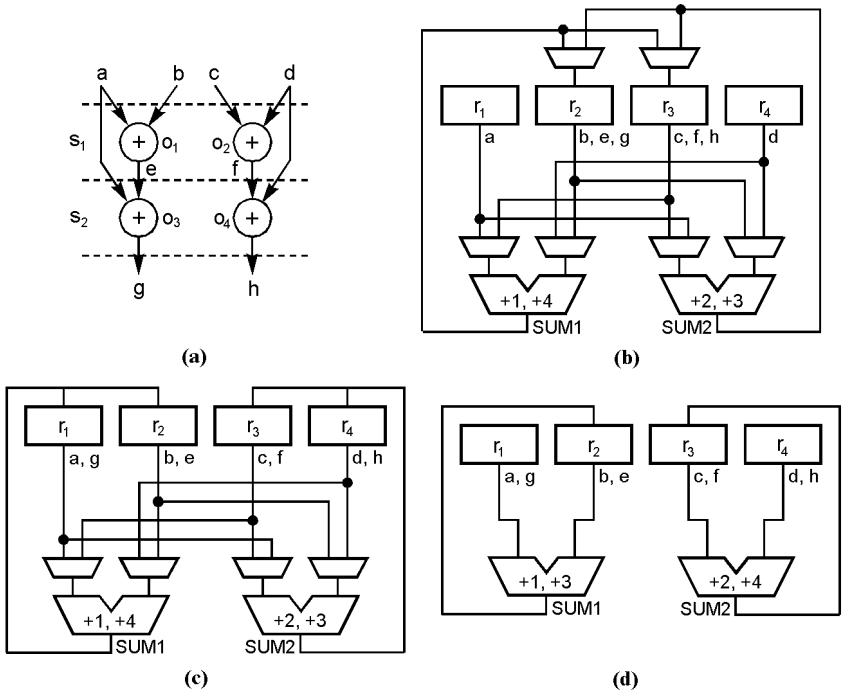


Figura 45. Interdependența dintre asignarea unităților funcționale şi de memorie: (a) GFD planificat; (b) asignarea registrelor, fiind necesare şase multiplexoare; (c) reducerea numărului de multiplexoare, obținută prin realocarea registrelor; (d) soluție optimă fără multiplexoare, obținută prin modificarea asignării unităților funcționale.

Pentru asignarea dată a registrelor, sunt necesare şase multiplexoare 2:1 pentru interconectarea unităților în cazul (1) (**Figura 45(b)**). Se pot elimina două multiplexoare

(**Figura 45(c)**), prin modificarea asignării registrelor, astfel: $r_1 \leftarrow \{a, g\}$, $r_2 \leftarrow \{b, e\}$, $r_3 \leftarrow \{c, f\}$, $r_4 \leftarrow \{d, h\}$.

Dacă se modifică asignarea unităților funcționale ca în cazul (2), nu sunt necesare multiplexoare (**Figura 45(d)**). Astfel, asignarea este optimă, dacă costul interconexiunilor este măsurat prin numărul de multiplexoare necesare. Rezultă că atât asignarea unităților funcționale, cât şi cea a elementelor de memorie afectează în mod potențial optimizarea care se poate realiza prin alocarea interconexiunilor.

Exemplul precedent ridică de asemenea problema ordinii în care se execută operațiile componente ale alocării. Cerințele în privința interconexiunilor devin clare după s-a executat atât alocarea unităților funcționale, cât şi cea a elementelor de memorie. Mai mult, alocarea unităților funcționale poate lua decizii corecte dacă alocarea elementelor de memorie este realizată în prealabil, şi invers. Dacă se alege executarea unei operații înaintea celeilalte, prima operație aleasă nu poate utiliza informațiile de la operația a doua.

4.4. Metode constructive de tip greedy

O metodă constructivă porneşte de la o cale de date vidă, şi adaugă la aceasta unități funcționale, de memorie şi de interconectare, după cum este necesar. Pentru fiecare operație, metoda încearcă găsirea unei unități funcționale din calea de date construită parțial, care poate executa operația şi este inactivă în pasul de control în care operația trebuie executată. În cazul în care există două sau mai multe unități funcționale care îndeplinesc aceste condiții, se alege cea care determină o creştere minimă a costului de interconectare. Dacă nici o unitate funcțională din calea de date construită parțial nu îndeplineşte condițiile, se adaugă o nouă unitate funcțională din biblioteca de componente care poate executa operația.

Similar, se poate asigna o variabilă la un registru disponibil numai dacă durata sa de viață nu se suprapune cu cea a variabilelor asignate deja la acest registru. Un nou registru este alocat numai dacă nici unul din registrele alocate nu îndeplinesc condiția anterioară. Dacă există mai multe alternative pentru asignarea unei variabile la un registru, se selectează cea care determină o creştere minimă a costului căii de date.

Algoritmul din **Figura 46** descrie metoda de alocare constructivă de tip greedy. Fie *EFN* setul de entități funcționale nealocate şi CD_{curent} calea de date proiectată parțial. Entitățile funcționale considerate pot fi variabile care trebuie asignate registrelor, operații care trebuie asignate unităților funcționale, sau transferuri de date care trebuie asignate unor unități de interconectare. Inițial, CD_{curent} este vid. Procedura *ADD*(*CD*, *efn*) modifică structural calea de date *CD* prin adăugarea componentelor necesare pentru entitatea funcțională *efn*. Funcția *COST*(*CD*) evaluează costul unei căi de date proiectate parțial (*CD*) din punct de vedere al spațiului şi al performanțelor. CD_{temp} este o cale de date temporară creată în scopul evaluării costului c_{temp} de efectuare a fiecărei modificări a căii de date CD_{curent}.

```
CD_curent = φ ;
while EFN ≠ φ do
    CostMinim = ∞;
    for toate efn ∈ EFN do
        CD_temp = ADD(CD_curent, efn);
        c_temp = COST(CD_temp);
        if c_temp < CostMinim then
            CostMinim = c_temp;
            EntitateOpt = efn;
        endif
    endfor
    CD_curent = ADD(CD_curent, EntitateOpt);
    EFN = EFN - EntitateOpt;
endwhile
```

Figura 46. Algoritmul de alocare constructivă.

