ZÁPADOČESKÁ UNIVERZITA V PLZNI

FAKULTA ELEKTROTECHNICKÁ

Katedra aplikované elektroniky a telekomunikací

DIPLOMOVÁ PRÁCE

Realizace šumového generátoru pro akustická měření

2012 **Jiří Mištera**

Realizace šumového generátoru pro akustická měření

Abstrakt

Tato práce se zabývá návrhem a realizací šumového generátoru v obvodu FPGA. Posuzují se zde vlastnosti jednotlivých přístupů k řešení a volí jedno z nich. Dále je dokumentována funkční implementace šumového generátoru. Součástí jsou také měření vytvářeného šumu a statistické testy generátoru pseudonáhodných čísel, na nichž je řešení založeno. V závěru jsou další návrhy na vylepšení stávajícího stavu.

Klíčová slova

Bílý šum, FPGA, Gaussovský šum, Generátor pseudonáhodných čísel, VHDL.

Implementation of the noise generator dedicated to acoustic measurements

Abstract

This thesis is focused on design of the noise generator. It is implemented in the VHDL language and dedicated to acoustic measurements. There are considered a few different solutions and chosen one of them. The chosen solution is described in detail and its implementation is documented. The part of thesis are statistical tests of pseudorandom numbers generator and measurements of output noise as well. In the conclusion are suggested possible changes and improvements.

Key words

FPGA, Gaussian noise, Pseudorandom nuber generator, VHDL, White noise.

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci, zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně, s použitím odborné literatury a pramenů uvedených v seznamu, který je součástí této diplomové práce.

Dále prohlašuji, že veškerý software, použitý při řešení této diplomové práce, je legální.

V Útušicích dne 4.5.2012

Jiří Mištera

Obsah

SEZNAM SYMBOLŮ A ZKRATEK 7

ÚVOD 8

1 ŠUM 9

1.1 VLASTNOSTI A PARAMETRY 9
1.2 VYUŽITÍ PRO AKUSTICKÁ MĚŘENÍ 11
1.3 ZDROJE 12

2 HARDWARE – VÝVOJOVÁ DESKA 13

2.1 FPGA 14
2.2 ZVUKOVÝ KODEK 14

3 DISKUSE ŘEŠENÍ V RÁMCI VÝVOJOVÉ DESKY 15

3.1 ČÍSLICOVÝ FILTR 15
3.2 GENERÁTOR PSEUDONÁHODNÝCH ČÍSEL 17
3.3 VARIANTY KOMPLETNÍHO SYSTÉMU 19

4 VYBRANÉ ŘEŠENÍ A FUNKCE 20

5 IMPLEMENTACE 21

5.1 CELKOVÉ SCHÉMA 23
5.2 BLOK ŠUMOVÉHO GENERÁTORU 24
5.3 GENERÁTOR PSEUDONÁHODNÝCH ČÍSEL 26
5.4 PŘEVOD NA NORMÁLNÍ ROZDĚLENÍ 30
5.5 PŘEVOD DO PLOVOUCÍ ŘÁDOVÉ ČÁRKY 33
5.6 DĚLENÍ NA DVA KANÁLY 35
5.7 ČÍSLICOVÝ FILTR 39
5.7.1 Formát instrukčního slova 45
5.7.2 Organizace paměti koeficientů 46
5.8 PŘEVOD DO KÓDU S POSUNUTOU NULOU 47
5.9 ZÁPIS DO FRONTY VZORKŮ 49
5.10 ROZHRANÍ MEZI FRONTAMI A KODEKEM 50
5.11 KONFIGURACE KODEKU 52

5.12 DALŠÍ POMOCNÉ ENTITY 57
5.13 OVLÁDÁNÍ 59

6 TEST GENERÁTORU PSEUDONÁHODNÝCH ČÍSEL..61

7 MĚŘENÍ VÝSTUPNÍHO ŠUMU...66

ZÁVĚR 68

SEZNAM LITERATURY A INFORMAČNÍCH ZDROJŮ 70

PŘÍLOHY...1

Seznam symbolů a zkratek

CDF	Cumulative Distribution Function (Kumulativní distribuční funkce)
FIR	Finite Impulse Response (Konečná impulzní odezva, resp. číslicový filtr s konečnou impulzní odezvou)
FPGA	Field – Programmable Gate Array (Pole programovatelých hradel)
I^2C	Inter – Integrated Circuit (IIC, Typ sběrnice pro komunikaci mezi integrovanými obvody)
IIR	Infinite Impulse Response (Nekonečná impulzní odezva, resp. číslicový filtr s nekonečnou impulzní odezvou)
LCG	Linear Congruential Generator (Lineární kongruentní generátor, algoritmus)
LFSR	Linear Feedback Shift Register (Posuvný registr s lineární zpětnou vazbou)
MWC	Multiply With Carry (Násobení s přenosem, algoritmus)
PLL	Phase – Locked Loop (Fázový závěs, smyčka fázového závěsu)
PRNG	Pseudo Random Numbers Generator (Generátor pseudonáhodných čísel)
PSD	Power Spectral Density (Výkonová spektrální hustota)
VHDL	VHSIC Hardware Description Language (Jazyk pro popis obvodů s velmi vysokou mírou integrace)

Úvod

V tomto projektu se snažím vytvořit číslicový systém, který by mohl sloužit jako šumový generátor pro elektroakustická měření. Abych zasadil tento problém do souvislostí, začínám v první kapitole s jednoduchou definicí šumu a jeho základními vlastnostmi. Dále uvádím oblasti, kde se dá využít při měření a končím přehledem způsobů, jak se v praxi šum vytváří. Tato kapitola je poměrně stručná, protože hlavní zájem chci věnovat vlastnímu řešení problému.

Ve druhé kapitole popisuji vývojovou desku. Jelikož je pouhým prostředkem k dosažení cíle, je zde jen nezbytný popis některých součástí. Tento popis poslouží jako definice prostředků a omezení, která mají přímý dopad na další řešení.

Ve třetí kapitole se snažím diskutovat možnosti, jak šumový generátor realizovat. Odhaduji zde složitost, rozsah a vlastnosti jednotlivých variant. Posuzuji nároky na paměť, výpočetní čas a další prostředky vzhledem ke stanoveným parametrům, které od řešení požaduji. Následující kapitola je pak shrnutí mých rozhodnutí. Popisuje varianty a postupy, které jsem vybral ke skutečné realizaci.

Pátá kapitola je stěžejní část celé práce. Dokumentuje šumový generátor jako celek a dále každou jeho součást. Popisuji zde funkci jednotlivých komponent, postupy činnosti, rozhraní a jejich vzájemné propojení do celku. U částí, které se věnují některému z klíčových kroků, podrobněji rozvádím jejich návrh. To se týká například číslicového filtru, nebo úpravy rozdělení náhodných čísel. Nakonec je zde popsané ovládání generátoru na vývojové desce.

V šesté kapitole se věnuji testování generátoru náhodných čísel. Uvádím zde výsledky testů na periodicitu, náhodnost a statistických testů normality. V další kapitole jsou výsledky z laboratorního měření výstupního šumu. Jsou zde vybraná spektra šumu v celém pásmu i se zařazenými číslicovými filtry.

V závěru se pak snažím hodnotit dosažené výsledky. Také navrhuji další úpravy směřující k vylepšení a problémy, které by bylo vhodné blíže prozkoumat.

1 Šum

Šum je velmi obecný pojem používaný napříč mnoha obory, a to nejen technickými. Jedná se o zpravidla nežádoucí náhodný signál, který se v neideálních systémech přidává k užitečné informaci a tím ji zkresluje.

V kontextu elektroniky se nejčastěji setkáváme s tepelným šumem. Ten je způsoben chaotickým pohybem nosičů náboje v materiálu. Pohyb nosičů náboje, obvykle elektronů, má dvě složky. Systematickou, který nazýváme elektrickým proudem, a náhodnou. Náhodný pohyb jednotlivých elektronů je sice o několik řádů rychlejší, ale v různých směrech. Součet těchto náhodných pohybů je při obrovském počtu elektronů téměř nulový, nicméně vyskytují se v něm fluktuace. Tyto fluktuace se pak projeví jako náhodný malý proud, který se přičítá k užitečnému signálu.

Několik dalších typů šumu, například výstřelový, souvisí s procesy v polovodičových přechodech. Stručný popis jednotlivých typů se zdroji a vlastnostmi uvádí například desátá kapitola [2]. Lze se také setkat s pojmem kvantizační šum. Ten však nemá fyzikální příčnu, jde spíše o užitečný způsob, jak nahlížet na přechod od spojitého signálu k diskrétnímu.

1.1 Vlastnosti a parametry

Podobně jako pro libovolný fyzikální signál, můžeme i pro šum zavést pojem střední a efektivní hodnota. Dále pak lze definovat tzv. crest faktor jako poměr špičkové a efektivní hodnoty signálu. Tento poměr je někdy uváděn přímo, jindy se používá vyjádření v decibelech.

Střední hodnota spojitá $$X_{avg} = \frac{1}{T}\int_0^T x(t)dt \tag{1.1.1}$$

Střední hodnota diskrétní $$X_{avg} = \frac{1}{N}\sum_{n=1}^{N} x_n \tag{1.1.2}$$

Efektivní hodnota spojitá $$X_{rms} = \sqrt{\frac{1}{T}\int_0^T x^2(t)dt} \tag{1.1.3}$$

Efektivní hodnota diskrétní $$X_{rms} = \sqrt{\frac{1}{N}\sum_{n=1}^{N} x_n^2} \tag{1.1.4}$$

Crest faktor $$C = \frac{|X_{peak}|}{X_{rms}} \tag{1.1.5}$$

Důležitou vlastností šumu je rozložení výkonu ve spektru. Podle toho lze definovat několik speciálních druhů šumu. Vžilo se pro ně pojmenování podle barev, které by měly, pokud by šlo o frekvenční pásmo světla. Budeme-li považovat šum za signál s výkonovou spektrální hustotou úměrnou $1/f^{\beta}$, pak lze druh šumu rozlišit hodnotu parametru β. Nejběžnější šumy jsou:

A) Bílý šum je takový, který má stejný výkon v libovolných pásmech stejné šířky, například 150 – 160 Hz a 1000 – 1010 Hz. Jeho výkonové spektrum je ploché a parametr $\beta = 0$.

B) Růžový šum má parametr $\beta = 1$. Jeho výkonová spektrální hustota je úměrná 1/f, klesá tedy přibližně o 3dB na oktávu. Stejný výkon je obsažen v pásmech, která mají stejný poměr hraničních frekvencí, například 150 – 160 Hz a 3000 – 3200 Hz.

C) Červený šum. Tento šum má parametr $\beta = 2$, lze ho vytvořit integrací bílého šumu, nebo pomocí algoritmu jednorozměrné náhodné procházky. Úzce souvisí s Brownovým pohybem, proto je někdy kvůli chybnému překladu jména Brown označován jako hnědý.

Jelikož je šum produktem jednoho či více náhodných procesů, jeho autokorelace by s rostoucí délkou vzorku měla konvergovat k Diracově delta distribuci. Dalším významným parametrem je statistické rozdělení amplitud. I v rámci jedné „barvy“ může mít šum různé hustoty pravděpodobnosti výskytu amplitud. Tyto parametry jsou vzájemně nezávislé.

Tepelný šum lze s velkou přesností považovat za bílý šum. Jeho amplituda má normální rozdělení popsané Gaussovou funkcí, což je, až na konstanty, e^{-x^2}. Má-li mít Gaussova funkce význam hustoty pravděpodobnosti, musí se normalizovat a po rozšíření o několik parametrů získá známý tvar :

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad (1.1.6)$$

Parametr μ je střední hodnota a σ standardní odchylka. Gaussova funkce je kladná a nenulová v celém oboru reálných čísel, nicméně s rostoucí vzdáleností od střední hodnoty extrémně rychle klesá k nule. Důsledkem toho velmi rychle klesá pravděpodonost výskytu velkých amplitud v gaussovském šumu. Pro ideální gaussovský šum by crest faktor neměl smysl, neboť X_{peak} může být teoreticky neomezená. V praxi je, nehledě na technická omezení, u konečně dlouhého diskrétního signálu amplituda vždy omezena. Crest faktor se s ohledem na normální rozdělení dá vztáhnout na podíl vzorků dosahujících maximální úrovně.

1.2 Využití pro akustická měření

V akustice se šumy využívají při několika typech měření. Bílý šum lze snadno využít ke zjištění frekvenční přenosové charakteristiky. Měřeným systémem či řetězecem se nechá tento šum projít a zaznamená se výstup. Výstupní signál se analyzuje a jelikož vstupní signál měl spektrum ploché, představuje spektrum výstupu přímo přenosovou charakteristiku. Schematicky je takové měření naznačeno na obrázku 1, v principu je ovšem možné použít i jiné druhy signálů.

Obr. 1 – Schéma měření přenosové charakterisitky šumem

Měřený systém je zde typicky reproduktor či reprosoutava, mikrofon, nebo v podstatě libovolná nízkofrekvenční elektronika. Stejným způsobem se měří akustická pohltivost materiálů, nebo průzvučnost v případě stavební akustiky.

Při stavebních úpravách různých prostor a jejich ozvučování je také třeba měřit dozvuk. To se týká například přednáškových sálů, koncertních síní, nebo různých společenských a kongresových místností. Velký a frekvenčně nevyrovnaný dozvuk má nepříznivý vliv na srozumitelnost mluveného slova. Toto měření se provádí tak, že se prostor napřed vybudí dostatečně hlasitým šumem. Po dosažení ustálené intenzity se zdroj šumu vypne a zaznamená se doznívání zvuku. Následná analýza ukazuje, jak rychle jsou jednotlivé frekvence zvuku v prostoru pohlcovány.

Další možností, jak využít šum, je při zkoumání šíření vibrací a hledání rezonančních módů nejrůznějších konstrukcí. To je významné například v automobilovém průmyslu, kdy je třeba zamezit šíření hluku od motoru do interiéru vozu.

Pásmově omezený růžový šum, nebo šum vhodně váhovaný (např. dle IEC 268-5) se používá jako zatěžovací signál. Tímto měřením lze stanovit maximální dlouhodobý příkon reproduktorů či reprosoustav.

Mimo elektroakustiku má šum řadu další využití. Od testování chybovosti přenosu v komunikačních systémech, přes měření ve vysokofrekvenční oblasti, až po rušení odposlechů a podobné speciální aplikace. To však již nespadá do oblasti zájmu této práce.

1.3 Zdroje

Jako zdroje analogového šumu se často používají přímo ty fyzikální procesy, které jsou jeho přirozeným původcem.

V případě tepelného šumu je snímáno a zesilováno šumové napětí rezistoru. Na tomto principu jsou i definované normály, kdy je snímaný materiál udržován na konstantní teplotě. V případě vysokoteplotního normálu jde o 673 K, u nízkoteplotního je to teplota vroucího dusíku, tedy 77 K. Tento princip je využitelný od subsonických frekvencí až do pásma mikrovln a tepelný šum lze považovat za bílý a gaussovský v širokém pásmu.

Další používané zdroje jsou výbojky. Zdrojem šumu jsou zde náhodné srážky elektronů ve výboji s atomy a ionty inertního plynu, kterým je výbojka plněna. Tento zdroj šumového proudu má velmi široké pásmo a používá se hlavně pro vysoké frekvence až do stovek GHz.

Výstřelový šum je důsledkem faktu, že elektrický proud není spojitý tok, ale je tvořen jednotlivými elektrony. Pro dosažení větší intenzity je vytvářen průrazem PN přechodu. Zde může posloužit zenerova dioda, nebo závěrně polarizovaný přechod emitor – báze tranzistoru. Jako kvalitní širokopásmový zdroj jsou však používány lavinové diody. Výstřelový šum způsobený jednotlivými elektrony je zde lavinovým jevem zesilován. Lavinové diody se dají využít jeho zdroj šumu od akustických frekvencí až po mikrovlny.

Dále je také uváděna jako zdroj výstřelového šumu vakuová dioda, pracující v nasyceném režimu. Kvůli poměrně velké kapacitě mezi anodou a katodou je ale použitelná jen do několika set MHz.

V nízkofrekvenční technice a elektroakustice je díky kvalitním A/D a D/A převodníkům většina zpracování prováděna číslicově. V okamžiku, kdy je zpracovávaný signál v podstatě posloupností čísel, vzniká těsná souvislost mezi šumem a řadou náhodných čísel. Tento vztah je užitečný obousměrně. Často je vzorkování přirozeného šumu používáno jako generátor náhodných čísel. Opačným postupem můžeme náhodná čísla využít k vytvoření šumu.

V řadě aplikací nevyžadujeme, aby byl šumový signál zcela neperiodický. Jindy nám k provedení měření stačí relativně krátký úsek. V takových případech pak ani není zapotřebí používat skutečný generátor, ale lze ho nahradit přehrávačem s nějakou formou záznamu.

Další variantou jsou generátory založené na pseudonáhodných číslech. Nejsou to zdroje skutečně náhodného šumu, ale pokud má výstupní signál potřebné rozdělení amplitud a spektrum, lze ho úspěšně použít. V praxi se vyskytují generátory založené na sekvencích maximální délky, tedy speciálních případech LFSR, ale i dalších algoritmech.

2 Hardware – Vývojová deska

Platformou pro vývoj šumového generátoru se stala deska Altera DE2. Jde o výukový kit s poměrně bohatou výbavou. Deska je osazena celou řadou obvodů pro komunikaci a multimédia, včetně několika typů pamětí. Pro funkci šumového generátoru je vybavena dostatečně.

V tomto projektu lze následující součásti považovat za nezbytné :

- Obvod FPGA
- Oscilátor 50 MHz
- Zvukový kodek
- Vstupní spínače a tlačítka

Další za potenciálně využitelné :

- Paměť SRAM 512 kB
- Paměť SDRAM 8 MB
- Paměť FLASH 4 MB
- Slot pro SD paměťovou kartu
- Dvouřádkový znakový LCD
- Sedmisegmentové displeje
- Sada LED

Deska obsahuje i řadu dalších obvodů a periferií, která s tématem přímo nesouvisí. I když si lze v některých případech představit jejich využití, bylo by něco takového značně nekoncepční. Je vhodné, aby generátor pro svoji funkci využíval jen minimum součástí.

Nevyužité součásti :

- RS232 transciever
- USB host controller
- 10/100 Ethernet controller
- Video DAC
- IrDA transciever
- Další nezmíněné

2.1 FPGA

Názvem Cyclone je označována řada nízkonákladových FPGA obvodů od Altera Corporation. Na vývojové desce je osazen EP2C35 z rodiny Cyclone II. Tento obvod, již poměrně starý, je vyráběn 90nm technologií a pochází z roku 2004. V tuto chvíli jsou aktuální obvody z rodiny Cyclone V, vyráběné 28nm technologií. Přes značný pokrok ve vývoji je celá řada Cyclone nadále podporována a doporučena pro nové návrhy.

Ve zmíněném EP2C35 jsou k dispozici tyto prostředky :

- 33216 logických elementů
- 105 RAM bloků M4K, celkem 483840 bitů
- 70 celočíselných násobiček 9 × 9 bitů
- 4 fázové závěsy

2.2 Zvukový kodek

Klíčové zařízení pro šumový generátor je DA převodník. V tomto případě využíváme integrovaný kodek WM8731. Jde o nízkopříkonový audiokodek určený zejména pro přenosná zařízení s bateriovým napájením.

Převodníky podporují vzorkovací frekvence 8 – 96 kHz a šířka slova vzorku může být 16, 20, 24 nebo 32 bitů. Kodek integruje i sluchátkový zesilovač s nastavitelným ziskem, nicméně na desce není sluchátkový výstup využit. Nachází se zde pouze výstup linkový.

Zvuková data se kodeku předávají sériově, přičemž komunikace může probíhat v několika režimech. Takt může udávat jak kodek, tak zařízení s ním komunikující. Režimy se pak liší hlavně v poloze přenášených bitů vzhledem k hranám signálu, jenž určuje, zda se vysílá vzorek pro levý nebo pravý kanál.

Veškerá nastavení jsou obsažena v deseti registrech. Zápis do těchto registrů probíhá přes rozhraní, které opět dovoluje dva režimy komunikace : Trojvodičový, kompatibilní s SPI a dvojvodičový, který je obdobou I^2C. Zapojení na desce DE2 ovšem dovoluje pouze druhou variantu.

Při použití kodeku pro přehrávání je většina nastavení v defaultních hodnotách, konfigurace je však nutná. Po zapnutí napájení je kodek v úsporném režimu a je nutné aktivovat části, které hodláme využívat.

3 Diskuse řešení v rámci vývojové desky

Ze všech možností tvorby šumového signálu jsou pro plně číslicový systém vhodné dvě. První varianta vytváří šum na základě sekvence pseudonáhodných čísel, druhá využívá předpřipraveného záznamu uloženého v paměti. Další problém představuje požadavek na omezení pásma výstupního šumu. Nabízí se číslicový filtr, ovšem generátor využívající přehrávání z paměti může obsahovat celou sadu již předem filtrovaných záznamů.

Pro následující posouzení nároků na prostředky uvažujme tyto modelové situace :

- Vzorkovací frekvence výstupního signálu
  Mimimálně 48 kHz
  Typicky 192 kHz
- Počet bitů na vzorek
  Minimálně 16
  Typicky 24
- Počet nezávislých výstupních kanálů
  Minimálně 1
  Typicky 2
- Počet variant šumu s různým pásmovým omezením
  Minimálně 41
  Typicky 45

3.1 Číslicový filtr

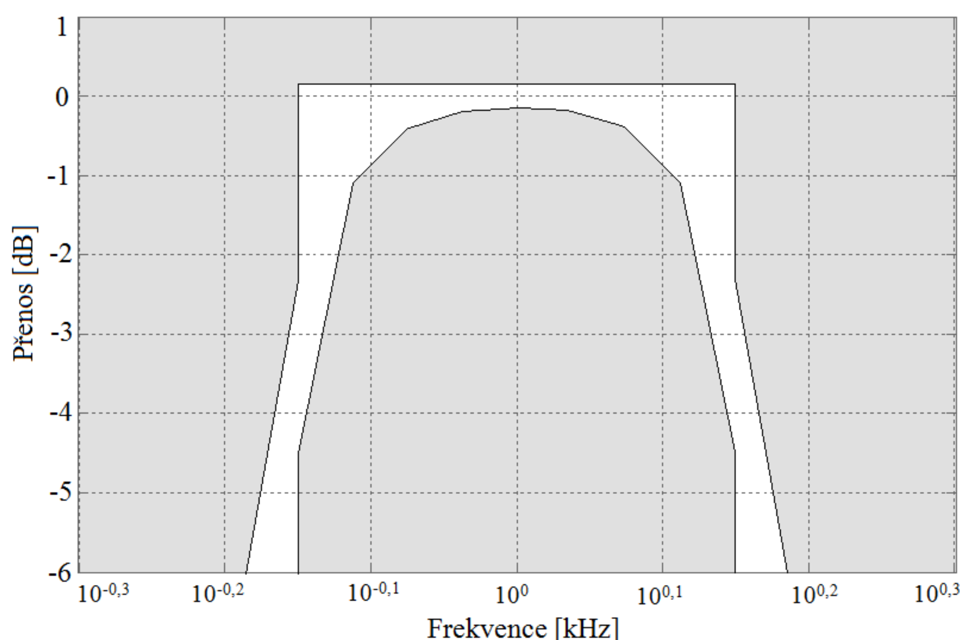
Návrh číslicového filtru musí stanovit druh a strukturu filtru. Kritéria k posouzení jsou rychlost zpracování vzorku, potřebné aritmetické prostředky a paměť nutná pro uložení sady koeficientů. To vše s přihlédnutím k obecným vlastnostem číslicových filtrů.

Na vývojové desce je oscilátor s frekvencí 50 MHz, při požadované vzorkovací frekvenci je na zpracování jednoho vzorku k dispozici 260 – 1041 taktů. Pokud bude celý systém koncipován jako pipeline, může tuto dobu využít ke zpracování signálu každá její součást.

Číslicový filtr je většinou velmi pravidelná struktura, pro kterou lze snadno docílit vhodného kompromisu mezi dobou zpracování a nutnými prostředky. Jedním extrémem je sekvenční vykonání všech aritmetických úkonů jednou násobičkou a sčítačkou. Druhý extrém pak představuje plně paralelní zapojení.

Pásmové propusti oktávových a třetinoktávových filtrů definuje norma ANSI S1.11. Žádoucí průběh přenosové charakteristiky splňuje Butterworthova aproximace šestého řádu. Nabízí se tedy filtry IIR, které lze navrhovat jako ekvivalent analogových filtrů. Díky tolerančnímu pásmu je ale v principu možné, aby byl vyhovující průběh dosažen i s filtry FIR. Zvlnění propustného pásma musí být maximálně ±0.15 dB, ±0.3 dB nebo ±0.5 dB dle třídy filtru. Útlum mimo propustné pásmo by měl být lepší než 60 – 75 dB, opět dle třídy filtru.

