FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

SEMINARSKA NALOGA PRI PREDMETU
KRIPTOGRAFIJA IN TEORIJA KODIRANJA 2

20. julij 2011

Implementacija generatorja psevdo-naključnih števil

Povzetek

Koračno alternirajoči generator (angl. Alternating Step Generator - ASG) spada v vrsto generatorjev psevdo-naključnih števil in je sestavljen iz treh podgeneratorjev. Glavna karatkeristična značilnost ASG-ja je, da enega izmed podgeneratorjev uporablja za kontroliranje notranje ure ostalih dveh. V seminarski nalogi sem implementiral ter opisal tovrstni generator, zanj napisal simulacijsko kodo, podal teoretično podlago ter kratko varnostno analizo.

Avtor:
Matic PEROVŠEK

Mentor:
Aleksandar JURIŠIĆ

Kazalo

1 Generatorji naključnih števil **2**

2 LFSR **2**

2.1 Linearna rekurzivna šifra 3

2.1.1 Lastnosti linearne rekurzivne šifre 3

2.2 Pomični register z linearno povratno zanko 4

3 Alternirajoči generator **4**

3.1 Definicija ASG 6

3.2 Perioda in linearna zahtevnost 6

3.3 Korelacijski napad na ASG 7

3.4 Implementacija 8

3.4.1 Vhodi in izhodi 8

3.4.2 Notranje spremenljivke procesa 9

3.4.3 Postopek 9

3.5 Simulacija in rezultati 10

4 Zaključek **12**

5 Priloga **15**

5.0.1 Datoteka asg.vhd 15

5.0.2 Datoteka asg_test.vhd 17

5.0.3 Datoteka xor_gates.vhd 20

1 Generatorji naključnih števil

Pri obravnavi kriptografskih sistemov pogosto naletimo na pojem naključnih vrednosti (skrivni ključ pri šifriranju DES, praštevili p in q pri RSA algoritmu,...). Vrednosti morajo biti vedno čimbolj naključne, da se onemogočijo napadi, ki uporabljajo požrešne metode za iskanje ključev. Zato so pomemben del kriptografskih sistemov ravno naključna števila. Naključne vrednosti bi lahko dobili s fizičnimi pojavi kot so razcepi ionov, meti kovancev, termičnimi šumi ...

V programski opremi se namesto zgoraj omenjenih fizičnih pojavov uporabljajo generatorji psevdo-naključnih števil (angl. pseudo-random number generator - PRNG), s katerimi se generirajo zaporedja števil, ki skušajo čimbolj aproksimirati lastnosti zaporedij naključnih števil. Zaporedja niso popolnoma naključna, saj so deterministično dobljena iz majhne množice začetnih vrednosti - začetnih stanj PRNG-ja imenovanih semena (angl. seed states). Ta so lahko bodisi eno izmed prej generiranih naključnih števil bodisi se jih dobi iz strojnih generatorjev psevdo-naključnih števil, lahko pa tudi iz vhodnih enot (vhod tipkovnice, miške, zakasnitve trdega diska, itd.). Zaželjeno je, da se pridobiva podatke iz čim večjega števila virov, saj se tako onemogočajo napadi na generatorjevo notranje stanje.

V kritpografiji tokovnih šifer se sporočila ponavadi šifrirajo z psevdonaključnimi zaporedji števil, zato je dobra generacija teh zaporedij še toliko pomembnejša. Generatorji ponavadi temeljijo na množici stanj končnih avtomatov, pogostokrat na linearnih povratnih pomičnih registrih. Da se izognemo nekaterim vrstam napadov, morajo generirati zaporedja z dolgo periodo ter visoko linearno kompleksnostjo ter dobrimi statističnimi lastnostmi.

Navkljub dejstvu, da se s strojnimi generatorji naključnih števil da generirati 'naključnejša' števila, se psevdo generatorji naključnih števil v praksi uporablajo v predvsem v kriptografiji ter s simulacijskimi nameni v proceduralni generaciji [4].

2 LFSR

Linearni povratni pomični register (angl. linear feedback shift register - LFSR) je sestavni del mnogih PRNG, mednje sodi tudi ASG. LFSR je popularen zaradi lahke implementacije ter dolge periode. V tem razdelku bomo opisali linerno rekurzivno šifro, na kateri temelji, predstavili dobre lastnosti

zaradi katerih se uporablja ter na koncu podali še primer.

2.1 Linearna rekurzivna šifra

Linearna rekurzivna šifra je sinhrona tokovna šifra [3], pri kateri je $B = C = K = \mathbb{Z}_s$, pri čemer je B končna množica čistopisov (angl. plaintext), C končna množica možnih tajnopisov (angl. ciphertext) ter K prostor ključev, $\mathbb{Z}_s$ pa je končna množica celih števil po modulu s, kjer $s \in \mathbb{N}$. Zaporedje ključev linearne rekurzivne šifre je določeno z linearno rekurzivno enačbo reda m s konstantnimi koeficienti nad $\mathbb{Z}_s$. Naj bo $z_m z_{m-1} \ldots z_1$ zadnjih m generiranih ključev, pri čemer je m stopnja rekurzivne šifre in $m \in \mathbb{N}$. Začetni ključ določi vrednosti $z_m z_{m-1} ... z_1$ za prvi korak. Nov koeficient izračunamo z enačbo:

$$z_0 = c_m z_m + c_{m-1} z_{m-1} + ... + c_1 z_1 \mod s = \sum_{j=1}^{m} c_j z_j \mod s \qquad (1)$$

2.1.1 Lastnosti linearne rekurzivne šifre

Veliko generatorjev toka temelji na lineranih rekurzivnih šifrah zaradi hitrosti ter dobrih lastnosti, ki jih posedujejo [1],[7]:

- dobre statistične lastnosti,
- dolga perioda.

