VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV MIKROELEKTRONIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF MICROELECTRONICS

IMPLEMENTACE VÝPOČTU FFT V OBVODECH FPGA A ASIC

FFT IMPLEMENTATION IN FPGA AND ASIC

DIPLOMOVÁ PRÁCE
MASTER´S THESIS

AUTOR PRÁCE — Bc. Vojtěch Dvořák
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. Lukáš Fujcik Ph.D.
SUPERVISOR

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav mikroelektroniky

Diplomová práce

magisterský navazující studijní obor
Mikroelektronika

Student: Bc. Vojtěch Dvořák ***ID:*** 118345
Ročník: 2 ***Akademický rok:*** 2012/2013

NÁZEV TÉMATU:

Implementace výpočtu FFT v obvodech FPGA a ASIC

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s použitím rychlé Fourierovy transformace (FFT) pro spektrální analýzu signálů v digitálních obvodech. Navrhněte vhodný způsob implementace výpočtu FFT v obvodech FPGA a ASIC s ohledem na zvolenou strategii. Návrh popište v jazyce VHDL a ověřte jeho správnou funkci pomocí simulace. Porovnejte výsledky syntézy pro vybrané typy obvodů z hlediska dosažitelné frekvence, plochy a spotřeby. V rámci diplomové práce vytvořte program pro automatické generování kódu podle zadaných parametrů FFT.

DOPORUČENÁ LITERATURA:

[1] LYONS, Richard G. *Understanding Digital Signal Processing*. 2nd ed. New Jersey : Bernard Goodwin, 2004. 665 s. ISBN 0-13-108989-7.

Termín zadání: 11.2.2013 ***Termín odevzdání:*** 30.5.2013

Vedoucí práce: doc. Ing. Lukáš Fujcik, Ph.D.

prof. Ing. Vladislav Musil, CSc.

předseda oborové rady

UPOZORNĚNÍ:

ABSTRAKT

Cílem diplomové práce je navrhnout implementaci algoritmu rychlé Fourierovi transformace, kterou lze použít v obvodech FPGA nebo ASIC. Implementace bude modelována v prostředí Matlab a následně bude použit tento návrh jako referenční model pro popis implementace algoritmu rychlé Fourierovy transformace v jazyce VHDL. Pro ověření správnosti návrhu bude vytvořeno verifikační prostředí a provedena verifikace. V poslední části práce bude navržen program, který bude generovat zdrojové kódy pro různé parametry modulu provádějícího rychlou Fourierovu transformaci.

KLÍČOVÁ SLOVA

Číslicové zpracování signálu, diskrétní Fourierova transformace, rychlá Fourierova transformace, DFT, FFT, VHDL, programovatelné logické obvody, FPGA, zákaznické obvody, ASIC, verifikace, syntéza digitálních obvodů

ABSTRACT

The aim of this thesis is to design the implementation of fast Fourier transform algorithm, which can be used in FPGA or ASIC circuits. Implementation will be done in Matlab and then this form of implementation will be used as a reference model for implementation of fast Fourier transform algorithm in VHDL. To verify the correctness ofdesign verification enviroment will be created and verification process wil be done. Program that will generate source code for various parameters of the module performing a fast Fourier transform will be created in the last part of this thesis.

KEYWORDS

Digital signal processing, discrete Fourier transform, fast Fourier transform, DFT, FFT, VHDL, programable logic circuits, FPGA, application-specified integrated circuit, ASIC, verification, synthesis of digital circuits

DVOŘÁK, V. *Implementace výpočtu FFT v obvodech FPGA a ASIC*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2013. 71 s. Vedoucí diplomové práce doc. Ing. Lukáš Fujcik, Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma Implementace výpočtu FFT v obvodech FPGA a ASIC 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 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.

V Brně dne ....

(podpis autora)

PODĚKOVÁNÍ

Děkuji vedoucímu diplomové práce doc. Ing. Lukáši Fujcikovi Ph.D. za účinnou metodickou, pedagogickou a odbornou pomoc a další cenné rady při zpracování projektu.

Předkládaná diplomová práce byla realizována v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

V Brně dne ....

