

ПРИЛОЖЕНИЕ П4: ПРОЕКТ QC-T011

Проектът *QC-T011* е развитие на *QC-T010*. Адекватността на модела е подобрена като сплитерът *BS2* “поглъща” (абсорбира) остатъчните състояния на фотона в съответствие с обратното преобразуване на Адамар.

/*
 * qc-T011.xc
 *
 * Промени спрямо qc-T010.xc:
 * - сплитерът BS2 «поглъща» (абсорбира) състоянията на фотона
 *
 * Създаден на: 19.06.2013
 * Авторы: Милен Луканчевски, Бисер Николов
 *
 * СУПЕРПОЗИЦИЯ
 * (ЕКСПЕРИМЕНТ С РАЗДВОЯВАНЕ ПЪТЯ НА ФОТОНА)
 *
 * Цел: Моделиране на суперпозицията на квантови частици.
 * Осн. идея: Експериментът се основава на интерферометъра на
 Мах-Цендер (Mach-Zehnder),
 * предназначен за интерференция на единични квантови частици.
 Илюстрира
 * както суперпозицията на единични qubit-ове, така и действието
 на
 * гейта на Адамар - един от основните квантови вентили.
 *
 * Моделирано явление: Суперпозиция, гейт на Адамар.
 *
 * Експериментална постановка:
 * - източник на единични фотони с предварително зададено
 квантово състояние;
 * - две полупрозрачни огледала под 45 градуса, служещи за
 сплитери (BS1, BS2);
 * - две огледала (с пълно отражение) под 45 градуса;
 * - два детектора на фотони.
 *
 * Източникът излъчва фотоните поединично.
 *
 * Огледалата могат да се настройват, като им се задава:
 * - полуотражение, за да могат да се използват като сплитери;
 * - пълно отражение.
 *
 * Основни елементи на модела:
 * - активни (задача/процес);
 * - пасивни (канали, съобщения).
 *
 * Източникът на светлина, огледалата и детекторът се представят
 като процеси.
 * Всяко едно полуотражателно огледало изпълнява ролята сплитер
 и се
 * представя като вентил на Адамар. Сплитерът BS2 отговаря за
 поглъщането на
 * остатъчните състояния при обратното преобразуване на Адамар.
 *
 * Квантовото състояние на фотоните се представя чрез единичен
 qubit.
 * Фотоните се предават като съобщения между тези процеси.

```

*
* Литература за използвания метод:
* [A.11, A.18, A.23, A.24, A.И.2]
* [B.6, B.12, B.13, B.14]
*
*/

// #include <xs1.h>
#include <platform.h>

// #include <stdlib.h>
// #include <string.h>
// #include <limits.h>
#include <math.h>

// НАЧАЛНО КВАНТОВО СЪСТОЯНИЕ НА ФОТОНА,
// ГЕНЕРИРАН ОТ ИЗТОЧНИКА LE
#define LE_QS 0
// 0 - квантово състояние QS_0 с кет-вектор |0>
// 1 - квантово състояние QS_1 с кет-вектор |1>

typedef enum {FALSE=0, TRUE} BOOL;
typedef unsigned int UINT;
typedef unsigned char BYTE;

#define STOP return

typedef enum {QS_0 = 0, QS_1, QS_S} QS;
typedef enum {FULL_REFLECTION=0, HALF_REFLECTION} REFLECTION;

typedef struct
{
    int r_part;
    int i_part;
} COMPLEX;

typedef struct
{
    double common_factor;
    COMPLEX m[2][2];
} OPERATOR_2x2;

typedef struct
{
    double common_factor;
    COMPLEX c_0;
    COMPLEX c_1;
} SINGLE_QUBIT;

typedef struct
{
    UINT id;           // identifier

