UNIUNEA EUROPEANĂ

GUVERNUL ROMÂNIEI

Instrumente Structurale 2007-2013

Platformă de e-learning și curriculă e-content pentru învățământul superior tehnic

Testarea Sistemelor

# 17. Testarea funcţională fără modele ale defectelor

**Testarea funcțională**

Metodele de generare a testelor studiate în cursurile anterioare s-au bazat pe *modelul structural* al unui sistem aflat sub test iar obiectivul acestor metode era producerea testelor pentru defecte structurale de tipul blocajelor sau al scurt-circuitelor. Dar modelele structurale detaliate ale unor circuite complexe nu sunt în general disponibile deoarece nu sunt livrate de producătorii respectivi. Si chiar daca astfel de modele ar fi disponibile, metodele de generare a testelor bazate pe structuri nu ar fi folosibile, actualmente, din cauza complexității deosebite a circuitelor VLSI sau ULSI. In continuare vom examina *metode de testare externă funcțional având la bază modele funcționale* ale sistemelor respective.

**Aspecte fundamentale**

Un model funcțional reflectă specificațiile funcționale ale unui sistem şi într-o bună măsură este independent de implementarea sa. Din aceste motive testele funcționale deduse dintr-un model funcțional pot fi folosite nu numai în vederea verificării existenței unor defecte fizice, dar şi pentru verificarea unei implementări corecte a respectivului sistem. În acest sens să remarcăm că testele deduse dintr-un model structural care reflectă implementarea nu pot aserta când a fost corect implementată o operaţie şi când nu a fost corect implementată.
Obiectivul testării funcționale este *verificarea operării corecte a unui sistem în raport cu specificațiile sale funcționale.* Acest obiectiv poate fi abordat prin două metode distincte. O prima metodă presupune modele ale defectelor funcționale specifice şi încearcă să genereze teste ce detectează defecte definite prin modelele respective. Spre deosebire de această primă metodă, a doua metodă nu ţine seama de tipurile posibile de funcționare incorectă şi încearcă să deducă teste bazate numai pe funcționarea corectă a sistemului. Intre aceste două căi de abordare a problemei există şi o a treia ce defineşte implicit un model al defectului, şi care presupune că poate apare orice defect. Testele funcționale ce detectează aproape orice defect se spune că sunt exhaustive, deoarece acestea trebuie să verifice complet funcționarea corectă a sistemului. Din cauza lungimii testelor ce ar rezulta, testarea exhaustivă poate fi aplicată în practică numai la circuite de complexitate redusă. Prin utilizarea unei oarecari cunoaşteri asupra structurii sistemului şi printr-o uşoara restrângere a universului defectelor garantat detectate, se pot obţine *teste pseudo-exhaustive* ce pot fi substanţial mai scurte decât cele exhaustive. In continuare vom examina aceste trei abordări ale testării funcționale:
(1) testarea funcţională fără modele ale defectelor;
(2) testarea exhaustivă si pseudoexhaustivă;
(3) testarea funcțională folosind modele ale defectelor specifice.

## 2. Testarea funcțională fără modele ale defectelor
### 2.1. Metode euristice

Metodele de testare funcționala euristice sau ad-hoc, încearcă să exercite funcțiile sistemului. Spre exemplu, un test funcțional al unui bistabil ar putea cuprinde următoarele:

1. Validarea tranziției din 0 în 1 (setarea) şi validarea tranziției din 1 in 0 (resetarea).

2. Validarea capacităţii de păstrare a stării.
Exemplul următor ilustrează o procedură euristică de testare a unui microprocesor.

*Exemplul 1:* Funcţionarea unui microprocesor este definită, în general, prin schema bloc arhitecturală împreuna cu setul asociat de instrucţiuni. Figura 1 arată diagrama bloc a microprocesorului INTEL 8080. Se va presupune că un testor extern este conectat la magistralele de date, adrese şi control ale microprocesorului. Testorul furnizează microprocesorului instrucţiuni de executat şi verifică rezultatele.

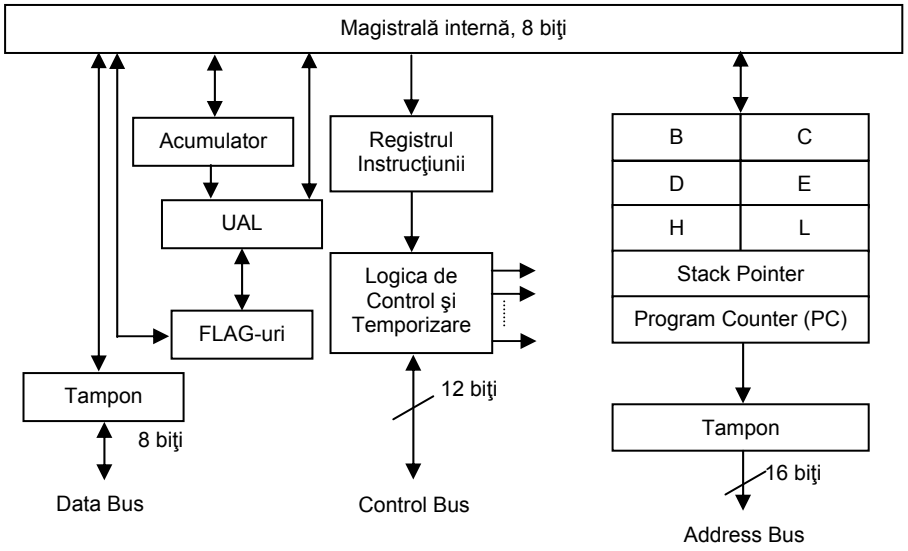


Figura1. Arhitectura microprocesorului INTEL 8080.

Un test funcţional tipic constă din următorii paşi:
1. test al contorului program (*CP*):
a. Testorul resetează 8080, aceasta aduce la zero şi *CP*.
b. Testorul plasează o instrucţiune NOP pe magistrala de date şi cere ca 8080 s-o execute repetat. Repetarea execuţiei instrucţiunii NOP are drept consecinţă incrementarea *CP* prin toate cele $2^{16}$ stări. Conţinutul registrului *CP* este accesibil pentru verificare pe magistrala de adrese.
2. test al registrelor *H* şi *L*:
a. Testorul scrie secvenţe de 8 biţi în *H* si *L* folosind instrucţiuni *MVI* (mută imediat).
b. Executând o instrucţiune *PCHL* transferă conţinutul registrelor *H* şi *L* în registrul *CP*. Aici se foloseşte faptul că registrul *CP* a fost deja testat în pasul 1. Această secvenţă de test este repetată pentru toate cele 256 de cuvinte posibile, de 8 biţi fiecare.
3. test al registrelor *B,C,D* şi *E*:
Intr-o manieră similară, testorul scrie un cuvânt de 8 biţi într-un registru *R* din {*B,C,D,E*}. Apoi *R* este transferat în *CP* via *H* sau *L* (*R* nu poate fi transferat direct în *CP*). Aici se beneficiază de testarea *CP* şi *HL* efectuată în paşii 1 şi 2.
Aceste teste sunt executate pentru toate cele 256 de cuvinte posibile de 8 biţi fiecare.

4. testarea pointer-ului stivei (*PS*):
Registrul *PS* este incrementat şi decrementat trecând prin toate stările posibile şi accesat via *CP*.
5. testarea registrului *Acumulator*:
Sunt scrise în *Acumulator*, şi apoi citite din acest registru, toate cuvintele posibile. Aceste operaţii pot fi făcute direct sau prin intermediul registrelor deja testate.
6. testarea unităţii aritmetico-logice (*UAL*) şi a *FLAG*-urilor:
Acest test încearcă toate instrucţiunile aritmetice şi logice. Operanzii sunt furnizaţi direct sau via registrele deja testate. Rezultatele sunt accesate similar.
*FLAG*-urile sunt verificate prin instrucțiunile de salt condiționat al căror efect (adresa următoarei instrucțiuni) este observat via registrul *control-program* (*CP*).
7. testarea tuturor celorlalte linii de control şi instrucţiuni anterior netestate.

O condiție importantă în testarea funcțională a unui micro-procesor este proprietatea de *ortogonalitate* a setului său de instrucțiuni.

Un set de instrucțiuni ortogonal permite ca fiecare operație care poate fi folosită în moduri de adresare diferite să fie executată în orice mod posibil de adresare. Această caracteristică implică faptul că mecanismele de decodificare ale codului operaţiei şi ale calculului adresei sunt independente.
Dacă setul de instrucțiuni nu este ortogonal, atunci fiecare operație trebuie testată pentru toate modurile sale de adresare. Această testare nu este necesară pentru seturile de instrucţiuni ortogonale astfel că pentru astfel de situaţii secvențele de test sunt semnificativ mai scurte.