Pornind de la setul *EFN*, bucla *for* determină entitatea funcţională nealocată care conduce la creşterea minimă a costului (*EntitateOpt*), dacă este adăugată la calea de date. Aceasta se realizează prin adăugarea individuală a fiecărei entităţi funcţionale nealocate din setul *EFN* la CD_{curent}, şi evaluarea costului rezultat. Procedura *ADD* modifică apoi CD_{curent} prin adăugarea entităţii *EntitateOpt* la calea de date. *EntitateOpt* este apoi ştearsă din setul entităţilor funcţionale nealocate. Iteraţiile din bucla *while* continuă până când toate entităţile funcţionale au fost alocate (deci $EFN = \phi$).

Pentru a se utiliza metoda constructivă, trebuie abordate două probleme: calculul funcţiei de cost şi ordinea în care entităţile funcţionale nealocate sunt adăugate la calea de date. Această ordine poate fi determinată în mod static sau dinamic. În cazul ordonării statice, obiectele sunt ordonate înainte de a începe construirea căii de date. Ordinea nu este modificată în timpul procesului de alocare. În cazul ordonării dinamice, pentru selecţia unei operaţii sau variabile care va fi adăugată la calea de date, se evaluează fiecare entitate funcţională nealocată pe baza costului de modificare a căii de date parţiale, alegându-se entitatea care necesită modificarea cea mai puţin costisitoare. După fiecare alocare, se reevaluează costurile asociate cu entităţile rămase nealocate. Algoritmul prezentat utilizează metoda dinamică.

Metoda constructivă se încadrează în categoria algoritmilor greedy. Deşi aceşti algoritmi sunt simpli, în multe cazuri soluţia obţinută nu este optimă.

4.5. Metoda de partiţionare

Pentru a se îmbunătăţi calitatea rezultatelor alocării, au fost propuse metode de partiţionare, în care procesul de alocare este divizat într-o secvenţă de operaţii independente; fiecare operaţie este transformată într-o problemă bine definită în cadrul teoriei grafurilor, fiind apoi soluţionată printr-o tehnică cunoscută.

În timp ce o metodă constructivă poate alterna etapele de alocare a unităților funcționale, de memorie şi de interconectare, metodele de partiționare vor termina o operație înaintea execuției alteia. Din cauza interdependențelor între aceste operații, nu se garantează o soluție optimă chiar dacă toate operațiile sunt soluționate în mod optim. De exemplu, un circuit care utilizează trei sumatoare poate necesita un număr mai mic de multiplexoare decât cel care utilizează numai două sumatoare. Astfel, o strategie de alocare care minimizează utilizarea sumatoarelor este justificată numai dacă costul unui sumator este mai ridicat decât cel al unui multiplexor.

Se va descrie o tehnică de alocare bazată pe o metodă din teoria grafurilor. Cele trei operații de alocare a unităților funcționale, de memorie şi de interconectare pot fi soluționate independent prin metoda partiționării grafurilor în grupuri.

```
/* crează un super-graf G'(S, E'); */
S = φ ; E' = φ ;
for fiecare v_i ∈ V do  s_i = {v_i}; S = S ∪ {s_i}; endfor
for fiecare e_i,j ∈ E do  E' = E' ∪ {e'_i,j}; endfor
while E' ≠ φ  do
    /* determină s_Index1, s_Index2 cu numărul maxim de vecini comuni */
    MaxComuni = -1;
    for fiecare e'_i,j ∈ E' do
        c_i,j = | VECIN_COMUN(G', s_i, s_j) |;
        if c_i,j > MaxComuni then
            MaxComuni = c_i,j;
            Index1 = i; Index2 = j;
        endif
    endfor
    SetComun = VECIN_COMUN(G', s_Index1, s_Index2);
    /* şterge toate muchiile care conectează s_Index1 sau s_Index2 */
    E' = ŞTERGE_MUCHIE(E', s_Index1);
    E' = ŞTERGE_MUCHIE(E', s_Index2);
    /* unește s_Index1 şi s_Index2 în s_Index1Index2 */
    s_Index1Index2 = s_Index1 ∪ s_Index2;
    S = S - s_Index1 - s_Index2;
    S = S ∪ {s_Index1Index2};
    /* adaugă muchii de la s_Index1Index2 la super-nodurile din SetComun */
    for fiecare s_i ∈ SetComun do
        E' = E' ∪ {e'_i,Index1Index2};
    endfor
endwhile
```

Figura 47. Algoritmul de partiționare în grupuri.

Fie $G = (V, E)$ un graf, unde V este setul vârfurilor şi E setul muchiilor. Fiecare muchie $e_{i,j} \in E$ conectează două vârfuri diferite v_i şi $v_j \in V$. Un subgraf SG al grafului G este definit ca (SV, SE), unde $SV \subseteq V$ şi $SE = \{e_{i,j} \mid e_{i,j} \in E, v_i, v_j \in SV\}$. Un graf este

complet dacă şi numai dacă pentru fiecare pereche a vârfurilor sale există o muchie care le conectează. Un *grup* al grafului G este un subgraf complet al grafului G. Problema partiționării unui graf într-un număr minim de grupuri astfel încât fiecare nod să aparțină unui singur grup este numită *partiționare în grupuri*. Pentru soluționarea acestei probleme se utilizează de obicei metode euristice.