Žal pa imajo slabo lastnost nizke linearne zahtevnosti $L(s)$, ki jo definiramo kot:

$$L(s) = \begin{cases} 0 & ; s \text{ je ničelno zaporedje} \\ \infty & ; \text{ noben LFSR ne geneira s} \\ \text{dolžina najkrajšega LFSR, ki generira } s & ; \text{ sicer} \end{cases} \qquad (2)$$

V teoriji polinomskih deliteljev obstaja veliko različnih lastnosti zaporedij, ki so dobljena s posamezni polinomi. Za generiranje so najpomembnejši primitivni polinomi. Ti nam omogočajo, da dosežemo tako psevdo-naključno zaporedje, ki bo imelo glede na stopnjo LFSR maksimalno dolžino.

Nerazcepen polinom $p(x)$ je primitiven, če je polinom x generator grupe vseh neničelnih elementov faktorske grupe $\mathbb{Z}_2[x]/p(x)$ [13]. Dolžina zaporedja periode ustvarjene s primitivnim polinomom je tako enaka $2^n - 1$ bitov [8],[9].

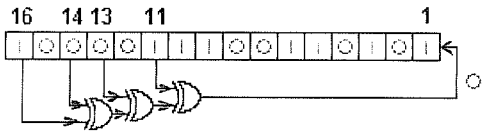
Biti	Primitivni polinom	Perioda
2	$x^2 + x + 1$	3
3	$x^3 + x^2 + 1$	7
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
10	$x^{10} + x^7 + 1$	1023
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	16383
16	$x^{16} + x^{14} + x^{13} + x^{11} + 1$	65535
50	$x^{50} + x^{27} + x^{26} + 1$	1125899906842623
100	$x^{100} + x^{37} + 1$	1.26 E+30
150	$x^{150} + x^{53} + 1$	1.42E+45
200	$x^{200} + x^{42} + x^{41} + x + 1$	1.6E+60

Tabela 1: Primitivni polinomi nekaterih potenc. Povzeto po [9].

Primitivne polinome iz tabele 1 za LFSR-je imenujemo tudi karakteristični polinomi. Obstaja povezava med koeficienti ter karakterističnimi polinomi. Za LFSR, ki ustreza enačbi (1), obstaja karakteristični polinom $p(x) = a_m x^m + a_{m-1} x^{m-1} + \ldots + a_1 x + 1$, ki pripada končnemu obsegu $GF(2^m)$. Za vsak koeficient c_j iz enačbe 1 velja $c_j = a_j$.

2.2 Pomični register z linearno povratno zanko

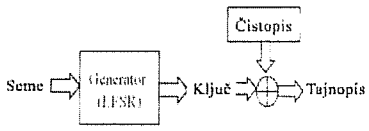
Linearno rekurzivno šifro enostavno implementiramo v strojni opremi z LFSR. LFSR je velikosti $m \in \mathbb{N}$ in ima notranjo uro. Za m velja: $m \geq 16$. V pomičnem registru je na začetku inicializacijski vektor $(z_m, \ldots, z_1)$ enak začetni vrednosti oz. prvotnemu ključu. Na vsakem koraku LFSR-jeve ure z enačbo 1 izračunamo z_0, ter izpišemo z_m. Nato $z_{m-1}, \ldots, z_1$ pomaknemo za eno mesto v levo ter na položaj z_1 zapišemo novo vrednost z_0.



Slika 1: LFSR v standardni obliki. Za ta primer velja $m = 16$, karakteristični polinom pa je enak $x^{16} + x^{14} + x^{13} + x^{11} + 1$.

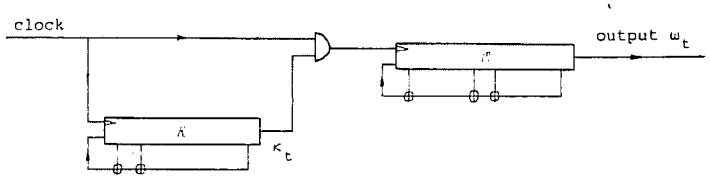
3 Alternirajoči generator

Zaradi dobrih statističnih lastnosti ter dolge periode se linearne rekurzivne šifre uporablja kot sestavni del 'pravih' tokovnih šifer. Linearno zahtevnost izboljšamo z dodajanjem nelinearnih elementov.



Slika 2: LFSR kot tokovna šifra

Eden izmed načinov, kako zadovoljiti vsem lastnostim iz poglavlja 2.1.1 je, da uro enega LFSR-ja kontroliramo s pomočjo drugega LFSR-ja. Primer generatorja, ki to počne, je t.i. stop-and-go generator. Njegova slaba lastnost je, da lahko potrebuje tudi več urinih ciklov za generiranje enega psevdonaključnega bita.