(podpis autora)

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

ÚVOD

Fourierova transformace je matematický nástroj pro převod posloupnosti vzorků signálu v časové oblasti na posloupnost hodnot ve frekvenční oblasti, obecně známé jako spektrum signálu. Ve frekvenční oblasti se nabízí více možností na úpravu signálu, a proto se Fourierova transformace stala velmi důležitou součástí při zpracování signálu.

Diskrétní Fourierova transformace (v následujícím textu bude používána zkratka DFT) pracuje s digitálním signálem, vstupní hodnoty jsou vzorkované hodnoty spojitého signálu a výstupní hodnotou jsou frekvence obsažené ve spektru analyzovaného signálu. Rovnice (1) obsahuje předpis pro DFT [1].

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-j*2\pi*\frac{k}{N}n} \tag{1}$$

kde: X_k je k-tá harmonická frekvence ve frekvenční oblasti

x_n je n-tý vzorek signálu v časové oblasti

k je pořadí výstupního vzorku

n je pořadí vstupního vzorku

N je velikost vektoru vstupních hodnot

Diskrétní Fourierova transformace je bohužel výpočetně velice náročná kvůli vysokému počtu aritmetických operací, počet operací násobení je roven N^2 a počet operací sčítání je N*(N-1) [1]. S rostoucím počtem vzorků tedy prudce narůstá i počet operací a pro výpočty v reálném čase je tento postup téměř nepoužitelný. Změna přišla až v roce 1965, kdy James Cooley a John Tukey objevili nový algoritmus výpočtu DFT, který výrazně snížil počet matematických operací. Tento algoritmus byl později nazván Rychlá Fourierova transformace (z anglického názvu Fast Fourier Transform vnikla často používaná zkratka FFT, která bude používána i v následujícím textu).

Rychlá Fourierova transformace se brzy uplatnila v mnoha odvětvích lidské činnosti, kde je třeba pracovat se signálem. Je vhodná pro implementaci do signálových procesorů díky velké úspoře výpočetních kroků a tedy i času, stejně tak je vhodná pro přímou implementaci na čip, kdy díky menšímu množství operací snižuje plochu čipu a tedy i cenu čipu při zachování výpočetní rychlosti.

Úkolem v tomto projektu je navrhnout algoritmus rychlé Fourierovy transformace v jazyce VHDL pro implementaci do obvodů ASIC a FPGA. U těchto návrhů je požadována především úspora místa na čipu (především u obvodů ASIC patří tento požadavek kvůli ceně výroby takového čipu právě mezi nejdůležitější požadavky) a zároveň je požadována pokud možno co nejvyšší rychlost, v ideálním případě by mělo zpoždění odpovídat právě počet vstupních vzorků. Těmto požadavkům je návrh také přizpůsoben. V první části práce bude nastíněno odvození algoritmu rychlé Fourierovy transformace z diskrétní Fourierovi transformace, bude vypočtena očekávána plocha na čipu pro různé velikosti FFT a poměr frekvence hodinového signálu a frekvence, s kterou přicházejí vstupní data. Bude také popsán model v prostředí Matlab, který

později poslouží jako vzor pro tvorbu referenčního modelu při verifikaci. Dále bude popsán postup návrhu a architektura výsledného obvodu. Správnost návrhu bude ověřena verifikací. Následně bude provedena syntéza do vybraných obvodů FPGA a obvodů ASIC. V poslední části práce bude vytvořen generátor zdrojových kódů FFT modulu pro různé velikosti FFT, různou šířku dat a další volitelné parametry obvodu.

