VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

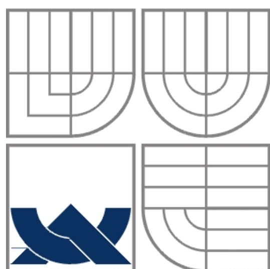
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENEROVANIE PSEUDONÁHODNÝCH ČÍSEL V FPGA

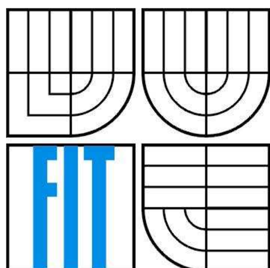
BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE PAVOL KORČEK
AUTHOR

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

GENEROVÁNÍ PSEUDONÁHODNÝCH ČÍSEL V FPGA

PSEUDORANDOM NUMBER GENERATION IN FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE PAVOL KORČEK
AUTHOR

VEDOUCÍ PRÁCE DOC. ING. LUKÁŠ SEKANINA, PH.D.
SUPERVISOR

BRNO 2007

Zadanie

1. Oboznámte sa s problematikou generovania náhodných čísel (najmä v hardvéri).
2. Oboznámte sa s návrhovým systémom pre FPGA a výukovou platformou obsahujúcou FPGA.
3. V jazyku VHDL implementujte zvolené generátory.
4. Overte činnosť obvodu implementáciou v FPGA.
5. Štatisticky vyhodnoťte kvalitu vytvorených generátorov.
6. Zhrňte dosiahnuté výsledky.

Licenčná zmluva

Licenčná zmluva je uložená v archíve Fakulty informačných technológií Vysokého učení technického v Brne.

Abstrakt

V tejto bakalárskej práci boli prebrané možnosti implementácie hardvérových generátorov pseudonáhodných čísel. Konkrétne pojednáva o dvoch najpoužívanejších spôsoboch generovania pseudonáhodných čísel v číslicových systémoch založených na princípe lineárneho spätnoväzbového registra (LFSR) a systéme založenom na celulárnych automatoch (CA). Z každej skupiny boli navrhnuté a v jazyku C popísané dva vhodné modely, ktoré sa v praxi najčastejšie používajú. Taktiež bolo takto implementované i zaujímavé kombinačné zapojenie LFSR, ktoré sa v praxi taktiež občas používa. Implementované generátory boli ohodnotené pomocou vysoko kvalitnej sady štatistických testov Diehard. Ďalšia časť práce spočívala v návrhu a implementácií vybraných generátorov jazykom popisujúcim hardvér. Týmto bol VHDL. Pomocou neho bolo vytvorených niekoľko modelov. Ide najmä o typy sériových a paralelných LFSR. Napokon bol sériový LFSR implementovaný i do výukovej platformy FITKit. Práca sa taktiež zaoberá implementáciou generátorov do hardvéru s ohľadom na veľkosť zabranej plochy.

Kľúčové slová

Pseudonáhodné čísla, Lineárny spätnoväzbový generátor, Celulárny automat, Diehard, FPGA, FITKit

Abstract

This bachelor thesis analyzes various implementations of pseudorandom number generators. In particular, two most-widely used mechanisms in generating the pseudorandom numbers in digital systems based on linear feedback shift register (LFSR) and cellular automata (CA) are described. Two models which are also widely used in practice were chosen from each group and implemented in C language. Additionally, another interesting combinatorial scheme of LFSR which is also sometimes used was implemented. Evaluation of the generators using Diehard set of statistical test was performed as well. Another part of this work dealt with implementing the chosen generators in a hardware description language. The choice was made for VHDL and several models, including the serial and parallel type of LFSR, were described in this language. Finally, a serial type of LFSR was implemented on the educational platform FITKit. The demands for area consumption of implemented generators were also investigated.

Keywords

Pseudorandom numbers, Linear feedback shift register, Cellular automata, Diehard, FPGA, FITKit

Citácia

Korček, P.: Generovanie pseudonáhodných čísel v FPGA, bakalárska práca, Brno, FIT VUT v Brne, 2007.

Generovanie pseudonáhodných čísel v FPGA

Prehlásenie

Čestne prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením doc. Ing. Lukáša Sekaniny, Ph.D., a že som uviedol všetky literárne pramene a publikácie, z ktorých som v priebehu svojej práce čerpal.

……………………
Pavol Korček
24.apríla 2007

Poďakovanie

Na tomto mieste by som rád poďakoval doc. Ing. Lukášovi Sekaninovi, Ph.D. za jeho odbornú pomoc a veľmi cenné rady pri spracovaní tejto práce.
Taktiež by som chcel poďakoval celému ústavu UPSY na Fakulte informačných technológií Vysokého učení technického v Brne za organizovanie odborných seminárov a za ochotu pomôcť pri riešení rôznych problémov súvisiacich s touto prácou.

© Pavol Korček, 2007.
Táto práca vznikla ako školské dielo na Vysokom učení technickom v Brne, Fakulte informačných technológií. Práca je chránená autorským zákonom a jej použitie bez udelenia oprávnení autorom je nezákonné, s výnimkou zákonom definovaných prípadov.

Obsah

1 Úvod ... 2

2 Generovanie pseudonáhodných čísel ... 3

2.1 Úvodom ... 3

2.2 Typy generátorov ... 3

2.2.1 Softvérová realizácia ... 4

2.2.2 Hardvérová realizácia ... 6

2.3 Transformácie rozložení ... 15

2.3.1 Metóda inverznej transformácie ... 16

2.3.2 Vylučovacia metóda ... 16

2.3.3 Kombinovanie predchádzajúcich prístupov ... 16

3 Testovanie generátorov ... 17

3.1 Úvodom ... 17

3.2 Štatistické testovanie ... 17

3.2.1 Testy *Diehard* ... 18

3.3 Iné spôsoby testovania ... 21

3.4 Vlastné testovanie ... 21

3.4.1 Implementácia zvolených generátorov v *jazyku C* ... 22

3.4.2 Spôsob testovania ... 22

3.4.3 Výsledky testov ... 23

4 Implementácia zvolených generátorov ... 25

4.1 Úvodom ... 25

4.2 Návrh číslicových systémov ... 25

4.2.1 Jazyk *VHDL* pri popise hardvéru ... 26

4.2.2 Vlastná implementácia ... 27

4.3 FITKit ... 27

4.3.1 Popis kitu ... 27

4.3.2 Prepojovací systém FITKitu ... 31

4.3.3 Generátor pseudonáhodných čísel pre FITKit ... 33

5 Syntéza generátorov ... 36

5.1 Úvodom ... 36

5.2 Výsledky syntézy pre zvolené implementácie ... 36

6 Záver ... 38

Literatúra ... 39

Zoznam príloh ... 42

1 Úvod

Potreba vyvíjať rôzne typy generátorov pseudonáhodných čísel sa odvíja od ich veľkého použitia v praxi. Príkladom tradičného využitia je hra s hracími kockami. Obecne to však nie sú len hry, kde sa náhodné čísla hojne používajú. Mnoho vedeckých oblastí využíva takýchto čísel na modelovanie nedeterministických a teda nepredpovedateľných dejov, ktoré nás v bežnom živote obklopujú. Tieto deje aby boli naozaj nepredpovedateľné potrebujú skutočne kvalitné generátory, ktoré musia byť pred použitím vždy kvalitne otestované a analyzované.

Táto práca sa zaoberá obecným prehľadom, návrhom, štatistickým testovaním a samotnou implementáciou niekoľkých najpoužívanejších typov hardvérových generátorov pseudonáhodných čísel. Testované generátory sú vyberané či už podľa toho ako často sa daný typ v praxi používa alebo taktiež s ohľadom na niektoré vlastnosti akou okrem tých štatistických môže byť zložitosť, rýchlosť či plocha, ktorú by výsledný obvod po návrhu zabral. Ako cieľovou architektúrou je podľa zadania rekonfigurovateľné hradlové pole FPGA (*Field Programmable Gate Array*), ktoré je súčasťou moderného výukového kitu (*FITKit*) [7] dostupného na Fakulte informačných technológií Vysokého učení technického v Brne ako učebná pomôcka pre rôzne z vyučovaných seminárov.

Štruktúra tejto práce je nasledujúca. Druhá kapitola popisuje akým spôsobom sa obecne generujú pseudonáhodné čísla v softvéri a v hardvéri. Pozornosť je tu podrobnejšie smerovaná samozrejme na generátory hardvérové. Na koniec tejto kapitoly sú spomenuté možnosti transformácií rozložení vygenerovaných čísel, ktoré dostaneme z generátorov. Číselné postupnosti, ktoré získavame, sú totiž zväčša rovnomerne rozložené na určitom intervale a to nám nie vždy a za každých okolností môže vyhovovať. Ďalšia z kapitol popisuje štatistické testovanie generátorov a dôkladnejšie popíše jeden z najpoužívanejších testovacích systémov vôbec. Taktiež sú v tejto časti spomenuté iné spôsoby testovania, respektíve vlastnosti, na ktoré pri návrhu kvalitného generátora nesmieme zabudnúť. Zvyšná časť kapitoly sa venuje vlastnému testovaniu vybraných generátorov a zhodnoteniu výsledkov testovania. Nasledujúca kapitola je venovaná implementácií generátorov do programovateľnej logiky a možnosťami využitia niektorých vlastností samotného programovateľného poľa. Taktiež je tu možné sa dočítať o už spomínanom kite a o problémoch pri vlastnej implementácií generátora pseudonáhodných čísel. Predposledná kapitola zhrňuje výsledky syntézy pre rôzne typy generátorov pseudonáhodných čísiel.

2 Generovanie pseudonáhodných čísel

2.1 Úvodom

Náhodnosť a náhodné čísla boli a sú využívané pre rôzne účely [6]. S príchodom počítačov ale inžinieri zistili, že je potrebné zaviesť prostriedky, ktorými dostanú náhodnosť i do počítačových programov. Prekvapením pre niekoho však môže byť fakt, že nie je jednoduché donútiť počítač, aby robil niečo náhodne. Počítač iba nasleduje a postupne vykonáva inštrukcie v presne danom poradí, a to znamená že je jeho činnosť úplne predvídateľná.

Počítačoví experti sa preto rozhodli zaviesť náhodnosť do počítačov vo forme pseudonáhodných číselných generátorov. Ako je už z názvu vidieť, pseudonáhodné čísla nie sú skutočne náhodné. Sú získané z matematickej formule alebo jednoducho vzaté z dopredu prepočítaného zoznamu. Výsledkom dlhého výskumu pseudonáhodných čísel a moderných algoritmov pre ich generovanie je to, že čísla vygenerované takýmito spôsobmi vyzerajú akoby boli skutočne náhodné. Základnou charakteristikou pseudonáhodných čísel, oproti tým skutočne náhodným je to, že sú predvídateľné. To znamená, že môžu byť predpovedané, ak vieme akým spôsobom sme získali prvé číslo z danej sekvencie. Pre niektoré účely môže byť predvídateľnosť dobrou vlastnosťou, no pre iné kritické aplikácie však zásadne nevhodnou alebo dokonca nebezpečnou.

Kapitola popisuje niekoľko druhov generátorov podľa toho najzákladnejšieho delenia, a to na generátory softvérové a hardvérové. Je uvedený ich princíp a značenie, tak ako aj možná softvérová realizácia, či náčrt obvodovej realizácie hardvérového typu. Nakoniec spomeniem možnosti, akými je možné previesť čísla generované v rovnomernom rozložení do rozložení iných, nami zvolených.

2.2 Typy generátorov

Generátorov pseudonáhodných čísel existuje veľmi mnoho. Pokiaľ ich chceme rozdeľovať podľa nejakých kritérií, tak tou základnou by mala určite byť oblasť použitia. Je totiž vždy veľmi dôležité k akému účelu bude vytvorený generátor slúžiť. Nie každej oblasti môže vytvorený generátor vyhovovať. Taktiež nemusí vyhovovať ani generátor v praxi používaný a testovaný, keďže mohol byť vyvíjaný na špecifickom vybavení.

Najzákladnejším rozdelením použitia je ohľad na to, či ide o použitie generátora v softvéri alebo v hardvéri. Podľa tohto sa môžu odvíjať i ďalšie špeciálne požiadavky naň kladené. U tých softvérových to môže byť napríklad požiadavka na rýchlosť, veľkú periódu a kvalitu, ktorá sa považuje akoby za implicitnú vlastnosť všetkých generátorov pseudonáhodných čísel. U tých hardvérových zas môže ako hlavné kritérium (okrem už spomínaných vlastností) byť veľkosť, a teda

u návrhu použitého v tejto práci, veľkosť zabraného miesta na čipe. Podľa toho môžeme napríklad rozdeliť tieto generátory na veľké alebo kompaktné. Častokrát sa ale, ako je v tejto práci neskôr vidieť, niektoré vlastnosti navzájom ovplyvňujú. V hardvéri je napríklad problémom vytvoriť kompaktný generátor, ktorý by produkoval štatisticky lepšie výsledky ako iný, robustnejší.

2.2.1 Softvérová realizácia

Pre generovanie pseudonáhodných čísel na číslicovom počítači existuje celá rada rôznych algoritmov. Najčastejšie používané generátory využívajú princíp *lineárneho kongruentného generátora* [18]:

$$x_{i+1} = (ax_i + b) \mod m$$

kde

- operácia *mod* znamená zvyšok po celočíselnom delení;
- *a, b* a *m* sú nami vhodne zvolené konštanty.

Tento generátor generuje čísla s rovnomerným rozložením v rozsahu $0 \le x_i < m$. Pre prevod na požadovaný rozsah $< 0, 1)$ musíme výsledok x_{i+1} deliť modulom *m*. Pri počiatočnom nastavení x_0 (v anglickej literatúre nájdeme termín *„seed"*) každé použitie uvedeného výrazu generuje ďalšie číslo. Keďže však počet možných hodnôt v intervale $0 \le x_i < m$ je obmedzený, začne sa pri vhodne zvolených konštantách)najneskôr po *m* generovaných číslach opakovať rovnaká postupnosť. To znamená, že generátor má určitú *periódu generátora.*

Pre dosiahnutie potrebných štatistických vlastností výslednej postupnosti je nutná voľba vhodných konštánt *a, b, m*. Pre ich voľbu však existujú pravidlá [18]. Modul *m* je vhodné voliť tak, aby mal hodnotu 2^n, kde *n* je počet bitov čísel typu *unsigned integer*. Potom nemusíme vykonávať operáciu modulo, pretože ta je implicitnou vlastnosťou celočíselných výpočtov na počítači. Týmto sa generovanie zrýchli, pretože operácia celočíselného delenia je obecne časovo náročná.

V literatúre obvykle nájdeme vhodne zvolené konštanty, ktoré boli už testované. Niektoré z možností sú uvedené v nasledujúcej tabuľke :

Zdroj	a	b	m
ANSI C	1 103 515 245	12 345	2^{31}
Numerical Recipes	1 664 525	1 013 904 223	2^{32}
RAND	69 069	1	2^{32}
Park & Miller	16 807	0	$2^{31} - 1$

Tabuľka 1 : *Konštanty používané v praxi.*

Pri použití jednoduchých kongruentných generátorov musíme mať na pamäti aj ich negatívne vlastnosti. Pri nevhodnej voľbe parametrov dôjde v výraznému zhoršeniu štatistických vlastností generátora. Veľkým problémom kongruentných generátorov je taktiež závislosť po sebe idúcich generovaných čísel. Toto sa prejavuje a je nevhodné najmä pri generovaní *n-tic* náhodných čísel. Preto tieto jednoduché generátory nie sú vhodné pre ozaj náročné použitia.

Kongruentný generátor generuje celé čísla v rozsahu $0 \le x_i < m$ s rovnomerným rozložením. Pretože často potrebujeme tzv. normalizované rozloženie s rozsahom $< 0, 1)$ musíme použiť operáciu delenia modulom m a výsledok uložiť ako číslo v plávajúcej čiarke. Napríklad v *ISO C* môžeme použiť štandardnú funkciu `rand`, ktorá generuje pseudonáhodné čísla v rozsahu 0...`RAND_MAX`. Pokiaľ potrebujeme generovať náhodné celé čísla, je potrebné postupovať veľmi opatrne, pretože kongruentné generátory majú málo náhodné najmenej významné bity generovaných čísel. Napríklad:

```
i = 1 + ( rand() % 10 )
```

generuje celé čísla v intervalu 1...10 ale so zlými charakteristikami. Preto je vhodnejšie použiť:

```
i = 1 + 10 * ( rand()/(RAND_MAX+1.0) )
```

Príkladom jednoduchého generátora zapísaného v jazyku C je napríklad [33]:

```
unsigned int x, a, b;

void Reset()
{
     x = 0;
     a = 69069;
     b = 1;
}

unsigned int GenerateNext()
{
     x = x * a + b;
     return x;
}
```

Ako vidieť, implementácia je veľmi jednoduchá. Predpokladáme, že výpočet prebieha s presnosťou na *32 bitov*. Tento generátor nepatrí síce k najlepším, avšak je rýchly a má periódu dlhú až 2^{32}.

Existujú generátory, ktoré kombinujú princíp kongruentného generátora s ďalšími operáciami a dosahujú tak výrazne lepších výsledkov. Pre náročné použitia je možné použiť generátor *Mersenne Twister* [23] ktorý je veľmi rýchly a kvalitný alebo už menej známe *Blum Blum Shub* [26] a generátor *Fortuna* [29].

2.2.2 Hardvérová realizácia

Na rozdiel od softvérových realizácií generátorov máme u tých hardvérových viacero možností prístupu. Technológiu generovania pseudonáhodných čísel v hardvéri môžeme rozdeliť do troch hlavných oblastí:

Generátory s využitím analógových prostriedkov využívajú toho, že je možné do hardvéru implementovať fyzikálny proces. Takýmto náhodným fyzikálnym procesom môže byť napríklad *diódový šum*. Generátory tohto typu obsahujú častokrát *zosilňovače* a *analógovo-digitálne* prevodníky pre prevod analógového údaja na digitálny.

Generátory založené na čisto digitálnom spracovaní sú generátory podrobne popisované v práci ďalej. Ide o implementáciu určitého algoritmu do číslicového obvodu. Oproti predchádzajúcim majú nevýhodu v tom, že sa generované číselné postupnosti menej približujú k tým skutočne náhodným, avšak výhodou týchto obvodov je ich podstatne väčšia rýchlosť generovania. Niektoré typy týchto generátory spájajú analógový princíp generovanie spolu s číslicovým generovaním a vzniká tak kombinovaný prístup.

Kombinácia predchádzajúcich prístupov je využívaná napríklad tak, že sú číslicové generátory inicializované hodnotami vygenerovanými náhodnými analógovými javmi. Zaujímavú kombináciu týchto prístupu využili v práci [5], kde použili javu v odbornej terminológii označovaného ako *jitter*. Generátory tohto typu sú taktiež ako tie analógové využívané iba málokedy, skôr na experimenty [10,22].

Najviac sú rozšírené *generátory číslicové*. Je to pre ich jednoduchosť návrhu, ďalej pre dostatočnú rýchlosť, no najmä preto, že fyzikálne procesy (a ich prevod do číslicovej podoby) nie je až tak jednoduché implementovať.

Oblasti využitia hardvérových generátorov sú naozaj rôzne. Odvíjajú sa ale častokrát od samotného použitia pseudonáhodných čísel a postupností v daných oblastiach. Ako príkladom môžem uviesť:

- šifrovanie a dešifrovanie dát (tvorba kryptografických kľúčov),
- bezdátové komunikácie (Wi-fi),
- počítačové siete (CDMA, W-CDMA),
- vstavané testy zariadení (BITS),
- kontrolné súčty dát (kontrola integrity),
- kompresia dát,
- kódery a dekodéry a
- čítače.

Najčastejšie sa je možné stretnúť s číslicovými hardvérovými generátormi založenými na princípe lineárneho spätnoväzbového registra a s jeho rôznymi variáciami. Je jednoduchý a rýchly. Už menej používané sú zložitejšie systémy založené na celulárnych automatoch.

