VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2017 Bc. Lukáš Bari

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

GENERÁTOR PAKETŮ NA PLATFORMĚ FPGA

PACKET GENERATOR ON THE FPGA PLATFORM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Lukáš Bari

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. David Smékal

BRNO 2017

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
TECHNICKÉ V BRNĚ

Diplomová práce

magisterský navazující studijní obor **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Lukáš Bari ***ID:*** 154675

Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Generátor paketů na platformě FPGA

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je navrhnout nástroj ke generování síťového provozu na platformě FPGA. Jedná se o realizaci generátoru síťového provozu na rozhraní 10 Gb/s.
Seznamte se s programovacím jazykem VHDL, FPGA kartami COMBO, vývojovým frameworkem NetCOPE a platformou Xilinx Vivado.
Navrhněte architekturu modulu v jazyce VHDL a ověřte jeho funkčnost v simulaci a ve fyzické implementaci na kartě COMBO. V závěru diskutujte dosažené výsledky.

DOPORUČENÁ LITERATURA:

[1] PINKER, Jiří, Martin POUPA. Číslicové systémy a jazyk VHDL. 1. vyd. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-7300-198-5.

[2] GROLÉAT, T., A. BOURGE, M. ARZEL, Y. LE BALCH, S. VATON a H. BOUGDAL. Flexible, extensible, open-source and affordable FPGA-based traffic generator [online]. 2012, [cit. 2016-09-03]. Dostupné z: https://portail.telecom-bretagne.eu/publi/public/fic_download.jsp?id=16866

Termín zadání: 1.2.2017 ***Termín odevzdání:*** 24.5.2017

Vedoucí práce: Ing. David Smékal
Konzultant:

doc. Ing. Jiří Mišurec, CSc.
předseda oborové rady

ABSTRAKT

Práca sa zaoberá teóriou a návrhom generátora sieťového provozu na platforme FPGA. Pre popis je použitý programovací jazyk VHDL. Práca zahrňuje zoznámenie sa s vývojovými postupmi a návrhovými prostriedkami potrebnými pre vytvorenie celkového projektu. Taktiež obsahuje zoznámenie sa s potrebnými technológiami FPGA, NetCOPE a kartami COMBO. Na základe týchto informácií, bol v praktickej časti navrhnutý, otestovný a implementovaný generátor paketov pre kartu Combo-80G. Pri realizácií bolo použite vývojové prostredie NetCOPE.

KLÍČOVÁ SLOVA

Packet, FPGA, LFSR, Xilinx, VHDL, Combo, NetCOPE

ABSTRACT

The thesis deals with the theory and design of the network traffic generator on the FPGA platform. The VHDL programming language is used for the description. The work involves getting acquainted with the development processes and design tools needed to create the overall project. It also includes familiarity with the necessary FPGA, NetCOPE and COMBO cards. Based on this information, was designed, tested and implemented packet generator project for the Combo-80G card. For implementation was used framework from NetCOPE.

KEYWORDS

Packet, FPGA, LFSR, Xilinx, VHDL, Combo, NetCOPE

BARI, L. Generátor paketů na platformě FPGA. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2017. 59 s. Vedoucí semestrální práce Ing. David Smékal

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma "GENERÁTOR PAKETŮ NA PLATFORMĚ FPGA" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4
Trestního zákoníku č. 40/2009 Sb.

Místo …… ……

(podpis autora)

ÚVOD

Počítačové systémy sa stávajú stále zložitejšie. V oblasti sieťového spracovania informácií je potreba stále vyšie nároky na výpočetný výkon k dosiahnutiu vyšších rýchlosti. K týmto účelom je možné použiť technológiu FPGA, ktorá prináša možnosť akcelerácie hardveru. S tým však prichádzajú aj vyšie nároky na vývojarov. Preto vzniká snaha popisovať chovanie hardvéru pomocou vyšších programovacích jazykov. Cieľom tejto práce bolo zoznámiť sa s možnostoťami generovania paketov na platforme FPGA pomocou vývojového prostredia Xilinx Vivado a jazyka VHDL a navrhnúť princip generátora paketov.

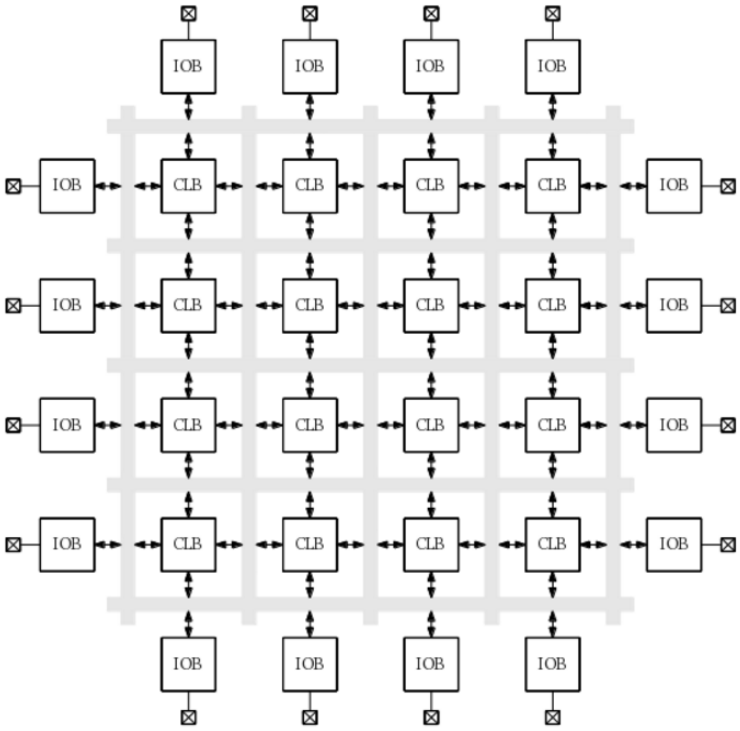
V prvej časti práce je vysvetlená funkcionalita štruktúry FPGA vo všeobecnosti. Použité vývojové prostriedky a ich význam, pri vytváraní projektu v prostredi Xilinx Vivado, ako aj priblíženie použitého popisného jazyka VHDL. Ďalej je venovaná kapitola popisu karty COMBO a použitého čipu Virtex-7. Oboznámie sa s vývojovim frameworkom od firmy NetCOPE a jeho vrstvami, komunikačnými prostriedkami a ovládaním. Poslednou časťou teoretického úvodu je oboznámenie sa s teóriou znáhodňnovania generovanej informácie v hardvery, posúvneho registra LFSR s priamou náväznosťou na praktickú časť diplomovej práce.

V druhej časti, je na začiatok vysvetlený popis funkcie navrhnutého bloku, pseudo-náhodného generátora, v jazyku VHDL, ako jedným zo základných blokov finálneho projektu. Zoznámenie sa s vytvorenou štruktúrou generovaného paketu UDP a následneho zabalenia do IP paketu. Nastavenie jednotlivých polí záhlavia, a vnornie do kódu. Záver je práce je venovaný postupných krokom finalizácie projektu, respektíve simulácie, syntézy a následnej implementácie projektu a diskusii venovanej dosiahnutým výsledkom.

1. Programovateľné hradlové polia

Programovateľné hradlové polia sú najvýznamnejšiou skupinou programovateľných logických obvodov. Sú to najuniverzálnejšie programovateľné logické obvody s najvšeobecnejšej vnútornou štruktúrou.

Veľmi zjednodušená principiálna štruktúra programovateľného hradlového poľa na Obr. 1 vystihuje základné a najdôležitejšie vlastnosť FPGA, ktorá ich odlišuje od ich predchodcov.



Obr. 1: Zjednodušená štruktúra obvodu typu FPGA [1]

Obvody označované ako FPGA (Field Programmable Gate Array) majú zo všetkých užívateľom programovateľných obvodov najvšeobecnejšej štruktúru a zároveň zvyčajne obsahujú najväčšie množstvo využiteľných logických prostriedkov. Obvod FPGA sa skladá z blokov vlastnej programovateľnej logiky CLB (Configurable Logic Blocks), vstupno-výstupných buniek IOB (Input-Output Blocks), programovateľnej prepojovacej matice a konfiguračných prostriedkov riadiacich konkrétne nastavenie jednotlivých blokov a ich vzájomné prepojenie.

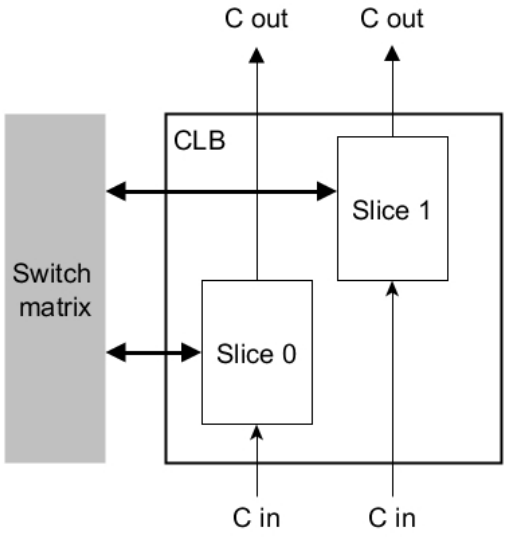
Jednotliví výrobcovia navyše pridávajú do FPGA ďalšie funkčné bloky rozširujúce možnosti ich obvodov. Dnešné typické programovateľné hradlové polia obsahujú synchróne blokové pamäte, bloky pre správu hodinových signálov, rýchle sériové vstupno-výstupné rozhrania, bloky pre realizáciu aritmetických operácií, mikroprocesory, pamäťové radiče a podobne.[3]

1.1 Štruktúra FPGA

Programovateľné hradlové polia sú tvorené veľkým množstvom menších konfigurovateľných blokov všeobecné logiky, ktoré sú navzájom prepojiteľné prepojovacou maticou. Komunikáciu s okolitým prostredím zaisťujú vstupno-výstupné bloky. Okrem logických blokov, vstupno-výstupných blokov a prepojovacej matice obsahujú FPGA ešte ďalšie špeciálne funkčné bloky, ktoré sa u jednoduchších PLD nepoužívali. Jedná sa napríklad o bloky statických pamätí RAM, správy hodinových signálov, špeciálne vstupno-výstupné bloky a podobne. [3]

1.1.1 Logické bloky

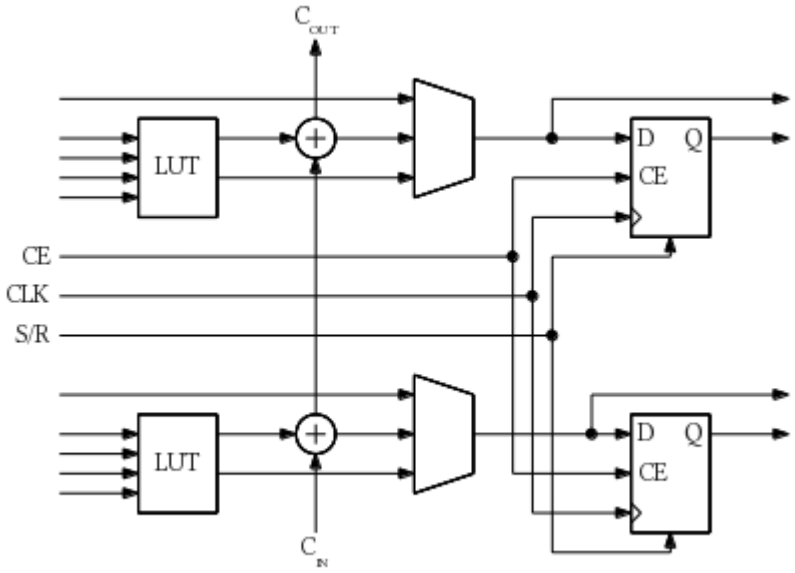
Konfigurovateľné logické bloky sú základným stavebným prvkom FPGA. Jedná sa o bloky umožňujúce realizáciu kombinačných aj sekvenčných logických funkcií viacerých jednotiek až desiatok binárnych vstupov s niekoľkými výstupmi. Firma Xilinx označuje tieto základné bloky Configurable Logic Blocks (CLB). Vzájomné prepojenie jednotlivých konfigurovateľných logických blokov zaisťuje programovateľná prepojovacia matica.



Obr. 2: Konfigurovateľný logický blok

Konfigurovateľné logické bloky sú zvyčajne realizované ako kombinácia dvoch menších blokov označovaných ako logické rezy, ako je znázornené na obrázku Obr. 2. Názov vychádza z názvoslovia firmy Xilinx – Slices. Zjednodušené všeobecné schéma jedného rezu je na Obr. 3.

Skutočná realizácia elementárneho bloku logiky je v dnešných FPGA oveľa zložitejšia a rezy zvyčajne obsahujú dvojnásobné množstvo základných logických prvkov. Zjednodušená schéma je lepšie pre pochopenie základnej štruktúry.



Obr. 3: Všeobecný logický rez

Typický logický rez sa skladá zo štyroch vzájomne sa doplňujúcich častí:

- Programovateľné logické tabuľky
- Reťazec rýchleho šírenia prenosu
- Multiplexory
- Registre

Programovateľné logické tabuľka LUT (look-up table) slúži k realizácii ľubovoľnej kombinačné logické funkcie. Ide v podstate o pamäť typu RAM, ktorej obsah je pri bežnom použití pevne daný konfiguračnými dátami. Realizuje teda jednu alebo dve binárne funkcie niekoľkých (typicky štyroch, piatich, alebo šiestich) vstupných binárnych premenných. Okrem tejto základnej funkcie umožňujú LUT zvyčajne realizáciu aj iných typov funkčných blokov ako sú malé pamäte RAM a ROM, alebo posuvné registre.

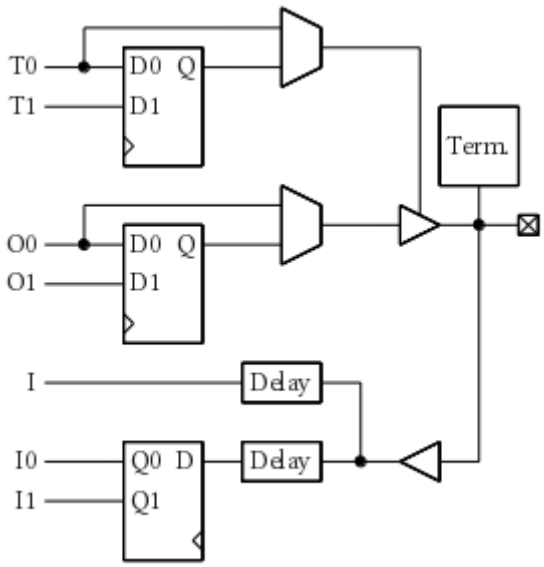
Reťazec rýchleho šírenia prenosu (carry chain) umožňuje vytvorenie rôznych aritmetických obvodov, ako je napríklad sčítačka. Zatiaľ čo bežné vstupy a výstupy logických rezov sú pripojené k pomalej globálnej prepojovacej matici, vstup a výstup prenosu C_{IN} a C_{out} sú priamo prepojené so susednými konfigurovateľnými logickými bloky v rovnakom stĺpci.

Multiplexery umožňujú realizáciu logických funkcií viac premenných ako je šírka jednej LUT. Funkcia je potom realizovaná skombinovaním výstupov niekoľkých LUT v jednom logickom reze pomocou multiplexerov. Obdobné funkcionality je síce možné dosiahnuť kombináciou niekoľkých logických blokov, ale prepojenie medzi jednotlivými CLB je oveľa pomalšia ako priamy spoj medzi LUT a multiplexory.

Registre v logických rezoch umožňujú realizáciu sekvenčných logických funkcií. Pri konfigurácii rezu je zvyčajne možné nastaviť vlastnosti jednotlivých registrov ako napríklad využitie clock-enable, polaritu a typ set / reset vstupu a podobne. Špeciálne vstupy pre registre na Obr. 3 sú obvykle zdieľané všetkými registre v rámci jedného konfigurovateľného logického bloku alebo aspoň v rámci rezu. To znamená, že všetky registre musia používať rovnaký hodinový signál, clock-enable, aj reset. Nie každý register však musí clock-enable a reset používať, prípadne môže reset využívať iným spôsobom ako ostatné registre.

1.1.2 Vstupno-výstupné bloky

Vstupno-výstupné bloky (IOB) sprostredkúvajú spojenie medzi FPGA a vonkajším prostredím. Každý vonkajší signál vstupuje do FPGA prostredníctvom IOB a každý výstupný signál opúšťa FPGA opäť cez IOB. Zjednodušená schéma vstupno-výstupného bloku je na Obr. 4. Vstupno-výstupný blok obsahuje vstupné a výstupné registre, budiče a prijímače, oneskorovacie linky, obvody impedančného prispôsobenia a ochranné obvody. Vstupné aj výstupné registre sú u všetkých dnešných FPGA realizované pomocou kombinácie dvoch registrov tak, aby umožňovali vstup a výstup DDR signálov. Samozrejme môžu byť nakonfigurované do bežného režimu SDR.



Obr. 4: Vstupno-výstupný blok

Výstup signálu je realizovaný voliteľným výstupným registrom nasledovaným trojstavovým budičom. Výstup budiča je priamo pripojený na pin FPGA. Trojstavový budič je riadený druhou dátovou cestou.

Vstupný externý signál je privedený na konfigurovateľný prijímač, v schéme predstavovaný iba jednoduchým budičom. V skutočnosti sa jedná o rôzne prijímače pre jednotlivé podporované I/O štandardy nasledované multiplexom. Výstup prijímača môže byť cez konfigurovateľnú oneskorovacie linku privedený

priamo na prepojovaciu maticu FPGA alebo na DDR / SDR vstupný register, a až z neho na prepojovaciu maticu. Pre vstup diferenciálnych signálov obsahuje prijímač ešte druhý vstup pripojený do susedného I/O bloku.