1 RYCHLÁ FOURIEROVA TRANSFORMACE

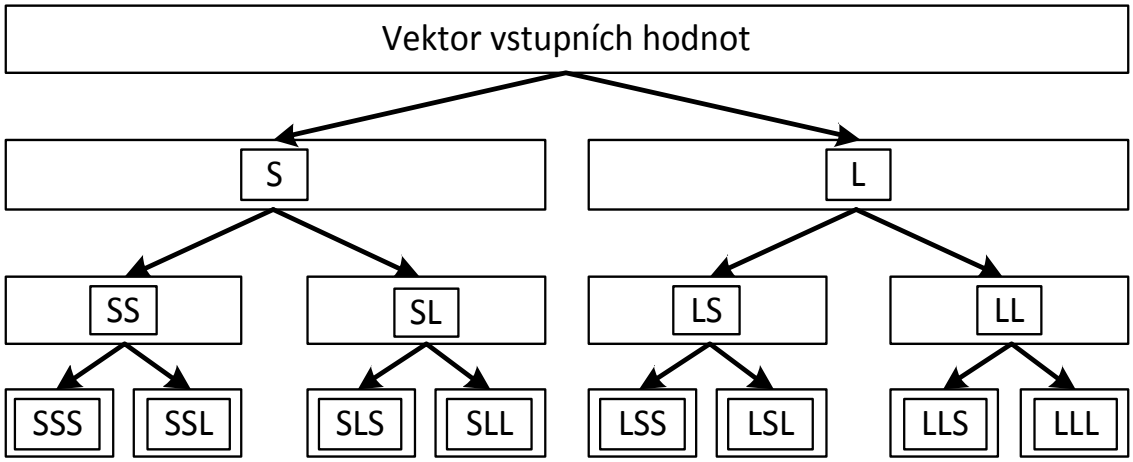
Rychlá Fourierova transformace je název pro skupinu algoritmů umožňujících výpočet diskrétní Fourierovi transformace s podstatně menším počtem matematických operací v porovnání s přímým výpočtem DFT. Jak už bylo zmíněno v úvodu, výraz rychlá Fourierova transformace se používá od roku 1965, kdy byl první takový algoritmus popsán Jamesem Cooley a Johnem Tukey. Později byly popsány i další algoritmy, kterými se však tato práce nebude zabývat. Jedná se totiž o algoritmy vhodné pro jiný počet vstupních vzorků než je mocnina dvou, o algoritmy, ve kterých je minimalizován počet operací násobení, ale významně zvýšen počet operací sčítání, případně jsou to algoritmy optimalizovány pouze pro reálná čísla [3]. Jejich význam je především při implementaci výpočtu do procesoru a to pouze ve speciálních případech a nejsou zcela vhodné pro implementaci do obvodů ASIC ani obvodů FPGA. Tato práce se tedy bude zabývat implementací původního Cooley – Tukey FFT algoritmu.

Jak už bylo nastíněno v úvodu, počet matematických operací u DFT prudce narůstá s velikostí DFT, tedy s počtem vzorků signálu. To je dáno faktem, že každá výstupní hodnota se počítá ze všech vstupních hodnot. Jak v roce 1965 dokázali autoři původního algoritmu James Cooley a John Tukey, právě v tomhle postupu se mnohokrát opakuje stejný výpočet, který lze po jistých úpravách v postupu výpočtu DFT provést jen jednou a použít pro všechny výstupní vzorky. Tyto úpravy zahrnující změnu pořadí vstupních vzorků, výpočtu několika málo koeficientů v porovnání s původní DFT a následným postupným sčítáním jednotlivých vzorků a násobení těmito koeficienty.

Přestože od objevu Cooley – Tukey FFT algoritmu uplynulo už více než půl století a mezitím bylo popsáno několik dalších algoritmů FFT, je tento první algoritmus dodnes nejčastěji používaný. Jedná se o příklad algoritmu typu „rozděl a panuj“. Takový algoritmus pracuje následujícím způsobem: nejdříve dojde k rozdělení vstupního vektoru na menší části, na těchto menších částech provede diskrétní Fourierovu transformaci a výsledné obrazy jednotlivých částí spojí do spektra celého signálu [1].

V obecném případě může popsaný algoritmus rozdělit vektor vstupních hodnot na menší vektory o velikosti mocniny čísla dvě, na kterých je provedena DFT. Avšak nejčastější formou FFT je tzv. Radix-2, kdy dochází k rekurzivnímu dělení vektorů na polovinu jejich původní velikosti, až je výsledkem vektor vstupních vzorků o velikosti 2. Existují také další varianty tohoto algoritmu, kdy například vektor vstupních dat není dělen na poloviny, ale na čtvrtiny (varianta Radix-4), výsledkem pak není dvojice hodnot ale čtveřice, na které je prováděna DFT. V této práci se však budeme zabývat převážně nejjednodušší variantou Radix-2, která je, co se týče počtu matematických operací, výhodnější. Postup FFT algoritmu Radix-2 je zakreslen na Obrázku 1. V prvním kroku rozděluje vstupní vzorky na polovinu – na sudé (S) a liché (L) vzorky. Tímto vzniknou dva vektory vstupních hodnot (v obrázku označeny jako vektor S a vektor L), ve kterých lze opět všem vzorkům přiřadit sudý a lichý index a tyto vektory znovu rozdělit na poloviční vektory a výstupem jsou čtyři vektory (SS, SL, LS, LL) o