Algoritmul din **Figura 47** descrie o metodă euristică propusă de *Tseng* şi *Siewiorek* pentru soluționarea problemei de partiționare în grupuri. Din graful original $G(V, E)$ se obține un super-graf $G'(S, E')$. Fiecare nod $s_i \in S$ este un super-nod care poate conține un set de unul sau mai multe vârfuri $v_i \in V$. E' este identic cu E, cu excepția faptului că muchiile din E' conectează super-noduri din S. Un super-nod $s_i \in S$ este un vecin comun al super-nodurilor s_j şi $s_k \in S$ dacă există muchiile $e_{i,j}$ şi $e_{i,k} \in E'$. Funcția *VECIN_COMUN*(G', s_i, s_j) returnează setul de super-noduri care sunt vecini comuni ai s_i şi s_j din G'. Procedura *ŞTERGE_MUCHIE*(E', s_i) şterge toate muchiile din E' pentru care s_i este un super-nod terminator.

Inițial, fiecare vârf $v_i \in V$ al grafului G este plasat într-un super-nod separat $s_i \in S$ al grafului G'. În fiecare pas, algoritmul determină super-nodurile s_{Index1} şi s_{Index2} din S care sunt conectate printr-o muchie şi au numărul maxim de vecini comuni. Setul *SetComun* conține toți vecinii comuni ai s_{Index1} şi s_{Index2}. Toate muchiile care pornesc de la s_{Index1} şi s_{Index2} din G' sunt şterse. Cele două super-noduri găsite sunt unite într-un singur super-nod, $s_{Index1Index2}$, care conține toate vârfurile lui s_{Index1} şi s_{Index2}. Sunt adăugate noi muchii de la $s_{Index1Index2}$ la toate super-nodurile din *SetComun*. Operațiile de sus se repetă până când nu mai există muchii în graf. Vârfurile conținute în fiecare super-nod $s_i \in S$ formează un grup al grafului G.

Figura 48 ilustrează acest algoritm. În graful din **Figura 48(a)**, $V = \{v_1, v_2, v_3, v_4, v_5\}$ şi $E = \{e_{1,3}, e_{1,4}, e_{2,3}, e_{2,5}, e_{3,4}, e_{4,5}\}$. Inițial, fiecare vârf este plasat într-un super-nod separat (s_1 .. s_5 în **Figura 48(b)**). Muchiile $e'_{1,3}$, $e'_{1,4}$ şi $e'_{3,4}$ ale super-grafului G' au numărul maxim de vecini comuni dintre toate muchiile (**Figura 48(b)**). Este selectată prima muchie, $e'_{1,3}$, şi se execută paşii următori pentru a obține graful din **Figura 48(c)**:

1. s_4, singurul vecin comun al s_1 şi s_3, este depus în *SetComun*.
2. Se şterg toate muchiile care conectează super-nodurile s_1 şi s_3 cu alte super-noduri ($e'_{1,3}$, $e'_{1,4}$, $e'_{2,3}$ şi $e'_{3,4}$).
3. Super-nodurile s_1 şi s_3 sunt combinate într-un nou super-nod s_{13}.
4. Se adaugă o muchie între s_{13} şi fiecare super-nod din *SetComun*; deci este adăugată muchia $e_{13,4}$.

În următoarea iterație, s_4 este unit cu s_{13}, obținându-se super-nodul s_{134} (**Figura 48(d)**). În final, s_2 şi s_5 sunt unite în super-nodul s_{25} (**Figura 48(e)**). Grupurile sunt $s_{134} = \{v_1, v_3, v_4\}$ şi $s_{25} = \{v_2, v_5\}$ (**Figura 48(f)**).

În scopul aplicării tehnicii de partiționare în grupuri pentru problema alocării, trebuie ca din descrierea respectivă să se obțină mai întâi modelul sub forma unui graf. Considerăm ca exemplu alocarea registrelor. Scopul principal al alocării registrelor este de a se minimiza costul registrelor prin partajarea la maxim a registrelor comune între variabile. Pentru soluționarea problemei alocării registrelor, se construieşte un graf $G = (V, E)$, în care fiecare vârf $v_i \in V$ reprezintă în mod unic o variabilă v_i, şi există o muchie $e_{i,j} \in E$

dacă şi numai dacă variabilele v_i şi v_j pot fi memorate în acelaşi registru (deci, dacă duratele lor de viață nu se suprapun). Toate variabilele ale căror vârfuri corespunzătoare se află într-un grup al grafului G pot fi memorate într-un singur registru. O partiționare în grupuri a grafului G reprezintă o soluție pentru problema alocării elementelor de memorie din calea de date, care necesită un număr minim de registre.