```

```

    SINGLE_QUBIT q_state; // quantum state
} PHOTON;

// период на светене на светодиодите 500E6 x 10E-9 = 5 sec
const int T_PERIOD = 500000000;

out port LD0 = PORT_LED_3_0;
out port LD1 = PORT_LED_3_1;

void taskLightEmitter(chanend chanRight);
void taskMirror(chanend chanLeftIn, chanend chanUpOut,
                chanend chanDownIn, chanend chanRightOut,
                REFLECTION refl, int intCount);
void taskLightDetector(chanend chanIn, out port led,
                       QS qsToDetect);

QS getQS(SINGLE_QUBIT src);

SINGLE_QUBIT gateHadamard(SINGLE_QUBIT src);
COMPLEX __complex_mult(COMPLEX src1, COMPLEX src2);
COMPLEX __complex_sum(COMPLEX src1, COMPLEX src2);
COMPLEX __complex_div_by_scalar(COMPLEX src1, double src2);
double __complex_module(COMPLEX src);
int __gcd (int a, int b);

void Randomize(void);
UINT RNG_CRC32(UINT uintSeed, UINT uintPoly);
UINT RNG_ROSC(void);

int main(void)
{
    chan chanQuantum[12];

    par
    {
        // LE
        on stdcore[0]: taskLightEmitter(chanQuantum[0]);

        // M00
        on stdcore[1]: taskMirror(chanQuantum[4], chanQuantum[5],
                                chanQuantum[1], chanQuantum[6],
                                FULL_REFLECTION, 1);

        // M01
        on stdcore[2]: taskMirror(chanQuantum[6], chanQuantum[10],
                                chanQuantum[7], chanQuantum[11],
                                HALF_REFLECTION, 2);

        // M10
        on stdcore[0]: taskMirror(chanQuantum[0], chanQuantum[1],
                                chanQuantum[2], chanQuantum[3],
                                HALF_REFLECTION, 1);

        // M11
        on stdcore[1]: taskMirror(chanQuantum[3], chanQuantum[7],

```

```

        chanQuantum[8], chanQuantum[9],
        FULL_REFLECTION, 1);

    // LD0
    on stdcore[3]: taskLightDetector(chanQuantum[10], LD0, QS_0);
    // LD1
    on stdcore[3]: taskLightDetector(chanQuantum[11], LD1, QS_1);
}

return 0;
}

void taskLightEmitter(chanend chanRight)
{
    PHOTON phRight;

    #if (LE_QS == 0)
        phRight.id = 1;
        phRight.q_state.common_factor = 1;
        phRight.q_state.c_0.r_part = 1;
        phRight.q_state.c_0.i_part = 0;
        phRight.q_state.c_1.r_part = 0;
        phRight.q_state.c_1.i_part = 0;
    #elif (LE_QS == 1)
        phRight.id = 1;
        phRight.q_state.common_factor = 1;
        phRight.q_state.c_0.r_part = 0;
        phRight.q_state.c_0.i_part = 0;
        phRight.q_state.c_1.r_part = 1;
        phRight.q_state.c_1.i_part = 0;
    #else
        #error INVALID LE_QS
    #endif

    chanRight <: phRight;

    STOP;
}

void taskMirror(chanend chanLeftIn, chanend chanUpOut,
               chanend chanDownIn, chanend chanRightOut,
               REFLECTION refl, int intCount)
{
    PHOTON phLeft, phUp, phDown, phRight;
    SINGLE_QUBIT q_state;
    // За реализацията на "поглъщането"/absorbtion при BS2
    BOOL boolUpSent = FALSE, boolRightSent = FALSE;

    while(intCount--)
    {
        select
        {
            case chanLeftIn :> phLeft:

```

```

{
  if(refl == HALF_REFLECTION)
  {
    q_state = gateHadamard(phLeft.q_state);

    phUp.id = phLeft.id;
    phUp.q_state = q_state;

    phRight.id = phLeft.id;
    phRight.q_state = q_state;

    switch(getQS(q_state))
    {
      case QS_0:
        if(!boolUpSent)
        {
          chanUpOut <: phUp;
          boolUpSent = TRUE;
        }
        break;
      case QS_1:
        if(!boolRightSent)
        {
          chanRightOut <: phRight;
          boolRightSent = TRUE;
        }
        break;
      case QS_S:
        if(!boolUpSent)
        {
          chanUpOut <: phUp;
          boolUpSent = TRUE;
        }
        if(!boolRightSent)
        {
          chanRightOut <: phRight;
          boolRightSent = TRUE;
        }
        break;
    }
  }
  else // FULL_REFLECTION
  {
    chanUpOut <: phLeft;
  }
  break;
}
case chanDownIn :> phDown:
{
  if(refl == HALF_REFLECTION)
  {
    q_state = gateHadamard(phDown.q_state);