#### Abordarea Începe-cu-Puţin (Start-Small)

Exemplul 1 ilustrează maniera de abordare *Începe-cu-Puţin* a testării funcționale pentru sisteme complexe în care rezultatele testării făcute până la un anumit moment dat sunt folosite în etapele ulterioare ale testării respectivului sistem. În acest mod partea testată a unui sistem este extinsă în fracţiuni bine determinate şi de complexitate abordabilă. Aplicabilitatea abordării *Începe-cu-Puţin* depinde de gradul de separare dintre componentele aferente diferitelor instrucţiuni. Obiectivul acestei abordări este simplificarea procesului de localizare a defectelor. În mod teoretic, dacă prima eroare apare în pasul *i* din secvența de test, aceasta înseamnă ca există un defect într-o componentă ce nu a mai fost testată în decursul actualei testări şi/sau un defect aferent unei operații testate la pasul *i*.

O tehnică de ordonare a instrucţiunilor unui microprocesor corespunzător metodei de abordare *Începe-cu-Puţin* ţine seama de *cardinalitatea* şi de *observabilitatea* unei instrucțiuni.

**Cardinalitatea** unei instrucțiuni se definește ca fiind numărul de registre accesate pe durata fazei de execuție a unei instrucțiuni (adică după faza de extragere şi decodificare). Astfel instrucțiunea NOP care are numai faza de extragere şi decodificare, are cardinalitatea 0.

**Observabilitatea** unei instrucţiuni arată măsura în care rezultatul operațiilor registru efectuate de instrucțiuni sunt direct observabile la nivelul liniilor primare de ieşire ale microprocesorului.
instrucțiunile sunt testate în ordinea crescătoare a cardinalităţii lor. În aceasta manieră instrucțiunile ce afectează mai puține registre sunt testate primele. Printre instrucțiunile cu aceeaşi cardinalitate se acordă prioritate acelora care au o observabilitate mai mare.

#### Problema acoperirii defectelor fizice

Problema majoră a testării funcționale euristice este necunoaşterea calităţii testelor funcționale. În adevăr, fără un model al defectului este extrem de dificil de dezvoltat o măsură riguroasă a calităţii testării.
În consecinţă o chestiune importantă este când anume un test funcțional dedus euristic, realizează o bună detecție a defectelor fizice. În cazurile în care este disponibilă o descriere structurală de nivel coborât privitor la implementarea sistemului, experienţa a arătat ca nivelul de acoperire al unui test funcțional euristic este posibil să se situeze în plaja dintre 50 si 70 de procente. Deci, în general, astfel de teste nu ating un nivel satisfăcător de acoperire al defectelor fizice, dar pot oferi o bună bază de pornire ce poate fi îmbunătăţită pentru a obține un test de-o calitate mai ridicată. De reținut că, totuși, modelele structurale de nivel coborât sunt de obicei ne-disponibile pentru sisteme complexe construite integrat.

Măsurile euristice pot fi folosite pentru a estima "completitudinea" unui test în raport cu diagrama de control a sistemului. Aceste măsuri sunt bazate pe monitorizarea activării operaţiilor într-un model de tip RTL (*Register Transfer Logic*). Spre exemplu, dacă modelul conține o instrucțiune de felul:

**if** *x* **then** *operaţiunea*$_1$ **else** *operaţiunea*$_2$

atunci, tehnica aceasta determină dacă testele aplicate fac condiția *x* atât adevărată cât şi falsă. Un test "*complet*" este necesar să exercite toate ieşirile posibile din blocurile de decizie. O măsură este raportul dintre numărul de ieşiri din blocurile de decizie, executate pe durata testului şi numărul total posibil de astfel de ieşiri. O a doua măsură, mult mai complicată, trasează căi de decizie, adică combinaţii de decizii consecutive. Spre exemplu, dacă instrucțiunea anterioară este urmată de instrucțiunea:

**if** *y* **then** *operaţiunea*$_3$ **else** *operaţiunea*$_4$

atunci sunt posibile patru căi de decizie de lungime 2, corespunzătoare celor patru combinaţii de valori ale variabilelor *x* şi *y*. A doua măsură este raportul dintre numărul deciziilor de lungime *k* exersate pe durata testului şi numărul total posibil de astfel de decizii.
Un aspect important al testării funcţionale, adesea neglijat de metodele euristice, este acela al verificării, suplimentare a operațiilor unui sistem, şi a faptului că nu au loc şi operații neprevăzute. Spre exemplu, suplimentar verificării unui transfer corect al datelor în registrul $R_1$, se verifică şi faptul că aceleași date nu apar şi în registrul $R_2$, apariţie ce ar constitui o eroare de funcționare. Astfel, verificând numai funcționarea corectă a operației respective - așa cum se întâmplă de regulă în cazul metodelor euristice - este evident departe de a fi suficient.

### 2.2 Testarea funcțională cu diagrame de decizii binare

Diagramele de decizii binare sunt instrumente de modelare funcțională şi pot fi folosite şi pentru scopuri de testare. Pentru început se va examina o descriere a calculului unei funcții *f,* descrise printr-o diagramă de decizie binară.
Se intră în diagramă prin ramura etichetată prin *f*. Intr-un nod intern *i* al diagramei se face branşarea pe ramura din stânga sau din dreapta după cum valoarea variabilei *i* este 0 sau 1. *Valoarea de ieşire* a căii urmate de-a lungul traversării diagramei este valoarea obţinută la sfârşitul căii. *Paritatea inversiunii* unei căi este numărul modulo-2 de puncte de inversiune întâlnit de-alungul căii. Pentru o cale de traversare cu valoare de ieşire *v* şi paritatea inversiunii *p*, valoarea funcției *f* este *v*+*p* (sumat modulo-2).

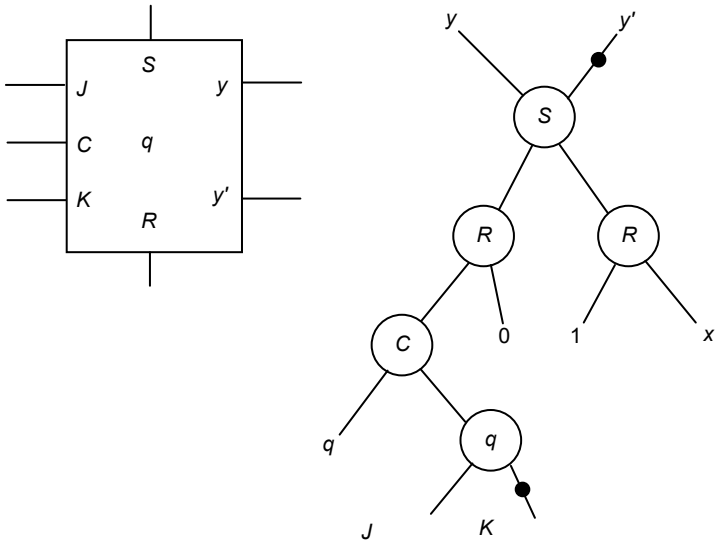


Figura 2. Diagrama de decizii binare pentru bistabilul *JK*.

*Exemplul 2:* Se consideră diagrama unui bistabil *JK* şi valoarea variabilei *y* de-alungul căii determinate de $S = 0$, $R = 0$, $C = 1$ şi $q = 1$:

$$K \oplus 1 = K'$$ (sumat modulo-2).

O traversare a unei diagrame de decizie binară implică o anumită poziţionare a variabilelor de-alungul căii. Astfel o traversare se spune că defineşte *un mod de operare* al sistemului (sau dispozitivului) respectiv. O cale a cărei valoare de ieşire este *x* denotă un mod de operare incorect, ilegal, în care valoarea ieşirii nu poate fi evaluată. Pentru bistabilul *JK* din figura 2 poziţionarea $S = 1$ şi $R = 1$ este ilegală.
Fiecare mod (legal) de operare poate fi văzut ca definind un *experiment* de test care partiționează variabilele unei funcții în trei mulțimi disjuncte:

- *variabile fixate*, ale căror valori binare determină calea asociată cu modul de operare;
- *variabile senzitivizate*, ale căror valori determină valoarea liniei de ieşire;
- *variabile nespecificate*, ale căror valori nu afectează valoarea liniei de ieşire (valorile acestora sunt notate prin *x*).