Programovateľné spomaľovacie linky umožňujú realizáciu časového posunu vstupného signálu. Tým je možné napríklad korigovať vzájomný fázový posun signálov, prípadne posun signálu voči hodinám.

Ochranné obvody, ktoré nie sú na zjednodušenom schéme nakreslené, chránia logiku IOB pred poškodením z vonkajšieho sveta v dôsledku EMI (Electromagnetic Interference) alebo nesprávneho použitia či zapojenia. Obvody impedančného prispôsobenia umožňujú prispôsobiť vnútornú impedanciu budiča v FPGA, impedanciu externého vedenia Novšie FPGA podporujú dynamické riadenie výstupnej impedancie, čo je dôležité napríklad pre pripojenie rôznych variant DDR a QDR rozhraní alebo napríklad PCIe.

1.1.3 Prepojovacie prostriedky

Vzájomné prepojenie jednotlivých funkčných blokov FPGA zaisťujú rôzne programovateľné prepojovacie matice a spoje. Typické hradlové pole obsahuje tri rôzne typy prepojovacích prostriedkov:

- globálne
- lokálne
- špeciálne

Globálne prepojovacie prostriedky sú najuniverzálnejší a umožňujú vzájomné prepojenie ľubovoľných funkčných blokov FPGA. Sú realizované ako niekoľko vrstiev rôzne organizovaných spojov s prepojovacími maticami pre jednotlivé funkčné bloky. Vlastná organizácia spojov sa líši medzi výrobcami aj medzi jednotlivými produktovými radmi. Často bývajú spoje organizované ako sada horizontálnych a vertikálnych vodičov rôznych dĺžok. Programovateľné prepojovacie matice umožňujú pripojenie vstupov a výstupov funkčných blokov k jednotlivým prepojovacím vodičom. Hoci globálne prepojovacie prostriedky poskytujú najviac stupňov voľnosti v prepojení jednotlivých blokov, majú zase niektoré nevýhody. Každý globálny spoj je pomerne dlhý vodič, s veľkým množstvom programovateľných spínačov a teda veľkou kapacitou, čo sa prejaví na rýchlosti šírenia signálov.

Lokálne prepojovacie prostriedky umožňujú prepojenie iba susedných funkčných blokov. Jedná sa napríklad o prepojenie reťazcov šírenia prenosu logických rezov, prepojenie diferenciálnych signálov vstupno-výstupných blokov, ale aj krátke rýchle spoje medzi susednými konfigurovateľnými logickými blokmi. Lokálne spoje sú oveľa kratšie, s menej spínačmi a nižšou kapacitou ako spoje globálne. Lokálne spoje teda spôsobujú oveľa nižšie oneskorenie signálov ako globálne spoje.

Špeciálne prepojovacie prostriedky slúžia na spojenie medzi vyhradenými vstupy a výstupy funkčných blokov. Tieto špeciálne prepojovacie prostriedky sú

optimalizované tak, aby spôsobovali čo najmenšie oneskorenie prechádzajúcich signálov. Samozrejme nemôžu byť použité pre bežné signály, ale iba pre jeden konkrétny typ signálov. Napríklad hodinové spoje môžu byť pripojené len k vyhradeným hodinovým vstupom a výstupom jednotlivých funkčných blokov, ale už nie k ostatným bežným portom. Množstvo špeciálnych prepojovacích prostriedkov v FPGA je zvyčajne veľmi obmedzené a často bývajú realizované v dvoch rôznych prevedeniach ako globálne a lokálne podobne ako bežné signálové spoje.

1.2.4 Špeciálne funkčné bloky

Dnešné programovateľné hradlové polia obsahujú ďalšie, často jednoúčelové, funkčné bloky uľahčujúce použitie FPGA. Typickými zástupcami tejto kategórie sú blokové pamäte, násobičky a iné aritmetické obvody, obvody pre správu a generovanie hodinových signálov a ďalšie.

Blokové pamäte sú zvyčajne dvoj-portové statické pamäte RAM s kapacitou jednotiek až desiatok kilo-bytov a s konfigurovateľnou šírkou adresných a dátových zberníc. Takýto pamäťový blok je vysoko univerzálny. Môže byť použitý ako jedno-portová aj dvoj-portová pamäť typu RAM aj ROM. Dátové zbernice môžu byť obvykle nastavené na šírku 1, 2, 4, 8/9, 16/18, 32/36, alebo 64/72 bitov. Čísla za lomkou predstavujú varianty s paritným bity. Dvoj-portové blokové pamäte uľahčujú realizáciu rôznych typov špeciálnych pamätí ako sú FIFO, kruhový buffer, CAM a pod.

Aritmetické obvody sú reprezentované zvyčajne celočíselnými násobičkami, prípadne zložitejšími DSP bloky umožňujúcimi realizáciu náročnejších matematických operácií v jednom hodinovom cykle. Typickým príkladom jednoduchších, ale najčastejších operácií realizovaných pomocou DSP blokov je MAC (Multiply-And-Accummulate), čo je základná operácia číslicovej filtrácie a takmer všetkých DSP algoritmov.

Bloky pre správu a generovanie hodinových signálov sú založené na PLL (Phase-Locked Loop) alebo DLL (Delay-Locked Loop). Zložitejšie hodinové bloky vo väčších FPGA často obsahujú aj niekoľko PLL a DLL. Tieto bloky ponúkajú funkcie ako je syntéza a delenie hodinových signálov, generovanie fázovo posunutých hodín, kompenzácie vnútorných i vonkajších oneskorení a fázových posunov a podobne. Vstupy a výstupy týchto blokov sú zvyčajne pripojené priamo k špeciálnym spojom určeným pre šírenie hodinových signálov. Na rozdiel od predchádzajúcich dvoch typov špeciálnych funkčných blokov nejdú hodinové obvody jednoducho nahradiť bežnú logikou FPGA.

Sériovej transceivery umožňujú pomerne jednoduchú realizáciu veľmi rýchlych sériových rozhraní. Blok transceiveru zvyčajne obsahuje diferenciálny budič a prijímač, serializér a deserializér, kodér a dekodér, extraktor hodinového signálu a ďalšie pomocné obvody. Pomocou transceiverov je možné sériovo komunikovať s okolím rýchlosťami v rádoch jednotiek až desiatok Gb/s. Interné

rozhranie v FPGA je potom realizované synchrónnymi paralelnými dátami s oveľa nižšou hodinovou frekvenciou. Sériové transceivery sú nevyhnutné pre realizáciu mnohých štandardných sériových rozhraní ako sú napríklad PCIe, XAUI pre 10Gbps Ethernet a mnohé ďalšie.

Z ostatných funkčných blokov, ktoré sa vyskytujú v FPGA si zaslúžia zmienku radiče pamätí a mikroprocesory. Niektoré hradlové polia obsahujú hotový radič dynamických pamätí, ktorý zvyčajne podporuje pamäte typu SDR, DDR, DDR2 a DDR3 SDRAM. Hotový blok radiča ušetrí veľa logických prostriedkov a často významne zvýši dátovú priepustnosť externej dynamickej pamäte oproti realizácii radiče pomocou bežnej logiky.

2. Vývojové prostriedky

Programovateľné logické obvody, s akokoľvek dokonalou vnútornou architektúrou, by neboli prakticky použiteľné bez zodpovedajúcej podpory vývojových prostriedkov. Teoreticky, je pri znalosti vnútornej architektúry konkrétneho PLD (Programmable Logic Device), možné priamo vytvoriť finálnu schému prepojenia jednotlivých funkčných blokov. Pri jednoduchých PLD, ako sú obvody PAL alebo GAL je niečo také aj prakticky realizovateľné, ale u dnešných zložitých FPGA je to niečo úplne nereálne.

2.1 Typy vývojových prostriedkov

Základné vývojové prostriedky pre programovateľnú logiku musia zabezpečiť transformáciu vstupného návrhu alebo popisu funkcie do výstupného formátu použiteľného pre konfiguráciu alebo naprogramovanie konkrétnej súčiastky. V skutočnosti je to len jedna z mnohých funkcií dnešných vývojových prostriedkov používaných na aplikáciu programovateľné logiky. Vývojové prostredia pre FPGA sa stávajú neustále zložitejšie a komplexnejšie. Sú často podobné veľkým vývojovým systémov pre digitálne integrované obvody a mnohé používané nástroje sú totožné.

Typické vývojové prostriedky pre programovateľnú logiku je možné rozdeliť do niekoľkých základných skupín podľa ich funkcie:

- Implementácia
- Verifikácia
- Hardware

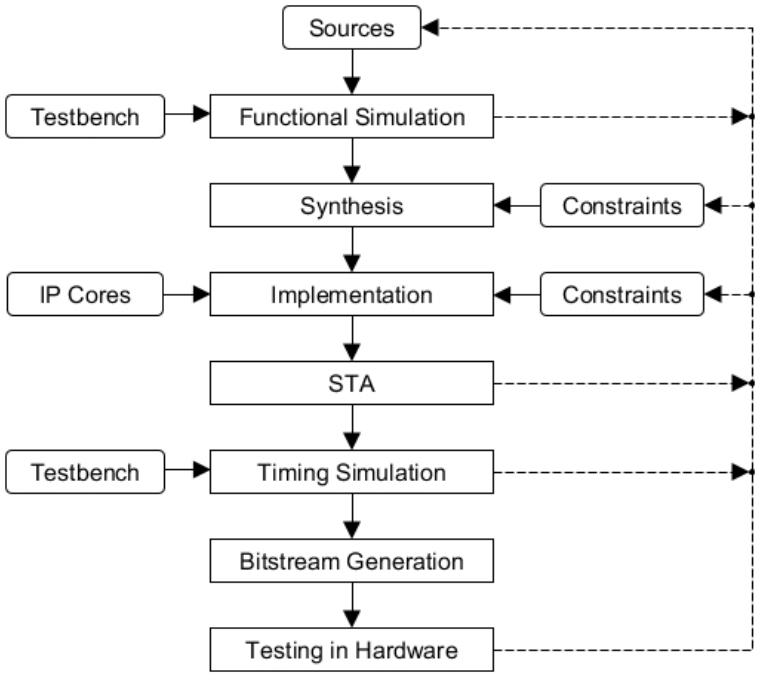
Implementačné nástroje slúžia pre prevod návrhu funkcie do výstupného formátu použiteľného pre fyzickú konfiguráciu programovateľné súčiastky. Do tejto skupiny patria nástroje pre syntézu a implementáciu.

Verifikačný nástroje slúžia na overenie funkcie ako samotného návrhu, tak aj jednotlivých medzivýsledkov implementačného procesu. Do tejto skupiny patria číslicové alebo zmiešané simulátory a rôzne analytické nástroje. Môže sa jednať o nástroje pre statickú časovú analýzu (STA) alebo napríklad analyzátory výstupov implementačných prostriedkov.

Hardvérové prostriedky slúžia predovšetkým pre verifikáciu, overenie funkcie výsledného obvodu a pre jeho charakterizáciu. Jedná sa napríklad o rôzne vývojové a testovacie dosky, programátorov, programovacie adaptéry a JTAG adaptéry pre konfiguráciu, meranie, testovanie a ladenie. Používajú sa často pre overenie funkcie a charakterizáciu pred finálnym návrhom vlastnej dosky plošných spojov. Niektoré z hardvérových prostriedkov slúži k produkčnému programovanie PLD alebo ich konfiguračných pamäťou.

2.2 Vývojové postupy

Schéma typického postupu vývoja pre programovateľné logické obvody je znázornené vývojovým diagramom na obrázku Obr. 5.



Obr. 5: Schéma vývojového postupu

Jedná sa o zjednodušený postup vychádzajúci z postupu vývoja číslicových integrovaných obvodov.

V diagrame na obrázku Obr. 5 sú použité anglické výrazy, pretože niektoré z nich nemajú slovenské ekvivalenty. Význam jednotlivých blokov diagramu je popísaný ďalej.

2.2.1 Sources (Zdroje)

Súbor vstupných dát, ktoré popisujú predpokladanú funkciu výsledného obvodu. Môže sa jednať o kombináciu textových súborov v rôznych jazykoch, obvodových a blokových schém, stavových diagramov a podobne.

Do procesu návrhu programovateľných logických obvodov vstupuje veľa rôznych dát, slúžiacich pre popis funkcie, pomocné dáta a skripty pre simulácie a verifikácie, pomocné dáta a skripty pre implementáciu a ďalšie. Požadovaná výsledná funkcia programovateľného obvodu môže byť popísaná kombináciou niekoľkých nasledujúcich základných metód:

- Popis pomocou niektorého jazyka pre popis hardvéru (HDL- Hardware Description Language).
- Opis pomocou podporovaného vysokoúrovňového jazyka - HLL (High-Level Language).
- Bloková schéma zapojenia, kde jednotlivé bloky sú vnútorne realizované ľubovoľnou inou metódou.
- Obvodové schéma zapojenia využívajúce pripravené knižnice základných primitív aj funkčných blokov na vyššej úrovni.
- Ďalšie podporované formy grafického popisu ako napríklad stavové alebo vývojové diagramy, vysokoúrovňové diagramy MATLAB / Simulink a podobné.
- IP jadra vo forme netlistu alebo pevného makra.
- Ďalšie špeciálne súbory ako napríklad definícia procesorových subsystémov.

2.2.2 Testbench

Pomocné súbory pre simulácie a verifikácie. Zvyčajne sa jedná o generovanie testovacích vektorov a vyhodnocovanie výsledkov testov vo vhodnom jazyku. Súčasťou bývajú tiež simulačné modely použitých IP jadier a externých súčiastok. Testovacie prostredie zvané Testbench zvyčajne zaisťuje prepojenie vlastného návrhu programovateľného obvodu do celého simulačného systému, generovanie vstupných signálov a dát pre budenie testovaného modulu a kontrolu výstupov. Kontrola správnej funkcie môže byť zabezpečená automaticky priamo testovacím prostredím. Funkciu jednoduchších obvodov je možné vykonávať napríklad sledovaním vstupných a výstupných časových priebehov a stavov jednotlivých signálov v konkrétnych miestach obvode a okamihoch simulácie.

2.2.3 IP Cores

Hotové funkčné bloky využité v návrhu. Môže sa jednať o zakúpená jadra, alebo vlastné univerzálne moduly pripravené pre znovu-využitie v mnohých rôznych projektoch. Jadrá sú pre implementáciu použité vo forme netlistov alebo iných formátov priamo podporovaných implementačnými nástrojmi.

Veľmi významnú skupinu implementačných nástrojov tvoria práve nástroje pre prácu s IP jadrami. Značná časť funkcie návrhu pre dnešné veľké FPGA je často realizovaná pripravenými hotovými funkčnými blokmi. Pre zjednodušenie ich konfigurácia, parametrizácia, modifikácie, prepojenie a integrácia existujú rôzne špecializované nástroje. Samozrejme existujú aj nástroje uľahčujúce vytváranie, balenie a distribúciu vlastných jadier.

Základné nástroje pre prácu s jadrami umožňujú konfiguráciu parametrov autonómnych IP jadier. Výsledkom je zvyčajne zdrojový kód alebo netlist použiteľný spoločne s ostatnými zdrojmi pre syntézu a verifikáciu. Tieto nástroje

sa používajú na prípravu často používaných bežných funkčných blokov, ako sú napríklad rôzne typy pamätí, bloky realizujúce aritmetické operácie, rôzne špecializované rozhranie a podobne. Ak chce napríklad vývojár použiť v návrhu FIFO, stačí do projektu vložiť IP jadro pre FIFO a v jeho konfigurácii vybrať šírku dátových zberníc, hĺbku pamäte a nakonfigurovať potrebné riadiace signály. Výsledkom je potom obvykle netlist, ktorý sa môže v návrhu používať ako akýkoľvek iný bežný modul.

S rastúcimi možnosťami programovateľnej logiky je dnes v FPGA bežné použitie procesorov alebo celých procesorových systémov. Špecializované nástroje umožňujú zostavenie a konfiguráciu rozsiahlych procesorových systémov, vzájomné prepojenie jednotlivých blokov pomocou rôznych zberníc. Okrem výstupov použiteľných pre implementáciu a verifikáciu umožňujú tieto nástroje vygenerovať výstupy pre následný vývoj software. Samozrejmosťou býva možnosť spoločnej simulácie návrhu hardvér a softvér v čase, keď ešte nie je k dispozícii finálne hardware.

2.2.4 Constraints

Jedná sa o súbory popisujúce rôzne obmedzujúce podmienky, ktorými sa nástroje pre syntézu a implementáciu musí riadiť. Typicky ide o fyzické vlastnosti vstupno-výstupných signálov ako je priradenie pinov na puzdre súčiastky, nastavenie elektrických vlastností budičov a prijímačov a podobne. Ďalšou skupinou sú obmedzenia týkajúce sa umiestnenia a prepojenie funkčných blokov na čipe programovateľné súčiastky. Posledná dôležitá skupina obmedzujúcich podmienok riadia časové pomery v obvode ako je napríklad nastavenie hodinových frekvencií, povolenej oneskorenie prechodu signálu a podobne.

2.2.5 Functional Simulation

Pre programovateľnú logiku je najdôležitejší verifikačný krok funkčná simulácia. Mala by overiť funkčnú správnosť návrhu. Ak je návrh v poriadku, je možné pokročiť k ďalšiemu bodu, inak treba návrh upraviť tak, aby funkčné požiadavky overované simulácií splnil.

2.2.6 Synthesis

Prvý krok vlastnej konverzie vstupného popisu funkcie do formátu použiteľného pre konfiguráciu programovateľné súčiastky. Výstupom syntézy je takzvaný netlist, čo je v podstate schéma popísanej textovým alebo binárnym formátom. Netlist obsahuje už funkčné bloky cieľovej technológie, ako sú napríklad LUT, registre, budiče, a kombinuje ich navzájom zoznamy jednotlivých spojov.