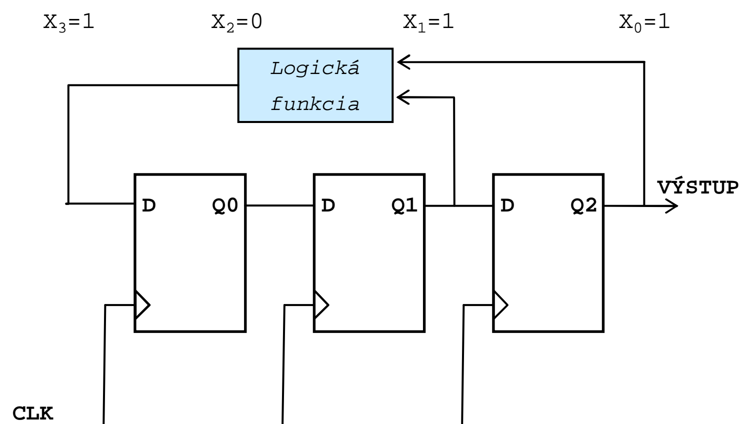
2.2.2.1 Linear Feedback Shift Register

Linear feedback shift register (*LFSR*), alebo ako niektoré domáce zdroje literatúry uvádzajú lineárny spätnoväzbový posuvný register je posuvný register, tvorený klopnými obvodmi typu *D*, ktorý využíva spätnej väzby na to, aby sám seba modifikoval pri každej zmene hodinového signálu. Spätná väzba je tu teda využívaná k tomu, aby register mohol v nasledujúcom cykle získať nový, unikátny bitový stav. Voľba dĺžky registra LFSR, typu logickej funkcie, štruktúry LFSR a pozícií odberu spätnej väzby (v anglickej literatúre nájdeme termín *„taps“*) nám dovoľujú kontrolovať implementáciu generátorov založených na tomto princípe.

LFSR môže byť vytvorený použitím *Galoisovej* (vnútornej, internej) alebo *Fibonacciho* (vonkajšej, externej) schémy alebo konfigurácie hradiel a registrov [35].

Fibonacciho schéma

Vo Fibonacciho schéme slúžia výstupy z niekoľkých registrov i ako vstup logickej funkcie, pomocou ktorej je späť do registra vkladaná nová hodnota (*spätná väzba, feedback*) pri posuve. Obrázok 1 ukazuje 3-bitové Fibonacciho LFSR:



Obrázok 1: ***3-bitové Fibonacciho zapojenie LFSR.***

Pokiaľ do posuvného registra vložíme novú inicializačnú hodnotu (seed) a na vstup hodinového signálu je privedený hodinový signál, výstup LFSR (*Q2*) bude produkovať pseudonáhodné sekvencie logických 1 a 0. Ako príklad sekvencie, ktorá je produkovaná obvodom na obrázku 1 pri inicializácii hodnotou „111“ s logickou funkciou *XOR,* ukazuje tabuľka 2:

Čas (t)	Q2	Q1	Q0
t_0	1	1	1
t_1	1	1	0
t_2	1	0	0
t_3	0	0	1
t_4	0	1	0
t_5	1	0	1
t_6	0	1	1
t_7	1	1	1

Tabuľka 2: ***LFSR sekvencia pri inicializácii hodnotou „111" a s logickou funkciou XOR.***

Pseudonáhodná sekvencia môže byť taktiež generovaná pomocou logickej funkcie *XNOR*. Od toho či použijeme ako logickú funkciu *XOR* alebo *XNOR*, ktoré sa v praxi najčastejšie využívajú[1], závisí aj problém s inicializačnou hodnotou. Normálne môže mať LFSR 2^{k-1} jedinečných stavov v jednej sekvencii (kde číslo *k* značí počet bitov LFSR). Napriek tomu, LFSR je obyčajný register, ktorý ale dokáže uchovávať v sebe až 2^k hodnôt. Pokiaľ LFSR s logickou funkciou *XOR* prechádza svojimi stavmi, potom jediný stav, ktorý nebude prítomný, je stav samých logických 0 (v našom prípade „000"). Pomocou *XOR* hradla vo spätnej väzbe LFSR nie je schopný vygenerovať stav samých logických 0. Pokiaľ by naopak register náhodou obsahoval samé „0" nie je schopný sa už z tohto stavu nikdy dostať. Tomuto stavu sa obyčajne hovorí *zakázaný*, keďže LFSR doň nemôže vstúpiť ani sa z neho dostať. Pre LFSR s logickou funkciou *XNOR* v spätnej väzbe je *zakázaným stavom* stav samých logických 1.

Dĺžky pseudonáhodných sekvencií tvorené s použitím logickej funkcie *XOR* alebo *XNOR* sú vždy rovnaké [35]. Táto dĺžka závisí na dĺžke posuvného registra a počte a pozícií odberových miest (*taps*). Počet a pozícia týchto miest je obyčajne reprezentovaná polynómom. Príkladom polynómu použitého v mnou uvedenom príklade, ktorého značenie využíva aj *Xilinx* [35] je:

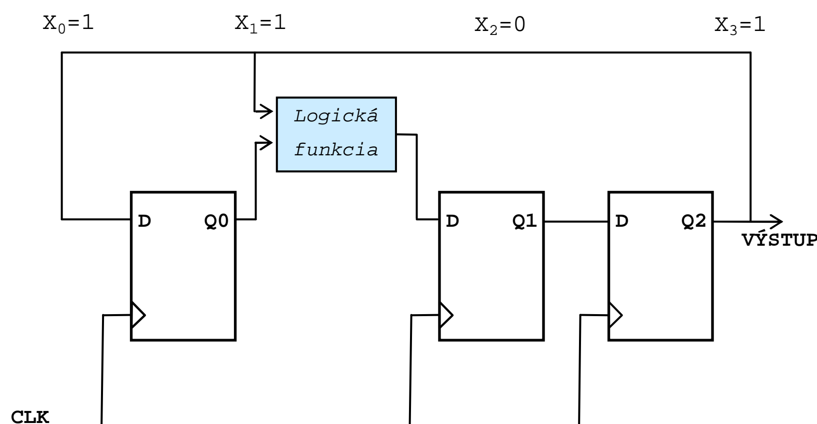
$$P(X) = X^3 + X^1 + 1$$

Viac informácií ohľadne polynómov a ich význame je možné sa dozvedieť v ďalšom texte.

Galoisovo schéma

Pri Galoisovej schéme sú hradlá logických funkcií umiestnené medzi registrami. Obrázok 2 ukazuje implementáciu takéhoto LFSR z predchádzajúceho príkladu. Konfigurácia produkuje sekvenciu, ktorá je uvedená v tabuľke 2. Ako môžme vidieť, výstupná sekvencia *(Q2)* je rovnaká ako v predchádzajúcom prípade, avšak s posunutím o jeden cyklus.

[1] Obecne nemusí tvoriť logickú funkciu iba hradlo typu *XOR* a *XNOR*.



Obrázok 2: ***3-bitové Galoisovo zapojenie LFSR.***

Čas (t)	Q2	Q1	Q0
t_0	1	1	1
t_1	1	0	1
t_2	0	0	1
t_3	0	1	0
t_4	1	0	0
t_5	0	1	1
t_6	1	1	0
t_7	1	1	1

Tabuľka 3: ***LFSR sekvencia pri inicializácii hodnotou „111" a s logickou funkciou XOR.***

Opäť i u tohto typu implementácie závisí dĺžka pseudonáhodnej sekvencie na veľkosti registra a počte a pozícií odberových miest pre logickú funkciu. Tento počet a pozícia môže byť opäť reprezentovaná ako polynóm. V našom prípade:

$$P(X) = 1 + X^1 + X^3$$

Na záver je treba dodať, že Galoisovo schéma sa v praxi skoro vôbec nevyužíva, keďže jeho implementáciou dostaneme pomalší a priestorovo rozmernejší [2,9,16] generátor inak rovnakých parametrov ako použitím Fibonacciho schémy. V ďalšom sa teda budeme sústrediť na Fibonacciho konfiguráciu.

Notácia zápisu pomocou polynómu

Notácia zápisu pomocou polynómu je obyčajne používaná k popisu funkcionality LFSR. Je potrebné však podotknúť, že táto konvencia reprezentácie pomocou polynómu prináša trochu zmätok v reprezentácií LFSR. Ako bolo spomenuté, notáciu ktorá bude popísaná využíva i *Xilinx* a rozlišuje medzi Fibonacciho a Galoisovou schémou. Túto notáciu taktiež používa *X-CDMA* špecifikácia *3G TS 25.213* [35].

Obrázok 2 zobrazuje príklad Galoisovho LFSR dĺžky 3 s $X_0 = 1$, $X_1 = 1$, $X_2 = 0$, $X_3 = 1$. Príklad na tomto obrázku reprezentuje polynóm:

$$P(X) = X^0 + X^1 + X^3$$

Zo zápisu a z obrázku je vidieť miesta, z ktorých je odoberaný signál (taps). Samozrejme, že je možné ako obvykle značiť namiesto X^0 iba číslo 1. Takýto polynóm môže byť ale zapísaný i vo forme reťazca bitov, kde každý z bitov korešponduje priamo s koeficientom v polynóme. Výraz najvyššieho rádu zodpovedá najvýznamnejšiemu bitu v reťazci *(MSB)* a naopak, výraz X^0 zodpovedá najmenej významnému bitu *(LSB)*. Je treba poznamenať, že jediný najvyšší výraz bude mať vždy koeficient 1, keďže z princípu musí byť vždy pripojený. Potom ho v bitovej postupnosti nie je potrebné zapisovať. Ak napríklad $p(x) = X_3 + X_1 + X_0$, tak môžeme písať reťazec „011" kde $1.X_3$ je implikované, nasleduje $0.X_2$, $1.X_1$ a $1.X_0$.

Systém Fibonacciho notácie je naznačený na obrázku 1. Je tu príklad 3 bitového LFSR dĺžky 3 s $X_0 = 1$, $X_1 = 1$, $X_2 = 0$ a $X_3 = 1$. Polynóm má tvar:

$$P(X) = X^3 + X^1 + X^0$$

Opäť namiesto X^0 píšeme často iba číslo 1. Význam polynómu je rovnaký ako u predchádzajúceho zapojenia.

Porovnaním oboch zápisov zistíme, že najvyšší faktor v polynóme určí bitovú veľkosť registra LFSR (v tomto prípade 3). Oba typy polynómov reprezentujú taktiež miesta odberu signálu pre logické funkcie, ibaže v Galoisovom type sú čísla indexov registrov zvyšované v smere posuvu registra a u Fibonacciho typu je to naopak.

Voľba vhodného polynómu

Nakoniec treba spomenúť to, že voľba polynómu, ktorý vytvára vnútornú štruktúru LFSR nie je ľahkovážnou záležitosťou. Nie každý polynóm môže pre nás vytvárať generátor, ktorý bude mať vhodnú štruktúru výsledného obvodu. Ak budeme chcieť vytvoriť napríklad 128 bitový generátor, nebude určite vhodné, keď pozícia odberu signálov bude čoby len na každom druhom z registrov. Preto hľadáme polynómy s najnižším možným počtom členov.

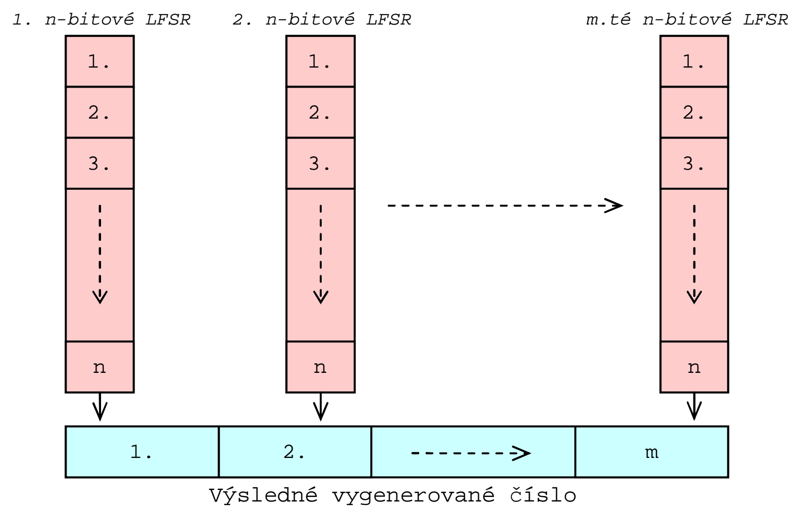
Ďalšiu vlastnosť, ktorú ovplyvňuje generujúci polynóm je dĺžka periódy. Ta by mala byť obecne čo najväčšia, a preto sme nútený použiť polynómy primitívne[1], teda ďalej neredukovateľné [24,11].

V práci boli vybrané polynómy, ktoré odporúčajú i mnohé zdroje [35,2,14] pre použitie s 32 bitovými generátormi. Súpis všetkých tých najvhodnejších polynómov je v *prílohe A*. Ostatné polynómy je možné generovať programovo.

[1] Polynóm je neredukovateľný v *GF(2)* pokiaľ nemôže byť ďalej rozdelený na netriviálne polynómy. Napr.: x^2+x+1 je neredukovateľný, ale x^2+1 už nie, keďže $x^2+1 = (x+1).(x+1)$.Viac napríklad v [12].

2.2.2.2 Paralelné štruktúry LFSR

Sú uvedené v samostatnej kapitole, keďže o ich využití sa píše len málo i napriek tomu, že majú, ako je možné neskôr vidieť, vynikajúce výsledky v testoch. Ide v podstate len o spojenie viacerých jednoduchých LFSR tak, že z každého z nich sa odoberá výstup, ktorý tvorí časť novo generovaného čísla. V tomto prípade potom potrebujeme na *m* bitové číslo *m* samostatných registrov LFSR:



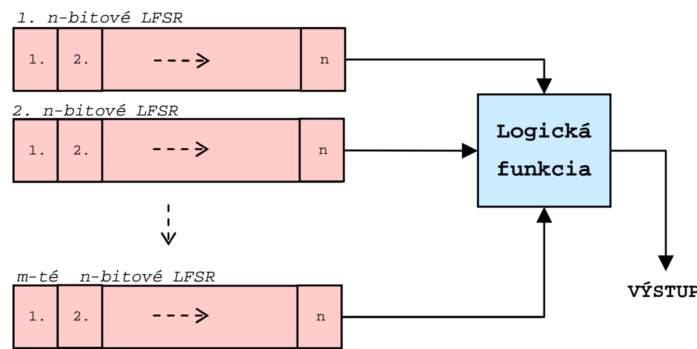
Obrázok 3: ***Paralelné zapojenie m n-bitových LFSR.***

U takto vytvoreného generátora však nastávajú nové problémy. Pri zhodných vnútorných štruktúrach jednotlivých registrov a pri rovnakej počiatočnej hodnote vytvoríme generátor, ktorý generuje síce náhodne, ale len samé postupnosti logických 1 alebo logických 0. Obecne máme ale na výber jednu z dvoch možností. Buď každý z LFSR inicializujeme samostatne, a v tomto prípade by potom ich štruktúra mohla byť rovnaká, alebo pozmeníme štruktúru každého z nich. Pri inicializácií inými hodnotami potrebujeme vždy toľko počiatočných hodnôt koľko bitový máme výsledný register pre uloženie generovaného čísla. Keby sme však zmenili štruktúry jednotlivých registrov, máme možnosť použiť pre všetky rovnakú počiatočnú hodnotu. Musíme však dávať pozor na to, že je k dispozícií len obmedzený počet polynómov, ktoré vytvárajú generátor s vhodnými vlastnosťami. A aj keď ich ale nájdeme dostatok, môžeme naraziť na problém, že je polynóm zložený z veľa členov, čo vedie k obmedzovaniu či už v návrhu alebo iných vlastností generátora.

Čo sa týka dĺžky periódy takto vytvoreného generátora, tak tá je zhodná s dĺžkou periódy toho LFSR, ktoré má periódu najväčšiu. Ale iba v tom prípade, ak sú veľkosti registrov voči sebe násobkom. Pokiaľ nie sú, nemusíme generovať už rovnomerné rozloženie [16]. Je možné teda vytvoriť aj 32 bitový generátor, ktorý sa bude skladať s rovnakého počtu napr. 16 bitových LFSR. Potom by perióda nebola 2^{32} ale iba 2^{16}. Ako je vidieť, rozsah generovaných čísel už nie je 2^{k-1} ale 2^k, pretože paralelná štruktúra môže generovať aj také čísla, kde sa vyskytujú samé „0“, či samé „1“.

2.2.2.3 Kombinované štruktúry LFSR

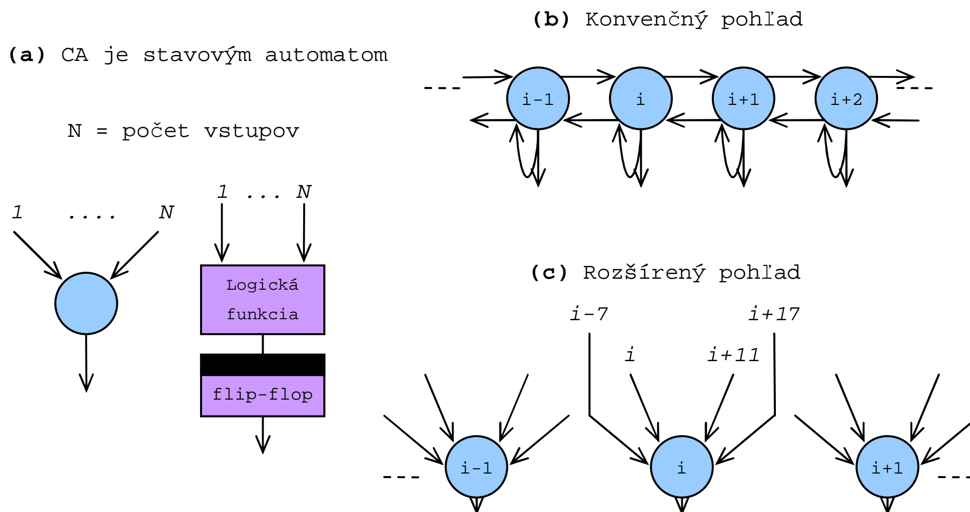
Ako ďalší spôsob zapojenia kombinovaním niekoľkých LFSR je zapojenie uvedené na obrázku 4. Najčastejšie sa pri tomto prístupe používa do 5 osobitných LFSR, ktorých výstupy sú privedené k vstupu logickej funkcie (opäť najčastejšie *XOR* či *XNOR*). Výstupom logickej funkcie je potom sériová bitová pseudonáhodná postupnosť. Veľkosť jej periódy je opäť závislá na veľkosti registrov, obdobne ako v predchádzajúcom prípade. Výhodou oproti čisto paralelnému zapojeniu máme v tom, že je možné pre tak malý počet registrov nájsť dostatok vhodných polynómov, ktorými vytvoríme vnútornú štruktúru registrov. I keď sa môže zdať, že čisto sériový výstup je obmedzujúci, i takéto zapojenie našlo v praxi svoje uplatnenie [13]. V práci je použitý generátor (*comb-LFSR*) s piatimi 32-bitovými LFSR (*polynómy zvolené podľa priloženého programu*) a logickou funkciou typu *XOR*.



Obrázok 4 :*Kombinované zapojenie LFSR.*

2.2.2.4 Generátory založené na celulárnych automatoch

Celulárny automat (*CA*) môže byť chápaný ako dynamický systém, diskrétny v čase i priestore [34]. Klasický pohľad (Obrázok 5) na CA je ako pole buniek (v anglickej literatúre nájdeme termín „*cell*“) s homogénnou funkcionalitou obmedzené pravidelnou mriežkou určitej dimenzie. Najčastejšie sú tieto mriežky v tvare jednorozmerných reťazcov/kruhov alebo dvojrozmerných plátov/toroidov. Reťazce a pláty vznikajú výsledkom pevne stanovených koncových prepojení. Kruhy a toroidy vznikajú zas ako výsledok periodických prepojení koncov. Každá z buniek uchováva v sebe aktuálny stav, ktorý je upravovaný v každom časovom kroku funkciou lokálnych (prípadne nelokálnych) prepojení. Pre dvojstavový CA s počtom vstupov *N* existuje 2^{2^N} možných implementácií [34,1].