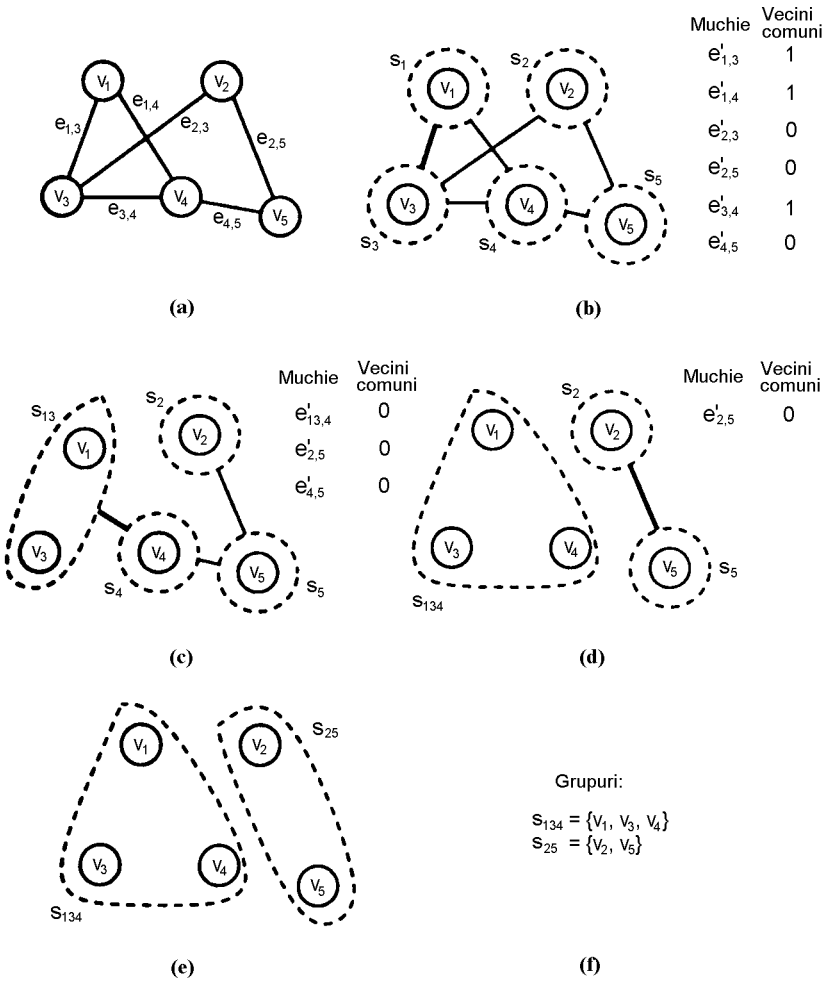


Figura 48. Partiționarea în grupuri: (a) graful dat G; (b) determinarea vecinilor comuni pentru muchiile grafului G'; (c) super-nodul s_{13} format prin considerarea muchiei $e'_{1,3}$; (d) super-nodul s_{134} format prin considerarea muchiei $e'_{13,4}$; (e) super-nodul s_{25} format prin considerarea muchiei $e'_{2,5}$; (e) grupurile rezultate s_{134} şi s_{25}.

Figura 49 prezintă o soluție a problemei alocării registrelor obținută prin utilizarea algoritmului de partiționare în grupuri.

Atât alocarea unităților funcționale, cât şi alocarea interconexiunilor pot fi formulate ca probleme de partiționare în grupuri. Pentru alocarea unităților funcționale, fie-

care vârf al grafului reprezintă o operație. Între două vârfuri există o muchie dacă sunt satisfăcute două condiții:

1. cele două operații sunt planificate în paşi de control diferiți, şi
2. există o unitate funcțională care poate executa ambele operații.

O soluție de partiționare a acestui graf va reprezenta o soluție pentru problema alocării unităților funcționale. Deoarece fiecărui grup i se asignează o unitate funcțională, toate operațiile ale căror vârfuri corespunzătoare se află într-un grup sunt executate de aceeaşi unitate funcțională.

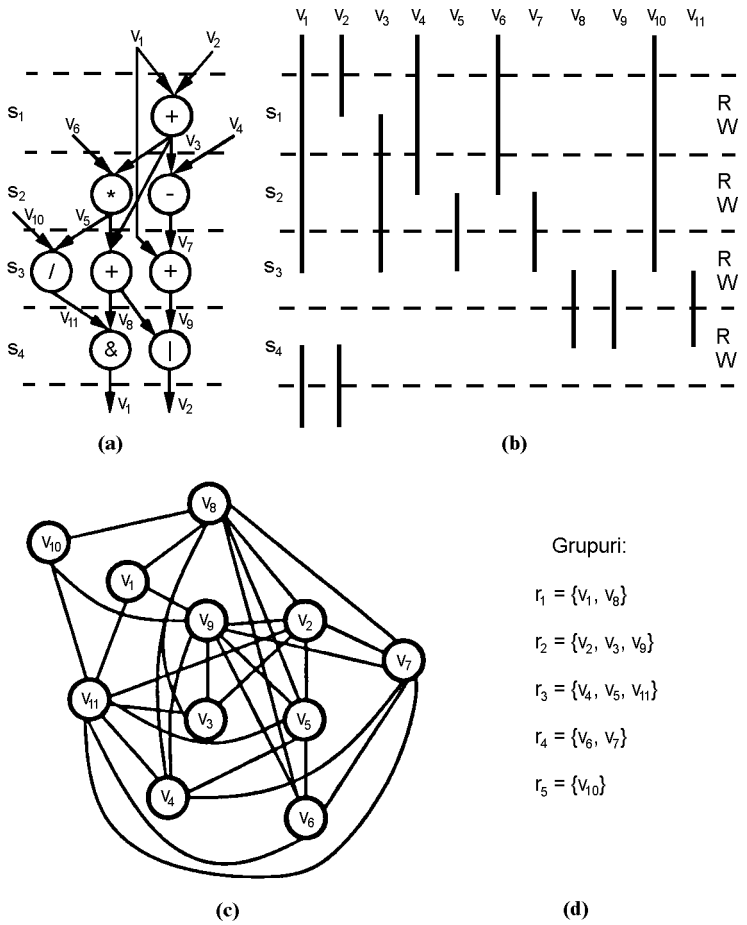


Figura 49. Alocarea registrelor utilizând partiționarea în grupuri: (a) GFD planificat; (b) duratele de viață ale variabilelor; (c) graful de alocare a registrelor; (d) soluția partiționării în grupuri.

Pentru alocarea unităților de interconectare, fiecare vârf corespunde unei conexiuni între două unități. Între două vârfuri există o muchie dacă cele două conexiuni nu sunt utilizate în mod concurent, în fiecare pas de control. O soluție de partiționare a unui asemenea graf implică partiționarea conexiunilor în magistrale sau multiplexoare. Cu alte cuvinte, toate conexiunile ale căror vârfuri corespunzătoare se află în acelaşi grup utilizează aceeaşi magistrală sau multiplexor.