2.2.7 Implementation

Slúži pre správne umiestnenie netlistu do konkrétneho programovateľného obvodu. Skladá sa z niekoľkých krokov, ktorých výsledkom je špeciálna verzia netlistu, kde každý blok aj každý spoj je jednoznačne umiestnený na konkrétnu pozíciu v programovateľné súčiastke. Implementačné nástroje potrebujú okrem

popisu funkcie ešte ďalšie vstupné dáta, ktoré ovplyvňujú výsledok implementačného procesu. Jedná sa predovšetkým o definícii jednotlivých fyzických obmedzujúcich podmienok, tzv. Constraints. Je nutné jednotlivým vstupno-výstupným signálom priradiť konkrétnej piny programovateľné súčiastky a nastaviť elektrické parametre vstupných prijímačov a výstupných budičov. Ďalšie parametre, ktoré majú vplyv na implementáciu sú časové vlastnosti hodinových signálov, dátových vstupov a výstupov, rovnako ako niektorých interných signálov. Dôležité je napríklad správne ošetriť prechody medzi asynchrónnymi hodinovými signálmi alebo dlhší kombinačné logické cesty. Posledný oblasťou je možnosť ručného umiestnenie niektorých funkčných blokov na čipe FPGA alebo vymedzenie oblastí, v ktorých smie byť daný blok automaticky umiestnený. Ide o takzvaný floorplannig. Ďalšou skupinou vstupov pre implementačný proces je správne nastavenie implementačných nástrojov. To je zvyčajne možné robiť interaktívne v GUI, pomocou konfiguračných súborov, alebo implementačných skriptov. Skriptovanie implementačných nástrojov pre programovateľnú logiku sa vykonáva typicky v jazyku TCL, niektoré nástroje podporujú aj Perl.

2.2.7 STA

Statická časová analýza je veľmi významnou súčasťou verifikačného procesu. Pre overenie skutočnej funkčnosti návrhu je na rovnakej významovej úrovni ako napríklad overenie funkcie pomocou simulácie. Každý návrh programovateľné logiky musí obsahovať okrem vlastného popisu funkcie tiež súbor obmedzujúcich podmienok, tzv. Constraints, ktoré definujú napríklad priradenie pinov jednotlivým vstupno-výstupným portom, správne nastavenie elektrických parametrov vstupov a výstupov, a okrem iného tiež definujú časové pomery v obvode. Z časových constraints je najdôležitejšia definícia frekvenciou hodinových signálov a požiadaviek na časovanie vstupných a prípadne i výstupných pinov. Ďalej je možné a často aj nevyhnutné definovať relatívne časové vzťahy medzi rôznymi hodinovými doménami, prípadne rôzne špeciálne nároky na časovanie v konkrétnych miestach obvode, ktoré nie sú zo samotného funkčného popisu zrejmé.

Nástroje pre statickú časovú analýzu použijú výstup implementácie, časové constraints a charakterizačné dáta použitého programovateľného obvodu a vykonajú kompletnú analýzu časovanie všetkých použitých synchrónnych elementov v celom programovateľnom obvode. Výstupný netlist obsahuje okrem prepojenia jednotlivých obvodových prvkov aj ich konkrétne umiestnenie na ploche čipu a použité prepojovacie prostriedky, takže časová analýza je schopná určiť oneskorenie všetkých spojov. Vďaka tomu, že programovateľné logické obvody sú od výrobcov úplne charakterizované vrátane presných modelov časovanie, stačí pre verifikáciu návrhu splniť nasledujúce tri podmienky:

- Správna funkcia návrhu zodpovedajúca špecifikácii overená funkčnou simuláciou.
- Všetky hodinové vstupné signály majú definované časovanie (constraints).
- Požiadavky na časovanie sú overené statickú časovou analýzou.

Na rozdiel od plne zákazkového vývoja ASIC potom odpadajú ďalšie časovo náročné kroky ako sú časové simulácie po jednotlivých implementačných fázach. Vďaka plnej charakterizácii programovateľných logických obvodov sú časové simulácie plne zastúpené funkčný simulácií v kombinácii so statickou časovou analýzou.

2.2.8 Timing Simulation

Časová simulácia sa pri vývoji pre programovateľnú logiku používa veľmi zriedka. Pri tejto simulácii sa namiesto vstupného popisu funkcie používa výstup implementácie prevedenej obvykle do netlistu v simulačnom jazyku. Tento netlist obsahuje aj kompletné informácie o skutočných meškaniach jednotlivých blokov a spojov. Pri vývoji je, za správnych postupov, správna funkcia výsledku zaistená už vďaka funkčnej simuláciu a STA. Programovateľné súčiastky i implementačné nástroje sú dostatočne charakterizované od výrobcu a funkčnú správnosť spoločne s overením splnenia časových požiadaviek pomocou STA zaručuje správnosť výsledku.

2.2.9 Bitstream Generation

Generátor bitstreamu prevedie výstupný netlist z implementácie na súbor, ktorý už priamo riadi nastavenie jednotlivých konfiguračných prepojok programovateľnej súčiastky. Tento súbor je už priamo použiteľný pre fyzické naprogramovanie alebo konfiguráciu PLD.

2.2.10 Testing in Hardware

Posledným krokom verifikácie je otestovanie funkcie v skutočnom programovateľnom obvode spoločne so skutočnými externými súčiastkami. Súčasťou testovania je zvyčajne počiatočné overenie funkcie, meranie parametrov a charakterizácia. V prípade problémov je možné využiť rôzne hardvérové prostriedky pre ladenie.

2.3 Návrhové jazyky

V súčasnosti sú najpoužívanejšie jazyky z rodiny HDL Verilog a VHDL. Hoci sa tieto jazyky líšia syntaxom aj sémantikou, oba poskytujú podobné prostriedky na opis logických obvodov. To, že sú tieto HDL určené prevažne na opis obvodov neznamená, že ich nie je možné použiť aj pre simuláciu a verifikáciu. Iba neposkytujú toľko nástrojov pre formálnu verifikáciu ako špecializované

verifikačné jazyky. Pre túto prácu bol použitý jazyk VHDL, preto sa ďalej budeme bližšie venovať práve jemu. [1]

2.3.1 Abstrakcia vo VHDL

Jazyk VHDL umožňuje popis obvodov na rôznych úrovniach abstrakcie:

- Behaviorálna
- RTL
- Štrukturálna
- Hradla

Behaviorálna úroveň znamená popis správania obvodu na vysokej úrovni abstrakcie. Túto úroveň poskytujú klasické HDL iba čiastočne a nie všetky nástroje ju vždy podporujú. Príkladom môže byť napríklad nejaký komplexný výpočet v plávajúcej rádovej čiarke zapísaný prirodzene vo forme matematickej rovnice.

RTL úroveň je typická oblasť nasadenia klasických HDL. Ide o nižšiu úroveň abstrakcie ako behaviorálny opis, kedy je funkcia definovaná kombináciou kombinačných a sekvenčných prvkov. Popis na tejto úrovni je zvyčajne podporovaný všetkými nástrojmi.

Štrukturálna úroveň popisuje vzájomné prepojenie jednotlivých funkčných modulov a je textovou obdobou blokových schém.

Úroveň hradiel opisuje obvod na najnižšej úrovni, kedy je funkcia vyjadrená s jednotlivými logickými hradlami. Táto úroveň zodpovedá klasickému schémy zapojenia.

Je obvyklé, že opis logického obvodu v VHDL kombinuje niekoľko úrovní abstrakcie.[1]

2.4 Vývojové prostredie

Ako Vývojové prostredie budeme používať Xilinx Viado Design Suite, ktorého súčasťou je jednak aplikácia Vivado, Xilinx SDK a potom tiež Vivado HLS. Vivado je aktuálny vývojový systém firmy Xilinx. Vivado podporuje iba novšie obvody Rad 7 a UltraScale. Jedná sa o pomerne nové Integrované prostredie, plne skriptovateľné v jazyku TCL, ktoré nahrádza predchádzajúcej generácie nástrojov ISE, XPS, WebPack, ChipScope Iné. Aplikácia Vivado je určená pre vytváranie návrhu pre FPGA, vykonáva sa tu navrh obvodu, logická syntéza, Implementácia generovanie bitstreamu. Prostredie Xilinx SDK slúži na programovanie ARM procesora. Aplikácia Vivado HLS slúži na vytváranie komponentov vo vysokoúrovňovom jazyku a za použitia syntézy na systémovéj úrovni je vygenerovaná komponenta s RTL popisom obvodu.

3. Karty COMBO

Jedným z cieľov bolo navrhnúť projekt tak, aby ho bolo možné projekt aplikovať na sieťové karty. Praktická realizácia projektu prebehla na sieťových kartách Combo vyvíjaných spoločnosťou Netcope. Implementácia prebehla na karte Combo – 80G [7].

3.1 Combo-80G

COMBO-80G FPGA karta viz Obr. 6. Je dvoj-portová hardvérová akcelerovaná sieťová karta podporujúca technológiu 40G a 10G Ethernet. Karta je určená na inštaláciu v hostiteľskom počítači prostredníctvom zbernice PCI Express, ktorá umožňuje vysokorýchlostné prenosy údajov medzi kartou a hostiteľským počítačom. V porovnaní s predchádzajúcimi generáciami kariet COMBO poskytuje vyššiu dátovú priepustnosť medzi kartou a hostiteľským počítačom a viac pamäťových modulov, čo má za následok vyššiu celkovú kapacitu pri zachovaní ostatných užitočných funkcií, konkrétne presného časového označenia. Zvyčajne sa používa v aplikáciách na monitorovanie sieťovej prevádzky, spracovanie a generovanie, ako aj hardvérová akcelerácia výkonných výpočtov a časovo kritických úloh.[8]

3.2 Parametre karty Combo – 80G

- Obvod FPGA Xilinx Virtex-7.
- 2× QSFP+ možnosť použitia monomódu, multimódu, CWDM alebo metalických transceiverov.
- Prevodník 4× 10G na 40G pre pripojenie technológie 10 Gigabit Ethernet.
- Rozhraní PCI Express ×8 3.0 s propustností do software 50 Gb/s.
- 2× obvod pamäti QDRII+ SRAM s kapacitou 72 Mb.
- 2× obvod pamäti RLDRAM 3 s kapacitou 576 Mb..
- 16× obvod pamäti DDR3 SDRAM s kapacitou 4 Gb.
- Synchronizácia externým PPS (Pulse per Second) signálom.
- Bootovanie firmvéru za chodu (bez nutnosti reštartovanie hostiteľského počítača).

Obr. 6: Karta Combo-80G [11]

3.3 Obvod FPGA Virtex - 7

Karty Combo-80G sú osadené obvody FPGA rodiny Virtex od spoločnosti Xilinx. Konkrétne obvodom Virtex-7 (XC7VX690T-FFG1157). Obvody FPGA tvoria kompromis medzi obvodmi ASIC navrhnutými pre konkrétnu aplikáciu (vysoký výkon, nízka spotreba, niekoľkoročný vývoj, vyplatí sa len v obrovských sériách) a obvody zloženými z diskrétnych súčiastok (nízky výkon, vysoká spotreba, krátky vývoj, vhodný pre malé série). Kombinačné časť logické bunky tvoria tabuľka hodnôt LUT, ktorá má u rodín Virtex-7 šesť vstupov. Sekvenčné časť logické bunky tvoria klopný obvod typu D. Presné konfigurácie obvodov FPGA Virtex-7 osadených na kartách Combo-80G je možné vyčítať z príslušných katalógových listov [8].

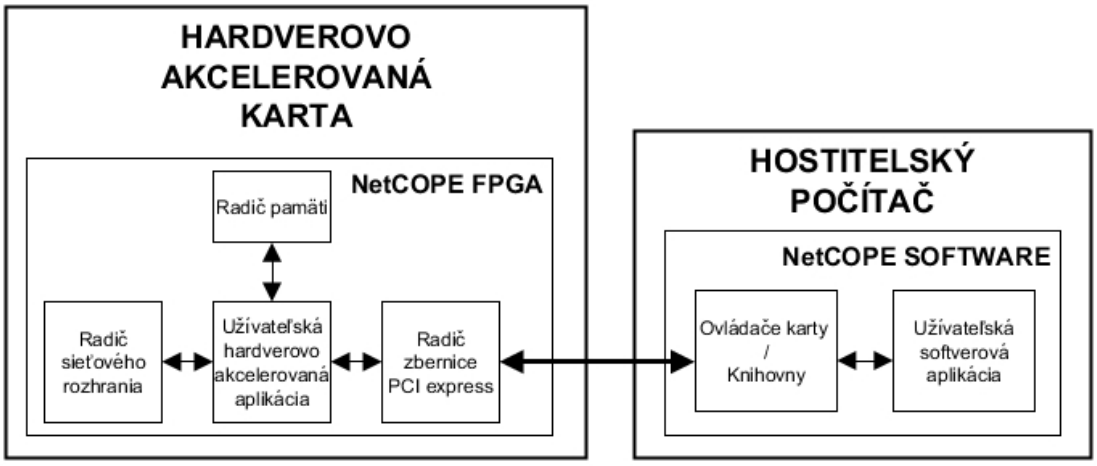
3.4 Rozhranie PCI Express

PCI Express je vysokorýchlostná sériová zbernica založená na point-to-point topológiu sa vzájomne oddelenými linkami spájajúcimi zariadenie s hlavnou zbernicou zvanou root complex. Každá linka sa skladá z dvoch diferenciálnych prenosových párov. Jeden z párov slúži pre príjem a druhý pre vysielanie dát. Komunikácia na linke prebieha formou zapuzdrených paketov, pričom je možný súčasný prenos dát oboma smermi. PCI Express sloty môžu mať jednu (×1), dve (×2), štyri (×4), osem (×8), šestnásť (×16) alebo tridsaťdva liniek (×32). Pri komunikácii po viacerých linkách súčasne je vysielaný paket rozdelený a prenášaný po niekoľkých linkách súčasne.

Karta Combo-80G používa na komunikáciu a pre napájanie svojej spotreby rozhranie PCI Express 3.0 × 8. Ide o tretiu generáciu PCI Express s ôsmimi linkami. Proti PCI Express 1.1 × 8 sa líšia enkódovánim 128 / 130b a vyššiu dátovú priepustnosťou 985 MB/s na linku. Celková maximálna dátová priepustnosť PCI Express 3.0 × 8 je 7880 MB/s. [9]

4. NetCOPE

NetCOPE je framework určený na vývoji aplikácií pre hardvérovo akcelerované karty Combo. Sú v ňom implementované radiče a rozhranie pre prácu s perifériami kariet Combo. Nad týmito radiči je vytvorená abstraktná vrstva pre vývoj užívateľských aplikácií. Tie je preto možné len s minimálnymi úpravami prenášať medzi rôznymi typmi kariet Combo. Zjednodušená bloková schéma celého prostredia NetCOPE je znázornené na Obr. 7.][14] [15]

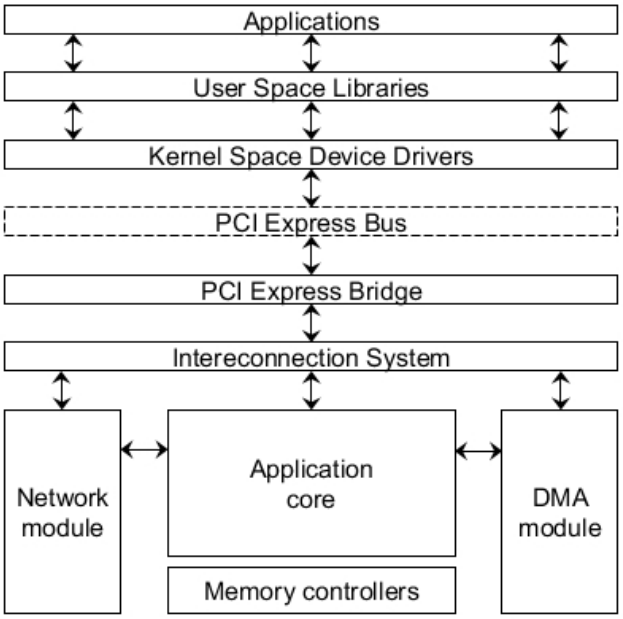


Obr. 7: Bloková schéma vývojového prostredia NetCOPE

Z obrázku je zrejmé, že sa framework skladá z dvoch častí. Firmvér NetCOPE určený pre hardwarovo akcelerovanú kartu s obvodom FPGA a softvér NetCOPE určený pre hostiteľský počítač. Firmvér a softvér NetCOPE spolu vzájomne komunikujú cez rozhranie PCI Express.

4.1 Vrstvy prostredie NetCOPE

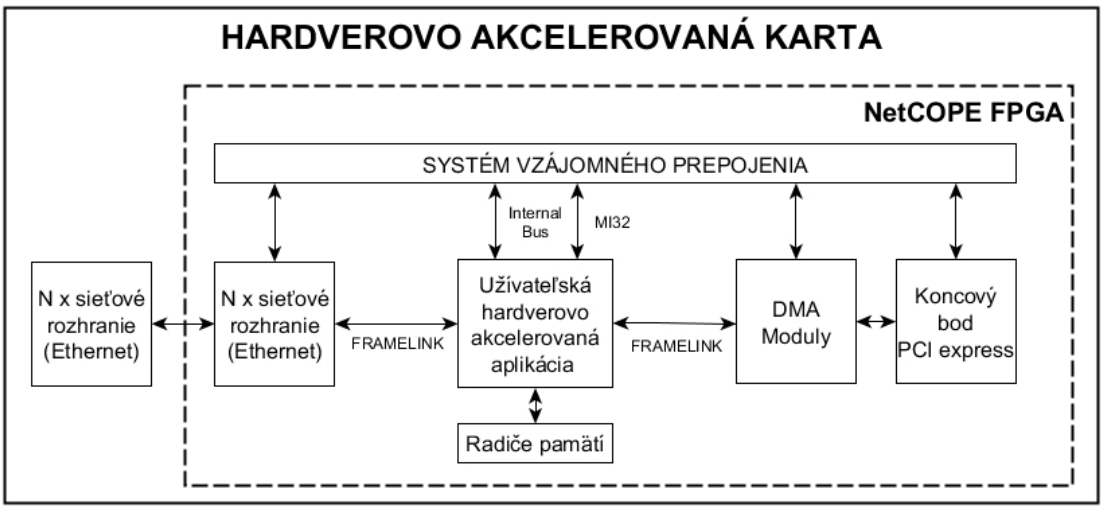
Framework NetCOPE možno rozdeliť na niekoľko vrstiev, ktorých štruktúra je znázornená na Obr. 8. Komunikáciu s hardvérom zabezpečuje hardvérová vrstva frameworku NetCOPE, ktorá vytvára na hardvéri nezávislú abstraktnú vrstvu. Do nej sa implementuje užívateľom napísaná hardwarovo akcelerovaná aplikácia určená pre obvod FPGA. Nad touto vrstvou je softvérová časť frameworku NetCOPE, ktorá cez rozhranie PCI Express zabezpečuje prenos dát z hardvérovo akcelerovanej aplikácie v obvode FPGA do hostiteľského počítača a jednoduchý prístup k nim z najvyššej vrstvy. Tá je určená pre užívateľom napísanú softvérovú časť aplikácie, pomocou ktorej možno v hostiteľskom počítači spracovávať prenesené dáta.



