ПРИЛОЖЕНИЕ П6: ПРОЕКТ QC-T020

Проектът *QC-T020* съдържа кода на предложения предварителен вариант на паралелен структурен модел на *квантовата телепортация*, като освен вентила на Адамар е реализиран и *2-qubit* квантов *CNOT* вентил. Основните структурни елементи на модела са: *EPR* източник на сплетени квантови частици, входен квантов канал, кодер, 2 bit класически канал, декодер, приемник на телепортираната квантова информация.

```
/*
 * qc-T020.xc
 *
 *  Създаден на: 19.06.2013
 *       Автори: Милен Луканчевски, Бисер Николов
 *
 *  КВАНТОВА ТЕЛЕПОРТАЦИЯ
 *
 *  Цел: Моделиране на квантова телепортация
 *  Осн. идея:
 *
 *  Моделирано явление: Квантова телепортация
 *
 *  Структурни елементи:
 *  - EPR източник
 *  - входен канал с квантова информация
 *  - кодер
 *  - 2-битов класически канал
 *  - декодер
 *  - приемник на телепортираната квантова информация
 *
 *  Основни елементи на модела:
 *  - активни (задача/процес);
 *  - пасивни (канали, съобщения).
 *
 *  Активните структурни елементи се представят като процеси.
 *
 *  Квантовото състояние на фотоните се представя чрез единичен
qubit.
 *  Фотоните се предават като съобщения между тези процеси.
 *
 *  Литература за използвания метод:
 *  [А.23, А.24, А.И.2]
 *  [Б.6, Б.12, Б.13, Б.14]
 *
 */

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

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

// КВАНТОВО СЪСТОЯНИЕ НА ФОТОНА, КОЕТО ТРЯБВА ДА СЕ ТЕЛЕПОРТИРА
#define LE_QS  0
// 0 - квантово състояние QS_0 с кет-вектор |0>
// 1 - квантово състояние QS_1 с кет-вектор |1>

#define RNG_TYPE  1
// 1 - Генератор тип RNG  с примитивна функция crc32 на
```

```
изпълнителната среда;
// 2 - Генератор тип TRNG с кръговите осцилатори от
изпълнителната среда
//     поддържа се при XS1-L, но не и при XS1-G4.

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 struct
{
  double r_part;
  double 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;

const int T_PERIOD = 500000000;  // период на светене на
светодиодите
                // 500E6 x 10E-9 = 5 sec
out port LD0 = PORT_LED_3_0;
out port LD1 = PORT_LED_3_1;

void taskLightEmitter(chanend chanOut1, chanend chanOut2,
                      chanend chanOut3);
void taskEPRSource(chanend chanIn1, chanend chanIn2,
                   chanend chanOut1, chanend chanOut2);
void taskEncoder(chanend chanIn1, chanend chanIn2,
                 chanend chanOutClassic2Bits,
                 chanend chanOutEntanglement);
void taskDecoder(chanend chanIn, chanend chanInClassic2Bits,
```

```
                chanend chanOut, chanend chanInEntanglement);
void taskReceiver(chanend chanIn);

BYTE BellStateMeasurement(SINGLE_QUBIT src, SINGLE_QUBIT &dst);

QS getQS(SINGLE_QUBIT src);

SINGLE_QUBIT gateHadamard(SINGLE_QUBIT src);
SINGLE_QUBIT gateX(SINGLE_QUBIT src);
SINGLE_QUBIT gateZ(SINGLE_QUBIT src);
SINGLE_QUBIT gateCNOT(SINGLE_QUBIT ctrl, 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);
double __fgcd (double fa, double fb);

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

int main(void)
{
  chan chanQuantum[6];
  chan chanClassic2Bits;
  chan chanEntanglement;

  par
  {
    // LE
    on stdcore[0]: taskLightEmitter(chanQuantum[0],
chanQuantum[1], chanQuantum[2]);

    // EPR Source
    on stdcore[1]: taskEPRSource(chanQuantum[1], chanQuantum[2],
                    chanQuantum[3], chanQuantum[4]);
    // Encoder
    on stdcore[1]: taskEncoder(chanQuantum[0], chanQuantum[3],
                     chanClassic2Bits, chanEntanglement);

    // Decoder
    on stdcore[3]: taskDecoder(chanQuantum[4], chanClassic2Bits,
                     chanQuantum[5], chanEntanglement);
    // Receiver
    on stdcore[3]: taskReceiver(chanQuantum[5]);
  }

  return 0;
}
```

```
void taskLightEmitter(chanend chanOut1, chanend chanOut2,
                      chanend chanOut3)
{
  PHOTON phOut1, phOut2, phOut3;

#if (LE_QS == 0)
  phOut1.id = 1;
  phOut1.q_state.common_factor = 1;
  phOut1.q_state.c_0.r_part = 1;
  phOut1.q_state.c_0.i_part = 0;
  phOut1.q_state.c_1.r_part = 0;
  phOut1.q_state.c_1.i_part = 0;
#elif (LE_QS == 1)
  phOut1.id = 1;
  phOut1.q_state.common_factor = 1;
  phOut1.q_state.c_0.r_part = 0;
  phOut1.q_state.c_0.i_part = 0;
  phOut1.q_state.c_1.r_part = 1;
  phOut1.q_state.c_1.i_part = 0;
#else
  #error INVALID LE_QS
#endif

  phOut2.id = 2;
  phOut2.q_state.common_factor = 1;
  phOut2.q_state.c_0.r_part = 1;
  phOut2.q_state.c_0.i_part = 0;
  phOut2.q_state.c_1.r_part = 0;