Slika 3: Shema stop-and-go generatorja

Pri step-and-go generatorju se izhod LFSR-ja M ponovi vsakič, ko LFSR K vrne 0. To nam na eni strani prinese dolge periode in visoko linearno kompleksnost, po drugi strani pa to vedno pomeni slabo statistiko (npr. $p(00) \cong p(11) \cong 3/8$, $p(01) \cong p(10) \cong 1/8$, kjer je $p(XY)$ verjetnost, da algoritem vrne zaporedoma X in Y). Poleg tega pa dejstvo, da se izhod ω_t spremeni le, ko je κ_t=1, določa eno polovico vseh enic, ki so v zaporedju K, kar lahko močno olajša rekonstrukcijo zaporedja K. Podobne pomanjkljivosti obstajajo v vseh znanih stop-and-go generatorjih.

3.1 Definicija ASG

Koračno alternirajoči generator (angl. Alternating Step Generator - ASG) je tesno povezan s step-and-go generatorjem, ohranja namreč vse njegove pozitivne lastnosti ter odpravlja pomanjkljivosti.

ASG je sestavljen iz treh podgeneratorjev $K, M, \overline{M}$, ki so medseboj povezani tako, da se M izvede, ko je izhod podegeneratorja K enak 1, $\overline{M}$ pa, ko je 0.

Matematično se da ASG generator opisati takole: Naj bodo $\kappa, \mu, \overline{\mu}$ zaporedja, dobljena z generatorji $K, M, \overline{M}$, če so njihove notranje ure med seboj neodvisne. Če definiramo $f_t := \sum_{s=o}^{t-1} k_s$, kjer $s \in \mathbb{N}_0$, $t \in \mathbb{N}_0$, in je k_s s-ti člen zaporedja κ ter velja $\overline{f}_t = t - f_t$, potem se da izhod ω_t opisati z:

$$\omega_t = \mu_{f_t} \oplus \overline{\mu}_{\overline{f}_t} \tag{3}$$

3.2 Perioda in linearna zahtevnost

V tem poglavju bomo podali linearno zahtevnost ter periodo ASG generatorja. Za izrek o periodi in linearni zahtevnosti potrebujemo definicijo de Bruijnovega zaporedja.

Definicija 1. *De Bruijnovo*[1] *zaporedje B(k, n) je ciklično zaporedje dane abecede A z dolžino k, za katerega se vsako možno podzaporedje dolžine n iz A pojavi kot zaporedje znakov le enkrat [11].*

Izrek 1. *Če velja:*

- *κ je de Bruijnovo zaporedje s periodo 2^k,*
- *karakteristična polinoma $p(x)$ in $\overline{p}(x)$ zaporedij μ in $\overline{\mu}$ sta neracepna in različna ter imata stopnji m in $\overline{m}$, periodi pa M in $\overline{M}$,*
- *$M, \overline{M} > 1$; $\gcd(M, \overline{M}) = 1$,*

potem sta perioda T in linearna zahtevnost L enaki:

$$T = 2^k M \overline{M},$$
$$(m + \overline{m})2^{k-1} < L \leq (m + \overline{m})2^k.$$

Dokaz je kompleksen, podal ga je C.G. Günther v svojem delu [2].

[1] Nicolaas Govert de Bruijn je nizozemski matematik. Med njegova dela spadata De Bruijnovo zaporedje ter De Bruijn–Newmanova konstanta.

3.3 Korelacijski napad na ASG

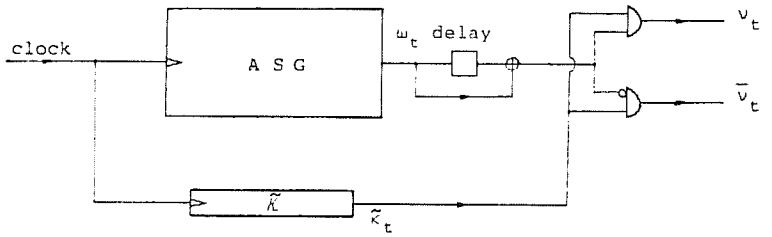
Korelacijski napadi so možni, ko obstaja določena korelacija med izhodnim stanjem enega izmed LFSR-jev ter izhodom funkcije (v našem primeru je to kar prvi LFSR κ), ki kombinira izhodnja stanja ostalih LFSR-jev. Če upoštevamo operacijo na izhodu ASG-ja na sliki 4 ter enačbo (3), dobimo:

$$\omega_t \ominus \omega_{t-1} = \begin{cases} \mu_{f_t} \oplus \overline{\mu}_{\overline{f}_t} \oplus \mu_{f_t-1} \oplus \overline{\mu}_{\overline{f}_t} \\ \overline{\mu}_{\overline{f}_t} \ominus \mu_{f_t} \oplus \overline{\mu}_{\overline{f}_t-1} \ominus \mu_{f_t} \end{cases} \quad \text{če } \kappa_{t-1} = \begin{cases} 1 \\ 0 \end{cases} . \tag{4}$$

Ker velja $p \ominus p = 0$, je poskusno zaporedje $\widetilde{K}$ na sliki 4 v korelaciji s K z relacijo:

$$\omega_t \oplus \omega_{t-1} = \begin{cases} \mu_{f_t} \oplus \mu_{f_t-1} \\ \overline{\mu}_{\overline{f}_t} \oplus \overline{\mu}_{\overline{f}_t-1} \end{cases} \quad \text{če } \kappa_{t-1} = \begin{cases} 1 \\ 0 \end{cases} . \tag{5}$$

Prav tako pa velja, da je vsota dveh linearnih periodičnih zaporedij spet linearno periodično zaporedje. Znak, da je $\widetilde{\kappa} = \kappa$, je v nepovečevanju linearne zahtevnost zaporedij v oz. $\overline{v}$ preko dolžine M oz. $\overline{M}$. To določa tudi Berlekamp-Massey algoritem[2].