Obr. 2 – Toleranční schéma oktávového filtru třídy 0



Výhoda FIR filtru je v jeho bezpodmínečné stabilitě, nevýhoda ve značně vysokém řádu, zejména při vysokých nárocích na útlum mimo propustné pásmo. Experimenty s návrhem filtru pomocí filter design toolboxu v systému Matlab ukazují, že pro třetinoktávovou pásmovou propust, by byl nutný filtr řádu přibližně 600 – 900. Takový by mohl splnit i nejtvrdší uvedené podmínky. Výraznějšího zjednodušení by bylo možné dosáhnout jen za cenu uvolnění parametrů a návrhu filtru jen podle kritérií třídy 1 nebo dokonce třídy 2.

Dalším aspektem je přesnost koeficientů. Aby nedošlo k porušení tolerančního schématu, je nutné, aby koeficienty měly nejméně 13 bitů, prakticky by šlo spíše o 16 bitů. Lze tedy očekávat potřebu 320 – 648 kb paměti jen pro koeficienty filtrů. Velmi pravděpodobně by bylo nutné použít paměť mimo FPGA. Jelikož filtrace může probíhat 260 taktů, rozsah logiky by byl zřejmě stále únosný. Stavové registry lze umístit do RAM a potřebných několik set násobení může vykonat 6 – 8 násobiček. Takové řešení filtru je však značně komplikované.

IIR filtr splňující podmínky normy je naproti tomu velmi jednoduchý. Lze ho realizovat například jako kaskádu tří sekcí druhého řádu. Jelikož některé koeficienty mohou být triviální, je možné filtraci provést na pouhých 9 násobení a 9 odečítání. Jiné struktury jsou možné, ale v tomto jednoduchém případě se příliš neliší v počtu nutných aritmetických operací.

Nevýhoda je riziko nestability takového filtru. K nestabilitě přispívá omezená přesnost koeficientů, zaokrouhlování ve výpočtech a to vše zejména u filtrů, jejichž kritické frekvence jsou mnohem nižší, než je vzorkovací kmitočet.

Pro malý počet artimetických operací je v případě IIR filtru možné uvažovat i o aritmetice s plovoucí řádovou čárkou. Problémy s omezenou přesností koeficientů odpadají, neboť v IEEE 753 double precision má mantisa 53 bitů. Nároky na paměť pro koeficienty by i v takovém případě byly pouhých 26 kb. S ohledem na maximální čas průchodu vzorku filtrem lze filtr realizovat s jednou floating point násobičkou a jednou odečítačkou.

3.2 Generátor pseudonáhodných čísel

V současné době existuje celá řada algoritmů pro tvorbu pseudonáhodných sekvencí. Velmi se liší kvalitou výstupu i nároky na paměť a výpočetní čas. Pro využití v roli šumového generátoru požadujeme kvalitní statistické vlastnosti. Jiná kritéria nemusí být už tak důležitá. Například nemožnost určit stav a tedy předpovědět budoucí čísla z dosavadních výsledků. Taková vlastnost je zásadní například pro kryptografii, ale z hlediska náhodnosti a spektra výsledného šumu je nepodstatná.

V nenáročných úlohách lze nejsnadněji implementovat LFSR – Posuvné registry s kombinační zpětnou vazbou, ty však nedávají příliš kvalitní výsledky. Ani LCG – Lineární kongruentní generátory nejsou samy o sobě ideální, byť jsou v počítačovém software masivně rozšířené.

Další často používané algoritmy jsou :

- Multiplikativní LCG
- Násobení s přenosem (MWC)
- Komplementární MWC
- XOR-Shift

Kromě těchto existují další, specializované například pro použití v Monte Carlo simulacích. Některé algoritmy samy o sobě mohou mít poměrně krátkou periodu, nebo

nedokáží projít všemi statistickými testy. Vhodnou metodou jak vytvořit generátor s dostatečně dlouhou periodou i kvalitou výsledků je kombinace více algoritmů. Vlastnosti výše uvedených uvádí sedmá kapitola [1] a také doporučuje algoritmus, který je kombinací LGC, MWC a dvou XOR-Shiftů. Jeho perioda je dostatečně dlouhá (3.138×10^{57}), výstup projde sadou Diehard testů a po výpočetní stránce jde o dvě násobení, tři sčítání a několik bitových posuvů s exkluzivními součty. Výsledkem je 64 bitové celé číslo, z nějž lze použít jakékoliv bity.

Tento algoritmus je v literatuře publikován v jazyce C++. Zde ho uvádím v, dle mého názoru, přehlednějším pseudokódu.

```
int64() {
  u = u * 2862933555777941757 + 7046029254386353087
  v = v xor (v >> 17)
  v = v xor (v << 31)
  v = v xor (v >> 8)
  w = 4294957665 * (w and 0xFFFFFFFF) + (w >> 32)
  x = u xor (u << 21)
  x = x xor (x >> 35)
  x = x xor (x << 4)
  return (x + v) xor w
}
```

Generátor je nutné inicializovat následujícím postupem. *Seed* může mít libovolnou hodnotu s výjimkou 4101842887655102017 proto, aby úvodní hodnota *u* nebyla nulová.

```
init() {
  v = 4101842887655102017
  w = 1
  u = seed xor v
  int64
  v = u
  int64
  w = v
  int64
}
```

Každé další volání *int64* po skončení inicializace vrací další prvek pseudonáhodné sekvence. Zmíněná dvě násobení jsou jedno 64 bitové a jedno 32 bitové. Je možné je převést na čtyři násobení 32 × 32 bitů a sčítání. Dle potřebné rychlosti a množství volných prostředků lze algoritmus realizovat s 8 – 32 devítibitovými násobičkami.

3.3 Varianty kompletního systému

Za tři hlavní koncepce celého systému můžeme považovat tyto :

A) Generátor pseudonáhodných čísel a digitální filtr

Toto řešení je nejméně náročné na paměť, protože ta je potřeba jen pro koeficienty filtru. V závislosti na řádu a struktuře filtru lze odhadnout, že půjde řádově o desítky kilobitů při použití IIR. Případně o řád více při použití FIR. Takové množství dat lze umístit do paměťových bloků v FPGA a externí paměť není nezbytně nutná.

Vyšší nároky, dle použitého algoritmu, jsou na aritmeticko-logickou část. Ty lze ovšem obecně těžko stanovit, neboť různé postupy výpočtu pseudonáhodných čísel se rozsahem velmi liší. Uvedený konkrétní algoritmus bude vyžadovat zejména násobičky.

Výhodou je, že výsledný šum je v praxi neperiodický. Číslicové řešení je sice ryze deterministické, takže generované sekvence čísel periodu mají, nicméně ta je při vhodném výběru algoritmu nevyužitelně dlouhá.

Další výhodná vlastnost je, že samotný generátor čísel lze beze změny použít pro libovolnou vzorkovací frekvenci, pokud stíhá při daném taktu poskytovat výsledky. Snadno lze také vytvořit více nezávislých kanálů.

B) Širokopásmový záznam a digitální filtr

Číslicový filtr je zde stejný, jako v předcházejícím případě. Zdrojem signálu je paměť s uloženým úsekem šumového signálu. Zde je již nezbytné použít další paměť, například flash, protože záznam představuje 768 kb – 4.6 Mb na jednu jeho sekundu.

Toto řešení lze stále použít i pro více vzorkovacích frekvencí, ovšem je nutné přihlédnout k délce periody. Záznam může obsahovat vzorek velmi kvalitního šumu.

C) Sada záznamů pro všechna požadovaná pásma

Varianta s předem filtrovanými záznamy je nejúspornější z pohledu aritmetických a logických prostředků FPGA. Za předpokladu, že je k dispozici dostatečně velká a jednoduše přístupná paměť. Externí paměť musí pojmout 35 – 210 Mb na každou sekundu délky záznamů. Toho lze u DE2 dosáhnout jen s SD kartou a je pravděpodobné, že řadič pro komunikaci s kartou eliminuje výhodu malého rozsahu.

Záznamy jsou připraveny pro konkrétní vzorkovací frekvenci a jejich použití pro jinou je velmi omezené.

4 Vybrané řešení a funkce

Jako celkový koncept jsem vybral variantu s generátorem pseudonáhodných čísel a číslicovým filtrem. Filtr bude IIR s aritmetikou v plovoucí řádové čárce. Takové řešení by mělo být dosti flexibilní a zároveň se veškerá funkcionalita vejde do FPGA bez nutnosti používat externí paměť.

Kodek osazený na vývojové desce umožňuje vzorkovací frekvenci maximálně 96 kHz, ta bude použita, nicméně považuji za výhodné, aby projekt byl schopen bez větších změn dodávat vzorky i s frekvencí 192 kHz.

Generátor bude dvoukanálový, přičemž oba kanály budou nezávislé s vlastním nastavením zisku a filtrace. Filtr bude disponovat kompletní sadou koeficientů pro obvyklé oktávové a třetinoktávové pásmové propusti.

Celý postup tvorby šumu bude probíhat tak, že se budou generovat náhodná čísla s rovnoměrným rozdělením. Statistické rozdělení se upraví na normální. Tyto hodnoty se převedou na formát IEEE 754 double precision a projdou filtrem. Následně se převedou na celá čísla v kódu s posunutou nulou pro potřeby komunikace s kodekem a zapíšou do fronty vzorků. Všechny tyto stupně zpracování budou zřetězeny do pipeline.

Další nezbytné prvky budou entity pro konfiguraci kodeku pomocí I^2C. Situace je zjednodušená tím, že postačuje schopnost na sběrnici zapisovat. Dále je zapotřebí kodeku zajistit hodinový signál, ať již vhodným dělením dostupných oscilátorů, nebo použitím PLL. Nakonec je nutné alespoň základní ovládání celého systému a signlizace nejdůležitějších parametrů.

Nejvíce limitujícím faktorem v tomto projektu jsou celočíslné násobičky. Při použití rychlejší varianty výpočtu v algoritmu PRNG zabere tento 32 násobiček. Zapojení pro násobení double precision floatů vyžaduje 18 násobiček a je ve filtru pouze jednou. Z toho plyne, že projekt může obsahovat dva nezávisle filtrované kanály při využití 68 ze 70 násobiček, za předpokladu, že PRNG bude dost rychlý a další násobičky již nebudou zpotřebí.

Proces filtrace, včetně nezávislého nastavení zisku vyžaduje 10 násobení a 9 odečítání. Násobení trvá 5 taktů a odečítání 7 taktů, celá aritmetika tedy zabere 113 taktů. Další čas bude potřeba na zápisy do registrů, nicméně výsledná doba bude jistě menší než 260 taktů. Filtr tedy může signál s vzorkovací frekvencí 192 kHz bez problémů zpracovávat.

Popsané řešení podle doposud uvedených odhadů považuji za realizovatelné.

5 Implementace

Funkční celky, které tvoří vlastní generátor šumových vzorků, jsou zapojeny do pipeline. Jelikož způsob signalizace a předávání dat je společný všem, je popsán již zde na úvod. Toto řešení jednak přináší potřebnou rychlost zpracování. Dále pak usnadňuje vývoj jednotlivých bloků, které jsou značně nezávislé.

Jednotlivé funkční bloky jsou propojeny několika signály. V první řadě jde o vlastní předávaná data od předchůdce k následníkovi. Dále je to signál *ready* oznamující následníkovi, že na datovém výstupu je nová platná hodnota. Nakonec signálem *accept* následník potvrzuje předchůdci, že nová data převzal ke zpracování.

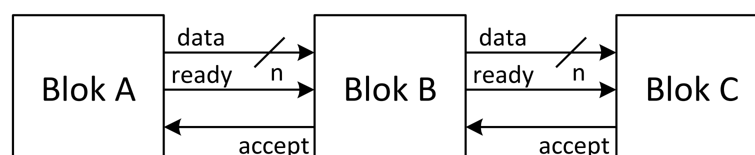
Každý blok, s výjimkou prvního a posledního tedy disponuje touto sadou portů :

```
entity example is
  port (
    clk         : in  std_logic;
    rst         : in  std_logic;
    in_data     : in  std_logic_vector(N downto 0);
    in_ready    : in  std_logic;
    in_accept   : out std_logic;
    out_data    : out std_logic_vector(N downto 0);
    out_ready   : out std_logic;
    out_accept  : in  std_logic;
end entity example;
```

Signál *clk* je hodinový a *rst* je reset aktivní v jedničce. Vstupy *in_data* a *in_ready* jsou data a jejich oznámení od předchůdce, *in_accept* slouží k jejich potvrzení. Obdobně výstupy *out_data* a *out_ready* slouží k předání výsledků na následovníkovi, *out_accept* je pak vstup potvrzující převzetí dat tímto následovníkem.

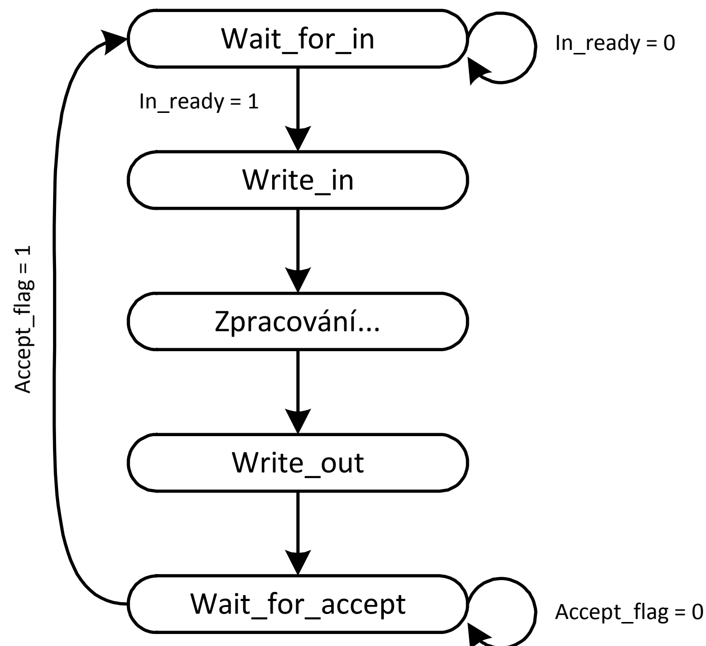
Bloky obecně pracují tak, že čekají na signál *ready* od předchůdce a jakmile ho obdrží, zapíší si data do vstupního registru. Bezprostředně poté potvrdí signálem *accept* převzetí. Jakmile předchůdce obdrží potvrzení, deaktivuje signál *ready* a začne pracovat na nových datech. Blok, který data převzal je po vlastním zpracování zapíše do výstupního registru. Zároveň nastaví signál *ready* a čeká, dokud další blok v pořadí tento výsledek nepřevezme.

Obr. 3 – Schéma zřetězení bloků do pipeline



Každý blok má vnitřní registr *accept_flag*, jehož hodnota udává, zda byla naposledy zapsaná hodnota ve výstupním registru již odebrána následníkem. Blok si svůj registr vynuluje v průběhu zpracování dat a nastaví pokud je aktivní signál *accept*. Registr *accept_flag* je pak použit jako součást přechodové funkce stavového automatu.

Obr. 4 – Obecný stavový diagram automatů v pipeline



Obecnou funkci bloků v pipeline popisuje diagram na obrázku 4. V úvodním stavu se čeká na oznámení nových dat od předchůdce. Jakmile jsou oznámena, zapisují se ve stavu *Write_in* do vstupního registru. Následuje potvrzení signálem *in_accept* a vlastní zpracování. Po dokončení zpracování dat jsou zapsána do výstupního registru *out_data* a nejpozději v tuto chvíli je nulován *accept_flag*. Poslední stav nastaví oznámení *out_ready* a v tomto stavu automat setrvává, dokud není *accept_flag* opět nastaven potvrzením *out_accept*. Jakmile k tomu dojde, proces se opakuje.

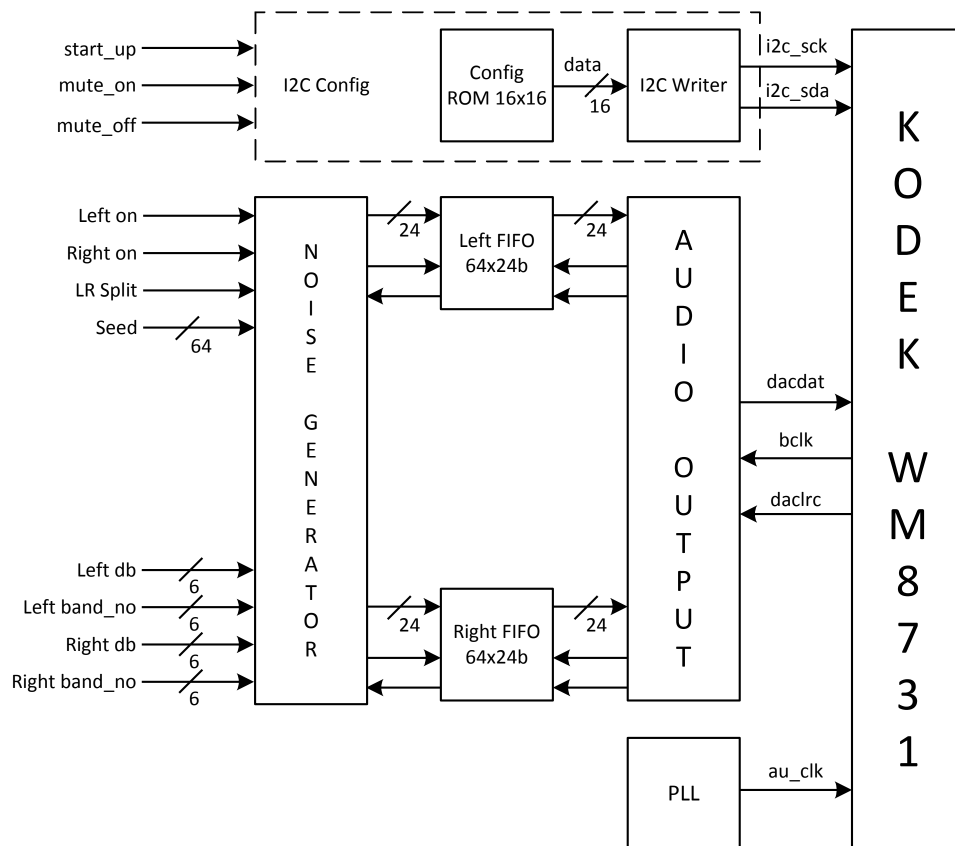
Výjimka je *rng_uniform*, který je na začátku pipeline a nemusí tedy čekat na předchůdce. Při tomto postupu je tedy tok dat skrze pipeline řízen odebíráním výsledků z jejího konce. Všechny bloky se snaží zpracovat vstupy na výsledky a čekat na jejich odebrání. Pokud se po zapnutí systému nespustí kodek, který by vyprazdňoval fronty, pipeline se postupně naplní hodnotami. Poté je začne *fifo_feeder* odebírat a zapisovat do fronty. V okamžiku zaplnění fronty se celý proces zastaví a všechny bloky budou s připravenými výsledky čekat ve svých stavech *wait_for_accept*.

5.1 Celkové schéma

Na obrázku 5 je blokové schéma zachycující celkové řešení projektu. Pro přehlednost zde není zakreslen hodinový signál ani reset. Dále tu také nejsou některé komponenty, které se přímo netýkají funkce šumového generátoru a vztahují se k ovládání a signalizaci.

Celý systém se skládá z mnoha komponent. Pro snazší manipulaci a přehlednost je samotný generátor šumových vzorků zapouzdřen v entitě *noise_generator*. Tato entita představuje pouze strukturální propojení obsažených částí a bude popsána dále.

Obr. 5 – Celkové schéma šumového generátoru



Význam všech signálů je popsán spolu s entitami, na které jsou přímo napojeny. N*oise_generator* je připojen na dvě fronty *fifo_64w24b*. Tyto fronty jsou odvozeny z megafunkce *dcfifo* a předávají vzorky mezi dvěma hodinovými doménami. Každá fronta je fyzicky realizována v jednom M4K bloku paměti. Čtení z front a následný sériový přenos dat je taktován zvukovým kodekem, takže není obecně synchronní se zbytkem projektu.

Zbývající blok *i2c_config*, zodpovědný za nastavení kodeku přes sběrnici I^2C, je blíže popsán v kapitole 5.11.

5.2 Blok šumového generátoru

Entita *noise_generator* obsahuje celou pipeline vytvářející a zpracovávající šumové vzorky. Dále zahrnuje také paměť koeficientů a převodní funkce pro nastavení zisku.

```
entity noise_generator is
  port (
    running       : out std_logic_vector(5 downto 0);
    clk           : in  std_logic;
    rst           : in  std_logic;
    lr_dependency : in  std_logic;
    rng_seed      : in  std_logic_vector(63 downto 0;
    left_reset    : in  std_logic;
    left_band_no  : in  std_logic_vector(5 downto 0);
    left_gain     : in  std_logic_vector(5 downto 0);
    left_active   : in  std_logic;
    fifo_l_full   : in  std_logic;
    fifo_l_data   : out std_logic_vector(23 downto 0);
    fifo_l_wr_req : out std_logic;
    right_reset   : in  std_logic;
    right_band_no : in  std_logic_vector(5 downto 0);
    right_gain    : in  std_logic_vector(5 downto 0);
    right_active  : in  std_logic;
    fifo_r_full   : in  std_logic;
    fifo_r_data   : out std_logic_vector(23 downto 0);
    fifo_r_wr_req : out std_logic);
end entity noise_generator;
```

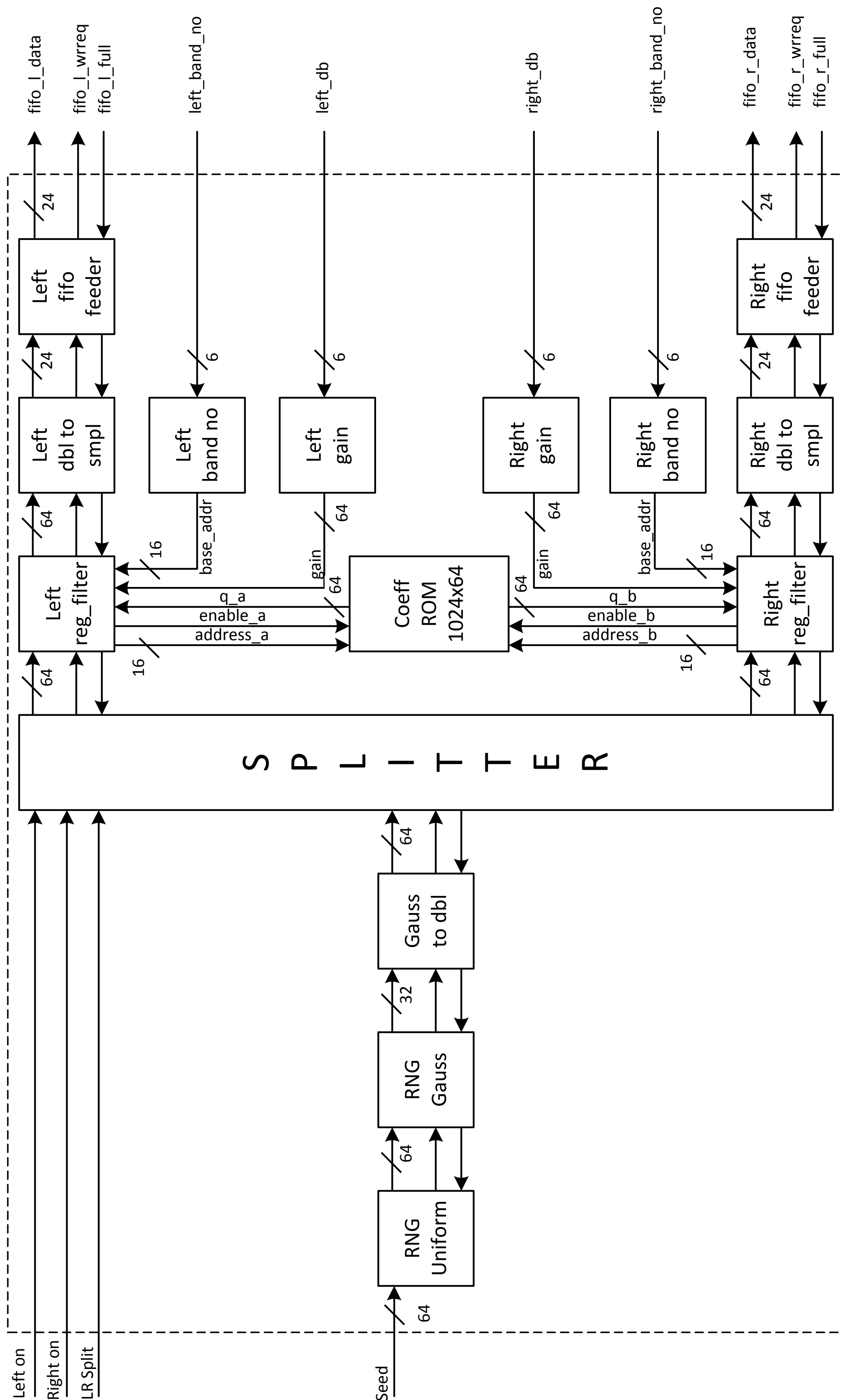
Signál *running* je zde pouze pro účely testování a signalizace, jde o výstup složený ze signálů *out_accept* z celé pipeline od generátoru po konec levého kanálu. Při vhodném vydělení frekvence umožňuje vizuálně sledovat aktivitu v pipeline.

