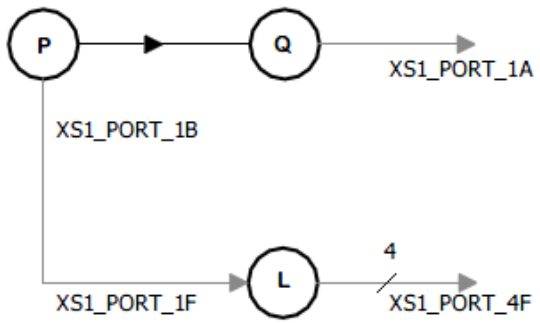
2.1. ГЕНЕРАТОРИ С ИЗМЕСТВАЩИ РЕГИСТРИ С ЛИНЕЙНА ОБРАТНА ВРЪЗКА

Решава се втората от поставените в т. 1.4 задачи: в паралелната изпълнителна среда да се реализира група от основните генератори на псевдослучайни последователности с оглед последващо изучаване на техните свойства и особености.

Формулировка на задачата:

1. Да се предложи обща структура на паралелната система, независима от конкретния вид на генератора.
2. Да се предложи реализация на *LFSR* с конфигурация Фибоначи.
3. Да се предложи реализация на *LFSR* с конфигурация Галоа.
4. Да се предложи реализация на *Алгоритъм-М* като вариант за стохастическо обединение на последователностите, изработвани от два различни генератора.

Паралелната система, обединяваща решението на поставените задачи има следното *CSP* уравнение {*P* || *Q* || *L*}, т.е. тя се декомпозира в три паралелни процеса. Процесът *P* вика генераторните функции и предава бит по бит по изходния си канал към *Q* генерираната случайна последователност. Процесът *Q* пакетира в *32* bit думи получените на входния си канал побитови последователности и ги записва в масива на извадката. Процесът *L* изпълнява помощната функция да индицира работата на *P* на светодиодната индикация.



Фиг. 2.1. Схема на връзките в паралелната система

Логическият граф на системата е обединен с използваните физически сигнални линии и е представен на фиг. 2.1.

Генерираната побитова случайна последователност се предава от *P* към *Q* по канала `chanPC`. Процесът *Q* я препредава към еднобитовия изходен порт `XS1_PORTA_1A`. Процесът *L* управлява светодиодната индикация, включена към 4 bit порт `XS1_PORT_4F`. Процесът *P* формира по порт `XS1_PORT_1B` стробиращ сигнал по време на нормалния цикъл на генерация. Процесът *L* следи стробиращия сигнал по порт `XS1_PORT_1F`.

На горното ниво на декомпозиция основните етапи на работа се задават от главната функция `main()`

```
int main(void)
{
    chan chanPC;

    oportSync <: 0;

    Randomize();

    par
    {
          taskP(chanPC);
          taskQ(chanPC);
          taskL();
    }

    oportLed <: 0;

    return 0;
}
```

Първоначално се дезактивира стробиращия сигнал между *P* и *L*, като на еднобитовия порт `oportSync` се записва 0. След това се извършва рандомизация на генераторите чрез обръщението към функцията `Randomize()`.

Операторът `par` стартира съставния паралелен процес {*P* || *Q* || *L*}. При изхода от паралелния процес се гаси светодиодната индикация чрез запис на *0* в *4* bit изходен порт `oportLed`. В конкретния случай и трите процеса, влизащи в състава на паралелната композиция са безкрайни[1].

1 Както е известно, едно от общите условия за коректност на даден алгоритъм е той да се терминира. Съществуват обаче изключения от това правило. Например командният интерпретатор, като част от ОС, не се терминира. Аналогична е и ситуацията при системите за работа в реално

Процесът *P* вика избраната генераторна функция, зададена с макроса `RNG_TYPE`. За проследяване на етапите на работа на *P* тук е показан само участъка за обръщение към една от генераторните функции - `RNG_LFSR_F()`. Пълният вариант е включен в Приложение *П1*.

```
void taskP(chanend chanRight)
{
    UINT uintMsg;
    int intButStop;

    // изчакване на старт
    iportButStart when pinseq(0) :> void;

    // Генериране на крайна последователност
    // N - размер на извадката в words
    // M - дължина на регистъра
    for(int i = 0; i < N*M; i++)
    {
          uintMsg = RNG_LFSR_F(0, 0x80000057);

          chanRight <: uintMsg;
          oportSync <: 1;
    }

    // Генериране на неограничена последователност
    while(TRUE)
    {
          iportButStop :> intButStop;
          if(intButStop == 0)
          {
                oportSync <: 0;

                // изчакване на повторен старт
                iportButStart when pinseq(0) :> void;
          }

          uintMsg = RNG_LFSR_F(0, 0x80000057);
          chanRight <: uintMsg;
          oportSync <: 1;
    }
```

В началото на процеса с оператора `wait` се изчаква начален сигнал за стартиране, който е нулево ниво на порт `iportButStart`.