Un experiment furnizează *o specificaţie parţiala* a funcţiei, corespunzător unui mod de operare particular. Diagrama din figura 2 este o reprezentare a funcţiei $y = y(S,R,C,q,J,K)$. O astfel de specificaţie parţială a acestei funcţii poate fi:

$$y(0,0,0,q,x,x) = q.$$

Se poate arăta că mulţimea experimentelor deduse prin traversarea (parcurgerea) tuturor căilor corespunzătoare unei funcţii de ieşire, furnizează o *specificaţie completă* a funcţiei respective. Astfel, fiecare combinaţie posibilă de variabile este acoperită de un experiment şi numai de unul singur. În plus, experimentele sunt *disjuncte*, adică fiecare pereche de experimente diferă cel puţin prin valoarea unei variabile fixate.
Anumite diagrame de decizie binară pot conţine noduri adiacente, cum ar fi nodurile A şi B din figura 3, unde ambele ramuri din A re-converg în B. Se consideră două traversări care implică nodul A, una cu A = 0 şi cealaltă cu A = 1. Evident rezultatele acestor traversări sunt complementare. Atunci se pot combina cele două traversări într-una singură şi să se trateze A ca pe o variabilă senzitivă a experimentului rezultat. Fie *v* rezultatul traversării cu A = 0. Rezultatul traversării combinate este *v+A* (sumat modulo-2).

| *S* | *R* | *C* | *q* | *J* | *K* | *y* |
|---|---|---|---|---|---|---|
| 0 | 1 | *x* | *x* | *x* | *x* | 0 |
| 1 | 0 | *x* | *x* | *x* | *x* | 1 |
| 0 | 0 | 0 | *q* | *x* | *x* | *q* |
| 0 | 0 | 1 | 0 | *J* | *x* | *J* |
| 0 | 0 | 1 | 1 | *x* | *K* | *K'* |

Figura 3. Experimente de testare pentru bistabilul *JK*.

Un experiment dedus prin traversarea unei diagrame de decizie binară nu este prin acesta un test, ci numai o specificaţie funcţională parţială pentru un anumit mod de operare. Experimentele pot fi folosite în moduri diferite pentru generarea testelor. Deoarece valoarea ieşirii se va schimba cu orice schimbare a unei variabile senzitive a experimentului, un procedeu mult folosit este generarea tuturor combinaţiilor variabilelor senzitive pentru fiecare mod de operare. Dacă variabilele senzitive sunt intrări ale funcţiei, aceasta strategie tinde sa creeze căi *I-E* (*Intrare-iEşire*) de-alungul cărora multe dintre defectele interne sunt posibil sa fie detectate.
Anumite variabile ale unei funcţii s-ar putea să nu apară ca fiind senzitive într-un set de experimente. Acesta este cazul, spre exemplu, variabilelor *S* şi *R* în figura 2. În continuare se va examina un procedeu care generează un test în care o ieşire este făcută senzitiva faţă de o variabilă *s*. Principiul este combinarea a doua experimente $e_1$ si $e_2$, În care *s* este unica variabila fixata cu valori opuse, iar valorile ieşirii sunt (sau pot fi făcute) complementare. Cu alte cuvinte, avem nevoie de doua cai ce diverg într-un nod a cărui variabila este *s* şi care conduc la valori complementare ale ieşirilor.

Se reia diagrama din figura 2 şi se presupune ca se doreşte să se facă *y* senzitivă la valoarea variabilei *S*. Când *S* = 1 unicul experiment legal necesită *R* = 0 şi produce *y* = 1. Pentru a face *y* senzitiv la S se caută o cale cu *S* = 0 ce conduce la *y* = 0. (Se observă că nu se poate poziţiona *R* = 1, deoarece *R* a fost poziţionat la 0.) O astfel de cale necesită *C*

= 0 şi $q$ = 0. Aceasta arată că pentru a face ieşirea senzitiva la valoarea variabilei $S$, trebuie mai întâi să se aducă bistabilul în starea zero.

Principalul avantaj al diagramelor de decizie binara este furnizarea unui model funcţional complet şi succint al unui sistem (dispozitiv), model de la care pornind se poate deduce cu uşurinţă un set de experimente corespunzător fiecărui mod de operare al sistemului (dispozitivului).

## 3 Testarea exhaustivă şi pseudo-exhaustivă

### Modelul universal al defectului

Testele exhaustive detectează toate defectele definite de *modelul universal al defectului.* Acest model implicit al defectelor presupune ca este posibil orice defect (permanent), exceptând acele defecte ce măresc numărul de stări din sistem (circuit, dispozitiv). Pentru un circuit combinaţional $C$ ce realizează funcţia $Z(x)$, modelul universal al defectului face ipoteza ca orice defect $d$ schimba doar funcţia realizata de circuit, în funcţia $Z_d(x)$. Singurele defecte neincluse în acest model sunt acelea care transformă circuitul combinaţional $C$ într-un circuit secvenţial; scurtcircuitele ce introduc bucle de reacţie şi defectele de întrerupere din circuitele CMOS aparţin acestei categorii. Pentru un circuit secvenţial modelul universal al defectului presupune că orice defect ce schimbă tabelul de stări nu introduce şi stări noi. Se poate observa că modelul universal al defectului introduce o mulţime totală a defectelor (un univers de defecte) care, practic, nu este enumerabilă.

### 3.1 Circuitele combinaţionale

Pentru a testa toate defectele definite de modelul universal al defectului dintr-un circuit combinaţional $C$ cu $n$ LPI, este necesar sa aplicam toate cele $2^n$ posibile combinaţii de intrare (vectori de intrare). Creşterea exponenţială a numărului necesar de vectori de test limitează aplicabilitatea practica a acestei metode de *testare exhaustiva* numai la circuite cu cel mult 20 de LPI. In continuare vom examina *metode de testare pseudo-exhaustivă* ce pot testa aproape toate defectele definite de modelul universal al defectului cu un număr de vectori de test, sensibil mai mic decât $2^n$.

#### 3.1.1 Circuite parţial-dependente

Fie $O_1, O_2, \ldots, O_m$ linii primare de ieşire ale unui circuit cu $n$ LPI, Si fie $n_i$ numărul de LPE ce alimentează $O_i$. Un circuit În care nici o LPE nu depinde de toate LPI (adică $n_i < n$), se numeşte *circuit parţial-dependent*. Pentru astfel de circuite testarea pseudo-exhaustivă poate fi realizata prin aplicarea tuturor celor $2^{n_i}$ combinaţii la cele $n_i$ intrări alimentând fiecare LPE $O_i$.

#### 3.1.2 Tehnici de partiţionare

Tehnicile de testare pseudoexhaustivă descrise anterior nu sunt aplicabile *circuitelor total-dependente*, în care cel puţin o LPE depinde de toate LPI. Chiar pentru circuite parţial-dependente, mărimea unui test pseudo-exhaustiv poate încă să fie prea mare pentru a fi acceptabil În practica. în astfel de cazuri, testarea pseudo-exhaustivă poate fi realizata prin *tehnici de partiţionare.*

Principiul este partiționarea unui circuit în *segmente* astfel încât numărul de intrări al fiecărui segment sa fie semnificativ mai mic decât numărul de LPI din circuit. Apoi segmentele sunt exhaustiv testate. Problema principala a acestei tehnici este aceea că, în genere, intrările unui segment nu sunt LPI iar ieşirile unui segment nu sunt LPE. Astfel este necesar un mijloc de control al intrărilor segmentului dinspre LPI şi un mijloc de observare al ieşirilor segmentului către LPE. O metoda de realizare a acestui deziderat, numita *partiționarea senzitivizată*, se bazează pe senzitivizarea unor căi dinspre LPI ale circuitului către intrările segmentului şi de la ieşirile segmentului către LPE ale circuitului.

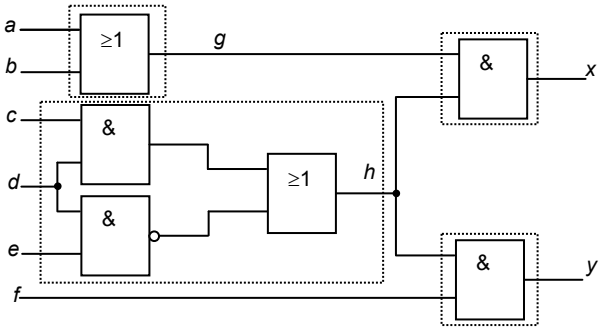


Figura 4a. Circuitul exemplului 3.

*Exemplul 3:* Sa consideram circuitul din figura 4a. Se partiţionează acest circuit în patru segmente. Primul segment consta din sub-circuitul a cărui ieşire este *h*. Celelalte trei segmente constau, respectiv, din porțile *g, x* si *y*.

| | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *x* | *y* |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 0 | 0 | 0 | 1 | | 1 | | 1 |
| 2 | | | 0 | 0 | 1 | 1 | | 1 | | 1 |
| 3 | | | 0 | 1 | 0 | 1 | | 1 | | 1 |
| 4 | | | 0 | 1 | 1 | 1 | | 0 | | 0 |
| 5 | | | 1 | 0 | 0 | 1 | | 1 | | 1 |
| 6 | | | 1 | 0 | 1 | 1 | | 1 | | 1 |
| 7 | | | 1 | 1 | 0 | 1 | | 1 | | 1 |
| 8 | | | 1 | 1 | 1 | 1 | | 1 | | 1 |

Figura 4b.