Rng_seed zavádí inicializační hodnotu do generátoru pseudonáhodných čísel. Signály *lr_dependency, left_active* a *right_active* nastavují režim splitteru a jejich význam je objasněn v kapitole 5.6.

Zvláštní signály *left_reset* a *right_reset* jsou zavedeny do filtrů namísto *rst*. Umožňují resetovat filtry i když není všeobecný *rst* aktivní. Při změně sady koeficientů, tedy se změnou hodnoty *left_band_no / right_band_no*, jsou aktivovány aby došlo k nulování stavových registrů ve filtrech.

K nastavení zisku a výběru filtru pro daný kanál slouží *left_gain, left_band_no* a jejich protějšky pro pravý kanál. Obě hodnoty se dále zpracovávají v pomocných entitách popsaných v kapitole 5.12.

Obr. 6 – Schéma architektury noise_generator



5.3 Generátor pseudonáhodných čísel

Obvodové řešení pro algoritmus uvedený v kapitole 3.2 je složeno ze dvou entit. Vnitřní *rng_int64* je kombinační funkce, která odpovídá rutině `int64` ve zmíněném algoritmu. Vnější *rng_uniform* pak tuto funkci využívá k výpočtu a doplňuje inicializaci a komunikaci s následující částí zpracovávací pipeline.

```
entity rng_int64 is
  port (
    old_u, old_v, old_w : in  std_logic_vector(63 downto 0);
    new_u, new_v, new_w : out std_logic_vector(63 downto 0);
    y : out std_logic_vector(63 downto 0));
end entity rng_int64;
```

Vstupními hodnotami jsou aktuální hodnoty proměnných u, v a w, což jsou 64b celá čísla. Výstupy funkce jsou nové hodnoty pro tyto proměnné a člen pseudonáhodné posloupnosti y, taktéž 64b celé číslo.

Sama entita *rng_int64* využívá několika komponent :

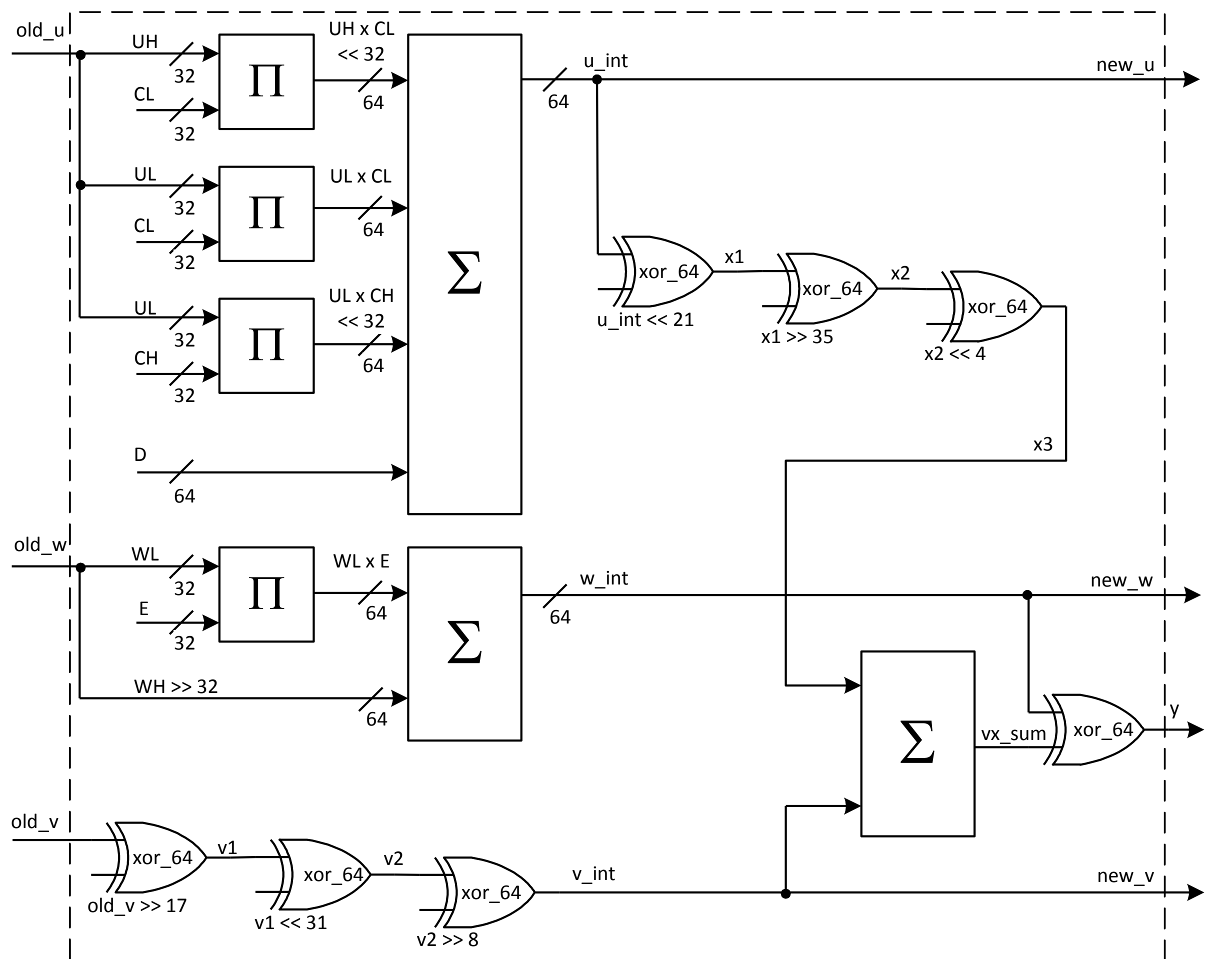
- *xor_64b* – Exkluzivní součet pro 64b široké operandy. Používá se pro kroky odpovídající dvěma xorshiftům ve výpočtu.
- *add_2u64* – Sčítačka odvozená z knihovní funkce *lpm_add_sub*. Sčítá dva 64b operandy bez znaménka, přenos se ignoruje.
- *add_4u64* – Paralelní sčítačka pro čtyři 64b operandy bez znaménka, je odvozena od *parallel_add* a přenos opět není důležitý. Je využita pro součet součinů při násobení po částech.
- *mul_2u32* – Celočíslná násobička 32 × 32 bitů odvozená od funkce *lpm_mult.*

LCG, který je součástí *rng_int64,* vyžaduje násobení 64 × 64 bitů. Toto násobení lze realizovat se spotřebou 32 devítibitových násobiček funkcí odvozenou od *lpm_mult*. Jelikož potřebujeme pouze spodní polovinu výsledku, můžeme docílit úspory když převedeme zmíněné násobení na součet součinů 32 × 32 bitů s posuvy.

$$A_HA_L \times B_HB_L = (A_L \times B_L) + (A_H \times B_L) << 32 + (A_L \times B_H) << 32 + (A_H \times B_H) << 64$$

Z tohoto tvaru je patrné, že poslední člen spodní polovinu výsledku nijak neovlivní a není tedy třeba ho počítat. Tento výpočet si vyžádá pouze 24 devítibitových násobiček.

Obr. 7 – Schéma architektury rng_int64



První tři násobičky ve chématu a paralelní sčítačka řeší výpočet :

```
u = u * 2862933555777941757 + 7046029254386353087
```

kde UH je horní a UL spodní polovina signálu *old_u*, stejně tak CH a CL jsou poloviny násobící konstanty. Signál D je pak druhá, přičítaná konstanta ve výpočtu. Paralelní sčítačka slouží zároveň k sečtení dílčích součinů i výpočetní konstanty.

Poslední násobička s druhou sčítačkou slouží k výpočtu :

```
w = 4294957665 * (w and 0xFFFFFFFF) + (w >> 32)
```

kde konstanta je označena jako signál E a druhý činitel je spodní polovina signálu *old_w*. V kódu je tato spodní polovina získána vymaskováním pomocí bitového součinu, ovšem v obvodovém řešení je možné ji použít přímo. Zbytek výpočtu tvoří přičtení horní poloviny *old_w* posunuté na pozici spodní poloviny.

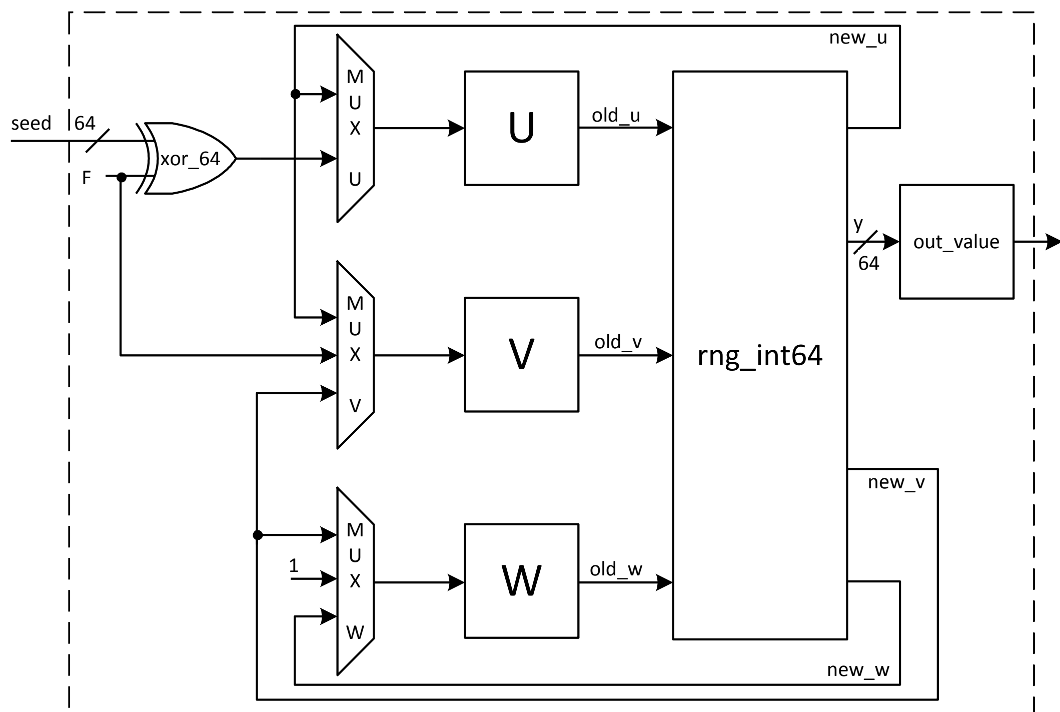
Signál *new_v* je vytvořen kaskádou posuvů a exkluzivních součtů. Jde o první xorshift v kódu funkce a další počítá pomocný signál *x* z hodnoty *new_u*. Poslední sčítačka a *xor_64b* z výsledků jednotlivých dílčích algoritmů počítají žádané náhodné číslo *y*.

Dosud popisovaný ekvivalent funkce `int64` je dále využit jako komponenta v entitě *rng_uniform*. Ta doplňuje funkcionalitu na plnohodnotný generátor čísel s rovnoměrným rozdělením.

```
entity rng_uniform is
  port (
    clk        : in  std_logic;
    rst        : in  std_logic;
    seed       : in  std_logic_vector(63 downto 0);
    out_accept : in  std_logic;
    out_value  : out std_logic_vector(63 downto 0);
    out_ready  : out std_logic);
end entity rng_uniform;
```

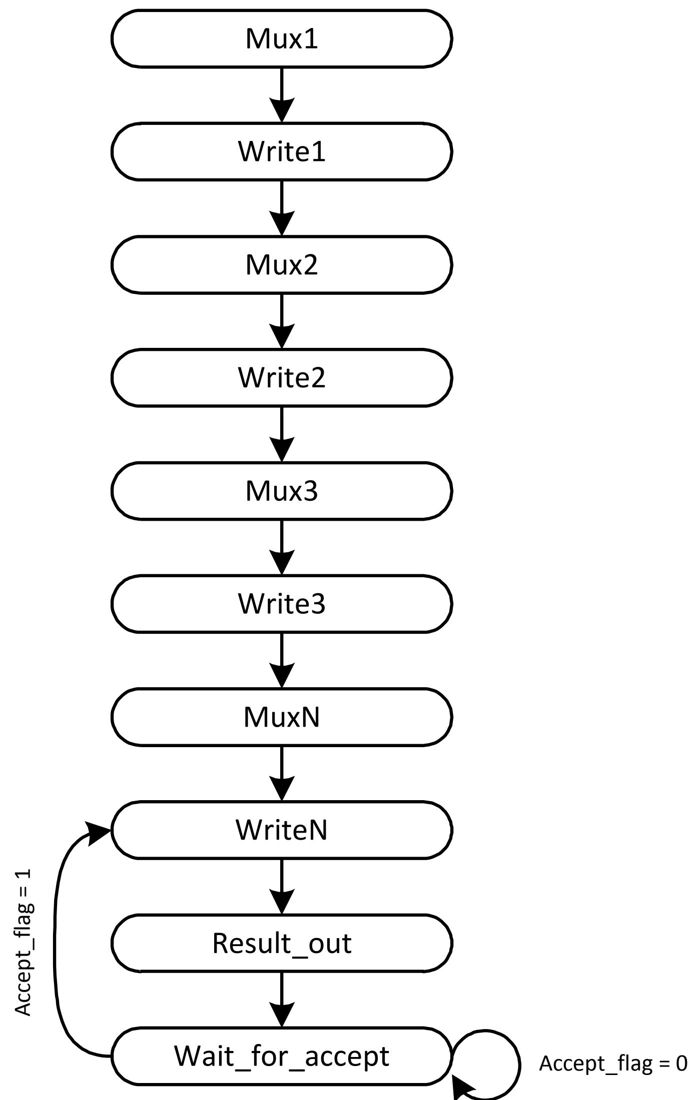
Vstupní signál *seed* je 64b konstanta, která slouží k inicializaci generátoru po uvolnění resetu. Výstupem je *out_value*, pseudonáhodné 64b celé číslo. Zbylé signály jsou hodiny, reset a *out_accept* spolu s *out_ready* slouží k předávání výsledků k dalšímu zpracování.

Obr. 8 – Zjednodušené schéma architektury rng_uniform



Nadstavba nad *rng_int64* je poměrně jednoduchá a zachycuje ji schéma na obrázku 9. Při běžné funkci jsou multiplexory nastaveny tak, že jsou do registrů *u, v* a *w* zapisovány nové hodnoty vypočtené v rámci *rng_int64*. Ostatní volby slouží pouze v procesu inicializace. Nastavení multiplexorů a signály „clock enable“ pro aktivaci zápisu do registrů ovládá stavový automat.

Obr. 9 – Stavový diagram automatu rng_uniform



Aktivní reset udržuje automat ve stavu *Mux1*, po jeho odeznění projde automat bezpodmínečnou sekvencí stavů až k *MuxN*. Během těchto kroků se generátor uvede do stavu, který odpovídá vykonání rutiny `init` z kapitoly 3.2

V prvním vstupu do stavu *WriteN* se do registrů *u,v* a *w* zapisují výchozí hodnoty pro první platný výsledek. Ten je v následujícím kroku zapsán na výstup *out_value* a oznámen signálem *out_ready*. Další cyklus zápisů pokračuje potvrzením převzetí výsledku následujícím článkem pipeline.

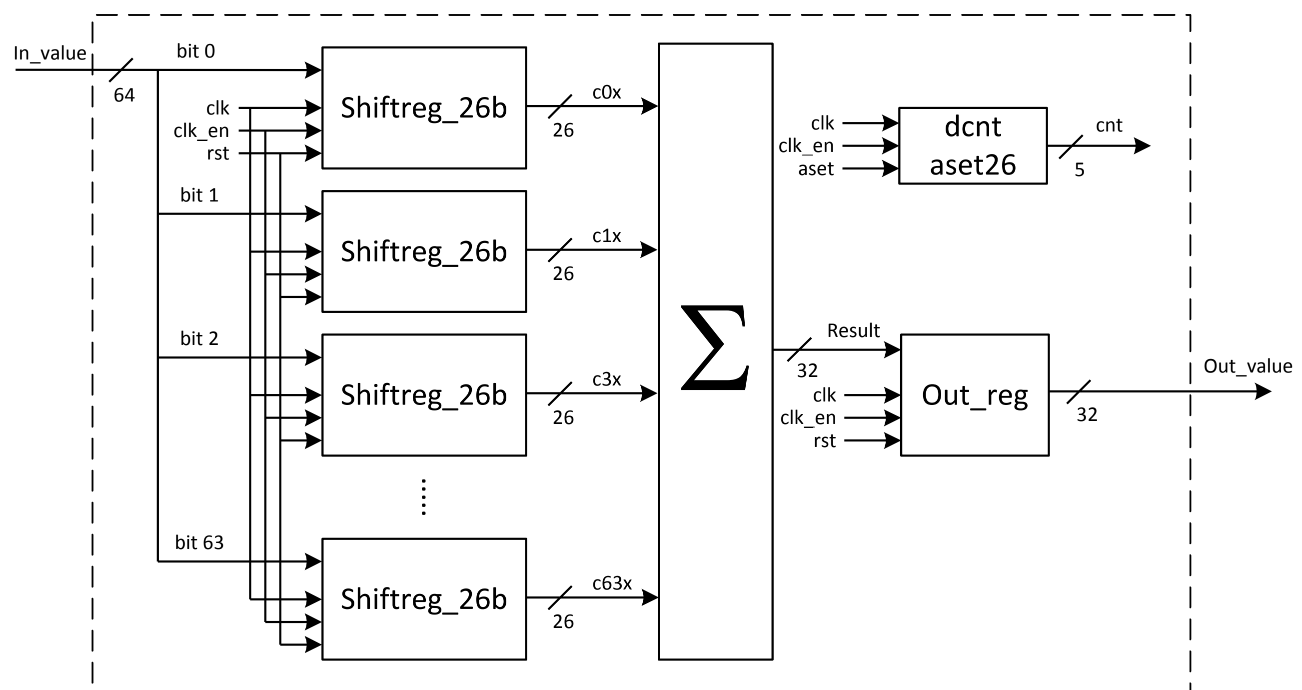
5.4 Převod na normální rozdělení

Entita *rng_gauss* využívá sekvenci vstupních náhodných čísel s rovnoměnrým rozložením pravděpodobnosti k tvorbě sekvence výstupních čísel s normálním rozložením pravděpodobnosti.

```
entity rng_gauss is
  port (
    clk        : in  std_logic;
    rst        : in  std_logic;
    in_value   : in  std_logic_vector(63 downto 0);
    in_ready   : in  std_logic;
    in_accept  : out std_logic;
    out_accept : in  std_logic;
    out_value  : out std_logic_vector(31 downto 0);
    out_ready  : out std_logic);
end entity rng_gauss;
```

Kromě resetu, hodinového signálu a řízení toku dat v pipeline má tato entita pouze jediný vstupní a výstupní signál. Vstup *in_value* je 64b celé číslo bez znaménka a obsahuje náhodnou hodnotu z celého rozsahu. Výstup *out_value* je 32b celé číslo bez znaménka, jehož hodnota může nabývat maximálně 0xFFFFFFC0.

Obr. 10 – Zjednodušené schéma architektury rng_gauss



Každý bit vstupního čísla se považuje za samostatný náhodný generátor, hodnoty se nasouvají do sady 26b posuvných registrů *shiftreg_26b*. Po zaplnění registrů se zapíše součet jejich hodnot do výstupního registru jako výsledek. Sčítání provádí paralelní sčítačka *add_64u26* odvozená od *parallel_add*. Za předpokladu, že každý z 64 sčítanců představuje 26b číslo s rovnoměrným rozdělením pravděpodobnosti, pak má střední hodnotu a rozptyl :

$$\mu = \frac{2^{26}-1}{2} = 33554431{,}5 \qquad (5.4.1)$$

$$\sigma^2 = \frac{(2^{26}-1)^2}{12} = 3{,}753 \times 10^{14} \qquad (5.4.2)$$

Pro součet náhodných veličin platí :

$$\mu(X+Y) = \mu(X) + \mu(Y) \qquad (5.4.3)$$

$$\sigma^2(X+Y) = \sigma^2(X) + \sigma^2(Y) + 2cov(X,Y) \qquad (5.4.4)$$

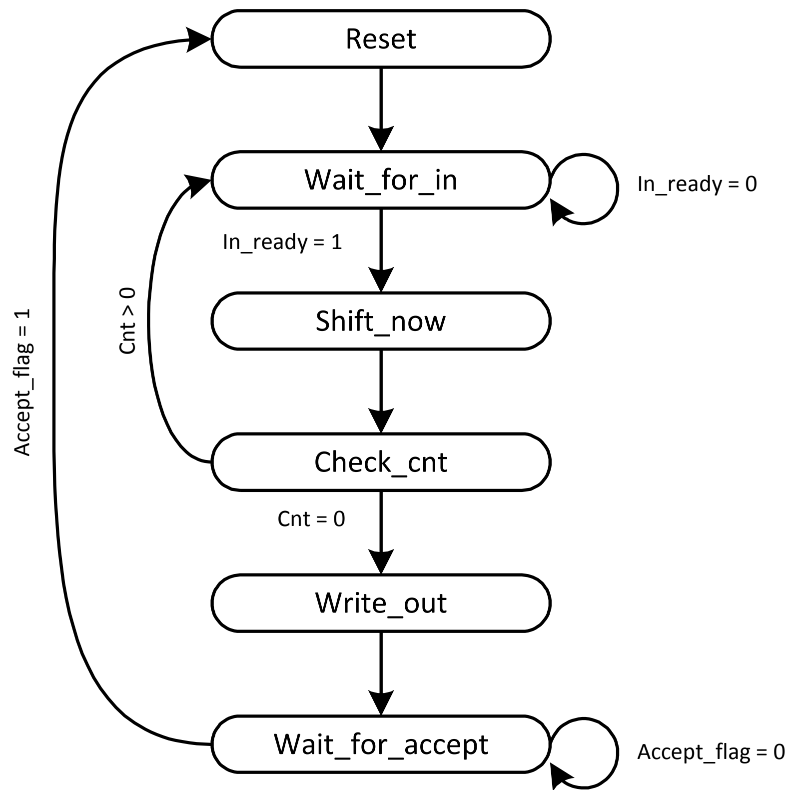
kde kovariační člen je nulový, pokud jsou X a Y nezávislé veličiny. Pak můžeme střední hodnotu i rozptyl součtu vypočítat prostým vynásobením počtem sčítanců :

$$\mu = \frac{2^{26}-1}{2} \times 64 = 2147483616 \qquad (5.4.5)$$

$$\sigma^2 = \frac{(2^{26}-1)^2}{12} \times 64 = 2{,}402 \times 10^{16} \qquad (5.4.6)$$

Standardní odchylka je pak $\sigma = 1{,}55 \times 10^8$ a výstupní čísla mají rozdělení velmi blízké normálnímu v rozsahu $\mu \pm 13{,}9\sigma$. Tento rozsah je velmi široký, pravděpodobnost výskytu krajních hodnot je pouze $2{,}4 \times 10^{-501}$. U normálního rozdělení je pravděpodobnost hodnoty mimo interval $\pm 6\sigma$ přibližně 2×10^{-9}, to by při 192 kHz znamenalo průměrně jeden vzorek za 43 minut. Pro praktické využití dynamického rozsahu DA převodníku je tedy vhodné dále při zpracování zařadit zesílení, které zajistí kompromis mezi průměrnou úrovní signálu a četností výskytu saturovaných vzorků. Při hodnotě zisku 2,4 by se měl mimo interval $\pm 5{,}9\sigma$ vyskytnout pouze jeden vzorek za 23 min.

Obr. 11 – Stavový diagram automatu rng_gauss



Aktivní reset udržuje automat ve stejnojmenném stavu, zároveň se nastavuje dekrementální čítač na hodnotu 26. Po uvolnění resetu se přejde do stavu *wait_for_in*, kde se čeká na platnou vstupní hodnotu, oznamovanou signálem *in_ready*. Jednotlivé bity *in_value* jsou přivedeny na vstupy posuvných registrů a bezprostředně po oznámení nové platné hodnoty jsou do registrů zapsány. Zároveň je snížena hodnota čítače. Následně je potvrzeno převzetí vstupní hodnoty a kontroluje se, zda již čítač dospěl k nule a registry jsou plné. Pokud ne, čeká se na další vstup, pokud ano, je paralelní součet všech registrů zapsán na výstup. Oznámí se nová platná hodnota výstupu a čeká se na potvrzení jejího převzetí dalším článkem pipeline. Po potvrzení se automat vrací na začátek přes stav *reset*, protože je nutné opět nastavit čítač na hodnotu 26.