Slika 4: Slika prikazuje možen korelacijski napad na ASG. Na izhodu ASG-ja se izračuna vrednost $\omega_t \ominus \omega_{t-1}$. Vzporedno z njim je vezan generator $\widetilde{\kappa}$. S pomočjo logičnih in vrat se izračunata v oz. $\overline{v}$. Ko se njuna linearna zahtevnost ne povečuje preko dolžin M oz. $\overline{M}$, za prvi notranji (izbirni) LFSR ASG-ja κ velja, da je $\widetilde{\kappa} = \kappa$.

Korelacijski napad le zmanjša število možnosti, katere mora preveriti napad z grobo silo na približno tretji koren možnosti pri temeljitem pregledu [2].

[2]Za Berlekamp-Massey algoritm je značilno, da za dano zaporedje konstruira najkrajši LFSR, ki to zaporedje generira. S tem algoritmom se da računati tudi linearno zahtevtnost zaporedja, glej [5].

Za tipičen parameter $T(\kappa) \approx 2^{127}$ bi bilo za pregled vseh možnosti potrebno 10^{18} let, če predpostavimo, da se jih na sekundo da pregledati 10^{12}.

3.4 Implementacija

FPGA (angl. Field Programmmable Gate Array) so prvič izdelali leta 1985 v podjetju Xilinx [12]. Gre za čip, ki je sestavljen iz blokov ter logičnih vrat, ki jih uporabnik lahko programira. Povezave in logične funkcije so shranjene na SRAM celicah, ki so rekonfigurabilne. Pri implementaciji sem se odločil za čip Xilinx Spartan3 verzije xc3s1000-5fg320, katerega ogrodje sestavljajo konfiguracijski logični bloki, vhodno/izhodni bloki, dvo-kanalni blokovni pomnilnik RAM, razni množilniki ter vezje za upravljanje z uro DCM [6]. Delovanje programa sem preizkusil na simulatorju tega čipa v programu Xilinx ISim. Tabela 2 prikazuje izkoriščenost tega čipa, ko simuliramo implementacijo, opisano v tem poglavju.

Že od samega začetka FPGA razvijalci uporabljajo HDL (angl. Hardware Descritpion Logic) jezike. Med najbolj popularna sodita Verilog in VHDL (angl. VHSIC HDL). Sta si zelo podobna vendar z majhno razliko: Verilog naj bi bil boljši za opisovanja na nivoju vrat, medtem ko naj bi bil VHDL boljši za abstrakcijo na višjem nivoju [6]. Sam sem se odločil implementirati generator psevdo-naključnih števil v VHDL programskem jeziku.

	Uporaba	Razpoložljivost
Skupno število pomikalnih registrov	98	15,360
Število 4 vhodnih vpoglednih tabel (angl. 4 input LUTs)	77	15,360
Število zasedenih rezin (angl. occupied Slices)	74	7,680
Št. vezanih vhodno/izhodnih blokov (angl. bonded IOBs)	132	221
Število multiplekserjev BUFGMUX	2	8

Tabela 2: Pregled koriščenja naprave Spratan3 xc3s1000-5fg320. Dobljeno pri simulaciji naše implementacije v programu Xilinx ISim.

3.4.1 Vhodi in izhodi

Ena izmed lastnosti programiranja v VHDL jeziku je, da programer najprej določi vhode in izhode programa. Proces *asg* ima 6 vhodov ter 5 izhodov. Vhodi *set_seed,seed0,seed1,seed2* so namenjeni ponastavitvi notranjih stanj treh LFSR-jev, katere ASG vsebuje. *Set_seed* je le logična vrednost, ki

sproži ponastavitev stanj, *seed0, seed1,seed2* pa so 16-bitne vrednosti, namenjene vsaka svoji LFSR enoti. Vhod *clk* je namenjen simulaciji notranje ure, medtem ko vhod *out_enable* generatorju napove, da naj začne generirati naključno število.

Med izhodi najdemo *istate_lfsr2, istate_lfsr2* in *istate_lfsr2*, ki so namenjeni le lažji predstavljivosti delovanja in se v končni verziji generatorja ne bi pojavljali. Prikazujejo namreč trenutna notranja stanja vseh treh LFSR-jev.

Izhod *rand_int* vrne celo naključno število, ki ga je ASG generiral. Dolžina števila je generična in se jo določi ob zagonu generatorja (generična spremenljivka *width*).

3.4.2 Notranje spremenljivke procesa

Kot smo že omenili, proces *asg* uporablja spremenljivke *istate_lfsr0, istate_lfsr1* ter *istate_lfsr2* za hrambo notranjih stanj. Te spremenljivke so v bistvu 16-bitni vektorji dvojiških vrednosti in so na začetku nastavljene na naključno izbrane vrednosti:

$$\text{'1101100011010101', '0101001101011100' in '1010110111010010'.}$$

V praksi bi ta stanja bila pridobljena iz nekega naključnega vhoda, npr. vhod miške. To da so notranja stanja 16-bitna, je namenjeno le lažji predstavljivosti, za praktično uporabo bi namreč morala biti stanja čim večja, saj 16-bitni LFSR lahko doseže le $2^{16} - 1$ različnih notranjih stanj.