Obr. 8: Vrstvy štruktúry NetCOPE

4.2 Firmware NetCOPE

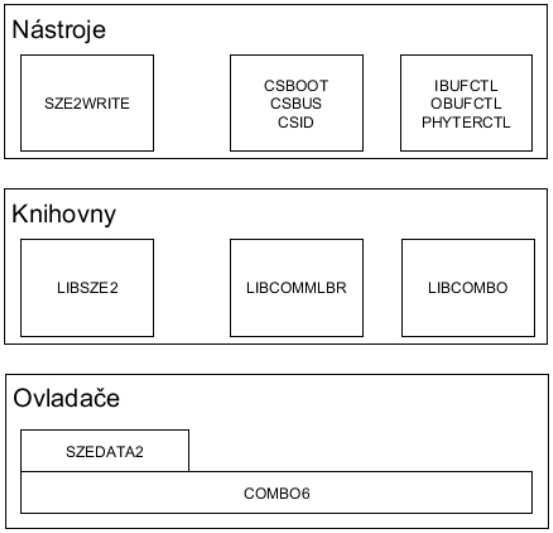
Časť frameworku NetCOPE určenú pre karty Combo tvorí firmware, ktorého všeobecné bloková schéma je znázornené na Obr. 9. Užívateľom vyvíjaná aplikácia sa implementuje do aplikačného modulu, pričom modifikácie iných blokov niesú nutné. Za týmto účelom je vo firmware predpripravená štruktúra s radičmi a ovládačmi, pomocou ktorých je možné z aplikačného modulu ľahko pristupovať k pamäťovým obvodom osadeným na karte, paketom zo sieťového prevádzky a podobne. Firmware NetCOPE ďalej zabezpečuje komunikáciu medzi aplikačným modulom a PCI Express rozhraním (hostiteľským počítačom).



Obr. 9: Všeobecná bloková schéma firmvéru NetCOPE pre karty Combo

4.3 Software NetCOPE

Komunikáciu medzi aplikáciou bežiacou na hostiteľskom počítači a firmvérom NetCOPE bežiacom na karte Combo zabezpečuje softvérová časť frameworku NetCOPE. Jej architektúra zložená z troch vrstiev je znázornená na Obr. 10.



Obr. 10: Architektúra softvéru NetCOPE pre hostiteľský počítač

Najnižšia vrstvu softvérovej časti frameworku tvoria ovládače, ktoré zaisťujú základnú komunikáciu medzi operačným systémom hostiteľského počítača a kartou Combo s firmvérom NetCOPE. Druhú vrstvu tvoria knižnice, ktoré abstrahujú užívateľskú softvérovú aplikáciu z najvyššej vrstvy a odovzdávajú dáta ovládačom. Najvyššiu vrstvu softvérovej časti NetCOPE tvorí užívateľom vytvorená softvérová aplikácia, ktorá slúži na spracovanie informácií zaslaných z hardvérovo akcelerované aplikácie v obvode FPGA na karte Combo. Pre zjednodušenie vývoja softvérové aplikácie, testovanie alebo ladenie možno použiť nástroje, ktoré sú súčasťou softvérovej časti NetCOPE.

4.4 Komunikácia NetCOPE

Komunikácia medzi firmvérom a softvérovou aplikáciou prebieha vo frameworku NetCOPE cez rozhranie PCI Express. Pre zjednodušenie a unifikáciu komunikácie cez rozhranie PCI Express pre rôzne typy kariet Combo sú vo frameworku vytvorené zbernice MI32, Internal Bus a protokol FrameLink. MI32 je zbernica s nízkou priepustnosťou, možnosťou adresovania a ľahkou réžiou vhodná pre občasný prenos registrov malej veľkosti. Internal Bus je zbernica s vysokou priepustnosťou, možnosťou adresovania a zložitou réžiou vhodná pre prenos konkrétnych veľkých objemov dát. Protokol FrameLink má vysokú priepustnosť bez možnosti adresovania a je vhodný pre kontinuálne prenosy veľkých objemov dát. Potrebám diplomovej práce na prenos malých registrov medzi firmvérom a softvérom s možnosťou ich adresovanie najlepšie zodpovedá

zbernice MI32. Pri praktickej realizácii projektu bola využitá práve ona. V ďalšom texte preto budú zbernice Internal Bus a protokol FrameLink opomenuté.

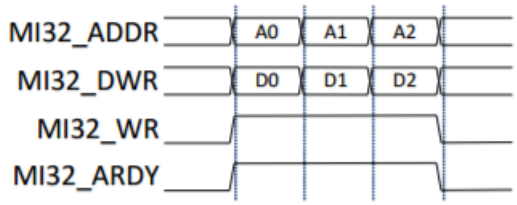
4.4.1 Zbernica MI32

Komunikácia po zbernici MI32 sa z hľadiska softvérovej aplikácie skladá z operácií čítania a zápisu registrov o pevnej dĺžke 32 bitov. Zápis na zbernici MI32 znamená prenos registra s riadiacimi signálmi zo softvérovej aplikácie do firmvéru. V prípade diplomovej práce môže byť príkladom zápisu na zbernici MI32 vyslanie registra s požiadavkami na generovanie ľubovoľného bloku paketu. Čítanie zo zbernice MI32 znamená odosielanie registri s informáciami z firmvéru do softvérovej aplikácie. Príkladom čítania zo zbernice MI32 je dotaz na register so stavom testu pamäťového obvodu. Prehľad signálov používaných pri komunikácii po zbernici MI32 je uvedený v Tab. 1.

Tab. 1: Prehľad signálov pre komunikáciu po zbernici MI32 [8]

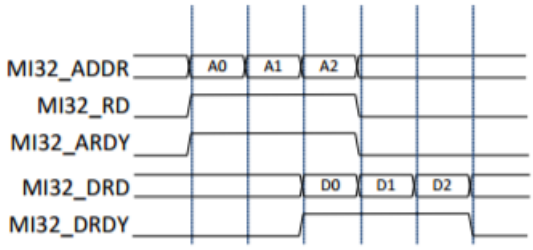
Signál	Šírka v bitoch	Vlastník	Určený pre operaciu	Význam
MI32_ADDR	32	software	čítanie a zápis	adresa registru
MI32_WR	1	software	zápis	požiadavok na operáciu "zápis"
MI32_DWR	32	software	zápis	obsah registra pre zápis
MI32_RD	1	software	čítanie	požiadavok na operáciu "čítanie"
MI32_DRD	32	firmware	čítanie	obsah registra pre čítanie
MI32_DRDY	1	firmware	čítanie	platné data pre čitanie
MI32_ARDY	1	firmware	čítanie a zápis	firmvér zaznamenal požiadavok na čitanie/zápis vrátane adresy
MI32_BE	4	software	čítanie a zápis	platnosť bytov v registry MI32_DWR, MI32_DRD; nepoužíva sa

Komunikácia po zbernici MI32 je vždy inicializovaná softvérovou aplikáciou, ktorá vystaví požiadavku na zápis (log. 1 na signálu mi32_wr) alebo čítanie (log. 1 na signále mi32_rd). Súčasne s touto požiadavkou je vystavená adresa mi32_addr. Tá u diplomovej práce určuje konkrétne pamäťový obvod a typ registra. Ak softvér vykonáva operáciu zápisu, vystaví súčasne ešte obsah registra pre zápis na signálu mi32_dwr. Firmvér na zaznamenanie a spracovanie požiadavky reaguje vystavením signálu mi32_ardy. Príklad zápisu troch registrov na zbernici MI32 je znázornený na Obr. 11.



Obr. 11: Zápis troch registrov na zbernici MI32 [8]

Ak softvér požaduje čítanie zo zbernice MI32, odpovedá firmvér na zaznamenanie požiadavke vystavením signálu mi32_ardy. Následne sa požiadavka na čítanie spracuje. Akonáhle firmvér na signály mi32_drd vystaví register pre čítanie, iniciuje túto skutočnosť aktívnym signálom mi32_drdy. Príklad čítania troch registrov zo zbernici MI32 je znázornený na Obr. 12.



Obr. 12: Čítanie troch registrov zo zbernice MI32 [8]

Okrem všetkých vyššie zmieňovaných signálov sa v špecifikácii zbernice MI32 nachádza ešte signál mi32_be o veľkosti 4 bity. Ten určuje, ktoré zo štyroch bytov signálu mi32_dwr pri zápise (Resp. Mi32_drd pri čítaní) sú platné. V praxi sa napriek priebehom na Obr. 11 a Obr. 12 tento signál ignoruje a je vždy nastavený na hodnotu 0xF, kedy sú platné všetky byty. V súlade s touto konvenciou sa adresa zbernice MI32 inkrementuje po násobkoch štyroch. Adresy zbernice MI32 (bez absolútneho offsetu) sa preto inkrementuje podľa vzoru 0x00, 0x04,0x08, 0x0C, 0x10, 0x14 atď.

4.5 Ovládače pamätí

Jednou z častí firmvéru NetCOPE sú ovládače pamäťových obvodov (tzv. Wrappery). Wrapper je modul, ktorý je jedným rozhraním pripojený priamo k pamäťovému obvodu a vykonáva operácie čítania a zápisu z / do pamäťového obvodu pri dodržaní všetkých potrebných časovaniach a vystaveniach riadiacich signálov. Druhým rozhraním je wrapper pripojený k hardvérovému aplikačnému modulu, od ktorého prijíma pokyny pre vykonanie operácií. Komunikačný protokol a signály tohto rozhrania sú navrhnuté tak, aby bola obsluha Wrapper z hardvérového aplikačného modulu čo najjednoduchšia. Súčasťou Wrapper-a môže byť asynchrónne oddelenie medzi rozhraním pre pripojenie k hardvérovému aplikačnému modulu a rozhraním pripojenom k pamäťovému obvodu. V takom prípade nemusia byť požiadavky na čítanie a zápis z hardvérového aplikačného modulu a rozhranie pripojené k pamäťovému obvode vzájomne synchrónne a používať rovnakú hodinovú doménu.

Wrappery pre pamäte RAM na kartách Combo sú vytvorené pomocou nástroja Memory Interface Generator (MIG), ktorý je súčasťou vývojových prostredí Xilinx ISE Design Suite a Xilinx Vivado Design Suite používaných pri práci s frameworkom NetCOPE. V niektorých prípadoch môžu byť na rozhraní Wrapper vo frameworku NetCOPE pripojené ďalšie moduly, ktoré komunikačné rozhranie

pamäte dodatočné upravujú (modifikácie používateľského rozhrania, asynchrónne oddelenia a pod.). Pre praktickú realizáciu projektu testera pamäťou RAM nie je nutné poznať vnútorné obvodové zapojenie Wrapper a časti jeho rozhrania pripojenú medzi wrapper a samotný pamäťový obvod. Podstatná je len znalosť časti rozhrania Wrapper, ktorá je určená pre pripojenie k hardvérovému aplikačnému modulu

5. Teória znáhodňovania

Elektronické obvody sú, ak sú navrhnuté správne, deterministické. Alebo by mali byť. To znamená, že ak je v čase Tn na vstupoch určitá kombinácia dát a zároveň poznáme vnútorný stav obvodu (čo tu, na rozdiel od sveta elementárnych častíc, dokážeme), tak môžeme presne povedať, aký stav bude v čase Tn+1. Vďaka tomu elektronické obvody fungujú tak, ako majú - ak sú teda správne navrhnuté, správne zapojené, správne prevádzkované, správne odtienené od vonkajších vplyvov. Napriek tomu ale niekedy potrebujeme niečo „znáhodnit".

5.1 Generátor náhodných čísel

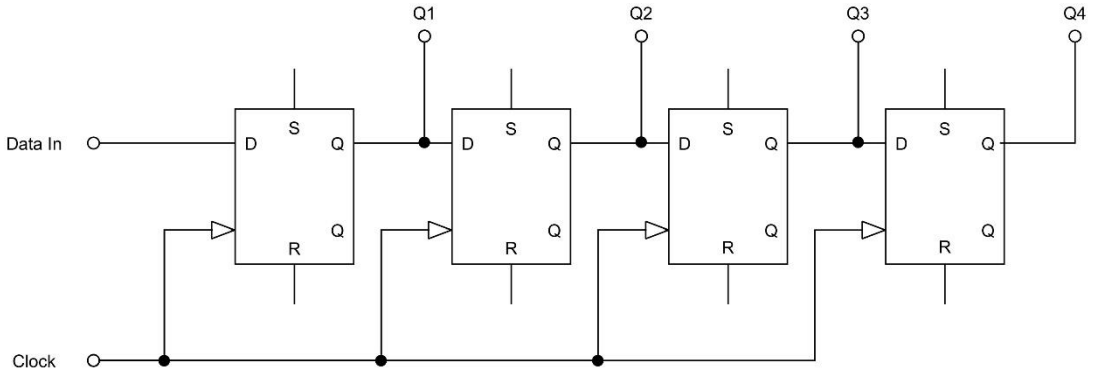
Jedna možnosť je použiť naozajstný generátor náhodných čísel, čo je väčšinou nejaké hardvérové zariadenie, ktoré generuje náhodný signál na základe nejakého fyzikálneho javu. Napríklad rozpad rádioaktívne látky, meranie šumu, alebo vstupu od užívateľa. Rozpad rádioaktívnych látok nebýva úplne bežne dostupný. S šumom je to lepšie. Principiálne vezmeme nejakú súčiastku, ktorá generuje šum, napríklad polovodičovú diódu, výstupný šum riadne zosílime, vzorkujeme, prevádzame do digitálnej podoby a podľa najmenej významného bitu určujeme aktuálnu hodnotu náhodného signálu. Nevýhoda je, že šum je ovplyvniteľný zvonku, napríklad sa nám môže v obvode indukovať nejaký rádiový signál. Na druhej strane môžeme takéto ovplyvnenie považovať za "chaos zvyšujúci faktor ...". Ak to zapojenie umožňuje, je dobré použiť vstup od užívateľa, napríklad stlačenie tlačidiel alebo pohyby myši, a využiť to, že intervaly stlačení sú rádovo nižšie ako rýchlosť behu počítadla a prichádzajú, z hľadiska systému, naozaj v náhodné okamihy. Problém nastane, keď potrebujeme náhodné čísla „neustále", zatiaľ čo vstup od užívateľa príde z hľadiska systému „za extrémne dlhú dobu".

Využíva sa techniky, kedy sa síce zahrňujúceho impulzy, ale hodnoty nejdú po sebe v očakávanom poradí 1, 2, 3, 4, 5, 6 ..., ale napr. pri trojbitovom čítači ako 1, 4, 6, 7, 3, 5, 2 (hodnota 0 sa nevyskytuje). Takúto postupnosť odhalíme hladko, ale ak použijeme šestnásťbitový, ktorý má na výstupe čísla 1 až 65535 v pseudonáhodnej poradí postupnosť sa objaví znovu, ale neskôr. Ale nič nám nebráni použiť počítadlá 32 bitov, 60 bitov, 128 bitov kludne 168 bitov. V takom prípade sa postupnosť opakuje po cca 1050 hodnotách. Ak použijeme napríklad 50MHz hodinový signál, objaví sa rovnaká postupnosť po 7,4 × 1042 sekundách, čo je 2.37 × 1035 rokov. [4]

5.2 Posúvný register

Posuvný register si predstavme ako sadu registrov, zapojených za sebou tak, že s každým pulzom hodín sa informácie z registra N posunie do registra N+1. Viz. Obr. 13. Pri prevode paralelných dát na sériové nahráme do všetkých registrov naraz požadované dáta, a potom na výstupe z posledného registra čítame bit po bite. Pri opačnom prevode posielame sériové dáta na vstup posuvného registra,

s každým hodinovým pulzom sa posunú o jednu pozíciu, a akonáhle máme načítaný kompletný bajt, prečítame si ho z jednotlivých registrov.[5]



Obr. 13: Posuvný register

5.2.1 LFSR

Kruhový posuvný register vznikne tým, že výstup zavedieme späť na vstup. Špeciálnym prípadom kruhového registra je kruhový posuvný register s lineárnou spätnou väzbou, čiže LFSR (Linear feedback shift register). Ak napríklad na predchádzajúcom obrázku zavedieme negovaný výstup Q4 na vstup "Data In", získame štvorbitový Johnsonov čítač. Kruhové posuvné registre (tiež "kruhové čítače") môžeme zapojiť aj zložitejšie - vstup budíme nie samotným výstupom, ale signálom, zloženým z viacerých bitov.