Deşi metoda de partiționare aplicată alocării unităților de memorie poate minimiza cerințele de memorie, această metodă ignoră interdependența dintre alocarea unităților de memorie şi cea a interconexiunilor. Metoda precedentă a fost extinsă de *Paulin* şi *Knight* prin asocierea unor ponderi cu muchiile grafului, care reflectă impactul asupra complexității interconexiunilor datorită partajării registrelor de către variabile. Unei muchii i se asociază o pondere mai mare dacă partajarea unui registru de cele două variabile corespunzătoare celor două vârfuri ale muchiei reduce costul interconexiunilor. Pe de altă parte, muchiei i se asociază o pondere mai mică dacă partajarea determină o creştere a costului de interconectare. Algoritmul modificat va selecta grupurile care au muchii cu ponderi mai mari. Astfel, este probabil că partajarea unui registru comun de către variabile va reduce costul de interconectare.

4.6. Metoda de rafinare iterativă

Fiind dată o cale de date a cărei sinteză s-a realizat prin metode constructive sau de partiționare, calitatea acesteia poate fi îmbunătățită prin realocare. Ca un exemplu, considerăm realocarea unităților funcționale. Este posibilă reducerea costului de interconectare prin simpla interschimbare a asignărilor unităților funcționale pentru două operații. De exemplu, dacă se porneşte de la calea de date din **Figura 45(c)**, interschimbarea asignărilor unităților funcționale pentru operațiile o_3 şi o_4 va reduce costul de interconectare prin patru multiplexoare 2:1 (**Figura 45(d)**).

De asemenea, poate fi utilă modificarea unor asignări ale variabilelor la registre. În **Figura 45(b)**, dacă se modifică asignarea variabilei *g* de la registrul r_2 la registrul r_1, şi a variabilei *h* de la r_3 la r_4, se obține o cale de date îmbunătățită, cu un număr mai redus de multiplexoare (**Figura 45(c)**).

Problemele principale în cazul metodei de rafinare iterativă sunt legate de tipul modificărilor care trebuie efectuate asupra căii de date, selecția unui tip de modificare în timpul unei iterații şi criteriul de terminare pentru procesul de rafinare.

Cea mai simplă metodă poate fi o simplă schimbare a asignărilor. În cazul acestei metode, modificarea efectuată asupra căii de date este limitată la interschimbarea a două asignări (deci, a unor perechi de variabile sau perechi de operații). Algoritmul de interschimbare a perechilor execută o serie de modificări în calea de date, în scopul reducerii costului acesteia. Întâi, se evaluează toate interschimbările posibile ale asignărilor operațiilor planificate în acelaşi pas de control. Evaluarea se realizează pe baza reducerii costului căii de date datorită unei modificări a interconexiunilor. Apoi, se alege interschimbarea care determină câştigul maxim, şi calea de date este actualizată conform modificării efectuate. Acest proces este repetat până când interschimbările nu mai conduc la un câştig pozitiv.

Algoritmul din **Figura 50** descrie această metodă. Fie CD_{curent} structura curentă a căii de date, iar CD_{temp} o cale de date temporară creată pentru evaluarea costului fiecărei interschimbări a asignării operațiilor. Funcția *COST*(*CD*) evaluează costul căii de date *CD*. Costurile căilor de date CD_{curent} şi CD_{temp} sunt reprezentate de c_{curent} şi c_{temp}. Procedura *SWAP*(*CD*, o_i, o_j) interschimbă asignările operațiilor o_i şi o_j de acelaşi tip, şi actualizează calea de date *CD*. În fiecare iterație a buclei interioare, *CâştigCurent* reprezintă

reducerea costului căii de date datorită interschimbării operațiilor în acea iterație. *CâştigMax* păstrează reducerea maximă a costului care poate fi obținută printr-o singură interschimbare a operațiilor.

```
repeat
    CâştigMax = –∞;
    c_curent = COST(CD_curent);
    for toți pașii de control s do
        for fiecare o_i, o_j de acelaşi tip planificate în pasul s, i ≠ j do
            CD_temp = SWAP(CD_curent, o_i, o_j);
            c_temp = COST(CD_temp);
            CâştigCurent = c_curent - c_temp;
            if CâştigCurent > CâştigMax then
                CâştigMax = CâştigCurent;
                OpOpt1 = o_i; OpOpt2 = o_j;
            endif
        endfor
    endfor
    if CâştigMax > 0 then
        CD_curent = SWAP(CD_curent, OpOpt1, OpOpt2);
    endif
until CâştigMax ≤ 0
```

Figura 50. Algoritmul de interschimbare a perechilor.

Această metodă are două dezavantaje. În primul rând, utilizarea numai a interschimbării perechilor poate fi necorespunzătoare pentru explorarea tuturor posibilităților de rafinare. De exemplu, nici o interschimbare a variabilelor nu poate conduce la modificarea căii de date din **Figura 45(b)** pentru a se obține calea de date din **Figura 45(d)**. În al doilea rând, strategia greedy de a alege întotdeauna cea mai profitabilă modificare poate conduce procesul de rafinare la un minim local. În timp ce pentru rezolvarea celei de-a doua probleme se poate utiliza o metodă probabilistică, cu prețul unui timp de calcul mai ridicat, prima problemă necesită o soluție mai sofisticată, ca de exemplu interschimbarea mai multor asignări în acelaşi timp.

Presupunem că operația o_i a fost asignată unității funcționale uf_j şi una din variabilele sale de intrare a fost asignată la registrul r_k. Eliminarea operației o_i din unitatea uf_j nu va elimina interconexiunea de la r_k la uf_j, dacă există o altă operație asignată anterior unității uf_j, cu variabilele de intrare asignate registrului r_k. Procesul de rafinare iterativă trebuie să abordeze problema prin considerarea simultană a unor obiecte multiple. Trebuie să se țină cont de relațiile dintre entitățile de diferite tipuri. De exemplu, câştigul obținut la realocarea unei operații poate fi mai ridicat dacă variabilele sale de intrare sunt de asemenea realocate simultan.