čtvrtinové délce oproti původnímu vstupnímu vektoru. Tímto způsobem by bylo možné pokračovat tak dlouho, až místo vstupního vektoru zůstanou vždy dvojice vzorků. Z dělení vektoru na poloviny plyne i požadavek na počet vstupních hodnot, který je roven mocnině čísla dvě. [1]



Obrázek 1: Rekurzivní dělení vektoru vstupních dat pro DFT na poloviční vektory

1.1 Odvození Cooley – Tukey algoritmu FFT z DFT

Jak bylo popsáno v minulé kapitole, algoritmus rychlé Fourierovy transformace byl odvozen z diskrétní Fourierovy transformace rekurzivním rozdělováním vstupních vzorků na poloviny. Vzhledem k tomu, že se pracuje se vstupními daty, které reprezentují časovou oblast signálu, bývá tento algoritmus také označován jako decimace v časové oblasti (Decimation in time – DIT) [1]. Podívejme se na toto odvození nyní trochu podrobněji s pomocí rovnice (1).

V prvním kroku je podle Obrázku 1 potřeba rozdělit vstupní vektor x_n na vektory poloviční délky. Tyto vektory označíme x_{2n} a x_{2n+1}, přičemž v prvním jmenovaném vektoru budou prvky se sudým indexem (x_0, x_2, x_4, …) a ve druhém vektoru budou prvky s lichým indexem (x_1, x_3, x_5, …). Potom můžeme rovnici diskrétní Fourierovy transformace rozepsat do rovnice (2).

$$X_k = \sum_{n=0}^{N-1} x_n * e^{-j*2\pi*\frac{k}{N}n} = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} * e^{-j*2\pi*\frac{k}{N}*2n} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} * e^{-j*2\pi*\frac{k}{N}(2n+1)} \qquad (2)$$

Na první pohled jsme touto úpravou nic nezískali, rozdělili jsme sice výpočet na poloviny, ale počet operací se nezmenšil. Pokud se podíváme na druhý součet obsahující liché prvky, v exponenciále je člen $2n+1$. Z tohoto členu můžeme vytknout jedničku a následně z celého součtu vytknout exponenciálu bez proměnné n.

$$X_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} * e^{-j*2\pi*\frac{k}{N}*2n} + e^{-j*2\pi*\frac{k}{N}} * \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} * e^{-j*2\pi*\frac{k}{N}(2n)} \qquad (3)$$

Exponenciální člen před součtem lichých prvků bývá označován jako otáčecí činitel (častěji se používá anglický výraz twiddle factor). Můžeme si všimnout, že je nezávislí na pozici vstupního vzorku, ale závisí jen na pozici výstupního vzorku. To znamená, že pro výpočet jednoho konkrétního výstupního vzorku se jedná o konstantu nezávislou na vstupním vektoru a její hodnota bude stejná pro všechny výpočty. Vlastnosti a důvod použití otáčecího činitele budou popsány v kapitole 1.2.

Pokud bychom tuto transformaci použili na vstupní vektor o velikosti $N = 2$ a tuto hodnotu do rovnice (3) dosadili, oba součty budou probíhat jen pro $n = 0$. Když tuto hodnotu dosadíme do exponenciálního členu, jeho hodnota bude rovna 1. Rovnice (3) pak se změní na jednoduchý součet

$$X_k = x_0 + e^{-j*2\pi*\frac{k}{N}} * x_1 . \tag{4}$$

Jak vidíme, celou transformaci jsme převedli na jedno násobení a jedno sčítání. U vstupního vektoru o velikosti $N = 2$ jsme žádné úspory sice nedosáhli, ale pro větší vstupní vektory se ona požadovaná úspora matematických operací projeví. Pokud budeme uvažovat vektor vstupních hodnot o velikosti $N = 4$, můžeme rovnici (3) ještě dále upravit rozdělením obou součtů na další poloviny. Označme si vektor sudých prvků jako V_S a vektor lichých prvků jako V_L, pak lze tyto vektory rozdělit stejným způsobem jako v rovnici (3).