Figura 4b arată cei opt vectori necesari pentru testarea exhaustiva a segmentului *h* şi pentru observarea liniei *h* la LPE *y*. Deoarece *h* = 1 este condiția de observare a liniei *g* la LPE *x*, se poate profita de 5 vectori din 8 în care *h* = 1 pentru a testa exhaustiv segmentul *g* (a se vedea figura 4c).

| | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *x* | *y* |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 0 | 0 | 0 | 1 | | 1 | | 1 |
| 2 | | | 0 | 0 | 1 | 1 | | 1 | | 1 |
| 3 | | | 0 | 1 | 0 | 1 | | 1 | | 1 |
| 4 | | | 0 | 1 | 1 | 1 | | 0 | | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | | | 0 | 1 | 1 | 0 | | 0 | | 0 |
| 10 | | | 0 | 0 | 0 | 0 | | 0 | | 0 |

Figura 4c.

Se adaugă deasemenea vectorii 9 şi 10 pentru a completa testul exhaustiv al segmentului *y*. Analizând testele aplicate până acum segmentului *x*, se poate observa că lipsesc combinațiile în care *h* = 0; acestea pot fi aplicate folosind vectorii 4 şi 9. Figura 4d arată setul de test rezultat numărând 10 vectori, comparativ cu $2^6 = 64$ vectori necesari pentru testarea exhaustiva sau cu $2^5 = 32$ vectori cât ar fi necesitat testarea pseudoexhaustivă fără partiționare.

| | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* | *x* | *y* |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | | | 0 | 0 | 0 | 1 | | 1 | | 1 |
| 2 | | | 0 | 0 | 1 | 1 | | 1 | | 1 |
| 3 | | | 0 | 1 | 0 | 1 | | 1 | | 1 |
| 4 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |
| 7 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 |
| 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 9 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 10 | | | 0 | 0 | 0 | 0 | | 0 | | 0 |

Figura 4d.

Un set de test pseudo-exhaustiv bazat pe partiționarea senzitivizată detectează orice defect care schimbă tabelele de adevăr ale unui segment. Deoarece un circuit poate avea mai multe partiționări posibile, acest model al defectelor depinde, într-o oarecare măsură de alegerea setului de segmente. Partiționarea unui circuit astfel încât mărimea testului pseudoexhaustiv asociat să fie minimală este o problema *NP*-completă. De remarcat, în final, că tehnicile de partiționare presupun o cunoaştere a modelului structurii interne a circuitului.

### 3.2 Circuitele secvențiale

Pentru un circuit secvențial, modelul universal al defectului presupune ca orice defect modifica tabele de stări ale circuitului fără să crească numărul de stări ale circuitului. O secvența de intrare care detectează orice defect definit prin acest model, distinge o maşină dată cu *n* stări de o altă maşină cu aceleaşi intrări şi ieşiri şi cu cel mult *n* stări. Existența unei astfel de *secvențe de verificare* este dată de următoarea teoremă:

*Teorema 1.* Pentru orice maşină secvențială tare conexă *M* cu *n* stări, există o secvență pereche de intrări-ieşiri care pot fi generate de maşina *M*, dar nu pot fi generate de orice altă maşină *M'* cu *n* sau mai puține stări.

Deoarece generarea unei secvențe de verificare se bazează pe tabelul de stări al unui circuit, aceasta abordare a testării exhaustive este aplicabilă numai circuitelor de mică complexitate. Pe scurt, o secvență de verificare pentru o maşina secvențială constă din următoarele trei faze:

1. inițializarea, adică aducerea maşinii *M* într-o stare cunoscută de start;
2. verificarea proprietății că maşina *M* are *n* stări;
3. verificarea fiecărei intrări din tabelul de stări;

Deducerea unei secvențe de verificare este mult facilitată dacă *M* are *o secvență de distingere*. Fie $Z_i$ secvența de ieşire generată de *M*, pornind din starea $q_i$, ca răspuns la o

secvența de intrare $X_D$. Daca $Z_i$ este unica pentru fiecare $i$=1, 2, ... , $n$, atunci $X_D$ este o secvență de distingere. (O maşină ce nu are o secvență de distingere se poate modifica, prin adăugarea unei ieşiri, astfel încât sa aibă o astfel de secvență.) Importanța secvenței $X_D$ constă în aceea că prin observarea răspunsului mașinii $M$ la $X_D$ se poate determina starea în care se afla $M$ în momentul aplicării secvenței $X_D$.
Verificarea proprietății ca maşina $M$ are $n$ stări distincte necesita o secvența de intrări-ieşiri ce conține $n$ sub-secvențe de forma $X_D$ - $Z_i$ pentru $i$ = 1, 2, ... , $n$. Se poate folosi acum $X_D$ pentru a verifica orice intrare din tabela de stări. O tranziție de forma $N(q_i,x) = q_j$, $Z(q_i,x) = z$, este verificata prin doua sub-secvențe intrări-ieşiri de forma:

$$X_DX'X_D\text{-}Z_pZ'Z_i \quad \text{şi} \quad X_DX'xX_D\text{-}Z_pZ'zZ_j.$$

Prima secvență arată că $X_DX'$ conduce $M$ din starea $q_p$ în starea $q_i$. Bazându-ne pe aceasta, putem conchide că atunci când se aplică intrarea $x$ în a doua sub-secvență, maşina $M$ se găsește în starea $q_i$. În final $X_D$ verifica faptul că $x$ aduce maşina $M$ în starea $q_j$.
Secvențele care verifică că maşina $M$ are $n$ stări distincte şi verifica fiecare intrare în tabela de stări pot fi adesea suprapuse pentru reducerea lungimii secvenței de verificare.

#### 4 Testarea funcțională cu modele specifice ale defectelor

##### 4.1 Modele ale defectelor funcționale

Defectele funcționale tind să reprezinte maniera de manifestare a defectelor fizice asupra operării unui sistem modelat funcțional. Un set de defecte funcționale trebuie să fie *realist*, în sensul ca o comportare defectuoasa indusa de acestea trebuie, în general, sa se potrivească comportamentului defectuos indus de defectele fizice. Un model de defect funcțional se poate considera *bun* dacă testele generate să detecteze defectele care le definește oferă o mare acoperire pentru defectele unice de tip blocaj din modelul structural detaliat al sistemului. (Pentru ca nu se cunoaște gradul de comprehensiune al unui model funcțional al defectelor, nu se poate folosi acoperirea defectelor funcționale a unui test ca o măsură semnificativa a calității testului.)
Un model funcțional al defectelor poate fi explicit sau implicit. Un *model explicit* identifica fiecare defect individual, şi fiecare defect poate deveni o ţintă a generării testului. Pentru a fi util, un model explicit al defectului funcțional trebuie sa *definească un univers al defectelor, rezonabil ca mărime*, astfel încât procesul de generare al testului sa fie computațional fezabil. Aceasta spre deosebire de un *model implicit* care identifica clase de defecte cu proprietăți "similare", astfel încât toate defectele din aceeași clasă pot fi detectate prin proceduri similare. Avantajul unui model implicit al defectului este acela ca nu reclama o enumerare explicita a defectelor dintr-o clasă.

###### Defectele de adresare

Multe operații dintr-un sistem numeric se bazează pe decodificarea adresei unei entități precise. Exemple tipice includ de regula operații de felul acesta:

- adresarea unui cuvânt dintr-o memorie;
- selectarea unui registru în funcție de un câmp al unui cuvânt instrucțiune al unui procesor;
- decodificarea unui cod de operație pentru a determina instrucțiunea ce trebuie executată.

Caracteristica comuna a acestor scheme este folosirea unei adrese de *n* biți pentru selectarea uneia dintre cele $2^n$ entități distincte. *Defectele funcționale ale adresării* reprezintă urmările unor defecte fizice din implementarea hardware a mecanismului de selecție în operarea sistemului. Ori de câte ori entitatea *i* trebuie sa fie selectata, prezența unui defect de adresare poate conduce la:

- selectarea niciunei entități;
- selectarea entității *j* în locul entității *i*;
- selectarea entității *j* în plus de entitatea *i*.

Mai general, un set de entități $\{j_1, j_2, \ldots, j_k\}$ pot fi selectate în locul sau în plus de entitatea *i*.
O trăsătură importantă a acestui model de defect este că obligă procesul de generare a testului să verifice că sunt realizate funcțiile dorite şi că deasemenea nu au loc nici un fel de alte operații necerute. Acest aspect fundamental al testării funcționale este adesea pierdut din vedere de metodele euristice.
Defectele de adresare folosite în dezvoltarea modelelor defectelor implicite pentru microprocesoare sunt examinate în cele ce urmează.

### 4.2 Modele ale defectelor pentru microprocesoare

Într-o primă etapă se vor introduce modelele defectelor funcționale pentru microprocesoare. Procedeele de generare ale testelor folosind aceste modele ale defectelor vor fi prezentate într-o etapă ulterioară.