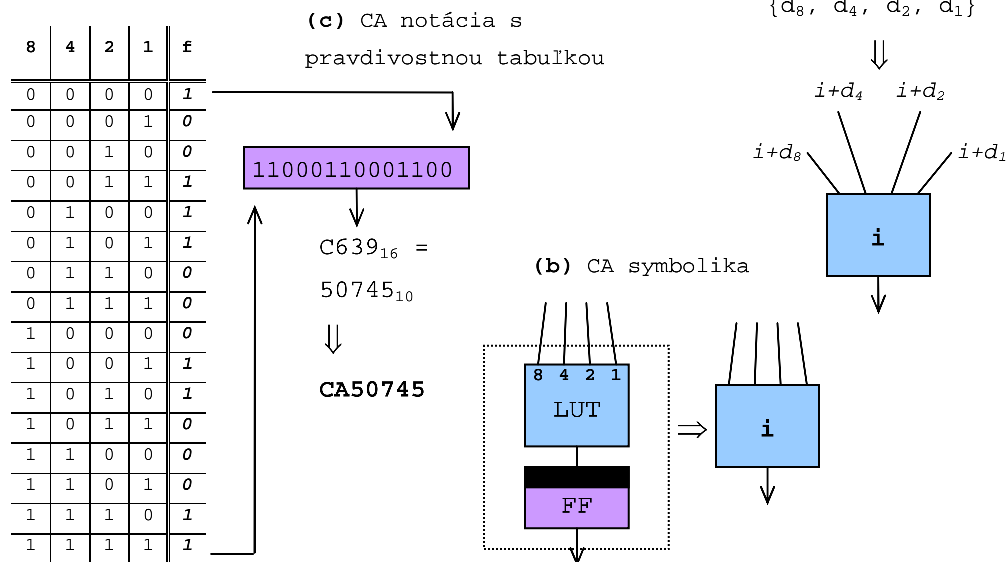


Obrázok 5: ***Celulárne automaty (CA) sú stavové automaty, kde N je definované ako počet vstupov. (b) Majú symetrické, lokálne prepojenia. (c) Dovoľujú ale i nesymetrické a nelokálne prepojenia.***

Funkcia jednej CA bunky môže byť popísaná pravdivostnou tabuľkou ako to zobrazuje obrázok 6a nižšie. Ako je vidieť na obrázku 6b, každej z buniek automatu zodpovedá jeden jednobitový register. Pokiaľ použijeme pre každú z buniek štyri vstupy, ktoré budú ovplyvňovať jej stav, tak existuje 16 rôznych podmienok, na ktoré bude bunka reagovať. Počet unikátnych reakcií bunky môže byť zobrazený ako 16 bitové číslo, ktoré zodpovedá 2^{16} unikátnym konfiguráciám. Na obrázku 6c je notácia [34], ktorá chápe toto číslo ako unikátnu identifikáciu funkcionality automatu. Táto notácia ale prestáva byť použiteľnou u nelokálnych prepojení s viacerými vstupmi ovplyvňujúcimi stav bunky. Pre *n* vstupov totiž generujeme 2^n bitové čísla, čo prináša obmedzenia v zobrazení takýchto konfigurácií s príliš veľkým *n*.

Jedným z možných riešení tohto problému je riešenie ako to zobrazuje obrázok 6d. V tomto prípade sa prepojenie nenaznačuje už pomocou pravdivostnej tabuľky ako v predošlom prípade, ale podľa relatívneho posunutia vzhľadom k určitej bunke.

Pomocou takto vytvoreného CA dokážeme vytvoriť generátor pseudonáhodných čísel tak, že generátor jednoducho inicializujeme hodnotou a v každom novom hodinovom takte nám paralelne vyprodukuje novú hodnotu. Nevýhodou generátorov tohto typu je však viditeľne vyššia zložitosť oproti LFSR a nejednoduché pátranie po vhodne vyplnenej pravdivostnej tabuľke, ktorá by mala implikovať čo najlepšie hodnoty.



Obrázok 6 : ***Funkcionalita CA vychádza priamo z pravdivostnej tabuľky.***
(b) Každá z buniek môže byť implementovaná ako 4-vstupá LUT s 1-bitovým registrom stavu.
(c) Notácia vychádza priamo z tabuľky. „0“ sú doplnené kvôli prevencii kolízie s 3 vstupnými CA.
(d) Pripojenie k bunke je naznačené ako relatívne pozície ostatných buniek.

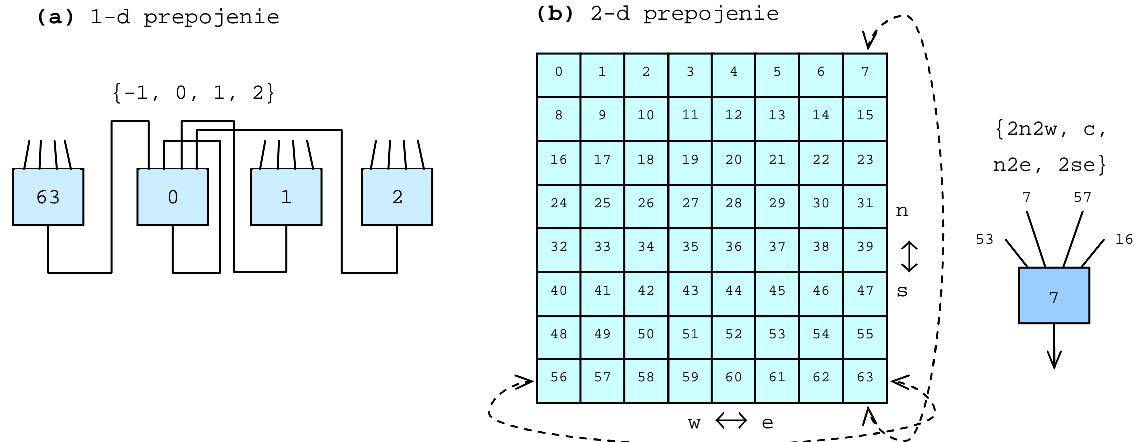
Voľba vhodného CA

U LFSR je možné niekoľkými danými postupmi dospieť k polynómu, ktorý nám bude generovať po jeho implementácií s LFSR sekvencie s tými najlepšími vlastnosťami. U implementácií s CA však narazíme na problém, ktorým je to, že neexistuje spoľahlivá metodika na určenie najlepšieho generátora založeného na tomto princípe. Najčastejšie sa generátory vytvárajú pokusmi alebo získavajú z predchádzajúcich výsledkov, čo sú ale zas iba predchádzajúce pokusy. Existuje však i niekoľko málo metód, ktorými je možné vlastnosti generátora aspoň odhadnúť [34].

V tejto práci boli ako generátory založené na CA vybrané tie konfigurácie, ktoré boli už aspoň čiastočne odskúšané a otestované. Bol vybraný jeden jednoduchší (*CA30*) a jeden z tých zložitých ale kvalitných (*CA50745*). Ich konfigurácia zodpovedá číselnému zápisu podľa predchádzajúceho textu.

2.2.2.5 Viacrozmerné a zložitejšie zapojenia CA

Systém zložitejšieho periodického prepojenia pre jednorozmernú a dvojrozmernú dimenziu zobrazuje názorne obrázok 7 ďalej.



Obrázok 7 : ***Možnosť 1-d prepojenia.***
(b) ***Ukážka 2-d prepojenia.***

Obrázok 7a ukazuje jednorozmerný kruh s relatívnym zapojením {-1, 0, 1, 2} z perspektívy bunky číslo 0. Z periodického prepojenia vyplýva, že bunka 0 a bunka 63 sú susedné, preto pokiaľ relatívna pozícia je -1, bunka 0 bude napojená práve na túto bunku 63.

Obrázok 7b ukazuje dvojdimenzionálnu sieť s periodickou väzbou, ktorá transformuje plát na povrch toroidu, kde každý riadok tvorí kruh a taktiež každý stĺpec tvorí kruh. U zapojenia bunky sa najčastejšie používa anglický kompasový systém. Na ukážku *2se* napríklad znamená pripojenie na bunku, ktorá je vzdialená 2 kroky smerom na juh ($\downarrow$) a jeden krok smerom na východ ($\rightarrow$). Obrázok 6b zobrazuje prepojenie implikované prepojením {2n2w, c, n2e, 2se} vždy vzhľadom k strednej bunke.

Generátory založené na princípe viacrozmerných CA s periodickými prepojeniami sú však natoľko zložité, že sa v praxi veľmi nevyužívajú i napriek tomu, že sú podľa testov celkom kvalitné [1]. Ich implementáciou v číslicovom obvode zistíme, že zložitejšie schémy zaberajú veľa miesta na čipe a sú v hardvéri i celkom pomalé [1] oproti predchádzajúcim prístupom. Ako sme sa ale presvedčili skôr, existujú jednoduchšie implementácie v praxi viac využívané (LFSR). Zaujímavou implementáciou generátora založeného na princípe celulárnych automatov je i lineárno-hybridná kombinácia [34].

2.3 Transformácie rozložení

Základom pre generovanie akéhokoľvek rozloženia je generátor čísel s rovnomerným rozložením na určitom intervale. Keď získame generátor, ktorý generuje čísla napríklad v intervale [0,1) rovnomerne, môžeme viacerými metódami previesť toto rozloženie na rozloženia iné (exponenciálne apod.) a taktiež, môžeme tento rozsah len niekoľkými jednoduchými aritmetickými operáciami previesť na rozsah iný. Navyše tieto metódy nie sú až tak zložité, aby ich nebolo možné

implementovať v hardvéri. Pokiaľ sú transformácie a postupy korektné, tak pri nich už nedochádza k zhoršovaniu štatistických vlastností výslednej číselnej postupnosti.

V tejto časti spomeniem len niekoľko možnosti akými spôsobmi je možné sa dopracovať k požadovanému rozloženiu. Pre generovanie niektorých rozložení však existujú iné metódy, ktoré sú vytvorené pre konkrétne rozloženie. Napríklad pre generovanie normálneho alebo Gaussového rozloženia sa využíva sčítavanie niekoľkých rovnomerne rozložených čísel [6].

2.3.1 Metóda inverznej transformácie

Metóda využíva toho, že vieme vypočítať, alebo sme nejakým spôsobom získali (napr. graficky), inverznú funkciu k distribučnej funkcii F požadovaného rozloženia. Obecne postupujeme tak, že si vypočítame inverznú funkciu F^{-1} a naším generátorom vygenerujeme číslo $(x_1, x_2, x_3,...)$ s rovnomerným rozložením na intervale $(0,1)$. Vygenerované číslo dosadíme do vypočítanej inverznej funkcie $(F^{-1}(x_1),F^{-1}(x_2),F^{-1}(x_3),...)$ a ako výsledok po výpočte získame číslo nové. Toto číslo už máme transformované do požadovaného rozloženia.

2.3.2 Vylučovacia metóda

Metóda pracuje na princípe náhodnej voľby bodu v ploche obdĺžnika, ktorý ohraničuje graf funkcie hustoty pravdepodobnosti daného rozloženia. Je teda založená na princípe generovania dvoch postupností s rovnomerným rozložením. Jedna z nich predstavuje potenciálne hodnoty výslednej postupnosti. Pomocou druhej z nich zaisťujeme požadované pravdepodobnostné rozloženie. Algoritmus pre generovanie náhodnej veličiny s funkciou hustoty $f: (x_1, x_2) \rightarrow (0, M)$ môžeme zapísať v troch krokoch:

- Generujeme číslo x z rovnomerného rozloženia (x_1, x_2).
- Generujeme číslo y z rovnomerného rozloženia $(0, M)$.
- *Ak je* $y < f(x)$, prehlásime x prvkom generovanej postupnosti, inak postup opakujeme.

Metóda vylučovacia nie je však vhodná pre rozloženia s neobmedzeným rozsahom, prípadne s veľmi malým pomerom plochy pod funkciou a plochy ohraničujúceho obdĺžnika. Potom by sme sa príliš často „netrafili“ a metóda by mohla byť veľmi neefektívna. Naopak výhodu má oproti predchádzajúcemu prístupu v tom, že je použiteľná i pre rozloženie kde nepoznáme alebo nedokážeme vypočítať inverznú funkciu distribučnej funkcie.

2.3.3 Kombinovanie predchádzajúcich prístupov

Pre niektoré rozloženia je možné použiť kombinácie (napríklad aj predchádzajúcich) prístupov pre rôzne oblasti funkcie hustoty rozloženia. Výsledok sa potom poskladá z jednotlivých výpočtov. Kombinovanie je možné využiť u komplikovanejších priebehov hustoty pravdepodobnosti. Viac v [6].

3 Testovanie generátorov

3.1 Úvodom

Každý navrhnutý generátor musí byť pred použitím otestovaný. Kvalita generátorov môže byť zhodnotená rôznymi spôsobmi. Môžeme sa rozhodnúť, že budeme sledovať rýchlosť generátora, ktorou dokáže generovať čísla, alebo už spomínanú periódu a pod. Keďže však je úlohou generovať pseudonáhodné čísla, budeme zatiaľ hodnotiť generátory podľa toho, aké čísla generuje. Budeme teda hodnotiť najmä štatistické vlastnosti vygenerovaných číselných postupností.

Jedným zo základných spôsobov je spočítať informačnú hustotu, alebo entropiu na skupine čísel. Vyššia entropia v skupine čísel znamená taktiež vyššiu obťažnosť predvídateľnosti čísel v danej postupnosti. Sekvencia skutočne dobrých náhodných čísel bude mať vysokú entropiu, no vysoká entropia nemusí garantovať dobrú „náhodnosť". (Príkladom môže byť súbor komprimovaný softvérovým programom ako *gzip* či *winzip*, ktorý bude mať síce vysokú entropiu, avšak dáta vnútri sú presne štruktúrované, a teda vôbec nie náhodné). Pre testovanie sa síce počítanie entropie niekedy používa, ale existuje mnoho iných štatistických vlastností pseudonáhodných čísel, ktoré kvalitne testy sledujú a vyhodnocujú.

Táto kapitola popisuje spôsoby štatistického testovania a ako príkladom je tu podrobnejšie vysvetlený jeden z najpoužívanejších testov vôbec. Taktiež kapitola podrobne rozoberá spôsob, ktorým boli testované navrhnuté generátory a sumarizuje výsledky z testov.

3.2 Štatistické testovanie

Prvé testy pre náhodné čísla boli publikované už roku 1938 [31]. Testy boli založené na rôznych štatistických princípoch, ktoré mali odhaliť, či teoretické predpoklady o náhodných číslach sú zhodné s experimentálnymi výsledkami. Štyri hypotetické testy brali najviac ohľad na tzv. *„nulovú hypotézu"*, teda myšlienku, že každé číslo v danej postupnosti má rovnakú pravdepodobnosť výskytu a postupnosť je teda náhodná:

Frekvenčný test bol veľmi jednoduchý. Kontroloval výskyt každého z čísel, ktorý by mal byť rovnaký. (Príkladom môže byť generátor generujúci čísla v rozsahu 1 až 100 kde by počty vygenerovaných čísel 1,2,3,4,... museli byť zhodné).

Sériový test testoval v podstate na rovnakom princípe len so sekvenciou dvoch číslic v čase (napr.: 00, 01, 02, 03, ...), kde sa porovnávala ich získaná frekvencia výskytu s hypotetickým predpokladom očakávaného rovnomerného rozloženia výskytu.

Poker test testoval danú postupnosť piatich číslic v čase (aaaaa, aaaab, aaabb,...). Testovanie bolo založené na hre s rovnakým názvom.

Test medzier sa pozeral na vzdialenosti medzi nulami (00 znamenalo vzdialenosť 0, 010 znamenalo vzdialenosť 1, 02250 znamenalo 3, ...). Jednotlivé vzdialenosti sa mali vyskytovať v rovnakom počte.

Ak daná postupnosť bola schopná prejsť všetkými vyše uvedenými testami na určitej hladine významnosti (obecne 5%), tak bola označená ako *„lokálne náhodná"*. Autori testov rozlišovali *„lokálnu náhodnosť"* od *„skutočnej náhodnosti"*. Mnoho testovaných sekvencií vygenerovaných ozajstne náhodnými metódami totiž nezískali svojimi výsledkami označenie *„lokálnej náhodnosti"* na danej hladine významnosti [6]. Veľmi dlhé sekvencie testovaných čísel môžu obsahovať veľa čísel iba s jedinou číslicou. Toto stále môže byť dostatočne náhodné v celej sekvencii, avšak v menších blokoch už nie (hovoríme, že testovaná sekvencia zlyhá na testoch).

Ako sa náhodné čísla stávali a stávajú viac a viac rozšírené a potrebné, viac sofistikovanejších a zložitejších testov je potrebné na nich vykonať. Dnešný spôsob testovania nie je jednoduchý. Najčastejšie je skupina vygenerovaných čísel rôzne rozdelená alebo je na nej ako celku aplikované množstvo štatistických výpočtov, ktoré nám dajú prehľad o tzv. *„stupni náhodnosti"*. Napokon je však na nás, akú si zvolíme hranicu výstupných ukazateľov, podľa ktorej budeme rozčleňovať generátory do kategórií na vhodné alebo pre našu aplikáciu nepostačujúce.

Pokiaľ sa pre zatiaľ obmedzíme na štatistické vlastnosti postupností vygenerovaných generátormi, tak máme celkom pestrý výber. Existuje celkom dosť programov použiteľných pre vykonávanie rôznych štatistických testov číselných postupností. Veľká rada z nich je voľne dostupná. Niektoré z testov sú zatiaľ síce publikované iba v knihách a ich prepis do spustiteľných programov sa len chystá. Vhodnou sa ukazuje byť v praxi asi najpoužívanejšia skupina testov *Diehard* [17], ktorá obsahuje najviac testov, a ktorá bola taktiež použitá v tejto práci na testovanie vytvorených alebo skúmaných generátorov.

3.2.1 Testy *Diehard*

Ako bolo spomínané už skôr, *Diehard* je sada asi pätnástich štatistických testov[1], ktorých autorom je *George Marsaglia.* Je napísaná v jazyku *Fortran*, no neskôr bola pre jej rozšírenosť kompletne prepísaná do jazyka *ISO C*. Používaná je de-fakto ako štandard pre ohodnotenie pseudonáhodných postupností. *Diehard* je popisovaný ako najkvalitnejší a zároveň najnáročnejší spôsob testovania, keďže i mnoho hardvérových či softvérových, už v praxi používaných generátorov, dopredu označených za kvalitné práve na tejto sade testov zlyhalo [27].

---

[1] Niektoré z testov sa opakujú s obmenou vstupných parametrov, a preto je v niektorej literatúre uvedený i iný celkový počet.

