Simularea calculelor cuantice

Referat 2

drd. Adina –Luminiţa BĂRÎLĂ

conducător ştiinţific
prof. univ. dr.ing. Ştefan Gheorghe PENTIUC

septembrie 2014

Investeşte în oameni !
FONDUL SOCIAL EUROPEAN
Proiect cofinanțat din Fondul Social EUROPEAN prin Programul Operaţional Sectorial Dezvoltarea Resurselor Umane 2007 – 2013

Această lucrare a beneficiat de suport financiar prin proiectul

Performanță sustenabilă în cercetarea doctorală și post doctorală - PERFORM

Contract nr POSDRU 159/1.5/S/138963

Axa prioritară 1 - Educaţia şi formarea profesională în sprijinul creşterii economice şi dezvoltării societăţii bazate pe cunoaştere

Domeniul major de intervenţie 1.5 - „Programe doctorale şi postdoctorale în sprijinul cercetării”

CUPRINS

1 Simularea calculelor cuantice 4
1.1 Necesitatea simulării calculului cuantic 4
1.2 Clasificarea simulatoarelor de calculatoare cuantice 5
1.3 Simulatoare ce folosesc calculul paralel şi distribuit 10
2 Limbaje de programare cuantică 12
3 Limbajul QCL (Quantum Computation Language) 13
3.1 Caracteristicile limbajului QCL 13
3.2 Structura unui program QCL 13
3.3 Expresii clasice 14
3.4 Regiştri cuantici şi expresii cuantice 16
3.5 Instrucţiuni 18
3.6 Subrutine 19
3.7 Operatori elementari 22
4 Implementarea în QCL a algoritmilor cuantici 25
4.1 Algoritmul Deutsch 25
4.2 Algoritmul Deutsch-Jozsa 28
4.3 Algoritmul Bernstein-Vazirani 31
4.4 Algoritmul Simon 34
4.5 Algoritmul lui Grover 42
4.6 ALgoritmul lui Shor 47
5 Bibliografie 53

1 Simularea calculelor cuantice

1.1 Necesitatea simulării calculului cuantic

Hardware-ul cuantic disponibil în prezent este insuficient pentru explorări detaliate ale algoritmi cuantici propuşi. Din acest motiv, este utilă crearea şi testarea algoritmilor cuantici utilizând simulări pe hardware-ul clasic.

Simulatoarele de calculatoare cuantice sunt definite ca fiind programe de calculator care pot fi rulate pe o maşină clasică pentru a simula acţiunile unui computer cuantic. Aceste simulări implică utilizarea maşinilor care lucrează în acord cu legile fizicii clasice newtoniene pentru a simula maşini care lucrează în acord cu legile mecanicii cuantice.

Implementarea şi simularea algoritmilor cuantici cunoscuţi utilizând un simulator de calculator cuantic va stimula dezvoltarea de noi algoritmi cuantici. De asemeni, simulatoarele pot permite investigări în corectarea erorilor cuantice şi fezabilitatea pe scară largă a calculatoarelor cuantice. Utilizând simulatoare de calculatoare cuantice e posibilă observarea stărilor cuantice în timpul calculului (observarea stărilor intermediare ale calculului), observare ce este interzisă de legile mecanicii cuantice. Posibilitatea de a accesa starea maşinii simulate este utilă pentru a analiza efectele erorilor. De asemeni, unele simulatoare permit ca stările maşinii simulate să fie salvate şi încărcate în/din fişier, ceea ce are avantajul că fişierele cu stările cuantice pot fi partajate facilitând astfel duplicarea experimentelor de către alţi cercetători.

Aşa cum a observat Feynman, singurul mod de a modela efectiv un sistem de mecanică cuantică este prin utilizarea altui sistem de mecanică cuantică. Un simulator de calculator cuantic este o încercare de a modela un sistem cuantic pe un sistem clasic şi nu e posibil să se facă acest lucru eficient. Pentru a simula un vector de stare într-un spaţiu Hilbert de dimensiune 2^n un calculator clasic trebuie să manipuleze vectori conţinând 2^n numere complexe, în timp ce un calculator cuantic necesită doar n qubiţi, ceea ce îl face mult mai eficient în ceea ce priveşte spaţiul de stocare. Cu fiecare qubit suplimentar simulat are loc o creşterea exponenţială a numărului de calcule şi o scădere a timpului de execuţie care este notabilă chiar în simulări ale sistemelor ce implică un număr relativ mic de qubiţi.

Simularea unui calcul cuantic este exponenţială atât în timp cât şi în spaţiu. Dacă ar fi posibilă construirea unui simulator de calculator cuantic eficient atunci nu ar mai fi necesară construirea unui calculator cuantic utilizând hardware cuantic. Simulatorul cuantic însuşi ar fi un calculator cuantic.

În prezent, există o mare varietate de simulatoare de calcul cuantic (şi altele sunt în curs de dezvoltare), acestea variind în ceea ce priveşte complexitatea simulărilor posibile, reprezentările utilizate pentru structurile de date cuantice, implementările algoritmilor cuantici şi acurateţea simulării însăşi.

1.2 Clasificarea simulatoarelor de calculatoare cuantice

O clasificarea a simulatoarelor de calculatoare cuantice a fost realizată în 2006 de către H. De Raedt şi K. Michielsen. Ei au împărţit software-ul existent în cinci categorii:

- limbaje de programare pentru calculatoare cuantice
- compilatoare cuantice
- simulatoare de circuite cuantice sau simulatoare la nivel de porţi cuantice
- software care simulează modele pentru realizarea fizică a calculatoarelor cuantice
- simulatoare pur pedagogice

1.2.1 Limbaje de programare pentru calculatoare cuantice

Limbajele de programare exprimă semantica unui proces de calcul într-o manieră abstractă şi generează automat o secvenţă de operaţii elementare pentru a controla calculatorul.
Din categoria limbajelor de programare cuantică De Raedt menţionează Quantum Computation Language - QCL, Q language, Quantum Superpositions, QuBit, Quantum Entanglement, Q-gol, Quantum Fog, QDD şi Quantum Lambda Calculus.
QCL este unic între acestea fiind atât un limbaj de programare care poate lucra pe orice arhitectură de calculator cuantic bazat pe qubiţi, cât şi un limbaj de simulare a calculului cuantic.

1.2.1.1 QCL

QCL – Quantum Computer Language – este un limbaj de programare de nivel înalt pentru programare cuantică dezvoltat de Bernhard Ömer. Este independent de arhitectură şi are o sintaxă asemănătoare limbajelor procedurale clasice precum C şi Pascal. Include fişiere program pentru simularea unei implementări a algoritmului lui Shor şi fişiere pentru simularea altor aspecte ale calculului cuantic. QCL a fost dezvoltat sub Linux.

Prima versiune a apărut în 1998, iar versiunea actuală a apărut în 2006.

1.2.1.2 Q language

Q language este o implementare C++ a unui limbaj de programare cuantică.

1.2.1.3 Quantum Superpositions

Quantum Superpositions este o bibliotecă PERL care permite programatorilor să utilizeze variabile care pot păstra mai multe valori în acelaşi timp. Versiunea 1.03 a apărut în 2000. Versiunea curentă (2.02) a apărut în aprilie 2003.

1.2.1.4 Quantum Entanglement

Quantum Entanglement este o librărie PERL, apărută în 2002, care permite utilizatorilor să pună variabile într-o superpoziţie de stări, să interacţioneze între ele şi să le observe.

1.2.1.5 Quantum Fog

Quantum Fog este o aplicaţie Macintosh pentru modelarea situaţiilor fizice care prezintă comportament cuantic. Este un instrument pentru investigarea şi discutarea grafică a problemelor de măsurare cuantică în termeni de de reţele cuantice Bayesiene. Simulează un calculator cuantic pentru scop general. A apărut în 1997, iar versiunea curentă este 2.0, apărută în 2006.

1.2.1.6 QDD

QDD este o bibliotecă C++ care oferă un set relativ intuitiv de construcţii de calcul cuantic în contextul unui mediu de programare C++. Emularea calculului cuantic se bazează pe reprezentarea BDD (Binary Decision Diagram) a stărilor cuantice (spre deosebire de QCL care utilizează reprezentarea prin numere complexe). Include o implementare a algoritmului lui Shor. Versiunea 0.2 a fost lansată în septembrie 1999, versiunea 0.3 este în februarie 2003.

1.2.1.7 Quantum Lambda Calculus

Este un limbaj funcţional bazat pe Scheme pentru exprimarea şi simularea algoritmilor cuantici.

1.2.2 Compilatoare cuantice

Un compilator cuantic primeşte la intrare o transformare unitară şi returnează o secvenţă de operaţii elementare pe unul sau doi qubiţi ce realizează operaţia cuantică specificată.
Din această categorie, De Raedt menţionează Qubiter şi GQC.

1.2.2.1 Qubiter

Qubiter este un compilator cuantic scris în C++. Primeşte la intrare o matrice unitară şi returnează la ieşire o secvenţă echivalentă de operaţii cuantice elementare, precum CNOT, rotaţii de qubiţi. Aceste secvenţe pot fi reprezentate grafic prin circuite de qubiţi.

Qubiter 1.0 a apărut în mai 1998. Versiunea curentă este 1.11 apărută în 2007.

1.2.2.2 GQC

GQC este un compilator cuantic online care returnează un circuit pentru CNOT în termeni de transformări unitare specificate de utilizator.

1.2.3 Simulatoare de circuite cuantice (simulatoare la nivel de porţi cuantice)

Simulatoarele de circuite cuantice - la un nivel abstract, calculul cuantic pe un calculator cuantic ideal constă în aplicarea unor transformări unitare asupra unui vector (de stare) cu componente valori complexe. Un calculator convenţional poate executa aceste operaţii la fel de bine, dacă se asigură suficientă memorie pentru a stoca toate componentele vectorului. Simulatoarele de circuite cuantice oferă un mediu software pentru simularea calculatoarelor cuantice ideale.

Din această categorie fac parte QCAD, QuaSi, JaQuzzi, Libquantum, QCS.

1.2.3.1 QCAD

QCAD este un mediu grafic Windows98/2000 pentru proiectarea circuitelor cuantice. Poate exporta circuitele ca fişiere BMP sau EPS, simula circuitele şi arăta stările qubiţilor. Versiunea curenta este QCAD 2.0 apărută în 2011.

1.2.3.2 QuaSi

QuaSi este un simulator de circuite cuantice pentru scop general. Pemite utilizatorilor să construiască şi să simuleze circuite grafice într-o interfaţă grafică (GUI). Sunt incluse circuite demonstrative pentru algoritmii lui Shor, Grover şi Deutsch-Jozsa. QuaSi permite utilizarea a maxim 20 de qubiţi. De asemeni, sunt oferite apleturi Java pentru utilizare pe Internet.

QuaSi(2) a apărut în martie 2002.

1.2.3.3 JaQuzzi

JaQuzzi este un simulator de computere cuantice, dezvoltat în 2000, pentru proiectarea, testarea şi vizualizarea algoritmilor cuantici până la 20 de qubiţi. Programul poate rula independent sau ca aplet. Pentru a utiliza JaQuzzi este necesară o maşină virtuală Java (versiunea 1.3 sau o versiune ulterioară).

1.2.3.4 QCSimulator

QCSimulator este un simulator de computere cuantice pentru Macintosh şi Windows. Este utilizată o interfaţă grafică pentru a construi reprezentarea prin circuit a unui algoritm cuantic şi pentru a simula algoritmii cuantici prin interschimbarea elementelor unitare cu Mathematica. Sunt incluşi algoritmul de factorizare al lui Shor,

algoritmul de căutare al lui Grover, transformata Fourier discretă şi un sumator pentru două numere. Versiunea 1.1 a apărut în martie 2000.

1.2.3.5 Libquantum

Libquantum este o bibliotecă C pentru simularea unui calculator cuantic ideal. Sunt disponibile operaţii de bază pentru manipularea regiştrilor precum porţile Hadamard şi Controlled-NOT. Măsurătorile pot fi efectuate fie pe un singur qubit, fie pe întregul registru cuantic. De asemeni, sunt incluse implementări ale algoritmului de factorizare al lui Shor şi algoritmului de căutare al lui Grover. Libquantum conţine caracteristici pentru studiul decoerenţei şi corecţiei erorilor cuantice. Libquantum este dezvoltat pe platformă GNU/Linux şi necesită instalarea unui compilator C cu suport pentru numere complexe.

Versiunea 0.2.2 a apărut în 2003. Versiunea curentă este Libquantum 1.1.1 şi a apărut în ianuarie 2013.

1.2.3.6 OpenQUACS

OpenQUACS (Open-Source QUAntum Computer Simulator) este o bibliotecă scrisă în Maple care simulează capabilităţile unui computer cuantic ideal. Simulatorul oferă un tutorial complet. Sunt incluşi câţiva algoritmi cuantici precum algoritmul lui Deutsch, teleportarea cuantică, algoritmul de căutare al lui Grover şi un sumator cuantic. A fost dezvoltat în 2000.

1.2.3.7 QuCalc

QuCalc este o bibliotecă de funcţii matematice pentru simularea circuitelor cuantice şi rezolvarea problemelor de calcul cuantic. Versiunea 2.13 a apărut în 2001, iar versiunea curentă este 4.0.

1.2.3.8 QGAME

QGAME (Quantum Gate And Measurement Emulator) este un sistem, scris în Common Lisp, care permite utilizatorului să ruleze algoritmi cuantici pe un computer digital. Are o interfaţă grafică care permite şi persoanelor fără cunoştinţe de Lisp să lucreze cu QGAME. Utilizează Macintosh Common Lisp (MCL) şi lucrează doar sub MacOS cu MCL. Nu toate caracteristicile lui QGAME sunt disponibile din interfaţa grafică. QGAME însuşi este o platformă indepentă (rulează pe orice platformă pentru care este disponibil mediul Common Lisp); doar interfaţa grafică necesită Macintosh Common Lisp. Versiunea 1 a fost lansată în iunie 2002.

1.2.3.9 QCompute

QCompute este un simulator de computer cuantic scris în Pascal care execută operaţii pe un număr arbitrar de qubiţi. A apărut în iulie 1997.

1.2.3.10 Quantum

Quantum este un simulator de circuite cuantice scris în C++ pentru mediul Windows. Include o implementare a algortimului lui Shor şi a fost dezvoltat în 1997.

1.2.3.11 Eqcs

Eqcs este o bibliotecă ce permite clienţilor să simuleze un calculator cuantic. Include un program ce arată crearea unei porţi controlled NOT. A fost dezvoltată în 1999, iar versiunea curentă a apărut în martie 2012.

1.2.3.12 QCS

QCS (Quantum Computer Simulator) este un simulator de computer cuantic scris în C++. A fost dezvoltat în 2001.