Tímto způsobem je z 26 vstupních hodnot vytvořena jedna výstupní. Generátor je schopen dodat jedno číslo na 106 taktů, což odpovídá frekvenci 471 kHz. To postačuje pro dva kanály s vzorkovací frekvencí 192 kHz

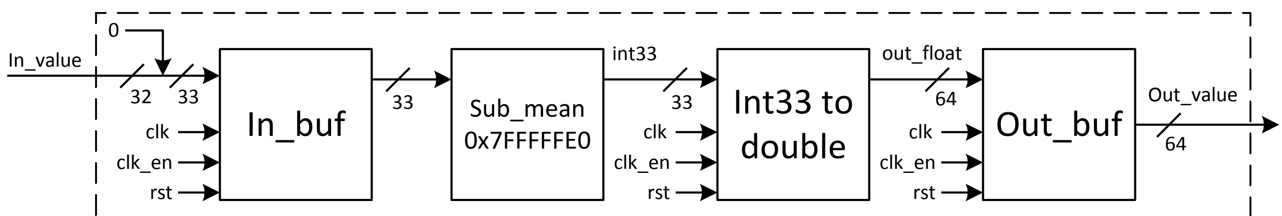
5.5 Převod do plovoucí řádové čárky

Entita *gauss_to_dbl* zajišťuje posunutí vstupních hodnot tak, aby jejich statistické rozdělení bylo symetrické kolem nuly a jejich převod do formátu double precision IEEE 754.

```
entity gauss_to_dbl is
  port (
    clk        : in  std_logic;
    rst        : in  std_logic;
    in_value   : in  std_logic_vector(31 downto 0);
    in_ready   : in  std_logic;
    in_accept  : out std_logic;
    out_value  : out std_logic_vector(63 downto 0);
    out_ready  : out std_logic;
    out_accept : in  std_logic);
end entity gauss_to_dbl;
```

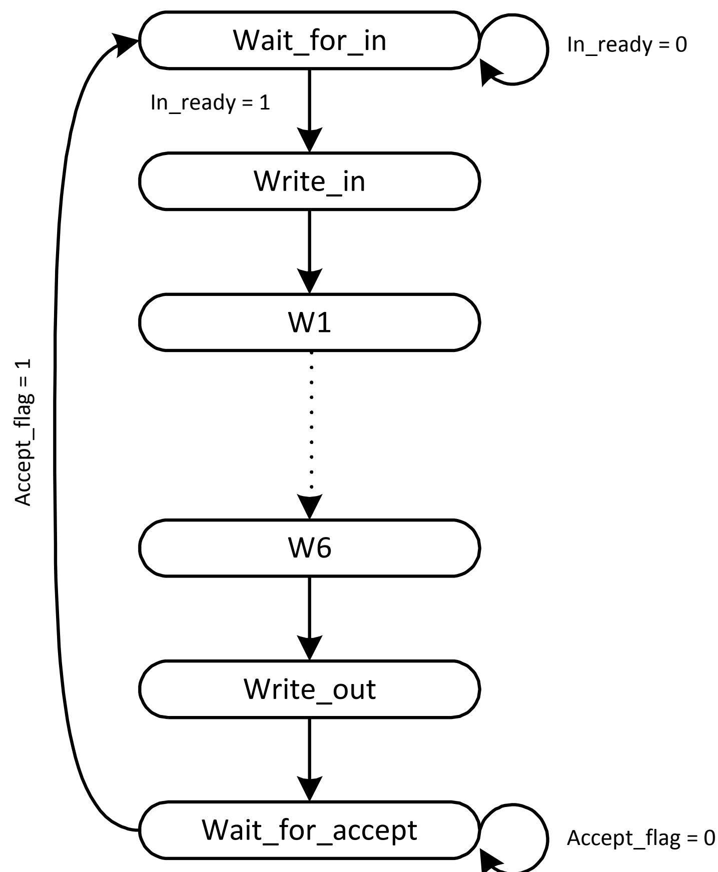
Kromě resetu, hodinového signálu a řízení toku dat v pipeline má tato entita pouze jediný vstupní a výstupní signál. Vstup *in_value* je 32b celé číslo bez znaménka a může obsahovat hodnotu 0x00000000 až 0xFFFFFFC0. Výstup *out_value* je ve formátu double precision a může obsahovat celé číslo z rozsahu -2147483616 až 2147483616.

Obr. 12 – Zjednodušené schéma architektury gauss_to_dbl



Posunutí vstupních hodnot se provádí odečtením teoretické střední hodnoty. Pro odečtení je využita entita *sub_mean* odvozená od knihovní funkce *lpm_add_sub*. Odčítačka pracuje s 33b operandy se znaménkem. Vstup je rozšířen na 33 bitů doplněním nulového bitu na nejvyšší pozici, druhý operand je vnitřně nastavná konstanta 0x7FFFFFE0. Rozšířením operandů na 33b je zároveň vyloučena možnost podtečení. Výsledný rozdíl je pomocí entity *int33_to_double* převeden do formátu s plovoucí řádovou čárkou. Tato entita je vytvořena pomocí Altera MegaCore *altfp_convert* a převod trvá 6 taktů.

Obr. 13 – Stavový diagram automatu gauss_to_double



Stavový automat je velmi jednoduchý a přímočarý. Reset udržuje stav *Wait_for_in*, kde automat i po jeho odeznění čeká na indikaci platné vstupní hodnoty signálem *in_ready*. Po jejím obdržení přepíše vstup do registru a potvrdí přijetí signálem *in_accept*. Následuje šest po sobě jdoucích stavů *W1* až *W6* kdy je aktivní převodník *int33_to_double*. Výsledek se zapíše do výstupního registru a signálem *out_ready* se indikuje nová platná hodnota. Poté automat čeká, dokud další část řetězce nepotvrdí její přijetí a následně se proces opakuje.

5.6 Dělení na dva kanály

Jedna z nejsložitějších entit je *splitter*. Zajišťuje napojení dvou zvukových kanálů na jeden generátor pseudonáhodných čísel. Do kanálů mohou být posílána různá nebo totožná data. Oba kanály je také možné nastavením splitteru umlčet. První způsob umlčení je zapisování nul na vstupy, zatímco veškeré následné zpracování stále probíhá. Druhý způsob je úplné zamezení funkce pipeline blokováním signálů *l_ready / r_ready*.

```
entity splitter is
  port
    clk       : in  std_logic;
    rst       : in  std_logic;
    lr_split  : in  std_logic;
    left_on   : in  std_logic;
    right_on  : in  std_logic;
    in_value  : in  std_logic_vector(63 downto 0);
    in_ready  : in  std_logic;
    in_accept : out std_logic;
    l_value   : out std_logic_vector(63 downto 0);
    l_ready   : out std_logic;
    l_accept  : in  std_logic;
    r_value   : out std_logic_vector(63 downto 0);
    r_ready   : out std_logic;
    r_accept  : in  std_logic);
end entity splitter;
```

Entita má jedno vstupní rozhraní pro hodnoty z generátoru a dvě výstupní rozhraní pro další zpracování dat v pipeline. Datové signály *in_value*, *l_value* a *r_value* jsou ve formátu double float. Oznamovací a potvrzovací signály mají obvyklé významy a chování.

Kromě resetu a hodinového signálu je zde pak ještě trojice vstupů pro nastavení pracovního režimu. Signály *left_on* a *right_on* stanovují, zda má být levý či pravý kanál aktivní či umlčen a *lr_split* udává, zda jim budou poskytována stejná, nebo různá data. Chování pro jednotlivé kombinace těchto signálů udává následující tabulka :

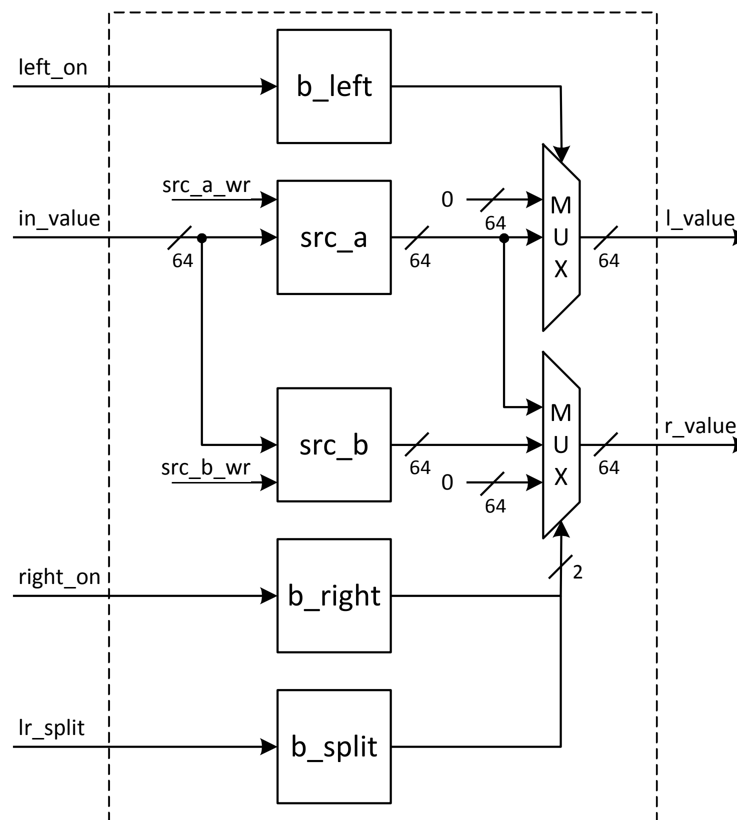
Tab. 1 – Režimy splitteru

LR_Split	***Left_on / Right_on***	***Levý kanál / Pravý kanál***
0	0	Blokován
0	1	Shodná data
1	0	Nuly
1	1	Různá data

Splitter obsluhuje maximálně dva aktivní nezávislé kanály. V tomto režimu využívá dva vstupní buffery *src_a* a *src_b*, do kterých střídavě zapisuje data dodávaná generátorem. Ve všech ostatních režimech využívá jen buffer *src_a*.

Signály pro nastavení režimu *lr_split*, *left_on* a *right_on* jsou zapisovány do registrů *b_split*, *b_left* a *b_right.* Jelikož jde o signály řídící přechody stavového automatu, je nežádoucí aby se jejich hodnota mohla měnit v libovolném okamžiku. Zápis do registru je proto prováděn v době, kdy automat čeká na odběr výsledků. Hodnoty v těchto registrech jsou také využity k blokování signálů pro oznamování výstupních hodnot *l_ready / r_ready* a potvrzení jejich převzetí *l_accept / r_accept.* Tím je zajištěno zablokování umlčeného kanálu při hodnotě signálu *lr_split = 0.*

Obr. 14 – Zjednodušené schéma architektury splitter



Podle aktuálně nastaveného režimu jsou na výstupy připojeny buď vstupní registry, nebo nulové konstanty. Pokud je kanál blokován, má přes multiplexor také nastavenou nulu, ale navíc nepřijímá signál *accept* ani nevydává signál *ready*. Zápisy do registrů a stavy řídicích signálů obsluhuje stavový automat.

Navzdory poměrně složitému stavovému automatu není jeho činnost příliš komplikovaná. Diagram na obrázku 15 na další stránce je poněkud zpřehledněn. Návraty do stavu *Split* ze spodní poloviny jsou pouze ve formě popisku. Mnoho stavů má shodný význam a mají pak shodné označení, rozlišené číslem. Jména stavů jsou pak zkratky popisující jejich význam. *WfIN* jsou stavy, kdy automat čeká na vstupní data, při *SetA* a *SetB* se zapisuje do vstupních bufferů. *WfAcc* jsou stavy, kdy se čeká na odběr výsledků a několik dalších bude vysvětleno dále.

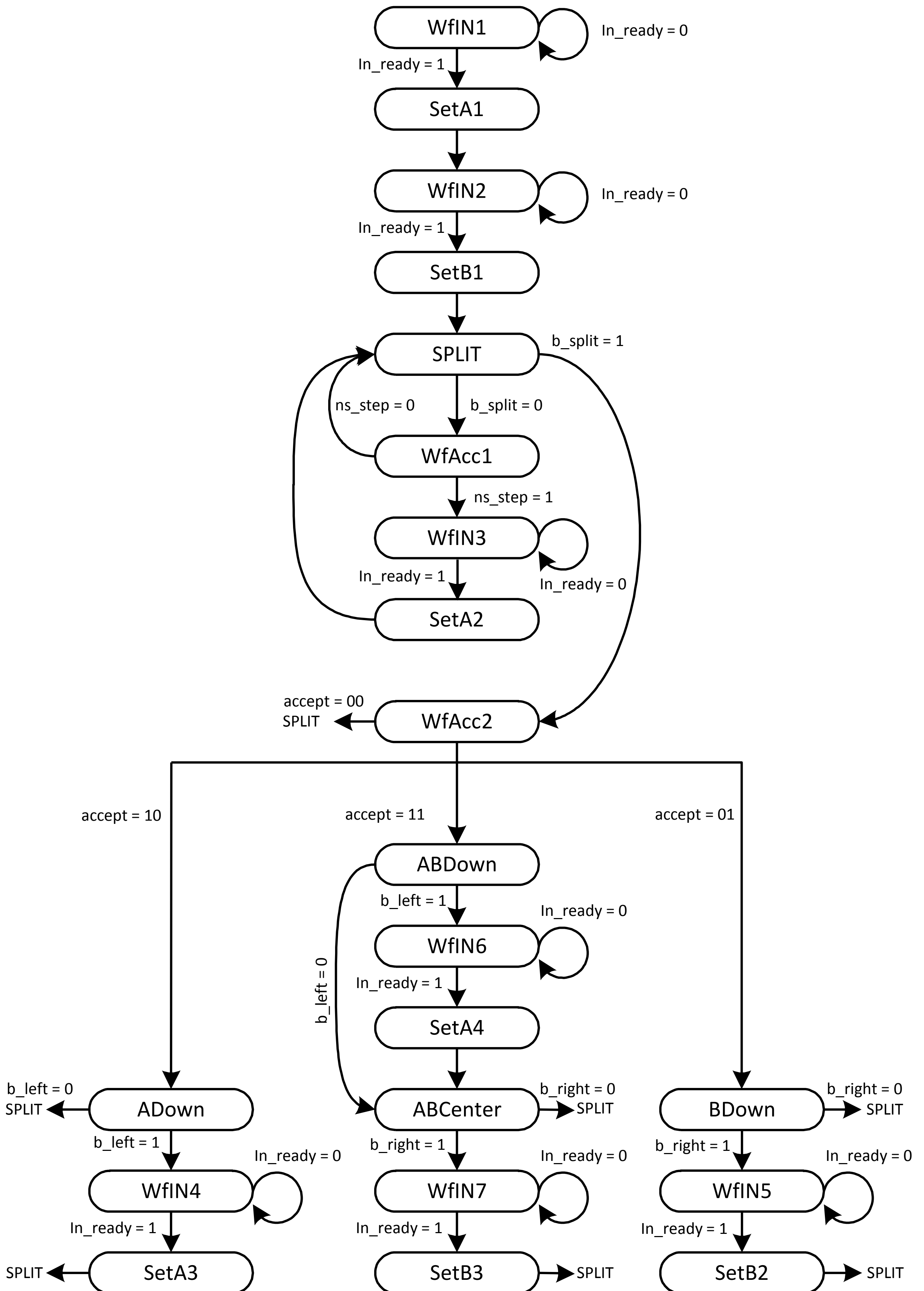
Na začátku činnosti, po uvolnění resetu, automat zapíše úvodní hodnoty do obou vstupních registrů a přejde do ústředního stavu *Split*. V závislosti na hodnotě signálu *b_split* pak přechází do jednoho ze stavů *WfAcc* a pokud nebyl potvrzen odběr výsledku některým z kanálů, opět se vrací do *Split*.

Pokud je *b_split = 0*, mají oba kanály dostávat stejná data. Tento jednoduchý případ je obsloužen *WfIN2* a *SetA2*. Každý aktivní kanál, tedy žádný, jeden nebo oba, se napojí na registr *src_a*. Signál *ns_step* nabývá hodnoty 1 pouze v případě, že všechny aktivní kanály potvrdily odběr předchozího výsledku. Neaktivní kanály zůstávají blokovány a na stav jejich signalizace se nebere ohled.

Pokud je *b_split = 1*, jsou oba kanály napojeny na vlastní vstupní registr a jsou zásobovány z generátoru střídavě. Neaktivní kanály v tomto nastavení pracují, ale místo dat z generátoru je v jejich registru trvale nula. Zde je třeba ve stavu *WfAcc2* rozlišovat, který z kanálů potvrdil odběr výsledku. Pokud jen levý, pokračuje se levou větví diagramu na *ADown*, pokud oba, pak střední větví na *ABDown* a v případě pravého kanálu je to zbývající pravá větev začínající *BDown*.

V těchto „Down“ stavech se nuluje odpovídající *accept_flag* a rozhoduje se, zda je kanál aktivní a je tedy třeba odebrat hodnotu z generátoru, nebo se mu jako neaktivnímu posílá nula. V tom případě se odběr hodnoty z generátoru přeskočí. Tím je docíleno toho, že čísla z generátoru jsou odebírána pouze v případě, že je jich reálně zapotřebí.

Obr. 15 – Stavový diagram automatu splitter



5.7 Číslicový filtr

Reg_filter – nejsložitější entita v projektu, představuje speciální zapojení číslicového filtru. Proces výpočtu je plně sekvenční a pracuje se vzorky i koeficienty ve formátu double float. Dle struktury jde o kanonickou formu IIR druhého řádu. Není zcela obecná, pro zjednodušení předpokládá některé triviální koeficienty. Opakovaným průchodem vzorku a multiplexováním stavových registrů se vytváří ekvivalent kaskády tří sekcí. Celá kaskáda pak odpovídá Butterworthově aproximaci pásmové propusti šestého řádu s nastavitelým ziskem.

```
entity reg_filter is
  port (
    clk         : in  std_logic;
    rst         : in  std_logic;
    in_data     : in  std_logic_vector(63 downto 0);
    in_ready    : in  std_logic;
    in_accept   : out std_logic;
    out_data    : out std_logic_vector(63 downto 0);
    out_ready   : out std_logic;
    out_accept  : in  std_logic;
    rom_data    : in  std_logic_vector(63 downto 0);
    rom_addr    : out std_logic_vector(15 downto 0);
    rom_rdclk   : out std_logic;
    base_addr   : in  std_logic_vector(15 downto 0);
    gain        : in  std_logic_vector(63 downto 0));
end entity reg_filter;
```

Mezi porty entity *reg_filter* jsou obvyklé signály pro předávání dat v pipeline, reset a hodinový signál. Navíc je zde rozhraní pro napojení paměti koeficientů, signál s hodnotu zesílení a základní adresa do paměti koeficientů.

Jak vstupní *in_data*, tak i výstupní *out_data* jsou ve formátu double float, oznamování a potvrzování dat v pipeline má obvyklé chování. Také signál *rom_data* je slovo ve formátu double float čtené z adresy *rom_addr*. Okamžik čtení z paměti je řízen r*om_rdclk,* který má význam „clock enable“. Koeficienty filtru jsou navrženy na jednotkový zisk a po provedení filtrace se šum násobí hodnotou double floatu *gain*.

Zbývající signál *base_addr* slouží k výběru sady koeficientů pro filtr. Adresa do paměti je v průběhu procesu určována součtem *base_addr* s číslem právě potřebného koeficientu.

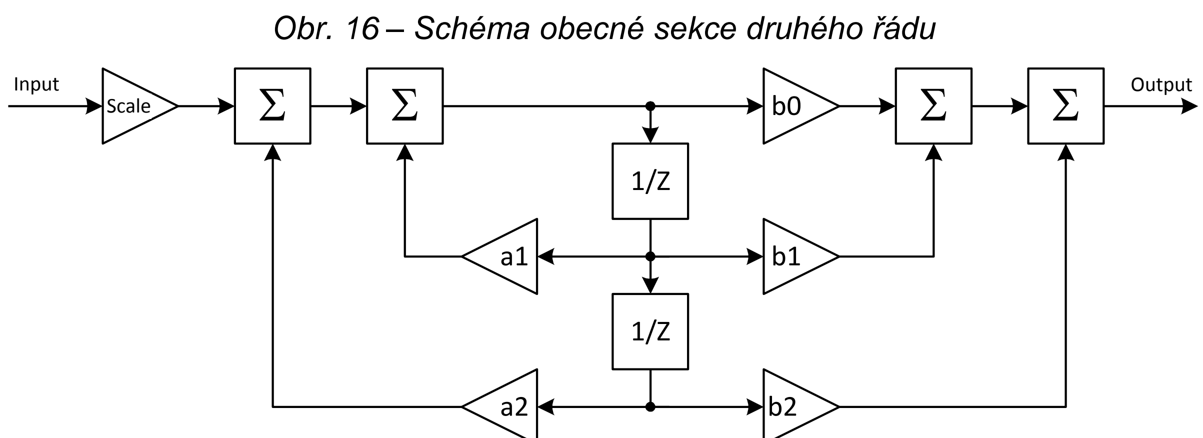
Filtr ke své činnosti využívá několik komponent :

- Šestnáctibitovou celočíslnou sčítačku k určení adresy koeficientu *add_2u16*. Jde o odvozeninu od knihovní funkce *lpm_add_sub*. Jelikož počet koeficientů v paměti je výrazně menší, než rozsah adres, k přetečení při řádné funkci nemůže dojít.
- Double float sčítačku / odečítačku pro aritmetiku filtrace *add_sub_double*. Tato komponenta je vytvořena pomocí megafunkce *altft_add_sub* a výpočet součtu či rozdílu trvá 7 taktů.
- Double float násobičku pro aritmetiku filtrace a aplikaci zesílení *mul_double*. Tato je vytvořena pomocí megafunkce *altft_mult* a násobí na 5 taktů.
- Paměť instrukcí – postupných nastavení multiplexorů a dalších signálů ve struktuře filtru. Slova v paměti *reg_filter_irom* definují postup filtrace. Jde o megafunkci využívající paměťové bloky M4K ve funkci jednoportové ROM.
- Vzestupný čítač vytvářející adresu do paměti instrukcí *cnt_8b_aclr*. Komponenta je odvozenina knihovní funkce *lpm_counter*.

Potřebný filtr šestého řádu je možné vytvořit pomocí několika struktur. Některé výhody, například menší citlivost na přesnost koeficientů, jsou v double float aritmetice irelevantní. Různá náročnost na počet aritmetických operací je zde důležitějším kritériem. Struktura s minimálním počtem násobení zde není nejvýhodnější, protože neušetří násobičky a navíc prodlouží výpočet zvýšeným počtem sčítání.

Filtr v kompaktním tvaru i ve formě kaskády sekcí druhého řádu vyžaduje shodný počet aritmetických operací, ovšem kaskáda bude mít jednodušší strukturu. Pracovní registry se mohou sdílet. Jednu sekci druhého řádu pro kaskádní zapojení filtru ukazuje obrázek 16. V této podobě výpočty sekce obnáší šest násobení a čtyři sčítání.

Obr. 16 – Schéma obecné sekce druhého řádu



Jelikož filtr je Butterworthova aproximace pásmové propusti, lze zapojení zjednodušit. Některé koeficienty mohou mít triviální hodnotu což ušetří aritmetické operace.

Původní přenosová funkce obecné sekce druhého řádu

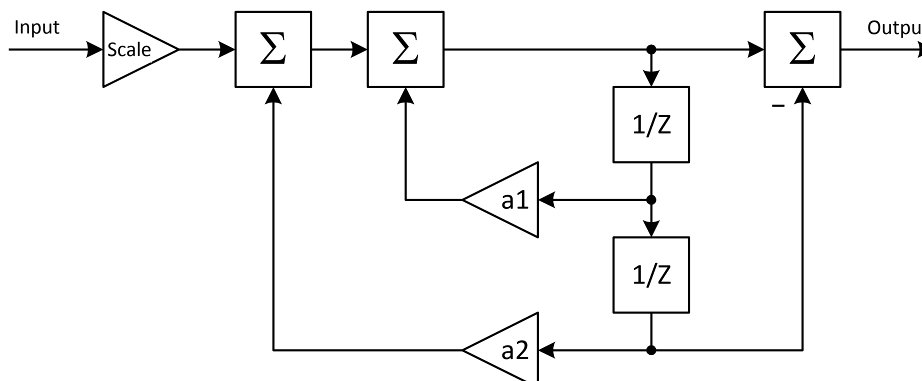
$$H(Z) = Scale \cdot \frac{b_0 + b_1 Z^{-1} + b_2 Z^{-2}}{1 - a_1 Z^{-1} - a_2 Z^{-2}} \tag{5.7.1}$$

má v našem jednodušším případě tvar

$$H(Z) = Scale \cdot \frac{1 - Z^{-2}}{1 - a_1 Z^{-1} - a_2 Z^{-2}} \tag{5.7.2}$$

Takto zjednodušená sekce pracuje se třemi násobeními a třemi sčítáními. Násobení -1 lze provést triviální inverzí jednoho bitu. Následující obrázek ukazuje schéma zjednodušené sekce.