  phOut2.q_state.c_1.i_part = 0;

  phOut3.id = 3;
  phOut3.q_state.common_factor = 1;
  phOut3.q_state.c_0.r_part = 1;
  phOut3.q_state.c_0.i_part = 0;
  phOut3.q_state.c_1.r_part = 0;
  phOut3.q_state.c_1.i_part = 0;

  chanOut1 <: phOut1;
  chanOut2 <: phOut2;
  chanOut3 <: phOut3;

  STOP;
}

void taskEPRSource(chanend chanIn1, chanend chanIn2,
                   chanend chanOut1, chanend chanOut2)
{
  PHOTON phIn1, phIn2, phOut1, phOut2;

  par
  {
```

```
    chanIn1 :> phIn1;
    chanIn2 :> phIn2;
  }

  phOut1 = phIn1;
  phOut2 = phIn2;

  phOut1.q_state = gateHadamard(phIn1.q_state);
  phOut2.q_state = gateCNOT(phOut1.q_state, phIn2.q_state);

//  par
  {
    chanOut1 <: phOut1;
    chanOut2 <: phOut2;
  }
}

void taskEncoder(chanend chanIn1, chanend chanIn2, chanend
chanOutClassic2Bits, chanend chanOutEntanglement)
{
  PHOTON phIn1, phIn2;
  SINGLE_QUBIT q_state1, q_state2, q_stateEntanglement;
  BYTE byteClassic2Bits = 0;

  par
  {
    chanIn1 :> phIn1;
    chanIn2 :> phIn2;
  }

  // Кодиране на 2 bit информация за класическия канал
  q_state2 = gateCNOT(phIn1.q_state, phIn2.q_state);
  q_state1 = gateHadamard(phIn1.q_state);

  byteClassic2Bits =
    BellStateMeasurement(q_state2, q_stateEntanglement);

  chanOutClassic2Bits <: byteClassic2Bits;
  chanOutEntanglement <: q_stateEntanglement;
}

void taskDecoder(chanend chanIn, chanend chanInClassic2Bits,
                 chanend chanOut, chanend chanInEntanglement)
{
  PHOTON phIn, phOut;
  BYTE byteClassic2Bits;
  SINGLE_QUBIT q_stateEntanglement;  // Entanglement State

  par
  {
    chanIn :> phIn;
    chanInClassic2Bits :> byteClassic2Bits;
```

```
    chanInEntanglement :> q_stateEntanglement;
  }

  // Декодиране (корекция на грешката)
  phOut = phIn;
  phOut.q_state = q_stateEntanglement;
  if(byteClassic2Bits & 0x2)
  {
    phOut.q_state = gateZ(phOut.q_state);
  }
  if(byteClassic2Bits & 0x1)
  {
    phOut.q_state = gateX(phOut.q_state);
  }

  chanOut <: phOut;
}

void taskReceiver(chanend chanIn)
{
  PHOTON phIn;

  chanIn :> phIn;
}

BYTE BellStateMeasurement(SINGLE_QUBIT src, SINGLE_QUBIT &dst)
{
  dst = src;

  //  TODO

  return 0;
}

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 = __fgcd(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;
}

SINGLE_QUBIT gateX(SINGLE_QUBIT src)
{
  OPERATOR_2x2 X;
  SINGLE_QUBIT dst;
  double module1, module2;
  int factor;

  // Оператор за инверсия
  X.common_factor  = 1;
  X.m[0][0].r_part = 0;
  X.m[0][0].i_part = 0;
  X.m[0][1].r_part = 1;
  X.m[0][1].i_part = 0;
  X.m[1][0].r_part = 1;
  X.m[1][0].i_part = 0;
  X.m[1][1].r_part = 0;
  X.m[1][1].i_part = 0;

  dst.common_factor = X.common_factor*src.common_factor;
```

```
  dst.c_0 = __complex_sum(__complex_mult(X.m[0][0], src.c_0),
                          __complex_mult(X.m[0][1], src.c_1));
  dst.c_1 = __complex_sum(__complex_mult(X.m[1][0], src.c_0),
                          __complex_mult(X.m[1][1], src.c_1));

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

  factor = __fgcd(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;
}

SINGLE_QUBIT gateZ(SINGLE_QUBIT src)
{
  OPERATOR_2x2 Z;
  SINGLE_QUBIT dst;
  double module1, module2;
  int factor;

  // Оператор за обръщане на фазата на 180 градуса
  Z.common_factor  = 1;
  Z.m[0][0].r_part = 1;
  Z.m[0][0].i_part = 0;
  Z.m[0][1].r_part = 0;
  Z.m[0][1].i_part = 0;
  Z.m[1][0].r_part = 0;
  Z.m[1][0].i_part = 0;
  Z.m[1][1].r_part = -1;
  Z.m[1][1].i_part = 0;

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

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

  factor = __fgcd(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;
}
```

```
SINGLE_QUBIT gateCNOT(SINGLE_QUBIT ctrl, SINGLE_QUBIT src)
{
  SINGLE_QUBIT dst = src;

  // Fig. 10.1 [A.20]
  if(getQS(ctrl) == QS_1)
  {
    dst = gateHadamard(src);
    dst = gateZ(dst);
    dst = gateHadamard(dst);
  }
  else if(getQS(ctrl) == QS_S)
  {
    if(getQS(src) == QS_0 || getQS(src) == QS_1)
    {
      dst = gateHadamard(src);  // Преминава в суперпозиция
    }
    else
    {
      //  TODO - при dst в суперпозиция
    }
  }

  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;
}

/* НОД
 * по алгоритъма на Евклид
 */
int __gcd (int a, int b)
{
  int c;

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

  return b;
}

double __fgcd (double fa, double fb)
{
  int a = (int) fa;
  int b = (int) fb;
  int c;

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