1.2.4 Software care simulează modele pentru realizarea fizică a calculatoarelor cuantice

Această categorie de simulatoare constă în software care utilizează un Hamiltonian dependent de timp pentru implementarea transformărilor unitare asupra qubiţilor. Aceste simulatoare permit emularea diverselor modele hardware pentru calculatorul cuantic, adică simulează modele de realizări fizice ale calculatorului cuantic. Simulatoarele din această categorie pot fi utilizate şi ca simulatoare la nivel de porţi.

1.2.4.1 QCE

QCE (Quantum Computer Emulator) este un instrument software care emulează un calculator cuantic precum şi implementarea fizică a unui hardare cuantic. QCE utilizează Hamiltonieni dependenţi de timp şi evoluţii unitare pentru a simula procesele fizice care guvernează operaţiile unui procesor cuantic. QCE oferă un mediu pentru depanarea şi execuţia algoritmilor cuantici în condiţii experimentale realiste. Pachetul QCE include multe exemple precum algoritmul lui Shor pentru factorizarea întregilor, găsirea ordinului, transformata Fourier cuantică, diferite implementări ale algoritmului lui Deutsch-Jozsa, algoritmul lui Grover de căutare. Softwre-ul constă într-o interfaţă grafică utilizator (GUI) simulatorul însuşi. Rulează sub Windows 98/NT4/2000/ME/(XP cu SP1)Windows 7 şi Linux+VMware pe maşini cu procesoare Intel/AMD.

Versiunea curentă este QCE 10.11 apărută în noiembrie 2010.

1.2.4.2 QSS

QSS (Quantum System Simulator) este un instrument software care simulează calcule cuantice cu Hamitonian dependent de timp. Oferă o interfaţă grafică utilizator simplă care permite utilizatorilor să specifice Hamitonienii complecşi ca sume a unora mai mici. Motorul simulatorului este proiectat ca un modul independent, deci este independent de interfaţa utilizator şi poate fi dezvoltat uşor. QSS rulează sub sistemul de operare Windows.

1.2.5 Simulatoare pur pedagogice

Aceste programe sunt utilizate în scop pedagogic, pentru a ilustra elementele care sunt relevante pentru calculul cuantic şi construirea simulatoarelor de calculatoare cuantice: animaţii ale sistemelor cuantice, instrumente de bază pentru notaţia Dirac, simularea corecţiei erorilor cuantice, simularea algoritmilor cuantici cunoscuţi, etc.

Din această categorie fac parte:

- Quantum Turing machine simulator (simulator pentru maşina Turing cuantică)
- QTM simulator
- Quantum Search Simulator
- Grover's algorithm
- Shor's algorithm simulator

1.3 Simulatoare ce folosesc calculul paralel şi distribuit

Datorită naturii exponenţiale a calculului cuantic, resursele computaţionale pentru simulare sunt imense. Astfel, în ultimii ani au fost dezvoltate simulatoare ce folosesc calculul paralel şi distribuit.

Primul simulator de acest tip (Parallel Quantum Computer Simulator) a fost dezvoltat de Obenland şi Despain în 1998. Acesta este scris în C şi rulează în sisteme paralele cu memorie ditribuită. Simulatorul primeşte la intrare descrierea unui circuit cuantic specificat în termeni de porţi logice şi implementează porţi cuantice CNOT cu unul, doi sau trei biţi de control precum şi porţi de rotaţie. Acestea sunt implementate sub forma unei secvenţe de transformări laser definite ca în schema capcanei de ioni introdusă de Cirac şi Zoller. Circuitele au fost create pentru a permite simularea algoritmilor lui Shor şi Grover.

În 2003 Glendinning şi Ömer au introdus o variantă de paralelizare a librăriei QC-lib, mediul de simulare oferit pentru execuţia programelor QCL în absenţa resurselor cuantice. Ei au descris paralelizarea sa folosind MPI (Message Passing Interface) şi au prezentat performanţele obţinute în urma rulării pe un cluster Beowulf. Utilizarea mai multor procesoare a determinat o accelarare semnificativă atât în cazul transformării

Hadamard cât şi în cazul algoritmului lui Grover. Astfel, în cazul transformării Hadamard s-a ajuns la 6 s pe 8 procesoare folosind 19 qubiţi (faţă de 14,8 s pe un singur procesor).

În 2003, Geva Patz a proiectat şi implementat un mediu care să permită simularea calculului cuantic pe calculatoare clasice utilizând un cluster cu 32 de noduri. Testele efectuate au arătat că modelul matricilor pentru reprezentarea circuitelor cuantice nu este potrivit datorită costurilor mari de comunicaţii între noduri. S-a demonstrat că, pentru medii de simulare care rulează în sisteme distribuite, este mai potrivită descrierea algoritmilor cuantici folosind operatorul produs tensorial. Această abordare are totuşi un dezavantaj: nu toate problemele de calcul cuantic permit o astfel de paralelizare.

Fraunhofer Computing Portal reprezintă un serviciu de calcul şi un spaţiu de lucru colaborativ pentru simularea şi investigarea algoritmilor cuantici. Deşi nu mai este disponibil, scopul acestui proiect era de a pune la dispoziţia cercetătorilor un motor de simulare format din mai multe noduri de calcul prin intermediul unei interfeţe web uşor accesibilă. Pentru motorul de simulare a fost proiectat un software modular care să permită integrarea facilă a diverselor tehnici de simulare a proceselor cuantice. În cadrul acestui proiect a fost introdus limbajul QML - Quantum Markup Language prin intermediul căruia sunt specificate datele de intrare pentru simulare. Deşi proiectul utilizează un cluster Linux cu o reţea de interconectare Myrinet, nodurile de calcul nu sunt folosite pentru execuţia în paralel a simulării unui circuit cuantic, ci pentru a furniza diferite servicii de rezolvare a unor anumite probleme folosind un anumit algoritm (de ex. construcţia şi diagonalizarea unei matrice pentru calcularea întregului spectru a unui operator Hamiltonian de dimensiune limitată, sau reprezentarea eficientă a stării în raport cu spaţiul prin analiza topologiei porţilor pentru simularea circuitelor cuantice de mari dimensiuni).

În 2007 a fost prezentat un pachet software în Fortran 90, dezvoltat pentru simularea unui calculator cuantic ideal. Software-ul poate rula pe diferite arhitecturi de calculatoare, de la PC-uri la maşini paralele cu memorie distribuită şi/sau partajată. Numărul maxim de qubiţi este setat de memoria maşinii de calcul pe care rulează simulatorul. Autorii au prezentat performanţele software-ului simulând computere cuantice cu maxim 36 qubiţi, utilizând până la 4096 de procesoare şi 1 TB de memorie.

În 2009 Frank Tabakin şi Bruno Juliá-Díaz au prezentat un alt software de simulare cu capabilităţi de procesare paralelă - QCMPI. Acesta este un instrument de cercetare accesibil care permite evaluarea rapidă a algoritmilor pentru un număr mare de qubiţi, precum şi studiul stabilităţii şi corecţiei erorilor proceselor de calcul cuantic. QCMPI extinde pachetul QDENSITY pentru Mathematica prin implementarea procesării paralele folosind MPI. Sunt oferite ca exemple coduri algoritmul de factorizare al lui Shor şi algoritmul de căutare al lui Grover.

O nouă abordare a simulării calculelor cuantice a apărut recent odată cu proliferarea calculului folosind hardware-ul grafic. Datorită marii puteri de calcul a unităţilor de procesare grafică (GPU - Graphical Processing Unit), acestea au devenit din ce în ce mai populare pentru rezolvarea problemelor de natură generală. Cum simularea calculelor cuantice este o problemă cu complexitate exponenţială, se pretează la execuţia în paralel pe arhitectura multiprocesor SIMD oferită de cele mai recente plăci grafice. O astfel de abordare pentru simularea calculelor cuantice poate fi implementată folosind arhitectura CUDA (Compute Unified Device Arhitecture) oferită de NVIDIA, accelerarea raportată fiind de până la 85 faţă de implementarea pe CPU.

2 Limbaje de programare cuantică

Deşi nu există hardware cuantic, există mai multe aspecte importante care justifică necesitatea dezvoltării unui limbaj de programare cuantică:

• examinarea teoretică a algoritmilor cuantici. În prezent, algoritmii cuantici sunt descrişi, în principal, fie prin intermediul unui pseudocod parţial formal, fie sub forma unui circuit cuantic. Totuşi, pseudocodul nu prezintă aceeaşi exactitate precum un limbaj bine definit, iar modelul circuitului cuantic nu este suficient de expresiv.

• examinarea experimentală a algoritmilor cuantici. În unele cazuri nu este posibilă demonstrarea formală a corectitudinii sau eficienţei unui anumit algoritm. Aceasta ar putea depinde de presupuneri matematice nedemonstrate sau de metode euristice, caz în care este necesară testarea algoritmului. Chiar şi în absenţa unui calculator cuantic, prin simulare ar putea fi inspectate abilităţile şi problemele unui algoritm (cel puţin pentru un număr mic de qubiţi).

Au fost dezvoltate diferite limbaje de programare cuantică ce pot fi clasificate în funcţie de scopul programării în ”limbaje de programare practice” şi ”limbaje de programare formale”. Limbajele de programare cuantică din prima categorie (Q language, QCL) au ca scop simularea sau programarea efectivă a calculatoarelor cuantice când acestea vor deveni disponibile. Limbajele din a doua categorie sunt destinate analizei teoretice a programelor cuantice. Din acest punct de vedere, specificarea formală a unui limbaj de programare cuantică este separată în două părţi: sintaxa şi semantica.

Din punctul de vedere al modelării semanticii unui program cuantic, se disting limbaje imperative (QCL, qGCL, Q language), limbaje funcţionale (QFC), calcul lambda, limbaje pentru procese concurente (QPAlg, CQP). Un alt aspect foarte important în specificarea unui limbaj de programare cuantică îl reprezintă modelul matematic pentru descrierea legilor fizicii cuantice. Cele mai comune abstracţiuni utilizate în calculul cuantic sunt modelul operatorilor unitari, modelul matricei de densitate şi calculul lambda cuantic.

Caracteristicile ce trebuiesc îndeplinite de un limbaj de programare cuantică sunt:

• completitudine - să permită implementarea oricărui algoritm cuantic valid;

• extensie clasică - să includă o paradigmă de calcul clasic de nivel înalt pentru a permite integrarea cu minim de efort a pre- şi post-procesării clasice;

• separabilitate - programarea clasică să poată fi separată de cea cuantică pentru ca acele calcule ce nu pot beneficia de vreo accelerare prin execuţia pe un dispozitiv cuantic, să fie rulate pe o maşină clasică;

• expresivitate - să permită construcţii de nivel inalt;

• independenţa hardware - limbajul trebuie să fie independent de implementarea hardware a dispozitivului ce urmează a fi exploatat, permiţând recompilarea codului pentru diferite arhitecturi de calculatoare cuantice fără intervenţia programatorului.

3 Limbajul QCL (Quantum Computation Language)

3.1 Caracteristicile limbajului QCL

QCL este un limbaj de programare cuantică de nivel înalt, scris în C++. Este open-source şi rulează sub Linux. QCL foloseşte o reprezentare a stării cuantice sub formă de numere complexe. Alte simulatoare folosesc reţele Bayesiene (Quantum Fog, Qubiter) sau diagrame binare de decizie (QDD) pentru reprezentarea stării cuantice.
Caracteristicile sale principale sunt:

- control clasic cu funcţii, flow-control, operaţii de intrare/ieşire şi diferite tipuri de date clasice (int, real, complex, boolean, string)
- două tipuri de operatori cuantici: generali (`operator`) şi porţi reversibile pseudo-clasice (`qufunc`)
- tipuri de date cuantice (qubit registers)
- funcţii pentru manipularea regiştrilor cuantici
- limbaj universal: poate implementa şi simula toţi algoritmii cuantici cunoscuţi

Interpretorul `qcl` integrează un simulator numeric şi un mediu shell pentru utilizare interactivă. Interpretorul `qcl` simulează un calculator cuantic cu un număr arbitrar de qubiţi (implicit este lungimea cuvântului sistemului) şi este apelat cu următoarea comandă:

qcl [`options`] [`QCL-file`]...

QCL, ca limbaj de programare, este proiectat să lucreze cu orice arhitectură de computer cuantic. Cum dezvoltarea hardware-ului necesar va mai lua timp, QCL suportă şi simularea computerelor cuantice şi oferă comenzi speciale pentru accesarea stării simulate a maşinii.

Interpretorul qcl poate simula calculatoare cuantice cu un număr arbitrar de qubiţi. Sunt stocaţi doar vectorii bazei cu amplitudine non-zero.

3.2 Structura unui program QCL

Un program QCL este o secvenţă de instrucţiuni şi definiţii citite dintr-un fişier sau direct de la prompt (în acest caz intrarea este restricţionată la o linie terminată prin caracterul ‘;’)

qcl-input ← *{stmt|def}*

Instrucţiunile pot fi comenzi simple, apeluri de proceduri, stucturi de control complexe şi sunt executate cand apar.

```
qcl> if random()>=0.5 { print "red"; } else { print "black"; }
: red
```

Definiţiile nu sunt executate, dar sunt legate de o valoare (variabilă sau constantă) sau de un bloc de instrucţiuni printr-un un simbol (identificator)

```
qcl> int counter=5;
qcl> int fac(int n) { if n<=0 {return 1;} else {return n*fac(n-1);} }
```

Astfel, fiecare simbol are un tip asociat, care poate fi un tip de date sau un tip-rutină, şi stabileşte dacă simbolul este accesat printr-o referinţă sau prin apel.

Expresiile sunt compuse din literali, referinţe de variabilă şi sub-expresii combinate prin operatori şi apeluri de funcţii.

```
qcl> print "5 out of 10:",fac(10)/fac(5)^2,"combinations."
:5 out of 10: 252 combinatii.
```

Comentariile pot fi introduse în două moduri, la fel ca în limbajul C:

- comentariu pe un rând – începe cu `//`
- comentariu pe mai multe rânduri – începe cu `/*` şi se termină cu `*/`

Includerea fişierelor este realizată prin comanda

`include "`*`filename`*`"`

care determină interpretorul să proceseze fişierul *`filename.qcl`*. Fişierul este căutat în directorul curent sau în directorul include implicit (acesta poate fi schimbat cu opţiunea `include-path`.

3.3 Expresii clasice

3.3.1 Tipuri de date

Tipurile de date clasice ale QCL sunt: `int`, `real`, `complex`, `boolean` şi `string`.