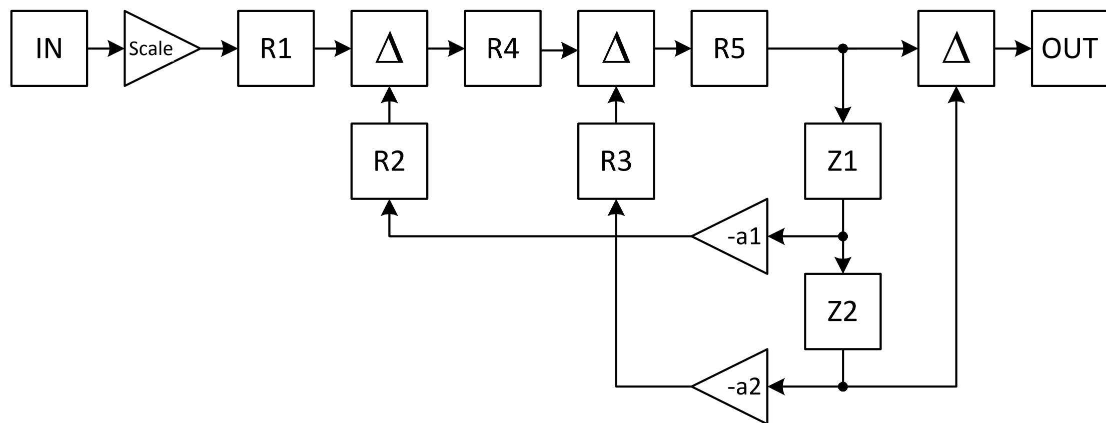
Obr. 17 – Schéma zjednodušené sekce druhého řádu



Nutnosti měnit znaménko před posledním sčítáním se lze úplně vyhnout použitím odčítačky namísto sčítačky. Jelikož z hlediska návrhu filtru v systému Matlab je výhodné mít opačná znaménka i u koeficientů a_n, nahradíme sčítačky ve schématu odčítačkami.

Při skutčné implementaci bude výpočet prováděn sekvenčně pouze jednou násobičkou a jednou odčítačkou. Operandy i výsledky aritmetických operací se proto musí ukládat do registrů. Schéma takové sekce včetně vstupních, pracovních i výstupních registrů je na obrázku 18 na další stránce.

Obr. 18 – Schéma zjednodušené sekce včetně registrů



Sekvenčně pak lze provést výpočet například takto v osmi krocích :

1) R1 = IN × Scale
2) R2 = Z1 × a1
3) R3 = Z2 × a2
4) R4 = R1 – R2
5) R5 = R4 – R3
6) OUT = R5 – Z2
7) Z2 = Z1
8) Z1 = R5

Vhodnou změnou pořadí výpočtů můžeme docílit toho, že v době výpočtu Z2 × a2 již nebude nutné znát hodnotu R2. Takto můžeme registr R2 opět využít k uložení součinu a registr R3 nemusí vůbec existovat. Úspornější varianta pracuje takto :

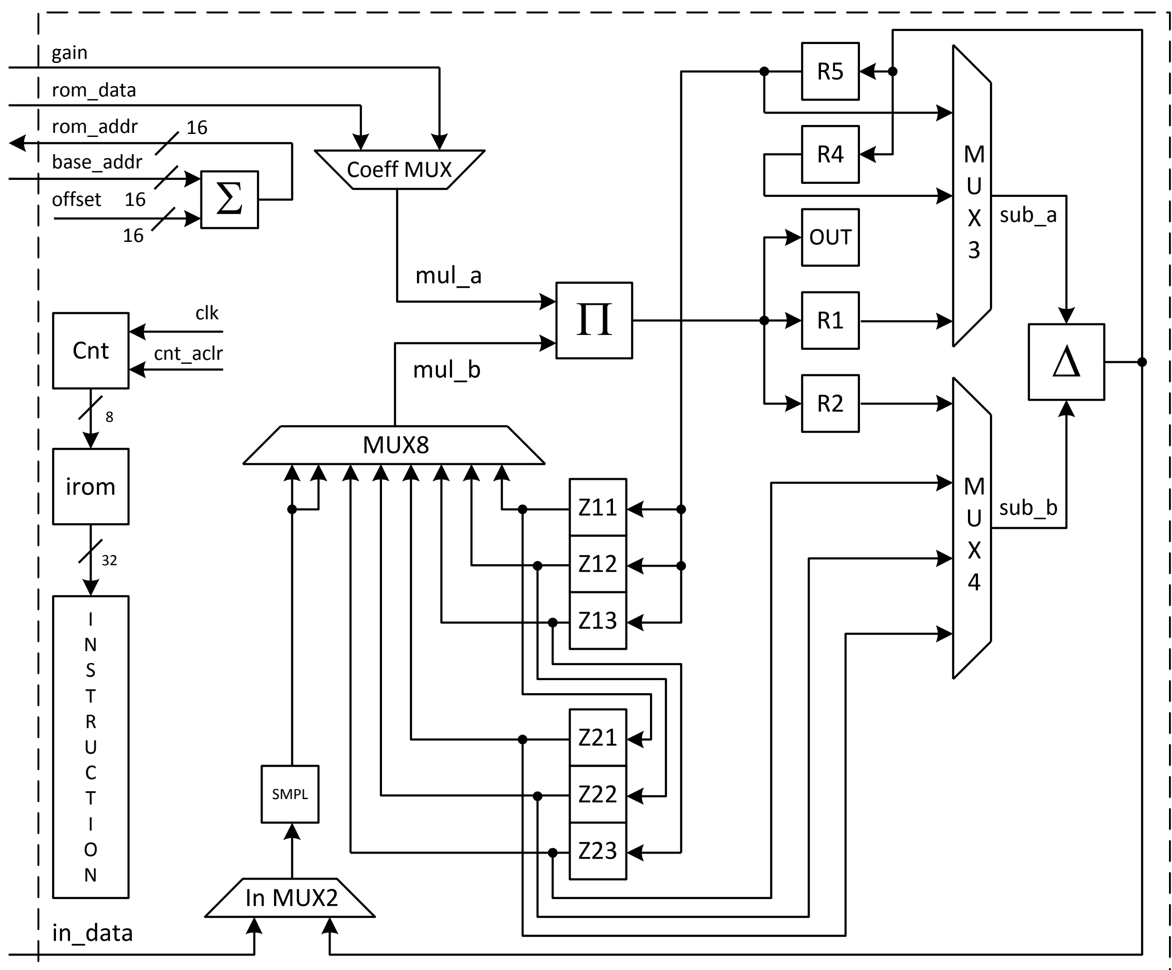
1) R1 = IN × Scale
2) R2 = Z1 × a1
3) R4 = R1 – R2
4) R2 = Z2 × a2
5) R5 = R4 – R2
6) OUT = R5 – Z2
7) Z2 = Z1
8) Z1 = R5

Skutečné provedení entity *reg_filter* pak odráží právě popsanou sekci druhého řádu spolu s výpočetním postupem. Schéma struktury je na obrázku 19. Pracovní registry R1, R2, R4 a R5 jsou dle potřeby přes multiplexory připojovány k násobičce nebo odčítačce. Tyto součásti se používají opakovaně, pouze stavové registry musí být přítomny trojmo.

Dále je nutné do vstupního registru zapisovat jak vstupující data, tak i výsledek výpočtu předchozí sekce. Stejně tak jeden z operandů pro násobení je potřeba nakonec nahradit signálem *gain*. To znamená další dva multiplexory. Celkově tedy v průběhu filtrace musíme ovládat pět multiplexorů a ve vhodné okamžiky zapisovat do dvanácti registrů.

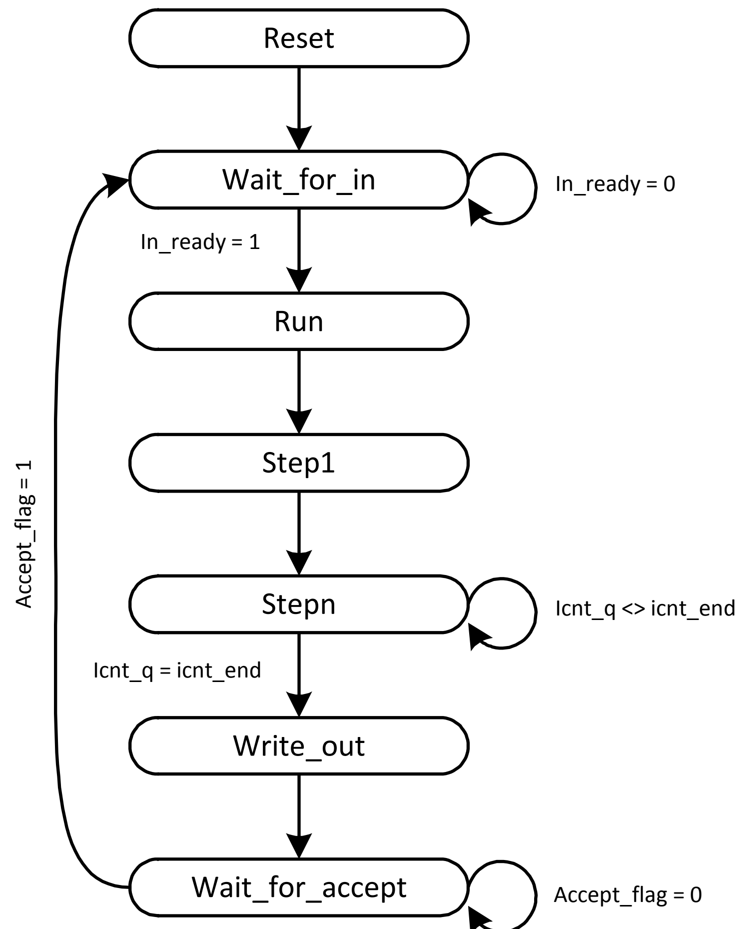
Sigály pro ovládání celé struktury jsou uloženy pro každý krok výpočtu jako slovo v paměti *reg_filter_irom*. Proces filtrace pak probíhá tak, že jsou tato slova postupně všechna přečtena a použita k řízení činnosti. Každé slovo přečtené z *reg_filter_irom* a zapsané do registru *instruction* obsahuje offset pro výpočet adresy koeficientu, nastavení multiplexorů a aktivaci hodinového signálu pro registry a aritmetické komponenty.

Obr. 19 – Zjednodušená architektura reg_filter



Pozn. : Všechna propojení mají šířku 64 bitů s výjimkou jinak označených.

Obr. 20 – Stavový diagram automatu reg_filter



Činnost *reg_filter* je následující : Aktivní signál *rst* udržuje automat ve stavu *reset*, kdy jsou zároveň resetovány aritmetické komponenty a nulován čítač *cnt* spolu se stavovými registry Z11 – Z23. Po uvolnění *rst* automat přechází do *wait_for_in* kdy čeká na platná vstupní data.

Po obdržení signálu *in_ready* je ve stavu *run* uvolněno nulování čítače, vše zatím zůstává v klidovém nastavení. O takt později ve stavu *step1* je přečtena první instrukce z paměti *reg_filter_irom* která provede nastavení multiplexorů, zápis vstupu do pracovního registru *smpl* a načtení prvního potřebného koeficientu.

Další takty automat setrvává ve stavu *stepn* kdy je postupně inkrementován čítač *cnt*, a jsou čteny další instrukce. Zde také dojde k potvrzení převzetí dat nastavením *in_accept*. Sekvence instrukcí postupně nastavuje propojení komponent, zápisy do registrů a aktivitu aritmetických komponent tak, aby proběhly potřebné výpočty. Postupně se vypočítá průchod první sekcí filtru, výsledek se zapíše zpět do registru *smpl*, stejně se provede výpočet průchodu druhou a třetí sekcí. Postup se liší pouze načtenými koeficienty a použitými

stavovými registry. Nakonec se ještě provede násobení hodnotou *gain* a výsledek je zapsán do výstupního registru *out_data*.

Celý proces filtrace znamená načtení 195 instrukčních slov. Když čítač dosáhne koncové hodnoty *icnt_end*, přechází automat přes *write_out* do *wait_for_accept*, kdy je již na výstupu platná hodnota a je nastaven signál *out_ready*. Zde se čeká na převzetí výsledku dalším článkem řetězce.

Po převzetí výsledku se automat vrací do *wait_for_in* kdy je nulován čítač *cnt* a čeká se na nová vstupní data. Návrat neprobíhá přes *reset*, protože se musí zachovat obsah stavových registrů, který se účastní dalších výpočtů.

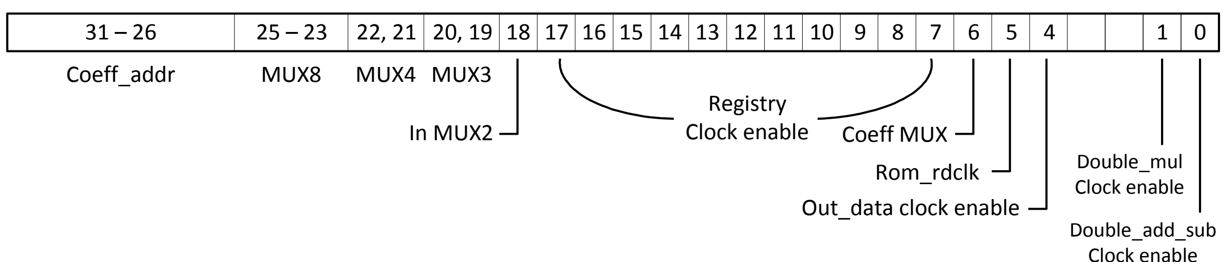
5.7.1 Formát instrukčního slova

Vlastní činnost filtru určuje obsah paměti *reg_filter_irom*, který obsahuje 32b slova. Každé slovo odpovídá konkrétnímu nastavení mnoha signálů. Rozdělení bitů instrukčního slova je schematicky znázorněno na obrázku 21.

Od nejnižších bitů jde o aktivaci hodin pro double float sčítačku a násobičku. Bity 2 a 3 jsou nevyužité. Bit 4 povoluje zápis do výstupního registru. Následuje aktivace čtení z paměti koeficientů a nastavení multiplexoru, který přepíná mezi koeficienty a signálem *gain*.

Bity 7 – 17 povolují zápisy do registru SMPL, stavových registrů Z23, Z22, Z21, Z13, Z12, Z11, a pracovních registrů R5, R4, R2 a R1. Následují adresy zbývajících multiplexorů a v nejvyšších šesti bitech je offset k výpočtu adresy koeficientu.

Obr. 21 – Význam bitů v instrukčním slově



5.7.2 Organizace paměti koeficientů

Paměť koeficientů *rom_1024x64* je vytvořena pomocí megafunkce *altsyncram* konfigurované jako dvouportová ROM. Filtr levého i pravého kanálu je napojen na vlastní port, takže mohou paměť velmi snadno sdílet.

Každá sada koeficientů představuje devět slov, od nejnižší adresy jsou postupně uloženy koeficienty pro tři sekce : *Scale1, a11, a21, Scale2, a12, a22, Scale3, a13, a23.*

První sada, počínaje adresou nula obsahuje koeficienty 1, 0, -1 pro všechny tři sekce. Tím je dosaženo přenosové funkce

$$H(Z) = Scale \cdot \frac{1+Z^{-2}}{1-a_1Z^{-1}-a_2Z^{-2}} = 1 \cdot \frac{1+Z^{-2}}{1+Z^{-2}} = 1 \qquad (5.7.2.1)$$

S těmito koeficienty filtr nemění procházející signál. Tato sada je použita, pokud se vyžaduje širokopásmový šum. Následuje jedenáct oktávových a 32 třetinoktávových pásmových propustí. Celý výpočet, od stanovení hraničních frekvencí, přes výpočet koeficientů až po vytvoření souboru s obsahem paměti *rom_1024x64.mif* je proveden skriptem systému Matlab. Pro oktávové filtry jsou stanoveny centrální a hraniční frekvence pásem takto :

$$f_{C1} = 1kHz \cdot 2^{n} \qquad (5.7.2.2)$$

$$f_{L1} = f_C \cdot 2^{-1/2} \qquad (5.7.2.3)$$

$$f_{H1} = f_C \cdot 2^{1/2} \qquad (5.7.2.4)$$

kde pro pokrytí rozsahu 20 Hz – 48 kHz je parametr $n \in \{-5, -4, \dots, 4, 5\}$ a obdobně pro třetinoktávová pásma platí :

$$f_{C3} = 1kHz \cdot 2^{n/3} \qquad (5.7.2.5)$$

$$f_{L3} = f_C \cdot 2^{-1/6} \qquad (5.7.2.6)$$

$$f_{H3} = f_C \cdot 2^{1/6} \qquad (5.7.2.7)$$

zde pro pokrytí rozsahu 20 Hz – 48 kHz potřebujeme parametr $n \in \{-16, -15, \dots, 16, 17\}$. Takto stanovené hraniční frekvence jsou použity k výpočtu koeficientů filtru. Jelikož Matlab pracuje nativně s double floaty, není třeba dalších úprav. Skript získané koeficienty jen vhodně seřadí a vytvoří z nich soubor pro inicializaci paměti.

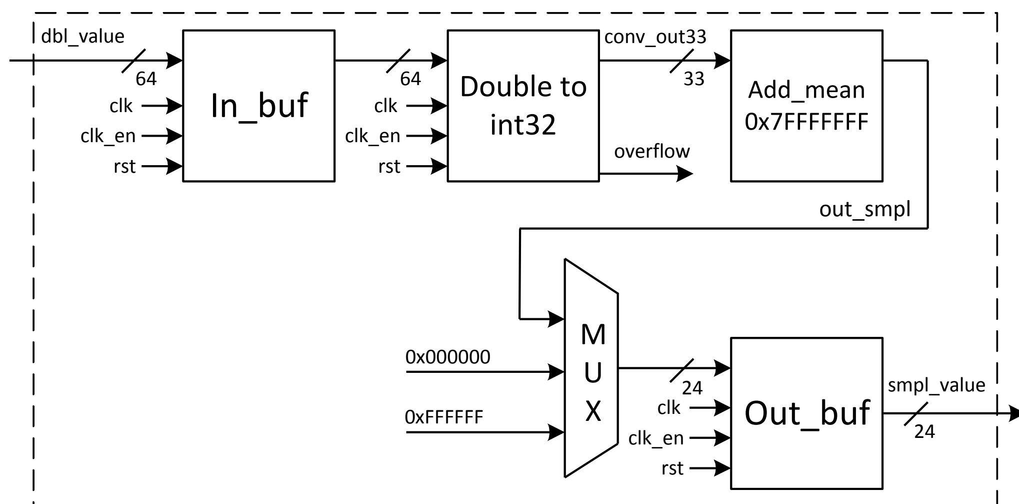
5.8 Převod do kódu s posunutou nulou

Entita *dbl_to_smpl* převádí hodnoty ve formátu double precision IEEE 754 na vzorky pro zvukový kodek. Jde o 24b celá čísla v kódu s nulou v polovině rozsahu.

```
entity dbl_to_smpl is
  port (
    clk         : in  std_logic;
    rst         : in  std_logic;
    dbl_value   : in  std_logic_vector(63 downto 0);
    dbl_ready   : in  std_logic;
    dbl_accept  : out std_logic;
    smpl_value  : out std_logic_vector(23 downto 0);
    smpl_ready  : out std_logic;
    smpl_accept : in  std_logic);
end entity dbl_to_smpl;
```

Kromě resetu, hodinového signálu a řízení toku dat v pipeline má tato entita opět jediný vstupní a výstupní signál. Vstup *dbl_value* je double float. Nominální rozsah hodnot $\pm 2^{31}$ který odpovídá plnému rozsahu výstupního signálu může být překročen. Výstup *smpl_value* je celé číslo 0 až $2^{24} - 1$, nulová hodnota vstupního signálu odpovídá 2^{23} na výstupu.

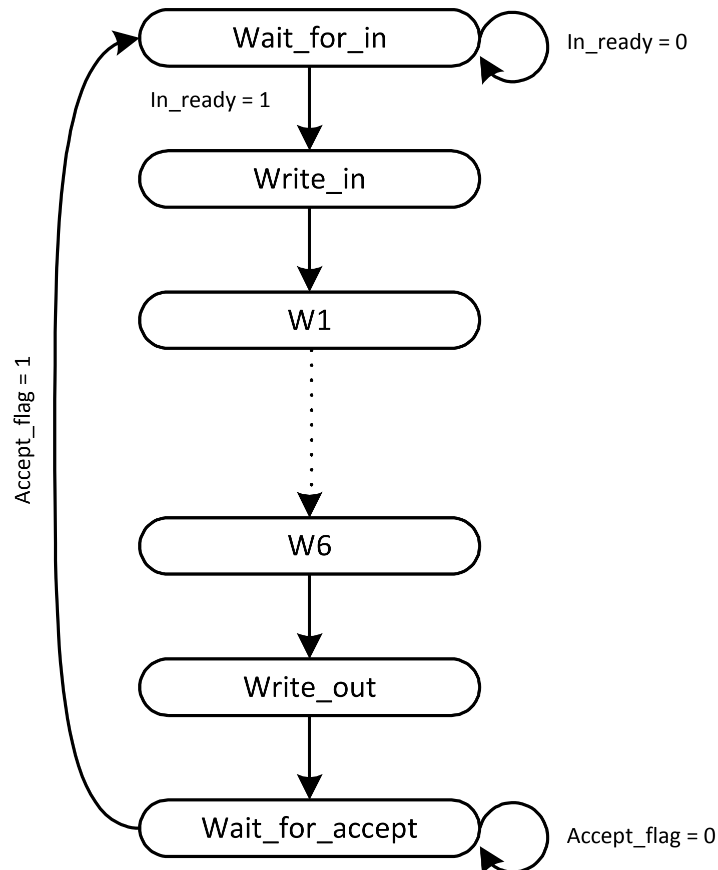
Obr. 22 – Zjednodušené schéma architektury dbl_to_smpl



Vstupní hodnota je nejprve pomocí *double_to_int32* převedena na 32b číslo se znaménkem. Tato entita je opět vytvořena pomocí Altera MegaCore *altfp_convert* a převod trvá 6 taktů. Při této konverzi může dojít k přetečení, které bude dále zohledněno. Výsledek je rozšířen na 33 bitů duplikováním nejvýznamnějšího bitu. To zajišťuje, že v následujícím přičtení nedojde k dalšímu přetečení. Sčítačka *add_mean* provádí posun přičtením 2^{31}.

Takto je nominální vstupní rozsah $\pm 2^{31}$ převeden do celočíslného rozsahu 0 až 2^{32}. Výsledek je pak horních 24 bitů součtu. Pouze v případě, že při konverzi na celé číslo došlo k přetečení, pak je dle znaménka vstupu zapsána krajní hodnota 0 či $2^{24} - 1$.

Obr. 23 – Stavový diagram automatu dbl_to_smpl



Stavový automat je shodný automatem entity *gauss_to_double*, neboť činnost se v jednotlivých krocích významně neliší. Reset udržuje stav *Wait_for_in*, kde automat i po jeho odeznění čeká na indikaci platné vstupní hodnoty signálem *dbl_ready*. Po jejím obdržení přepíše vstup do registru a potvrdí přijetí signálem *dbl_accept*. Následujících šest po sobě jdoucích stavů *W1* až *W6* je aktivní převodník *double_to_int32*. K výsledku se přičítá posunutí a součet, případně krajní hodnota, se zapíše ve stavu *Write_out* do výstupního registru. Signálem *smpl_ready* se indikuje nová platná hodnota. Poté automat čaká, dokud další část řetězce nepotvrdí její přijetí a následně se proces opakuje.