$$\begin{aligned} V_S &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} * e^{-j*2\pi*\frac{k}{N}*2n} = \sum_{n=0}^{\frac{N}{4}-1} x_{4n} * e^{-j*2\pi*\frac{k}{N}*4n} + \sum_{n=0}^{\frac{N}{4}-1} x_{4n+2} * e^{-j*2\pi*\frac{k}{N}*(4n+2)} = \\ &= \sum_{n=0}^{\frac{N}{4}-1} x_{4n} * e^{-j*2\pi*\frac{k}{N}*4n} + e^{-j*2\pi*\frac{k}{N}*2} * \sum_{n=0}^{\frac{N}{4}-1} x_{4n+2} * e^{-j*2\pi*\frac{k}{N}*(4n)} \end{aligned} \tag{5}$$

$$\begin{aligned} V_L &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} * e^{-j*2\pi*\frac{k}{N}*(2n+1)} = \sum_{n=0}^{\frac{N}{4}-1} x_{4n+1} * e^{-j*2\pi*\frac{k}{N}*(4n+1)} + \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} * e^{-j*2\pi*\frac{k}{N}*(4n+3)} = \\ &= e^{-j*2\pi*\frac{k}{N}} * \sum_{n=0}^{\frac{N}{4}-1} x_{4n+1} * e^{-j*2\pi*\frac{k}{N}*(4n)} + e^{-j*2\pi*\frac{k}{N}*3} * \sum_{n=0}^{\frac{N}{4}-1} x_{4n+3} * e^{-j*2\pi*\frac{k}{N}*(4n+3)} \end{aligned} \tag{6}$$

Pro velikost vstupního vektoru $N = 4$ jsou pak jednotlivé součty opět rovny jednotlivým vzorkům. U vektoru V_L lze před druhým součtem upravit otáčecí činitel jako součin dvou otáčecích činitelů a výslednou rovnici lze zapsat jako

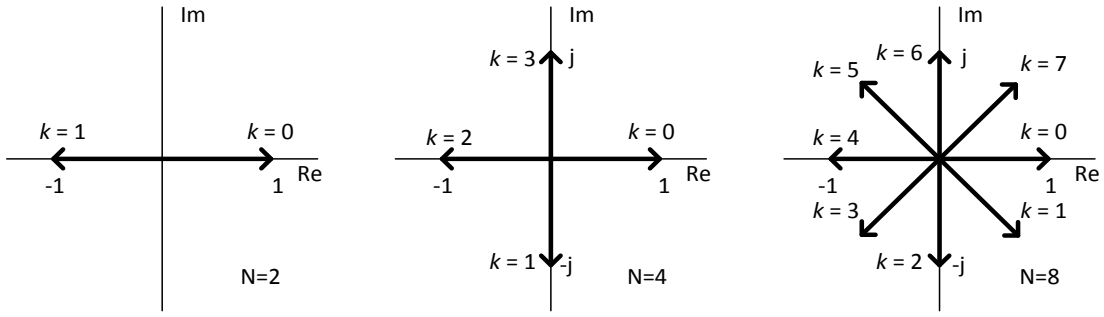
$$X_k = x_0 + e^{-j*2\pi*\frac{k}{N}*2} * x_2 + e^{-j*2\pi*\frac{k}{N}} * x_1 + e^{-j*2\pi*\frac{k}{N}*2} * e^{-j*2\pi*\frac{k}{N}} x_3 . \tag{7}$$

1.2 Otáčecí činitel

V předešlé kapitole jsme ukázali rozdělení kroků na jednotlivé prvky násobené otáčecím činitelem. Otáčecí činitel, respktive jeho anglický název „Twiddle factor“ je jedním ze základních pojmů u rychlé Fourierovy transformace [1]. Poprvé byl tento pojem použit v roce 1966 v článku o FFT od autorů Gentleman a Sande a obvykle se otáčecí činitel značí

$$W_N^k = e^{-j*2\pi*\frac{k}{N}}. \tag{8}$$

Podle rovnice (8) je otáčecí činitel komplexní exponenciála, která je v Gaussově rovině reprezentována jednotkovou kružnicí. Hodnota proměnné *N*, tedy velikost transformace, určuje počet bodů na kružnici, které pak odpovídají jednotlivým otáčecím činitelům. Na Obrázku 2 jsou vykresleny příklady otáčecího činitele v Gaussově rovině pro $N = 2$, $N = 4$ a $N = 8$.