Type	Description	Examples
`int`	integer	`1234, -1`
`real`	real number	`3.14, -0.001`
`complex`	complex number	`(0,-1), (0.5, 0.866)`
`boolean`	logic value	`true, false`
`string`	character string	`"hello world", ""`

Tabel 1. Tipuri de date clasice

3.3.2 Operatori

a)Operatori aritmetici

Op	Description	Example	Type	Value
^	power	(0,1)^-1.5	complex	$-\frac{1}{\sqrt{2}}(1+i)$
	integer power	(-2)^11	int	-2048
-	unary minus	--1	int	1
*	multiplication	(0,1)*(0,1)	complex	-1
/	division	3./2	real	$\frac{3}{2}$
	integer division	3/2	int	1
mod	integer modulus	100 mod 16	int	4
+	addition	1.5+1.5	real	3
-	subtraction	(1,2)-(0,2)	complex	1

Tabel 2. Operatori aritmetici

b)Operatori de comparaţie şi logici

Op	Description	Argument type
==	equal	all arithmetic, string
!=	unequal	all arithmetic, string
<	less	integer, real
<=	less or equal	integer, real
>	greater	integer, real
>=	greater or equal	integer, real
not	logic not	boolean
and	logic and	boolean
or	logic inclusive or	boolean
xor	logic exclusive or	boolean

Tabel 3. Operatori de comparație și logici

c)Operatorul de concatenare & combină două şiruri sau doi regiştri
d)Operatorul size # determină lungimea (numărul de qubiţi) unei expresii cuantice.

3.3.3 Funcţii

a)Funcţii trigonometrice

Funcţiile trigonometrice au argumente de tip `int`, `real` sau `complex` iar tipul valorii returnate este `real` sau `complex` în funcţie de tipul argumentului.

Funct.	Description	Funct.	Description
sin(x)	sine of x	sinh(x)	hyperbolic sine of x
cos(x)	cosine of x	cosh(x)	hyperbolic cosine of x
tan(x)	tangent of x	tanh(x)	hyperbolic tangent of x
cot(x)	cotangent of x	coth(x)	hyperbolic cotangent of x

Tabel 4. Funcții trigonometrice

b)Exponenţi şi logaritmi

Funcţiile `exp`, `log` şi `sqrt` lucrează cu toate tipurile numerice. Funcţia `sqrt` nu poate avea argumente negative, iar funcţia `log` trebuie să aibă argumente pozitive.

Funct.	Description
`exp(x)`	e raised to the power of x
`log(x)`	natural logarithm of x
`log(x,n)`	base-n logarithm of x
`sqrt(x)`	square root of x

Tabel 5. Exponenți și logaritmi

c)Funcţii pentru numere complexe

Funct.	Description
`Re(z)`	real part of z
`Im(z)`	imaginary part of z
`abs(z)`	magnitude of z
`conj(z)`	complex conjugate of z

Tabel 6. Funcții pentru numere complexe

Funcţia `abs` poate avea argument de tip `complex`, `real` sau `int`.

d)Funcţii de rotunjire

Funcţiile `ceil` şi `floor` au ca argument un număr real şi îl rotunjesc prin adaos, respectiv prin lipsă, la cel mai apropiat întreg. Tipul valorii returnate este `int`.

e)Calculul minimului şi maximului

Funcţiile `max` şi `min` au un număr arbitrar de argumente întregi sau reale şi returnează valoarea maximă, respectiv valoarea minimă. Valoarea returnată este de tip `int`, dacă toate argumentele sunt întregi, şi `real` în caz contrar.

f)cmmdc şi cmmmc

Funcţiile `gcd` şi `lcm` au ca argumente întregi şi returnează cmmdc, respectiv cmmmc al argumentelor.

g)Numere aleatoare

Pseudo funcţia `random()` returnează o valoare aleatoare din intervalul [0; 1).

3.4 Regiştri cuantici şi expresii cuantice

Memoria unui computer cuantic este o combinaţie de qubiţi. "Conţinutul memoriei" reprezintă starea combinată a tuturor qubiţilor, iar această stare se numeşte stare maşină (*machine state*).

Deci, starea maşină $|\Psi\rangle$ a unui computer de *n* qubiţi este un vector în spaţiul Hilbert $\mathcal{H} = \mathbb{C}^{2^n}$. Datorită naturii distructive a operaţiei de măsurare, $|\Psi\rangle$ nu poate fi observat direct.

QCL utilizează conceptul de registru cuantic ca o interfaţă între starea maşină şi algoritm. Un registru cuantic este un pointer la o secvenţă de qubiţi (mutual diferiţi). Toate operaţiile pe starea maşină (cu excepţia comenzii `reset`) au ca argumente regiştri cuantici.

Regiştrii cuantici legaţi la un nume simbolic se numesc *variabile cuantice*. QCL acceptă următoarele tipuri de regiştri:

1)registru cuantic general – un registru cuantic general cu *expr* qubiţi poate fi declarat astfel:

```
qureg identif[expr]
```

2)constantă cuantică – un registru poate fi declarat constant utilizând tipul `quconst`. O constantă cuantică trebuie să fie invariantă la toţi operatorii aplicaţi.

3)registru nealocat (empty) – se declară folosind tipul `quvoid`.

4)registru scratch – declarat cu tipul `quscratch`.

O expresie cuantică este o referinţă anonimă la un registru şi poate fi utilizată ca un argument al unui operator sau pentru a declara referinţe (numite) la regiştri.

Expr.	Description	Register
a	reference	$\langle a_0, a_1 \ldots a_n \rangle$
a[i]	qubit	$\langle a_i \rangle$
a[i:j]	substring	$\langle a_i, a_{i+1} \ldots a_j \rangle$
a[i\l]	substring	$\langle a_i, a_{i+1} \ldots a_{i+l-1} \rangle$
a&b	concatenation	$\langle a_0, a_1 \ldots a_n, b_0, b_1 \ldots b_m \rangle$

Tabel 7. Expresii cuantice

O referinţă registru se declară astfel:

```
type identif [=expr];
```

unde expresia cuantică *expr* este legată la registrul *identif* de tipul cuantic *type* (care poate fi `qureg` sau `quconst`).

Exemple:

```
qcl>qureg q[8];
qcl>qureg oddbits=q[1]&q[3]&q[5]&q[7];
qcl>qureg lowbits=q[0:3];
```

3.5 Instrucţiuni

3.5.1 Atribuire

Valoarea unei variabile clasice poate fi setată prin operatorul de atribuire =. Valoarea din partea dreaptă a semnului = trebuie să fie de acelaşi tip ca şi variabila.

3.5.2 Comenzile `input` şi `print`

Comanda `input` cere utilizatorului să introducă o valoare de intrare şi are următoarea sintaxă:

```
input [expr], identif;
```

Valoarea introdusă este atribuită variabilei *identif*. Opţional, poate fi indicat un şir de caractere (*expr*) care va fi afişat în locul mesajului standard.
Este interzisă utilizarea instrucţiunilor de intrare în definiţia funcţiilor şi operatorilor.

Comanda `print` are ca argumente o listă de expresii separate prin virgulă şi afişează la consolă valorile acestor expresii.

3.5.3 Blocuri

Un bloc este o listă de instrucţiuni încadrate între acolade. Spre deosebire de limbajul C, un bloc nu este o instrucţiune compusă şi este întotdeauna parte a unei structuri de control. Un bloc nu poate să conţină definiţii, iar acoladele sunt obligatorii chiar dacă blocul este format dintr-o singură instrucţiune.

3.5.4 Instrucţiunea condiţională

Instrucţiunea if permite execuţia condiţională a blocurilor, în funcţie de valoarea unei expresii booleene. Sintaxa instrucţiunii este următoarea:

```
if expr block1 [else block2]
```

3.5.5 Instrucţiuni repetitive

Instrucţiunea for are un contor (`identif`) de tip întreg care este incrementat de la valoarea $expr_{from}$ la valoarea $expr_{to}$. Corpul buclei este executat pentru fiecare valoare a contorului.

```
for identif = exprfrom to exprto [step exprstep] block
```

În interiorul corpului contorul este tratat ca o constantă. Dacă $expr_{step}$ nu e specificat, pasul este 1.

Bucla condiţională while este iterată cât timp condiţia expr este îndeplinită. Sintaxa este următoarea:

```
while expr block
```

O buclă condiţională until este iterată până când condiţia expr nu este îndeplinită şi are sintaxa:

```
block until expr;
```

3.5.6 Instrucţiunea exit

Instrucţiunea exit permite forţarea terminării anormale a unei subrutine şi are sintaxa următoare:
`exit [expr];`

Argumentul este un mesaj de eroare de tip `string`. Dacă funcţia este apelată fără argumente, subshell-ul curent este închis.

3.6 Subrutine

QCL oferă patru tipuri de subrutine: funcţii clasice, operatori pseudo-clasici (`qufunct`), operatori unitari generali (`operator`) şi proceduri (`procedure`). Declaraţia unei subrutine *`identif`* are următoarea sintaxă:

`type``|`*`routine`*`-type `*`identif arg-list body`*

unde *`routine-type`* poate fi `operator`, `qufunct` sau `procedure` iar *`arg-list`* reprezintă lista de de definiţii de argumente separate prin virgulă. O definiţie de argument este de forma:

`type identif`

iar corpul subrutinei (*`body`*) are forma:

`{{`*`const-def`*`|`*`var-def`*`} {`*`stmt`*`}}`

Cele patru forme de rutine formează o ierarhie de apel, adică o rutină poate invoca numai subrutine de acelaşi nivel sau mai mic.

Tip rutină
`procedure`
`operator`
`qufunct`
funcţie

QCL încorporează un limbaj de programare clasic pentru a oferi toate mijloacele necesare pentru a modifica starea programului. Însă, nu există un set nemodificabil de operatori unitari pentru a manipula starea maşinii cuantice, întrucât acesta ar necesita presupuneri referitoare la arhitectura calculatorului cuantic simulat. O rutină elementară `operator` sau `qufunct` poate fi încorporată prin declararea sa ca externă.

```
extern operator identif arg-list;
extern qufunct identif arg-list;
```

3.6.1 Proceduri

Procedurile reprezintă cel mai general tip de rutină şi sunt utilizate pentru a implementa structurile clasice de control ale algoritmilor cuantici. Procedurile permit modificări atât pentru starea program cât şi pentru starea maşină. Operaţiile exclusive pentru proceduri sunt:

- acces la variabile globale
- (pseudo) numere aleatoare prin utilizarea pseudo-funcţiei random
- operaţii neunitare pe starea maşinii prin utilizarea comenzilor reset şi measure
- intrari utilizator prin utilizarea comenzii input

Procedurile pot avea orice număr de argumente clasice şi cuantice şi pot apela toate tipurile de subrutine.

Exemplu (ruleta cuantică)

```
procedure quRoulette() {
    qureg q[5];
    int field;
    int number;
    input "Enter field number:",field;
    repeat {
        Mix(q);
        measure q,number;
        reset;
    } until number<=36;
    if field==number {
        print "Number",number,"You won!";
    } else {
        print "Number",number,"You lose.";
    }
}
```

3.6.2 Funcţii

Funcţia este cel mai restrictiv tip de rutină. Funcţiile definite de utilizator pot fi de orice tip clasic şi pot avea un număr arbitrar de parametri clasici. Valoarea funcţiei este trimisă rutinei apelante prin execuţia unei instrucţiuni `return`. Este permisă definirea

variabilelor locale la începutul funcţiei. De asemeni, în corpul unei funcţii sunt permise apeluri ale altor funcţii, precum şi autoapeluri.

Exemplu:

```
int fibonacci(int n) {  // calculeaza al n-lea termen din sirul
      int a=0;          // Fibonacci
      int b=1;
      int i;
      for i = 1 to n {
            b = a+b;
            a = b-a;
      }
      return a;
}
```

3.6.3 Operatori generali

Tipul de rutină `operator` este utilizat pentru operaţii unitare generale. Deci, acest tip de rutină este restricţionat la transformări reversibile ale stării maşină, iar comenzile reset şi measure sunt interzise.

3.6.4 Operatori pseudo-clasici

Tipul de rutină `qufunct` este utilizat pentru operatori pseudo-clasic şi funcţii cuantice, deci toate transformările trebuie să fie de forma:

$$|\Psi\rangle = \sum_i c_i|i\rangle \rightarrow \sum_{i,j} c_i\, \delta_{j\pi_i}|\Psi'\rangle$$

cu o oarecare permutare π.

3.6.4.1 Operatori condiţionali

Programele clasice permit execuţia condiţională a codului în funcţie de conţinutul unei variabile booleene. Într-un program cuantic, dacă este necesară aplicarea unei transformări asupra unui registru `a` în funcţie de conţinutul unui registru `e`, operatorul utilizat este un operator condiţional `U[[e]]` cu registrul de activare `e`. Un astfel de operator este un operator unitar de forma:

$$U_{[[e]]}: |i,\epsilon\rangle = |i\rangle|\epsilon\rangle_e \rightarrow \begin{cases} (U|i\rangle)|\epsilon\rangle_e dacă\ \epsilon = 111\ldots \\ |i\rangle|\epsilon\rangle_e altfel \end{cases}$$

Dacă registrul e are un singur qubit (qubit de activare), atunci un operator U pe *n* qubiţi poate fi aplicat condiţional asupra unui registru cuantic cuantic definind operatorul U' pe n+1 qubiţi astfel:

$$U' = \begin{pmatrix} I(n) & 0 \\ 0 & U \end{pmatrix}$$

3.6.4.2 Funcţii cuantice

O funcţie cuantică este un operator pseudo-clasic cu următoarea caracteristică:

$$F: |x\rangle_x |0\rangle_y \rightarrow |x\rangle_x |f(x)\rangle_y \; cu \; f: B^n \rightarrow B^n$$

Deoarece funcţiile cuantice sunt operatori pseudo/clasici a căror specificare este restricţionată la anumite tipuri de stări de intrare, acestea utilizează tipul `qufunct`.

3.7 Operatori elementari

3.7.1 Matrice unitare

3.7.1.1 Operatori unitari generali

Forma cea mai generală pentru a specifica un operator unitar este prin definirea elementelor matricei sale: un operator unitar U pe *n* qubiţi descrie o transformare

$$U: \mathbb{C}^{2^n} \rightarrow \mathbb{C}^{2^n}$$

corespunzătoare unei matrice 2^nx2^n în $\mathbb{C}$

$$U = \sum_{i,j}^{2^n} |i\rangle u_{ij} \langle j| = \begin{pmatrix} u_{0,0} & \cdots & u_{0,2^n-1} \\ \vdots & & \vdots \\ u_{2^n-1,0} & \cdots & u_{2^n-1,2^n-1} \end{pmatrix}$$

QCL oferă operatori externi pentru matrici unitare generale 2x2, 4x4 şi 8x8, pe care programatorul le poate utiliza pentru a implementa porţi personalizate pe 1, 2 şi 3 qubiţi.

```
extern operator Matrix2x2(
     complex u00, complex u01,
     complex u10, complex u11,
    qureg q);
extern operator Matrix4x4(. . . , qureg q);
extern operator Matrix8x8(. . . , qureg q);
```

Înainte de a fi aplicaţi, aceşti operatori sunt verificaţi dacă sunt într-adevăr unitari.

3.7.1.2 Rotaţia unui qubit

Rotaţia unui qubit este definită prin matricea de transformare U(θ):

$$U(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & \sin\frac{\theta}{2} \\ -\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}$$

Operatorul extern care implementează această transformare în QCL este `Rot`:

```
extern operator Rot(real theta, qureg q);
```

Acest operator funcţionează doar pentru regiştri pe un singur qubit.