5.9 Zápis do fronty vzorků

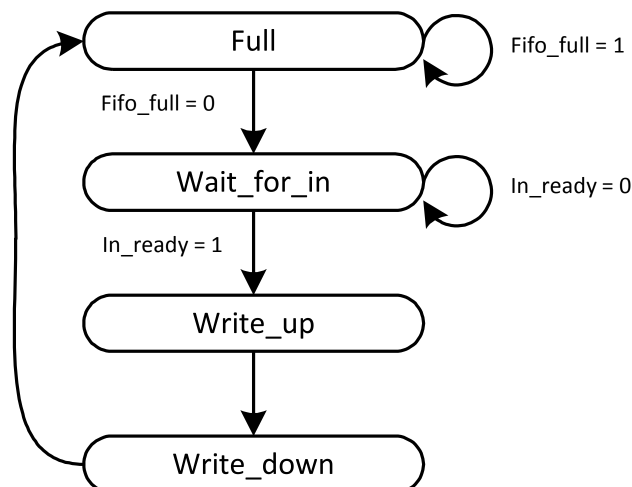
Entita *fifo_feeder* je velmi jednoduché rozhraní mezi frontou vzorků a zbytkem řetězce. Podle stavu zaplnění fronty odebírá vygenerované vzorky a zapisuje je.

```
entity fifo_feeder is
  port (
    clk         : in  std_logic;
    rst         : in  std_logic;
    in_data     : in  std_logic_vector(23 downto 0);
    in_ready    : in  std_logic;
    in_accept   : out std_logic;
    fifo_full   : in  std_logic;
    fifo_data   : out std_logic_vector(23 downto 0);
    fifo_wr_req : out std_logic);
end entity fifo_feeder;
```

Signály *clk*, *rst*, *in_ready* a *in_accept* mají obvyklý význam. Vstup *in_data* představuje zvukový vzorek. Zbylé signály jsou rozhraní s frontou. Vstup *fifo_full* idikuje její úplné zaplnění. Výstup *fifo_wr_req* je žádost o zápis hodnoty *fifo_data* do fronty.

Mezi porty *in_data* a *fifo_data* je přímé propojení, k žádnému zpracování zde již nedochází. Jelikož jde o poslední článek, řídí entita skrze řetězec potvrzování činnost celé pipeline. Veškerou funkčnost tvoří jeden triviální stavový automat obsluhující signalizaci.

Obr. 24 – Stavový diagram automatu fifo_feeder



Po resetu se začíná ve stavu *full*, zde automat zůstává v případě, že je fronta plná. Pokud plná není, čeká se na platnou vstupní hodnotu. Po obdržení signálu *in_ready* se ve stavu *write_up* nastaví žádost o zápis do fronty. K zápisu dojde o takt později ve stavu *write_down*, tehdy se také potvrdí přijetí hodnoty od předchozího článku pipeline a proces se opakuje.

5.10 Rozhraní mezi frontami a kodekem

Entita *audio_output_2ch* čte zvukové vzorky ze dvou front a sériově je předává kodeku. Tato součást není synchronní se zbytkem projektu a využívá hodinového signálu poskytnutého kodekem.

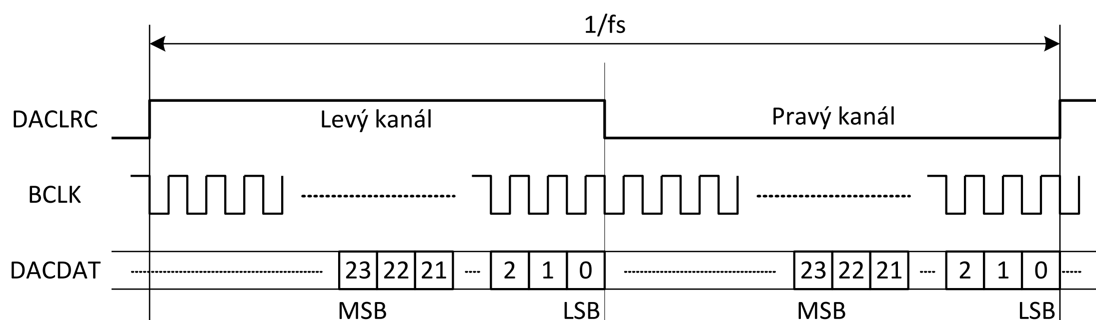
```
entity audio_output_2ch is
  port
    bclk   : in std_logic;
    daclrc : in std_logic;
    dacdat : out std_logic;
    fifo_l_data  : in  std_logic_vector(23 downto 0);
    fifo_l_rdreq : out std_logic;
    fifo_l_rdclk : out std_logic;
    fifo_r_data  : in  std_logic_vector(23 downto 0);
    fifo_r_rdreq : out std_logic;
    fifo_r_rdclk : out std_logic);
end entity audio_output_2ch;
```

K napojení na fronty slouží dvě sady signálů. Žádosti o čtení *fifo_l_rdreq / fifo_r_rdreq*, hodinové signály pro čtení *fifo_l_rdclk / fifo_r_rdclk* a samotné výstupy front s hodnotami vzorků *fifo_l_data / fifo_r_data*.

Vstup *bclk* je hodinový signál poskytovaný v master módu zvukovým kodekem. Signál *daclrc* svou úrovní udává, zda se právě přenášená data přísluší levému, nebo pravému kanálu. Zároveň jsou na jeho hrany zarovnána přenášená data. Oba signály jsou synchronní a v tomto případě platí, že *daclrc* = *bclk* / 128. Přenášené bity v signálu *dacdat* kodek čte na náběžnou hranu *bclk*.

Možností, jak přenést jednotlivé bity vzorků vzhledem ke hranám *daclrc,* je několik a závisí na nastavení kodeku. Komunikaci mezi entitou a kodekem při 24b slově a v režimu „right justify“ ukazuje následující diagram :

Obr. 25 – Časový diagram komunikace s kodekem



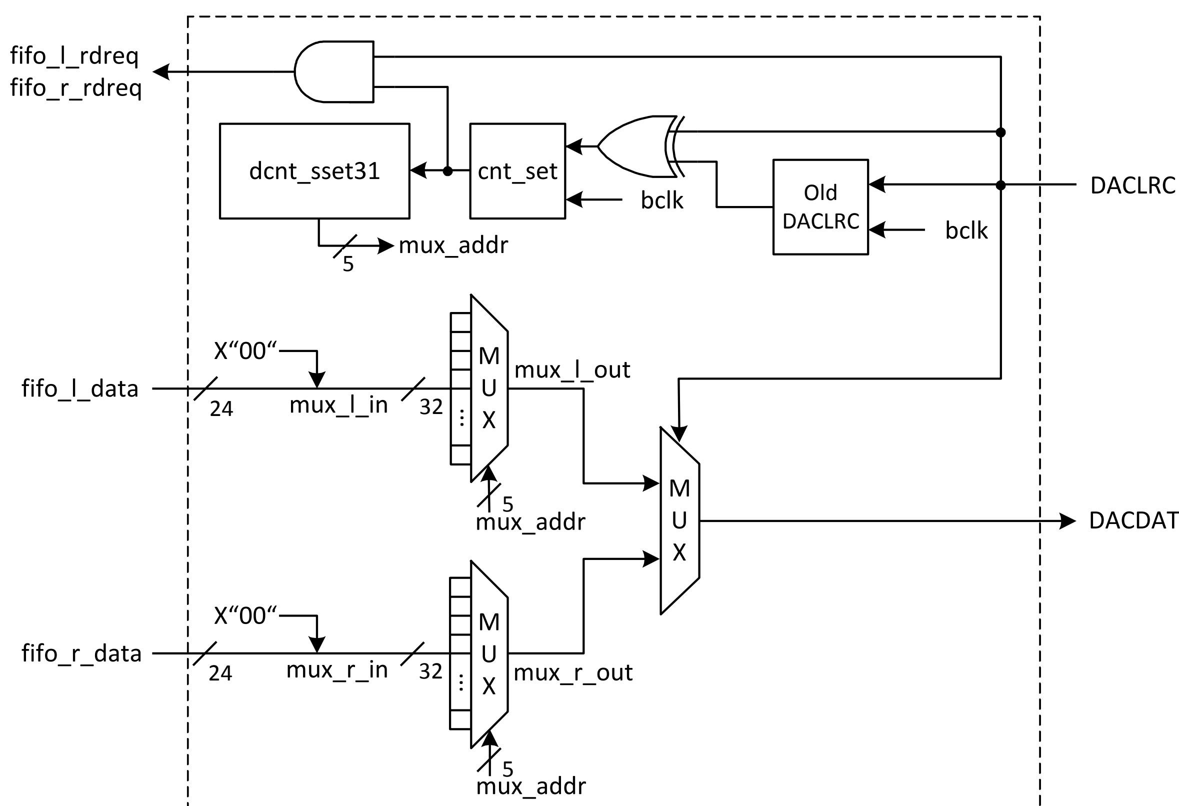
Z diagramu na obrázku 25 je vidět, že hrany signálu daclrc odpovídají sestupným hranám *bclk*. Do registru *old_daclrc* je zapisováno na vzestupnou hranu a spolu s XOR hradlem tak tvoří detektor hran. Jeho výstup je na první vzestupnou hranu *bclk* po hraně *daclrc* zapsán do registru *cnt_set*. Ten slouží k nastavení hodnoty čítače a k žádosti o čtení z fronty.

Pětibitový čítač *dcnt_sset31* je ve vhodný okamžik nastaven na maximální hodnotu, tím se synchronizuje výběr bitů multiplexory s hranami *daclrc*. Na sestupnou hranu *bclk* se dekrementuje a během první poloviny cyklu z obrázku 25 jednou podteče. V jedné půlperiodě *daclrc* jsou obsaženy právě dvě periody čítače, proto zůstává nastavení multiplexorů stále správné.

Čtení z front je také ovládáno *cnt_set*, je ale hradlováno *daclrc* aby ke čtení došlo jen po vzestupné hraně tohoto signálu, zatímco samotné *cnt_set* je aktivní i po sestupné. Čtení tedy probíhá v době, kdy změny vstupů nevadí. Platné hodnoty pro kodek jsou odesílány až v posledních 24 taktech z 64.

Vstupy *fifo_l_data / fifo_r_data* jsou rozšířeny na 32 bitů nulami a jednotlivé bity takto vzniklých signálů *mux_l_in / mux_r_in* jsou vybírány multiplexory. Signál *daclrc* pak přepíná poslední multiplexor a stanovuje tak, zda jsou odesílány bity pro pravý nebo levý kanál.

Obr. 26 – Schéma architektury audio_output_2ch

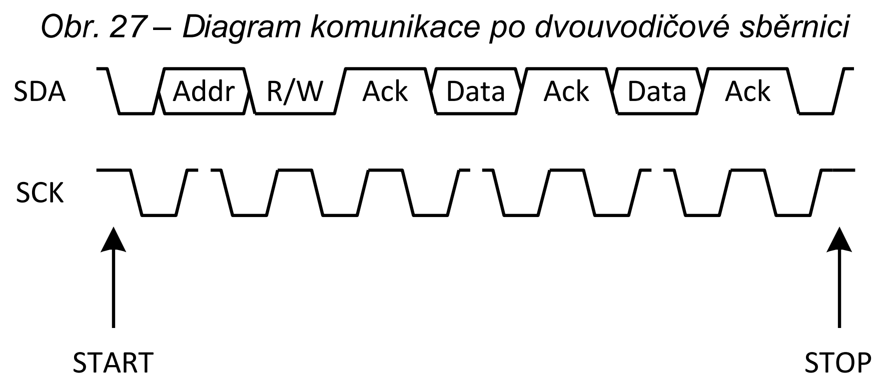


5.11 Konfigurace kodeku

Kodek obsahuje sadu registrů pro nastavení parametrů a funkčních režimů. Do těchto registrů je možné zapisovat přes dvojvodičové rozhraní, které je zjednodušenou obdobou I^2C.

Komunikace po sběrnici probíhá v několika krocích. Nejprve se vyvolá START stav, který oznamuje zařízením na sběrnici, že začíná vysílání. Pak se odvysílají tři bloky dat po osmi bitech. Po každém bloku se vysílá bit s hodnotou jedna, kdy adresát potvrzuje příjem uvedením sběrnice do nuly jakožto dominantního stavu. Vysílání je zakončeno STOP stavem. Tento postup je zobrazen na následujícím diagramu :

Obr. 27 – Diagram komunikace po dvouvodičové sběrnici



Tímto způsobem je zapsáno do jednoho registru. První blok dat je pouze sedmibitová adresa zařízení a bit určující, že provádíme zápis. Horních 7 bitů v druhém bloku je adresa registru, do kterého se zapisuje a spodní bit spolu s třetím blokem představuje 9 bitů konfiguračních dat. Pro každý zapisovaný registr je nutné opakovat celý tento proces.

Kodek na desce nemá osazen sluchátkový výstup. Využíváme linkový výstup a tudíž nelze řídit hlasitost. Celá komunikace se tak omezí na nastavení správných režimů po zapnutí a funkci mute. Vždy jde o zápisy a pokud rezignujeme na kontrolu potvrzovacích bitů, představuje celá komunikace jen nastavování vhodné sekvence bitů na dva výstupy.

Entita *i2c_write* převádí 16 bitů dat na výše ukázané průběhy a slouží tedy k jednomu kompletnímu zápisu do registru. Konkrétní obsah pro tyto registry, je uložen jako slova v paměti *config_rom*. Tyto entity jako použity jako komponenty uvnitř *i2c_config*, která je využívá k odesílání kompletního nastavení po spuštění, případně umlčení, dle svých vlastních vstupů.

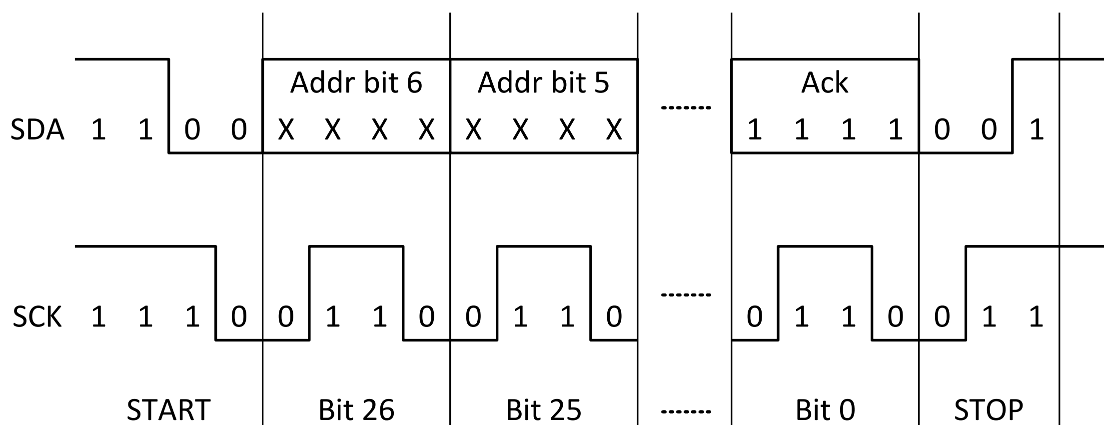
Entita *i2c_write* vysílá zadaná data na dvouvodičovou sběrnici. Zároveň vytváří i taktovací signál a podmínky pro začátek a ukončení komunikace.

```
entity i2c_write is
  generic (
     dev_address : std_logic_vector(6 downto 0) := "0011010");
  port (
     rst   : in  std_logic;
     clk   : in  std_logic;
     data  : in  std_logic_vector(15 downto 0);
     start : in  std_logic;
     busy  : out std_logic;
     sck   : out std_logic;
     sda   : out std_logic);
end entity i2c_write;
```

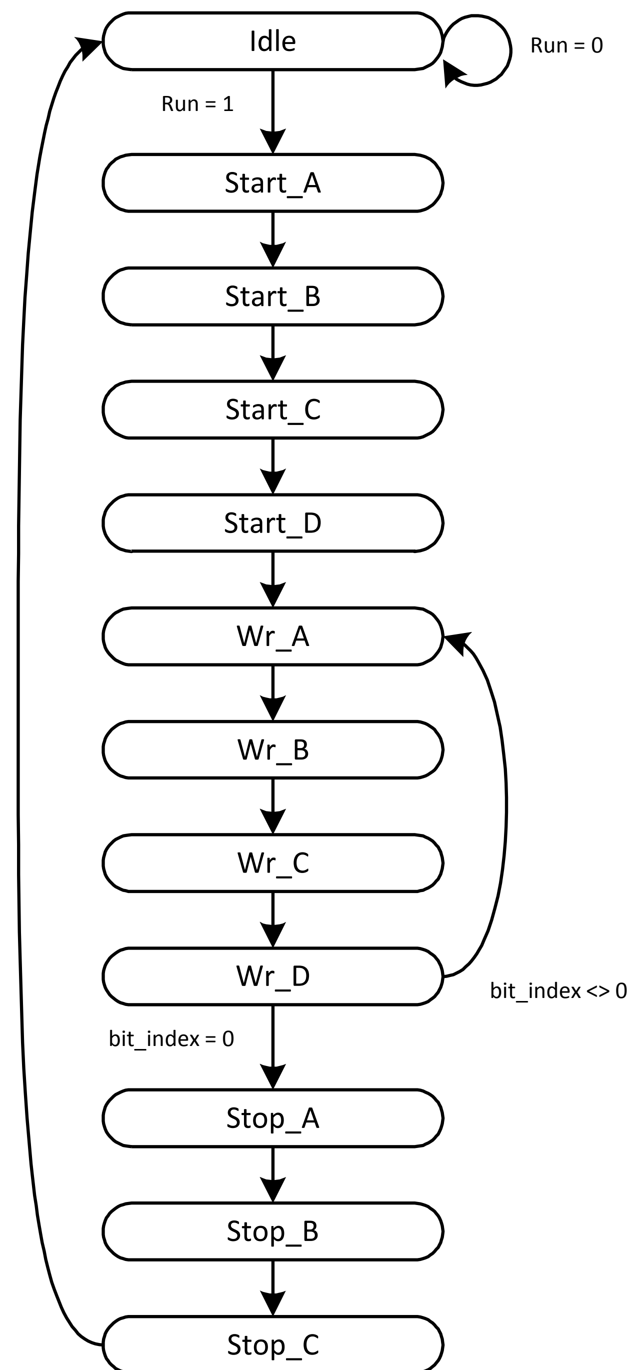
Adresa zařízení je zavedena jako generická konstanta, signál *data* pak již obsahuje jen dva osmibitové bloky s adresou registru a jeho obsahem. Vysílání začíná nastavením jedničky na vstupu *start*, zatímco probíhající vysílání je indikováno signálem *busy*. Výstupy *sda* a *sck* jsou data a hodiny dvouvodičové sběrnice.

Funkce je poměrně jednoduchá, na počátku se data spolu s adresou zařízení a pevnými hodnotami dalších bitů zapíšou do registru *data_reg*. Tento registr obsahuje 27 bitů, které je třeba odeslat na sběrnici. Stavový automat pak pracuje s frekvencí cca. 400 kHz, jež odpovídá čtyřnásobku přenosové rychlosti sběrnice. Nastavováním hodnot signálů *sck* a *sda* vytvoří žádoucí průběhy. Princip jejich tvorby je zřejmý z diagramu na obrázku 28.

Obr. 28 – Diagram komunikace s hodnotami sck a sda



Obr. 29 – Stavový diagram automatu i2c_write



Automat čeká ve stavu *idle*. Pokud je požadováno vysílání, začíná vytvářet startovací podmínku na sběrnici, zatímco zapisuje vstupní data do pracovního registru *data_reg*. Nastaví také signál *busy*. Čítač *bit_index* je nastaven na hodnotu 26, tedy číslo nejvyššího odesílaného bitu. Během stavů *Wr_A* až *Wr_D* je tento bit na výstupu. Čítač se dekrementuje a dokud nedosáhne nuly, tyto stavy se opakují. Po odvysílání všech bitů automat vytvoří pomocí stavů *Stop_A* až *Stop_C* ukončovací podmínku, vrací se do *idle* a signál *busy* se nuluje.

Entita *config_rom* je jednoduchá asynchronní rom s šestnáctibitovými slovy. Každé slovo je adresa a obsah jednoho registru, jsou to vstupní data pro entitu *i2c_write*.

```
entity config_rom is
  port (
    data : out std_logic_vector(15 downto 0);
    addr : in natural range 0 to 15);
end entity config_rom;
```

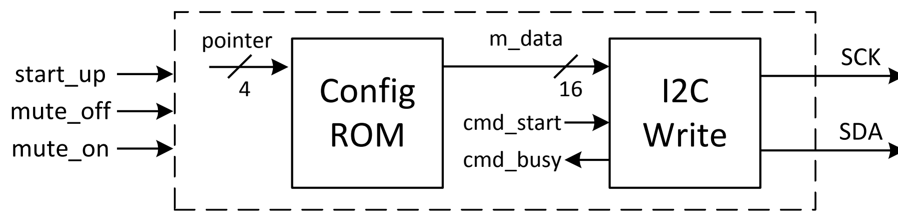
Postupným nastavením *addr* na hodnoty 0 až 10 dochází ke čtení kompletního nastavení kodeku. Na adrese 11 a 12 jsou hodnoty pro registr *AA_PATH_CONTROL* které zapojují a odpojují D/A převodník. Tím je realizována funkce mute na úrovni kodeku. Zbylé adresy 13 až 15 nejsou využity. Zdrojový kód této entity v komentářích odkazuje na konkrétní strany datasheetu [3], kde jsou jednotlivá nastavení dokumentována.

Obě doposud popsané entity jsou využity jako komponenty pro *i2c_config*. Tato entita je propojuje a ovládá tak, aby na základě svých prioritních vstupů odvysílala kodeku odpovídající nastavení.

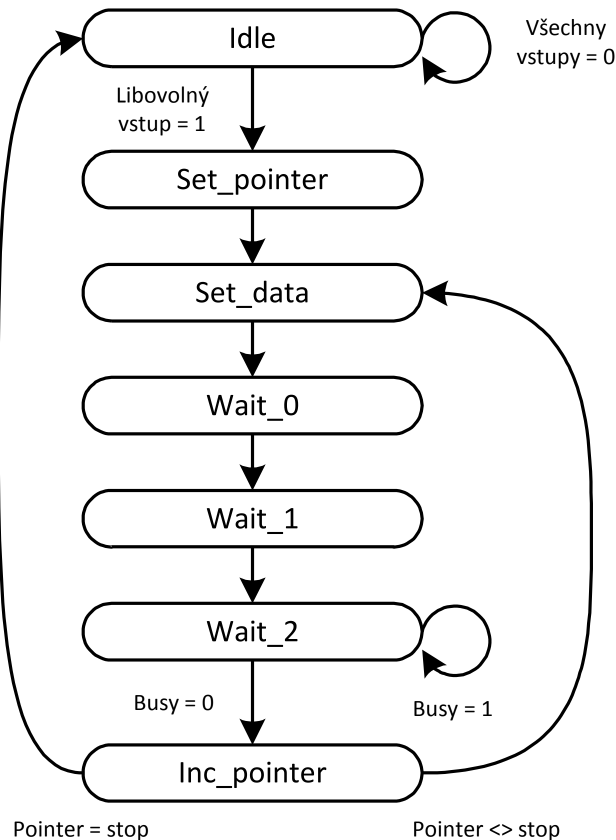
```
entity i2c_config is
  port (
     rst       : in std_logic;
     clk       : in std_logic;
     start_up  : in std_logic;
     mute_off  : in std_logic;
     mute_on   : in std_logic;
     sck       : out std_logic;
     sda       : out std_logic);
end entity i2c_config;
```

Nastavení jedničky na vstupu *start_up* spouští kompletní konfiguraci kodeku. Tento vstup má nejvyšší prioritu a jeho funkce bude vykonána, i kdyby zároveň došlo k nastavení *mute_on* nebo *mute_off*. Signálem *mute_on* se spustí zápis nastavení, při němž je D/A převodník odpojen. Tím dojde k vypnutí zvuku, přičemž tento signál má střední prioritu. Nejnižší prioritu má *mute_off*, který zvuk zapíná opětovným zapojením převodníku. Signály *sck* a *sda* jsou přímo napojeny na komponentu *i2c_write* a tato entita je dále neupravuje. Propojení v entitě ukazuje schéma na obrázku 30 na další stránce.

Obr. 30 – Zjednodušené schéma architektury i2c_config



Obr. 31 – Stavový diagram automatu i2c_config



Automat čeká ve stavu *idle* na aktivitu libovolného vstupu. Poté přechází do *Set_pointer*, kdy nastavuje aktuální a koncovou adresu pro čtení z paměti. Tyto hodnoty jsou nastaveny podle priorit a aktivity vstupů.

Ve stavu *Set_data* je z paměti vybaveno slovo s konfigurací a následující stav spouští jeho vysílání na sběrnici. *Wait_1* dává komponentě *i2c_write* čas k nastavení signálu *busy*, který indikuje probíhající vysílání a automat po tuto dobu čeká ve stavu *Wait_2*. Jakmile je odvysílání slova dokončeno, automat zvýší o jedničku adresu do paměti a pokud není dosaženo koncové hodnoty, vrací se do *Set_data* a vysílá další konfigurační slovo. V opačném případě se vrací do klidového stavu *idle*.

5.12 Další pomocné entity

Pro ovládání, vizualizaci a další činnosti je v projektu několik dalších entit. Jejich funkce je jednoduchá, takže budou popsány v rámci společné kapitoly.