#### Modele graf pentru microprocesoare

În vederea generării unui test funcțional, un microprocesor poate fi modelat printr-un graf bazat pe arhitectura sa şi pe setul de instrucțiuni respectiv. Fiecare registru accesibil de către utilizator este reprezentat de un nod în graf. Două noduri suplimentare, etichetate IN şi OUT, reprezintă conexiunile dintre microprocesor şi mediul exterior; în mod normal acestea sunt magistralele de date, adrese şi control care conectează microprocesorul la memorie şi la dispozitivele de I/E. Atunci când microprocesorul este testat, testorul controlează nodul IN şi observă nodul OUT. Un arc în graf dintre nodul A Si nodul B arata ca exista o instrucțiune a cărei execuție implica un transfer al informației de la nodul A la nodul B.

*Exemplul 4:* Sa consideram un microprocesor ipotetic cu urmatoarele registre:

- A acumulator;
- PC contorul program;
- SP pointer-ul stivei, conținând adresa vârfului stivei;
- R1 registrul de uz general;
- R2 registrul de lucru;
- SR - registrul subrutinelor, conținând adresa de întoarcere (se presupune că nu este folosită imbricarea apelurilor de subrutine);
- IX - registrul index.

Figura 5 arată modelul graf al acestui microprocesor. Tabelul din figura 6 ilustrează corespondența dintre anumite instrucțiuni ale microprocesorului şi arcele din graf. (notația (R) semnifica conținutul locației de memorie adresată de registrul R.) De

remarcat faptul ca un arc în graf poate corespunde mai multor instrucțiuni şi ca o instrucțiune poate crea mai multe arce în graf.

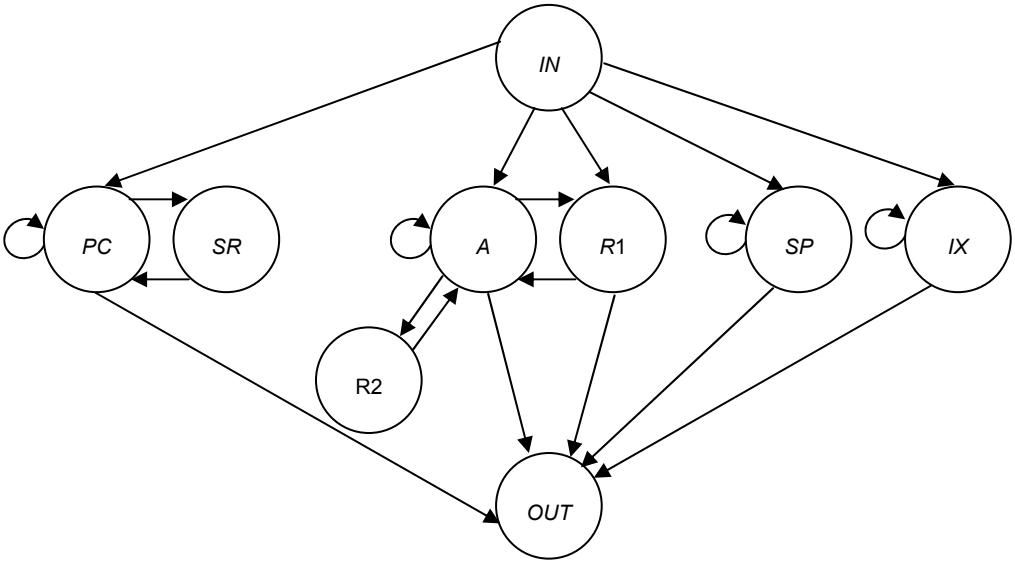


Figura 5. Modelul graf al unui microprocesor.

| Instrucțiune | Operație | Arc(e) |
|---|---|---|
| MVI R, a<br>R ∈ {A, R1, SP, IX} | R ← a | IN → R |
| MOV Ra, Rb<br>R ∈ {A, R1, SP, IX} | Ra ← Rb | Ra → Rb |
| ADD A,R1 | A ← A+R1 | A → A<br>R1 → A |
| JMP a | PC ← a | IN → PC<br>PC→ OUT |
| ADD A,(IX) | A ← A+(IX) | IX → AUT,<br>IN → A,<br>A → A |
| CALL a | SR ← PC,<br>PC ← a | PC → SR<br>IN → PC<br>PC → OUT |
| RET | PC ← SR | SR → PC<br>PC → OUT |
| PUSH R,<br>R ∈ {A, R1} | SP ← R<br>SP ← SP+1 | SP → AUT<br>R → AUT<br>SP → SP |
| POP R<br>R ∈ {A, R1} | SP ← SP-1<br>R ← (SP) | SP → SP<br>R → OUT<br>IN → R |
| INCR R<br>R ∈ {A, SP, IX} | R ← R+1 | R → R |
| MOV (IX),R<br>R ∈ {A, R1} | (IX) ← R | IX → AUT<br>R → OUT |

Figura 6. Setul de instrucțiuni al microprocesorului.

**Clase de defecte**

Defectele care afectează operarea corectă a unui microprocesor pot fi împărțite în următoarele clase:

1. defecte de adresare ce afectează funcția decodificare - registru;

2. defecte de adresare ce afectează funcția decodificare - instrucțiune şi secvențiere - instrucțiune;
3. defecte ale funcției de memorare - date (stocare - date);
4. defecte ale funcției manipulare - date;
5. defecte ale funcției transfer - date.

Modelul global al defectelor pentru un microprocesor permite un număr arbitrar de defecte dar dintr-o singura clasa de defecte, clasa aflata printre cele enumerate anterior.

#### 4.2.1 Modelul defectului pentru funcția decodificare-registru

Se notează decodificarea (selecția) unui registru R prin funcția $f_D(R)$ a cărei valoare libera de defecte este R pentru orice registru al microprocesorului. Ori de câte ori o instrucțiune accesează un registru R, defectul de adresare ce afectează funcția decodificare-registru conduce la una dintre următoarele urmări:

1. Nu se accesează nici un registru.
2. Se accesează un set de registre (ce poate include sau nu şi registrul *R*).

Primul caz este reprezentat prin $f_D(R) = i$, unde *i* reprezintă un registru inexistent. În acest caz o instrucțiune care încearcă să scrie în registrul *R* nu va modifica deloc conținutul acestuia iar o instrucțiune ce va încerca sa citească conținutul registrului *R* va recupera un vector UNU sau ZERO (depinzând de tehnologie), independent de conținutul registrului R; un vector UNU (ZERO) este un vector cu toți biții 1 (0).
În al doilea caz $f_D(R)$ reprezintă setul de registre accesat eronat. În această situație o instrucțiune ce va încerca să scrie data *d* în registrul R va scrie *d* în toate registrele din $f_D(R)$, iar o instrucțiune ce va încerca sa citească conținutul registrului *R* va recupera un SAU ori un ŞI la nivel de bit (depinde de tehnologie) al conținuturilor registrelor selecționate eronat.

*Exemplul 5:* Pentru microprocesorul din exemplul 4 se consideră defectul $f_D(R) = i$. Instrucțiunea CALL *a* va face un salt corect la *a*, dar nu va salva adresa de Întoarcere În SR. Acest defect va fi detectat prin executarea instrucțiunii RET, deoarece În acel moment va fi încărcat în PC un vector ZERO sau UNU.
Defectul $f_D(R2)=R1$ va cauza instrucțiunii MOV *R*2,*R*1 să se comporte ca o instrucțiune NOP, iar instrucțiunea MOV *A,R*1 va avea drept rezultat transferarea conținutului registrului *R*1 în registrul *A*. În prezenta defectului $f_D(R2) = \{R1, R2\}$, instrucțiunea MOV *R*2, *R*1 se executa corect, dar MOV *A*, *R*2 va transfera *R*1"R2 În A, unde " este notația pentru operația SAU ori SI la nivel de bit. ~

De remarcat ca un defect ce cauzează $f_D(R_i) = R_j$ şi $f_d(R_j) = R_i$) nu afectează operarea corectă (deoarece re-etichetează numai cele doua registre); astfel de defect nu este detectabil.

#### 4.2.2 Modelul defectului pentru funcția decodificare-instrucțiune şi pentru funcția secvenţiere - instrucțiune

**Modelul microprogramat pentru execuţia instrucțiunii**

O instrucțiune poate fi văzuta ca o *secvență de microinstrucțiuni,* în care *fiecare microinstrucțiune consta dintr-un set de micro-ordine* care sunt executate în paralel. Micro-ordinele reprezintă operații elementare de transfer-date şi manipulări-date; adică acestea constituie elementele de construcție de bază ale setului de instrucțiuni. Acest

*model microprogramat* este un model abstract, aplicabil indiferent cum este realizat microprocesorul.
Spre exemplu, instrucțiunea ADD *A*,*R*1 poate fi privită ca o secvență a următoarelor microinstrucțiuni: (1) două micro-ordine (paralele) pentru a aduce conținutul lui *A* şi *R*1 la intrările UAL, (2) un micro-ordin ADD şi (3) un micro-ordin de încărcare a rezultatului din UAL în A. De subliniat ca procesul de generare a testului (va fi descris în cele ce urmează) nu necesita cunoașterea structurii instrucțiunilor în termeni de microinstrucțiuni si micro-ordine.