Spremenljvike *output_bit2,output_bit1* ter *output_bit0* so le pomožne spremenljivke pri določanju izhodov, medtem ko spremenljivka *counter* skrbi, da se postopek generiranja ASG ponovi za vsak bit končnega izhodnega števila.

3.4.3 Postopek

Proces *asg* deluje nekako tako: neprestano preverja spremembe na signalih vhoda. Ko zazna spremembo na *clk*, najprej preveri, če je logični vhod *set_seed* pozitiven. Če je temu tako, takoj nastavi notranja stanja vseh treh LFSR-jev na vrednosti, ki jih dobi preko signalov *seed0, seed1,seed2*. Nato preveri, če je bila preko signala *out_enable* poslana zahteva po generaciji novega števila. Ko je zahteva izpolnjena, se notranji števec *counter* postavi na 0.

Sledi izvajanje prvega LFSR-ja. Najbolj levi bit notranjega stanja tega LFSR-ja predstavlja izhod, vsi ostali pa se premaknejo za eno mesto v levo. S pomočjo metode *xor_gates* se izračuna nova vrednost, ki se postavi na skrajno desni bit notranjega stanja. Na podlagi izhodnega bita se odloči za enega od drugih dveh LFSR-jev, v katerem se s pomočjo iste pomožne metode izračuna novo notranje stanje.

Metoda *xor_gates* uporablja za izračun novega stanja primitivni polinomom $x^{16} + x^{14} + x^{13} + x^{11} + 1$.

Na pomožnem vektorju *output_stream* se nato izvede bitni premik v levo, skrajno desni bit pa se postavi na vrednost, ki je izračunana z operacijo `xor` zadnjih dveh generiranih izhodnih vrednosti vsakega od drugih dveh LFSR-jev.

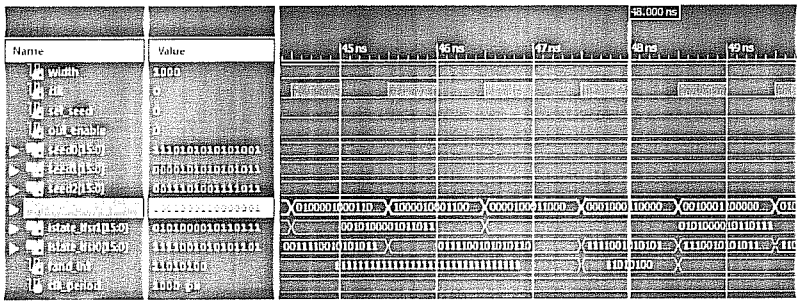
Sledi povečanje spremenljivke *counter* za 1. Ideja je, da se njegova vrednost poveča ob vsakem novogeneriranem bitu *output_stream*-a in ko doseže bitno dolžino izhodnega števila (podano preko generične spremenljivke *width*), se na izhod izpiše generirano naključno število, v nasprotnem primeru pa program vrača število -1.

3.5 Simulacija in rezultati

Za izvajanje simulacije sem uporabil Xilinx ISim Vhdl simulator. Najprej sem ustvaril signale *width*, *clk*, *set_seed*, *out_enable*, *seed0*, *seed1*, *seed2*, katere sem povezal na isto imenska vhodna vrata *asg* procesa ter signale *istate_flsr0*, *istate_lfsr1*, *istate_lfsr2 in rand_int* povezane na izhodna. Nato sem ustvaril proces *clk_process*, ki je namenjen simulaciji notranje ure. Ta proces spreminja svoj izhodni signal *clk* na vsake pol nanosekunde. Posledica tega je, da se vsako nanosekundo generira nov bit. Sprememba signala *clk* se lovi na začetku *asg* procesa in je vidna na vrhu slike 5. Ker je za proces *asg* potrebno še kaj več kot le spreminjanje ure, smo spisali še proces *stim_proc* z namenom stimulacije še drugih vhodov: *out_enable* ter *set_seed*.

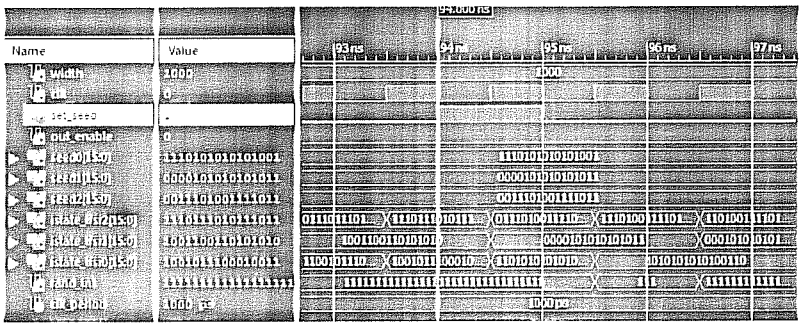
Proces *stim_proc* najprej čaka 40 ns, nato za 1 ns spremeni signal *out_enable* v 1, kar pove procesu *asg*, da naj prične generirati prvo naključno število. Čez 8ns (kolikor je s spremenljivko width nastavljena dolžina izhodnega števila) se na izhodnih vratih *rand_int* namesto vrednosti -1 pojavi prvo generirano število.