Predstavme si, že u vyššie uvedeného čítača privedieme na vstup Data In signál, ktorý vznikne ako Q1 xor Q4. U štvorbitového čítača môžeme zvoliť s dvouvstupovým XOR hradlom šesť kombinácií spätnej väzby (1,2), (1,3), (1,4), (2,3), (2,4), (3,4). Niektoré z nich vedú k veľmi krátkym cyklom (napríklad 1,2), iné k najdlhším možným (1,4). Praktické na LFSR je, že na dosiahnutie najdlhšieho možného cyklu (2^n-1 stavov) stačí dvoj-, či štvorbitovými hradlami. Aj LFSR o dĺžke 168 bitov vytvoríme jednoducho, zavedením spätnej väzby signálom (Q151 xor Q153 xor Q166 xor Q168 - číslujeme od 1).Existujú dva spôsoby zapojenia. Prvým je "many-to-one", čiže "veľa do jedného" - niekoľko výstupov sa XORuje do jedného vstupu. V našom prípade ak chceme dosiahnúť rýchleho generovania nieje toto usporiadanie veľmi vhodné. Hlavný dôvod je, že sa prejavi oneskorenie pri viacúrovňovom XORování. V takom prípade môžeme architektúru „many-to-one“ nahradiť architektúrou „one-to-many“, kde sa výstup (najvyšší bit) primiešava do niekoľkých miest v reťazi klopných obvodov. Miesta, kde má dôjsť k XORování, sú rovnaké ako u architektúry many-to-one, sekvencie zostane rovnako dlhá, ale postupnosť hodnôt bude iná. [6]

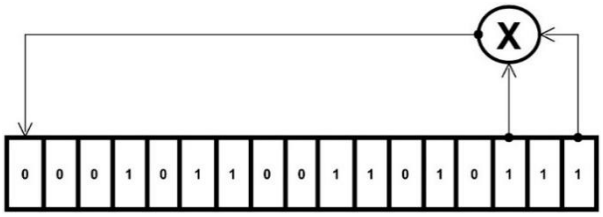
```
architecture one2many of lfsr is
signal d: std_logic_vector(8 downto 1) := (others=>'1');
begin
	process(clk) is
	begin
		if rising_edge(clk) then
			d(1) <= d(8);
			d(2) <= d(1);
			d(3) <= d(2) xor d(8);
			d(4) <= d(3) xor d(8);
			d(5) <= d(4) xor d(8);		-- ôsmy výstup sa XOR-uje s
							   výstupom na pozícii 2, 3 a 4.
			d(6) <= d(5);
			d(7) <= d(6);
			d(8) <= d(7);
		end if;
	end process;

	q<=d(8);
end architecture;
```

Príklad kódu 1.: 8bitový LFSR typu „one-to-many“

6. Návrh generátora náhodných čísel

Tento modul je založený práve na spomínanom princípe LFSR. LFSR je teda posuvný register, ktorého vstupný bit je lineárnou funkciou predchádzajúceho stavu. Jediná lineárna funkcia samotných bitov je XOR, teda jedná sa o posuvný register, ktorého vstupný bit je riadený XOR-om vybraných bitov celkovej hodnoty posuvného registra.

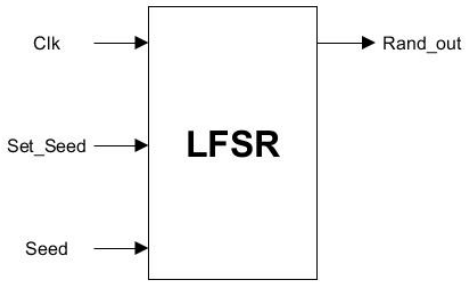
Počiatočná hodnota LFSR sa nazýva "seed" teda semiačko, a pretože operácia registra je deterministická, prúd hodnôt produkovaných registrom je predurčený jeho aktuálnym (alebo predchádzajúcim) stavom. Rovnako tak preto, že register má konečný počet možných stavov, je jasné, že nakoniec sa cyklus začne opakovať. Avšak, LFSR s dobre zvolenou funkciou spätnej väzby môže produkovať sled bitov, ktoré sa objavia náhodne, a ktorý má veľmi dlhý cyklus. Bitové pozície, ktoré majú vplyv na ďalšie stavy sa v terminológií posuvných registrov nazývajú „taps". Maximálna doba cyklu, počas ktorej sa sekvencia opakuje je 2^{n-1} pre n-bitový LFSR register. Hodnoty „taps" teda musia byť vybrané obozretne pre dosiahnutie maximálnej doby cyklu opakovania sa hodnôt. Boli vytvorené tabuľky, kde k registru o dĺžke „n" bitov odpovedajú hodnoty pozícií vhodných pre XOR-ovanie. Použitý bol dokument od firmy Xilinx pre získanie týchto hodnôt.[6] Dizajn použitého registra je navrhnutý pre spracovanie 8 bitov. Príklad jednoduchého XOR-ovania hodnôt LFSR registra „one-to many" je na Obr. 14.



Obr. 14: Použitie XOR funkcie na LFSR

6.1 Popis funkcie bloku

Blokový diagram na Obr. 15 výsledného LFSR obsahuje 3 vstupy. Prvým je hodinový signál. S každou nábežnou hranou tohto signálu dostaneme náhodné číslo na výstupe „rand_out". „Set_seed" je std_logic signál, ktorý je použitý ako inicializačný, teda je ním ovládaný signál „seed". A to tak, že ak je hodnota „set_seed" log1 tak je aktuálna hodnota registra nastavená práve na hodnotu dostupnú na vstupe „seed". Všetko je realizované synchrónne.



Obr. 15: Bloková schéma použitého LFSR

Výsledných 8 bitov výstupu Rand_out je vytvorených postupným spojením pomocných, respektíve, dočasných vektorov, pri procese generovania náhodného výstupu. Viz. Príklad kódu 2.

```
variable rand_temp : std_logic_vector (7 downto 0):=(0 => '1',others => '0');
variable temp : std_logic := '0';
variable rand : std_logic_vector (7 downto 0);
```

Príklad kódu 2.: Pomocné premenné

Prvá a zároveň aj základná je premenná „rand_temp“. Obsah tohto vektora je následne upravovaný a na záver načítaný do hodnoty „rand_out“. Prvotné nastavenie je ľubovoľné a nemá vplyv na kvalitu výslednej hodnoty. Tu je použitý vektor s logickou 0 na mieste LSB a na zvyšných pozíciách logická 1. Vektor hodnôt „rand“ je dôležitým prvkom vnorenia skutočnej „náhody“ do procesu. Ako zo zápisu vyplýva, je definovaný len jeho rozmer, obsah nie je definovaný a teda je vyplnený na základe aktuálneho stavu pamäti, vnútornej entropie , a hardvéru. Je to teda 8 úplne náhodných bitov zložených z logických jednotiek a núl, ktorú my ako užívatelia neovplyvníme. Z tejto vlastnosti ale vyplýva fakt, že pre skutočné skontrolovanie výstupov je nutné modul implementovať na fyzickú kartu, aby sa tento efekt prejavil. Premenná „temp“ je pri inicializovaní vynulovaná a budú do nej následne vkladané hodnoty po XOR-ovaní.

```
temp := rand(7) xor rand(5) xor rand(4) xor rand(3);
rand_temp(7 downto 1) := rand_temp(6 downto 0);
rand_temp(0) := temp;
rand_out <= rand_temp;
```

Príklad kódu 3.: Vytvorenie Rand_out vektora

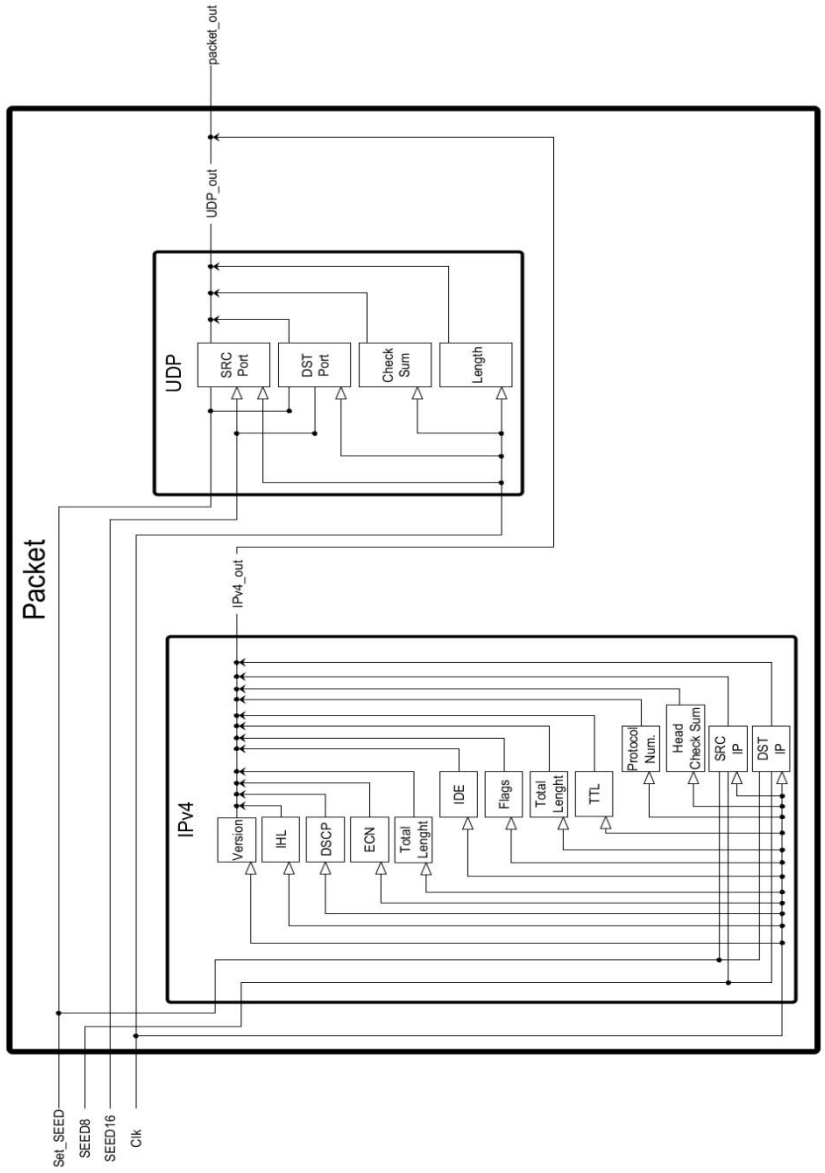
Výsledný stav premennej „rand_out“ je teda tvorený z vektoru „temp“, ktorý je výsledkom xorovania vybraných pozícií vektoru „rand“ a je vložený na pozíciu LSB (0-ltá pozícia). Nasleduje 7bitov posunutého vektora „rand_temp“ . Viz Príklad kódu 3. Tento cyklus sa stále opakuje za podmienok, že do systému vstúpi nábežná hrana hodinového cyklu a hodnota „set_seed“ má hodnotu log 1.

7. Štruktúra generovaného paketu

Ako základná štruktúra pre generovaný paket, bol zvolený formát paketu IP verzie 4. Obsahujúci službu transportnej vrstvy využívajúc UDP datagram, s malými modifikáciami pre potreby simulácie a vyhodnocovania výsledkov. Celková schéma princípu naviazania jednotlivých blokov je na Obr. 16. Výsledný vektor je výsledkom spojenia nižších vrstiev projektu, konkrétne ip_out (výstup z IP bloku) a udp_out (výstup z UDP bloku). Veľkosť vektoru „packet_out“ je 224 bitov. Jednotlivé bitové pozície sú obsadené na základe bitového priradenia.

```
packet_out(223 downto 64) <= ip_out( 159 downto 0);
packet_out( 63 downto 0) <= udp_out( 63 downto 0);
```

Príklad kódu 4.: Bitové priradenie vektoru packet_out

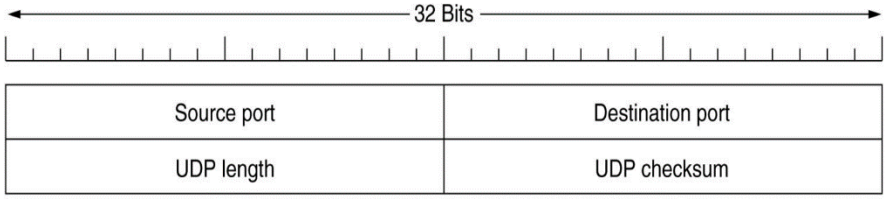


Obr. 16: Navrhnutá logická schéma paketu

7.1 UDP

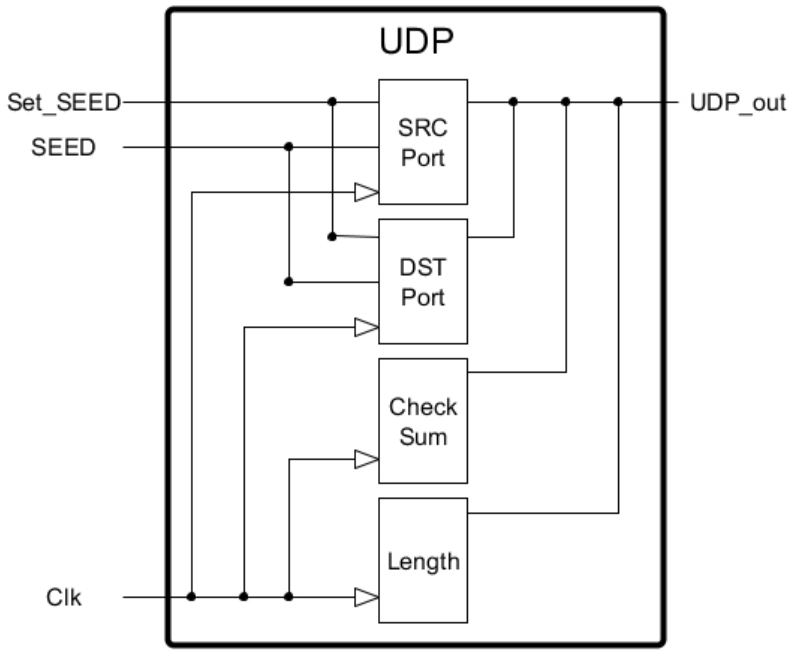
Protokol UDP slúži k negarantovanému prenosu dát medzi aplikáciami jednotlivých staníc. Negarantuje ani doručenie datagramov príjemcovi a ani negarantuje, že datagramy budú príjemcovi doručené v takom poradí, v akom boli vyslané. Používa sa preto u aplikácií, ktoré nemajú vysoké nároky na spoľahlivosť, ale zároveň kladú vysoké nároky na včasnosť doručenia. Takýmito aplikáciami sú typicky aplikácie využívajúce prenos obrazov a zvukov (napr. videokonferencie).

Druhou triedou aplikácií, pre ktoré je protokol UDP výhodný, sú aplikácie s prenosom krátkych správ (napr. prideľovanie IP adries protokolom DHCP). Štruktúra hlavičky datagramu je na Obr. 17. Vidíme, že hlavička UDP datagramu je veľmi jednoduchá. Význam jednotlivých polí je nasledujúci. V tomto projekte je pole Checksum nahradené poľom Sequence number. Teda poradovým číslom, s ktorým bol paket vyslaný.[10]



Obr. 17: Hlavička UDP paketu [10]

Logická schéma návrhu entít vo vývojom prostredí je znázornená na Obr. 18. Výstupy z jednotlivých blokov sú následne bitovo priradene do celkového výstupného vektora „UDP_out“ o celkovej veľkosti 64bitov pre ďalšie spracovanie aplikácie.



Obr. 18: Navrhnutá logická schéma UDP záhlavia

7.1.1 Zdrojový port a cieľový port

Zdrojový port identifikuje aplikáciu, ktorá paket odoslala. Toto pole označuje port odosielateľa a mal by byť považovaný za port, na ktorý odpovedať v prípade potreby. Ak je komunikácia len jednosmerná (tj. odosielateľ nedostáva odpovede), nastavuje sa na hodnotu „0“. Ak je zdrojovým uzlom klient, číslo portu je pravdepodobne len momentálne a mení hodnotu. Ak je zdrojovým uzlom server, číslo portu bude z rozsahu takzvaných „well known ports“, teda porty, ktorých čísla sú vopred priradené pre použitie danej služby.

Cieľový port je povinný, definuje cieľovú aplikáciu, pre ktorú je paket určený. Označuje port prijímača. Podobne ako zdrojové číslo portu, v prípade, že klient je cieľový hostiteľ potom číslo portu bude pravdepodobne pominuteľné, a v prípade, že cieľový hostiteľ je server, potom číslo portu bude priradené na základe služby.

Funkcia oboch entít je totožná. Výstupný vektor je popísaný v oboch prípadoch ako std_logic_vector s veľkosťou 16 bitov. Hodnoty taps pre výpočet XORovaním boli zvolené 15, 14, 12, 3 (Indexované od 0). To nám zaručuje $2^{16} = 65,536$ výstupných hodnôt, teda maximálny počet, aký je možné pri 16 bitovom registri dosiahnuť.[12]

7.1.2 Dĺžka datagramu

Pole, ktoré určuje dĺžku v bajtoch UDP hlavičky a dát UDP. Minimálna dĺžka je 8 bajtov, pretože to je dĺžka hlavičky. Veľkosť poľa nastavuje teoretický limit na 65535 bajtov (8 bajtov hlavička + 65,527 bajtov dát) pre UDP datagram. Praktický limit pre dĺžku dát, ktorá je uložená u podkladového protokolu IPv4 je 65,507 bajtov (65.535 - 8 bajtov UDP hlavička - 20 bajtov hlavičky IP). Pri IPv6 jumbograms, je možné mať UDP pakety o veľkosti väčšej než 65,535 bajtov. RFC 2675 určuje, že dĺžka poľa je nastavená na nulu, ak je dĺžka UDP hlavička a UDP dát je väčšia ako 65.535.

V projekte je obsah polia určujúceho dĺžku paketu (Length) nastavené na pevnú hodnotu, teda 8. Odpovedajúc veľkosti hlavičky 64bitov bez pridaných dát viz Obr. 19.