5. Concluzii

În procesul de sinteză se porneşte de la specificaţia funcţionării unui sistem numeric şi de la un set de restricţii, urmărindu-se obținerea unei structuri care implementează specificaţia dorită şi satisface restricţiile. La nivelul algoritmic, specificaţia se prezintă sub forma unui algoritm. Rezultatul *sintezei de nivel înalt* este o descriere a unui sistem numeric sincron în domeniul structural, la nivelul transferurilor între registre.

Translatarea descrierii algoritmice într-o descriere structurală nu este unică. Implementarea poate varia de la soluții secvențiale la soluții complet paralele. Compromisul principal care trebuie rezolvat în cadrul sintezei de nivel înalt este cel între modelarea serială şi cea paralelă, deci între o implementare eficientă din punct de vedere al spațiului, dar lentă, şi una costisitoare din punct de vedere al spațiului, dar rapidă.

Există mai multe etape care trebuie parcurse pentru transformarea unei descrieri algoritmice într-o structură corespunzătoare la nivelul transferurilor între registre.

- Prima etapă este de a obține o *reprezentare internă* bazată pe grafuri, echivalentă cu decrierea algoritmică, atât pentru fluxul de date cât şi pentru fluxul de control.
- Asupra reprezentării interne pot fi efectuate operații de transformare, numite *transformări funcționale*.
- Transformarea fluxului de date şi de control într-o structură la nivelul transferurilor între registre constă din următoarele operații principale: *planificarea*, *alocarea resurselor* şi *asignarea resurselor*.
- Înaintea efectuării operației de planificare, se poate executa *partiționarea* sistemului, care constă în asignarea operațiilor unor grupuri de unități funcționale şi alocarea acestor grupuri pe baza distanței existente între componentele grupurilor.

Deoarece planificarea, alocarea resurselor şi asignarea resurselor sunt interdependente, modulele de sinteză corespunzătoare acestor etape nu sunt independente, şi nu există o anumită ordine prestabilită în care trebuie efectuate aceste operații. Soluționarea tuturor problemelor într-un mod optim nu este, în general, posibilă. Au fost dezvoltate diferite strategii pentru a elimina dependențele între diferitele etape, şi metode euristice care permit rezultate apropiate de cele optime pentru etape separate.

Planificarea şi alocarea pot fi considerate ca probleme de partiționare. Algoritmii de planificare partiționează asignarea variabilelor şi operațiile în intervale de timp, iar alocarea le partiționează în unități funcționale şi de memorie.

Dacă planificarea este executată înainte de alocare, aceasta impune restricţii suplimentare asupra alocării. De exemplu, două operaţii planificate în acelaşi interval de timp nu pot fi executate de aceeaşi unitate funcţională. Similar, dacă alocarea este executată înaintea planificării, rezultă restricţii asupra planificării. De exemplu, două operaţii asignate la aceeaşi unitate funcţională nu pot fi executate în acelaşi interval de timp.

Planificarea şi alocarea nu sunt singurii algoritmi necesari pentru sinteza de nivel înalt. După ce operaţiile sunt asignate intervalelor de timp şi unităţilor, sunt necesari algoritmi suplimentari pentru generarea şi optimizarea unităţilor funcţionale, de memorie şi de control. Sinteza unităţilor de control este numită sinteza secvenţială, iar sinteza unităţilor funcţionale este numită sinteza logică. Sinteza unităţilor de memorie poate fi realizată prin metode de sinteză secvenţială şi combinaţională.

Între modelele semantice ale limbajelor de descriere hardware şi arhitecturile utilizate pentru sinteză pot exista diferenţe importante. Din acest motiv, este necesară o *reprezentare canonică intermediară* care facilitează implementarea descrierilor hardware prin diferite arhitecturi, utilizând diferite sisteme de sinteză.

Pornind de la descrierea funcţională, în timpul sintezei de nivel înalt sunt adăugate mai multe tipuri de informaţii reprezentării intermediare: stări, unităţi RT şi conexiuni, informaţii de control a căii de date, şi informaţii de secvenţiere a stărilor. Este necesară corelarea acestor informaţii suplimentare cu descrierea funcţională iniţială, utilizând adnotări şi/sau legături de asignare. Reprezentarea intermediară a proiectului trebuie deci să conţină în mod explicit asignarea stărilor, selecţia şi asignarea unităţilor, structura RT interconectată, şi controlul simbolic. O asemenea reprezentare permite parcurgerea tuturor fazelor sintezei de nivel înalt, de la specificaţia abstractă la implementarea finală, prin reprezentarea rezultatelor intermediare ale sintezei, ca şi a legăturilor între etapele intermediare ale sintezei.

Compilarea modelelor hardware la nivel arhitectural implică o *analiză semantică* completă, care cuprinde analiza *fluxului de date* şi a *fluxului de control*, şi *verificări de tip*. *Analiza semantică* a arborilor sintactici conduce la generarea unei forme intermediare, care reprezintă implementarea descrierii originale pe o maşină abstractă. Astfel, se pot executa optimizări funcţionale asupra unui asemenea model, abstractizând parametrii tehnologici ai circuitului. *Analiza fluxului de date şi de control* determină structura ierarhică a grafului utilizat şi topologia entităţilor sale. Această analiză cuprinde mai multe operaţii, fiind utilizată ca o bază pentru optimizarea funcţională. Ea include extragerea duratei de viaţă a variabilelor.

Descrierile unităţilor hardware sunt reprezentate de obicei prin grafuri, cum este modelul GFCD. Aceste grafuri diferă între ele prin modul de reprezentare a construcţiilor de control şi reprezentarea transferurilor de date în cadrul unui graf al fluxului de date. Au fost prezentate trei *reprezentări bazate pe grafuri* care sunt utilizate în sinteza de nivel înalt: reprezentări disjuncte ale fluxului de control şi de date; reprezentări hibride ale fluxului de control şi de date; reprezentări prin arbori sintactici.