Testy *Diehard*, so zdrojovými kódmi pre rôzne platformy, sú voľné dostupné k stiahnutiu z internetových zdrojov [17]. Obsahuje testy, ktoré sú založené na rôznych pravdepodobnostných predpokladoch, ktoré musia spĺňať pseudonáhodné čísla. Okrem testov vyššie uvedených (*frekvenčný, sériový, poker* a *test medzier*) v pozmenenej podobe, obsahuje i testy založené na nasledujúcich princípoch a predpokladoch (názvy uvedené v originálnom anglickom znení pre orientáciu vo výsledkoch, viď *príloha C*):

- **Birthday Spacings:** Vyberieme náhodné body na veľkom intervale z generovaných čísel. Veľkosti medzi bodmi by sa mali približovať k Poissnovému rozloženiu. Názov a princíp je založený na paradoxe dátumov narodení [25].
- **Overlapping Permutations:** Analyzujeme sekvencie piatich nasledovných náhodných čísel. *120* možných usporiadaní *(5!)* by sa malo vyskytovať štatisticky s rovnakou pravdepodobnosťou.
- **Ranks of matrices** prebieha s rôznymi veľkosťami matíc. Napríklad i nad maticami *31 x 31*: Pomocou najľavejších 31 bitov z 31 čísel testovanej postupnosti vytvoríme maticu 31 x 31 nad *{0,1}*. Spočítame hodnosti matíc, ktoré sú z intervalu 0 až 31. Hodnosti pod 28 sa vyskytujú iba ojedinele, preto sú počty týchto matíc zoskupované spoločne. Hodnosti počítame na vzorke *40 000* matíc a test dobrej zhody (χ^2 *test*) je vykonaný na počte matíc s hodnosťami *31, 30, 29 a* $\leq$ *28*. Tieto sumy musia nasledovať určité rozloženie.
- **Monkey Tests.** Spracujme sekvencie vybraných bitových čísel ako „slová“. Spočítame prekrývajúce sa slová v reťazci. Počet „slov“, ktoré sa nevyskytli by mali nasledovať známe rozloženie. Meno a princíp je založený na „nekonečnom opičom teoréme“ [30].
- **Count the 1's:** v bitovej reprezentácií čísel počítame výskyt „1“ z každej vybranej sekvencie a zo zvolených bytov. Každý z bytov ich môže obsahovať od 0 do 8 s predpokladaným výskytom *1, 8, 28, 56, 70, 56, 28, 8 a 1* v *256* možných kombináciách. Jednotlivé byty prevedieme na písmená (0,1 alebo 2 výskyty na A, 3 značia B, 4 znamenajú písmeno C, 5 výskytov vytvorí D a ostatné výskyty značia písmeno E). Počty výskytov písmen sa zmenia na *37, 56, 70, 56, 37*. Existuje teda 5^5 možných slov pozostávajúcich z piatich písmen a z reťazca dlhého *256 000* znakov sa spočítajú výskyty jednotlivých slov. Výskyty by mali nasledovať normálne rozloženie. Blízkosť k nemu nám udáva kvalitu postupností.
- **Parking Lot Test:** V štvorci so stranou dĺžky *100* náhodne, podľa generovaných čísel, umiestnime (zaparkujeme) kružnice o polomere 1 (reprezentujúce autá). Potom sa pokúšame takto zaparkovať druhé, tretie atď. Pokiaľ sa nám nepodarí dané auto umiestniť, opakujeme pokus so zaparkovaním na ďalšie náhodne vybrané miesto, pričom počítame počet celkových pokusov a počet úspešných zaparkovaní. Ak zakreslíme tieto dve hodnoty do grafu, krivka ktorú získame bude podobná tej, ktorú získame takýmto postupom

u perfektného generátora. Keďže u testov nie je dostupná grafika, používa sa nasledujúci princíp s k, teda počtom úspešných zaparkovaní po $n = 12000$ pokusoch. Simuláciou by sme mali získať priemer *3523* a odchýlku blízku k *21,9*. Takže *(k - 3523)/21,9* by mala byť štandardná normálna veličina, ktorá prekonvertovaná na uniformnú premennú, slúži ako vstup pre *Kolmogorov-Smirnov* test (*KSTEST*) [6].

- **Minimum Distance Test:** Náhodne umiestnime (opäť podľa generovaných čísel) *8000* bodov vo štvorci veľkosti strany *10000*. Následné nájdeme minimálnu vzdialenosť medzi $(n^2 - n)/2$ pármi bodov. Ak sú body skutočne nezávisle uniformne rozložené, potom štvorec týchto vzdialeností minimálnych vzdialeností je (veľmi blízko) k exponenciálnemu rozloženiu so stredom *0,995*. Takže $1-e^{\left(-d^2/0{,}995\right)}$ by malo byť uniformné na *[0,1)*. Na týchto hodnotách je opäť vykonaný KSTEST, ako test uniformity náhodných bodov vo štvorci (predpoklad).
- **Random Spheres Test:** Náhodne vyberieme *4000* bodov v kocke s hranou veľkosti *1000*. Zmeriame okruh každého bodu, ktorý predstavuje minimálnu vzdialenosť k inému bodu. Tieto najmenšie dosahy by mal byť (veľmi blízko) exponenciálnemu rozloženiu s daným stredom $(120\pi/3)$. Trojrozmerné okolie taktiež nasleduje exponenciálne rozloženie so stredom *30*. Predpokladom je, že tieto hodnoty sú uniformne rozložené so stredom $1-e^{\left(-r^3/30\right)}$. KSTEST je vykonaný na 20 p-hodnotách.
- **The Squeeze Test:** Násobíme 2^{31} náhodnými desatinnými číslami na intervale *[0,1)* pokiaľ nedosiahneme hodnoty 1. Opakujeme *100000* krát. Na počte desatinných číslic potrebných k dosiahnutiu hodnoty 1 je vykonaný test zhody (χ^2 *test*) s predpokladaným výsledkom.
- **Overlapping Sums Test:** Generujme dlhú sekvenciu *U(1),U(2),..* náhodných desatinných čísel na *[0,1)*. Spočítajme sumy 100 čísiel sekvencie takto *S(1)=U(1)+ ... +U(100), S(2)=U(2)+...+U(101),...*. Lineárnou transformáciou súm *S* konvertujeme čísla na sekvenciu nezávislých premenných, ktoré sú prevedené na hodnoty pre KSTEST. P-hodnoty, ktoré sú výstupom KSTEST-u sú vstupom pre ďalší KSTEST.
- **Runs Test:** Generujme dlhú sekvenciu náhodných čísel opäť na intervale *[0,1)*. Spočítajme nástupné a zostupné hrany (behy). Matice kovariancie pre tieto počty nástupných a zostupných „behov“ sú známe, vykonáva sa preto už spomínaný test zhody. Behy sú uskutočnené 10 krát na sekvenciách s dĺžkou *10000*.
- **The Craps Test:** Hrajme *200000* krát hru s hracími kockami s tým, že počítame počet výhier za celú hru. Počet výhier nasleduje veľmi blízko normálne rozloženie so stredom *200000p* a odchýlkou *200000p(1-p)* kde *p=244/495*. Počet hodov potrebných k výhre môže byť od 1 až po nekonečno, no počty väčšie ako 21 sú zoskupované. Testy (χ^2) sú vykonané na počte hodov v zoskupení. Každé 32 bitové číslo zo zdrojového súboru je prevedené na hodnotu aktuálneho hodu [28].

Väčšina z *Diehard* testov má ako výsledok takzvané *p-hodnoty (p-values)*, ktoré by mali byť rovnomerne rozložené na intervale [0,1) pokiaľ testovaná postupnosť obsahuje skutočne náhodné a nezávislé čísla. Tieto p-hodnoty sú získané ako výsledok funkcie $p = f(x)$, kde f je z predpokladaného rozloženia náhodnej premennej X (často normálne rozloženie). Avšak f je iba (asymptotickou) aproximáciou. Preto netreba zúfať, keď získame ako výsledok testov nášho generátora občas p-hodnoty blízke 0, či 1 (ako napríklad *0,0011* alebo *0,9982*). Ak postupnosť nie je skutočne dosť náhodná, získame p-hodnoty rovné 0 a rovné 1 na *viac ako šiestich* testov z testovacej sady (generátor na danom teste zlyhal). Pri vyhodnocovaní nesmieme však opomenúť i to, že pokiaľ získame hodnoty $p < 0{,}025$ alebo $p > 0{,}975$ znamená to, že testovaná postupnosť zlyhala v teste na hladine významnosti 0,05. Takéto hodnoty sa vyskytujú vo veľkom množstve, ktoré *Diehard* vyprodukuje, ešte aj v prípade dobrých generátorov[1].

Z tohto všetkého ale vyplýva, že existuje viacero metodik a postupov na zhodnotenie výsledkov testov. V mojej práci je použitá metodika, ktorá je využívaná aj v iných prácach [1]. Každá zo získaných p-hodnôt môže byť označená ako *dobrá*, *zlá* alebo *stredná*. Pokiaľ *p-hodnota* $\geq 0{,}998$, tak je zaradená do kategórie zlých. Ďalej pokiaľ $0{,}95 \leq p < 0{,}998$, tak je hodnota zaradená do kategórie strednej. Inak sú hodnoty klasifikované ako dobré. Každej z troch tried je napokon priradené číslo, udávajúce získané *skóre*. Trieda dobrých ma skóre 0, trieda stredných 2 a zlé hodnoty získajú ohodnotenie 4.

3.3 Iné spôsoby testovania

Ako bolo spomínané v úvode, spôsobov ako otestovať generátory pseudonáhodných čísel je okrem štatistického testovania veľa. Jednou často využívanou možnosťou je zobrazenie generovaných čísel dvojrozmerne, kde na ose x a ose y sú striedavo nanášané jednotlivé čísla (viď *príloha D*). Je potrebné zobraziť však veľa dvojíc na to, aby sme mohli odhaliť, či je v číslach závislosť [18]. Podobný tomuto je i test, kde vypisujeme čísla v binárnej podobe pod seba [1]. Závislosť po sebe idúcich dvojíc ale dokáže odhaliť i *Diehard*, pokiaľ má dostatočné veľkú vzorku dát.

3.4 Vlastné testovanie

Vlastné testovanie prebiehalo pomerne zložitým postupom. Najprv boli vybrané niektoré typy generátorov, no najmä tie, ktoré som sa rozhodol v závere práce implementovať i vo *VHDL* (sériový a paralelný LFSR). Menne sú to LFSR (s označením *serial-LFSR*), kombinovaný LFSR (*comb-LFSR,* 3 registre LFSR, v každom iná vnútorná štruktúra), paralelný LFSR (*parallel-LFSR* s rovnakou

[1] Ako autor sám píše [22], je treba pamätať na to, že p-hodnota sa prosto *„vyskytne“*.

štruktúrou registrov), jednorozmerný celulárny automat CA30 (*1d-CA30*) a jednorozmerný celulárny automat CA50745 (*1d-CA50745*) oba s periodickým prepojením. Všetky generátory boli *32* bitovej dĺžky. Tieto typy museli byť potom naprogramované a odsimulované tak, aby sa dali dostatočne otestovať.

3.4.1 Implementácia zvolených generátorov v *jazyku C*

Pre dôvody testovania som sa rozhodol navrhnúť a implementovať generátory najprv v *jazyku C*. Okrem efektivity použitého algoritmu bolo potrebné sa zamyslieť i nad dátovou reprezentáciou údajov. V našom prípade ide o hardvérový register.

U reprezentácie máme obecne niekoľko možností pohľadu na výsledné vygenerované čísla. Buď ich budeme napríklad chápať (a teda aj reprezentovať) ako celé čísla či už so znamienkom alebo bezznamienkové, alebo napríklad ako čisto bitovú postupnosť určitej veľkosti. Ako dátovú reprezentáciu som si zvolil druhú zo spomínaných možností. Takýto bitový pohľad nám umožní jednoducho kontrolovať správnosť implementovanej funkcie. Navyše je táto reprezentácia potrebná pre jednoduchšie prepojenie s testovacím softvérom (*Diehard*), ktorý vyžaduje presne stanovenú formu vstupných dát. Pre účel uloženia číselnej hodnoty v registre nám slúži štruktúra, ktorá dovoľuje implementovať registre ľubovoľnej bitovej dĺžky:

```
typedef struct BitArray
{
   int      numBits;    //celkový počet bitov
   unsigned bits[1];    //jednorozmerné bitové pole
} BitArray;
```

Spolu s dátovou štruktúrou boli taktiež navrhnuté funkcie pre vytvorenie a inicializáciu počiatočnou hodnotou, ďalej funkcie pre posuv bitovej hodnoty v registri danými smermi a funkcie pre získanie hodnoty a pre nastavenie hodnoty v registri. Navyše okrem týchto základných funkcií bolo taktiež potrebné vypisovať konkrétny stav registrov v čase, teda k už spomínaným pribudla ešte funkcia na výpis obsahu registra, a napokon funkcia na korektnú deštrukciu zabranej pamäte.

3.4.2 Spôsob testovania

Ako bolo spomínané skôr, pre testovanie som zvolil sadu štatistických testov *Diehard*. Táto je síce kvalitná a pomerné často používaná, no má i svoje neduhy. Pokiaľ nebudú splnené špeciálne požiadavky na vstupný súbor, tak nebude vykonaný žiadny z testov a testovací program skončí s nič nehovoriacou chybou. Preto sa bolo potrebné prispôsobiť tak, aby mohli testy prebehnúť úspešne.

V prvom rade potrebujeme binárne súbory *32* alebo *31* bitových čísel veľkosti *11 468 800* bytov. To je čiastočne splniteľné nastavením generátorov na veľkosť registra 32

a generovaním presného počtu pseudonáhodných čísel. Ďalej je potrebné generovať celé čísla v zápise desiatkovom. Keďže však vytvorené generátory generujú iba čísla v bitovej podobe, je ich potrebné previesť do binárnej postupnosti, aby boli splnené požiadavky. Binárny súbor presne danej veľkosti a štruktúry je ale zložitejšou záležitosťou, a preto boli využité prostriedky, ktoré sú súčasťou testovacej sady (autor testov predpokladal zbytočnú zložitosť réžie pri testovaní). Priložený program pre vytváranie binárnych súborov (*ASCBIN*) ale zas vyžaduje čísla zapísané v šestnástkovom formáte, preto je potrebný ďalší program na prevod bitových čísel na čísla šestnástkove. Všetky spomínané programy sú súčasťou tejto práce. Pre ich spustenie a vytvorenie zdroja pre *Diehard* môžeme zadať napríklad:

```
./serial-LFSR 32 4000000 | ./bin2hex > serial-LFSR.out
./1d-CA30 32 4000001 123456 | ./bin2hex > 1d-CA30.out
```

Transformáciu do binárnej podoby prevedie spomínaný priložený program. Pre ďalšie informácie je možné pozrieť do zdrojových kódov programov, alebo zobraziť bohaté nápovede k nim samotným.

3.4.3 Výsledky testov

Výstupom z programu pre testovanie vygenerovaných postupností je súbor, ktorý u testov *Diehard* obsahuje mnoho nespracovaných hodnôt (p-hodnoty). Tie treba upraviť do takej podoby, aby mali nejakú konkrétnejšiu vypovedaciu hodnotu. *Príloha B* zobrazuje výsledky na jednotlivých testoch u každého z generátorov spusteného *2x* pri zakaždým iných počiatočných hodnotách (je možné vyčítať z *prílohy E*).

Pomocou metodiky s klasifikáciou do troch tried boli na výsledkoch každého z generátora urobené výpočty, ktorými sa dospelo k tabuľke zobrazenej v *prílohe C*. Okrem testovaných generátorov sú tu na jednotlivých testov uvedené aj tie výsledky, ktoré získala postupnosť získaná z internetových stránok organizácie zaoberajúcej sa náhodnými práve číslami [19]. Táto postupnosť je označovaná ako *True*, a mala by podľa zdroja spĺňať požiadavky na skutočne náhodné čísla. Jednoduchší výpis z *prílohy C* je uvedený v nasledujúcej tabuľke 4.

Typ Generátora	*Výsledné Poradie*	*Skóre získané z testov Diehard*
Skutočný (*True*)	1	22
Paralelné LFSR	2	154
Kombinované LFSR	3	286
CA50745	4	570
CA30	5	644
Sériové LFSR	6	756

Tabuľka 4 : *Výsledné poradie testovaných generátorov*.

Ako je vidieť, skutočne náhodné čísla naozaj nesklamali. Nižšie číslo značí lepší generátor. Ako ďalším pre nás dôležitým výsledkom je to, že generátor založený na paralelnom princípe LFSR sa umiestnil medzi ostatnými testovanými na najlepšej, druhej pozícií. Navyše získal skoro až dvojnásobne lepšie hodnotenie ako generátor umiestnený hneď za ním. Zaujímavosťou môže byť to, že generátor založený na jednoduchom sériovom zapojení LFSR zostal na poslednom mieste, ba dokonca zlyhal skoro na všetkých testoch. Je to zaujímavé práve z toho postu, že tento generátor (ako bolo uvedené v predchádzajúcom texte) je v praxi hojne používaný.

Ďalší spôsob ako som overoval kvalitu vytvorených generátorov, bolo vykonanie sériového testu. *Príloha D* zobrazuje výsledky testu tohto typu. Na každom je zobrazených *32000* dvojíc. Tento test odhalil to, že zložitejší typ celulárneho automatu CA50745 opakuje niektoré generované čísla, čo môže byť veľmi nevhodné. Ďalej sa iba potvrdzujú výsledky získané predchádzajúcou metodikou. Je vidieť silnú sekvenčnú závislosť u jednoduchého generátora LFSR. Taktiež sa objavil zvláštny druh závislosti i u jednoduchého celulárneho automatu CA30.

Vo všetkých testoch sa okrem paralelného zapojenia javilo ako štatistický kvalitné i zapojenie kombinované (*comb-LFSR*). Nevýhodou, ktorú je potrebné však ešte raz pripomenúť, je u tohto generátora to, že dokáže generovať iba *n* krát pomalej ako jeho paralelná verzia. Rýchlosť generátora sa ale predchádzajúcimi testami nedá zistiť.

Podobným spôsobom k sériovému testu sú taktiež získané zobrazenia, kde bola bitová reprezentácia generovaných čísel prevedená do príjemnejšej podoby pre získanie prehľadu o závislosti medzi generovanými číslami. Zobrazené sú všetky štyri generátory okrem kombinovanej formy LFSR, keďže z tohto pohľadu je možné odhaliť najviac iba závislosti po sebe idúcich čísel, ktorou kombinovaná forma zapojenia netrpí. Viac v *prílohe D*.

4 Implementácia zvolených generátorov

4.1 Úvodom

S ohľadom na rastúcu zložitosť dnešných číslicových systémov sú na návrhárov kladené stále vyššie a vyššie nároky. Návrhár musí vedieť tvoriť na vysokej úrovni abstrakcie, inak nebude schopný obsiahnuť nové aplikácie v celej šírke so všetkými dôsledkami na výkonnosť, cenu, spotrebu energií a ďalšie podstatné aspekty.

Táto kapitola práce popisuje okrem spôsobu akým sa implementujú číslicové obvody i samotnú implementáciu zvolených generátorov v hardvéri. Pri návrhu boli pritom použité dva extrémy, ktoré boli i otestované. Ide o v praxi často používané sériové zapojenie LFSR a v praxi používané menej, ale kvalitné paralelné zapojenie LFSR. Ďalej je popísaná implementácia generátora do výukovej pomôcky FITKit v jazyku *VHDL*, a taktiež cieľová architektúra obvodov FPGA spolu s efektívnym využitím týchto prostriedkov.

4.2 Návrh číslicových systémov

Rozvoj číslicových systémov je postavený na troch základných prvkoch [8]: vyspelosti cieľovej technológie (CMOS), pokročilých návrhových metódach a prostriedkoch (CAD) a na spoločenskej potrebe (napr. rozvoj grafických prostriedkov).

V poslednej dobe, práve vďaka novým technológiám rekonfigurovateľného hardvéru (napr. *PLA, CPLD, FPGA*), sa svet softvéru a hardvéru začínajú zbližovať natoľko, že je možné hovoriť o programovaní na úrovni aplikácie. Aplikácia sa popisuje bez ohľadu na to, či bude daný algoritmus nakoniec implementovaný v softvéri alebo hardvéri. Návrhár iba vyvíja algoritmus. Rozdelenie na časť, ktorá sa implementuje ako program, a na časť realizovanú v špeciálnom hardvéri, sa optimalizuje na základe množstva kritérií automatizované počítačom. Ďalším trendom je tzv. vyvíjajúci sa hardvér (*evolvable hardware*). Populárne povedané sa jedná o nahradenie návrhára počítačom. Viac v literatúre, napr. [23]. Uvedené techniky však nie sú všemocné a je iba na človeku ako návrhárovi, aby ich správne použil a využil.

Moderný návrh číslicových obvodov sa obyčajne skladá z niekoľkých sekvenčných krokov. Ako prvým je zadanie špecifikácie obvodu. Je to či už slovný popis, alebo prepis danej logickej funkcie do pravdivostnej tabuľky. Ďalej nasleduje návrh a implementácia obvodu. Na toto nám slúži v dnešnej dobe už dosť kvalitných jazykov pre popis hardvéru. Medzi najzákladnejšie patrí *VHDL*, *Verilog* a *HandelC*. Pomocou takéhoto popisu je možné napríklad vykonať simuláciu na funkčnej úrovni. Ďalším krokom návrhu je syntéza, pomocou ktorej sa zo známych programových konštrukcií vyvinú registre, násobičky, sčítačky apod. Takto nám teda vznikne schéma zložené zo základných

blokov cieľovej technológie, ktorou je v našom prípade FPGA. Napokon vzniknutý konfiguračný reťazec stačí nahrať do vybraného rekonfigurovateľného obvodu a získame tak navrhovaný obvod.