#### Modelul defectului

Modelul microprogramat pentru execuția instrucțiunii ne permite sa definim un model comprehensiv al defectului pentru funcția de decodificare-instrucțiune şi secvențiere - instrucțiune. Anume, defectele de adresare ce afectează execuția unei instrucțiuni *I* poate cauza una sau mai multe urmări ale defectului:

1. Unul sau mai multe micro-ordine nu sunt activate de micro-instrucțiunile instrucțiunii *I*.
2. Micro-ordinele sunt activate eronat de micro-instrucțiunile instrucțiunii *I*.
3. Un set diferit de microinstrucțiuni este activat în loc sau în plus de micro-instrucțiunile instrucțiunii *I*.

Acest model al defectului este general, deoarece permite execuția parțială a instrucțiunilor şi chiar execuția unor "noi" instrucțiuni, care nu sunt prezente în setul de instrucțiuni al microprocesorului.
Un defect care afectează o instrucțiune *I* este *simplu* daca cel mult un micro-ordin este activat eronat pe durata execuției instrucțiunii *I* (un număr oarecare de micro-ordine pot fi inactive). Astfel exista o corespondență *unu – la - unu* între setul de micro-ordine şi setul de defecte simple.
Două micro-ordine se numesc *independente* dacă niciunul dintre acestea nu modifică registrele sursă folosite de celălalt.
Modelul actual al defectului permite orice număr de defecte simple, în ipoteza că acestea sunt perechi-perechi independente.
Un exemplu de defect neinclus În acest model al defectului este defectul care activează eronat secvența de micro-ordine (ADD A,R1; MOV R2,A); astfel de defect se spune ca este *legat*.

### 4.2.3 Modelul defectului pentru funcția memorare-date

Acest model al defectului pentru funcția de memorare-date este o extensie directa a modelarii prin blocaje. Se permite oricărui registru din microprocesor sa aibă oricât de mulți biți blocați la 0/1.

### 4.2.4 Modelul defectului pentru funcția transfer-date

Funcția transfer-date implementează toate transferurile de date între nodurile modelului graf ale unui microprocesor (" transfer" având semnificația că datele sunt mutate dint-un loc într-altul fără sa fie modificate). Se reamintește că un arc de la nodul A la nodul B poate corespunde la mai multe instrucțiuni care au drept urmare un transfer de date de la A la B. Chiar dacă aceeași secțiune hardware poate implementă (unele dintre) aceste transferuri, în scopul construirii unui model independent de implementare, se presupune că fiecare instrucțiune ce are drept efect un transfer A→B definește o *cale de transfer*

*logica* separata de la A la B, şi fiecare astfel de cale de transfer poate fi independent defectă. Pentru microprocesorul din Exemplul 4, acest model implică faptul ca transferul IN $\rightarrow$ PC cauzat de o instrucţiune JMP *a*, poate fi defect, în timp ce acelaşi transfer cauzat de CALL *a,* poate fi lipsit de orice eroare. Modelul defectului pentru funcţia transfer - date presupune ca orice linie dintr-o cale de transfer poate fi *b*-*l*-0 sau *b*-*l*-1, iar oricare doua linii dintr-o cale de transfer pot fi scurtcircuitate.

#### 4.2.5 Modelul defectului pentru funcţia manipulare-date

Funcţia manipulare-date implica instrucţiuni ce modifica datele, cum ar fi operaţiile logice şi aritmetice, incrementarea sau decrementarea registrelor etc. Este practic imposibil sa se stabilească un model semnificativ al defectului funcţional pentru funcţia manipulare-date fără o cunoaştere a structurii UAL sau a altor unităţi funcţionale implicate (modalitatea de deplasare, de incrementare etc.). Abordarea uzuala în generarea testelor funcţionale presupune ca testele pentru funcţia manipulare-date sunt dezvoltate prin alte tehnici care să prevadă şi mijloace de aplicare a acestor teste precum şi modalităţi de observare a rezultatelor acestora.

Astfel, dacă se verifica întâi funcţia transfer-date, acest set de teste poate fi aplicat prin încărcarea operanzilor necesari în registre, executarea operaţiilor logico-aritmetice corespunzătoare şi examinând apoi rezultatele. Căile de date ce alimentează cu operanzi UAL şi care transfera apoi rezultatele din UAL sunt verificate o data cu UAL.

### 4.3 Procedee de generare a testelor

În continuare vor fi examinate procedeele de generare a testelor pentru modelele defectelor funcţionale introduse până în acest punct. Modelele defectelor sunt implicite iar procedeele de generare a testelor au rezoluţie la nivelul claselor de defecte fără să identifice individual vreun membru al clasei respective.

#### 4.3.1 Testarea funcţiei decodificare-registru

Deoarece defectele ce cauzează $f_D(R_i) = R_j$ şi $f_D(R_j) = R_i$ sunt nedetectabile, scopul în testarea funcţiei decodificare - registru este verificarea că pentru fiecare registru $R_i$ a microprocesorului mărimea mulţimii $f_D(R_i)$ este 1. Aceasta garantează că niciunul dintre defectele de adresare detectabile ce afectează funcţia de decodificare - registru nu este prezent.
Procedura de testare implica scrierea si citirea registrelor. Pentru fiecare registru $R_i$ se predetermină o secvenţă de instrucţiuni WRITE($R_i$) care transferă datele din nodul IN în $R_i$ şi o secvenţă READ($R_i$) care transfera conţinutul registrului $R_i$ către nodul OUT. Ori de câte ori există mai multe posibilităţi de scriere sau de citire a unui registru, se va alege secvenţa cea mai scurtă.

*Exemplul 6:* Pentru microprocesorul din exemplul 4 avem următoarele secvenţe :

WRITE(A) = (MVI A, *a*)
READ(A) = (MOV (IX), A)

WRITE(R2) = (MVI A, *a*; MOV R2, A)
READ(R2) = (MOV A, R2; MOV (IX), A)

WRITE(SR) = (JMP *a*; CALL *b*)

READ(SR) = (RET) ~

Fiecărui registru $R_i$ îi asociem *o etichetă* $l(R_i)$, care este lungimea secvenței READ($R_i$); În acest caz $l(R_i)$ reprezintă cea mai scurta "distanță" de la $R_i$ la nodul OUT. Folosind secvențele READ din Exemplul 6, $l$(A)= $l$(SR)=1 şi $l$(R2)=2.
Strategia procedeului de testare este să construiască treptat un set A de registre astfel încât:

1. $f_D(R_i) \neq \varnothing$ pentru fiecare $R_i \in A$
2. $f_D(R_i) \cap f_D(R_j) = \varnothing$ pentru fiecare $R_i, R_j \in A$.

Mulţimea $A$ va conține, eventual, toate registrele microprocesorului, şi atunci aceste condiții vor implica $|f_D(R_i)| = 1$ pentru fiecare $R_i$. În continuare este prezentat procedeul *Decodifică_registre* care realizează detectarea defectelor de adresare ce afectează funcția decodificare-registre. Procedeul este executat de un testor exterior care furnizează instrucțiunile pentru secvențele READ şi WRITE şi verifică datele recuperate prin secvențele READ. În virtutea principiului *Începe-cu-puţin* (start-small), registrele sunt rând pe rând adăugate mulțimii $A$, în ordinea crescătoare a etichetelor acestor registre. Aceeaşi ordine este folosită pentru citirea şi extragerea conţinutului registrelor din $A$.

```
Decodifica_registre()
begin
        A = 0
        adaugă un registru cu eticheta 1 la A
        for every registru Ri ce nu este in A
                begin
                        for Data = ZERO,UNU
                                begin
                                        for every registru Rj∈A WRITE(Rj) Data
                                                WRITE(Ri) NON-Data
                                                for every registru Rj∈A READ(Rj)
                                                READ(Ri)
                                end
                        adauga Ri la A
                end
end
```

Figura 7. Testarea funcției de decodificarea registrelor.

**Teorema 2:** Dacă procedura *Decodifica_registre()* se executa fără să se detecteze nicio eroare (în datele recuperate prin secvențele READ), atunci funcția decodificare-registre este liberă de defecte de adresare (detectabile).
**Demonstrație:** Se va demonstra că execuția completă şi fără erori a procedurii *Decodifica_registre*() arată că $|f_D(R_i)| = 1$ pentru fiecare registru $R_i$. Demonstrația se va face prin inducție. Fie $R_i$ registrul cu cea mai mică etichetă dintre registrele curent neaparţinând de A. Se presupune că toate registrele din $A$ au mulțimi ne-vide şi disjuncte $f_D(R_i)$. Se va arăta că lucrurile rămân în aceeași stare şi după adjuncția registrului $R_i$ la mulțimea $A$.