Na sliki 5 prvi stolpec predstavlja imena signalov, drugi stolpec prikazuje vrednost pozameznega signala v 48 ns, medtem ko na desni vidimo spreminjanje signala v odvisnosti od časa. Na sliki se jasno vidi preskok *clk* signala



Slika 5: Simulacijski prikaz v programu ISim - 45ns do 49ns.

vsake 0,5 ns. Prav tako se vidi, kako skrajno levi bit notranjega stanja prvega LFSR-ja določa vrednost notranjega stanja katerega od drugih dveh LFSR-jev se bo spremenila ob naslednjem visokem stanju signala *clk*. V 45 ns je najbolj levi bit notranjega stanja LFSR enak 0, zato ima LFSR0 spremenjeno stanje v 46ns. Podobno velja za stanje LFSR2 v 46 ns ter spremembo vrednosti stanja LFSR1 takoj zatem.



Slika 6: Simulacijski prikaz v programu ISim - 93ns do 97ns.

Slika 6 prikazuje, kako signal *set_seed* vpliva na notranja stanja. Signal se v 94ns spremeni v pozitivnega, nato se pol nano sekunde kasneje notranja stanja nastavijo na vrednosti, ki jih podajajo vhodi *seed*0('1110101010101001') *seed*1 ('0000101010101011') ter *seed*2 ('0011101001111011').
Ker so notranja stanja generirana deterministično iz začetnih stanj, se v primeru ponovitve celotne simulacije vračajo vedno iste vrednosti. Tabela 4 prikazuje vrednosti, ki jih vrača algoritem, če spremenimo dolžino izhodnih števil iz 8 na 16 bitov.

Čas	Binarna vrednost izhoda	Desetiška vrednost izhoda
48ns	11010100	212
69ns	10100111	167
96ns	111	7
126ns	10101000	168
172ns	1100000	96
213ns	10110100	180

Tabela 3: Naključna 8-bitna števila, ki jih vrne naša simulacija ob naključnih časih.

Čas	Binarna vrednost izhoda	Desetiška vrednost izhoda
64ns	1101010011111001	54521
77ns	1010011110101100	42924
104ns	11111111010	2042
134ns	1010100011100001	43233
180ns	110000010111111	24767
221ns	1011010001000111	46151

Tabela 4: Naključna 16-bitna števila, ki jih vrne naša simulacija ob naključnih časih.

4 Zaključek

LFSR-ji so statistično gledano odlični generatorji psevdo-naključnih števil, ponujajo namreč dobro porazdelitev ter enostavno implementacijo. Vendar se v praksi ne morejo dobro uporabiti, saj je njihovo zaporedje lahko napovedati. Koračno alternirajoči generatorji to pomanjkljivost zmanjšujejo z mnogo daljšimi periodami, navkljub temu pa ohranjajo enostavnost ter učinkovitost. Podana implementacija ASG-ja temelji na LFRS-jih s 16-bitnimi notranjimi stanji, kar je za praktično uporabo premalo. Da bi dosegli daljše periode preden se izhodno zaporedje začne ponavljati, bi uporabili LFSR-je z daljšimi notranjimi stanji, lahko pa bi izkoristili kar samo strukturo ASG-jev, ki nam omogoča kaskadno združevanje več ASG-jev. Namesto LFSR-jev lahko namreč kot podgenerator uporabimo kar druge ASG-je.

Literatura

[1] A. Kanso, *Modified clock-controlled alternating step generators*, Journal Computer Communications, vol. 32 Issue 5, March, 2009, pp. 787-799.

[2] C. Gunther, *Alternating step generators controlled by de Bruijn sequences*, Proceedings of Advances in Cryptology: EUROCRYPT 87, Amsterdam, The Netherlands, 13-15 April, LNCS 309, Springer, Berlin, 1988, pp. 5-14.

[3] J. Lano, *Cryptanalysis and design of synchronous stream ciphers*, June 2006, Heverlee, Belgium, pp. 16-18.

[4] O. Goldreich, S. Goldwasser *How to construct random funtions*, Journal of the Association for Computing Machinery, Vol.33, No.4, 1986, pp. 792-807

[5] N. Atti, G. M. Diaz–Toca and H. Lombard, *The Berlekamp-Massey Algorithm revisited*, Applicable Algebra in Engineerin, Communication and Computing, vol. 17, pp. 75-76

[6] FPGAs and Microcontrollers. *JC Electronica Homepage*. Web. 28 Feb. 2011. <http://www.jcelectronica.com/articles/FPGA and MCU.htm>.

[7] L. Simpson, E. Dawson, J. Golič, *LILI Keystream Generator*, Information Security Research Centre, Queensland University of Technology, Australia, 2001 pp. 2.

[8] J. No, Solomon W. Golomb, *Binary Pseudorandom Sequences of Period* $2^n - 1$ *with Ideal Autocorrelation*, IEEE Transactions in information theory Th. 44, March 1998, pp. 814-817.

[9] P. Alfk,*Efficient Shift Registers, LFSR Counters, and Long PseudoRandom Sequence Generators,* XAPP 052 July, 1996, pp. 5-7.

[10] I. Goldberg, D. Wagner, *Architectural considerations for cryptanalytic hardware*, CS252 Report, 1996, pp. 6-7.

[11] F.S. Annexstein, *Generating De Bruijn Sequences: An Efficient Implementation*, IEEE transactions on computers, vol. 46, 1997, pp. 1-2.