Reprezentarea iniţială prin grafuri ale fluxului de date şi de control este apropiată de descrierea originală, şi deci poate reprezenta construcţii sintactice ale limbajului care nu sunt utile sau relevante pentru sinteză. Asupra grafului iniţial pot fi aplicate *transfor-*

mări înaintea etapelor de planificare şi alocare, creând un alt graf care este mai potrivit pentru aceste etape ale sintezei. S-au prezentat următoarele categorii de transformări:

Transformări efectuate de compilator: propagarea constantelor şi a variabilelor; eliminarea operatorilor redundanți; reducerea numărului de accesuri la tablouri; reducerea complexității operatorilor; transformarea codului; înlocuirea construcțiilor sintactice specifice.

Transformări ale grafurilor: reducerea înălțimii arborilor; transformarea fluxului de control într-un flux de date; aplatizarea grafurilor fluxului de control şi de date; expandarea buclelor; expandarea construcțiilor condiționale.

Transformări specifice unităților hardware: transformări hardware la nivelul logic, RT şi sistem. Aestea sunt transformări locale care utilizează proprietățile unităților hardware la diferite nivele de proiectare pentru a optimiza reprezentarea intermediară.

Planificarea operațiilor este o etapă importantă în sinteza de nivel înalt deoarece influențează compromisul între performanță şi cost. Algoritmii de planificare trebuie adaptați în funcție de următoarele aspecte:

- arhitecturile utilizate pentru implementare;
- tipul unităților funcționale şi de memorie utilizate şi topologiile de interconectare ale acestora;
- construcțiile limbajului utilizat. Descrierile funcționale care conțin construcții condiționale şi de buclare necesită tehnici mai complexe de planificare.

Operațiile de planificare şi de alocare a unităților sunt interdependente. Caracterizarea calității unui algoritm de planificare dat este dificilă fără considerarea algoritmilor care execută alocarea. Două operații diferite de planificare cu acelaşi număr de paşi de control şi care necesită acelaşi număr de unități funcționale pot avea ca rezultat proiecte cu măsuri calitative substanțial diferite după executarea alocării.

Planificarea cu restricții de timp este importantă pentru sistemele destinate aplicațiilor de timp real. Algoritmii de planificare cu restricții de timp poate utiliza trei tehnici diferite: programarea matematică, euristici constructive şi rafinarea iterativă. Au fost prezentate metoda programării liniare (ca exemplu de programare matematică), o metodă euristică constructivă şi o tehnică de planificare iterativă.

Metoda programării liniare nu este practică pentru descrieri de dimensiuni mari; de aceea au fost dezvoltate metode euristice care pot fi executate în mod eficient. Eficiența crescută a acestor metode este obținută prin eliminarea revenirii din metoda programării liniare. Costul circuitului rezultat depinde în mare măsură de selecția următoarei operații care va fi planificată şi de asignarea acestei operații pasului de control cel mai potrivit.

Metoda euristică constructivă construieşte o soluție fără a efectua nici o revenire. Decizia de a planifica o operație într-un pas de control este luată pe baza unui graf al fluxului de date parțial planificat; nu se ține cont de asignările ulterioare ale operatorilor la acelaşi pas de control. Soluția rezultată nu este optimă, datorită lipsei unei strategii de anticipare şi a lipsei compromisurilor între deciziile inițiale şi cele întârziate.

Planificarea iterativă permite obţinerea unor performanţe mai ridicate prin replanificarea unor operaţii în cadrul unei planificări date. Aspectele esenţiale legate de replanificare sunt: alegerea unui candidat pentru replanificare, procedura de replanificare şi controlul procesului de îmbunătăţire. S-a descris o metodă bazată pe paradigma propusă iniţial pentru problema de bisecţie a grafurilor de *Kernighan* şi *Lin*.

Planificarea cu restricţii de resurse este întâlnită în numeroase aplicaţii unde există o limitare dată de spaţiul de pe cip. Restricţia este specificată de obicei sub forma numărului de unităţi funcţionale sau a spaţiului disponibil. Au fost prezentaţi doi algoritmi de planificare cu restricţii de resurse: metoda bazată pe liste şi metoda de planificare cu liste statice.

Alocarea căii de date constă din două operaţii principale: selecţia unităţilor şi asignarea unităţilor. *Selecţia unităţilor* determină numărul şi tipul componentelor RT care vor fi utilizate. *Asignarea unităţilor* implică asocierea variabilelor şi operaţiilor din graful planificat al fluxului de control şi de date cu unităţile funcţionale, de memorie şi de interconectare, asigurând funcţionarea corectă a implementării pentru setul selectat al componentelor.

Alocarea căii de date constă din următoarele operaţii diferite, dar interdependente: selecţia modulelor, alocarea unităţilor funcţionale, alocarea unităţilor de memorie şi alocarea interconexiunilor.

Problema alocării se poate soluţiona în trei moduri: prin metode de tip "greedy", care construiesc o implementare în mod progresiv prin traversarea GFCD; prin descompunerea problemei alocării în părţile sale constituente şi soluţionarea separată a acestora; prin metode iterative, care încearcă combinarea soluţiilor subproblemelor alocării.

Pentru a se utiliza *metoda constructivă*, trebuie abordate două probleme: calculul funcţiei de cost şi ordinea în care entităţile funcţionale nealocate sunt adăugate la calea de date. Această ordine poate fi determinată în mod static sau dinamic.