Na záver nesmieme však zabudnúť ale na to, že technológia sa mení s každým dňom, no koncepty dlho pretrvávajú. Vždy sa bude jednať o návrh algoritmu a jeho následnú fyzickú implementáciu strojom.

4.2.1 Jazyk *VHDL* pri popise hardvéru

Vývoj moderných jazykov *HDL* (*Hardware Description Language*) je typický snahou pokryť čo najväčšiu škálu úrovní popisu systému, pre ktorú by bol použiteľný. V súčasnosti sa stáva štandardom, ktorý túto požiadavku do určitej miery splňuje jazyk *VHDL* [4,15,21]. Tento jazyk bol i implementačným jazykom pre vybrané generátory.

VHDL (*Very High Speed Integrated Circuit Hardware Description Language*) je programovací jazyk pre popis hardvéru. Bol vyvinutý na podnet Ministerstva obrany USA, za účelom popísať chovanie ASIC (*Application-Specific Integrated Circuit*) obvodov, ktoré mali byť zakomponované do výzbroje. Tento jazyk je štandardizovaný organizáciou *IEEE*, a to znamená, že zaručuje vysokú kompatibilitu a prenositeľnosť medzi rôznymi systémami. Pomocou VHDL vieme popísať jednak hardvérové prvky ale tiež, vďaka prevzatým jazykovým konštrukciám z vyšších programovacích jazykov (na vývoj softvéru), obecné algoritmy. Tieto konštrukcie zabezpečujú vysokú flexibilitu hlavne pri simulácii a testovaní navrhnutého hardvéru.

Nie každý VHDL kód je však syntetizovateľný, teda previediteľný do binárneho súboru určeného pre cieľovú architektúru programovateľného logického obvodu, napr. FPGA. Preto pri písaní kódu je nutné uvážiť, či popisujeme číslicový obvod na simuláciu a testovanie, alebo ho chceme reálne vyrobiť. Len určitá podmnožina tohto jazyka je syntetizovateľná. Je obtiažne jednoznačne určiť jej hranice, ktoré závisia od toho, ako sa použitý syntetizátor vysporiada so zapísanými jazykovými konštrukciami a ich prevodom na reálne hardvérové prvky. Platí však zásada, čím bližšie návrhár popíše hardvér, tým menší priestor k omylu poskytuje syntetizátoru.

U zápisu vo VHDL môžeme použiť dva hlavné štýly:

- *behaviorálny* – popisujeme ako obvod funguje, čiže jeho chovanie.
- *štrukturálny* – je založený na hierarchickom popise jednotlivých komponent, zložených zo základných logických členov a ich prepojení signálmi.

Pri programovaní hardvéru je možné oba spomínané štýly spájať. Typické je popísať jednotlivé komponenty obvodu behaviorálne a následne ich štrukturálne prepojiť do jedného celku.

Práve vďaka jazykom s vyššou úrovňou abstrakcie, akým je i VHDL, je možné vo veľmi krátkej dobe vytvoriť prototyp zariadenia v FPGA, ktorý môže byť po testovaní nasadený do reálnej prevádzky.

4.2.2 Vlastná implementácia

Výsledkom vlastnej implementácie generátorov pseudonáhodných čísiel v jazyku VHDL sú rôzne typy jednoduchých i paralelných verzií LFSR. Obmeny jednotlivých typov sú vo veľkosti, štruktúre, privedení resetovacieho vstupu alebo majú paralelný vstup dát. U paralelnej verzie LFSR je zrejmé, že obvody majú implicitný paralelný vstup i výstup. Obvody generátorov sú samozrejme doplnené o multiplexory, ktoré povoľujú posuv obsahu (*S_EN*) alebo naplnenie počiatočnou hodnotou (*F_EN*).

4.3 FITKit

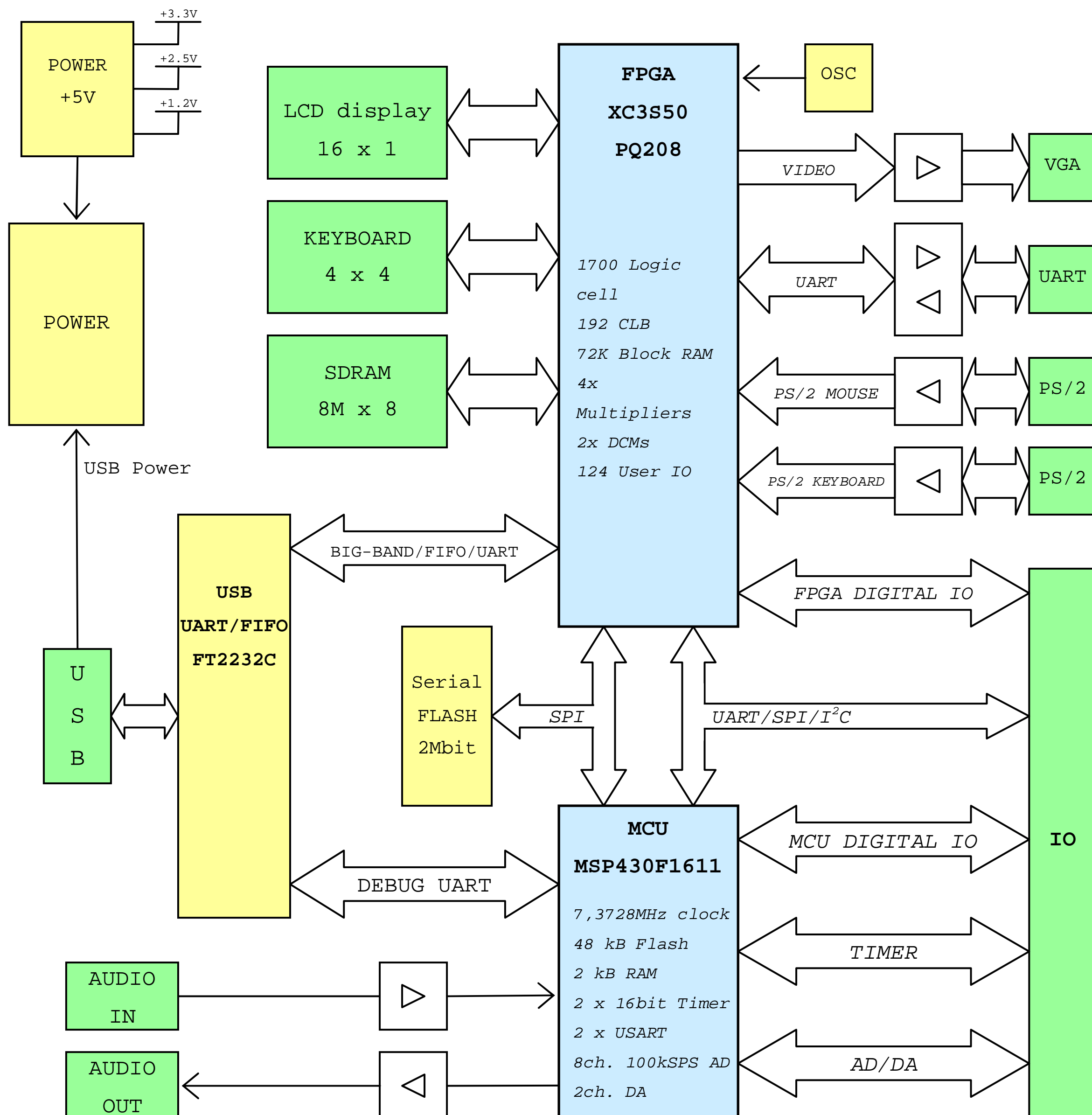
FITKit obsahuje výkonný mikrokontrolér (*MCU*) s nízkym príkonom spolu s radou moderných periférií. Dôležitým aspektom je tiež využitie pokročilého programovateľného hardvéru v podobe *FPGA*, ktorý je možné, tak ako softvér v počítači, skoro neobmedzene modifikovať pre rôzne účely podľa potrieb.

Cieľom nasadenia platformy *FITKit* do výuky je umožniť študentom, aby mohli prakticky navrhovať a realizovať nielen softvérové, ale taktiež hardvérové projekty, či celé aplikácie. Platforma *FITKit* umožňuje dosiahnuť značnú časť spektra znalostí, ktoré musí dnešný inžinier – informatik vedieť, aby bol schopný obstáť na globálnom trhu práce. Typickým takým príkladom v praxi sú tzv. *vstavané systémy*, ktoré sa v dnešnej dobe dominantne uplatňujú v bežnom živote každého z nás. Ide ku príkladu o videa, televízory, pračky, digitálne prehrávače a naozaj mnoho iných bežne používaných elektronických pomôcok. Typický vstavaný systém sa skladá z (mikro)procesora, špecializovaného hardvéru a aplikačného softvéru. Z toho vyplýva, že je potrebné vedieť a prakticky využiť znalosti nielen z oblasti aplikačného softvéru, ale i z oblastí zaoberajúcich sa hardvérom.

Pri návrhu aplikácií sa využíva skutočnosti, že vlastnosti hardvéru sa popisujú prevažne programovacím jazykom (spomínaným *VHDL,* či iným), ktorým teda budeme navrhovať vnútornú štruktúru obvodu *FPGA*. Generovanie programovacích dát do obvodu zariadia profesionálne návrhové systémy. Všetok potrebný softvér k dispozícii zdarma [7]. Softvér pre *MCU* sa naopak tvorí v *jazyku C* a prekladá sa pomocou prekladača *GNU.* Do *MCU* ho potom dostaneme nástrojmi, ktoré sú opäť k dispozícii. Celý projekt je koncipovaný ako *open-source* (pre SW) alebo *open-core* (pre HW).

4.3.1 Popis kitu

Ako bolo spomenuté skôr, *FITKit* je okrem *FPGA* a *MCU* zložený i z rady periférií, ktoré je možné vidieť na nasledujúcom obrázku 8 blokovej schémy zapojenia kitu. Pretože sa táto práca zaoberá návrhom generátora pre FPGA, nasledujúca kapitola je venovaná práve tejto technológií. Informácie o ostatných súčastiach kitu (MCU, LCD, klávesnica, SDRAM,...) je možné nájsť napríklad na stránkach projektu [7].



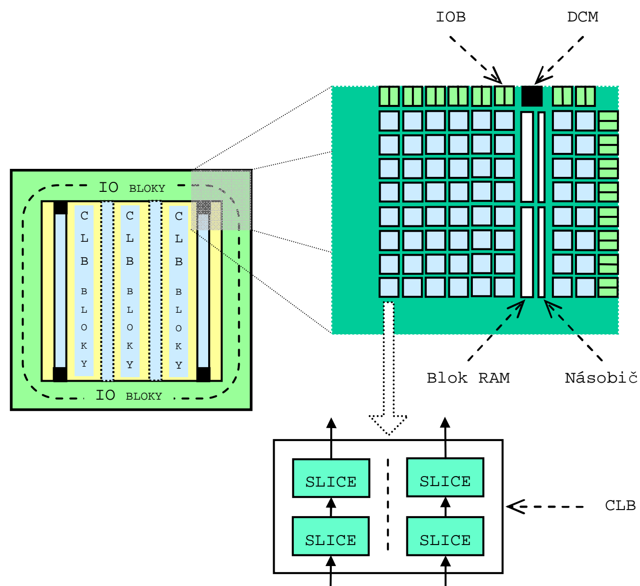
Obrázok 8 : ***Blokové schéma FITKitu.***

4.3.1.1 Architektúra použitého FPGA

Súčasné FPGA obvody predstavujú na poli rekonfigurovateľných hradlových polí špičkou v svojej technológií. Takýto obvod FPGA *Spartan 3 (XC3S50 PQ208 od firmy Xillinx)* je taktiež súčasťou výukového kitu. Kvalitný návrh aplikácie vyžaduje porozumeniu jeho štruktúry [15].

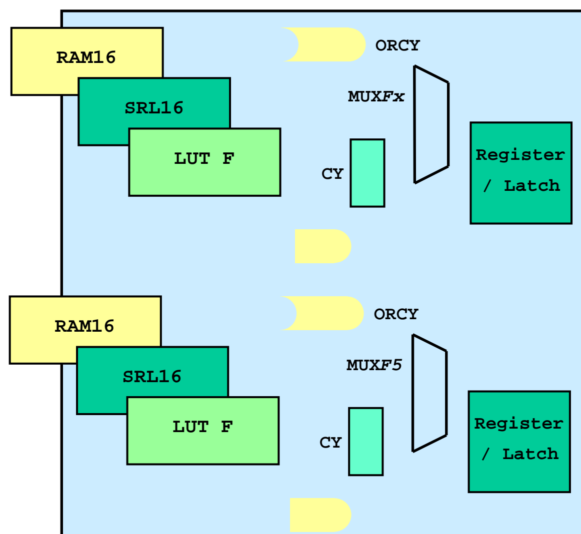
Ako je na obrázku 9 vidieť, náš FPGA obvod je tvorený maticou konfigurovateľných logických blokov (*CLB*), medzi ktorými sú prepojenia, vstavanými blokovými pamäťami (*Blok RAM*) spolu s násobičkami a obvodmi pre riadenie hodinového signálu (*DCM*). Po okolí obvodu FPGA sú umiestnené bloky vstupu a výstupu (*IO Bloky*), ktoré sú priamo napojené na vývody.

Každý CLB blok ďalej obsahuje štyri menšie logické elementy, ktorými sú tzv. „*SLICE*“ a dva nezávislé *carry* reťazce pre konštrukciu rýchlych sčítačiek. Bloky CLB sú navzájom prepojené, a taktiež sú prepojené i s globálnou prepojovacou maticou.



Obrázok 9: ***Štruktúra obvodu FPGA (rodina Xilinx, Spartan 3).***

Pri návrhu číslicových obvodov je pre nás dôležitá architektúra *SLICE*. Práve tá totiž obsahuje niektoré prvky, ktoré sú pre nás využiteľné. Obrázok 10 zobrazuje vnútornú štruktúru takéhoto *SLICE*.



Obrázok 10 : ***Vnútorná štruktúra SLICE obvodov FPGA.***

Jeden SLICE obsahuje dva funkčné generátory (*FG*) s funkciou *LUT, RAM16,* alebo *SRL16* podľa potreby. Ďalej dva záchytné registre (*Latch Register*), multiplexory (*MUXFx, MUXF5*), „Carry“ logiku a pomocnú logiku pre aritmetiku.

Využitie LUT

Pokiaľ chceme usporiť niektoré zdroje, ktoré nám ponúka obvod FPGA, je potrebné sa obzrieť za efektívnejším využitím niektorých komponent tohto obvodu. Tie sú dostupné vo forme funkčných generátorov (*FG*), ktorými je možné implementovať napríklad ľubovoľné logické hradlo.

LUT je 16 bitová pamäť (logické hradlo [15]) so štyrmi vstupmi a jedným výstupom. Realizuje obecnú binárnu funkciu štyroch premenných. Navyše je pomocou multiplexorov *MUXFx* a *MUXF5* možné jednoducho realizovať zložitejšie funkcie. *MUXFx* je potom označovaný ako *MUXF6* alebo *MUXF7*.

Z popisu je jasné, kde je možné tejto funkčnosti využiť. Je to práve v celulárnych automatoch. Problémom však ostáva, ako čo najelegantnejšie prepojíme jednotlivé hradla obvodu. Toto súvisí so spomínanou zložitosťou CA. Všetok tento návrh ostáva na návrhárovi.

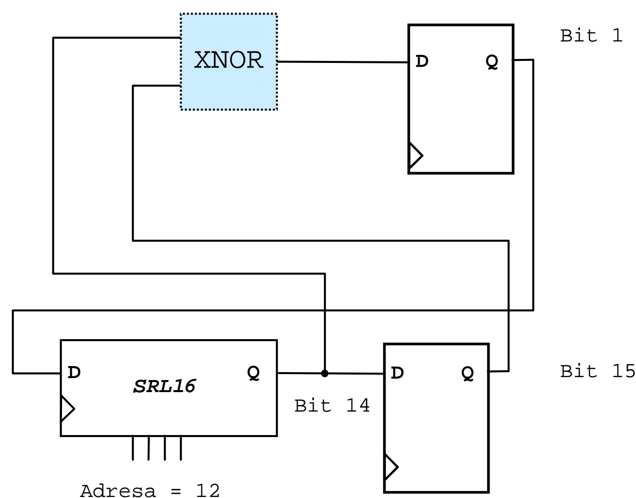
LUT je možné deklarovať ako komponenty vo VHDL, no obvykle sa to nerobí. Syntetizátor ich automatický vytvára na základe behaviorálneho alebo štruktúrneho popisu vo VHDL.

Využitie SRL16

Ďalšou úsporou miesta na navrhovanom čipe je možnosť využitia funkčného generátora ako registra *SRL*.

SRL16 je teda posuvný, v základnej verzii *16* bitový register, ktorý okrem synchrónneho zápisu s posunutím o jednu pozíciu umožňuje i asynchrónne čítanie zo zadanej pozície. Pomocou multiplexorov *MUXFx* a *MUXF5* umožňuje navyše jednoduché rozšírenie na register skoro ľubovoľnej dĺžky [15].

Z uvedeného popisu je zrejmé, že komponenty tohto typu sú využiteľné u obvodov LFSR. Implementácia 15 bitového LFSR, s použitím iba dvoch komponent SLICE z CLB je na obrázku 11:



Obrázok 11 : ***15 bitové LFSR s použitím ½ CLB.***

Jedná logická bunka zo SLICE je využitá k dvojvstupovému *XOR* obvodu a k prvému registru z 15 bitového LFSR. Druhá bunka je potom využitá k implementácii 13 bitového zvyšku obvodu LFSR pomocou SRL komponenty a poslednému z registrov. Priradením statickej hodnoty 12 k adresným vodičom je výstupná dĺžka registra SRL veľkosti 13. Najlepšie pozície odberu pre 15 bitový LFSR sú z miest 14 a 15 (viď *príloha A*).

Obecne je teda možné pomocou tohto efektívneho návrhu implementovať LFSR rôznej dĺžky. V zápise vo VHDL je potom potrebné pridať komponentu, ktorá označí danú časť ako SRL:

```
component SRLC16E
   port (
         D          : in std_logic;
         CE         : in std_logic;
         CLK        : in std_logic;
         A0         : in std_logic;
         A1         : in std_logic;
         A2         : in std_logic;
         A3         : in std_logic;
         Q          : out std_logic;
         Q15        : out std_logic
          );
end component;
```

Moderné syntetizačné nástroje ale dokážu (nie vždy, môže nás napríklad obmedzovať spôsob návrhu) už z niekoľkých registrov usporiadaných za sebou samé zistiť, že je výhodné využiť komponenty SRL [2]. Pozor však treba dávať na to, že posuvný register nesmie mať asynchrónny *RESET,* pretože funkčný generátor nevie resetovať svoj obsah (synchrónnym resetovacím vstupom sú vybavené takmer všetky komponenty na obvode). Posuvný register s týmto resetom bude automatický zostrojený z bežných registrov. Pre viac informácií je možné prezrieť literatúru [2,16].

4.3.2 Prepojovací systém FITKitu

I keď projekt *FITKit* beží na fakulte pomerne krátku dobu, je potrebné si uvedomiť, že už existuje mnoho funkčných príkladov, pomocou ktorých môžeme vytvárať programy v súlade s už spomínanými licenčnými podmienkami. Navyše existuje vyvíjajúca sa knižnica, ktorú je vhodné taktiež použiť pri implementácii. Tieto vedomosti využijeme pri ďalšom návrhu a implementácii.

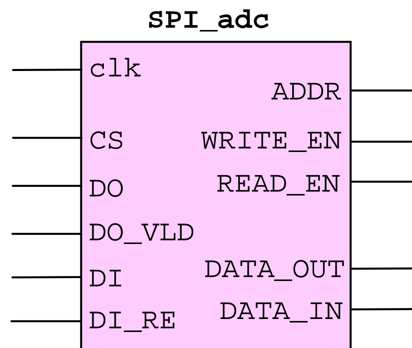
Komponenty kitu musia medzi sebou vedieť komunikovať. Pre komunikáciu je zvolený rýchly *sériový protokol*, pomocou ktorého kit vymieňa dáta či už s okolím alebo v medzi svojimi internými komponentmi. S ohľadom na túto koncepciu boli vyvinuté i vnútorné komponenty obvodu FPGA.