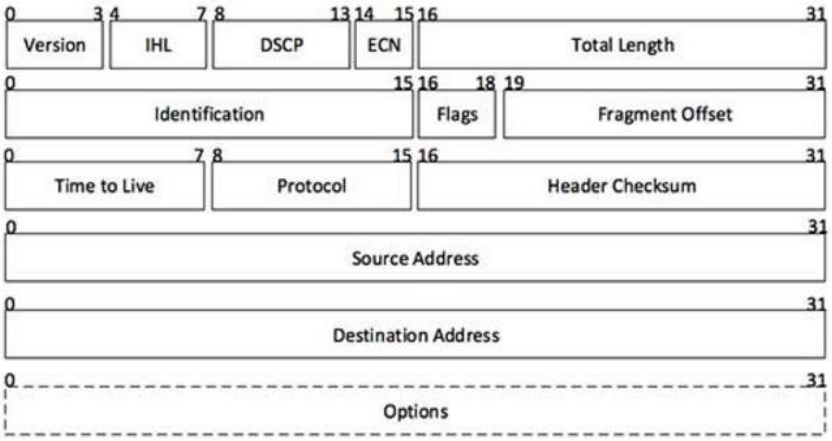
7.1.3 Kontrolný súčet

Toto pole zvyčajne obsahuje kontrolný súčet vypočítaný z celého datagramu, avšak aj z časti IP hlavičky. Tvorcovia protokolu v tomto ohľade nerešpektovali zásadu nezávislosti dátových jednotiek v jednotlivých vrstvách architektúry a dôsledkom potom bola nutnosť spoločne s nástupom protokolu IPv6 doplniť v staniciach softvér pre výpočet kontrolného súčtu - hlavičky oboch protokolov sú rôzne. Kontrolný súčet však nie je povinný. Transportná vrstva tým ušetrí čas, ale bezchybnosť dát potom musí riešiť vrstva aplikačné. Rezignáciu na kontrolný súčet odosielateľ oznamuje tým, že v tomto poli uvedie samé nuly. V

tejto súvislosti je potom potrebné ošetriť situáciu, keď výsledkom kontrolného súčtu je práve oných 16 nulových bitov. Odosielateľ v takomto prípade musí výsledok kontrolného súčtu invertovať, tj. v poli Kontrolný súčet bude obsahovať 16 jedničiek. [12]

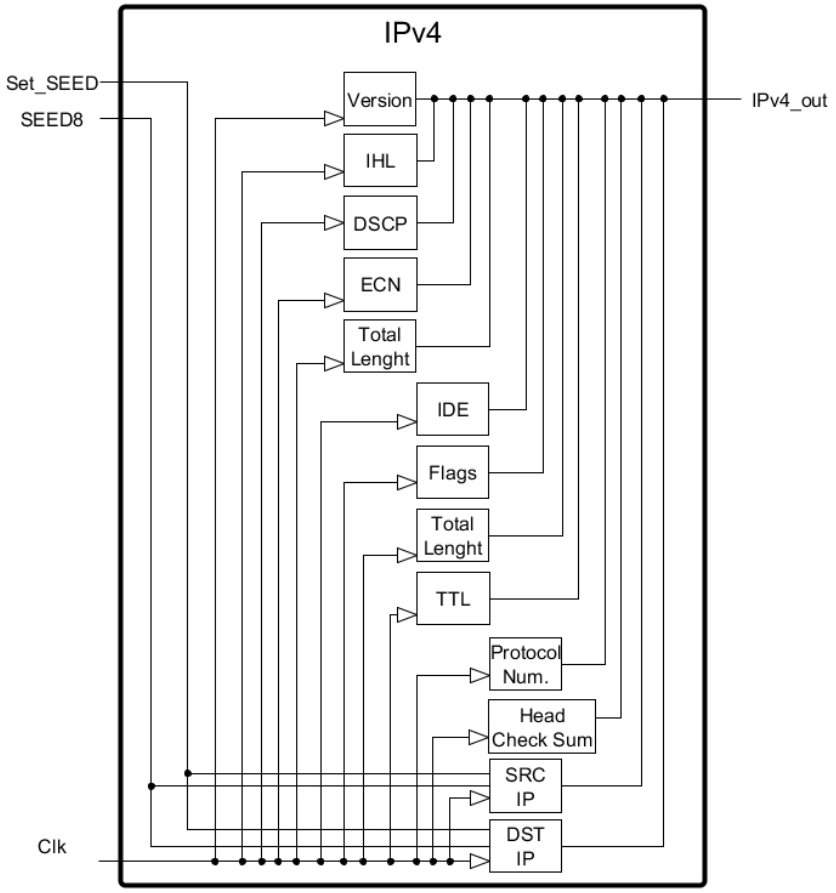
7.2 Internet protocol

Protokol IP vo verzii 4 je daný štandardom RFC 791 [9]. Základnou dátovou jednotkou tohto protokolu je blok bajtov, ktorý budeme nazývať paket. Štruktúra hlavičky paketu je uvedená na Obr. 19. Hlavička obsahuje služobné údaje a v tele paketu sa nachádzajú dáta, ktoré sú určené k prenosu (zvyčajne dátová jednotka transportnej vrstvy). Každý z prvých piatich riadkov obrázku reprezentuje reťazec 32 bitov, pričom v každom poli paketu je uvedený názov tohto poľa a jeho dĺžka v bitoch. V prvom riadku paketu sa nachádzajú nasledujúce položky. [12]



Obr. 19: Štruktúra hlavičky IP paketu

Logická schéma návrhu entít vo vývojom prostredí je znázornená na Obr. 20. Výstupy z jednotlivých blokov sú následne bitovo priradene do celkového výstupného vektora „IPv4_out“ o celkovej veľkosti 160 bitov pre ďalšie spracovanie aplikácie.



Obr. 20: Navrhnutá logická schéma IPv4 záhlavia

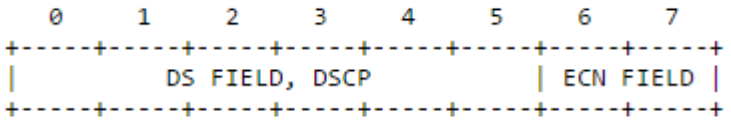
7.2.1 Version / IHL

V prípade IP verzie 4 sa tu nachádza číslo $4_{10} = 0100_2$. Dĺžka hlavičky: Údaj v tomto poli vyjadruje celkovú dĺžku hlavičky v násobkoch štyroch bajtov (32 bitov). Toto pole je tu z toho dôvodu, že dĺžka hlavičky nie je pevná a môže sa pohybovať v rozpätí od 20 B do 60 B. Minimálna hodnota poľa IHL je teda 5_{10} respektíve 0101_2 (5 × 32 bitov = 160 bitov = 20 bajtov.). Príjemca môže z poľa IHL zistiť, kde hlavička končí a kde začína telo paketu. Z toho vyplýva priradený výstupný vektor „version_num“ s hodnotou $V=0100_2$ a priradenia hodnoty pre pole IHL a výstupný vektor IHL_num rovný 0101_2. [12]

7.2.2 DSCP / ECN

Pole „Differentiated Services Code Point“, teda DSCP, bolo pôvodne definované ako pole ToS (Type of service). DSCP sa riadi podľa RFC 2474 (neskôr aktualizované s RFC 3168 a RFC 3260) pre technológiu DiffServ. Nové technológie si vyžadujú poskytovanie dát v reálnom čase a práve k tomu pomáha pole DSCP (na príklad VoIP). Odosielateľ v tomto poli uvádza svoje preferencie ohľadom požiadaviek na zaobchádzanie s paketom. Pomocou 6bitov uvádza, či pri prechode paketu sietí preferuje spoje s minimálnym oneskorením, maximálnou prenosovou

rýchlosťou, či najvyšší spoľahlivosťou. Na Obr. 21 je vidieť rozloženie bitov. Ich význam je vysvetlený v tabuľke Tab.2. [11] [13]



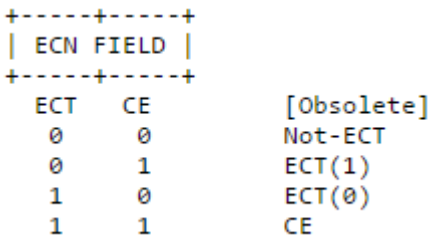
Obr. 21: Pole DSCP a ECN [8]

Tab. 2: Význam bitov poľa DSCP podľa RFC 971 [9]

Číslo bitu	Popis	Hodnota
1 - 3	Priorita paketu	0-7 (0=min)
4	Oneskorenie	0 = Normálne / 1 = Nízke
5	Priepustnosť	0 = Normálne / 1 = Vysoká
6	Spoľahlivosť	0 = Normálne / 1 = Vysoká

Explicit Congestion Notification (ECN), povoľuje end-to-end notifikáciu o preťažení bez nutnosti zahadzovať pakety. ECN je doplnkové pole, ktoré sa využíva len v prípade, že to oba koncové uzly podporujú a sú ochotné to používať. Pozostáva len z dvoch bitov teda 4 kódové slová (00_2 -> 11_2) [Obr.22]. Jeden bit popisuje ECT (ECN-Capable Transport) a druhý bit popisuje CE (Congestion Experienced).

Použitie kódového bodu CE s ECN umožňuje prijímaču prijímať aby sa zabránilo nadmernému omeškaniu v dôsledku retransmisie po stratách paketov.



Obr. 22: Pole ECN [9]

7.2.3 Total Length

V tomto poli je uvedená dĺžka celého paketu v bajtoch. Vektor pozostáva zo 16tich bitov. Z dĺžky tohto poľa vyplýva, že maximálna dĺžka paketu je $216 - 1 = 65535\ B$. Pakety o takejto dĺžke sú však nepraktické a tak sa používajú kratšie. Ak si komunikujúce počítača nedojednajú inú maximálnu hodnotu,

tak odporúčaná maximálna dĺžka generovaných paketov by mala byť 576 bajtov. Hodnota poľa v projekte je nastavená na 160_{10}, čo odpovedá hlavičke.

7.2.4 Identification / Flags / Fragment Offset

Nasledujúce 3 polia popisujú fragmentáciu paketov. V prípade fragmentácie paketu sa údaj z tohto poľa nachádza v rovnakom poli každého z jeho fragmentov. Adresát môže podľa týchto identifikátorov určiť, ktoré fragmenty patrí ktorému paketu. Na to slúži pole Identification pozostávajúce zo 16 bitov.

Stanice (tj. Počítač alebo smerovač) štandardne nastavuje veľkosť odosielaných paketov tak, aby sa zmestili do tela linkového rámca toho rozhranie, ktorým budú vysielané. Táto veľkosť sa označuje skratkou MTU ("Maximum Transmission Unit") a ako vieme z predchádzajúcej kapitoly, tak napríklad v sieťach Ethernet je štandardná hodnota MTU = 1500 B. Paket však môže byť na svojej ceste od odosielateľa k príjemcovi odovzdaný na vyslanie do spoja, ktorého MTU je menšia a paket teda nemožno preniesť vcelku. V tom prípade sa paket rozdelí na niekoľko kratších paketov (tzv. Fragmentov), pričom tieto fragmenty sú ďalej sieťou prenášané nezávisle. [12]

Príznaky fragmentácia (Flags):
- nevyužitý bit (rezerva),
- povolenie fragmentácia DF: 0 = paket možno fragmentovať, 1 = nedá,
- posledný fragment MF: 0 = áno, 1 = nasleduje ďalší fragment.

Hodnota poľa v projekte je nastavená na 010_2. To znamená, že sa paket nefragmentuje. Z jeho preddefinovanej veľkosti by to však ani nebolo potrebné.

Pole offsetu fragmentov sa meria v jednotkách s osembajtovými blokmi. Je dlhá 13 bitov a špecifikuje posun určitého fragmentu vzhľadom na začiatok pôvodného nefragmentovaného IP datagramu. Prvý fragment má odchýlku nula. To umožňuje maximálny posun (213 - 1) × 8 = 65 528 bajtov, ktorý by prekročil maximálnu dĺžku IP paketu 65 535 bajtov s dĺžkou záhlavia (65 528 + 20 = 65 548 bajtov). V projekte je hodnota offsetu nastavená na nulu. Konfigurácia pre nefragmenované pakety je popísaná nasledujúcim výrazom (rovnica (1)). [12]

$$(DF == 1) \;\&\&\; (MF == 0) \;\&\&\; (frag\_offset == 0) \qquad (1)$$

7.2.5 TTL

Osembitové pole doby času života (TTL), pomáha zabrániť pretrvávaniu datagramov (napr. v kruhoch) v sieti. Toto pole obmedzuje životnosť datagramu. Určuje sa v sekundách, ale časové intervaly kratšie ako 1 sekundu sa zaokrúhľujú na hodnotu 1. V praxi sa pole stalo počítaním skokov - keď datagram príde na smerovač, router znižuje pole TTL o jedno. Keď pole TTL narazí na nulu, smerovač zahodí paket a zvyčajne pošle odosielateľovi správu ICMP Time

Exceeded. Program “traceroute“ používa tieto správy ICMP Time Exceeded na vytlačenie smerovačov používaných paketmi na prechod od zdroja k cieľu. Dobu života paketu určuje jeho odosielateľ, pričom maximálna hodnota je 255 a odporúčaná 64. V projekte je hodnota náhodne nastavená na pevno na 10_{10}. Teda vygenerovaný paket sa po 10tich skokoch zahodí. [12]

7.2.6 Protocol

V tele paketu možno prenášať správy prakticky ľubovoľného protokolu sieťové, alebo iné vrstvy. Údaj v tomto poli určuje, ktorému konkrétnemu protokolu dáta prenášaná v poli Telo patrí. Uvedený parameter z IP paketu vytvára univerzálny prenosový kontajner sieťovej vrstvy. Napríklad pre protokoly ICMP, TCP a UDP sú hodnoty v tomto poli rovne číslam 1, 6 a 17. V projekte boli na generovanie zvolene UDP pakety → pole Protocol teda obsahuje hodnotu 17_{10}.

7.2.7 Header Checksum

Pri protokole IP sa bezchybnosť dát v tele paketu nekontroluje a toto pole obsahuje iba kontrolný súčet dát zo záhlavia. Ak hodnota tohto poľa nesúhlasí s výsledkom vlastného kontrolného súčtu príjemcu, tak je daný paket zničený. Každý smerovač musí pred odoslaním paketu vždy vypočítavať nový kontrolný súčet, pretože v záhlaví minimálne zmenil hodnotu TTL. V tejto súvislosti je ešte potrebné poznamenať, že sa nejedna o kontrolu pomocou cyklického kódu (tj. CRC), tak ako u protokolov linkové vrstvy. U protokolov rodiny TCP / IP sa používa algoritmus založený na aritmetike jednotkového doplnku. Zisťuje sa postupným sčítaním blokov s dĺžkou 16 bitov. Bloky sú v tomto súčte chápané ako celé čísla v aritmetike jednotkového doplnku, čo je jedna z foriem reprezentácie kladných a záporných čísel v počítači.

Postup je nasledujúci. Počiatočná hodnota kontrolného súčtu sa nastaví na Y = 000016, tj. na dva nulové bajty. Ak dĺžka kontrolovaných dát činí nepárny počet bajtov, tak sa dáta na konci doplní fiktívnom nulovým bajtom, ktorý sa neprenáša. Dáta sa rozdelí na bloky wi s dĺžkou 16 bitov (tj. Dvoch bajtov), kde i = 1 až N, pričom N je celkový počet blokov. Hodnotu kontrolného súčtu, tj. Y, získame tak, že bitový reťazec X po jednotlivých bitoch Invertuje. Ak by nám vyšlo, že Y = 000016, tak hodnotu Y upravíme na Y = FFFF16. Príjemca datagramu postupuje rovnako ako jeho odosielateľ a výsledok svojho výpočtu Y' potom porovná s prijatou hodnotou Y. V prípade ich zhody je datagram akceptovaný, v opačnom prípade je zlikvidovaný. Popísaný kontrolný súčet nemá tak dobré detekčné schopnosti ako cyklický kód, ale z historických dôvodov sa stále používa.

7.2.8 Source / Destination address

Štvrtý riadok hlavičky obsahuje IP adresu odosielateľa a piaty riadok obsahuje IP adresu príjemcu. IP adresy majú dĺžku 32 bitov a v systéme prepojených sietí ("Internetwork") jednoznačne určujú každú stanicu. [12]

V projekte je pole zdrojovej respektíve cieľovej adresy vyplňované privátnymi IP adresami triedy C. Tento rozsah je v rozmedzí od 192.168.0.0 do 192.168.255.255. Pre naše účely je však využitá len časť s maskou 255.255.254.0 (prefix /24), teda adresujeme len v priestore 192.168.1.x
($x \in X \mid X=\{<0;255>\}$).

7.2.9 Options

Pole „Options“ sa často nepoužíva. Hodnota v poli IHL musí obsahovať dostatok ďalších 32-bitových slov na uloženie všetkých možností. Zoznam možností môže byť ukončený pomocou možnosti EOL (End of Options List, 0x00). Je to potrebné iba vtedy, ak by sa koniec možností inak nezhodoval s koncom hlavičky. Možnosti, ktoré je možné vložiť do hlavičky, sú nasledujúce:

Tab. 3: Prehľad možností hodnôt poľa „Options“

Pole	Veľkosť (bit)	Popis
Kopírovanie	1	Hodnotu 1, ak je potrebné kopírovať všetky časti fragmentovaného paketu.
Trieda	2	Všeobecná kategória. 0 je pre "riadiace" možnosti a 2 je pre "ladenie a meranie". 1 a 3 sú vyhradené.
Číslo	5	Špecifikuje voliteľné položky
Dĺžka	8	Označuje veľkosť celého poľa (vrátane tohto poľa). Toto pole nemusí existovať pre jednoduché voliteľné položky.
Dáta	-	Veľkosť môže byť rôzna. Toto pole nemusí existovať pre jednoduché voliteľné položky.