3.7.1.3 Poarta Hadamard

```
extern operator Mix(qureg q);
```

3.7.1.4 Poarta Conditional Phase

```
extern operator CPhase(real phi, qureg q);
```

3.7.2 Operatori pseudo-clasici

3.7.2.1 Permutarea bazei

QCL oferă operatori externi pentru permutări ale bazei pe care programatorul îi poate utiliza pentru a implementa proprii operatori pseudo-clasici pe 1 până la 6 qubiţi.

```
extern qufunct Perm2(int p0, int p1, qureg q);
extern qufunct Perm4(int p0, int p1, int p2, int p3, qureg q);
extern qufunct Perm8(. . ., qureg q);
extern qufunct Perm16(. . ., qureg q);
extern qufunct Perm32(. . ., qureg q);
extern qufunct Perm64(. . ., qureg q);
```

3.7.2.2 Fanout şi Swap

Operaţia FANOUT este o funcţie cuantică pentru o clasă de transformări cu caracteristica FANOUT: $|x,y\rangle \rightarrow |x,x \oplus y\rangle$.

Operatorul Swap schimbă qubiţii a doi regiştri cu aceeaşi dimensiune.

SWAP: $|x,y\rangle \rightarrow |y,x\rangle$
Implementarea implicită a Fanout şi Swap este definită în fişierul `default.qcl`.

```
extern qufunct Fanout(qureg a, qureg b);
extern qufunct Swap(qureg a, qureg b);
```

Întrucât QCL nu impune un set specific de operatori elementari, utilizatorul este liber să-şi definească propriile implementări.

3.7.2.3 Not şi Controlled-Not

```
extern qufunct Not(qureg q);
extern qufunct CNot(qureg q,quconst c);
```

4 Implementarea în QCL a algoritmilor cuantici

4.1 Algoritmul Deutsch

4.1.1 Problema

Algoritmul lui Deutsch este cel mai simplu exemplu de algoritm cuantic şi a fost primul algoritm care a demonstrat că un computer cuantic poate rezolva o problemă specifică mai eficient decât un computer clasic.

Fie o funcţie (necunoscută) $f : \{0, 1\} \rightarrow \{0, 1\}$. Scopul algoritmului este determinarea unei proprietăţi a funcţiei f şi anume dacă aceasta este *constantă* $f(0) = f(1)$ sau *balansată* $f(0) \neq f(1)$.

În contextul informaticii cuantice, se defineşte o transformare unitară pe doi qubiţi, astfel:

$$U_f \left| x\right\rangle \left| y\right\rangle = \left| x\right\rangle \left| y \oplus f(x)\right\rangle$$

unde $\oplus$ reprezintă operaţia "sau exclusiv".

Circuitul cuantic corespunzător algoritmului lui Deutsch este următorul:

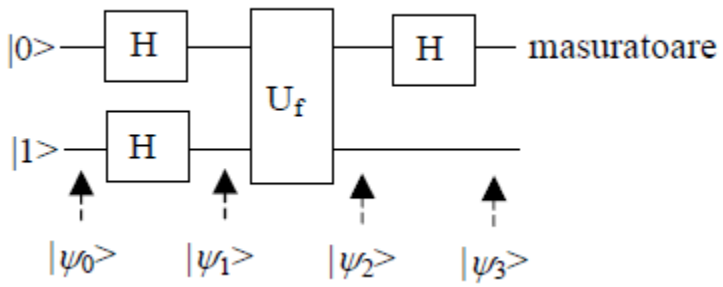

Fig. 1 Circuitul cuantic pentru algoritmul lui Deutsch

Stările sistemului sunt:

$$\left|\Psi_0\right\rangle = \left|0\right\rangle \left|1\right\rangle$$
$$|\Psi_1\rangle = \frac{1}{2}|0\rangle(|0\rangle - |1\rangle) + \frac{1}{2}|1\rangle(|0\rangle - |1\rangle)$$

$$|\Psi_2\rangle = \left(\frac{1}{\sqrt{2}}(-1)^{f(0)}|0\rangle + \frac{1}{\sqrt{2}}(-1)^{f(1)}|1\rangle\right)\left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)$$

Starea finală a primului registru este:

$$(-1)^{f(0)}|f(0) \oplus f(1)\rangle$$

Conform tabelei de adevăr a operaţiei XOR, f(0) ⊕ f(1) are valoarea 0 dacă valorile f(0) şi f(1) sunt ambele 0 sau ambele 1.
Astfel, daca in urma masuratorii primului qubit din starea finală $|\Psi_3\rangle$se obtine valoarea logica 0, functia este cu certitudine (cu probabilitate unitate) constanta. daca masuratoarea primului qubit da valoarea logica 1, functia este balansata.
Algoritmul cuantic a evaluat *f(0)*⊕ $f(1)$ într-un singur pas, fără a evalua valorile *f*(0) şi *f*(1).

4.1.2 Implementarea în QCL

Există patru funcţii posibile f:{0,1}→{0,1}

	f_1	f_2	f_3	f_4
0	0	0	1	1
1	0	1	0	1

Transformările Uf corespunzătoare acestor funcţii sunt:

$U_{f1}|0\rangle|0\rangle = |0\rangle|0 \oplus f1(0)\rangle = |0\rangle|0 \oplus 0\rangle = |0\rangle|0\rangle$
$U_{f1}|0\rangle|1\rangle = |0\rangle|1 \oplus f1(0)\rangle = |0\rangle|1 \oplus 0\rangle = |0\rangle|1\rangle$
$U_{f1}|1\rangle|0\rangle = |1\rangle|0 \oplus f1(1)\rangle = |1\rangle|0 \oplus 0\rangle = |1\rangle|0\rangle$
$U_{f1}|1\rangle|1\rangle = |1\rangle|1 \oplus f1(1)\rangle = |1\rangle|1 \oplus 0\rangle = |1\rangle|1\rangle$

$$U_{f1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = I$$

$U_{f2}|0\rangle|0\rangle = |0\rangle|0 \oplus f2(0)\rangle = |0\rangle|0 \oplus 0\rangle = |0\rangle|0\rangle$
$U_{f2}|0\rangle|1\rangle = |0\rangle|1 \oplus f2(0)\rangle = |0\rangle|1 \oplus 0\rangle = |0\rangle|1\rangle$
$U_{f2}|1\rangle|0\rangle = |1\rangle|0 \oplus f2(1)\rangle = |1\rangle|0 \oplus 1\rangle = |1\rangle|1\rangle$
$U_{f2}|1\rangle|1\rangle = |1\rangle|1 \oplus f2(1)\rangle = |1\rangle|1 \oplus 1\rangle = |1\rangle|0\rangle$

$$U_{f2} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = CNOT$$

$U_{f3}|0\rangle|0\rangle = |0\rangle|0 \oplus f3(0)\rangle = |0\rangle|0 \oplus 1\rangle = |0\rangle|1\rangle$
$U_{f3}|0\rangle|1\rangle = |0\rangle|1 \oplus f3(0)\rangle = |0\rangle|1 \oplus 1\rangle = |0\rangle|0\rangle$
$U_{f3}|1\rangle|0\rangle = |1\rangle|0 \oplus f3(1)\rangle = |1\rangle|0 \oplus 0\rangle = |1\rangle|0\rangle$
$U_{f3}|1\rangle|1\rangle = |1\rangle|1 \oplus f3(1)\rangle = |1\rangle|1 \oplus 0\rangle = |1\rangle|1\rangle$

$$U_{f3} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Transformarea U_{f3} poate fi realizată utilizând porţi NOT şi CNOT astfel:

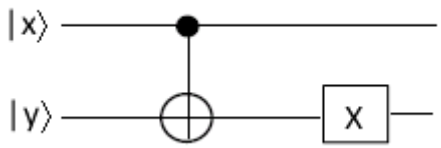

Fig. 2 Transformarea Uf3

$U_{f3}(x,y) = CNOT(x,y)\ NOT(y)$

$U_{f4}|0\rangle|0\rangle = |0\rangle|0 \oplus f4(0)\rangle = |0\rangle|0 \oplus 1\rangle = |0\rangle|1\rangle$
$U_{f4}|0\rangle|1\rangle = |0\rangle|1 \oplus f4(0)\rangle = |0\rangle|1 \oplus 1\rangle = |0\rangle|0\rangle$
$U_{f4}|1\rangle|0\rangle = |1\rangle|0 \oplus f4(1)\rangle = |1\rangle|0 \oplus 1\rangle = |1\rangle|1\rangle$
$U_{f4}|1\rangle|1\rangle = |1\rangle|1 \oplus f4(1)\rangle = |1\rangle|1 \oplus 1\rangle = |1\rangle|0\rangle$

$$U_{f4} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Implementarea algoritmului lui Deutsch în QCL este următoarea:

```
//se defineste transformarea Uf
//n reprezinta numarul functiei, conform tabelului
qufunct Uf(quvoid x, quvoid y, int n) {
```

```
  if (n==2) or (n==3) {
    CNot(y, x);
  }
  if (n==3) or (n==4) {
    X(y);
  }
}

procedure deutsch(int n) {
  //se declară cei doi regiştri
  qureg x[1];
  qureg y[1];
  int m;

  // se trece registrul y din starea |0⟩ in starea |1⟩
  Not(y);
  //se aplica operatorul Hadamard
  H(x);
  H(y);
  //se aplica transformarea Uf
  Uf(x,y,n);
  //se aplica operatorul Hadamard
  H(x);
  //se masoara registrul x
  measure x,m;
  print "f(0) xor f(1) = ", m;
  if m==0 {
    print "Functia este constanta";
  }
  else {
    print "Functia este balansata";
  }
}
```

4.2 Algoritmul Deutsch-Jozsa

4.2.1 Problema

Algoritmul Deutsch-Jozsa este generalizarea algoritmului Deutsch pentru cazul mai multor qubiţi. Fie funcţia booleană f: $\{0,1\}^n \rightarrow \{0,1\}$ unde n este un întreg pozitiv arbitrar. Asemeni algoritmului lui Deutsch, se urmăreşte dacă funcţia este constantă (adică fie f(x) = 0 pt toţi x $\in\{0,1\}^n$ fie f(x) = 1 pt toţi x $\in\{0,1\}^n$) sau balansată, adică numărul de intrări x $\in\{0,1\}^n$ pentru care funcţia ia valoarea 0 şi 1 este acelaşi:

$$|\{x \in \{0,1\}^n : f(x) = 0\}| = |\{x \in \{0,1\}^n : f(x) = 1\}| = 2^{n-1}$$

Se utilizează o transformare unitară U_f definită similar celei anterioare:

$$U_f \left| x\right\rangle \left| y\right\rangle = \left| x\right\rangle \left| y \oplus f(x)\right\rangle$$

pentru toţi x $\in\{0,1\}^n$ şi y $\in\{0,1\}$

Algoritmul cuantic este implementat de circuitul cuantic următor:

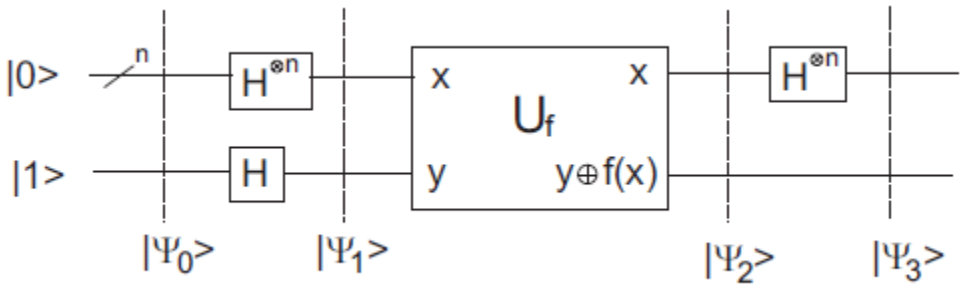

Fig. 3 Circuitul cuantic pentru algoritmul Deutsch-Jozsa

Stările sistemului sunt:

$$|\Psi_0\rangle = |0\rangle_n|1\rangle$$

$$|\Psi_1\rangle = H^{\otimes n}|0\rangle_n \otimes H|1\rangle = \left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)\cdots\left(\frac{|0\rangle+|1\rangle}{\sqrt{2}}\right)\left(\frac{|0\rangle-|1\rangle}{\sqrt{2}}\right)$$

$$|\Psi_1\rangle = \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}|x\rangle\left(\frac{1}{\sqrt{2}}|0\rangle-\frac{1}{\sqrt{2}}|1\rangle\right)$$

$$|\Psi_2\rangle = \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}|x\rangle\left(\frac{1}{\sqrt{2}}|0\rangle-\frac{1}{\sqrt{2}}|1\rangle\right)$$

$$|\Psi_3\rangle = \frac{1}{\sqrt{2^n}}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}\left(\frac{1}{\sqrt{2^n}}\sum_{y\in\{0,1\}^n}(-1)^{x\cdot y}|y\rangle\right)$$

$$|\Psi_3\rangle = \sum_{y\in\{0,1\}^n}\left(\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(x)+x\cdot y}\right)|y\rangle$$

Amplitudinea asociată cu starea clasică $|0^n\rangle$ este:

$$\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}$$

Aşadar, daca funcţia f este *constantă*, toţi cei 2^n termeni ce formează suma vor avea acelaşi semn ("+" pentru $f(x) = 0$ si "-" pentru $f(x) =1$, astfel că

$$\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(x)} = \pm 1$$

Dacă funcţia este balansata, exact jumatate din termenii ce formeaza suma au semnul "+" si cealalta jumatate a termenilor semnul "-", ceea ce înseamnă că

$$\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(x)} = 0$$

Probabilitatea ca rezultatul măsurătorii să fie $|0\rangle_n$ este:

$$\left|\frac{1}{2^n}\sum_{x\in\{0,1\}^n}(-1)^{f(x)}\right|^2 = \begin{cases}1 \text{ dacă } f \text{ este constantă}\\ 0 \text{ dacă } f \text{ este balansată}\end{cases}$$

4.2.2 Implementare utilizând porţi CNOT

Implementarea algoritmului Deutsch-Jozsa este prezentată mai jos. Pentru realizarea transformării Uf s-au utilizat porți CNOT.

```
procedure deutsch (int n) {
  //se declara cei doi registri
  qureg x[n];
  qureg y[1];
  int m;

  reset;
  // se trece registrul y din starea |0⟩ in starea |1⟩
  Not (x[N]);
  //se aplica operatorul Hadamard
  H(x);
  //se aplica operatorul Uf – simulat prin poarta CNOT
  CNot (y, x[0]);
  //se aplica operatorul Hadamard
  H(x) ;
  //se masoara registrul x
  measure x, m;
  if m == 0 {
    print " Functia este constanta";
  } else {
```

```
    print " Functia este balansata";
  }
}
```

4.3 Algoritmul Bernstein-Vazirani

4.3.1 Definirea problemei

Fie o funcţie $f:\{0,1\}^n \rightarrow \{0,1\}$ de forma

$$f(x) = a \cdot x = a_{n-1}x_{n-1} \oplus a_{n-2}x_{n-2} \oplus \ldots \oplus a_0 x_0$$

unde *a* este un n întreg non-negativ necunoscut.
Funcţia *f* calculează de fapt produsul scalar binar dintre numărul *a* şi un alt astfel de număr *x*. Problema Bernstein-Vazirani constă în determinarea tuturor biţilor lui *a*.