[12] W. Carter, K. Duong, R. H. Freeman, H. Hsieh, J. Y. Ja, *A User Programmable Reconfigurable Gate Array*, Custom Integrated Circuits Conference, May 1986, pp. 233-235

[13] T. Hansen, G. Mulle, *Primitive polynomials over finite fields*, Mathematics of computation vol. 59, 1992, p.p. 639-643

5 Priloga

5.0.1 Datoteka asg.vhd

```
1
2  -- Author:                Matic Perovšek
3  -- Create Date:           15:50:53 02/02/2011
4
5  -- Project Name:          generating step generator
6  -- Filename:              asg.vhd
7  -- Description:           Asg process - process for generating psevdo-random ↩
       number
8
9  -- Dependencies:          xor_taps.vhd
10 ----------------------------------------------------------------------------------
11
12 library ieee;
13 use ieee.std_logic_1164.all;
14 use ieee.std_logic_arith.all;
15 use ieee.std_logic_unsigned.all;
16 library work;
17 use work.xor_taps.ALL;
18
19 entity asg is
20     generic (width : integer := 16); --generic width of the output integer
21 port (
22           clk : in std_logic; --clock signal
23           out_enable : in std_logic; --signal to start generating the random↩
                 number
24
25           -- input states for LFSRs
26           set_seed : in std_logic;
27         seed0 : in std_logic_vector(15 downto 0);
28           seed1 : in std_logic_vector(15 downto 0);
29           seed2 : in std_logic_vector(15 downto 0);
30
31
32         rand_out : out std_logic; --last generated bit
33
34           --output states of LFSRs
35           istate_lfsr2 : out std_logic_vector(15 downto 0);
36           istate_lfsr1 : out std_logic_vector(15 downto 0);
37           istate_lfsr0 : out std_logic_vector(15 downto 0);
38
39           rand_int : out integer --output random number
40     );
41 end asg;
42
43 architecture Behavioral of asg is
44
45 begin
46
47 process(clk)
48
```

```
49 --help variables for LFSR0
50 variable state_lfsr0 : std_logic_vector (15 downto 0):= "1101100011010101"↩
       ; --(0 => '1',others => '0');
51 variable output_bit0 : std_logic := '0';
52
53 --help variables for LFSR1
54 variable state_lfsr1 : std_logic_vector (15 downto 0):="0101001101011100";↩
       --(0 => '1',others => '0');
55 variable output_bit1 : std_logic := '0';
56
57 --help variables for LFSR2
58 variable state_lfsr2 : std_logic_vector (15 downto 0):="1010110111010010";↩
       --(0 => '1',others => '0');
59 variable output_bit2 : std_logic := '0';
60
61 --random integer's currently generated bits
62 variable output_stream : std_logic_vector (width-1 downto 0):=(others => ↩
       '0');
63
64 --last generated bit variable
65 variable output_int : integer := -1;
66
67 --counter used for counting the number of bits already generated
68 variable counter : integer := width+1;
69
70 begin
71
72 if(rising_edge(clk)) then --if clock signal is high
73
74     --if requested set inner states of all LFSRs to given input values
75     if(set_seed = '1') then
76         state_lfsr0 := seed0;
77         state_lfsr1 := seed1;
78         state_lfsr2 := seed2;
79     end if;
80
81     --if requested start generating new output random integer
82     if(out_enable = '1') then
83         counter := 0;
84     end if;
85
86     --generate LFSR2's new state
87     output_bit2 := state_lfsr2(15); --save the output bit
88     state_lfsr2(15 downto 1) := state_lfsr2(14 downto 0); --shift register
89     state_lfsr2(0) := xor_gates(state_lfsr2,2); --calculate the new bit
90
91     if (output_bit2 = '1') then
92         --generate LFSR1's new state
93         output_bit1 := state_lfsr1(15); --save the output bit
94         state_lfsr1(15 downto 1) := state_lfsr1(14 downto 0);
95         state_lfsr1(0) := xor_gates(state_lfsr1,2);
96     elsif (output_bit2 = '0') then
97         --generate LFSR0's new state
98         output_bit0 := state_lfsr0(15); --save the output bit
99         state_lfsr0(15 downto 1) := state_lfsr0(14 downto 0);
100        state_lfsr0(0) := xor_gates(state_lfsr0,2);
101    end if;
```

```
102
103     counter:=counter+1;
104
105     --calculate new output stream state which might be used as the output ↩
            integer
106     output_stream(width-1 downto 1) := output_stream(width-2 downto 0); --↩
            shift the output stream
107     output_stream(0) := output_bit1 xor output_bit0; --calculate the new ↩
            bit from LFSR1's and LFSR0's output bits
108
109     --when counter equals the random integer's width output it, otherwise ↩
            return -1
110     if (counter=width) then
111         output_int := conv_integer(output_stream); --convert bool logic ↩
                vector to integer
112     else
113         output_int := -1;
114     end if;
115 end if;
116
117 --output the last generated bit
118 rand_out <= output_stream(0);
119
120 --send lfsr states to output ports
121 istate_lfsr2 <= state_lfsr2;
122 istate_lfsr1 <= state_lfsr1;
123 istate_lfsr0 <= state_lfsr0;
124
125 --output the random integer
126 rand_int<= output_int;
127 end process;
128
129 end Behavioral;
```