8. Simulácia

Simulácia, ako prvý krok kontroly pri vývoji projektu, bola vykonaná na bežnom užívateľskom PC s verziou Vivada 2016.3 s nainštalovanou základnou licenciou. Z toho dôvodu nebolo možné využiť priamo pri simulácií ako cieľový hardvér, priamo čip Virtex 7. Nie je to však v tejto fáze nevyhnuté a pre účely simulácie úplne postačí ponúkaný čip Artix 7. Tento fakt poslúži dobre aj ako porovnanie zaťaženia aplikácie generátora paketov na slabšom hardvéry.

8.1 Použitý testbench

Pre potreby správnej reprezentácie výsledkov simulácie je potrebné vytvoriť testovaciu časť kódu, ktorá preverí integritu kódu a korektnosť vykonávaných operácií. V projekte bolo vytvorených viacero testovaní pre jednotlivé funkcie respektíve entity kde bola overená ich funkčnosť.

Popis použitého testbenchu pre top modul aplikácie generátora paketov je nasledujúci. Na začiatok je definovaný vstup pre jednotlivé vstupné porty komponenty, s ktorými budeme testovať funkčnosť zvoleného modulu.

```
architecture behavior of UDP_datagram_tb is
  signal clk,set_seed,reset : std_logic := '0';
  signal seed8 : std_logic_vector(7 downto 0) := (0 => '1',others => '0');
  signal seed16 : std_logic_vector(15 downto 0) := (0 => '1',others => '0');
  signal packet_out : std_logic_vector(223 downto 0);
  constant clk_period : time := 1 ns;
```

Príklad kódu 5.: Definovanie vstupov pre testbench top modulu

Ako prvé je dobré vybrané signály „nulovať“ priradením logickej hodnoty 0. Ďalej priradíme hodnoty signálu „seed8“ respektíve „seed16“, čo sú vstupné vektory 8 respektíve 16bitové. Ako posledné definujeme konštantu clk_period, čo nie je nič iné len hodnota hodinovej periódy udaná v nanosekundách. Viz Príklad kódu 5.

```
port (clk : in std_logic;
        set_seed : in std_logic;
        seed8 : in std_logic_vector(7 downto 0);
        seed16 : in std_logic_vector(15 downto 0);
        reset : in std_logic;
        packet_out : out std_logic_vector(223 downto 0)
      );
  end component;
```

Príklad kódu 6.: Priradenie portov

Následne zvolíme, či sa jedná o vstupný alebo výstupný port (in/out) a o aký typ signálu sa jedná spolu s definovanou bitovou dĺžkou. Viz Príklad kódu 6. Na záver je potrebné popísať ako bude nami vytvorený test prebiehať. Pre priradenie signálov, ktoré budeme využívať, použijeme takzvané mapovanie,

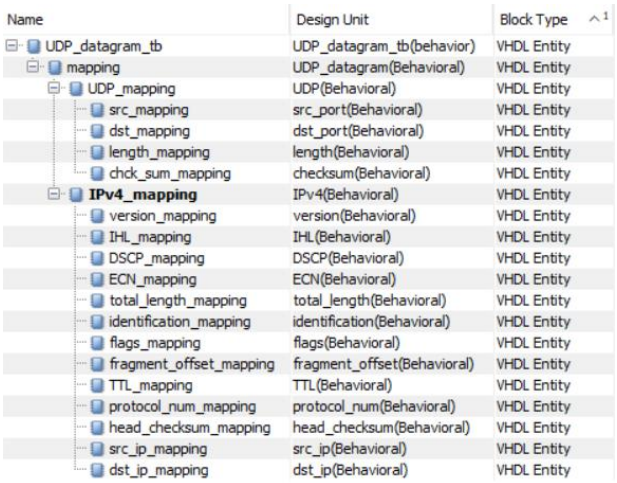
kde musíme signály vypísať v poradí ako sme ich definovali v Príklade kódu 6. Zvolíme počiatočnú hodnotu hodinového cyklu, odporúča sa začať v nule. V tejto hodnote ostane pol periódy (0,5ns) a zmení sa na logickú hodnotu „1“. Tým sme definovali ako sa bude meniť nábežná hrana clk. V ďalšom procese nastavujeme obdobne hodnotu set_seed, teda čas kedy chceme aby bol na vstup a následný výpočet použitý znova počiatočný vektor pre LFSR. V tomto prípade proces čaká 222ns aby nastavil logickú hodnotu na „1“ a začal sa proces vytvárania „náhody“.

```
begin
mapping: UDP_datagram port map(clk, set_seed, seed8, seed16, reset, packet_out);
clk_process :process
  begin
        clk <= '0';
        wait for clk_period/2;
        clk <= '1';
        wait for clk_period/2;
    end process;
  stim_proc: process
    begin
     wait for 222 ns;
       set_seed <= '1';
     wait for 1 ns;
       set_seed <= '0';
end process;
end behavior;
```

Príklad kódu 7.: Proces testovania

8.2 Výsledky simulácie

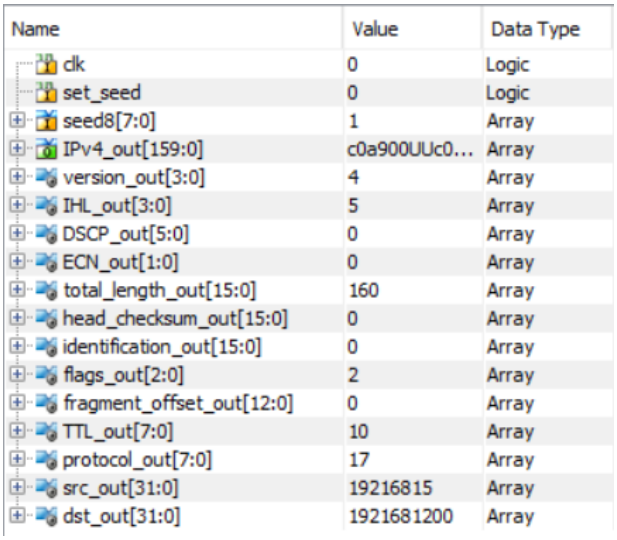
Pri skúmaní výsledkov simulácie, môžeme vybrať , ktorú úroveň a entitu aplikácie chceme skontrolovať. Vivado po prebehnutí úspešnej simulácie umožní náhľad na každú definovanú entitu zvlášť, ako je vidieť na Obr. 23.



Name	Design Unit	Block Type
UDP_datagram_tb	UDP_datagram_tb(behavior)	VHDL Entity
mapping	UDP_datagram(Behavioral)	VHDL Entity
UDP_mapping	UDP(Behavioral)	VHDL Entity
src_mapping	src_port(Behavioral)	VHDL Entity
dst_mapping	dst_port(Behavioral)	VHDL Entity
length_mapping	length(Behavioral)	VHDL Entity
chck_sum_mapping	checksum(Behavioral)	VHDL Entity
IPv4_mapping	IPv4(Behavioral)	VHDL Entity
version_mapping	version(Behavioral)	VHDL Entity
IHL_mapping	IHL(Behavioral)	VHDL Entity
DSCP_mapping	DSCP(Behavioral)	VHDL Entity
ECN_mapping	ECN(Behavioral)	VHDL Entity
total_length_mapping	total_length(Behavioral)	VHDL Entity
identification_mapping	identification(Behavioral)	VHDL Entity
flags_mapping	flags(Behavioral)	VHDL Entity
fragment_offset_mapping	fragment_offset(Behavioral)	VHDL Entity
TTL_mapping	TTL(Behavioral)	VHDL Entity
protocol_num_mapping	protocol_num(Behavioral)	VHDL Entity
head_checksum_mapping	head_checksum(Behavioral)	VHDL Entity
src_ip_mapping	src_ip(Behavioral)	VHDL Entity
dst_ip_mapping	dst_ip(Behavioral)	VHDL Entity

Obr. 23: Entity testbench-u

Pre ukážku a dobrú názornosť si bližšie ukážeme napríklad testbench entity IPv4 paketu a všetkého čo obsahuje na Obr. 24.



Name	Value	Data Type
clk	0	Logic
set_seed	0	Logic
seed8[7:0]	1	Array
IPv4_out[159:0]	c0a900UUc0...	Array
version_out[3:0]	4	Array
IHL_out[3:0]	5	Array
DSCP_out[5:0]	0	Array
ECN_out[1:0]	0	Array
total_length_out[15:0]	160	Array
head_checksum_out[15:0]	0	Array
identification_out[15:0]	0	Array
flags_out[2:0]	2	Array
fragment_offset_out[12:0]	0	Array
TTL_out[7:0]	10	Array
protocol_out[7:0]	17	Array
src_out[31:0]	19216815	Array
dst_out[31:0]	1921681200	Array

Obr. 24: Ukážka testbench IPv4 entity

Môžeme vidieť jednotlivé „pod-entity“, teda jednotlivé polia reálneho IPv4 paketu. V stĺpci „value“ respektíve „hodnota“ zase môžeme vyčítať obsah poľa. Výsledné hodnoty sa zobrazujú v decimálnej podobe pre intuitívnejšiu kontrolu. Hodnoty, ako je vidieť, korelujú s hodnotami popisovanými a definovanými v kapitole 7.2 .

9. Syntéza

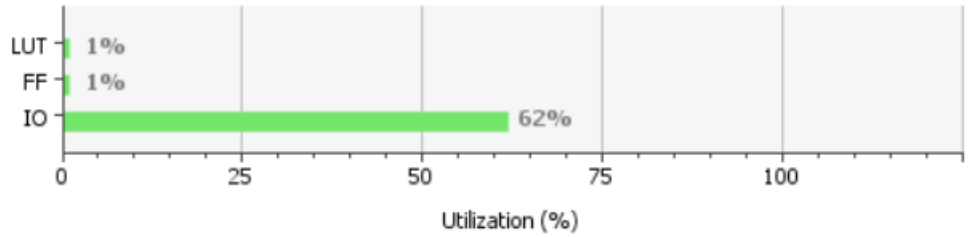
Na rozdiel od simulácie, pre syntézu aplikácie bol použitý vývojový server, s možnosťou vzdialeného prístupu, dostupný na adrese lfpga.utko.feec.vutbr.cz alebo IP adrese 192.168.200.2 . Aktuálne beží na operačnom systéme Linux, konkrétne distribúcia CentOS 6.6 x64. O výpočtovú silu sa stará CPU s konfiguráciou Intel® Xeon® E5-2643 (3.30 GHz) a RAM pamäti o veľkosti 6 GB. Informácie o použitom serveri sú zhrnuté v nasledujúcej tabuľke.

Syntéza bola vykonaná už na vývojovom servery z dôvodu validnejších výsledkov pre následnú implementáciu pri použití čipu Virtex 7. Pre porovnanie však bola spustená aj pre nižšiu radu čipov Artix 7 na rovnakom hostiteľskom počítači ako simulácia. Porovnanie utilizácie môžeme vidieť v Tab. 4. V zátvorkách je vždy uvedený maximálny počet daného prvku.

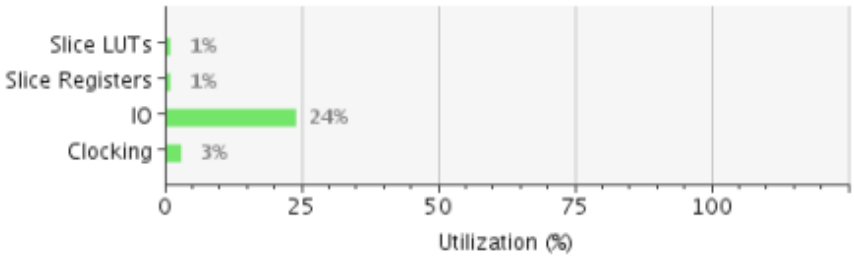
Tab. 4: Porovnanie syntézy pre Artix 7 / Virtex 7

-	**Artix 7**	**Virtex 7**
Slice LUT's	11 (134600)	11 (364200)
Slice Registers	23 (269200)	23 (728400)
Bonded IOB	248 (400)	248 (1032)
BUFGCTRL	1 (32)	1(32)

Z tabuľky je zrejmé že, viac možnosti a väčšie množstvo dostupných prostriedkov, ponúka práve Virtex 7. Zároveň jasne vidieť, že naša aplikácia využíva nepatrné percento potenciálu daných čipov u oboch variant. To môžeme vidieť aj na grafickom zobrazení (Obr. 25), ktoré ponúka Vivado pri kontrole výsledkov syntézy.



Obr. 25: Výsledky syntézy (Artix 7)



Obr. 26: Výsledky syntézy (Virtex 7)

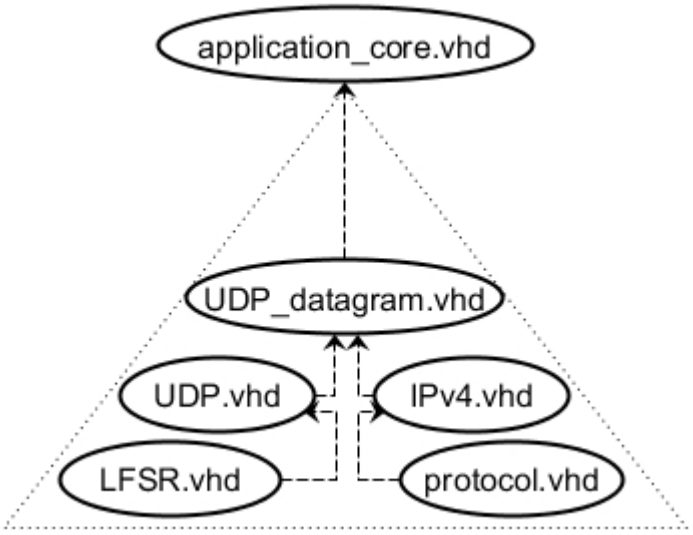
Ďalej sa budeme zaoberať, už len použitým čipom Vritex 7. Syntéza bola vykonaná pre hodinový signál s frekvenciou 80Mhz s veľkosťou výsledného vektoru 224 bitov. Ak tieto hodnoty vložíme do vzorca (2) pre výpočet priepustnosti systému, dostaneme:

$$80\times(10)\text{^}6\ Hz \times 224b = 17,920\ Gb/s \qquad (2)$$

Výsledná teoretická bitová rýchlosť je teda skoro 18 Gb/s.

9.1 Top modul

Aplikačný modul (application_core.vhd) je modul dizajnový pre integráciu užívateľovho projektu. Súbor application_core.vhd obsahuje inštanciu registrov a dekodér adries. Práve tento súbor funguje ako „prepoj" medzi našou súborovou štruktúrou a práve netcope-om s ktorým sa práve týmto procesov prepojí. Ako výsledný top modul, teda modul najvyššej vrstvy je súbor application_core.vhd. Kombinuje v sebe výstupy nižších entít, respektíve celú štruktúru generátora paketov. [Obr. 27]



Obr. 27: Architektúra súborov projektu

Naviazanie je prevedené prepojením výstupov z entity UDP_datagram na vstupy entity application_core. Viz Príklad kódu 8.

```
udp_datagram : entity work.UDP_datagram
    port map(
    clk                         => core_clk,
    set_seed                    => udp_set_seed,
    seed8                       => udp_seed8,
    seed16                      => udp_seed16,
    UDP_datagram_out    => udp_out
);
```

Príklad kódu 8.: Previazanie medzi UDP_datagram a application core

Výsledný vektor (223:0) je rozdelený na 7 častí po 32 bitov a postupne zapisovaný do 32bitových registrov „reg_experiment0-6“.

```
.....
          if (we_experiment0 = '1') then
          reg_experiment0 <= UDP_out (223 downto 192);          --zápis 32bitov z UDP_out do
                                                                reg_experiment0
          end if;
          if (we_experiment1 = '1') then
          reg_experiment1 <=  UDP_out (191 downto 160);
          end if;
          if (we_experiment2 = '1') then
          reg_experiment2 <=  UDP_out (159 downto 128);
          end if;
          if (we_experiment3 = '1') then
          reg_experiment3 <= UDP_out (127 downto 96);
          end if;
          if (we_experiment4 = '1') then
          reg_experiment4 <= UDP_out (95 downto 64);
          end if;
          if (we_experiment5 = '1') then
          reg_experiment5 <= UDP_out (63 downto 32);
          end if;
          if (we_experiment6 = '1') then
          reg_experiment6 <= UDP_out (31 downto 0);
        end if;
      end if;
    end if;
.....
  end process;
```

Príklad kódu 9.: Zápis do registrov „reg_experiment0-6“.

9.2 Výsledky

Po úspešnej syntéze aplikácie s previazaním na frameworkom NetCOPE stúplo, pomerne radikálne, percento utilizácie na hardvér. Kde sme sa dostali z približne 1% (11 z 364200) na 29% pri LUT tabuľkách. Pri registroch zase z hodnoty 1% (23 z 728400) na 13%. Tento fakt je spôsobený tým, že aj samotný framework potrebuje pre svoje fungovanie určitú pridanú dávku prostriedkov. Výsledné hodnoty po syntéze sú zhrnuté v Tab. 5, respektíve v Tab.6, ktoré boli vyčítané zo súboru *fpga_synth.util* v adresári *build,* vygenerovaného po úspešnej syntéze.