Základný prepojovací systém na čipu teda slúži pre komunikáciu medzi MCU a radičmi jednotlivých periférií implementovaných v FPGA. Prepojovací systém reprezentujú dve komponenty: *SPI control* a *SPI decoder*, na ktorý je pripojený adresný dekodér radiča komponenty. Pre nás je

najviac dôležitý práve adresný dekodér, pretože to je ta koncová komponenta, na ktorú sa budeme pripájať naším generátorom. Pre ostatné komponenty viď [7].

4.3.2.1 SPI dekodér

SPI dekodér prevádza interný SPI na štandardné rozhranie, na ktoré je možné pripojiť adresný dekodér daného radiča periférie. Táto komponenta sa použije pre každý radič periférie, ktorý bude ovládaný z MCU. Rozhranie komponenty ukazuje obrázok 12. Nižšie je popis rozhrania vo VHDL.



Obrázok 12 : ***Rozhranie komponenty SPI dekodéra.***

```
entity SPI_adc
   generic (
        ADDR_WIDTH         : integer;
        DATA_WIDTH         : integer;
        ADDR_OUT_WIDTH     : integer;
        BASE_ADDR          : integer
          );
   port (
     CLK       : in  std_logic;
     CS        : in  std_logic;
     DO        : in  std_logic;
     DO_VLD    : in  std_logic;
     DI        : out std_logic;
     DI_REQ    : in  std_logic;
     ADDR      : out  std_logic_vector (ADDR_OUT_WIDTH-1 downto 0);
     DATA_OUT  : out  std_logic_vector (DATA_WIDTH-1 downto 0);
     DATA_IN   : in   std_logic_vector (DATA_WIDTH-1 downto 0);
     WRITE_EN  : out  std_logic;
     READ_EN   : out  std_logic
       );
end entity;
```

SPI dekodér sa nastavuje pomocou generických parametrov. Tieto parametre nastaví dekodér podľa potrieb radiča pripojeného za ním. *Generické* parametre sú:

- ADDR_WIDTH – šírka adresy
- DATA_WIDTH – šírka vstupno-výstupných dát
- ADDR_OUT_WIDTH – šírka výstupnej adresy
- BASE_ADDR – hodnota bázovej adresy

Popis rozhrania je nesledujúci:

- CLK – synchrónne hodiny používane v celom FPGA
- Interné SPI (DO, DO_VLD, DI, DI_REG, CS) :
    - CS – povolenie komunikácie.
    - DO + DO_VLD – výstupné dáta SPI DO a detekcia nástupnej hrany SPI_CLK. Pri nástupnej hrane (SPI_CLK) sa čítajú hodnoty z dátových signálov platné v okamžiku DO_VLD.
    - DI + DI_REG – vstupné dáta DI a detekcia zostupnej hrany SPI_CLK. Pri zostupnej hrane (SPI_CLK) sa zapisujú hodnoty na dátové signály v okamžiku DO_REG.
- Rozhranie k adresnému dekodéru (ADDR, DATA_OUT, DATA_IN, WRITE_EN, READ_EN) :
    - ADDR – výstupná adresa
    - DATA_OUT – výstupné dáta
    - DATA_IN – vstupné dáta
    - WRITE_EN – okamžik zápisu výstupných dát do dekodéru
    - READ_EN – nastavenie vstupných dát pre čítanie. Tento signál hovorí dekodéru, že má nachystať dáta DATA_IN (vstup). Samotné dáta sa prečítajú až pri nasledujúcom nastavení signálu DI_VAL na logickú 1.

Viac podrobností o jednotlivých signáloch, či niektoré priebehy je opäť možné naliezť na stránkach projektu [7].

4.3.3 Generátor pseudonáhodných čísel pre FITKit

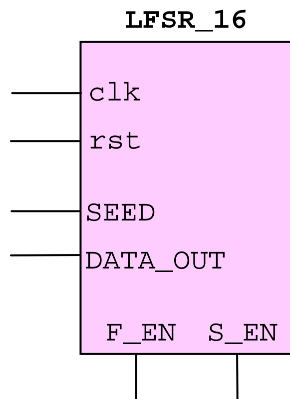
Pri záverečnom návrhu a implementácii generátora pseudonáhodných čísel bolo treba brať v ohľad viacero obmedzujúcich kritérií. Niektoré rozhodnutia boli ovplyvnené taktiež tým, že výsledná aplikácia by mala byť ukážkovou, či aplikáciou potvrdzujúcou teoretické predpoklady, a nie

aplikáciou použiteľnou pre generovanie náhodných čísel napríklad pre tvorbu kryptografických kľúčov.

Pri návrhu pre *FITKit* som sa rozhodol, že použijem pre daný účel funkciu, ktorá však umožňuje prenos iba 16 bitových čísel v rámci *MCU* ↔ *FPGA*. Z toho vyplýva hneď to, že implementovaný generátor bude 16 bitovej dĺžky. Pre ukážkovú aplikáciu je to pomerne slušný rozsah, ktorým môžeme generovať kladné čísla v intervale *1 – 65535* (0 je zakázaným stavom). Pokiaľ by bolo potrebné niekedy v budúcnosti implementovať generátory generujúce vo väčšom rozsahu, je možné použiť jednu z verzií, ktorá je implementovaná ako súčasť tejto práce, alebo vygenerovať vlastnú verziu. Všetko toto je možné nájsť na priloženom *CD* nosiči.

Ďalej pokiaľ ide o rozhranie medzi dekodérom SPI a samotným generátorom, tak v tomto prípade sme schopný ušetriť časť FPGA obvodu tým, že nebudeme implementovať adresný dekodér. Je to totiž zbytočné. Pri jednoduchej verzii LFSR registra je potrebné kvôli počiatočnej hodnote (*seed*) adresovať práve jeden register, z ktorého zároveň čítame generované čísla. Opäť pokiaľ bude niekedy potrebné, je možné nájsť paralelné verzie LFSR generátorov na priloženom *CD*.

Navrhnuté rozhranie generátora je nasledovné:



Obrázok 13 : *Rozhranie komponenty implementovaného generátora.*

Kde

- CLK – hodiny generátora spoločné pre celé FPGA,
- RST – resetovací vstup generátora,
- SEED – 16 bitový vstup pre vloženie počiatočnej hodnoty,
- DATA_OUT – 16 bitový výstup, z ktorého získavame generované čísla,
- F_EN – povolenie naplnenia obsahu registra počiatočnou hodnotou (*fill enable*) a
- S_EN – povolenie posuvu obsahu registra (*shift enable*).

Samotná entita generátora vo *VHDL* vyzerá nasledovne:

```
entity LFSR_16 is
 port(
        clk      :    in   std_logic;
        rst      :    in   std_logic;
        F_EN     :    in   std_logic;
        S_EN     :    in   std_logic;
        SEED     :    in   std_logic_vector (15 downto 0);
        DATA_OUT :    out std_logic_vector (15 downto 0)
 );
end entity LFSR_16;
```

Je vidieť, že nie sú použité generické parametre. To je práve z toho dôvodu, že štruktúra obvodov LFSR pre dané dĺžky nie je presne daná. Ako návrhár si môžem vybrať ľubovoľné prepojenie, alebo sa prikloniť k už otestovaným variantom (*príloha A*).

Prepojenie generátora s komponentou SPI dekodéra je zrejmé z nasledujúceho:

Generátor		*SPI dekodér*
SEED	⇔	DATA_OUT
DATA_OUT	⇔	DATA_IN
F_EN	⇔	WRITE_EN
S_EN	⇔	READ_EN

Z uvedeného je zrejmé, že pri čítaní obsahu sa stav registra upraví vždy o jeden krok. Je však možné upraviť zapojenie i tak, že pri každom po sebe idúcom čítaní stavu *nezískame* skutočne po sebe idúce dvojice, čím čiastočne zabránime nepríjemnému efektu spomínanej *závislosti po sebe idúcich dvojíc*. Zapojenie môžme upraviť tak, že pri každej zmene hodinového signálu sa zmení i obsah registra LFSR:

S_EN ⇔ *log. „1“*

Pri implementácií do FPGA s ohľadom na čo najnižšie zabraté miesto taktiež nesmieme zabudnúť ale na to, že posuvný register nesmie obsahovať asynchrónny *RESET*, pokiaľ má byť implementovaný pomocou blokov SRL tak, ako bolo spomínané vyššie v práci.

5 Syntéza generátorov

5.1 Úvodom

O syntéze môžme hovoriť ako o vytvorení *„netlistu“*, čiže zapojenia obvodových prvkov (logické členy, klopné obvody, registre atď.), teda vlastne vytvorenie schémy zapojenia s obvodovými prvkami, ktoré sú obsiahnuté v predpokladaných cieľových obvodoch s požadovanou funkciu. U zložitých obvodov CPLD a FPGA sú okrem vyššie uvedených obvodových prvkov do netlistu zaradené prvky špecifické pre cieľové obvody (obvykle sú spoločné pre určitú radu obvodov, napríklad *XC9500XL, Spartan-II, Virtex-II, Spartan 3* a podobne). Pokiaľ nie sú tieto špecifické prvky použité, je netlist prenositeľný i na iné cieľové obvody. Ak však použité sú, je prenositeľnosť obmedzená na tie obvody, ktoré tieto prvky obsahujú. Akákoľvek konštrukcia je v princípe realizovateľná bez týchto prvkov (s výnimkou veľmi špeciálnych prvkov, ako sú prvky pre úpravu hodinového signálu, pre riadenie odberu z napájacieho zdroja apod.). Ich použitie však môže výsledok syntézy výrazne zlepšiť a v mnohých prípadoch je tento výsledok bez nich tak zložitý, že je prakticky nepoužiteľný.

Táto kapitola stručne pojednáva o výsledkoch syntézy, ktorá bola vykonaná pre rôzne typy implementácií generátorov pseudonáhodných čísiel.

5.2 Výsledky syntézy pre zvolené implementácie

Obecne sa bez znalosti cieľovej technológie odporúča zápis kódu jazyka HDL (v našom prípade *VHDL*) pomocou behaviorálneho zápisu. Vyhneme sa tým problému s prenositeľnosťou. Pokiaľ ale je technológia známa (ako v našom prípade – FPGA, konkrétne *Spartan 3*), nič nám nebráni využiť jej prostriedky, a zefektívniť tak výsledný obvod, či už vzhľadom k ploche, spotrebe alebo rýchlosti.

Pre účely porovnania boli výsledky po syntéze (obsadená *plocha* na čipe a *pracovná frekvencia* obvodu) zhrnuté v tabuľke uvedenej v *prílohe F*. Zobrazené sú jednoduché typy *4, 8, 16, 32* a *64* bitových generátorov LFSR (*s jedným registrom – jednoduché*), i zložené typy (*s viacerými registrami LFSR*, z ktorých sa odoberá výstup tak, ako to uvádza predchádzajúci text), popísané či už behaviorálne alebo štruktúrou. Vnútorná štruktúra jednotlivých generátorov (daná polynómom), je v súlade s údajmi, ktoré je možné nájsť v *prílohe A*.

Dobrým výsledkom je to, že syntetizačný nástroj (*Xilinx ISE 8.2*) pri popise obvodu chovaním sám v priebehu syntézy zistil, že sa dá využiť prvkov cieľovej technológie (*SRL*) a buď ich využil, alebo pokiaľ to nebolo možné (*privedený reset*), aspoň na toto upozornil. Pokiaľ by sme ale i naďalej

chceli používať tieto obmedzujúce faktory, môžeme zvážiť využitie toho, že obvody FPGA dokážu synchrónne resetovať svoje prvky i vnútorne, a pokúsiť sa využiť tejto vlastnosti.

Taktiež pre účely porovnania sú v *prílohe F* zobrazené výsledky syntézy u generátorov, ktoré mali privedený reset (*popísané behaviorálne*), alebo ako jednoduchý typ jediného registra LFSR mali paralelný vstup i výstup dát (*popísané štruktúrne*). Tieto zaberú však omnoho viac hardvérových zdrojov oproti jednoduchším typom, čo sa dá predpokladať.

Druhá zo sledovaných vlastností, teda pracovná frekvencia, je pomerne vysoká u každého typu. Hodnoty sú tak vysoké i preto, lebo frekvencia ktorú získame po syntéze je odhad, ktorý nezapočítava oneskorenie na vodičoch. To znamená že nezohľadňuje, kam a ako je dizajn do FPGA namapovaný. Pre reálne výsledky je preto nutné vykonať *par* (*Place and Route*) a zapísať až tieto hodnoty, viď. *príloha G.* Frekvencie po paru pre všetky syntetizované obvody generátorov však vychádzajú približne rovnaké ako po syntéze (v rozmedzí od 309 MHz po 495 MHz), čo je celkom reálne, keďže dizajn je veľmi jednoduchý a oneskorenie *LUT* a *MUX* na zvolenom FPGA čipe je maximálne okolo *1ns.*

Na záver môžeme konštatovať, že pokiaľ by sme implementovali iný typ generátora pseudonáhodných čísel (pre platformu FITKit), asi sa nám len ťažko podarí vytvoriť rýchlejší generátor. Maximálna pracovná frekvencia generátora implementovaného ako ukážková aplikácia pre FITKit je 137 MHz (výsledok po *Place and Route*).

6 Záver

Obsahom tejto práce bolo teoreticky preskúmať typy generátorov pseudonáhodných čísel a zhodnotiť ich kvalitu pomocou štatistických testov. Napokon bolo potrebné implementovať zvolené typy preskúmaných generátorov.

Celá úloha bola pomerne dosť zaujímavá. Pri hľadaní informácií o hardvérových generátoroch som narazil na skutočne mnoho zaujímavých pohľadov. Zistil som, že rôznych implementácii je nespočetne veľa a na mne bolo sa rozhodnúť, ktoré z typov si vyberiem. Obmedzil som sa teda na tie, ktoré boli a sú v praxi najčastejšie používané. Potom prišla na radu ich implementácia do takej podoby, aby som ich mohol jednoducho otestovať. Časť práce, ktorá sa venuje práve testovaniu bola tiež nemenej zaujímavá ako ta predchádzajúca. Zistil som aké typy a princípy testovania sa využívajú a sám som ich nakoniec i použil. Sklamaním pre mňa však boli výsledky testov v tom smere, že sa v praxi používa až príliš často naozaj málo kvalitný generátor LFSR v jeho jednoduchej podobe. Ako malou náplasťou môže byť ale fakt, že takýto typ 32 bitového generátora zaberá iba jeden logický blok na obvode FPGA. Jeho paralelná verzia je síce kvalitná, no pomerne dosť robustná. Pre tieto odlišnosti som sa rozhodol v práci i porovnať jednotlivé generátory popísané rozličnými spôsobmi, s rozličným zapojením, a výsledky syntézy týchto obvodov prehľadne zhrnúť. Toto všetko ma naučilo písať nielen syntetizovateľný kód ale zistil som, že je dobré porozumieť i cieľovej technológii. Vo všeobecnosti boli nové technológie a moderné princípy, s ktorými som sa pri tejto práci stretol, veľmi náučné a do môjho sveta mi priniesli naozaj veľa. Napokon som vlastnou implementáciou zistil, že výstupy generátorov po implementácií v hardvéri sú zhodné s tými, ktoré boli implementované ako ich vzor v jazyku C, čo je celkom logické.

V budúcnosti by práce na generátoroch mohli pokračovať smerom k zaujímavým formám spomínaných hybridných celulárnych automatov. Pokiaľ by ale i táto technika zlyhala, tak ako aj u tých jednoduchých CA, pokúsiť sa implementovať hardvérové generátory v FPGA za spoluúčasti skutočne náhodných fyzikálnych procesov by mohlo byť skutočne veľkou výzvou...

Literatúra

[1] Abdulai, M.: *Inexpensive Parallel Random Number Generator for Configurable Hardware* [online]. Summer Undergraduate Program in Engineering Research (SUPERB). University of California, Berkley, 2003. [cit. apríl 2007]. URL: <http://www.eecs.berkeley.edu/Programs/ugrad/superb/papers2003/Mustapha%20Abdulai.pdf>.

[2] Alfke, P.: *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators* [online]. Application Note v 1.1, 1996. [cit. apríl 2007]. URL: <http://www.xilinx.com/bvdocs/appnotes/xapp052.pdf >.

[3] Boldiš, P.: *Bibliografické citace dokumentu podle CSN ISO 690 a CSN ISO 690-2: Část 1 – Citace: metodika a obecná pravidla.* Verze 3.3. ©1999–2004, poslední aktualizace 11.11. 2004. URL: <http://www.boldis.cz/citace/citace1.pdf>.

[4] Douša, J.: *Jazyk VHDL*. Praha: Vydavatelství ČVUT. 2003. ISBN 80-01-02670-1. 76s.

[5] Drutarovský, M., et al.: *A Simple PLL-Based True Random Number Generator for Embedded Digital Systems.* In Computing and Informatics, volume 23, number 5, 2004. str. 501-516. ISSN 1335-9150.

[6] Fajmon, B., Růžičková, I.: *Numerická matematika a pravděpodobnost* [Studijní opora k předmětu INM]. FEKT VUT v Brně, 2006.

[7] *FITKit* [online]. [cit. apríl 2007]. URL: <http://merlin.fit.vutbr.cz/FITkit/>.

[8] Fučík, O.: *Návrh číslicových systému* [Studijní opora k předmětu]. FIT VUT v Brně, 2007.

[9] George, M., Alfke, P.: *Linear Feedback Shift Registers in Virtex Devices* [online]. Application Note v 1.2, 2001. [cit. marec 2007]. URL: <http://www.xilinx.com/bvdocs/appnotes/xapp210.pdf>.

[10] Gerosa, A., Bernardini, R., Pietri, S.: *A fully integrated 8-bit, 20MHz, truly random numbers generator, based on a chaotic system.* In Southwest Symposium on Mixed-Signal Design, 2001. str. 87-92. ISBN: 0-7803-6742-1.

[11] Healy, D.: *Understanding Linear Feedback Shift Registers* [online]. [cit. apríl 2007]. URL: <http://www.yikes.com/~ptolemy/lfsr_web/>.

[12] Herout, P.: *Učebnice jazyka C.* České Budějovice: KOOP, třetí vydání, 2001. ISBN 80-85828-21-9. 269s.

[13] Christof, P.: *Past and Future of Cryptographic Engineering* [online]. European Competence for IT Security, 2003. [cit. marec 2007]. URL: <http://www.crypto.ruhr-uni-bochum.de>.

[14] Koopman, P.: *Maximal Lenght LFSR Feedback Terms* [online]. [cit. marec 2007]. URL: <http://www.ece.cmu.edu/~koopman/lfsr/index.html>.

[15] Kořenek, J., Martínek T.: *Návrh číslicových systému* [Studijní materiály k předmětu INC]. FIT VUT Brně, 2007.

[16] Lim, S., Miller, A.: *LFSRs as Functional Blocks in Wireless Applications* [online]. Application Note v 1.1, 2001. [cit. marec 2007].
URL: <http://www.xilinx.com/bvdocs/appnotes/xapp220.pdf>.

[17] Marsaglia, G.: *DIEHARD* [online]. Diehard Battery of Tests of Randomness. [cit. marec 2007].
URL: <http://www.stat.fsu.edu/pub/diehard/>.

[18] Peringer, P.: *Modelování a simulace* [Studijní opora k předmětu IMS]. FIT VUT v Brně. 2006.

[19] *Random.org.* [online]. [cit. marec 2007]. URL: <http://www.random.org/>.

[20] Shackleford, B., et al.: *High-Performance Cellular Automata Random Number Generators for Embedded Probabilistic Computing Systems.* In The 2002 NASA/DoD Conference on Evolvable Hardware (EH'02), 2002. str. 191-200. ISBN: 0-7695-1718-8.

[21] Synario Desing Automation: *VHDL Reference Manual* [online]. [cit. apríl 2007].
URL: <http://www.fm.vslib.cz/~kes/data/vhdl_ref.pdf/>.

[22] Tsoi, K.H., Lejny K.H., Leong P.H.W.: *Compact FPGA-based True and Pseudo Random Number Generators.* In Field-Programmable Custom Computing Machines, volume 11, 2003. str. 51-62. ISBN: 0-7695-1979-2.