Algoritmul Bernstein-Vazirani este implementat de următorul circuit cuantic:

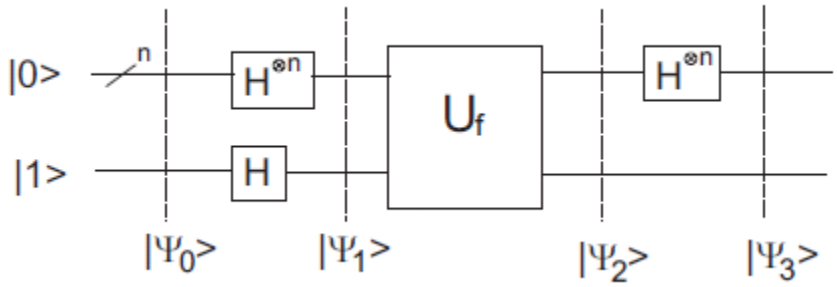

Fig. 4 Circuitul cuantic pentru algoritmul Bernstein-Vazirani

Similar algoritmului Deutsch-Jozsa, stările sistemului sunt:

$$|\Psi_0\rangle = |0\rangle_n |1\rangle$$

Acestei stări se aplică cele n + 1 porţi Hadamard paralele.

$$|\Psi_1\rangle = H^{\otimes n}|0\rangle_n \otimes H|1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)$$

$$|\Psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{f(x)}|x\rangle \left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)$$

$$|\Psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} (-1)^{a\cdot x}|x\rangle \left(\frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle\right)$$

Se aplică cele n transformări Hadamard. Starea rezultată este:

$$|\Psi_3\rangle = \frac{1}{2^n} \sum_{y\in\{0,1\}^n} \left(\sum_{x\in\{0,1\}^n} (-1)^{a\cdot x + x\cdot y} \right) |y\rangle$$

Considerăm suma peste *x*:

$$\sum_{x=0}^{2^n-1} (-1)^{a\cdot x}(-1)^{x\cdot y} = \sum_{x_1,\ldots,x_n=0}^{1} (-1)^{a_1x_1+\ldots+a_nx_n+x_1y_1+\cdots+x_ny_n}$$

$$= \sum_{x_1=0}^{1} \ldots \sum_{x_n=0}^{1} (-1)^{a_1x_1+\ldots+a_nx_n+x_1y_1+\cdots+x_ny_n}$$

$$= ((-1)^0 + (-1)^{a_1+y_1}) \sum_{x_2=0}^{1} \ldots \sum_{x_n=0}^{1} (-1)^{a_2x_2+\ldots+a_nx_n+x_2y_2+\cdots+x_ny_n}$$

$$= 2^n \delta_{a_1,y_1}\delta_{a_2,y_2}\cdots\delta_{a_n,y_n} = 2^n\delta_{a,y}$$

$$|\Psi_3\rangle = \frac{1}{2^n} \sum_{y\in\{0,1\}^n} \left(2^n\delta_{a,y}\right)|y\rangle = |a\rangle$$

Toţi cei n biţi ai lui a pot fi determinaţi prin măsurarea primului registru.

În figura următoare este prezentat circuitul cuantic pentru cazul în care *n* = 5 şi *a* = 19 = 10011. Acţiunea operatorului *Uf* este reprodusă prin intermediul unui set de porţi CNOT .

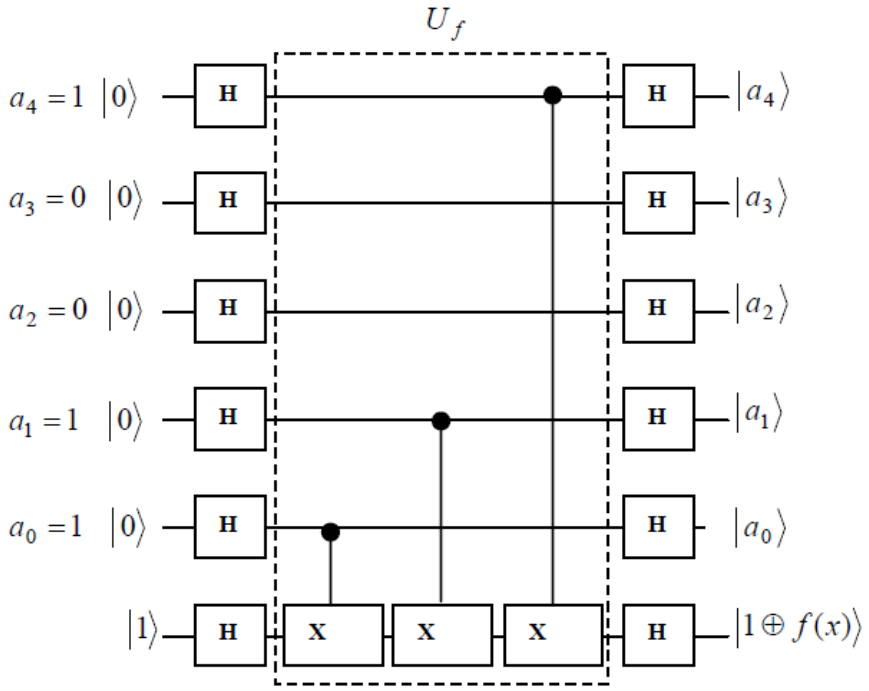

Fig. 5 Circuitul cuantic pentru n=5

4.3.2 Implementarea algoritmului în QCL

Algoritmul Bernstein-Vazirani pentru un registru cu 5 qubiţi

```
qureg x[5]; //se declara un registru cu 5 qubiti
qureg y[1]; //se declara un registru cu un qubit
int m; int i;
int vector bitv[n];
for i=0 to #x-1 {
   if random()>0.5 {bitv[i]=1;}
}

int a=0;
for i=0 to #x-1 {
   a = a + bitv[i]*2^i;
}
print "Valoarea aleasa pentru a este ", a;
// se trece registrul y din starea |0⟩ in starea |1⟩
X(y);
//se concateneaza registrii si se aplica operatorul Hadamard
H(x&y);
//se simuleaza actiunea portii Uf cu porti CNOT
for i=0 to #x-1{
   if (bitv[i]>0){ CNot(y,x[i]);}
}
//se concateneaza registrii si se aplica operatorul Hadamard
H(x&y);
```

```
//se masoara registrul x
measure x, m;
print "Valoarea determinata pentru a este ", m;
```

4.4 Algoritmul Simon

4.4.1 Definirea problemei

Fie funcţia f : $\{0; 1\}^n \rightarrow \{0; 1\}^n$, unde n este un întreg pozitiv. Similar problemelor precedente, accesul la funcţia f este restricţionat la interogări către o cutie neagră ce corespunde transformării unitare U_f. Exista un sir s ($s \in \{0,1\}^n$), astfel încât f(x) = f(y) $\Leftrightarrow$ y = x $\oplus$ s, pentru orice x; y $\in \{0; 1\}^n$. Problema consta în determinarea lui s.

Simon, propune un algoritm care pornind de la un registru $|x\rangle$ poate calcula eficient $|x\rangle|f(x)\rangle$ într-un numar de n pasi.

Intr-un circuit cuantic, se foloseşte o varianta a algoritmului Deutsch-Jozsa in care ambii registri contin *n* qubiti; reprezentarea schematica a algoritmului Simon este următoarea:

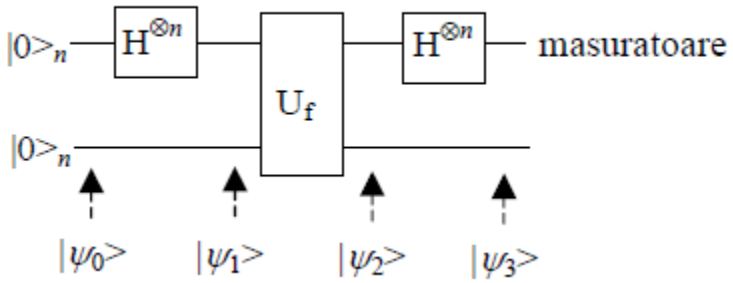

Fig. 6 Circuitul cuantic pentru algoritmul Simon

Stările sistemului sunt:

$$|\Psi_0\rangle = |0\rangle_n |0\rangle_n$$

$$|\Psi_1\rangle = \left(H^{\otimes n} \otimes I_{2^n}\right)|0\rangle_n|0\rangle_n = \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle_n|0\rangle_n$$

unde I_{2^n} este matricea unitate de dimensiune2^n.

$$|\Psi_2\rangle = U_f\left(\frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}|x\rangle_n|0\rangle_n\right) = \frac{1}{2^{n/2}}\sum_{x=0}^{2^n-1}|x\rangle_n|f(x)\rangle_n$$

Dupa ultima poarta Hadamard aplicata primului registru, se obţine:

$$|\Psi_3\rangle = 2^{-n}\sum_{y=0}^{2^n-1}\sum_{x=0}^{2^n-1}(-1)^{x\cdot y}|y\rangle_n|f(x)\rangle_n$$
$$= 2^{-(n+1)}\sum_{x,y=0}^{2^n-1}\left[(-1)^{x\cdot y}+(-1)^{(x\oplus a)\cdot y}\right]|y\rangle_n|f(x)\rangle_n$$

Se măsoară primul registru. Daca s·*y*=1 termenii din coeficientul lui $|y\rangle$ interferă distructiv, doar starile $|y\rangle$ cu s·*y*=0 ramanand in suma peste *y*. Rezultatul masuratorii este deci selectat aleator dintre toate valorile posibile *y* ce satisfac s·*y*=0, fiecare valoare avand probabililatea $2^{-(n+1)}$. Ruland algoritmul de mai multe ori, de fiecare data se obtine o alta valoare *y* care satisface s·*y*=0. Odata ce am gasit *n-1* astfel de valori independente se pot rezolva equatiile s·y_i=0, *i* = 1,.., *n* pentru a determina valoarea unica a lui s (care are *n* biti).

4.4.2 Algoritmul lui Simon pentru cazul n=2

Fie funcţia $f: \{0,1\}^2 \rightarrow \{0,1\}^2$ definită astfel:

```
f(00) = f(10) = 00
f(01) = f(11) = 01
```

Se observă că

```
f(00) = f(10) ↔ 10 = 00 + 10
f(01) = f(11) ↔ 11 = 01 + 10
```

Deci şirul s este 10.

4.4.2.1 Definirea transformării Uf

$U_f|00\rangle|00\rangle = |00\rangle|00 \oplus f(00)\rangle = |00\rangle|00 \oplus 00\rangle = |00\rangle|00\rangle$
$U_f|00\rangle|01\rangle = |00\rangle|01 \oplus f(00)\rangle = |00\rangle|01 \oplus 00\rangle = |00\rangle|01\rangle$
$U_f|00\rangle|10\rangle = |00\rangle|10 \oplus f(00)\rangle = |00\rangle|10 \oplus 00\rangle = |00\rangle|10\rangle$
$U_f|00\rangle|11\rangle = |00\rangle|11 \oplus f(00)\rangle = |00\rangle|11 \oplus 00\rangle = |00\rangle|11\rangle$

$U_f|01\rangle|00\rangle = |01\rangle|00 \oplus f(01)\rangle = |01\rangle|00 \oplus 01\rangle = |01\rangle|01\rangle$
$U_f|01\rangle|01\rangle = |01\rangle|01 \oplus f(01)\rangle = |01\rangle|01 \oplus 01\rangle = |01\rangle|00\rangle$

$U_f|01\rangle|10\rangle = |01\rangle|10 \oplus f(01)\rangle = |01\rangle|10 \oplus 01\rangle = |01\rangle|11\rangle$
$U_f|01\rangle|11\rangle = |01\rangle|11 \oplus f(01)\rangle = |01\rangle|11 \oplus 01\rangle = |01\rangle|10\rangle$

$U_f|10\rangle|00\rangle = |10\rangle|00 \oplus f(10)\rangle = |10\rangle|00 \oplus 00\rangle = |10\rangle|00\rangle$
$U_f|10\rangle|01\rangle = |10\rangle|01 \oplus f(10)\rangle = |10\rangle|01 \oplus 00\rangle = |10\rangle|01\rangle$
$U_f|10\rangle|10\rangle = |10\rangle|10 \oplus f(10)\rangle = |10\rangle|10 \oplus 00\rangle = |10\rangle|10\rangle$
$U_f|10\rangle|11\rangle = |10\rangle|11 \oplus f(10)\rangle = |10\rangle|11 \oplus 00\rangle = |10\rangle|11\rangle$

$U_f|11\rangle|00\rangle = |11\rangle|00 \oplus f(11)\rangle = |11\rangle|00 \oplus 01\rangle = |11\rangle|01\rangle$
$U_f|11\rangle|01\rangle = |11\rangle|01 \oplus f(11)\rangle = |11\rangle|01 \oplus 01\rangle = |11\rangle|00\rangle$
$U_f|11\rangle|10\rangle = |11\rangle|10 \oplus f(11)\rangle = |11\rangle|10 \oplus 01\rangle = |11\rangle|11\rangle$
$U_f|11\rangle|11\rangle = |11\rangle|11 \oplus f(11)\rangle = |11\rangle|11 \oplus 01\rangle = |11\rangle|10\rangle$

$$U_f = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

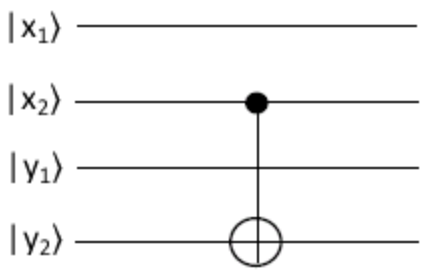

Fig. 7 Transformarea Uf in cazul n=2

4.4.2.2 Implementare în QCL

```
//se declara cei doi registri cu cate doi qubiti
qureg x[2]; qureg y[2];
int m;

{
reset;
//se aplica transformarea Hadamard
H(x);
//se aplica transformarea Uf
CNot(y[0], x[0]);
//se aplica transformarea Hadamard
H(x);
//se masoara registrul x
measure x,m;
print "Valoarea determinata este: ", m;
} until (m!=0);
```

4.4.3 Algoritmul lui Simon pentru cazul n=3

Fie f:$\{0,1\}^3 \rightarrow \{0,1\}^3$ definită astfel:

```
f(000) = 100
f(001) = 010
f(010) = 000
f(011) = 110
f(100) = 000
f(101) = 110
f(110) = 100
f(111) = 010

f(000) = f(110) ↔ 110 = 000 + 110
f(001) = f(111) ↔ 111 = 001 + 110
f(010) = f(100) ↔ 100 = 010 + 110
f(011) = f(101) ↔ 101 = 011 + 110
```

Deci şirul s este 110.