Tab. 5: Výsledky syntézy s NetCOPE frameworkom (LUT a registre)

Typ	Využité	Dostupné	Utilizácia [%]
Slice LUT's	127194	432368	29,41
LUT as Logic	104908	432368	24,26
LUT as Memory	22286	173992	12,80
LUT as Distributed RAM	3031	-	-
LUT as Shift Register	19255	-	-
Slice Registers	117309	864736	13,56
Register as Flip Flop	117309	864736	13,56
Register as Latch	0	864736	0
F7 Muxes	5451	216184	2,52
F8 Muxes	623	108092	0, 57

Tab. 6: Výsledky syntézy s NetCOPE frameworkom (RAM)

Typ	Využité	Dostupné	Utilizácia [%]
Block RAM Tile	465,5	1470	31,02
RAMB36/FIFO	444	1470	30,20
FIFO36E1	89	-	-
RAMB36E1	355	-	-
RAMB18	25	2940	0,85
RAMB18E1	25	-	-

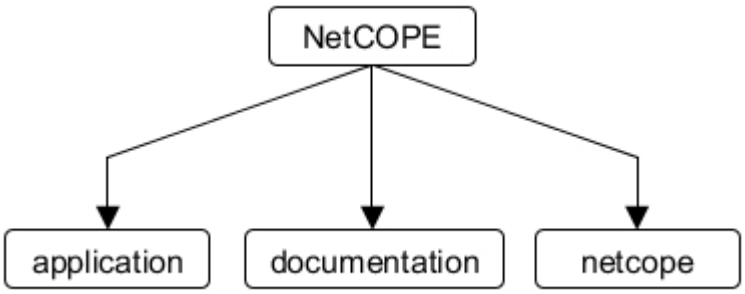
10. Implementácia

Pri implementácií na kartu COMBO bol použitý framework od spoločnosti NetCOPE. Konkrétne bola použitá bola verzia 8.02.02. Popisu prostredia NetCOPE, jeho štruktúre, vrstvám, firmvéru, softvéru a princípu komunikácie je venovaná kapitola č.4. V tejto časti sa budeme venovať postupu ako prebiehala implementácia a čo všetko bolo potrebné pre to aby prebehla úspešne.

Implementácia je proces prevodu obvodu z logických brán a klopných obvodov (vygenerovaných počas syntézy), IP jadrá a netlisty do konfiguračného súboru, ktorý môže byť nahraný do obvod FPGA. Prostredie NetCOPE využíva implementáciu prostredníctvom „Xilinx Vivado Design Suite package tools“. Implementácia závisí od cieľového okruhu a pozostáva z optimalizácie, umiestnenia (návrh miesta), fyzická optimalizácia a smerovania. [15]

10.1 Architektúra

Na úvod je dobré zoznámiť sa so súborovou architektúrou samotného frameworku poskytovaného NetCOPE-om [14]. Nebudeme sa však zaoberať úplne celou. NetCOPE balíček sa delí na prvej úrovni do troch častí [Obr. 23]. Nebudeme sa však zaoberať úplne celou a podrobne sa budeme venovať, pre implementáciu, najdôležitejším častiam.

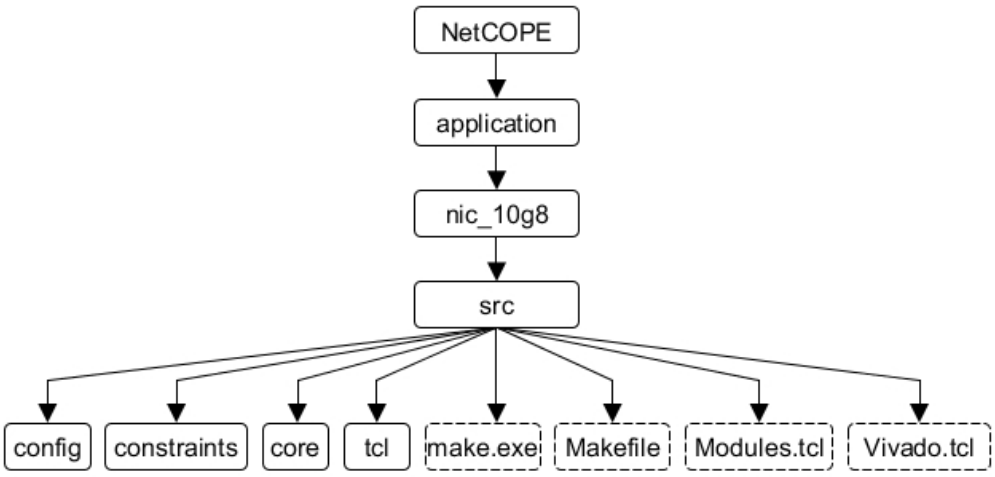


Obr. 23: Súborová architektúra NetCOPE (1.úroveň)

- **application** - Adresár obsahuje podštruktúru s konfiguračnými súbormi, modul používateľskej aplikácie a jeho profil simulácie.

- **documentation** - Adresár obsahuje celú dokumentáciu prostredia NetCOPE.

- **netcope** - Adresár obsahuje podštruktúru s firmvérom NetCOPE. Firmvér jeZložené zo súborov zdrojového kódu, súborov jadra IP a súborov netlist. Subštruktúra tiež obsahuje skripty, ktoré sa používajú počas prekladu a simulácie firmvéru NetCOPE.



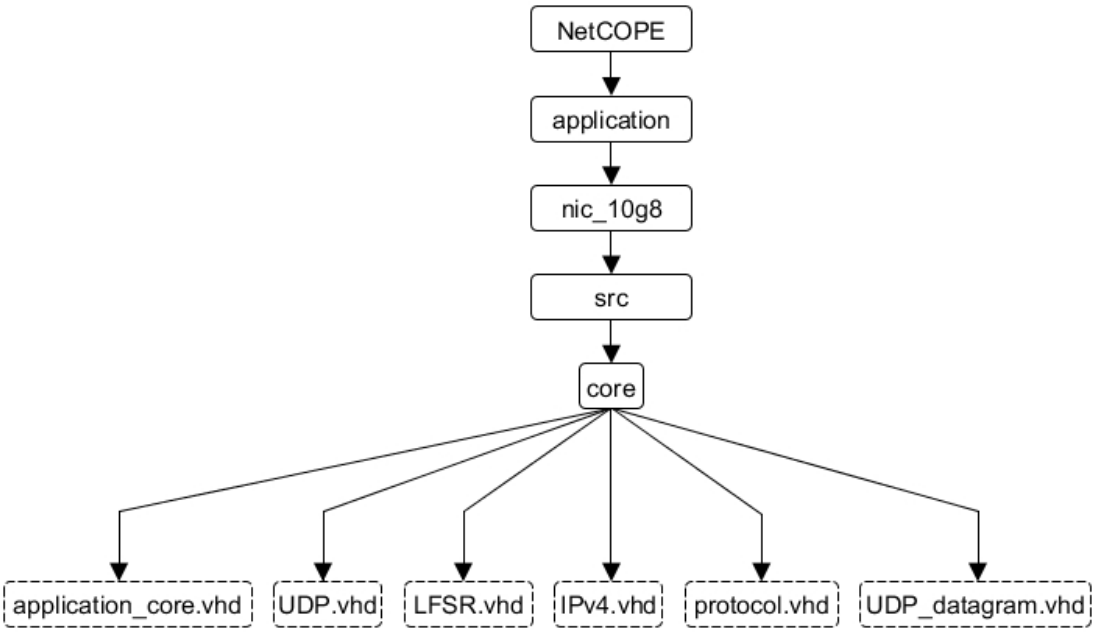
Obr. 24: Obsah adresára application

Ďalej sa budeme venovať adresáru „application“ [Obr. 24]. Aplikačný adresár obsahu je jeden alebo viac (podľa typu Combo karty) podzložky s názvom nic_X. "X" v nic_X označuje konfiguráciu sieťového rozhrania firmvéru. Napríklad nic_10g8, označuje firmvér poskytujúci osem sieťových rozhraní s prenosovou rýchlosťou 10 Gbps, nic_40g2 označuje firmvér poskytujúci dve sieťové rozhrania s prenosovou rýchlosťou 40 Gbps a nic_100g1 označuje firmvér poskytujúci jedno sieťové rozhranie s prenosovou rýchlosťou 100 Gbps.

- **config** - Adresár obsahuje súbor config.tcl s užívateľským nastavením firmware NetCOPE (napr. frekvencie hodinových signálov alebo konfigurácie kanálov DMA).
- **constraints** - Adresár obsahuje súbor constraints.xdc. Súbor je určený pre užívateľom definované obmedzenia. Súbor je prázdny vo východiskovej konfigurácii.
- **core** - Adresár obsahuje súbor application_core.vhd s modulom aplikačnej časti.
- **tcl** - Adresár obsahuje podadresáre vivado_msg_filter a vivado_user_drc s tcl skripami. Tieto skripty sa používajú na nastavenie správ počas prekladu firmvéru.
- **make.exe** - Spustiteľný súbor s nástrojom Make pre Windows OS. Tento súbor je možné použiť na automatický preklad firmvéru.
- **Makefile** - spustiteľný súbor s nástrojom Make pre Linux OS. Tento súbor je možné použiť na automatický preklad firmvéru.
- **Modules.tcl** - Konfiguračný súbor so zoznamom súborov zdrojového kódu používateľa.
- **vivado.tcl** - Konfiguračný súbor s preddefinovanými cestami a parametrami pre aplikáciu Make.

Ako najvyšší modul aplikácie generátora paketov je použitý spomínaný súbor „application_core.vhd“ umiestnený v adresáry *core* [Obr. 25]. Obsahuje mapovanie výstupov nižších entít aplikácie zobrazených na Obr. 16. Je v ňom

obsiahnutá hlavná kostra celej nadväznosti aplikácie na hardvér karty. Do totožného adresára boli umiestnené aj všetky nami definované podmoduly aplikácie generátora, pre ktoré je však potrebné definovať cestu v súbore *Modules.tcl* .



Obr. 25: Obsah adresára „core“

Po úspešnej syntéze a implementácií, NetCOPE vygeneruje súbor, takzvaný bitstream s predvoleným názvom fpga.bit. Ten sa následne prevádza pomocou skriptu obsahujúceho kód viz príklad kódu 10., do súboru formátu .mcs a nahráva do karty COMBO.

```
#!/bin/sh
promgen -p mcs -data_width 16 -w -u 0 fpga.bit -o fpga.mcs
```

Príklad kódu 10.: Vytvorenie .mcs súboru

11. ZÁVER

Cieľom diplomovej práce bolo navrhnúť nástroj na generovanie sieťovej prevádzky na platforme FPGA. Jedná sa o realizáciu generátora sieťovej prevádzky na rozhraní 10 Gb/s. Zoznámiť sa s programovacím jazykom VHDL, FPGA kartami COMBO, vývojovým frameworkom NetCOPE a platformou Xilinx Vivado. Navrhnúť architektúru modulu v jazyku VHDL a overiť jeho funkčnosť simuláciou, syntézou a vo fyzickej implementáciu na karte COMBO.

Návrh štruktúry modulu prebiehal vo vývojovom prostredí Xilinx Vivado vo verzii 2016.3 s použitím hardvérového popisného jazyka VHDL. Bola navrhnutá funkcionalita a prepojenie jednotlivých entít ako aj proces generovania a vkladania informácie do polí paketu. Pre generovanie náhodného obsahu bol využitý nami definovaný lineárny posuvný register LFSR pri nastavení vstupných hodnôt taps s maximálnym možným množstvom možností na výstupe akú daná veľkosť registra umožňuje. Pre porovnanie výsledkov prebehla simulácia na dvoch výkonnostne rôznych čipoch a to menej výkonnom Artix 7 a dokonalejšom Virtex 7.

Pre praktickú realizáciu bolo využité vývojové prostredie NetCOPE s verziou 8.02.02 a karta Combo-80g. Aplikačný modul zabezpečuje generovanie PDU 4. vrstvy OSI/ISO modelu, konkrétne UDP, zabalených v hlavičke IP protokolu 3.vrstvy OSI/ISO. Modul definuje príslušnú veľkosť jednotlivých polí PDU, naplní ich informáciou, odpovedajúcou teórií daných protokolov, a následne spája do výsledného celku - paketu. Ten je následne rozdelený na časti a uložený v registroch karty Combo80g. Celkové prostriedky karty boli využité na približne na 30% v prípade LUT tabuliek, 14% z dostupných registrov a 30% pamäti RAM. Z môžeme vyvodiť dobrú utilizáciu aplikačného modulu s veľkou rezervou, napriek niekoľko násobnému nárastu po implementácií s NetCOPE-om. Frekvencia hodinového cyklu je 80Mhz, čo prináša vo výsledku teoretickú bitovú rýchlosť 17,920 Gb/s. Ciele práce boli teda splnené.

Optimalizácia by bola možná paralelizáciou dvoch aplikačných jadier a dosiahnutím vyšších bitových rýchlostí. Ďalšou možnosťou ako ešte zvýšiť množstvo generovanej informácie v danom čase je zvýšenie generovanej frekvencie, s nami použitých 80 MHz, na vyššiu úroveň. Treba však podotknúť, že horná hranica daná výrobcom je 120 MHz.

Medzi možné rozšírenia práce patrí spomenutá optimalizácia a dosiahnutie vyššej priepustnosti generátora. Ďalším možným rozšírením je zabezpečenie prenosu vygenerovaného paketu, uloženého v registroch, na ethernetové rozhranie karty.

Zoznam použitých zdrojov

[1] PINKER, Jiří, Martin POUPA. Číslicové systémy a jazyk VHDL. 1. vyd. Praha: BEN - technická literatura, 2006, 349 s. ISBN 80-7300-198-5.

[2] GROLÉAT, T., A. BOURGE, M. ARZEL, Y. LE BALCH, S. VATON a H. BOUGDAL. Flexible, extensible, open-source and affordable FPGA-based traffic generator [online]. 2012, [cit. 2016-09-03]. Dostupné z: https://portail.telecom-bretagne.eu/publi/public/fic_download.jsp?id=16866

[3] SMITH, Ginna. FPGAs 101: Everything you need to know to get started [online]. [cit.2016-11-21]. Dostupné z : http://booksite.elsevier.com/samplechapters/9781856177061/01~Front_Matter.pdf

[4] BAETONIU, Catalin. *High Speed True Random Number Generators in Xilinx FPGAs* [online]. [cit.2016-11-21].Dostupné z: http://forums.xilinx.com/xlnx/attachments/xlnx/EDK/27322/1/HighSpeedTrueRandomNumberGeneratorsinXilinxFPGAs.pdf

[5] HAWKINS, D. W. *Linear feedback shift-registers (LFSR) and Pseudo-random binary sequences (PRBS)* [online]. California Institute of Technology,: Owens Valley Radio Observatory., 2011 [cit. 2016-11-21].

[6] Xilinx. *Efficient Shift Registers, LFSR Counters, and Long PseudoRandom Sequence Generators* [online]. 1996 [cit. 2016-11-21]. Dostupné z: http://www.xilinx.com/support/documentation/application_notes/xapp052.pdf

[7] COMBO-80G FPGA Card. Netcope. netcope [online]. Česká Republika [cit. 2017-04-16]. Dostupné z: http://www.comintindia.com/products/network-visibilitytesting/combo-80g-fpga-card

[8] 7 Series FPGAs Overview: Advance Product Specification. XILINX INC. [online]. USA, 2011, 2013 [cit. 2017-04-16]. Dostupné z: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[9] PCI Express Base Specification Revision 3.0. PCI-SIG. [online]. 2002, 2010 [cit. cit. 2017-04-16]. Dostupné z: http://composter.com.ua/documents/PCI_Express_Base_Specification_Revision_3.0.pdf

[10] UDP paket. *Http://www.cs.vsb.cz/* [online]. [cit. 2016-12-04]. Dostupné z: http://www.cs.vsb.cz/grygarek/PS/lect/tcpip.html

[11] NICHOLS, K. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers: RFC 2474* [online]. , 20 [cit. 2017-04-16]. Dostupné z: https://tools.ietf.org/html/rfc2474#page-7

[12] BURDA, K. *Návrh, správa a bezpečnost počítačových sítí*. Brno: VUT v Brně, 2014. s. 1-171.

[13] RAMAKRISHNAN, K. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers: RFC 3168* [online]. , 63 [cit. 2017-04-16]. Dostupné z: https://tools.ietf.org/html/rfc3168#page-10

[14] INVEA-TECH. *NetCOPE Framework: Quick Start Guide* [online] [cit. 2016-12-04].

[14] COMBO-80G FPGA Card. Netcope. netcope [online]. Česká Republika [cit. 2017-04-16]. Dostupné z: http://www.comintindia.com/products/network-visibilitytesting/combo-80g-fpga-card

[15] INVEATECH. *FPGA Framework for Rapid Development of Network Applications: Users Manual* [online]. 2015 [cit. 2017-05-15].

Zoznam skratiek, symbolov a veličín

ASIC	Application Specific Integrated Circuit
CE	Congestion Experienced
CLB	Configurable Logic Blocks
CWDM	Coarse Wavelength Division Multiplexing
DDR	Double Data Rate
DLL	Delay-Locked Loop
DSCP	Differentiated Services Code Point
ECN	Explicit Congestion Notification
EMI	Electro-Magnetic Interference
EOL	End Of Options
FPGA	Field Programmable Gate Array
GAL	Generic Array Logic
HDL	Hardware Description Language
ICMP	Internet Control Message Protocol
IHL	Internet Header Length
LFSR	Linear-Feedback Shift Register
LUT	Look-Up Table
MIG	Memory Interface Generator
MTU	Maximum Transmission Unit
PAL	Programmable Array Logic
PDU	Protocol Data Unit
PLD	Programmable Logic Device
PLL	Phase-Locked Loop
PPS	Pulse Per Second
STA	Static Time Analysis
TTL	Time To Live
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit

Zoznam príloh

Príloha A – Obsah priloženého DVD

DVD priložené k diplomovej práci obsahuje nasledujúce súbory a adresáre.

NetCOPE\applications\nic_10g8\src\	- adresár so súborom Modules.tcl
NetCOPE \applications\nic_10g8\src\core	- adresár so zdrojovými kódmi projektu pre NetCOPE
NetCOPE \applications\nic_10g8\build\	- adresár so súbormi fpga.bit a fpga.mcs
Vivado\src\	- adresár so zdrojovými súbormi, testbenchami pre Xilinx Vivado
Vivado\Constraints\	- adresár so súborom Constrain (top_clock)