Dacă $f(R_i) = \emptyset$, atunci READ($R_i$) returnează fie ZERO fie UNU. Deoarece READ($R_i$) se execută de două ori, o dată aşteptând ZERO şi o dată aşteptând UNU, una din situaţii va detecta eroarea. Dar deoarece ambele secvenţe se executa fără detectare de erori, atunci $f(R_i) \neq \emptyset$.

Daca pentru un registru oarecare $R_j \in A$, $f_D(R_j) \cap f_D(R_i) \neq \emptyset$, atunci WRITE($R_j$) scrie *Data* într-un registru al cărui conţinut este apoi schimbat cu *NON-Data* prin WRITE($R_i$). Aceasta eroare va fi detectata printr-una din cele doua secvenţe READ($R_j$). Deci, dacă ambele secvenţe READ($R_j$) se execută fără detectarea vreunei erori, atunci $f_D(R_j) \cap f_D(R_i) = \emptyset$. De remarcat că, deoarece $l(R_i) \geq l(R_j)$ pentru fiecare $R_j \in A$, citirea registrului $R_j$ nu necesită dirijarea conţinutului registrului $R_j$ prin registrul $R_i$.
Cazul iniţial al inducţiei, situaţia iniţială când $A$ conţine numai un singur registru, poate fi verificat prin argumente similare. Eventual, în final, $A$ va conţine toate registrele microprocesorului. Atunci relaţiile $f_D(R_i) \neq \emptyset$ şi $f_D(R_i) \cap f_D(R_j) \neq \emptyset$ implică $|f_D(R_i)| = 1$ pentru fiecare $R_i$. ◊

Fie $n_R$ numărul de registre al microprocesorului. Numărul de secvenţe WRITE şi READ generate de procedura *Decodifica_registru()* este proporţional cu $n_R^2$. Astfel, dacă toate registrele pot fi direct scrise şi citite, atunci numărul de instrucţiuni în secvenţa generată de test este deasemenea proporţional cu $n_R^2$. Pentru arhitecturi cu registre "*în profunzime incluse*", în cazul cel mai defavorabil secvenţa de test se apropie de $n_R^3$.

#### 4.3.2 Testarea funcţiilor decodificare-instrucţiune şi secvenţiere-instrucţiune

Scopul este detectarea tuturor defectelor simple ce afectează execuţia oricărei instrucţiuni. Pentru aceasta trebuie sa ne asiguram ca orice defect simplu ce afectează o instrucţiune $I$ cauzează erori fie În datele transferate la nodul OUT sau Într-un registru ce poate fi citit după ce s-a executat instrucţiunea $I$. Acest fapt trebuie sa fie valabil dacă micro-ordinele instrucţiunii $I$ nu sunt activate şi/sau dacă micro-ordine (independente) suplimentare nu sunt eronat activate. Lipsa unor micro-ordine în secvenţa activată este uşor de detectat, aşa încât orice instrucţiune care nu activează toate micro-ordinele ce-o compun în mod normal poate fi uşor făcută observabilă prin producerea unor rezultate incorecte. Pentru detectarea execuţiei unor micro-ordine adiţionale, se asociază diferite secvenţe de date, numite *cuvinte-cod*, registrelor microprocesorului. Fie $cw_i$ notaţia pentru cuvântul-cod asociat cu registrul $R_i$. Mulţimea cuvintelor-cod trebuie să satisfacă proprietatea ca *orice micro-ordin singular ce operează asupra cuvintelor-cod trebuie fie sa producă un non-cuvânt-cod, fie să încarce registrul $R_i$ cu un cuvânt-cod $cw_j$ al unui alt registru.*
Pentru $n_g$ registre având fiecare $n$ biţi, un set de cuvinte-cod care satisfac proprietatea enunţată poate fi obţinut folosind un cod de tipul *p-din-n*, unde fiecare cuvânt de cod are exact $p$ biţi poziţionaţi pe 1 ( daca $C_n^p \geq n$).

<u>*Exemplul 7:*</u> Se consideră opt registre $R_1$, $R_2$, ... , $R_8$, încărcate cu următoarele cuvinte-cod:

| | |
|---|---|
| $cw_1$ | 01101110 |
| $cw_2$ | 10011110 |
| $cw_3$ | 01101101 |
| $cw_4$ | 10011101 |
| $cw_5$ | 01101011 |
| $cw_6$ | 10011011 |

| | |
|---|---|
| $cw_7$ | 01100111 |
| $cw_8$ | 10010111 |

Se va vedea acum modul în care anumite micro-ordine activate defectuos produc non - cuvinte - cod:

1. ADD R1,R3 rezultă în R1 non – cuvântul - cod 11011011.
2. EXCHANGE R5,R7 are drept rezultat atât în R5 cât şi în R7, cuvinte-cod incorecte.
3. OR R7, R8 produce în R7 non – cuvântul - cod 11110111.

De remarcat că operaţiile realizate asupra unor non – cuvinte - cod pot conduce la cuvinte - cod. Spre exemplu, dacă R4 şi R5 au non – cuvintele - cod 00010101 şi respectiv 10011001, după OR R4, R5 rezultatul din R4 este un cuvânt - cod.
Daca toate registrele sunt încărcate cu cuvinte - cod corespunzătoare, defectele simple ce afectează execuţia unei instrucţiuni *I* va cauza fie un rezultat incorect al instrucţiunii *I* fie va cauza unui registru sa aibă drept conţinut un non – cuvânt - cod sau cuvântul - cod al unui alt registru. Deci, pentru a detecta aceste defecte, toate registrele trebuie sa fie citite după executarea instrucţiunii *I*. În cazul în care un registru nu poate fi citit în mod direct, atunci secvenţa de citire a acestuia trebuie să fie *nedistructivă*, adică să nu modifice conţinutul nici unui alt registru (exceptând poate registrul contor program).
◊

*Exemplul 8:* Pentru microprocesorul considerat în Exemplul 4, secvenţa READ(R2) folosita în secvenţa anterioara - (MOV A,R2; MOV(IX),A) - distruge conţinutul registrului A. O secvenţă nedistructivă READ(R2), care salvează şi apoi restaurează conţinutul registrului A, arată astfel:

READ(R2)=(PUSH A; MOV A,R2; MOV (IX),A; POP A)

În virtutea principiului *Începe-cu-puţin*, se vor verifica întâi secvenţele READ şi abia după aceea secvenţele corespunzătoare celorlalte instrucţiuni. Defectele ce afectează secvenţele READ se clasifică în raport cu tipurile de micro-ordine care le activează:

- tipul 1: micro-ordine care operează asupra unui registru; spre exemplu, incrementează, neagă sau roteşte;
- tipul 2: micro-ordine care cauzează un transfer de date între două registre; spre exemplu, transfer sau interschimb;
- tipul 3: micro-ordine care executa operaţii aritmetice sau logice asupra a două registre sursă; spre exemplu, suma.

Fie $S_1$ mulţimea registrelor modificate de micro-ordinele de tipul 1 şi fie $S_2$ mulţimea perechilor de registre ($R_i$, $R_j$) cuprinse în micro-ordinele de tipul 2, unde $R_j$ şi $R_i$ sunt respectiv, registrele sursa şi destinaţie. Fie $S_3$ mulţimea tripletelor de registre ($R_i$, $R_j$, $R_k$) unde $R_j$ şi $R_k$ sunt registrele sursă iar $R_i$ este registrul destinaţie. Procedeele *Read1*(), *Read2*() şi *Read3*() sunt concepute să detecteze respectiv defecte de tipul 1, 2 şi 3 care ar afecta execuţia unor secvenţe READ. Fiecare procedeu începe prin încărcarea cuvintelor - cod în registre. Deoarece în acest punct al testării se presupune că nu se ştie dacă secvenţele WRITE funcţionează corect (sunt libere de defecte) procedeele de testare iau în seama faptul că anumite registre s-ar putea să nu-şi conţină cuvintele - cod.
**Teorema 3:** Daca procedeele *Read1*(), *Read2*() şi *Read3*() se execută fără detectarea nici unei erori, atunci toate secvenţele READ sunt corecte (libere de defecte).

**Demonstrație:** Se va arată numai că procedura *Read3*() detectează toate defectele de tipul 3 care afectează secvențele READ. Demonstrația este similara în cazul procedeelor *Read1*(), *Read2*().