4.4.3.1 Definirea transformării Uf

$U_f|000\rangle|000\rangle = |000\rangle|000 \oplus f(000)\rangle = |000\rangle|000 \oplus 100\rangle = |000\rangle|100\rangle$
$U_f|000\rangle|001\rangle = |000\rangle|001 \oplus f(000)\rangle = |000\rangle|001 \oplus 100\rangle = |000\rangle|101\rangle$
$U_f|000\rangle|010\rangle = |000\rangle|010 \oplus f(000)\rangle = |000\rangle|010 \oplus 100\rangle = |000\rangle|110\rangle$
$U_f|000\rangle|011\rangle = |000\rangle|011 \oplus f(000)\rangle = |000\rangle|011 \oplus 100\rangle = |000\rangle|111\rangle$
$U_f|000\rangle|100\rangle = |000\rangle|100 \oplus f(000)\rangle = |000\rangle|100 \oplus 100\rangle = |000\rangle|000\rangle$
$U_f|000\rangle|101\rangle = |000\rangle|101 \oplus f(000)\rangle = |000\rangle|101 \oplus 100\rangle = |000\rangle|001\rangle$

$U_f|000\rangle|110\rangle = |000\rangle|110 \oplus f(000)\rangle = |000\rangle|110 \oplus 100\rangle = |000\rangle|010\rangle$
$U_f|000\rangle|111\rangle = |000\rangle|111 \oplus f(000)\rangle = |000\rangle|111 \oplus 100\rangle = |000\rangle|011\rangle$

$U_f|001\rangle|000\rangle = |001\rangle|000 \oplus f(001)\rangle = |001\rangle|000 \oplus 010\rangle = |001\rangle|010\rangle$
$U_f|001\rangle|001\rangle = |001\rangle|001 \oplus f(001)\rangle = |001\rangle|001 \oplus 010\rangle = |001\rangle|011\rangle$
$U_f|001\rangle|010\rangle = |001\rangle|010 \oplus f(001)\rangle = |001\rangle|010 \oplus 010\rangle = |001\rangle|000\rangle$
$U_f|001\rangle|011\rangle = |001\rangle|011 \oplus f(001)\rangle = |001\rangle|011 \oplus 010\rangle = |001\rangle|001\rangle$
$U_f|001\rangle|100\rangle = |001\rangle|100 \oplus f(001)\rangle = |001\rangle|100 \oplus 010\rangle = |001\rangle|110\rangle$
$U_f|001\rangle|101\rangle = |001\rangle|101 \oplus f(001)\rangle = |001\rangle|101 \oplus 010\rangle = |001\rangle|111\rangle$
$U_f|001\rangle|110\rangle = |001\rangle|110 \oplus f(001)\rangle = |001\rangle|110 \oplus 010\rangle = |001\rangle|100\rangle$
$U_f|001\rangle|111\rangle = |001\rangle|111 \oplus f(001)\rangle = |001\rangle|111 \oplus 010\rangle = |001\rangle|101\rangle$

$U_f|010\rangle|000\rangle = |010\rangle|000 \oplus f(010)\rangle = |010\rangle|000 \oplus 000\rangle = |010\rangle|000\rangle$
$U_f|010\rangle|001\rangle = |010\rangle|001 \oplus f(010)\rangle = |010\rangle|001 \oplus 000\rangle = |010\rangle|001\rangle$
$U_f|010\rangle|010\rangle = |010\rangle|010 \oplus f(010)\rangle = |010\rangle|010 \oplus 000\rangle = |010\rangle|010\rangle$
$U_f|010\rangle|011\rangle = |010\rangle|011 \oplus f(010)\rangle = |010\rangle|011 \oplus 000\rangle = |010\rangle|011\rangle$
$U_f|010\rangle|100\rangle = |010\rangle|100 \oplus f(010)\rangle = |010\rangle|100 \oplus 000\rangle = |010\rangle|100\rangle$
$U_f|010\rangle|101\rangle = |010\rangle|101 \oplus f(010)\rangle = |010\rangle|101 \oplus 000\rangle = |010\rangle|101\rangle$
$U_f|010\rangle|110\rangle = |010\rangle|110 \oplus f(010)\rangle = |010\rangle|110 \oplus 000\rangle = |010\rangle|110\rangle$
$U_f|010\rangle|111\rangle = |010\rangle|111 \oplus f(010)\rangle = |010\rangle|111 \oplus 000\rangle = |010\rangle|111\rangle$

$U_f|011\rangle|000\rangle = |011\rangle|000 \oplus f(011)\rangle = |011\rangle|000 \oplus 110\rangle = |011\rangle|110\rangle$
$U_f|011\rangle|001\rangle = |011\rangle|001 \oplus f(011)\rangle = |011\rangle|001 \oplus 110\rangle = |011\rangle|111\rangle$
$U_f|011\rangle|010\rangle = |011\rangle|010 \oplus f(011)\rangle = |011\rangle|010 \oplus 110\rangle = |011\rangle|100\rangle$
$U_f|011\rangle|011\rangle = |011\rangle|011 \oplus f(011)\rangle = |011\rangle|011 \oplus 110\rangle = |011\rangle|101\rangle$
$U_f|011\rangle|100\rangle = |011\rangle|100 \oplus f(011)\rangle = |011\rangle|100 \oplus 110\rangle = |011\rangle|010\rangle$
$U_f|011\rangle|101\rangle = |011\rangle|101 \oplus f(011)\rangle = |011\rangle|101 \oplus 110\rangle = |011\rangle|011\rangle$
$U_f|011\rangle|110\rangle = |011\rangle|110 \oplus f(011)\rangle = |011\rangle|110 \oplus 110\rangle = |011\rangle|000\rangle$
$U_f|011\rangle|111\rangle = |011\rangle|111 \oplus f(011)\rangle = |011\rangle|111 \oplus 110\rangle = |011\rangle|001\rangle$

$U_f|100\rangle|000\rangle = |100\rangle|000 \oplus f(100)\rangle = |100\rangle|000 \oplus 000\rangle = |100\rangle|000\rangle$
$U_f|100\rangle|001\rangle = |100\rangle|001 \oplus f(100)\rangle = |100\rangle|001 \oplus 000\rangle = |100\rangle|001\rangle$
$U_f|100\rangle|010\rangle = |100\rangle|010 \oplus f(100)\rangle = |100\rangle|010 \oplus 000\rangle = |100\rangle|010\rangle$
$U_f|100\rangle|011\rangle = |100\rangle|011 \oplus f(100)\rangle = |100\rangle|011 \oplus 000\rangle = |100\rangle|011\rangle$
$U_f|100\rangle|100\rangle = |100\rangle|100 \oplus f(100)\rangle = |100\rangle|100 \oplus 000\rangle = |100\rangle|100\rangle$
$U_f|100\rangle|101\rangle = |100\rangle|101 \oplus f(100)\rangle = |100\rangle|101 \oplus 000\rangle = |100\rangle|101\rangle$
$U_f|100\rangle|110\rangle = |100\rangle|110 \oplus f(100)\rangle = |100\rangle|110 \oplus 000\rangle = |100\rangle|110\rangle$
$U_f|100\rangle|111\rangle = |100\rangle|111 \oplus f(100)\rangle = |100\rangle|111 \oplus 000\rangle = |100\rangle|111\rangle$

$U_f|101\rangle|000\rangle = |101\rangle|000 \oplus f(101)\rangle = |101\rangle|000 \oplus 110\rangle = |100\rangle|110\rangle$
$U_f|101\rangle|001\rangle = |101\rangle|001 \oplus f(101)\rangle = |101\rangle|001 \oplus 110\rangle = |100\rangle|111\rangle$
$U_f|101\rangle|010\rangle = |101\rangle|010 \oplus f(101)\rangle = |101\rangle|010 \oplus 110\rangle = |100\rangle|100\rangle$

$U_f|101\rangle|011\rangle = |101\rangle|011 \oplus f(101)\rangle = |101\rangle|011 \oplus 110\rangle = |100\rangle|101\rangle$
$U_f|101\rangle|100\rangle = |101\rangle|100 \oplus f(101)\rangle = |101\rangle|100 \oplus 110\rangle = |100\rangle|010\rangle$
$U_f|101\rangle|101\rangle = |101\rangle|101 \oplus f(101)\rangle = |101\rangle|101 \oplus 110\rangle = |100\rangle|011\rangle$
$U_f|101\rangle|110\rangle = |101\rangle|110 \oplus f(101)\rangle = |101\rangle|110 \oplus 110\rangle = |100\rangle|000\rangle$
$U_f|101\rangle|111\rangle = |101\rangle|111 \oplus f(101)\rangle = |101\rangle|111 \oplus 110\rangle = |100\rangle|001\rangle$
$U_f|110\rangle|000\rangle = |110\rangle|000 \oplus f(110)\rangle = |110\rangle|000 \oplus 100\rangle = |100\rangle|100\rangle$
$U_f|110\rangle|001\rangle = |110\rangle|001 \oplus f(110)\rangle = |110\rangle|001 \oplus 100\rangle = |100\rangle|101\rangle$
$U_f|110\rangle|010\rangle = |110\rangle|010 \oplus f(110)\rangle = |110\rangle|010 \oplus 100\rangle = |100\rangle|110\rangle$
$U_f|110\rangle|011\rangle = |110\rangle|011 \oplus f(110)\rangle = |110\rangle|011 \oplus 100\rangle = |100\rangle|111\rangle$
$U_f|110\rangle|100\rangle = |110\rangle|100 \oplus f(110)\rangle = |110\rangle|100 \oplus 100\rangle = |100\rangle|000\rangle$
$U_f|110\rangle|101\rangle = |110\rangle|101 \oplus f(110)\rangle = |110\rangle|101 \oplus 100\rangle = |100\rangle|001\rangle$
$U_f|110\rangle|110\rangle = |110\rangle|110 \oplus f(110)\rangle = |110\rangle|110 \oplus 100\rangle = |100\rangle|010\rangle$
$U_f|110\rangle|111\rangle = |110\rangle|111 \oplus f(110)\rangle = |110\rangle|111 \oplus 100\rangle = |100\rangle|011\rangle$

$U_f|111\rangle|000\rangle = |111\rangle|000 \oplus f(111)\rangle = |111\rangle|000 \oplus 010\rangle = |111\rangle|010\rangle$
$U_f|111\rangle|001\rangle = |111\rangle|001 \oplus f(111)\rangle = |111\rangle|001 \oplus 010\rangle = |111\rangle|011\rangle$
$U_f|111\rangle|010\rangle = |111\rangle|010 \oplus f(111)\rangle = |111\rangle|010 \oplus 010\rangle = |111\rangle|000\rangle$
$U_f|111\rangle|011\rangle = |111\rangle|011 \oplus f(111)\rangle = |111\rangle|011 \oplus 010\rangle = |111\rangle|001\rangle$
$U_f|111\rangle|100\rangle = |111\rangle|100 \oplus f(111)\rangle = |111\rangle|100 \oplus 010\rangle = |111\rangle|110\rangle$
$U_f|111\rangle|101\rangle = |111\rangle|101 \oplus f(111)\rangle = |111\rangle|101 \oplus 010\rangle = |111\rangle|111\rangle$
$U_f|111\rangle|110\rangle = |111\rangle|110 \oplus f(111)\rangle = |111\rangle|110 \oplus 010\rangle = |111\rangle|100\rangle$
$U_f|111\rangle|111\rangle = |111\rangle|111 \oplus f(111)\rangle = |111\rangle|111 \oplus 010\rangle = |111\rangle|101\rangle$

Transformarea U_f este următoarea:

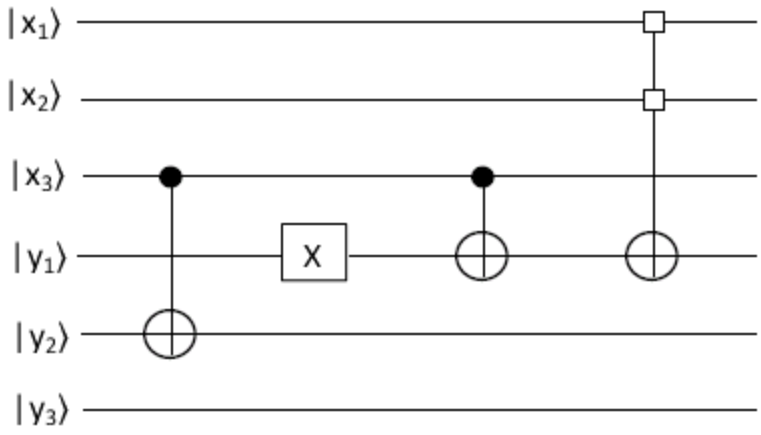

Fig. 8 Transformarea Uf in cazul n=3

S-a definit o poartă B care acţionează pe trei qubiţi a,b,c şi schimbă starea celui de-al treilea qubit (c) dacă dintre primii doi qubiţi este setat unul și numai unul (dacă a XOR b):

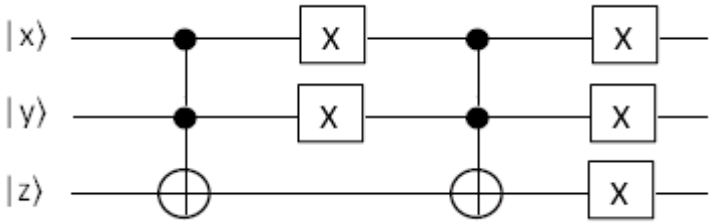

Fig. 9 Poarta B

4.4.3.2 Implementare în QCL

```
//poarta CCNOT
operator CCNot(qureg x, qureg y, qureg z)
{
  int i;
  for i=0 to #x-1
  {
    if (x[i]==1 and y[i]==1) {Not(z[i]);}
  }
}

//poarta B
operator B(qureg x, qureg y, qureg z)
{
  CCNot(x,y,z);
  Not(x);
  Not(y);
  CCNot(x,y,z);
  Not(x);
  Not(y);
  Not(z);
}

procedure simon()
{
  qureg x[3]; qureg y[3];
  int m1; int m2;
  {
  reset;
  //se aplica transformarea Hadamard
  H(x);
  //se aplica transformarea Uf
  CNot(y[1],x[0]);
  Not(y[2]);
  CNot(y[2],x[0]);
  B(x[2],x[1],y[2]);
  //se aplica transformarea Hadamard
  H(x);
  //se masoara registrul x
  measure x,m1;
```

```
  print "Prima valoare determinata este: ", m1;
  } until (m1!=0);

  //se reia algoritmul
  //pentru a obtine a doua ecuatie
  {
    reset;
    //se aplica transformarea Hadamard
    H(x);
    //se aplica transformarea Uf
    CNot(y[1],x[0]);
    Not(y[2]);
    CNot(y[2],x[0]);
    B(x[2],x[1],y[2]);
    //se aplica transformarea Hadamard
    H(x);
    //se masoara registrul x
    measure x,m2;
    print "A doua valoare determinata este: ", m2;
  } until ((m2!=0) and (m2!=m1));
}
```