време, където процесите по правило са безкрайни. Такъв е и разглежданият случай.

След това ясно се разграничават двата основни етапа на работа – етапа на формиране на началната ограничена последователност от *1024x32* bit, последван от етапа на формиране на неограничената последователност.

В началото на всяка итерация на втория етап се проверява входния порт `iportButStop`. Ако бутонът, свързан към този порт е натиснат, се спира генерацията и строба към процеса *L*, след което отново се чака сигнал за стартиране.

Останалият участък и от двата етапа съвпада:

- в променливата `uintMsg` се записва *1* bit стойност, върната от генераторната функция;
- тази стойност се изпраща по канала `chanRight` към процеса *Q*;
- стробиращият сигнал се поддържа активен на порт `oportSync`.

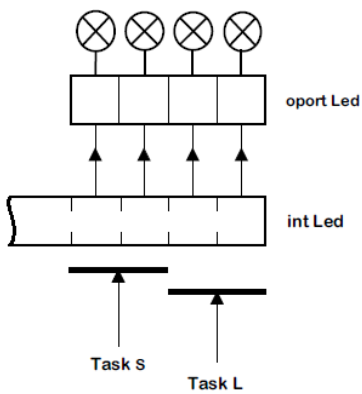
Структурата на процеса *Q* отговаря на двата основни етапа от работата на процеса *P*. Получената в рамките на първия етап последователност от *1024x32* bit се пакетира в *32* bit думи с оглед формирането на масива на извадката `uintRandomArr[]`. И на двата етапа, всеки един получен бит от случайната последователност се извежда към порт `oportRngBit`.

```
void taskQ(chanend chanLeft)
{
    int i, j;
    UINT uintMsg;

    for (i = 0; i < N; i++)
    {
          // пакетиране
          for (j = 0; j < M; j++)
          {
                chanLeft :> uintMsg;
                oportRngBit <: uintMsg;

                uintMsg = uintMsg << 31;
                uintRandomArr[i] = uintMsg |
(uintRandomArr[i] >> 1);
          }
    }

    while(TRUE)
    {
          chanLeft :> uintMsg;
          oportRngBit <: uintMsg;
    }
}
```

Процесът *L*, управляващ *4* bit светодиоден индикатор, има няколко особени момента:

- изчакването на стробиращия сигнал на порт `iportSync`;
- светването/изгасването на индикатора;
- изчакването на периода на сработване `LED_PERIOD` на таймера `timerT`.

```
void taskL(void)
{
    timer timerT;
    int intT;

    timerT :> intT;
    intT += LED_PERIOD;

    intLed = 0;
    oportLed <: intLed;

    while(TRUE)
    {
          // изчакване на вх. строб
          iportSync when pinseq(1) :> void;

          intLed = ~intLed;
          oportLed <: (intLed & 0x1);

          timerT :> intT;
          intT += LED_PERIOD;
          timerT when timerafter(intT) :> void;
    }
}
```



Фиг. 2.2. Управление на светодиодната индикация

За *L* са отделени младшите два светодиода, докато за контролиращия процес *S* – старшите два (фиг. 2.2).

Предложената обща структура на паралелната система е независима от конкретния вид на генератора. В работата са предложени реализации на три от най-разпространените типове генератори: *LFSR* с конфигурация Фибоначи и конфигурация Галоа, както и *Алгоритъм-М*.

И двете разновидности на *LFSR* генераторите използват обща *32* bit променлива на състоянието `uintShiftReg`. Нейната първоначална стойност е *1*. Желателно е тази начална стойност да се променя по случаен закон преди използването на генераторите. За това служи функцията `Randomize()`.

Случайният характер на началната стойност на променливата на състоянието `uintShiftReg` се осигурява на две стъпки

```
timer timerSeed;
UINT uintSeed;
timerSeed :> uintSeed;

// инициализация на LFSR
RNG_LFSR_F(uintSeed, 0x80000057);
```

На първата стъпка се чете текущото съдържание на таймера `timerSeed`. Тази стойност сама по себе си е случайна – макар при начална инициализация таймерът да е нулиран, периодът до стартирането на функцията `Randomize()` в повечето случаи е неопределен. Това е обща практика при рандомизиращите функции, насочени към определена платформа. Например в *STL* библиотеката на *C++*, рандомизиращата функция *randomize()* връща към *srand()* текущото показание на системното време[2]

```
srand((unsigned)time(NULL));
```

На втората стъпка от изпълнението на функцията `Randomize()`, променливата на състоянието `uintShiftReg` се установява чрез еднократно обръщение към съответната генераторна функция, в случая `RNG_LFSR_F()`.