Obrázek 2: Vykreslení otáčecího činitele v Gaussově rovině pro počet vzorků N=2, N=4 a N=8

Otáčecí činitel má několik důležitých vlastností, které jsou při výpočtu Cooley – Tukey FFT využity. Jde především o symetrii hodnot podle reálné osy, která umožňuje redukci počtu konstant, kterými jsou vzorky násobeny. Protože se jedná o periodickou funkci s periodou *N*, jsou všechny konstanty s vyšším řádem, než je hodnota *N*, shodné s konstantami řádu nižšího než *N* podle rovnice

$$W_N^{k+N} = W_N^k. \tag{9}$$

Podívejme se na příklad výpočtu otáčecího činitele pro $N = 4$ v rovnici (10). I zde je vidět, že otáčecí činitel W_4^3 je možné vypočítat jako součin $W_4^1 * W_4^2$. Tato vlastnost bude velmi užitečná při sestavení grafu signálových toků, známého jako motýlkový diagram.

$$\begin{aligned} W_4^0 &= e^{-j*2\pi*\frac{0}{4}} = 1 \\ W_4^1 &= e^{-j*2\pi*\frac{1}{4}} = e^{-j*\frac{\pi}{2}} = -j \\ W_4^2 &= e^{-j*2\pi*\frac{2}{4}} = e^{-j*\pi} = -1 \\ W_4^3 &= e^{-j*2\pi*\frac{3}{4}} = e^{-j*\frac{3\pi}{2}} = j = W_4^1 * W_4^2 \end{aligned} \tag{10}$$

1.3 Motýlkový diagram

Pokud nyní využijeme rovnice (7), (8), (9) a (10), můžeme vypočítat všechny výstupní hodnoty transformace pro šířku vstupního vektoru $N = 4$.

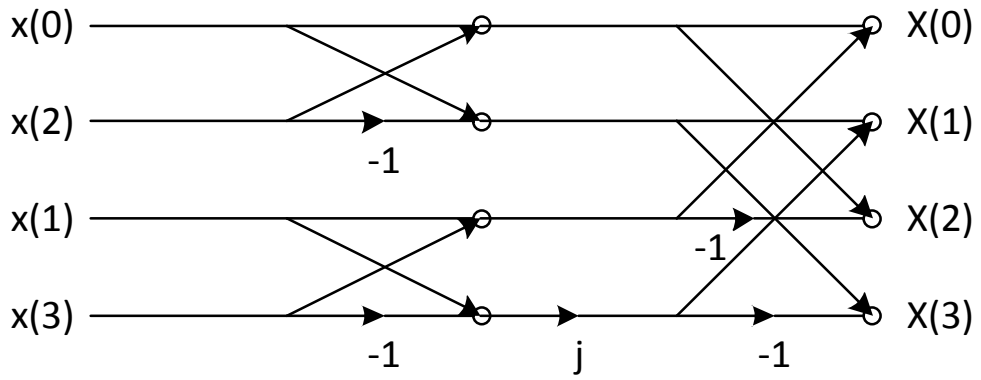
$$\begin{aligned} X_k &= x_0 + W_4^{2k} * x_2 + W_4^k * \left(x_1 + W_4^{2k} * x_3\right) \\ X_0 &= x_0 + x_2 + x_1 + x_3 \\ X_1 &= x_0 - x_2 + j * \left(x_1 - x_3\right) \\ X_2 &= x_0 + x_2 - \left(x_1 + x_3\right) \\ X_3 &= x_0 - x_2 - j * \left(x_1 - x_3\right) \end{aligned} \tag{11}$$