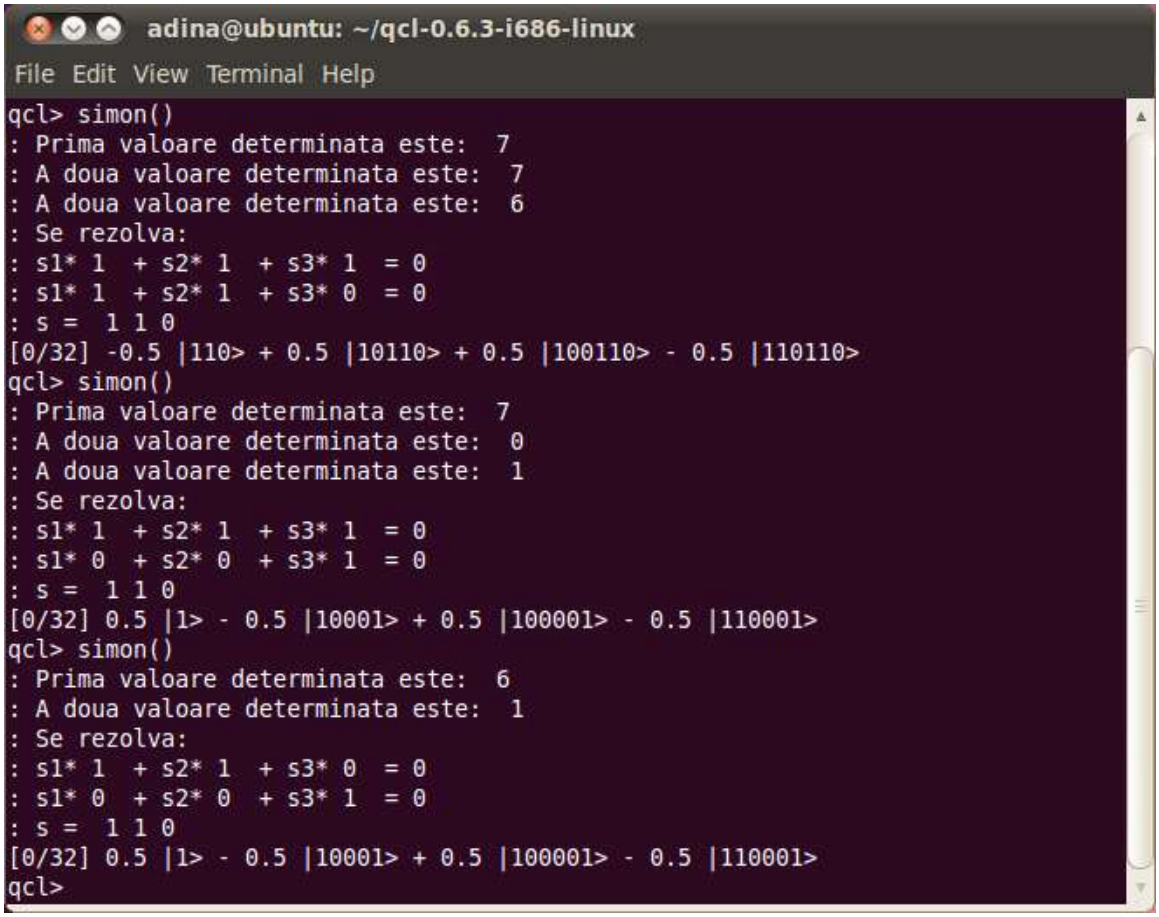
adina@ubuntu: ~/qcl-0.6.3-i686-linux
File Edit View Terminal Help
qcl> simon()
: Prima valoare determinata este: 7
: A doua valoare determinata este: 7
: A doua valoare determinata este: 6
: Se rezolva:
: s1* 1 + s2* 1 + s3* 1 = 0
: s1* 1 + s2* 1 + s3* 0 = 0
: s = 1 1 0
[0/32] -0.5 |110> + 0.5 |10110> + 0.5 |100110> - 0.5 |110110>
qcl> simon()
: Prima valoare determinata este: 7
: A doua valoare determinata este: 0
: A doua valoare determinata este: 1
: Se rezolva:
: s1* 1 + s2* 1 + s3* 1 = 0
: s1* 0 + s2* 0 + s3* 1 = 0
: s = 1 1 0
[0/32] 0.5 |1> - 0.5 |10001> + 0.5 |100001> - 0.5 |110001>
qcl> simon()
: Prima valoare determinata este: 6
: A doua valoare determinata este: 1
: Se rezolva:
: s1* 1 + s2* 1 + s3* 0 = 0
: s1* 0 + s2* 0 + s3* 1 = 0
: s = 1 1 0
[0/32] 0.5 |1> - 0.5 |10001> + 0.5 |100001> - 0.5 |110001>
qcl>
```

**Fig. 10 Rezultatul executiei algoritmului lui Simon**
```

4.5 Algoritmul lui Grover

4.5.1 Problema

Fie un sistem cu $N=2^n$ stări notate S_1, S_2, . . ., S_N. Aceste 2^n stări sunt reprezentate prin şiruri de n biţi. Fie o stare unică, S_v, care satisface condiţia C(Sv)=1, în timp ce pentru toate celelalte stări S, C(S)=0 (se presupune că pentru orice stare S, condiţia C(S) poate fi evaluată în timp constant). Se cere identificarea stării Sv.

Altfel spus, se cere să se găsească obiectul care satisface o anumita cerinţă dintr-o listă nesortată de *N* obiecte. Fără a pierde din generalitate se poate presupune că S = {0, 1, 2, . . . , N-1}. Cerinţa pe care trebuie să o îndeplinească obiectul este reprezentată printr-o funcţie dată sub forma unei cutii negre (black box). În cazul clasic, căutarea într-o listă neordonată de N elemente nu poate fi făcută mai rapid de un timp liniar O(N). Grover a prezentat un algoritm care poate găsi elementul căutat în $O(N^{1/2})$ paşi.

Circuitul care implementează algoritmul de căutare Grover este reprezentat în figura următoare.

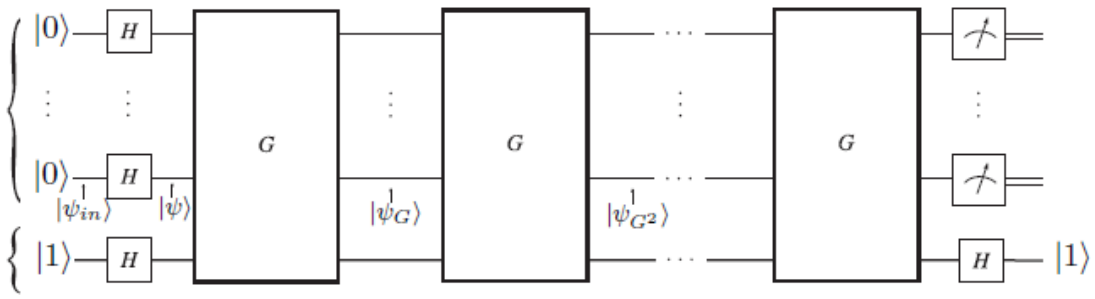

Fig. 11 Circuitul cuantic pentru algoritmul lui Grover

Stările sistemului sunt:

$$|\Psi_0\rangle = |0\rangle n|1\rangle$$

$$|\Psi_1\rangle = (H^{\otimes n}\otimes H)|0\rangle_n|1\rangle = \frac{1}{2^{\frac{n}{2}}}\sum_{x=0}^{2^n-1}|x\rangle_n\,(|0\rangle - |1\rangle)/\sqrt{2} = |\Psi\rangle(|0\rangle - |1\rangle)/\sqrt{2}$$

unde starea

$$|\Psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{N-1} |x\rangle_n$$

este obţinută prin aplicarea transformatei $H^{\otimes n}$ asupra lui $|0\rangle_n$. Această stare cuantică reprezintă superpoziţia tuturor stărilor posibile *x* din *n* qubiţi cu amplitudini egale date prin $1/\sqrt{N}$. Al doilea registru a fost iniţial în starea |1⟩, iar după aplicarea transformatei Hadamard se află în starea $|-\rangle = (|0\rangle - |1\rangle)/\sqrt{2}$.

4.5.2 Iteraţia Grover

Urmează iteraţia Grover, al cărei circuit este următorul:

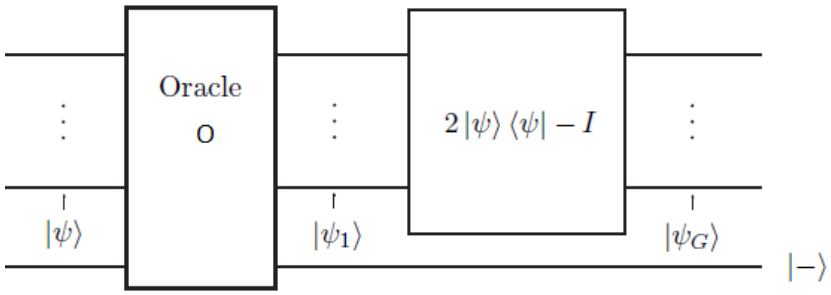

Fig. 12 Iteratia Grover

Iteraţia Grover constă în două transformări. Prima transformare marchează şi evidenţiază starea elementului căutat, iar a doua transformare amplifică amplitudinea de probabilitate a stării cuantice respective. Prima transformare este o transformare unitară care inversează starea elementului căutat . Mai exact, are loc inversarea fazei amplitudinii stării respective. Transformarea aplicată se numeşte "oracol" şi este definită astfel:

$$O|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$$

unde |x⟩ este o stare a primului registru (deci $x \in \{0, 1, \ldots, 2^n-1\}$), |y⟩ este o stare a celui de-al doilea registru (deci $y \in \{0,1\}$), iar f este o funcţie booleană , f: $\{0,1\}^n \rightarrow \{0,1\}$ cu $f(x)=1$ dacă $x=x_0$ este o solutie şi $f_0(x)=0$ în caz contrar.

Forma generala a actiunii oracolului se poate scrie astfel:

$$O(|x\rangle|-\rangle) = (-1)^{f(x)}|x\rangle|-\rangle$$

$$O(|\psi\rangle|-\rangle) = O\left(\frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle|-\rangle\right) = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}O(|x\rangle|-\rangle)$$
$$= \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle|-\rangle = |\psi_1\rangle|-\rangle$$

Starea primului registru după aplicarea oracolului este:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle$$

şi reprezintă o superpoziţie a tuturor stărilor bazei. Însă amplitudinea elementului căutat x_0 este negativă, în timp ce toate celelalte amplitudini sunt pozitive. Astfel, elementul căutat a fost marcat.

A doua transformare va amplifica amplitudinea stării elementului căutat prin aplicarea operatorului W = $2|\psi\rangle\langle\psi| - I$, numit inversare faţă de medie sau operator de difuzie.

$|\psi\rangle\langle\psi|$ reprezintă operatorul de proiecţie peste starea $|\psi\rangle$

$$(|\psi\rangle\langle\psi|)|a\rangle = \frac{1}{N}\begin{pmatrix}1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\\ \vdots & \vdots & \cdots & \vdots\\ 1 & 1 & \cdots & 1\end{pmatrix}\begin{pmatrix}a_1\\ a_2\\ \vdots\\ a_N\end{pmatrix} = \begin{pmatrix}\frac{\sum_{i=1}^{n}a_i}{N}\\ \frac{\sum_{i=1}^{n}a_i}{N}\\ \vdots\\ \frac{\sum_{i=1}^{n}a_i}{N}\end{pmatrix} = \overline{a}$$

iar $\overline{a}$ reprezintă vectorul valorilor medii. Astfel, operatorul W are ca efect inversarea tuturor stărilor în jurul mediei.

Transformarea de inversare faţă de medie este echivalentă cu succesiunea de transformări HZH, unde Z este operaţia de schimbare de fază condiţională, definită ca

$$Z: \left|x\right\rangle \rightarrow \begin{cases}|x\rangle, x = 0\\ -|x\rangle, x > 0\end{cases}$$

pentru orice stare $|x\rangle$ din baza de calcul (cu $0 \leq x \leq$ N-1), iar H este transformarea Walsh-Hadamard. Z se poate exprima ca Z = $2|0\rangle\langle 0|$-I, deci W = H($2|0\rangle\langle 0|$-I)H
Astfel, operatorul Grover poate fi scris:

G = HZHO

Ştiind că:

$$|\psi_1\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle = 2\,|\psi\rangle - \frac{2}{\sqrt{2^n}}|x_0\rangle$$

Starea primului registru după aplicarea operatorului de inversare faţă de medie devine:

$$|\psi_G\rangle = (2|\psi\rangle\langle\psi| - I) = (2|\psi\rangle\langle\psi| - I)\left(\,|\psi\rangle - \frac{2}{\sqrt{2^n}}|x_0\rangle\right)$$

$$|\psi_G\rangle = \frac{2^{n-2}-1}{2^{n-2}}|\psi\rangle + \frac{2}{\sqrt{2^n}}\,|x_0\rangle$$

sau

$$|\psi_G\rangle = \frac{2^{n-2}-1}{2^{n-2}} \cdot \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} x + \frac{2}{\sqrt{2^n}}\,|x_0\rangle$$

$|\psi_G\rangle$ este starea primului registru după aplicarea iteraţiei Grover . Al doilea registru este in starea $|-\rangle$. Amplitudinea stării marcate creşte cu O($1/\sqrt{N}$), iar amplitudinile stărilor nemarcate scad. Pentru ca probabilitatea să se măsoare starea marcată să fie apropiată de 1, este necesară aplicarea iteraţiei Grover de O($\sqrt{N}$) ori.

$$k_0 = round\left(\frac{\pi}{4}\sqrt{N}\right)$$

4.5.3 Implementarea în QCL

4.5.3.1 Implementarea iteraţiei Grover

Aşa cum s-a arătat mai sus, iteraţia Grover constă din două etape:

- inversarea stării soluţiei prin aplicarea operatorului oracol
- inversarea tuturor stărilor în jurul mediei prin aplicarea operatorului de difuzie

Operatorul oracol realizează o rotaţie cu π radiani a fazei tuturor vectorilor proprii care îndeplinesc condiţia de căutare:

$$O|x\rangle = \begin{cases} -|x\rangle, f(x) \\ |x\rangle, \neg f(x) \end{cases}$$

şi poate fi construit astfel:

```
O=!query(x,f) CPhase(π,f) query(x,f)
```

Condiţia de căutare poate fi implementată sub forma unei funcţii cuantice

query: $|x,0\rangle \rightarrow |x,f(x)\rangle$, cu $f:\{0,1\}^n \rightarrow \{0,1\}$