1) Taktování zvukového kodeku

Hodinový signál pro zvukový kodek by měl mít volitelně frekvenci 12.288 MHz nebo 18.432 MHz. Jelikož projekt je taktován z 50 MHz oscilátoru, je zapotřebí vytvořit některou ze zmíněných frekvencí pomocí fázového závěsu. Kmitočet jsem volil tak, aby byl vyšší, nicméně s co nejmenší chybou.

Entita *pll* je vytořena pomocí megafunkce *altpll* a pracuje jako dělička kmitočtu s poměrem 20 / 81. Vytváří hodinový signál s frekvencí 12.346 MHz, což představuje chybu +0.47% oproti ideální hodnotě.

2) Zobrazování na sedmisegmentovém displeji

Entita *segment* slouží k převodu čtyřbitového binárního čísla *data* na kód *seg*, který toto číslo zobrazí jako hexadecimální číslici na sedmisegmentovém displeji.

```
entity segment is
  port (
     data : in std_logic_vector (3 downto 0);
     seg  : out std_logic_vector(6 downto 0));
end entity segment;
```

Sedmisegmentové displeje v projektu ukazují vybrané číslo filtru, nastavený zisk a stav čítače saturovaných vzorků.

3) Dělička kmitočtu

Freq_div čítá náběžné hrany vstupního signálu a po dosažení nastavené hodnoty mění polaritu výstupu. Dělí tedy kmitočet dvojnásobkem nastaveného parametru *ratio*. Napojením výstupu na LED umožňuje vizuálně sledovat aktivitu signálů s vysokou frekvencí změn.

4) Ošetření spínačových vstupů

Entita *one_pulse* vytvoří po náběhu vstupního signálu jeden takt široký puls na výstupu. Na další změny vstupu pak zadanou dobu nereaguje.

5) Výběr koeficientů filtru z paměti

Filtr načítá koeficienty z paměti sám, ale je třeba mu zadat počáteční adresu. Vzhledem k organizaci paměti koeficientů je entita b*and_to_addr* pouhá násobička devíti.

```
entity band_to_addr is
  port (
    band_no   : in  std_logic_vector(5 downto 0);
    base_addr : out std_logic_vector(15 downto 0));
end entity band_to_addr;
```

Vstupní signál *band_no* je šestibitové číslo filtru, výstup *base_addr* je devítinásobek této hodnoty. Výstup je rozšířen na šestnáct bitů jen jako rezerva, ve skutečnosti není jeho rozsah využit. Násobení devíti je ve vnitřně provedeno jako 9x = x + 8x, protože takto lze operaci násobení převést na sčítání s bitovým posunem a ušetřit prostředky v FPGA.

6) Nastavení zisku

Zisk je vstupní parametr filtru ve formátu double float. Z důvodu snadného zadávání jsou vybrané hodnoty tabelovány v entitě *db_to_gain*.

```
entity db_to_gain is
  port (
    db   : in  std_logic_vector(5 downto 0);
    gain : out std_logic_vector(63 downto 0));
end entity db_to_gain;
```

Vstup *db* je šestibitový celočíslný kód, vhodný k zadání na spínačích vývojové desky. Výstupní signál *gain* je odpovídající hodnota zisku ve formátu double float.

Nulová vstupní hodnota má speciální význam – nulový výstup a tedy úplné umlčení na úrovni filtru. Hodnoty 1 až 63 pak odpovídají zisku -23 dB až +40 dB s krokem 1 dB.

Všechny hodnoty zároveň obsahují „skrytý“ zisk o hodnotě 2.4, který nastavuje vhodnou úroveň šumu s ohledem na dynamický rozsah převodníku a statistické rozdělení. Určení této hodnoty je uvedeno na konci kapitoly 5.4.

5.13 Ovládání

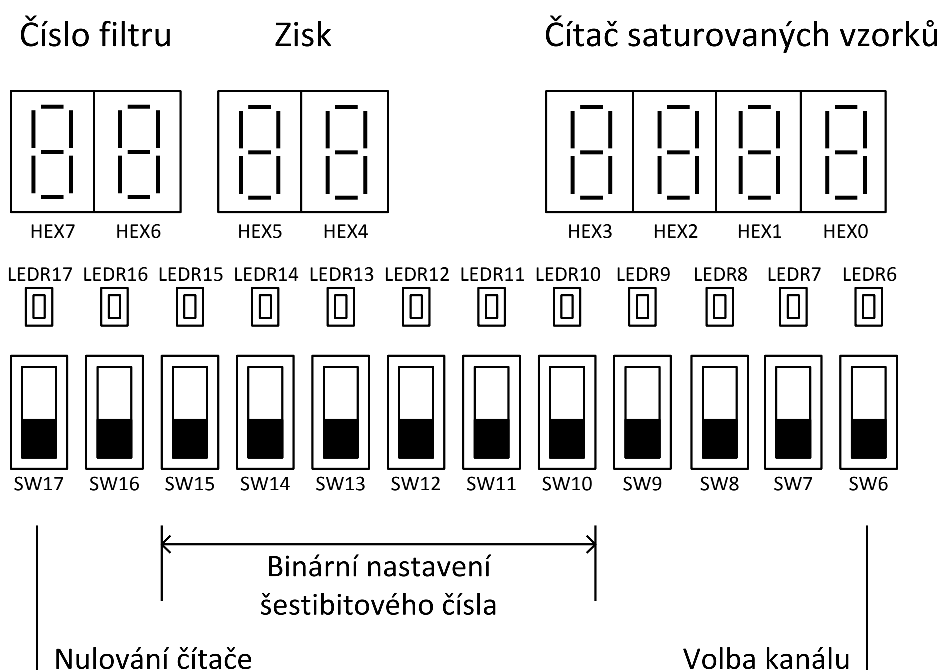
Na obrázcích 32 a 33 na další stránce je zobrazeno ovládací rozhraní vývojové desky. Jsou využity spínače, tlačítka, LED a sedmisegmentové displeje.

Displeje zobrazují právě nastavené číslo filtru a zisk pro jeden kanál. O který kanál jde, určuje stav přepínače SW6. Také čítač saturovaných vzorků se vztahuje k aktuálně zvolenému kanálu a ukazuje počet vzorků, u nichž se vyskytla hodnota 0x000000 nebo 0xFFFFFF. Čítač se dá nulovat krátkým přepnutím SW17.

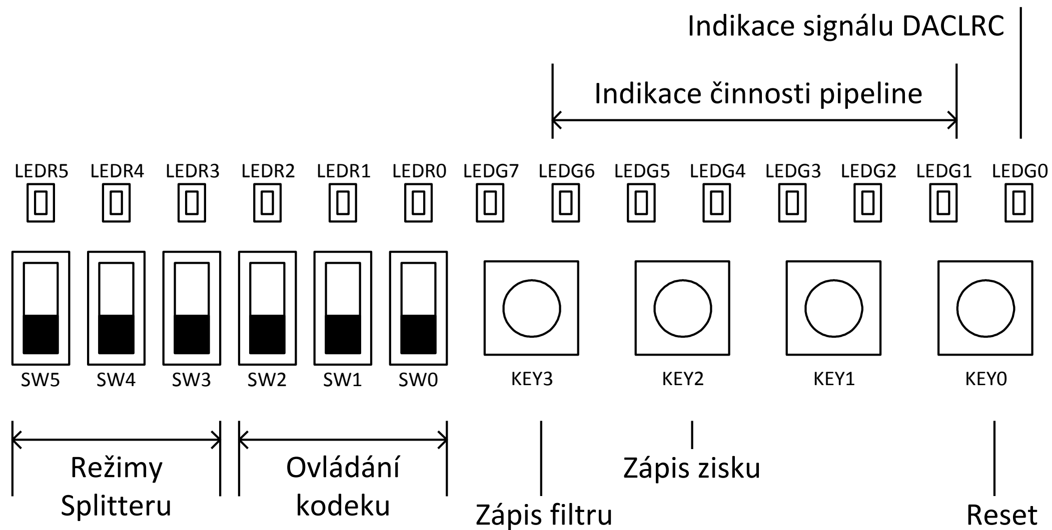
Hodnoty určující číslo filtru a nastavený zisk jsou šestibitové a lze je zadat pomocí přepínačů SW10 až SW15. Manipulace s těmito přepínači ještě nic nemění, hodnotu je třeba potvrdit. Právě potvrzením pomocí tlačítek KEY2 nebo KEY3 dojde k zápisu zvolené hodnoty, a to buď jako hodnoty zisku, nebo čísla filtru. Opět je vše prováděno pro akuálně zvolený kanál dle stavu SW6.

Funkční přepínače rozsvěcí ještě pro lepší přehlednost červené LED nad sebou. Zatímco přepínače SW7, SW8, SW9 a SW16 nemají žádnou funkci a jim odpovídající LED na ně nereagují.

Obr. 32 – Ovládací rozhraní projektu na vývojové desce (1/2)



Obr. 33 – Ovládací rozhraní projektu na vývojové desce (2/2)



V pravé polovině se nacházejí již zmíněná tlačítka KEY3 a KEY2. Tlačítko KEY1 sice rozsvěcí zelenou LEDG7 nemá však za současného stavu přiřazenu žádnou užitečnou funkci. KEY0 funguje jako reset celého zařízení.

Přepínače SW3 až SW5 nastavují režim splitteru. SW5 odpovídá signálu LR_Split a určuje, zda kanály dostávají shodná data, nebo ne. SW3 a SW4 pak aktivují či deaktivují jednotlivé kanály (levý a pravý v tomto pořadí). Přesný význam kombinací stavů je popsán v kapitole 5.6.

Signály ovládající konfiguraci kodeku jsou přiřazeny třem posledním přepínačům. Prvotní nastavení kodeku po zapnutí desky je třeba provést ručně krátkým přepnutím SW0. Přepínače SW1 a SW2 pak umožňují kodek umlčet, nebo opět zapnout zvuk. Tyto přepínače odpovídají signálům *start_up, mute_on* a *mute_off*, tak jak jsou popsány v kapitole 5.12.

Zelené diody jsou využity k vizuální kontrole činnosti celého systému. LEDG0 zobrazuje signál DACLRC od kodeku vydělený 96000 krát. Při vzorkovací frekvenci 96kHz tedy bliká jednou za sekundu.

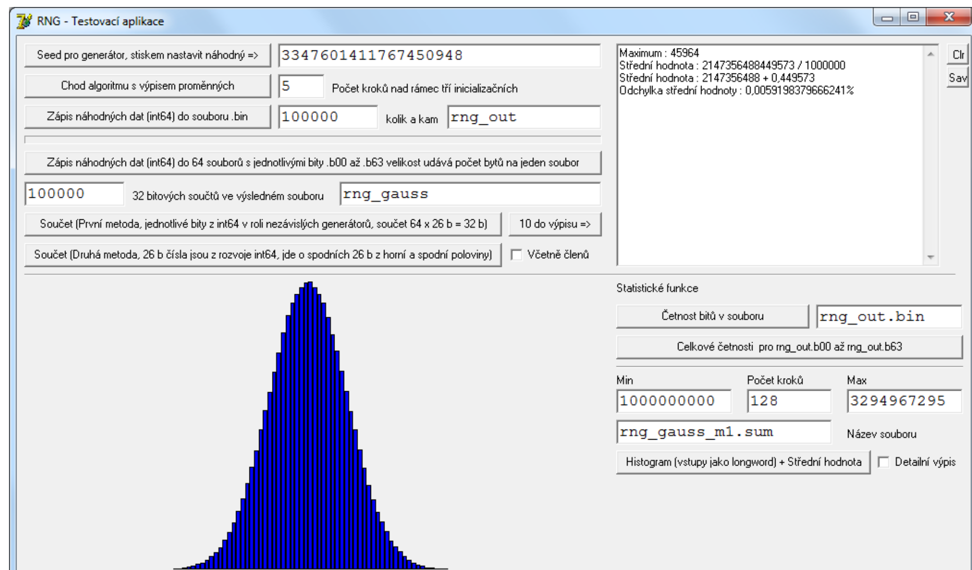
Ostatní diody LEDG1 až LEDG6 jsou vhodně vydělené signály *accept* z celé pipeline od generátoru čísel až po zápis do front. Umožnily hlavně při vývoji sledovat změny v rychlosti generátoru čísel v závislosti na režimu splitteru.

6 Test generátoru pseudonáhodných čísel

Pro předběžné testování při vývoji jsem napsal aplikaci, která mi umožnila ověřit činnost některých algoritmů před jejich implementací ve VHDL. Dále jsem také mohl porovnávat výsledky této aplikace se simulacemi při vývoji a včas ověřit, zda navržený postup neobsahuje zjevné chyby.

Aplikace umožňuje generovat a ukládat stejné výstupy, jako PRNG v mém projektu. Je schopná spočítat některé jednoduché statistiky a vykreslit histogram. Neobsahuje však žádné exaktní metody pro posouzení náhodnosti výstupu, zejména normality rozdělení amplitud. Pro další testy však může posloužit jako generátor datových vzorků.

Obr. 34 – Aplikace pro předběžné testy a vývoj

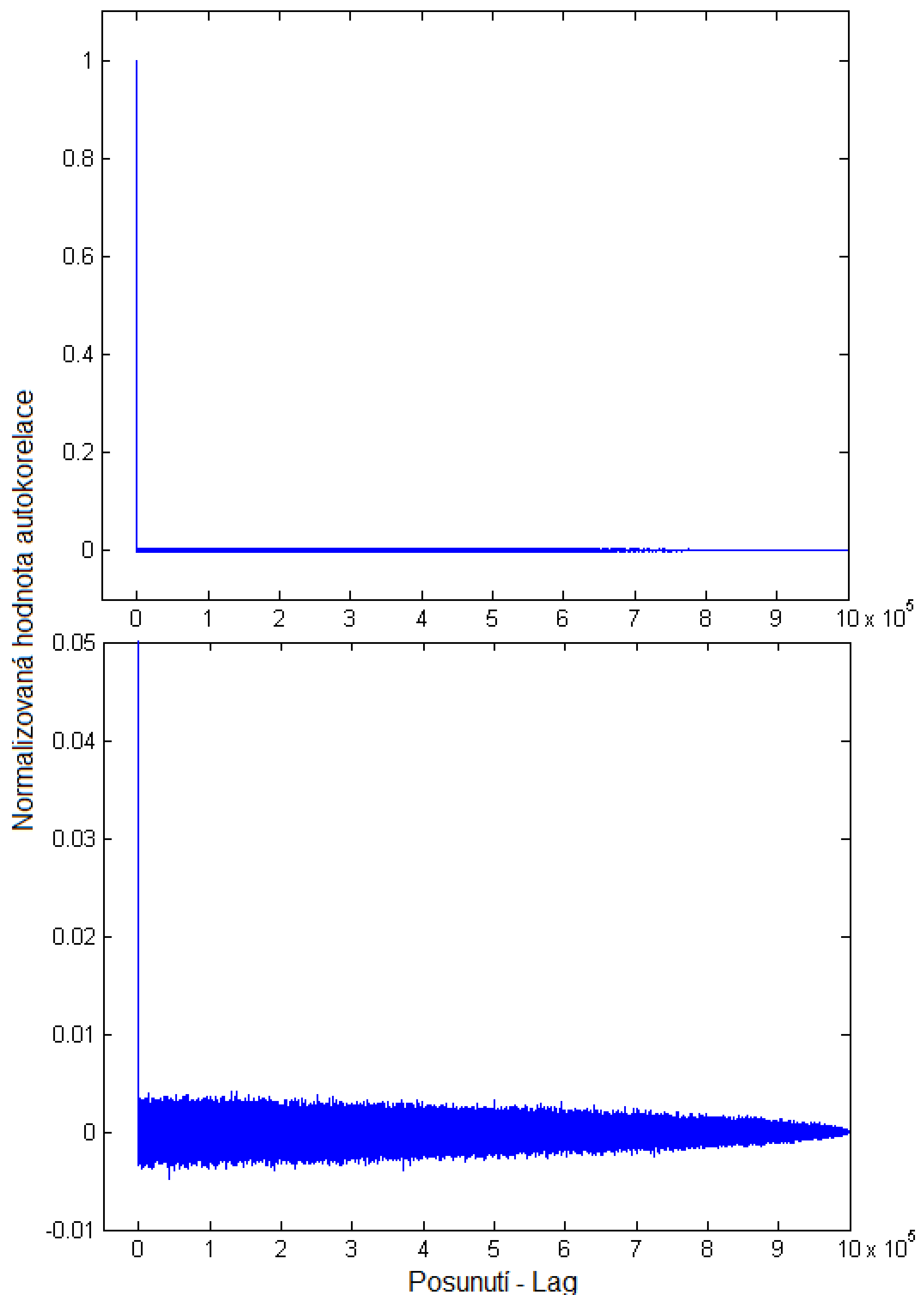


Pro další postup jsem použil tuto aplikaci k vygenerování vzorku 12 800 000 čísel v textovém formátu CSV. Tyto data jsem importoval do výpočetního prostředí Matlab a podrobil několika testům. Ověřoval jsem průběh autokorelační funkce, histogramu a CDF (Kumulativní distribuční funkce). Srovnával jsem očekávané parametry rozdělení dle kapitoly 5.4 s hodnotami vypočtenými ze vzorku dat. Dále jsem provedl několik statistických testů pro ověření normality rozdělení. Jmenovitě to byly : Chí – kvadrát test dobré shody, Lilliefors, Kolmogorov – Smirnov, a Jacque – Bera test.

A) Test periodicity – Autokorelační funkce

Grafy na obrázku 35 ukazují hodnotu autokorelace vzorku o délce jeden milion prvků. Hodnota je normalizovaná a očištěna od vlivu střední hodnoty. Průběh nevykazuje žádnou skrytou periodu, která by se projevila lokálními maximy funkce při nenulovém posunutí.

Obr. 35 – Graf autokorelační funkce



B) Parametry statistického rozdělení

Histogram na obrázku 36 ukazuje četnost hodnot ve vzorku o délce deset milionů čísel. Tvar odpovídá normálnímu rozdělení, stejně jako v případě CDF na obrázku 37 na další stránce. Vzhled těchto funkcí se vizuálně těžko ohodnotí, proto je zde také uveden přehlednější P-P graf a číselné parametry. Pro data s normálním rozdělením by měl být P-P graf lineární.

Z výše zmíněného vzorku byly vypočteny tyto parametry :

- Střední hodnota μ = 2 147 444 600

  To představuje oproti teoretické hodnotě 2 147 483 616 rozdíl +0.0018%

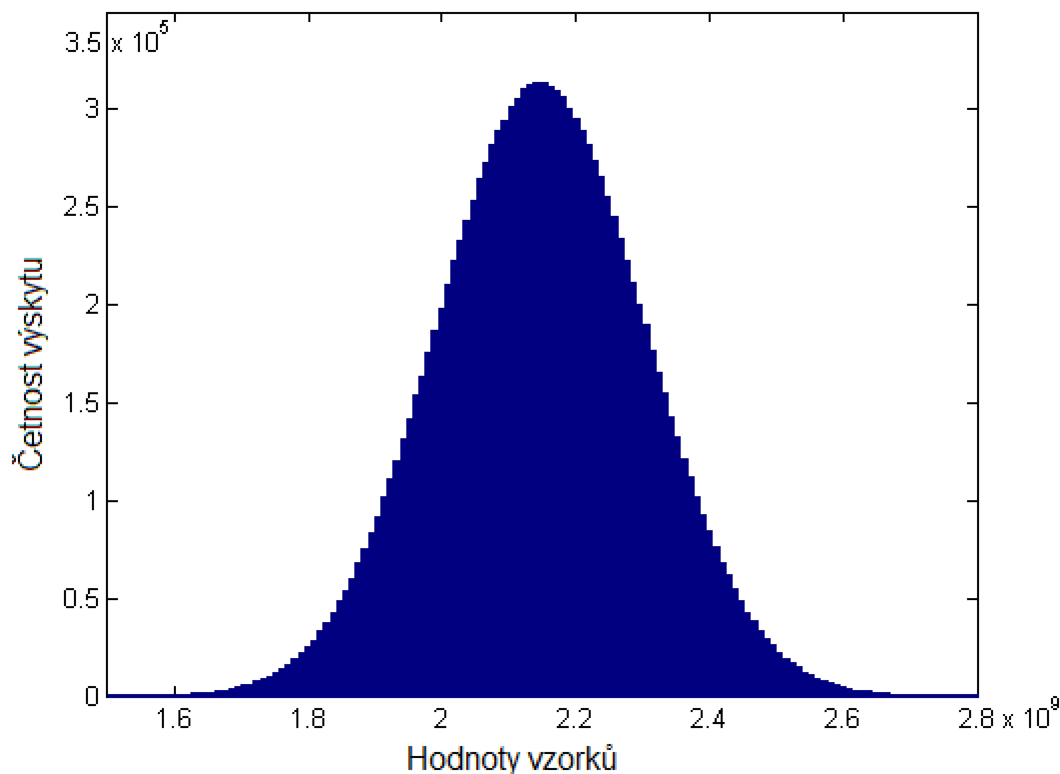
- Standartní odchylka σ = 155 061 462

  Rozdíl oproti teoretické hodnotě 154 981 280 činí -0.052%

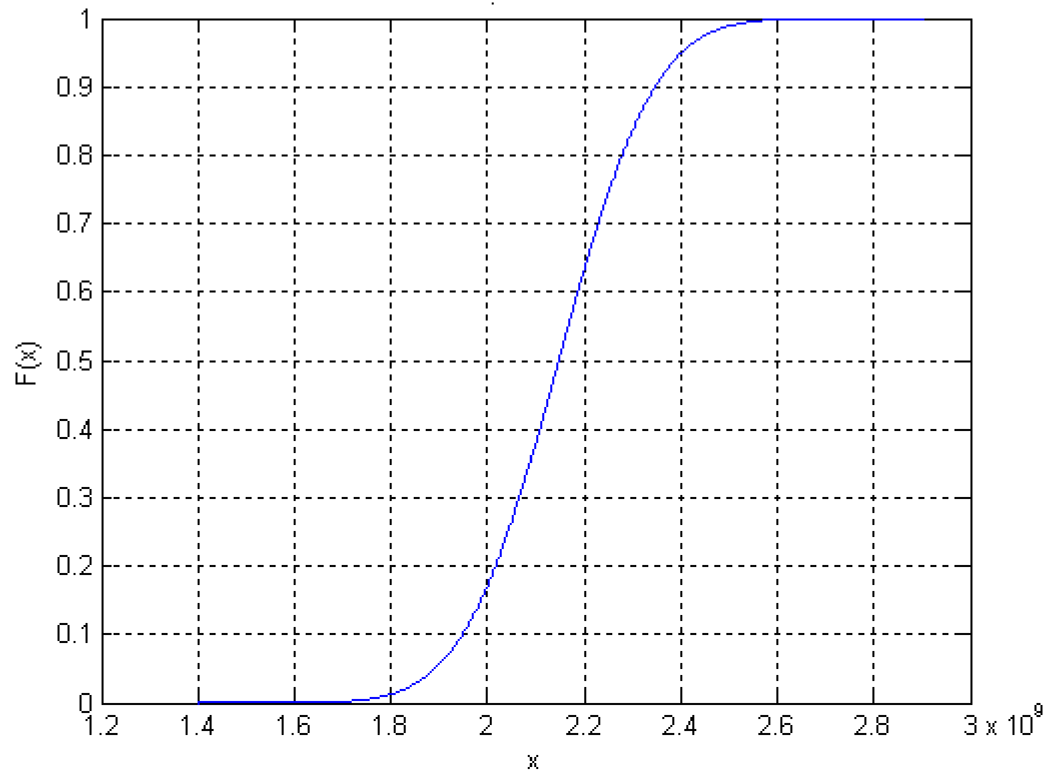
- Symetrii rozdělení lze posoudit z rozdílu střední hodnoty a mediánu.

  Rozdíl činí 86 000, medián je tedy 0.004% vyšší, než střední hodnota.