[23] Vašíček, Z.: *Obvodová realizace vyvíjejících se systému.* Diplomová práce, Brno. FIT VUT v Brně, 2006.

[24] Weisstein, E.: *Primitive Polynomial* [online]. Wolfram MathWorld. [cit. január 2007].
URL: <http://mathworld.wolfram.com/PrimitivePolynomial.html>.

[25] Wikipedia contributors: *Birthday paradox* [online]. Wikipedia, The Free Encyclopedia. [cit. marec 2007]. URL:
<http://en.wikipedia.org/w/index.php?title=Birthday_paradox&oldid=123601563>.

[26] Wikipedia contributors: *Blum Blum Shub* [online]. Wikipedia, The Free Encyclopedia [cit. február 2007]. URL:
<http://en.wikipedia.org/w/index.php?title=Blum_Blum_Shub&oldid=106032099>.

[27] Wikipedia contributors: *Diehard tests* [online]. Wikipedia, The Free Encyclopedia. [cit. marec 2007]. URL:
<http://en.wikipedia.org/w/index.php?title=Diehard_tests&oldid=95128683>.

[28] Wikipedia contributors: *Distribution (mathematics)* [online]. Wikipedia, The Free Encyclopedia. [cit. marec 2007]. URL:
<http://en.wikipedia.org/w/index.php?title=Distribution_%28mathematics%29&oldid=125591508>.

[29] Wikipedia contributors: *Fortuna (PRNG)* [online]. Wikipedia, The Free Encyclopedia [cit. február 2007]. URL:
<http://en.wikipedia.org/w/index.php?title=Fortuna_%28PRNG%29&oldid=97168357>.

[30] Wikipedia contributors: *Infinite monkey theorem* [online]. Wikipedia, The Free Encyclopedia. [cit. marec 2007]. URL: <http://en.wikipedia.org/w/index.php?title=Infinite_monkey_theorem&oldid=125820939>.

[31] Wikipedia contributors: *Maurice Kendall* [online]. Wikipedia, The Free Encyclopedia. [cit. február 2007]. URL: <http://en.wikipedia.org/w/index.php?title=Maurice_Kendall&oldid=113928735>.

[32] Wikipedia contributors: *Mersenne twister* [online]. Wikipedia, The Free Encyclopedia [cit. február 2007]. URL: <http://en.wikipedia.org/w/index.php?title=Mersenne_twister&oldid=123450074>.

[33] Wikipedia contributors: *Randomness* [online]. Wikipedia, The Free Encyclopedia [cit. február 2007]. URL: <http://en.wikipedia.org/w/index.php?title=Randomness&oldid=125798267>.

[34] Wolfram, S.: *Random sequence generation by cellular automata* [online]. [cit. december 2006]. URL: <http://www.stephenwolfram.com/publications/articles/ca/86-random/>.

[35] Xilinx, Inc.: *Linear Feedback Shift Register* [online]. Product Specification v 3.0. 2003 [cit. január 2007]. URL: <http://www.xilinx.com/ipcenter/catalog/logicore/docs/lfsr.pdf>.

Zoznam príloh

Príloha A. *Tabuľka najlepších pozícií odberov signálu (taps) pre LFSR s maximálnou dĺžkou periódy.*

Príloha B. *Výsledky testov Diehard na skúmaných generátoroch.*

Príloha C. *Sumarizované výsledky testov Diehard.*

Príloha D. *Výsledky sériového testu.*

Príloha E. *Bitové zobrazenie výstupu jednotlivých porovnávaných generátorov.*

Príloha F. *Výsledky syntézy zvolených obvodov.*

Príloha G. *Výsledky paru zvolených obvodov.*

Príloha H. *Návod na inštaláciu a použitie 16 bitového generátora pseudonáhodných čísel pre FITKit.*

Príloha I. *CD so zdrojovými kódmi.*

Príloha A

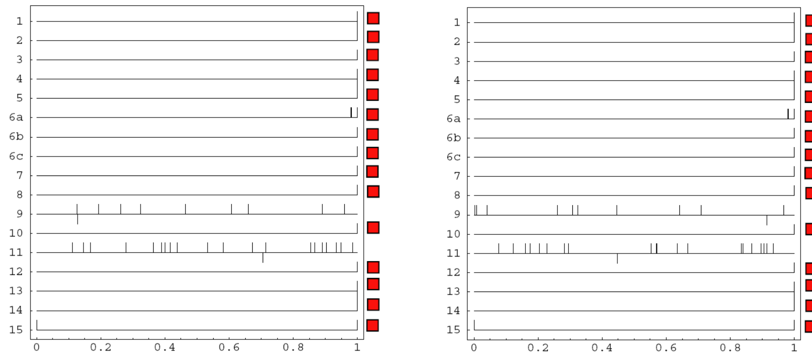
Pozície odberov signálu (*taps*) pre LFSR s maximálnou dĺžkou periódy. Dáta sú sumarizované podľa *Xilinx* [35]:

n	XNOR z	n	XNOR z	n	XNOR z	n	XNOR z
3	3,2	45	45,44,42,41	87	87,74	129	129,124
4	4,3	46	46,45,26,25	88	88,87,17,16	130	130,127
5	5,3	47	47,42	89	89,51	131	131,130,84,83
6	6,5	48	48,47,21,20	90	90,89,72,71	132	132,103
7	7,6	48	49,40	91	91,90,8,7	133	133,132,82,81
8	8,6,5,4	50	50,49,24,23	92	92,91,80,79	134	134,77
9	9,5	51	51,50,36,35	93	93,91	135	135,124
10	10,7	52	52,49	94	94,73	136	136,135,11,10
11	11,9	53	53,52,38,37	95	95,84	137	137,116
12	12,6,4,1	54	54,53,18,17	96	96,94,49,47	138	138,137,131,130
13	13,4,3,1	55	55,31	97	97,91	139	139,136,134,131
14	14,5,3,1	56	56,55,35,34	98	98,87	140	140,111
15	15,14	57	57,50	99	99,97,54,52	141	141,140,110,109
16	16,15,13,4	58	58,39	100	100,63	142	142,121
17	17,14	59	59,58,38,37	101	101,100,95,94	143	143,142,123,122
18	18,11	60	60,59	102	102,101,36,35	144	144,143,75,74
19	19,6,2,1	61	61,60,46,45	103	103,94	145	145,93
20	20,17	62	62,61,6,5	104	104,103,94,93	146	146,145,87,86
21	21,19	63	63,62	105	105,89	147	147,146,110,109
22	22,21	64	64,63,61,60	106	106,91	148	148,121
23	23,18	65	65,47	107	107,105,44,42	148	149,148,40,39
24	24,23,22,17	66	66,65,57,56	108	108,77	150	150,97
25	25,22	67	67,66,58,57	109	109,108,103,102	151	151,148
26	26,6,2,1	68	68,59	110	110,109,98,97	152	152,151,87,86
27	27,5,2,1	69	69,67,42,40	111	111,101	153	153,152
28	28,25	70	70,69,55,54	112	112,110,69,67	154	154,152,27,25
29	29,27	71	71,65	113	113,104	155	155,154,124,123
30	30,6,4,1	72	72,66,25,19	114	114,113,33,32	156	156,155,41,40
31	31,28	73	73,48	115	115,114,101,100	157	157,156,131,130
32	32,22,2,1	74	74,73,59,58	116	116,115,46,45	158	158,157,132,131
33	33,20	75	75,74,65,64	117	117,115,99,97	159	159,128
34	34,27,2,1	76	76,75,41,40	118	118,85	160	160,159,142,141
35	35,33	77	77,76,47,46	119	119,111	161	161,143
36	36,25	78	78,77,59,58	120	120,113,9,2	162	162,161,75,74
37	37,5,4,3,2,1	79	79,70	121	121,103	163	163,162,104,103
38	38,6,5,1	80	80,79,43,42	122	122,121,63,62	164	164,163,151,150
39	39,35	81	81,77	123	123,121	165	165,164,135,134
40	40,38,21,19	82	82,79,47,44	124	124,87	166	166,165,128,127
41	41,38	83	83,82,38,37	125	125,124,18,17	167	167,161
42	42,41,20,19	84	84,71	126	126,125,90,89	168	168,166,153,151
43	43,42,38,37	85	85,84,58,57	127	127,126		
44	44,43,18,17	86	86,85,74,73	128	128,126,101,99		

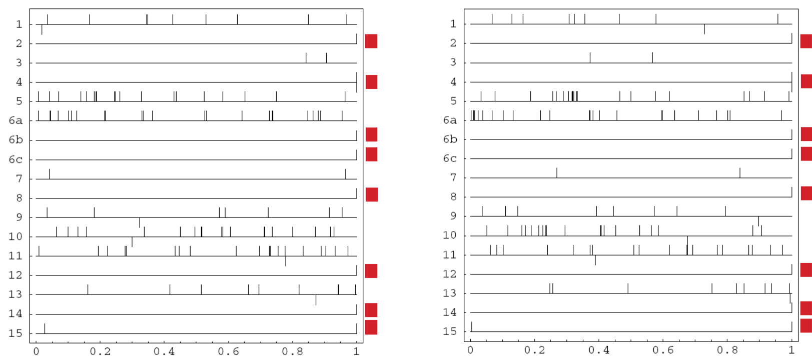
Príloha B

Výsledky testu *Diehard* na skúmaných generátoroch. Zobrazené sú testy pre 2 rôzne inicializačné hodnoty. Značka označuje *100%* zlyhanie daného generátora na danom teste.

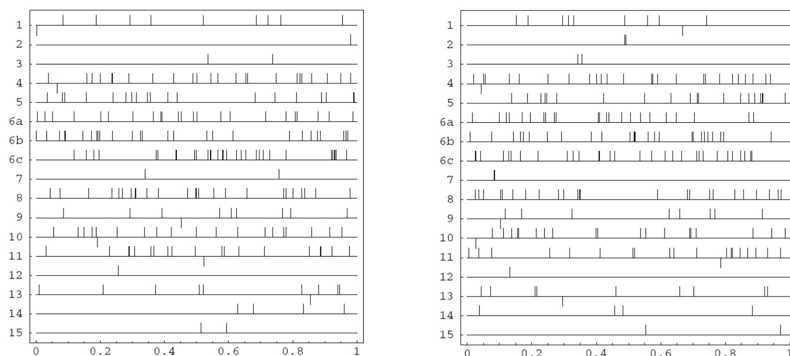
Generátor založený na princípe jednoduchého sériového LFSR (*serial-LFSR*):



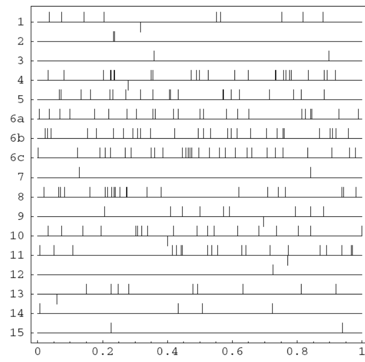
Generátor založený na princípe jednorozmerného celulárneho automatu (*1d-CA30*):

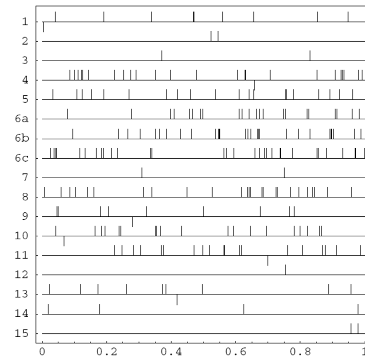


Generátor založený na princípe jednorozmerného celulárneho automatu (*1d-CA50745*):

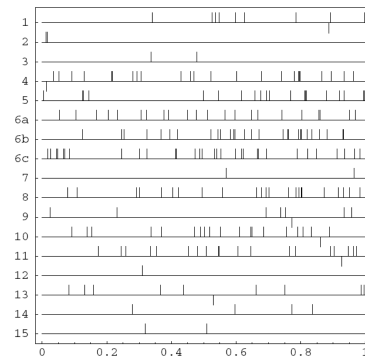


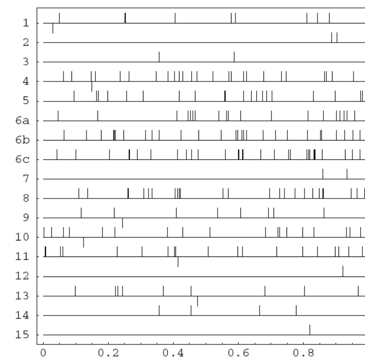
Generátor založený na princípe kombinovaného LFSR (*comb-LFSR*):





Generátor založený na princípe paralelnej konfigurácie LFSR (*paralell-LFSR*):





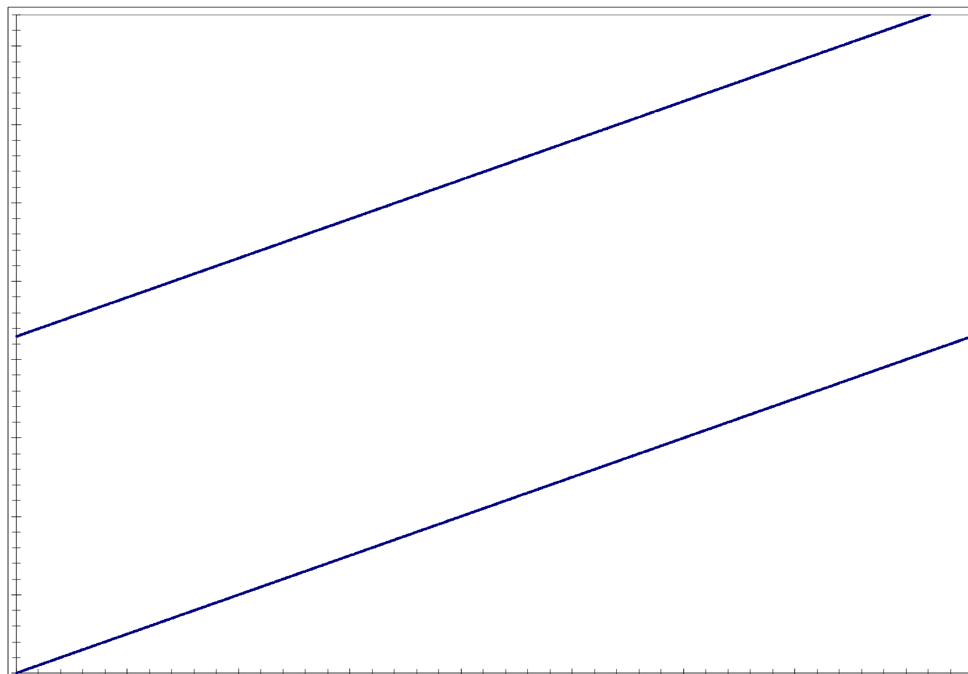
Príloha C

Tabuľka zobrazuje už sumarizované výsledky testov *Diehard* na jednotlivých generátoroch. Maximálne skóre (*Max score*) zobrazuje prípad, kedy by generátor zlyhal na všetkých testoch. *True* je naopak postupnosť, ktorá by mala byť skutočne náhodná [19].

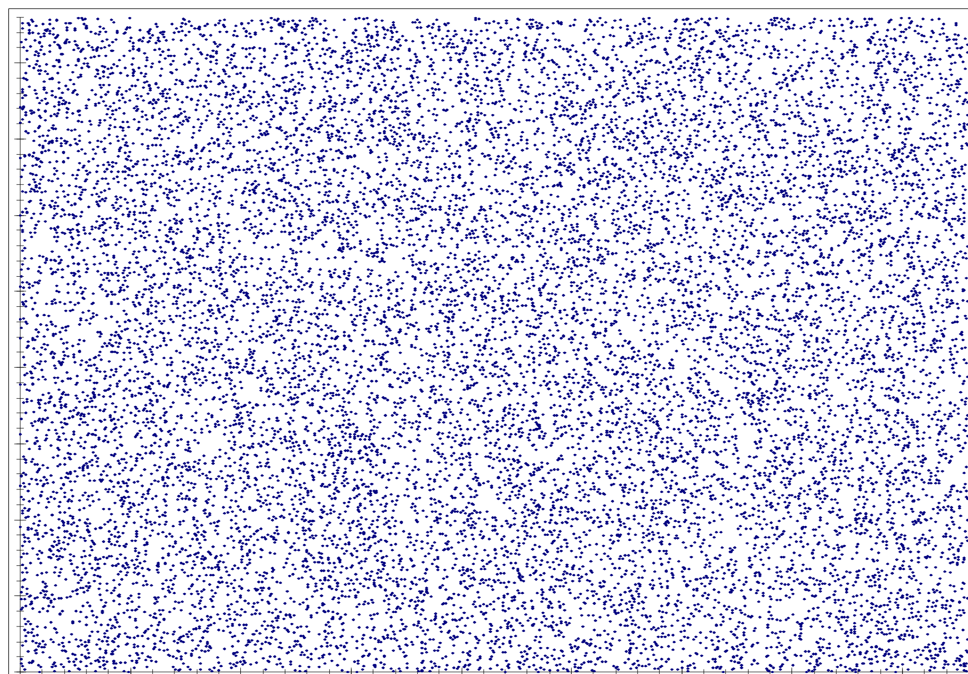
Test	***Max score***	***Serial LFSR***	***Parallel LFSR***	***Comb. LFSR***	***1D CA30***	***1D CA 50745***	***True***
Birthday	**36**	**36**	**2**	**8**	**0**	**8**	**0**
Overlapping permutation	**8**	**8**	**4**	**0**	**8**	**6**	**0**
Binary Rank 32x32	**8**	**8**	**6**	**2**	**2**	**6**	**0**
Binary Rank 6x	**104**	**104**	**8**	**40**	**104**	**80**	**4**
Bitstream	**80**	**80**	**0**	**0**	**60**	**60**	**4**
Overlapping pairs tests	**328**	**328**	**94**	**188**	**320**	**288**	**6**
Count the ones (stream)	**8**	**8**	**6**	**6**	**6**	**6**	**0**
Count the ones (specific)	**100**	**100**	**30**	**42**	**100**	**90**	**2**
Parking Lot	**44**	**4**	**0**	**0**	**4**	**2**	**0**
Minimum Distance	**4**	**4**	**0**	**0**	**2**	**2**	**0**
3D spheres	**84**	**4**	**2**	**0**	**4**	**4**	**4**
Squeeze	**4**	**4**	**0**	**0**	**4**	**2**	**0**
Overlapping Sums	**44**	**44**	**0**	**0**	**6**	**0**	**2**
Runs	**16**	**16**	**2**	**0**	**16**	**8**	**0**
Craps	**8**	**8**	**0**	**0**	**8**	**8**	**0**
Total	**876**	**756**	**154**	**286**	**644**	**570**	**22**

Príloha D

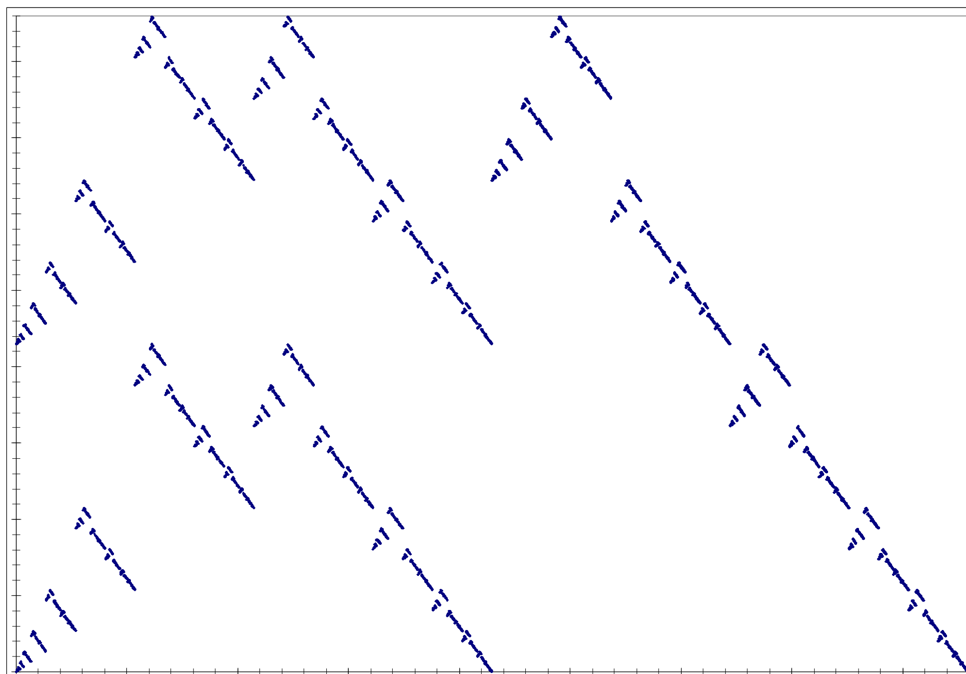
Výsledky sériového testu. Zobrazených vždy po *32 000* dvojíc.