Se consideră un defect de tipul 3 care afecteaza execuția secvenţei READ($R_i$) prin activarea eronată a micro-ordinelor ce folosesc $R_k$ şi $R_l$ ca registre sursă şi modifică registrul $R_j$. Dacă $R_k$ şi $R_l$ au propriile lor cuvinte-cod, prima secvenţă READ($R_j$) detectează un non - cuvânt - cod (sau un cuvânt - cod incorect) în $R_j$. Totuşi, dacă micro-ordinul activat de READ($R_i$) operează o dată incorectă în $R_k$ şi / sau $R_l$, atunci aceasta poate produce un cuvânt - cod corect în $R_j$, aşa că prima secvenţă READ($R_j$) să nu detecteze o eroare. În continuare, READ($R_k$) verifica conținutul registrului $R_k$; dar chiar dacă $R_k$ nu a fost încărcat cu propriul sau cuvânt - cod, este posibil ca prima secvenţă READ($R_j$) să-l schimbe pe $R_k$ astfel încât acest registru să aibă valoare corectă.

```
Read1()
begin
        for every Ri WRITE(Ri) cwi
        for every Ri
                for every Rj ∈ S1
                        begin
                                READ(Ri)
                                READ(Rj)
                                READ(Ri)
                                READ(Rj)
                        end
end
```

Figura 8. Testarea tipului 1 de defecte.

```
Read2()
begin
        for every Ri WRITE(Ri) cwi
        for every Ri
                for every (Rj,Rk)∈S2
                        begin
                                READ(Ri)
                                READ(Rj)
                                READ(Rk)
                                READ(Ri)
                                READ(Rj)
                        end
end
```

Figura 9. Testarea tipului 2 de defecte.

```
Read3()
begin
        for every Ri WRITE(Ri) cwi
        for every Ri
                for every (Rj,Rk,Rl)∈S3
                        begin
                                READ(Ri)
                                READ(Rj)
                                READ(Rk)
                                READ(Rl)
                                READ(Rk)
                                READ(Rl)
                                READ(Ri)
                                READ(Rj)
                        end
end
```

Figura 10. Testarea tipului 3 de defecte.

În mod similar, READ($R_l$) fie detectează o eroare, fie arată că $R_l$ are valoarea corectă (corespunzătoare). Dar READ($R_l$) poate schimba conținutul registrului $R_k$. Dacă următoarele READ($R_k$) şi READ($R_l$) nu detectează erori, atunci putem fi siguri că atât $R_k$ cât şi $R_l$ au acum drept conținut cuvinte - cod corecte. Astfel, atunci când a doua secvență READ($R_i$) este executată, micro-ordinul defectuos activat produce date incorecte în $R_j$ şi a doua secvență READ($R_j$) detectează această eroare. De aceea dacă nu se detectează nici o eroare se poate trage concluzia că READ($R_i$) este liberă de defecte de tipul 3. Cum *Read3*() repetă acest test pentru fiecare registru $R_i$, sunt detectate toate defectele eventuale de tipul 3 ce ar afecta secvențele READ. Procedurile *Read* detectează deasemenea şi anumite defecte care afectează secvențele WRITE. Procedura *Load* detectează toate defectele care afectează secvențele WRITE. Procedura *Load* presupune că secvențele READ sunt libere de defecte. Procedura următoare, numită *Instr,* verifică prezența tuturor defectelor ce ar afecta execuția fiecărei instrucțiuni din setul de instrucțiuni al microprocesorului. Procedura *Instr* presupune ca atât secvențele WRITE cât şi READ sunt libere de defecte. La Întrebarea referitoare la momentul când trebuie testată orice instrucțiune pentru fiecare mod de adresare, răspunsul depinde de ortogonalitatea setului de instrucțiuni.

```
Load()
begin
        for every Ri WRITE(Ri) cwi
        for every Ri
                begin
                        READ(Ri)
                        WRITE(Ri) cwi
                        READ(Rj)
                end
end
```

Figura 11. Testarea secvențelor WRITE.

```
Instr()
begin
        for every instructiune I
                begin
                        for every Ri WRITE(Ri) cwi
                        excută I
                        for every Ri READ(Ri)
                end
end
```

Figura12. Testarea tuturor instrucțiunilor.

secvența de test completă pentru funcția decodificare - instrucțiune şi secvențiere a-instrucțiune a microprocesorului se obține prin execuția secvenței de proceduri *Read1*(), *Read2*(), *Read3*(), *Load*() şi *Instr*().

În cazul cel mai defavorabil lungimea secvenței de test care verifică secvențele READ este proporțională cu $n_R^4$ iar lungimea secvenței de test care verifică execuția fiecărei

instrucțiuni este proporțională cu $n_R n_I$, unde $n_I$ este numărul de instrucțiuni din respectivul set.

#### 4.3.3 Testarea funcțiilor memorare - date şi transfer - date

Funcțiile memorare - date şi transfer - date sunt testate împreună pentru că un test care detectează defecte de tip blocaj pe liniile de transfer ale unei căi A→B detectează deasemenea defecte blocaje în registrele corespunzătoare nodurilor A şi B.

Se consideră o secvență de instrucțiuni ce activează o secvență de transferuri de date începând în nodul IN şi sfârşind în nodul OUT. O astfel de secvență se va numi *un transfer IN/OUT.* Pentru microprocesorul din Exemplul 4, secvența (MVI A,a; MOV R1,A; MOV (IX),R1) este un transfer IN/OUT care mută date de-a lungul căii:

$$\text{IN} \rightarrow \text{A} \rightarrow \text{R1} \rightarrow \text{OUT}.$$

Un test pentru căile de transfer care implică un transfer IN/OUT constă din repetarea transferului pentru diferite formații ale datelor, astfel încât :

1. Fiecare bit al căii de transfer este poziționat atât în 0 cât şi în 1.
2. Fiecare pereche de biți este poziționată la valori complementare.

Un exemplu de formații de date cu 8 biți ar putea fi următorul:

1 1 1 1 1 1 1 1
1 1 1 1 0 0 0 0
1 1 0 0 1 1 0 0
1 0 1 0 1 0 1 0

0 0 0 0 0 0 0 0
0 0 0 0 1 1 1 1
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1

În mod evident pentru fiecare cale de transfer cuprinsă într-un transfer IN / OUT, o secvență de test construită în această manieră va detecta toate defectele blocaje pe liniile căii de transfer şi toate scurtcircuitele dintre oricare două din aceste linii. Testul complet pentru funcțiile memorare - date şi transfer-date constă dintr-un set de transferuri astfel încât fiecare cale de transfer a microprocesorului este implicata în cel puțin un transfer de IN/OUT.

### 5 Concluzii.

Metodele de testare funcțională încearcă să reducă complexitatea problemei de generare a testelor, prin abordarea acesteia la un nivel de abstractizare mai elevat. Totuși, testarea funcțională nu a atins încă nivelul de maturitate şi de perfecționare al metodelor structurale.

Chiar dacă diagramele de decizie binara oferă modele funcționale care sunt uşor de folosit în generarea testelor, aplicabilitatea acestora în modelarea sistemelor complexe nu a fost încă încercată.

Testarea pseudo-exhaustiva este cea mai indicata pentru circuite cu o structură de tip logic-iterativ.
Pentru circuite combinaționale oarecari, unde cel puțin o LPE depinde de multe LPI, numărul de teste necesitat pentru testarea pseudo-exhaustiva devine prohibitiv.

Tehnicile de partiționare pot reduce numărul de teste dar se bazează pe cunoaşterea internă a circuitului şi aplicabilitatea acestora la circuite mari şi foarte mari este problematică din cauza lipsei, încă, a unor algoritmi de partiționare satisfăcători.
Modelele explicite funcționale ale defectelor se pare că produc un set prohibitiv de mare al defectelor ţintă.
Modelele funcționale implicite ale defectelor au fost folosite cu succes în testarea memoriilor cu acces aleator (RAM) şi în testarea dispozitivelor programabile cum ar fi microprocesoarele, pentru care vectorii de test pot fi dezvoltați ca secvențe de instrucțiuni.

Procesul de generare al testului se bazează pe arhitectură şi setul de instrucțiuni al microprocesorului şi produce procedee de test ce detectează clase de defecte fără a fi necesară o enumerare explicită a acestor defecte.
Acest proces nu a fost încă automatizat şi nu poate genera teste pentru funcțiile manipulare-date.
În general, metodele de testare funcțională sunt strâns cuplate cu tehnicile de modelare funcțională.
Astfel, aplicabilitatea metodei de testare funcțională este limitata la sisteme descrise printr-o tehnică de modelare particulară.
Deoarece exista o multitudine de tehnici diferite de modelare funcțională, pare a fi puțin probabil ca să se dezvolte o metodă de testare funcțională aplicabilă în general.
În plus, deducerea unui model funcțional folosibil în generarea testelor este adesea un proces manual, de lunga durată şi supus unei probabilități, destul de mari, de a fi eronat.