Algoritmul lui Grover poate fi utilizat pentru rezolvarea ecuaţiei $f(x) = 1$, în condiţiile în care nu sunt necesare informaţii adiţionale despre $f(x)$, decât că există o soluţie a ecuaţiei şi că aceasta este unică.
O interogare de test cu soluţia n poate fi implementată astfel:

```
qufunct query(qureg x,quvoid f,int n) {
  int i;
  for i=0 to #x-1 {     // x -> NOT (x XOR n)
    if not bit(n,i) { Not(x[i]); }
  }
  CNot(f,x);            // inversare f daca x=1111..
  for i=0 to #x-1 {     // x <- NOT (x XOR n)
    if not bit(n,i) { !Not(x[i]); }
  }
}
```

Operatorul de difuzie realizează inversarea tuturor stărilor în jurul mediei:

$$W = HZH$$

unde Z inversează semnul amplitudinii dacă şi numai dacă sistemul se află în starea $|0\rangle$.
Operatorul Z poate fi construit folosind poarta NOT şi operatorul CPhase de rotaţie condiţională a fazei. Implementarea QCL a operatorului de difuzie are următoarea formă:

```
operator diffuse(qureg q) {
  H(q);                 // transformare Hadamard
  Not(q);               // inversare q
  CPhase(pi,q);         // rotaţie dacă q=1111..
  !Not(q);              // undo inversare
  !H(q);                // undo transformare Hadamard
}
```

4.5.3.2 Implementarea algoritmului Grover

Implementarea completă a algoritmului lui Grover în QCL este următoarea:

```
procedure grover(int n) {
  int l=floor(log(n,2))+1;        // nr. de qubits
  int nr=ceil(pi/8*sqrt(2^l));     // nr. de iterations
  int m;
  int i;
  qureg q[l];
```

```
  qureg f[1];
  print l,"qubiti, ",m,"iteratii";
  {
    reset;
    H(q);                // transformare Hadamard
    for i= 1 to nr {     // iteratia Grover
      query(q,f,n);      // operatorul oracol
      CPhase(pi,f);
      !query(q,f,n);
      diffuse(q);        // operatorul de difuzie
    }
    measure q,m;         // măsurare
    print "s-a masurat",m;
  } until m==n;
  reset;                 // curatare registri locali
}
```

4.6 ALgoritmul lui Shor

4.6.1 Problema

Peter Shor a formulat în 1994 un algoritm care rezolvă problema factorizării unui întreg în timp polinomial. Algoritmul se aplică pentru numere întregi care nu sunt prime, pare sau puteri întregi ale unor numere prime, deoarece pentru astfel de numere există algoritmi clasici eficienţi pentru determinarea factorizarii.

Deci, utilizarea algoritmului lui Shor este utilă pentru rezolvarea următoarei probleme: dat un număr compus impar N, să se găsească un întreg d, $1<d<N$, care divide pe N. Mai mult, numărul N nu trebuie să fie putere a unui număr prim.

Algoritmul este următorul:
1.se alege un număr întreg aleator $x<N$
2.utilizând algoritmul lui Euclid, se calculează cmmdc(x,N)
3.dacă cmmdc(x,N) $\neq 1$ atunci am găsit un factor netrivial al lui N şi algoritmul se termină
4.dacă cmmdc(x,N) = 1, se află ordinul r al lui x
Ordinul lui x modulo N este cel mai mic număr întreg pozitiv r pentru care $x^r \equiv 1 \bmod N$.
5.dacă r este par şi dacă $0<y-1<y+1<N$ (unde $y=x^{r/2} \bmod N$) atunci factorii lui N sunt cmmdc($y\pm 1$, N) şi algoritmul se termină
6.se reia pasul 1

Nu există un algoritm clasic eficient pentru a afla ordinul lui x. Astfel, factorizarea se reduce la calculul ordinului unui întreg. Acesta poate fi aflat utilizând următorul circuit cuantic:

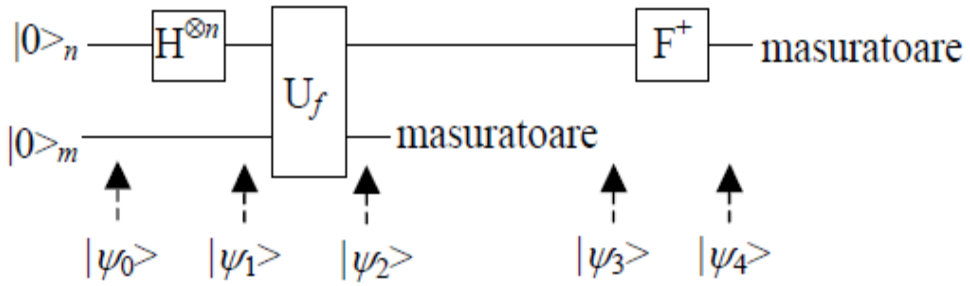

Fig. 13 Circuitul cuantic pentru algoritmului Shor

unde U_f este operatorul unitar

$$Uf\left(|j\rangle_n|k\rangle_m\right) = |j\rangle_n\left|x^j mod\ N\right\rangle_m$$

Primul registru are *n* qubiţi, unde n este ales astfel încât $N^2 \leq 2n < 2N^2$. Doar dacă r este o putere a lui 2 se va considera m=n.

Starea iniţială este

$$|\Psi_0\rangle = |0\rangle_n|0\rangle_m$$

Se aplică operatorul Hadamard asupra fiecărui qubit din primul registru.Starea sistemului cuantic devine:

$$|\Psi_1\rangle = \frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle_n|0\rangle_m$$

În continuare, se aplică operatorul U_f, starea sistemului cuantic devenind:

$$|\Psi_2\rangle = U_f|\Psi_1\rangle = \frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}U_f\left(|j\rangle_n|0\rangle_m\right) = \frac{1}{\sqrt{2^n}}\sum_{j=0}^{2^n-1}|j\rangle_n\left|x^j mod\ N\right\rangle_m$$

Deoarece U_f este liniar, acţionează asupra tuturor stărilor $|j\rangle_n|0\rangle_m$ simultan pentru toate cele 2^n valori ale *j* şi generează toate puterile lui *x* simultan. Această caracteristică este numită paralelism cuantic.

Deoarece $x^j \equiv x^{j+r} mod\ N$, funcţia $f(x,j) = x^j mod\ N$ este o funcţie periodică de perioadă *r*.
Putem scrie j=ar+b, $0 \le b \le r-1$ şi $0 \le a \le 2^n/r-1$. Atunci:

$$|\Psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{b=0}^{r-1} \sum_{a=0}^{\frac{2^n}{r}-1} |ar+b\rangle_n |x^{ar+b} mod\ N\rangle_m = \frac{1}{\sqrt{2^n}} \sum_{b=0}^{r-1} \sum_{a=0}^{\frac{2^n}{r}-1} |ar+b\rangle_n |x^b mod\ N\rangle_m$$

În acest moment se măsoară starea celui de-al doilea registru. Orice ieşire x^0, x^1, ..., x^{r-1} poate fi obţinută cu egală probabilitate. Presupunem că se măsoară x^{b_0}. După măsurătoare, starea sistemului cuantic devine:

$$|\Psi_3\rangle = \sqrt{\frac{r}{2^n}} \sum_{a=0}^{\frac{2^n}{r}-1} |ar+b_0\rangle_n |x^{b_0} mod\ N\rangle_m$$

După măsurătoare, constanta de normare devine $\sqrt{\frac{r}{2^n}}$ deoarece sunt $2^n/r$ termeni în suma de mai sus.

În pasul următor se aplică primului registru transformata Fourier cuantică inversă.

$$F^+|x\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{y=0}^{2^n-1} e^{-\frac{2\pi i x y}{2^n}} |y\rangle_n$$

Starea sistemului cuantic devine:

$$|\Psi_4\rangle = F^+|\Psi_3\rangle = \sqrt{\frac{r}{2^n}} \sum_{a=0}^{\frac{2^n}{r}-1} F^+|ar+b_0\rangle_n \, |x^{b_0} mod\ N\rangle_m$$

$$= \frac{1}{\sqrt{r}} \left[\sum_{j=0}^{2^n-1} S_j \, e^{-\frac{2\pi i j b_0}{2^n}} |j\rangle_n \right] |x^{b_0} mod\ N\rangle_m$$

unde

$$S_j = \frac{r}{2^n} \sum_{a=0}^{\frac{2^n}{r}-1} e^{-\frac{2\pi i j a r}{2^n}}$$

Dacă j este de forma $k \cdot \frac{2^n}{r}$, atunci $e^{-\frac{2\pi i j r}{2^n}}$=1 şi $S_j = 1$.
Dacă j nu este de forma $k \cdot \frac{2^n}{r}$, atunci

$$S_j = \frac{r}{2^n} \cdot \frac{1 - e^{-2\pi i j}}{1 - e^{-\frac{2\pi i j r}{2^n}}} = 0$$

Deci S_j este diferit de zero (S_j =1) dacă şi numai dacă j este de forma $k \cdot \frac{2^n}{r}$, k=0,1,…,r-1. Atunci:

$$|\Psi_4\rangle = \frac{1}{\sqrt{r}} \left(\sum_{k=0}^{r-1} e^{-\frac{2\pi i k b_0}{r}} \left| \frac{k \cdot 2^n}{r} \right\rangle_n \right) |x^{b_0} mod\ N\rangle_m$$

Se măsoară acum starea primului registru. Presupunem că se obţine valoarea $m=k_0 2^n/r$, unde k_0 poate fi orice valoare între 0 şi r-1 cu egală probabilitate.

Cunoscând m se determină r. Dacă valoarea măsurată este 0 (k_0=0), nu avem nicio informaţie despre r. În această situaţie partea cuantică a algoritmului trebuie rulată din nou. Dacă valoarea măsurată este diferită de zero ($k_0 \neq 0$), se împarte m la 2^n şi se obţine raportul k_0/r (nu se cunosc nici k_0, nici r). Se determină acum, utilizând un algoritm clasic, forma raţională a valorii $c= k_0/r$, adică se determină a şi b astfel încât $c=a/b$ şi cmmdc(a,b)=1. Atunci ordinul lui x este valoare b.

4.6.2 Implementare în QCL

4.6.2.1 Funcţii auxiliare

Implementarea algoritmului lui Shor utilizează următoarele funcţii:

`boolean testprime(int n)`
Testează dacă argumentul *n* este număr prim.

`boolean testprimepower(int n)`
Testează dacă argumentul *n* este o putere a unui număr prim.

`int powmod(int x,int a,int n)`
Calculează $x^a \bmod n$

`int denominator(real x,int qmax)`
Returnează numitorul q al celei mai bune aproximări raţionale a lui x ($\frac{p}{q} \approx x$) cu p, q < q_{max}

4.6.2.2 Procedura shor

Procedura `shor` verifică dacă argumentul întreg `number` este potrivit pentru factorizare cuantică algoritmul lui Shor până când este găsit un factor.

```
procedure shor(int number) {
   int width=ceil(log(number,2));
   qureg reg1[2*width];
   qureg reg2[width];
   int qmax=2^width;
   int factor;
   int m;
   real c;
   int x;
   int p;
   int q;
   int a;
   int b;
   int e; // e=x^(q/2) mod number
   if number mod 2 == 0
      { exit "number must be odd"; }
   if testprime(number)
      { exit "prime number"; }
   if testprimepower(number)
      { exit "prime power"; };

  {
      {
         x=floor(random()*(number-3))+2;
      } until gcd(x,number)==1;

   print "chosen random x =",x;
   Mix(reg1);
   expn(x,number,reg1,reg2);
   measure reg2;
   dft(reg1);
   measure reg1,m;
   reset;
   if m==0 {
      print "measured zero in 1st register. trying again ...";
   } else {
      c=m*0.5^(2*width);
      q=denominator(c,qmax);
```

```
      p=floor(q*c+0.5);
      print "measured",m,", approximation for",c,"is",p,"/",q;
      if q mod 2==1 and 2*q<qmax {
         print "odd denominator, expanding by 2";
         p=2*p; q=2*q;
      }
      if q mod 2==1 {
         print "odd period. trying again ...";
      } else {
         print "possible period is",q;
         e=powmod(x,q/2,number);
         a=(e+1) mod number;
         b=(e+number-1) mod number;
         print x,"^",q/2,"+ 1 mod",number,"=",a,",",
             x,"^",q/2,"- 1 mod",number,"=",b;
         factor=max(gcd(number,a),gcd(number,b));
      }
   }
   } until factor>1 and factor<number;
   print number,"=",factor,"*",number/factor;
}
```

5 Bibliografie

1. H. De Raedt, K. Michielsen. Handbook of theoretical and computational nanotechnology, volume 3, *Computational Methods for Simulating Quantum Computers*, American Scientific Publisher, Los Angeles, 2006. arXiv:quant-ph/0406210
2. B. Ömer, *Quantum programming in qcl*, Master's thesis, TU Vienna, 2000, http://tph.tuwien.ac.at/ oemer/doc/quprog.pdf
3. B. Ömer, *Structured Quantum Programming*, PhD thesis, TU Vienna, 2003, http://tph.tuwien.ac.at/ oemer/-doc/structquprog.pdf
4. D. Deutsch, *Quantum theory, the Church-Turing principle and the universal quantum computer*, Proceedings of the Royal Society of London A 400, pp. 97-117, 1985
5. D. Deutsch, *Quantum computational networks*, Proceedings of the Royal Society of London, A425, pp. 73–90, 1989
6. D. Deutsch, R. Jozsa, *Rapid Solution of Problems by Quantum Computation*, Proceedings of Royal Society of London. Vol. 439A, pp. 439-553, 1992
7. S. Arustei, V. Manta, *QCL Implementation of the Bernstein-Vazirani Algorithm*, Buletinul Institutului Politehnic din Iași, Automatic Control and Computer Science Section, vol. LIV(LVIII), no. 1, pages 35–44, 2008
8. D. R. Simon, *On the power of quantum computation*, SIAM Journal on Computing, 26, 1474-1483, 1997
9. L. K. Grover, *A fast quantum mechanical algorithm for database search*, Proceedings of 28th ACM STOC, pages 212–219, 1996
10. C. Lavor, LRU Manssur, R. Portugal, *Grover's Algorithm: Quantum Database Search*, arXiv:quant-ph/0301079
11. A.B. Mutiara, R. Refianti, *Simulation of Grover's Algorithm Quantum Search in a Classical Computer*, International Journal of Computer Science and Information Security 01/2010
12. P.W. Shor, *Algorithms for Quantum Computing: Discrete Logarithm and Factoring*, Proceedings of 35th Annual Symposium on Foundations of Computer Science, pp. 124-134, 1994
13. D. Unruh. *Quantum Programming Languages*. Informatik - Forschung und Entwicklung, vol. 21, no. 1, pages 55–63, 2006
14. S. Bettelli1, T. Calarco, L. Serafini. *Toward an architecture for quantum programming*, The European Physical Journal D - Atomic, Molecular, Optical and Plasma Physics, vol. 25, no. 2, pages 181–200, 2004