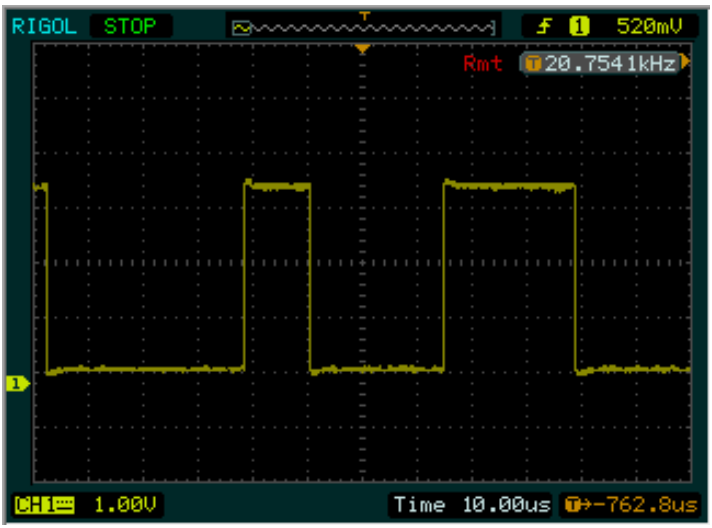
Генераторната функция на *LFSR* с конфигурация Фибоначи е `RNG_LFSR_F()`. Тя е изпълнена в два варианта, разграничавани с макроса `LFSR_F_VER`. И двата варианта отговарят на схемата от фиг.

2 Тази начална стойност, задавана на генераторната функция се обозначава като *seed* (*семенце*), откъдето идва и префикса *s* на функцията *srand()*.

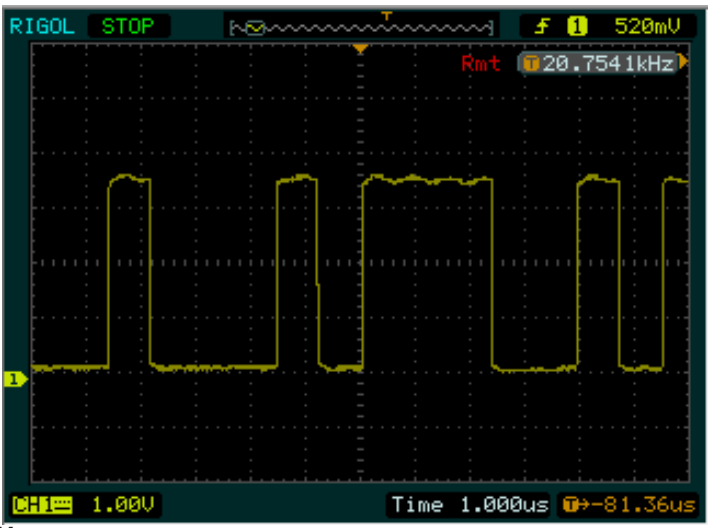
1.2. При първия вариант (`LFSR_F_VER == 1`) се работи с фиксирания полином

$$x^{32} + x^7 + x^5 + x^3 + x^2 + x + 1, \qquad (2.1)$$

чийто шестнайсетичен код по нотацията на *Филип Купман* с пропуск на младшия бит е *0x80000057*. При втория вариант (`LFSR_F_VER == 2`) функцията има задължителен втори параметър uintPoly, който задава конкретния 32 bit полином.



Фиг. 2.3. Част от осцилограмата на изходния сигнал на порт *oportRngBit* при *RNG LFSR*-Фибоначи