```

```

phUp.id = phDown.id;
phUp.q_state = q_state;

phRight.id = phDown.id;
phRight.q_state = q_state;

switch(getQS(q_state))
{
    case QS_0:
        if(!boolUpSent)
        {
            chanUpOut <: phUp;
            boolUpSent = TRUE;
        }
        break;
    case QS_1:
        if(!boolRightSent)
        {
            chanRightOut <: phRight;
            boolRightSent = TRUE;
        }
        break;
    case QS_S:
        if(!boolUpSent)
        {
            chanUpOut <: phUp;
            boolUpSent = TRUE;
        }
        if(!boolRightSent)
        {
            chanRightOut <: phRight;
            boolRightSent = TRUE;
        }
        break;
}
}
else // FULL_REFLECTION
{
    chanRightOut <: phDown;
}
break;
}
}

// При BS2 (второто полуотражателно огледало)
// трябва да се изпрати съобщение и към втория процес LD,
// за да се терминира
if(refl == HALF_REFLECTION)
{
    if(!boolUpSent)

```

```

    {
        chanUpOut <: phUp;
        boolUpSent = TRUE;
    }
    if(!boolRightSent)
    {
        chanRightOut <: phRight;
        boolRightSent = TRUE;
    }
}

STOP;
}

void taskLightDetector(chanend chanIn, out port led,
                       QS qsToDetect)
{
    PHOTON phIn;
    int intCounter = 0;

    UINT uintT;
    timer timerT;    // таймер за формиране на изчакване

    led <: 0;

    chanIn :> phIn;
    if(getQS(phIn.q_state) == qsToDetect)
    {
        intCounter++;

        led <: 1;
        timerT :> uintT;
        timerT when timerafter(uintT + T_PERIOD) :> void;
        led <: 0;
    }

    STOP;
}

QS getQS(SINGLE_QUBIT src)
{
    QS dst;

    if(__complex_module(src.c_0) && !__complex_module(src.c_1))
        dst = QS_0;
    else if(!__complex_module(src.c_0) && __complex_
module(src.c_1))
        dst = QS_1;
    else
        dst = QS_S;

    return dst;
}

```



```

}

SINGLE_QUBIT gateHadamard(SINGLE_QUBIT src)
{
    OPERATOR_2x2 H;
    SINGLE_QUBIT dst;
    double module1, module2;
    int factor;

    // Оператор на Адамар
    H.common_factor = M_SQRT1_2;
    H.m[0][0].r_part = 1;
    H.m[0][0].i_part = 0;
    H.m[0][1].r_part = 1;
    H.m[0][1].i_part = 0;
    H.m[1][0].r_part = 1;
    H.m[1][0].i_part = 0;
    H.m[1][1].r_part = -1;
    H.m[1][1].i_part = 0;

    dst.common_factor = H.common_factor*src.common_factor;
    dst.c_0 = __complex_sum(__complex_mult(H.m[0][0], src.c_0),
                           __complex_mult(H.m[0][1], src.c_1));
    dst.c_1 = __complex_sum(__complex_mult(H.m[1][0], src.c_0),
                           __complex_mult(H.m[1][1], src.c_1));

    module1 = __complex_module(dst.c_0);
    module2 = __complex_module(dst.c_1);

    factor = __gcd(module1, module2);

    dst.common_factor *= factor;
    dst.c_0 = __complex_div_by_scalar(dst.c_0, factor);
    dst.c_1 = __complex_div_by_scalar(dst.c_1, factor);

    return dst;
}

COMPLEX __complex_mult(COMPLEX src1, COMPLEX src2)
{
    COMPLEX dst;

    par
    {
        dst.r_part = src1.r_part*src2.r_part -
                     src1.i_part*src2.i_part;
        dst.i_part = src1.r_part*src2.i_part +
                     src1.i_part*src2.r_part;
    }

    return dst;
}

```

```

COMPLEX __complex_sum(COMPLEX src1, COMPLEX src2)
{
    COMPLEX dst;

    par
    {
        dst.r_part = src1.r_part + src2.r_part;
        dst.i_part = src1.i_part + src2.i_part;
    }

    return dst;
}

COMPLEX __complex_div_by_scalar(COMPLEX src1, double src2)
{
    COMPLEX dst;

    par
    {
        dst.r_part = src1.r_part/src2;
        dst.i_part = src1.i_part/src2;
    }

    return dst;
}

double __complex_module(COMPLEX src)
{
    double dst = sqrt(src.r_part*src.r_part +
                      src.i_part*src.i_part);

    return dst;
}

int __gcd (int a, int b)
{
    int c;

    while (a != 0)
    {
        c = a;
        a = b%a;
        b = c;
    }

    return b;
}

```