```
  }

  return (double) b;
}

/*************************************************************
 * ГЕНЕРАЦИЯ НА СЛУЧАЙНИ ПОСЛЕДОВАТЕЛНОСТИ
 *************************************************************/

UINT uintCRC32Reg = 1;  // CRC32 - глобално състояние на RNG
генератора

void Randomize(void)
{
  timer timerSeed;
  UINT uintSeed;
  timerSeed :> uintSeed;

  RNG_CRC32(uintSeed, 0xEB31D82E);
}

/*
 * ГЕНЕРАТОР НА ПСЕВДОСЛУЧАЙНА ПОСЛЕДОВАТЕЛНОСТ - RNG
 * (чрез примитивна функция на изпълнителната среда)
 */
UINT RNG_CRC32(UINT uintSeed, UINT uintPoly)
{
  if (uintSeed > 0)
    uintCRC32Reg = uintSeed;

  crc32(uintCRC32Reg, 0xFFFFFFFF, uintPoly);

  return uintCRC32Reg;
}

/*
 * ГЕНЕРАТОР НА ДЕЙСТВИТЕЛНА СЛУЧАЙНА ПОСЛЕДОВАТЕЛНОСТ RRNG/TRNG
 * (чрез наличен в изпълнителната среда източник на ентропия)
 */

// работен период на кръговите генератори
// 50 usec = 5000 x 10E-9 = 50E-6 sec
const int ROSC_PERIOD = 5000;

UINT RNG_ROSC(void)
{
  timer timerT;     // таймер за формиране на работния период
  UINT uintT;
  UINT uintResult;  // буфер на 32 bit случайна последователност
  UINT r0, r1, r2, r3;     // начална стойност на ROSC
  UINT r0a, r1a, r2a, r3a; // крайна стойност на ROSC
```

```
  uintResult = 0;

  // Disable Ring Oscilators
  setps(0x060B, 0x0);

  // Цикли на обръщение към кръговите генератори ROSC
  for(int i = 0; i < 8; i++)
  {
    // Read Ring Oscilators
    r0a = getps(0x070B);
    r1a = getps(0x080B);
    r2a = getps(0x090B);
    r3a = getps(0x0A0B);

    // Enable Ring Oscilators
    setps(0x060B, 0xF);

    // Изчакване на работния период
    timerT :> uintT;
    timerT when timerafter(uintT + ROSC_PERIOD) :> void;

    // Disable Ring Oscilators
    setps(0x060B, 0x0);

    // Read Ring Oscilators
    r0 = getps(0x070B);
    r1 = getps(0x080B);
    r2 = getps(0x090B);
    r3 = getps(0x0A0B);

    // освобождаване на старшата тетрада на буфера
    uintResult >>= 4;

    uintResult |= (
      ((r0 - r0a) & 0x1) << 3 |  // старши бит на тетрадата
      ((r1 - r1a) & 0x1) << 2 |
      ((r2 - r2a) & 0x1) << 1 |
      ((r3 - r3a) & 0x1)         // младши бит на тетрадата
                  ) << 28;  // изместване на формираната 4 bit
                            // случайна последователност
                           // в старшата тетрада на 32 bit буфер
  }

  // Връщане на формираната 32 bit действителна сл. посл.
  return uintResult;
}
/*****************************************************************/
```