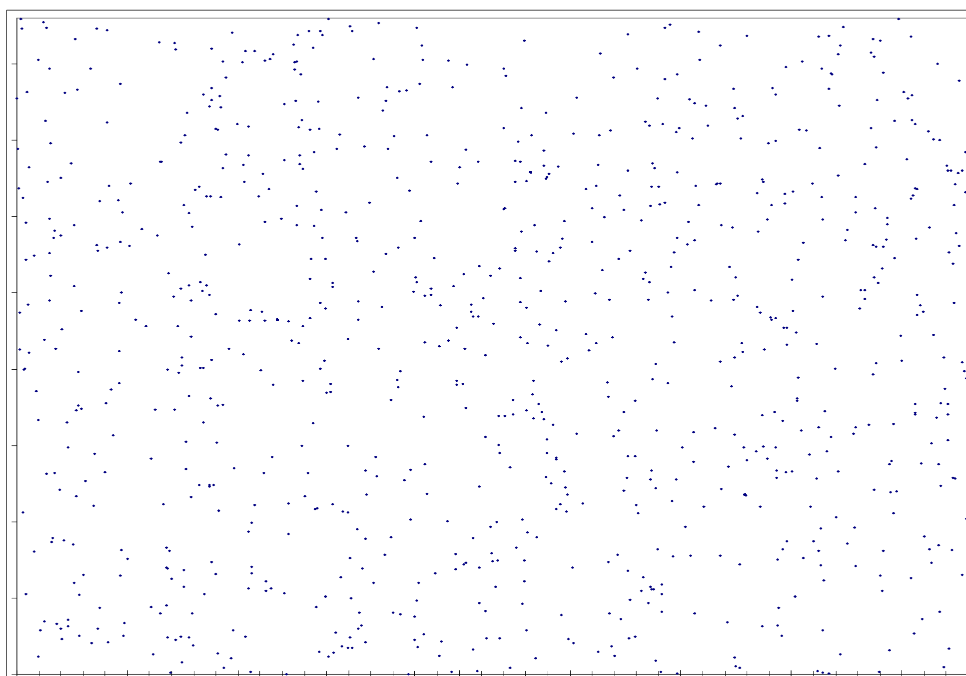
Sériový test jednoduchého LFSR.



Sériový test pre paralelné LFSR.



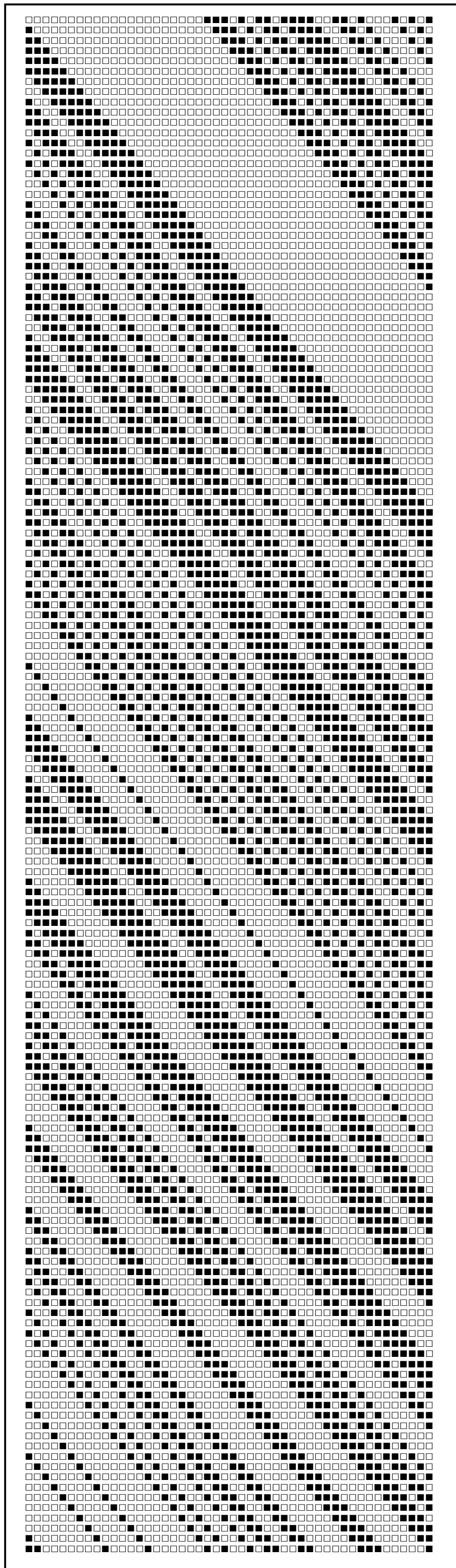
Sériový test pre CA30.



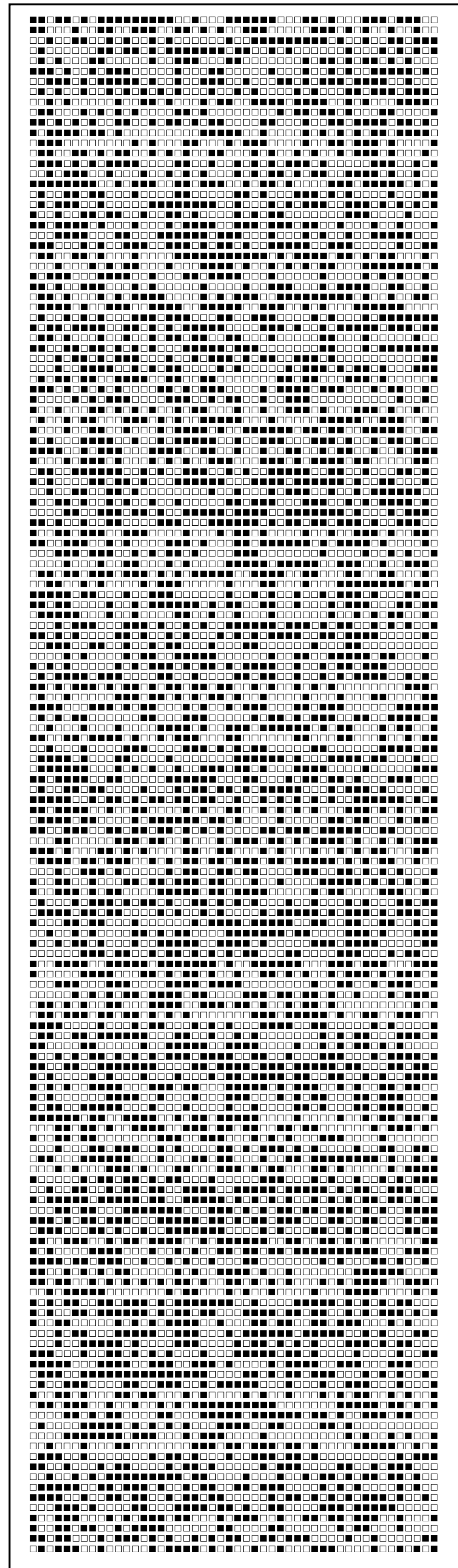
Sériový test pre CA50745.

Príloha E

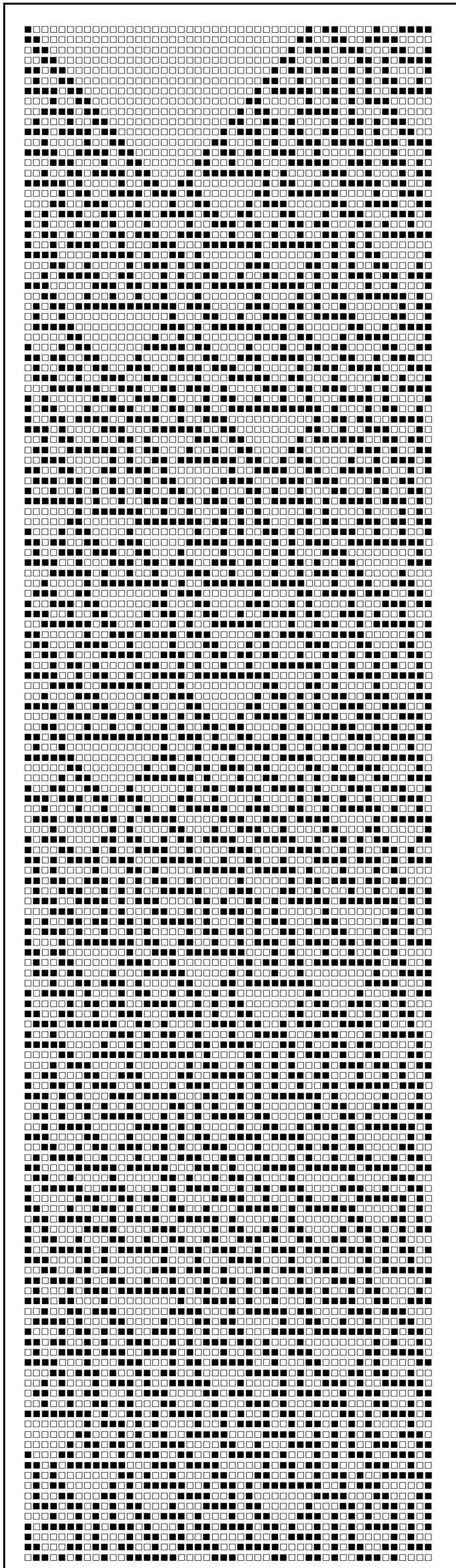
Bitové zobrazenie výstupu jednotlivých porovnávaných generátorov.



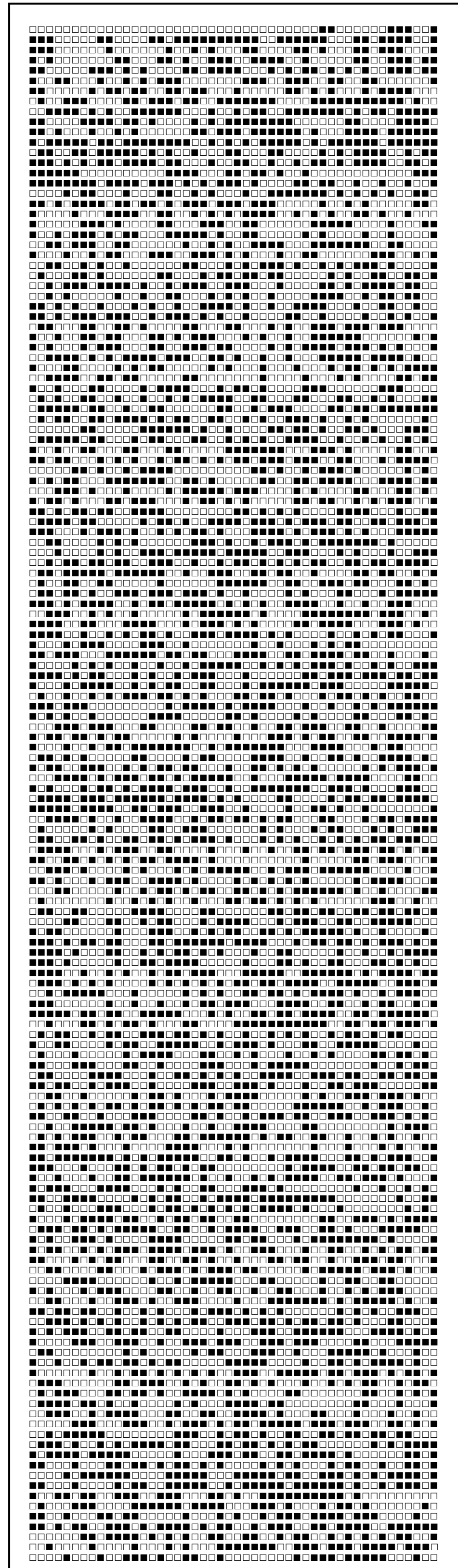
Bitové zobrazenie vygenerovaných čísiel pomocou sériového LFSR.



Bitové zobrazenie vygenerovaných čísiel paralelného zapojenia LFSR.



Bitové zobrazenie vygenerovaných čísiel pomocou 1D CA30.



Bitové zobrazenie vygenerovaných čísiel pomocou 1D CA50745.

Príloha F

Tabuľka odhadu obsadenia (počet zabratých *Slice* prvkov z celkového počtu *768*) a odhadu maximálnej pracovnej frekvencie (*f [MHz]*) generátora pseudonáhodných čísiel na čipe FPGA (*Spartan 3, XC3S50 PQ208. Xilinx*). Zobrazené sú výsledky **po *syntéze***. Obvody sú popísaných štruktúrne (pomocou klopných odvodov typu *D*) i behaviorálne. Generátory sú založené na princípe LFSR, pričom jednoduchý typ značí jediný register a paralelné zapojenie značí daný počet registrov LFSR vedľa seba, v zmysle predchádzajúceho výkladu. Sériová inicializácia značí sériovú inicializáciu vždy jedného z registrov LFSR. Paralelná značí naopak paralelný vstup i výstup dát.

Bitová dĺžka a štruktúra daná polynómom	Typ zapojenia LFSR	Štruktúrny popis				Behaviorálny popis			
		sériová inicializácia, bez resetu		paralelná inicializácia, s resetom		sériová inicializácia, bez resetu		sériová inicializácia, s resetom	
		počet Slice	*f [MHz]*	*počet Slice*	*f [MHz]*	*počet Slice*	*f [MHz]*	*počet Slice*	*f [MHz]*
4 $X^4+X^1+X^0$	*jednoduché*	2	338	4	479	2	338	3	479
	4 paralelne	6	338	-	-	9	338	11	479
8 $X^8+X^4+X^3+X^2+X^0$	*jednoduché*	3	338	6	473	3	338	6	379
	8 paralelne	21	338	-	-	31	338	10	379
16 $X^{16}+X^{12}+X^3+X^1+X^0$	*jednoduché*	3	338	10	339	3	338	10	379
	16 paralelne	50	338	-	-	62	338	156	379
32 $X^{32}+X^{31}+X^{30}+X^{10}+X^0$	*jednoduché*	2	316	19	339	3	316	19	379
	32 paralelne	103	316	-	-	121	316	606	379
64 $X^{64}+X^4+X^3+X^1+X^0$	*jednoduché*	3	316	38[*]	?[*]	4	316	38	379
	64 paralelne	303[*]	?[*]	-	-	325[*]	?[*]	>768[*]	?[*]

[*] Generátor popísaný týmto spôsobom zaberá omnoho viac prostriedkov (*Slices, IO Blocks*) ako je na zvolenom čipe FPGA k dispozícii, a preto ho nie je možné zhotoviť.

Príloha G

Tabuľka odhadu obsadenia (počet zabratých *Slice* prvkov z celkového počtu *768*) a odhadu maximálnej pracovnej frekvencie (*f [MHz]*) generátora pseudonáhodných čísiel na čipe FPGA (*Spartan 3, XC3S50 PQ208. Xilinx*). Zobrazené sú výsledky **po *pare*** *(Place and Route)*. Obvody sú popísaných štruktúrne (pomocou klopných odvodov typu *D*) i behaviorálne. Generátory sú založené na princípe LFSR, pričom jednoduchý typ značí jediný register a paralelné zapojenie značí daný počet registrov LFSR vedľa seba, v zmysle predchádzajúceho výkladu. Sériová inicializácia značí sériovú inicializáciu vždy jedného z registrov LFSR. Paralelná značí naopak paralelný vstup i výstup dát.

Bitová dĺžka a štruktúra daná polynómom	Typ zapojenia LFSR	Štruktúrny popis				Behaviorálny popis			
		sériová inicializácia, bez resetu		paralelná inicializácia, s resetom		sériová inicializácia, bez resetu		sériová inicializácia, s resetom	
		počet Slice	*f [MHz]*	*počet Slice*	*f [MHz]*	*počet Slice*	*f [MHz]*	*počet Slice*	*f [MHz]*
4	*jednoduché*	3	487	2	495	3	492	2	498
$X^4+X^1+X^0$	*4 paralelne*	10	355	-	-	9	307	8	377
8	*jednoduché*	5	408	4	357	5	432	5	355
$X^8+X^4+X^3+X^2+X^0$	*8 paralelne*	40	318	-	-	33	298	33	351
16	*jednoduché*	5	408	9	413	5	432	9	361
$X^{16}+X^{12}+X^3+X^1+X^0$	*16 paralelne*	80	337	-	-	65	339	129	261
32	*jednoduché*	4	396	17	321	4	432	17	355
$X^{32}+X^{31}+X^{30}+X^{10}+X^0$	*32 paralelne*	134	312	-	-	101	310	520	296
64	*jednoduché*	6	309	33[*]	?[*]	6	309	33	360
$X^{64}+X^4+X^3+X^1+X^0$	*64 paralelne*	406[*]	?[*]	-	-	340[*]	?[*]	>768[*]	?[*]

[*] Generátor popísaný týmto spôsobom zaberá omnoho viac prostriedkov (*Slices, IO Blocks*) ako je na zvolenom čipe FPGA k dispozícii, a preto ho nie je možné zhotoviť.

Príloha H

Návod na inštaláciu a použitie 16 bitového generátora pseudonáhodných čísel pre FITKit.

Pred samotným použitím FITKitu s ukážkovou aplikáciou generátora pseudonáhodných čísel je potrebné vykonať niekoľko krokov. Tým prvým by mala byť inštalácia potrebného softvérového vybavenia pre počítač, ku ktorému sa bude kit pripájať. Všetky tieto kroky je možné nájsť na oficiálnych stránkach projektu [7].

Po inštalácií a nastavení prostredia počítača (*COM porty*) môžeme pokračovať krokmi inštalácie generátora do kitu. Na tomto mieste je vhodné si na lokálny disk stiahnuť a rozbaliť aktuálnu verziu obsahu SVN [7]. Z priloženého CD disku ďalej skopírujeme program pre MCU a pre FPGA (*FITKit app*) do lokálnej zložky SVN (*konkrétne do apps*). V podložkách aplikácie pre MCU (*apps/FITKit app/sw*) a FPGA (*apps/FITKit app/top*) preložíme jednotlivé tieto časti príkazom *make*. Po preložení môžme aplikáciu pre MCU nahrať. To vykonáme príkazom *make load* (v *apps/FITKit app/sw*). Pokiaľ prebehlo nahranie úspešne, je možné sa pomocou terminálu [7] pripojiť k FITKitu. Pomocou terminálu, na ktorom zadáme *prog fpga,* nahrajme do FPGA konfiguráciu (*apps/FITKit app/top/output.bin*). Úspešným nahraním končí konfigurácia, a teda je možné generátor plne používať. Kroky vykonávane doposiaľ sú zhodné s tými, ktoré treba vykonať pred použitím ktorejkoľvek aplikácie z SVN.

Ovládanie generátora je opäť pomocou pripojeného terminálu. Je možné zadávať jeden z nasledujúcich príkazov s daným významom:

```
HELP     - zobrazí jednoduchú textovú nápoveď
STOP     - zastaví zobrazovanie generovaných čísel
START    - spustí zobrazovanie generovaných čísel
NEXT     - zobrazí ďalšiu generovanú hodnotu
CLEAR    - vynuluje obsah generátora (nastaví zakázaný stav)
RESET FPGA          - resetuje obsah fpga
SEED 0 to 2^16      - nastaví počiatočnú hodnotu generátora
```

Ostatné neznáme príkazy aplikácia ignoruje. Čísla, ktoré generátor generuje sa zobrazujú na displeji FITKitu. Aktuálna hodnota generátora sa taktiež zobrazí i po zadaní príkazu *HELP* priamo do terminálu.