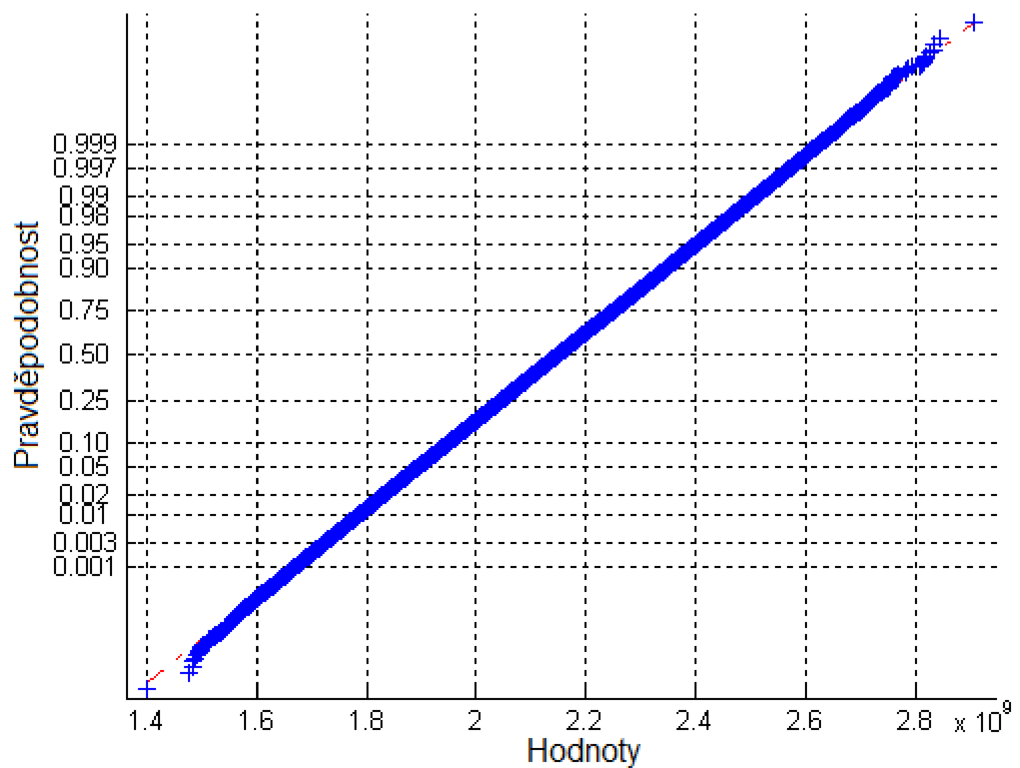
Obr. 36 – Histogram četnosti výskytu hodnot



Obr. 37 – Kumulativní distribuční funkce zjištěná ze vzorku dat



Obr. 38 – Pravděpodobnostní P–P graf vzhledem k normálnímu rozdělení



C) Testy normality

Pro účely testování jsem rozdělil vzorek dat na 100 000 sekvencí o délce 128 prvků. Pro každou tuto sekvenci pak proběhly testy na obvyklých hladinách významnosti.

K-S test porovnává dvě CDF a hodnotí nejvyšší nalezenou odchylku mezi nimi. V tomto provedení se srovnává CDF stanovená z testovaných dat a teoretická CDF vypočtená pro normální rozdělení dle parametrů z kapitoly 5.4. Nejvyšší odchylka pak vzhledem k uvažované hladině významnosti vede k případnému odmítnutí normality vzorku.

Hladina významnosti 0.05 znamená, že ze vzorků ideálních vlastností bude 5% neoprávněně zamítnuto. Zkoumané sekvence by tedy v ideálním případě měly vykazovat pětiprocentní neúspěšnost v testu.

V průběhu K-S testu bylo odmítnuto 5080 sekvencí, tedy o 0.08% více, než by odpovídalo ideálnímu stavu. Opakovaný test nad stejnými daty při hladině významnosti 0.01 zamítl 1064 sekvencí.

Lilliefors je obdobou K-S testu, ale nevyžaduje explicitně zadanou CDF. Tento test předpokládá jako referenci CDF normálního rozdělení, jejíž parametry se snaží odhadnout z testovaných dat.

Dalším testem, který nevyžaduje dodatečné informace o parametrech rozdělení je Jarque-Bera. Tento test posuzuje rozdělení z hlediska koeficientů šikmosti a špičatosti. Oba tyto koeficienty jsou pro normální rozdělení nulové.

Výrazně obecnější je χ^2 test dobré shody, který rozděluje data do tříd. Následně porovnává zjištěné četnosti v třídách s teoreticky předpokládanými. V tomto testu je možné ručně rozvrhnout hranice jednotlivých tříd a specifikovat parametry rozdělení. Pro výpočet však byla použita automatická varianta odhadující parametry rozdělení z testovaných dat.

Výsledky dosažené nad vzorkem dat pro jednotlivé testy shrnuje následující tabulka. Stojí za povšimnutí, že jediný test, který zná ideální rozdělení testovaných dat (K-S), vykazuje lepší výsledky, než ostatní testy, které tyto parametry pouze odhadují.

Tab. 2 – Výsledky testů normality

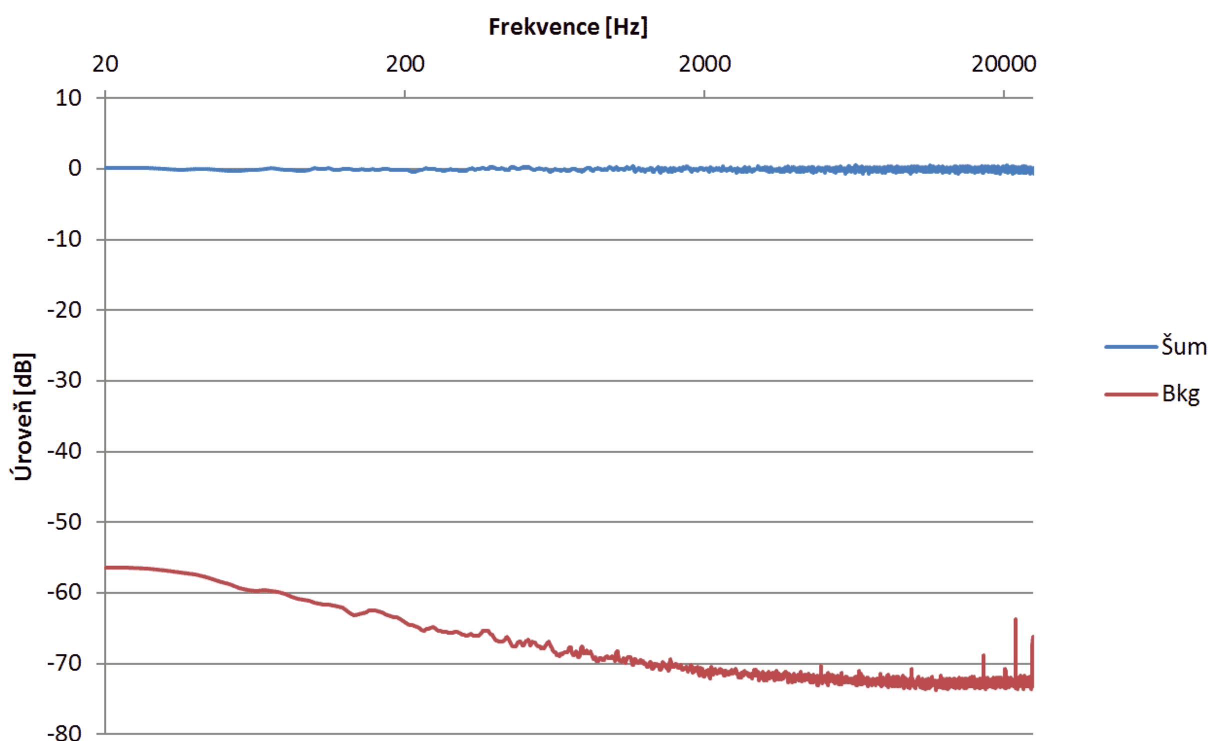
Test	Odmítnuto při $\alpha = 0.05$	Odmítnuto při $\alpha = 0.01$
χ^2	5470 (+0.470%)	1025 (+0.025%)
Kolmogorov – Smirov	5080 (+0.080%)	1064 (+0.064%)
Lilliefors	4919 (-0.081%)	991 (-0.090%)
Jacque – Bera	4524 (-0.476%)	817 (-0.183%)

7 Měření výstupního šumu

Pro zhodnocení zvukového výstupu jsem provedl laboratorní měření spektra generovaného šumu. Měřil jsem šum bez omezení pásma, pozadí při zablokovaném kanálu a šum se zařazenými filtry. Pro měření filtrů jsem vybral, vzhledem k šířce pásma analyzátoru, všechna oktávová pásma až do centrální frekvence 8 kHz a několik vybraných třetinoktávových.

Zisk v generátoru byl vždy nastaven tak, aby v nejširším z měřených pásem nedocházelo k limitaci více, než několika vzorků za sekundu. Všechna měření v sérii pak jsou provedena při tomto zisku, což ulehčuje jejich srovnání. Jelikož samotná úroveň není důležitá, pro názornost je v grafech normalizována k 0 dB.

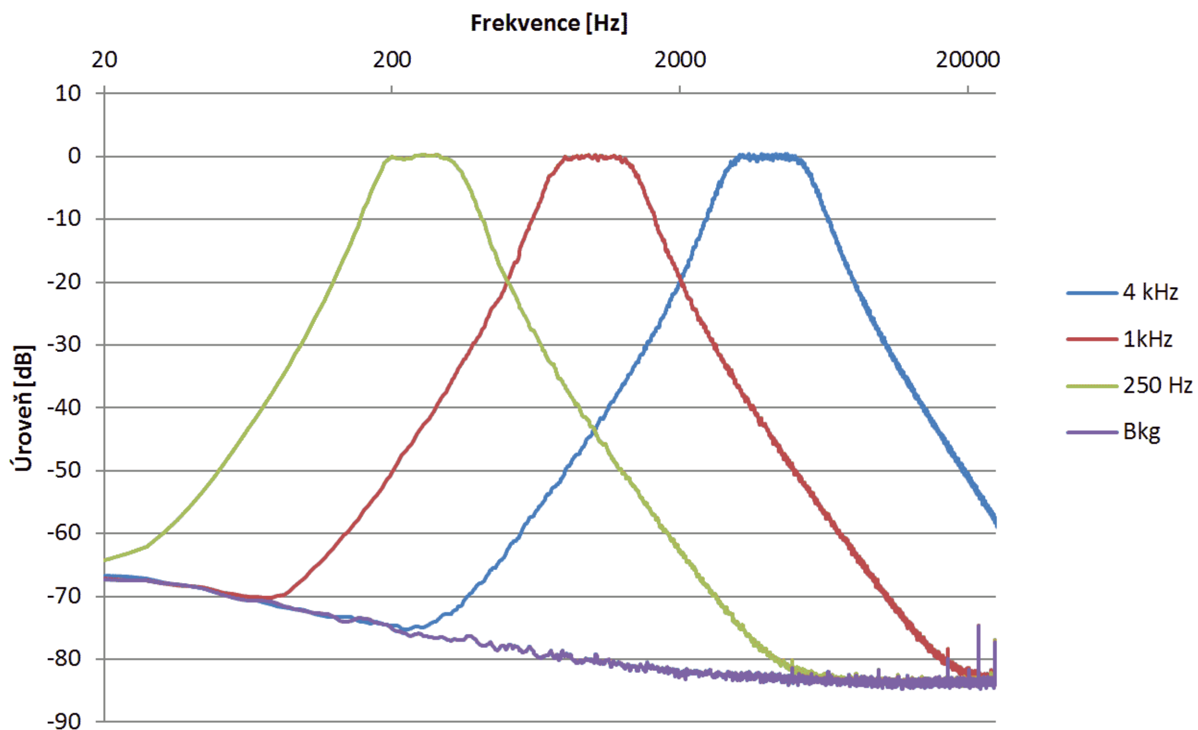
Obr. 39 – PSD šumu bez omezení pásmovou propustí



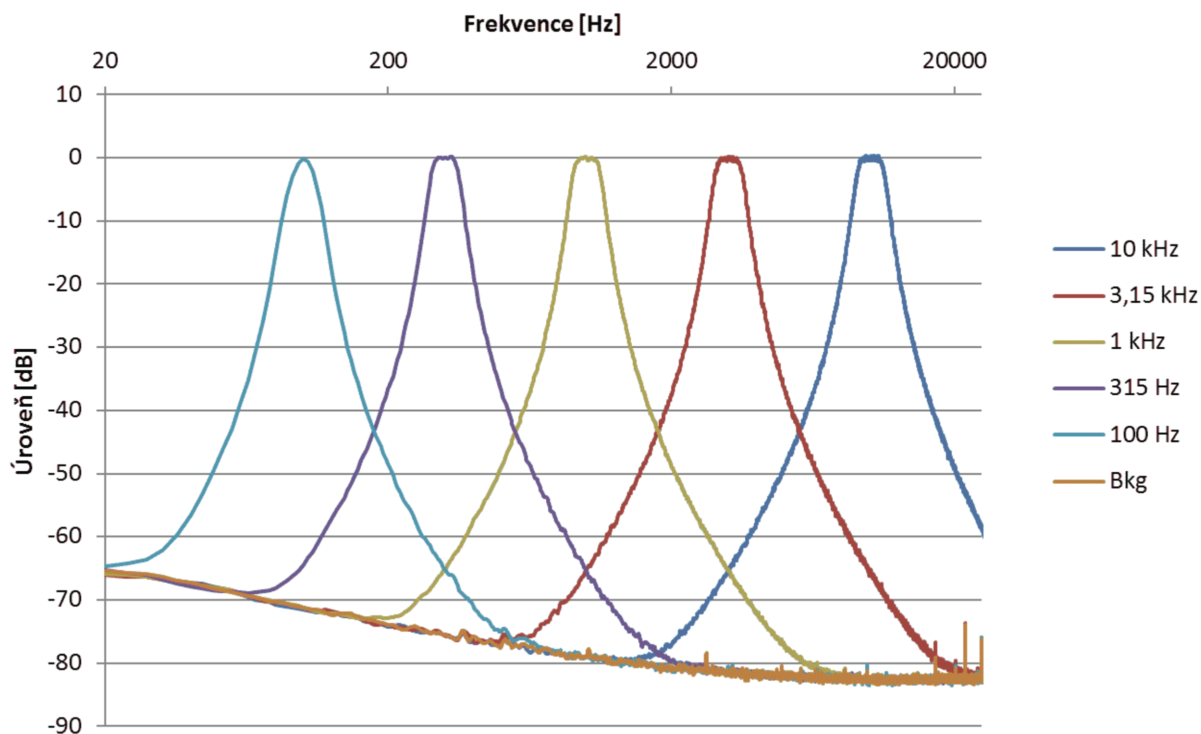
Obrázek 39 ukazuje výsledky měření generovaného šumu bez omezení pásmovou propustí. Spektrum je ploché, nevykazuje žádnou zjevnou systematickou odchylku. Rozptyl výsledků analýzy s rostoucím počtem průměrování dále klesá a v grafu představuje ±0.7 dB

Následující grafy na obrázcích jsou měření včetně pásmových propustí. Pro přehlednost nejsou zobrazena všechna změřená pásma, ale jen několik vybraných. Vždy je také doplněno měření pozadí při zablokovaném generátoru.

Obr. 40 – PSD šumů ve vybraných oktávových pásmech



Obr. 41 – PSD šumů ve vybraných třetinoktávových pásmech



Závěr

V průběhu práce na tomto projektu se mi podařilo vytvořit funkční šumový generátor. Výsledný systém má všechny původně žádané vlastnosti a podařilo se i dosáhnout minimálního počtu součástek. Mimo nezbytného kodeku je celý funkční celek šumového generátoru v FPGA aniž by bylo nutno využívat další součásti, zejména paměť.

Měření výstupního šumu neukazuje na zjevné chyby. Žádnou systematickou deformaci ve spektru se mi nepodařilo najít. Je však pravda, že by lepším nastavením vstupní úrovně analyzátoru bylo možné docílit při měření většího odstupu signálu od pozadí.

Při testování generátoru náhodných čísel jsem měl k dispozici výsledky testů pro data získaná pomocí algoritmů Mersenne twister a Ziggurat. Použil jsem stejnou metodiku, což je postup uvedený v šesté kapitole. Dosažené výsledky jsou většinou poněkud horší, počet zamítnutých vzorků ze sto tisíc je obvykle o několik desítek vyšší.

Během dokumentace a testů vyvstalo několik nedořešených otázek a na základě zkušeností z vývoje mohu navrhnout do budoucna několik vylepšení.

Zřetězení jednotlivých částí systému do pipeline je nezbytné a v současné podobě funkční. Přesto nyní prodlužuje zpracování dat v každém bloku o několik taktů. Ve většině případů jde o zanedbatelný vliv, neboť mnoho bloků má na provedení své činnosti řádově stovky taktů. Výjimkou je úvodní blok pipeline s generátorem čísel. V tomto bloku signalizace a předávání dat následníkovi násobně převažuje nad vlastní funkcí. Generátor sám je schopen produkovat nový výsledek každý takt. Bylo by proto vhodné sloučit tuto funkci s následujícím blokem, nebo zajistit jejich těsnější propojení. Současné řešení jsem ponechal v platnosti, protože i přes svoji neefektivitu je stále dostatečně výkonné.

Převod čísel z rovnoměrného rozdělení na normální je i v tak jednoduchém provedení zjevně použitelný, přesto neobstojí ve srovnání se sofistikovanějšími metodami. Na horších výsledcích v testech normality se patrně projevuje nedostatečná nezávislost členů odvozených z jednotlivých bitů náhodného generátoru. Na vině může být i omezený počet sčítanců. Řešením by mohlo být zachovat principu součtu, ale využít pro vstupní sčítance více generátorů s různými algoritmy, nebo jejich variantami. Druhá možnost je nahradit prosté sčítání jiným typem převodu, například algoritmem Ziggurat, nebo Box-Muller transformací.

Komponenta splitter pro napojení dvou kanálů na jeden generátor je podle mého názoru zbytečně složitá. Je to způsobeno tím, že se snaží nečíst z generátoru výsledky které by pak nebyly dále využity. S odstupem soudím, že požadavek na takové úsporné chování tuto komponentu značně zkomplikoval, aniž by měl nějaký reálný přínos.

Číslicový filtr pracuje podle posloupnosti kroků uložených jako slova v paměti. Tyto kroky jsou sestavené triviálně podle postupu výpočtu. Považuji za pravděpodobné, že se v postupu vyskytují operace dělené do více kroků, které by ve skutečnosti mohly proběhnout současně. Zde je pravděpodobně prostor pro optimalizaci.

Další potenciální vylepšení filtru by bylo jeho nahrazení kaskádou z horní a dolní propusti namísto specializace na pásmovou propust. Umožnilo by to obecnější nastavení než současná předdefinovaná pásma. Takové řešení by vyžadovalo více násobení, tudíž by bylo nutné použít FPGA s více násobičkami.

Pokud by se další verze tohoto generátoru vyvíjela ve větším FPGA bez stanoveného omezení rozsahu, bylo by patrně nejrozumnější opustit koncepci se splitterem a vybavit každý kanál vlastním generátorem čísel.

V současné podobě je pro syntézu projektu použito 15 300 logických elementů, tedy 46% celkového počtu. V těchto elementech je realizováno 12 300 logických funkcí a 8 000 registrů. Z integrované paměti je využito 69 kb, což je 14% celkové kapacity. Jelikož obsah paměti *reg_filter_irom* je syntetizován v logických elementech, bylo by možné úpravou docílit úspory části logických elementů na úkor poměrně nevyužité paměti.

Z celkového počtu 70 devítibitových násobiček je využito 64, zde syntezátor dokázal optimalizovat a oproti mému předpokladu čtyři ušetřit. Je zjevné, že další rozšíření by již muselo přinést lepší využití stávajících násobiček, nebo přesunout vývoj na větší FPGA.

Fázový závěs je použit jeden ze čtyř a prostředky pro propojení logických elementů jsou využity z 28%, zde se na omezení nenaráží.

Seznam literatury a informačních zdrojů

[1] PRESS, William H., TEUKOLSKY, Saul A., WETTERLING, William T., FLANNERY, Brian P. *Numerical Recipes – The Art of Sceintific Computing*. 3th ed. Cambridge university press, 2007. 1235 s. ISBN 0-511-33555-5.

[2] MARCINI, Ron. *Op amps for everyone – Design reference*. Texas Instruments Incorporated. Poslední změna 27.8.2002, 464 s. Dostupné online : http://focus.ti.com/lit/an/slod006b/slod006b.pdf

[3] Wolfson Microelectronics. *WM8731 Low power stereo codec*. Rev. 4.8. Poslední změna 14.4.2009. 64 s. Dostupné online : http://www.wolfsonmicro.com/products/codecs/WM8731/

[4] SMITH, Steven W. *The Scientist and Engeneer's Guide to Digital Signal Processing*. Califonia Technical Publishing, 2011. Dostupné online : http://www.dspguide.com/pdfbook.htm

[5] Altera Corporation. *Floating-Point megafunctions user guide*. Poslední změna 8.11.2011. 122 s. Dostupné online : http://www.altera.com/literature/ug/ug_altfp_mfug.pdf

[6] Altera Corporation. *Quartus II version 11.1 Handbook – Recommended Design Practices.* Poslední změna 24.10.2011. 34 s. Dostupné online : http://www.altera.com/literature/hb/qts/qts_qii51006.pdf

[7] The MathWorks, Inc. *Matlab R2012a Product documentation*. Cit. 22.4.2012. Dostupné online : http://www.mathworks.com/help/index.html

[8] COLLIER, Richard J., SKINNER, Douglas A. *Microwave Measurements, 3rd Edition.* The Institution of Engineering and Technology, London, 2007. 484 s. ISBN 978-0-86341-735-1

Přílohy

Příloha A – mif_creator.m

Skript systému Matlab pro návrh koeficientů pásmových propustí a jejich zpracování do podoby inicializačního souboru paměti *.mif.

```
MifDepth = 1024; % Počet slov paměti
Central = 1000;  % Střední frekvence
FileName = 'rom_1024x64.mif';

% Výpočet hraničních frekvencí 1/3 oktávových pásem
n = -16 : 17;
f192_3 = zeros(33,2);
for i=1:33
    f192_3(i,1) = Central * 2^(n(i)/3) / 2^(1/6);
    f192_3(i,2) = Central * 2^(n(i)/3) * 2^(1/6);
end

% Výpočet hraničních frekvencí oktávových pásem
n = -5 : 5;
f192_1 = zeros(11,2);
for i=1:11
    f192_1(i,1) = 1000 * 2^n(i) / 2^(1/2);
    f192_1(i,2) = 1000 * 2^n(i) * 2^(1/2);
end

% Návrh pásmových propustí - 1oct 192kHz
fs = 192000;
bands = size(f192_1,1);
c_192_1 = zeros(9 * bands, 1);
for b = 1 : bands
    f = fdesign.bandpass('n,f3dB1,f3dB2',6,f192_1(b,1),f192_1(b,2),fs);
    h = design(f, 'butter');

    % Pořadí koeficientů : scale1 a11 a21 scale2 a12 a22 scale3 a13 a23
    c_192_1(9*b - 8: 9*b, 1) = [h.ScaleValues(1); h.sosMatrix(1,5:6)';
                                h.ScaleValues(2); h.sosMatrix(2,5:6)';
                                h.ScaleValues(3); h.sosMatrix(3,5:6)'];
end


% Návrh pásmových propustí - 1/3oct 192kHz
fs = 192000;
bands = size(f192_3,1);
c_192_3 = zeros(9 * bands, 1);
for b = 1 : bands
    f = fdesign.bandpass('n,f3dB1,f3dB2',6,f192_3(b,1),f192_3(b,2),fs);
    h = design(f, 'butter');

    % Pořadí koeficientů : scale1 a11 a21 scale2 a12 a22 scale3 a13 a23
    c_192_3(9*b - 8: 9*b, 1) = [h.ScaleValues(1); h.sosMatrix(1,5:6)';
                                h.ScaleValues(2); h.sosMatrix(2,5:6)';
                                h.ScaleValues(3); h.sosMatrix(3,5:6)'];
end

% Nefiltrující koeficienty
c_allpass = [1; 0; -1; 1; 0; -1; 1; 0; -1];
```

```
% Složení jedné matice koeficientů
c = [c_allpass; c_192_1; c_192_3];
c_count = size(c, 1);

% Sloupec adres - Vzestupné číslování od nuly
addr = (0 : c_count - 1)';

% Zápis .mif souboru
file = fopen(FileName, 'w+');

fprintf(file,
'-- Koeficienty oktavovych a tretinoktavovych pasmovych propusti.\r\n');
fprintf(file,
'-- Jedna sada = 9 x 64b float (koeficienty 3 sekci druheho radu)\r\n');
fprintf(file,
'-- Poradi koeficientu : g1, a11, a12, g2, a12, a22, g3, a13, a33\r\n');
fprintf(file,
'-- Filtr 00       nefiltrujici filtr\r\n');
fprintf(file,
'-- Filtr 01..11   oktavova pasma pro fs = 192 kHz\r\n');
fprintf(file,
'-- Filtr 12..44   1/3 oktavova pasma pro fs = 192 kHz\r\n');

fprintf(file,
'WIDTH=64;\r\nDEPTH=%u;\r\n\r\nADDRESS_RADIX=UNS;\r\n',MifDepth);
fprintf(file,
'DATA_RADIX=HEX;\r\n\r\nCONTENT BEGIN\r\n');

for j = 1 : c_count
 fprintf(file, '%6u  :   %bX;\r\n', addr(j), c(j));
end

fprintf(file,'[%u..%u]  :   0000000000000000;\r\n',c_count, MifDepth - 1);
fprintf(file,'END;\r\n');

fclose(file);
```