Metodele de partiţionare au fost propuse pentru a se îmbunătăţi calitatea rezultatelor alocării. Procesul de alocare este divizat într-o secvenţă de operaţii independente; fiecare operaţie este transformată într-o problemă bine definită în cadrul teoriei grafurilor, fiind apoi soluţionată printr-o tehnică cunoscută. Din cauza interdependenţelor între aceste operaţii, nu se garantează o soluţie optimă chiar dacă toate operaţiile sunt soluţionate în mod optim.

În cazul *metodei de rafinare iterativă*, problemele principale sunt legate de tipul modificărilor care trebuie efectuate asupra căii de date, selecţia unui tip de modificare în timpul unei iteraţii şi criteriul de terminare pentru procesul de rafinare. Cea mai simplă metodă poate fi o simplă schimbare a asignărilor. Algoritmul de interschimbare a perechilor execută o serie de modificări în calea de date, în scopul reducerii costului acesteia.

Deşi au existat încercări pentru dezvoltarea unei reprezentări intermediare comune pentru sinteza de nivel înalt, sunt necesare cercetări pentru elaborarea unor reprezentări adaptate pentru diferite limbaje de descriere şi diferite sisteme de proiectare, destinate unor arhitecturi specifice. Este necesară reprezentarea uniformă a funcţionării sincrone şi asincrone, ca şi reprezentarea consistentă a restricţiilor de proiectare pe parcursul tuturor fazelor sintezei.

Este necesară formularea teoretică a problemelor sintezei de nivel înalt, dezvoltarea unor forme canonice şi a unor algoritmi de optimizare pentru diferite arhitecturi şi aplicaţii. Deşi limbajul standard de descriere *VHDL* este foarte răspândit, acesta este în principiu un limbaj de simulare. Este necesară dezvoltarea altor limbaje bazate pe limbajul *VHDL* pentru a reprezenta conceptele utilizate de proiectanţii de sistem pentru aplicaţiile particulare. De asemenea, este necesară elaborarea unor reprezentări mai generale pentru a fi utilizate de diferiţi algoritmi de sinteză şi sisteme de proiectare.

În domeniul planificării, sunt necesare cercetări pentru utilizarea unor biblioteci de componente, arhitecturi şi funcţii de cost mai realiste. Fiind dată o bibliotecă de unităţi funcţionale, algoritmul de planificare trebuie să realizeze selecţia unităţilor funcţionale astfel încât să se optimizeze atât planificarea, cât şi costul. Astfel, trebuie să se elaboreze algoritmi de planificare care combină planificarea cu selecţia modulelor. De asemenea, algoritmii de planificare trebuie combinaţi cu alocarea, deoarece planificarea şi alocarea sunt interdependente.

Funcţiile de cost utilizate în cadrul planificării trebuie să fie realiste. Indicatorii simpli de calitate, ca numărul operatorilor sau al componentelor RT, nu sunt suficienţi. Funcţiile de cost trebuie să includă parametri de proiectare fizică, în particular informaţii despre întârzieri legate de plasare şi rutare.

Bibliografie

1. P. Michel, U. Lauther, P. Duzy: *The Synthesis Approach to Digital System Design*. Kluwer Academic Publishers, 1992.
2. Giovanni De Micheli: *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.
3. D. Patel, M. Schlag, M. Ercegovac: *An Environment for the Multi-level Specification, Analysis, and Synthesis of Hardware Algorithms*. Lecture Notes in Computer Science, Vol. 201: Functional Programming Languages and Computer Architecture, 1985, pp. 238-255.
4. D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin: *Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.
5. D. D. Gajski, F. Vahid, S. Narayan, J. Gong: *Specification and Design of Embedded Systems*. P T R Prentice Hall, Englewood Cliffs, 1994.
6. D. L. Perry: *VHDL*. McGraw-Hill, 1991.
7. S. Mazor, P. Langstraat: *A Guide to VHDL*. Second Edition. Kluwer Academic Publishers, 1993.
8. S. Carlson, E. Girczyc: *Understanding Synthesis Begins with Knowing the Terminology*. EDN, Sept. 3, 1992, pp. 125-131.
9. P. Eles, K. Kuchinski, Z. Peng, A. Doboli: *Specification of Timing Constraints in VHDL for High-Level Synthesis*. Proceedings of ConTI '94-International Conference on Technical Informatics, vol. 5., pp. 26-36.
10. J. C. Napier: *Multilevel ASIC Modeling*. EDN, April 29, 1993, pp. 75-80.
11. J. P. Banâtre, D. Lavenier, M. Vieillot: *From High Level Programming Model to FPGA Machines.* Rapport de recherche n^{o} 2240, INRIA, 1994.
12. A. J. Martin: *Tomorrow's Digital Hardware will be Asynchronous and Verified*. Technical Report, Department of Computer Science, California Institute of Technology, Pasadena CA, 1993.
13. A. J. Martin: *Asynchronous Datapaths and the Design of an Asynchronous Adder*. Technical Report, Department of Computer Science, California Institute of Technology, Pasadena CA, 1991.
14. M. J. Pertel: *A Critique of Adaptive Routing*. Technical Report, Department of Computer Science, California Institute of Technology, Pasadena CA, 1992.
15. H. Barringer, G. Gough, B. Monahan, A. Williams: *The ELLA Verification Environment*. A Tutorial Introduction. Technical Report, Department of Computer Science, University of Manchester, 1994.
16. Yachyang Sun: Algorithmic *Results on Physical Design Problems in VLSI and FPGA*. PhD Thesis, University of Illinois, Urbana, 1994.
17. T-T. Hwang, R. M. Owens, M. J. Irwin, K. H. Wang: *Logic Synthesis for Field Programmable Gate Arrays*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. Vol. 13, No. 10, Oct. 1994, pp.1280-1287.
18. R. Halverson, Jr., A. Lew: *An FPGA-Based Minimal Instruction Set Computer*. Technical Report, Information and Computer Sciences Department, University of Hawaii at Manoa, Honolulu, 1995.