Prakticky se však tento způsob zápisu Cooley – Tukey FFT algoritmu nepoužívá, protože při větším vektoru vstupních dat by byl nepřehledný. Nejčastěji se takto FFT algoritmus zakresluje pomocí motýlkového diagramu (v angličtině Butterfly diagram [1]). Příklad motýlkového diagramu pro vstupní vektor velikosti $N = 4$ je nakreslen na Obrázku 3. Vstupní hodnoty byly podle zvyku zapsány v přeházeném, tzv. bitově – reverzovaném pořadí (toto bývá označováno jako DIT – Decimation In Time, decimace v čase) [1]. Bitově – reverzované pořadí vstupních hodnot se používá pro zvýšení přehlednosti motýlkového diagramu a především zajišťuje přirozené pořadí indexů výstupních hodnot, tedy 0, 1, 2, … N. Bitově - reverzované pořadí lze získat přepsáním indexu vzorku do dvojkové soustavy a následným „přečtením“ tohoto čísla odzadu. Takové pořadí zároveň odpovídá rozdělení vstupního vektoru na sudé a liché složky, jak bylo popsáno v kapitole 1.1. Příklad pro vstupní vektory o velikosti $N = 4$ a $N = 8$ je zapsán v rovnici (12).

$N = 4$	$N = 8$
$0 \Rightarrow 00 \rightarrow 00 \Rightarrow 0$	$0 \Rightarrow 000 \rightarrow 000 \Rightarrow 0$
$1 \Rightarrow 01 \rightarrow 10 \Rightarrow 2$	$1 \Rightarrow 001 \rightarrow 100 \Rightarrow 4$
$2 \Rightarrow 10 \rightarrow 01 \Rightarrow 1$	$2 \Rightarrow 010 \rightarrow 010 \Rightarrow 2$
$3 \Rightarrow 11 \rightarrow 11 \Rightarrow 3$	$3 \Rightarrow 011 \rightarrow 110 \Rightarrow 6$
	$4 \Rightarrow 100 \rightarrow 001 \Rightarrow 1$
	$5 \Rightarrow 101 \rightarrow 101 \Rightarrow 5$
	$6 \Rightarrow 110 \rightarrow ,011 \Rightarrow 3$
	$7 \Rightarrow 111 \rightarrow 111 \Rightarrow 7$

(12)

Motýlový diagram je příkladem grafu signálových toků. Šipka s číslem značí násobení signálu touto hodnotou, prázdné kolečko značí sčítání signálů. Na Obrázku 3 je vidět, že výpočet je rozdělen do dvou stupňů, v prvním stupni probíhají dvě operace DFT se dvojicemi vstupních dat (x(0) a x(2), x(1) a x(3)). Výsledek prvního stupně je přiveden do druhého stupně, kde opět probíhají dvě operace DFT se dvojicemi dat z prvního stupně, které byly vynásobeny odpovídajícím otáčecím činitelem.



Obrázek 3: Motýlový diagram algoritmu FFT pro N = 4

1.4 Náročnost výpočtu Cooley – Tukey algoritmu FFT

Náročnost výpočtu algoritmu FFT lze stanovit podle počtu matematických operací potřebných k provedení výpočtu. Právě v tomto ohledu vychází varianta algoritmu FFT Radix-2 nejlépe (nepatrně lepších výsledků dosahuje varianta Split-Radix, kde se používá zároveň Radix-2 a Radix-4, nicméně tato varianta není zcela vhodná pro implementaci do obvodů ASIC a FPGA, ale hodí se především pro výpočty pomocí procesorů; v obvodech ASIC a FPGA by mohla nalézt uplatnění při požadavku maximální rychlosti výpočtu, kdy by plocha obvodu však zantelně narostla).