5.0.2 Datoteka asg_test.vhd

```
1  ----------------------------------------------------------------------------
2  -- Author:              Matic Perovšek
3  -- Create Date:        18:01:23 14/03/2011
4
5  -- Project Name:       alternating step generator
6  -- Filename:           asg_test.vhd
7  -- Description:        Asg test file - this file is used for simulating the ↩
       asg process by
8  --                     simulating the clock period and sending ↩
       signals to it at certain times
9
10 -- Dependencies:       asg.vhd, xor_taps.vhd
11 ----------------------------------------------------------------------------
12
13 library ieee;
14 use ieee.std_logic_1164.all;
15
```

```
16 entity asg_test is
17 end asg_test;
18
19 architecture behavior of asg_test is
20
21     --signals for asg process
22    signal clk,set_seed,out_enable : std_logic := '0';
23
24     --seed signals for asg, all values are randomly chosen
25    signal seed0 : std_logic_vector(15 downto 0) := "1110101010101001";
26     signal seed1 : std_logic_vector(15 downto 0) := "0000101010101011";
27     signal seed2 : std_logic_vector(15 downto 0) := "0011101001111011";
28
29     --signals which will show states of LFSRs
30     signal istate_lfsr2 : std_logic_vector(15 downto 0);
31     signal istate_lfsr1 : std_logic_vector(15 downto 0);
32     signal istate_lfsr0 : std_logic_vector(15 downto 0);
33
34     signal rand_int : integer; --signal for asg's output integer
35    constant clk_period : time := 1 ns; --clock period definitions
36
37
38 begin
39
40     -- entity instantiation for the asg and lfsr components.
41    uut: entity work.asg generic map (width => 16)     --changing the width ↩
         value here requires setting different top values in xor_gates
42     PORT MAP (
43            clk => clk,
44               set_seed => set_seed,
45               out_enable => out_enable,
46            seed0 => seed0,
47               seed1 => seed1,
48               seed2 => seed2,
49
50               istate_lfsr2 => istate_lfsr2,
51               istate_lfsr1 => istate_lfsr1,
52               istate_lfsr0 => istate_lfsr0,
53
54               rand_int => rand_int
55         );
56
57    -- Clock process - process which simulates clock by changing the signal↩
         every clk_period/2 nanoseconds
58    clk_process :process
59    begin
60         clk <= '0';
61         wait for clk_period/2;
62         clk <= '1';
63         wait for clk_period/2;
64    end process;
65
66    -- Applying stimulation inputs.
67    stim_proc: process
68    begin
69
70         wait for 40 ns;
```

```
71          out_enable <= '1'; --start generating the random number, which ↩
                will be returned after width nanoseconds
72          wait for 1 ns;
73          out_enable <= '0'; --"clear" the signal, so that asg doesn't reset↩
                the counter every clock period
74
75          wait for 20 ns;
76          out_enable <= '1';
77          wait for 1 ns;
78          out_enable <= '0';
79
80          wait for 26 ns;
81          out_enable <= '1';
82          wait for 1 ns;
83          out_enable <= '0';
84
85          wait for 5 ns;
86          set_seed <= '1';      --tell the asg process to use the new seed ↩
                values
87          wait for 1 ns;
88          set_seed <= '0';
89
90          wait for 24 ns;
91          out_enable <= '1';
92          wait for 1 ns;
93          out_enable <= '0';
94
95          wait for 43 ns;
96          out_enable <= '1';
97          wait for 1 ns;
98          out_enable <= '0';
99
100         wait for 40 ns;
101         out_enable <= '1';
102         wait for 1 ns;
103         out_enable <= '0';
104
105     wait;
106   end process;
107
108 END;
```

5.0.3 Datoteka xor_gates.vhd

```
1  ----------------------------------------------------------------------------
2  -- Author:                Matic Perovšek
3  -- Create Date:           17:33:13 14/03/2011
4
5  -- Project Name:          alternating step generator
6  -- Filename:              xor_taps.vhd
7  -- Description:           xor taps function - funct.on for generating new bit ↩
       for LFSRs
8  --                              after shifting (XORing from tap values)
9
10 -- Dependencies:          /
11 ----------------------------------------------------------------------------
12 library ieee;
13 use ieee.std_logic_1164.all;
14 use ieee.std_logic_arith.all;
15 use ieee.std_logic_unsigned.all;
16
17 package xor_taps is
18 function xor_gates( random : std_logic_vector; --array of bits
19                             lfsr_num : integer  --selection of the ↩
                                characteristic polynomial
20                             ) return std_logic;
21 end xor_taps;
22
23 --Package body starts from here.
24 package body xor_taps is
25 --function for XORing from tap values.
26 function xor_gates(     random : std_logic_vector;
27                             lfsr_num : integer ) return std_logic is
28 variable xor_out : std_logic:='0'; --output variable
29 variable rand : std_logic_vector(random'length-1 downto 0):=random;
30
31 begin
32 if(lfsr_num = 0) then
33       --characteristic polynomial: x^14+x^13+x^3+x^2+1
34     xor_out := rand(14-1) xor rand(13-1) xor rand(3-1) xor rand(2-1);
35 elsif(lfsr_num =1) then
36     --characteristic polynomial: x^15+x^14+1
37     xor_out := rand(15-1) xor rand(14-1);
38 elsif(lfsr_num = 2) then
39     --characteristic polynomial: x^16+x^14+x^13+x^11+1
40     xor_out := rand(16-1) xor rand(14-1) xor rand(13-1) xor rand(11-1);
41
42     --NOTE: the -1s are required because the array 'random' is already ↩
         shifted
43 end if;
44
45 return xor_out;
46 end xor_gates;
47 --END function for XORing using tap values.
48 end xor_taps;
49 --End of the package.
```