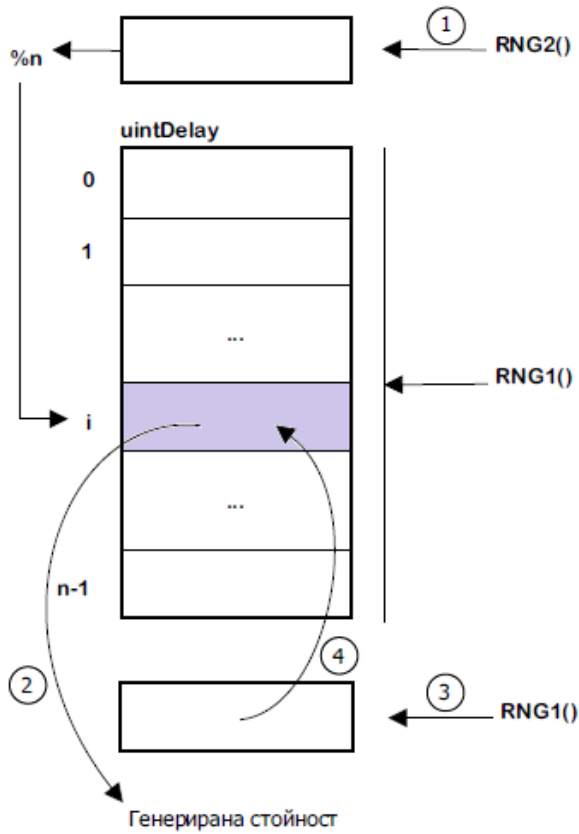


Фиг. 2.4. Част от осцилограмата на изходния сигнал на порт *oportRngBit* при *RNG LFSR*-Галоа

И при двата варианта функцията връща *1* bit от случайната последователност.

Реализацията на *RNG LFSR* с конфигурация Галоа е удачна за сравнение. Разликата е във веригата за обратна връзка (фиг. 1.3). Докато при конфигурация Фибоначи се използва един *32*-входен *XOR* елемент, тук се използват *32* двувходови *XOR* елемента. Статистическите свойства и апаратната сложност на двете конфигурации е еднаква. Софтуерната реализация на конфигурацията на Галоа обаче е много по-ефективна – една единствена побитова операция *XOR* е достатъчна за формиране сигнала на изхода на веригата за обратна връзка.

На фиг. 2.3 и 2.4 са приведени част от осцилограмите на изходния сигнал на порт `oportRngBit`, съответно за конфигурация Фибоначи и Галоа. В първия случай хоризонталната развивка е *10* μs, а във втория *1* μs. Съпоставянето на периода на един бит от сигнала показва, че реализацията с конфигурация на Галоа е около *10* пъти по-бърза, което се съгласува с теорията.



Фиг. 2.5. Схематично представяне на реализацията на *Алгоритъм-M*

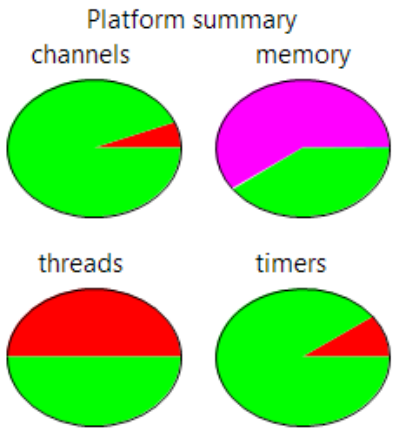
Алгоритъм-M (т. 1.1) е предложен от Кнут като вариант за стохастическо обединение на последователностите, изработвани от два различни генератора. Схематичното представяне на предложената реализация на алгоритъма е показано на фиг. 2.5. На фигурата се предполага, че инициализацията на вектора `uintDelay[]` вече е извършена.

Алгоритъмът използва два генератора, обозначени условно като *RNG1()* и *RNG2()*, и вектора `uintDelay[]`. В рандомизиращата функция `Randomize()` векторът `uintDelay[]` се инициализира чрез *RNG1()*.

Генераторната функция `RNG_ALG_M()` следва четирите стъпки от фигурата. На първата стъпка, чрез обръщение към *RNG2()* се формира индекса за достъп до вектора. По този индекс на втора стъпка се извлича елемента от вектора. Той отговаря на генерираното *32* bit число. На трета и четвърта стъпка чрез *RNG1()* се изработва нова случайна стойност, която замества извлечената от вектора. Основното изискване е размерността n на вектора да не е по-малка от *8192* (2^{13}). С други думи, основният недостатък на метода е големия обем памет, който му е нужен.

Функцията `RNG_ALG_M()` връща *32* bit от случайната последователност, за разлика от алгоритмите с изместващи регистри. Като пример, за *RNG1()* в реализацията се използва библиотечната функция *rand()*, а за *RNG2()* – *LFSR* генератор.

Необходимите за предложената реализация апаратни ресурси се оценяват със средствата на развойната среда *XDE* (фиг. 2.6). От наличните осем апаратни нишки се заемат четири. Използва се един от десетте таймера. За канала между *P* и *Q* са нужни две от общо *32* крайни точки. Заема се около *60*% от наличната в чипа *SRAM* памет.



Фиг. 2.6. Диаграма на използваните апаратни ресурси