Počet operací sčítání závisí na počtu jednotlivých DFT na dvojicích dat ukrytých v algoritmu FFT. Dobře je to vidět právě na motýlkovém diagramu na Obrázku 3. V prvním stupni jsou (stejně jako ve druhém stupni) dvě operace DFT, v každé z nich pak dvě operace sčítání. Pokud bychom pracovali s větším množstvím vzorků a motýlkový diagram rozšířili, bude počet takovýchto operací DFT v každém stupni odpovídat polovině velikosti vektoru vstupních dat. V každém stupni je počet operací sčítání stejný. Počet operací sčítání je tedy součin počtu vstupních vzorků a počtu stupňů [1]. Tento vztah je vyjádřen v rovnici (13).

$$Op(+) = N * \log_2(N) \tag{13}$$

Matematická operace násobení probíhá vždy mezi jednotlivými stupni, kdy data z předchozího stupně (v podstatě výsledky operací DFT na dvojici dat) jsou násobena otáčecím činitelem. V rovnici (5) je možné si všimnout, že násobeny jsou vždy jen vzorky s lichým pořadím. Z toho plyne, že operací násobení je právě polovina oproti počtu operací sčítání [1]. V rovnici (14) je tento vztah vyjádřen.

$$Op(+) = \frac{N}{2}\log_2(N) \tag{14}$$

Ve výpočtu není zohledněno, že některé operace násobení znamenají násobení číslem jedna, což například při implementaci algoritmu FFT do procesoru znamená, že skutečný počet operací násobení je menší. V případě architektury, která bude popsána v této práci, k násobení jedničkou reálně dochází, vynechání tohoto kroku by zde totiž nezpůsobilo žádné zrychlení výpočtu a počet operací v rovnici (14) je tedy přesný.

2 IMPLEMENTACE ALGORITMU FFT

Pro implementaci DFT byl zvolen Cooley – Tukey algoritmus FFT s rozdělením vektoru vstupních dat na dvojice vzorků (varianta Radix-2). Pokud bude v dalších částech práce použita zkratka FFT, bude se vždy jednat právě o Cooley – Tukey algoritmus FFT ve variantě Radix-2. Důvodem jeho použití je poměrně snadná implementace do hardwaru a potřeba použít jen jednu násobičku komplexních čísel a dvě sčítačky komplexních čísel, což v důsledku znamená úsporu kombinační logiky. Oproti dalším algoritmům FFT je pak jeho další výhodou větší variabilita velikostí vektoru vstupních dat. Je samozřejmě zavádějící porovnávat rychlost zpracování dvou algoritmů na stejné architektuře, bude tedy vhodné se nejprve podívat na porovnání požadavaných kombinačních prvků pro architektury určené pro variantu Radix-2 a Radix-4. Porovnání těchto variant nalezneme v Tabulce 1.

Tabulka 1: Porovnání počtu matematických operací pro Radix-2 a Radix-4

	Radix -2		Radix-4	
	Počet operací v jednom DFT	Obvodové prvky	Počet operací v jednom DFT	Obvodové prvky
násobení	1	3	3	9
sčítání	2	9	12	39

Jak je vidět v Tabulce 1, pokud by byla architektura uzpůsobena pro variantu algoritmu FFT Radix-4, počet obvodových prvků prodce stoupá a kombinační logika by zabrala velkou část čipu. Je dobré si také uvědomit, že rychlost výpočtu je pro Radix-4 v porovnání s variantou Radix-2 jen asi čtyřikrát lepší (rychlost odpovídá počtu operací DFT potřebných pro provedení celé transformace, s podmínkou, že každá tato operace DFT je provedena v jednom taktu hodinového signálu).

Tabulka 2: Porovnání parametrů výpočtu Radix-2 a Radix-4 pro stejnou architekturu

	Radix-2			Radix-4		
Velikost	Počet DFT (2)	clk_per / DFT (2)	Doba zpracování	Počet DFT (4)	clk_per / DFT (4)	Doba zpracování
4	4	1	4	1	6	6
16	32	1	32	8	6	48
64	192	1	192	48	6	288
256	1024	1	1024	256	6	1536
1024	5120	1	5120	1280	6	7680
4096	24576	1	24576	6144	6	36864

Pro porovnání varianty Radix-2 a Radix-4 při použití stejné architektury (architektura vhodná pro především pro variantu Radix-2) je třeba se podívat na poslední sloupce v Tabulce 2, které popisují dobu zpracování. Dobou zpracování se rozumí počet taktů hodinového signálu potřebných pro provedení FFT o dané velikosti. Jak je z Tabulky 2 patrné, oproti variantě Radix-4 je při použití stejné architektury potřeba méně výpočetních kroků a výpočet je tedy rychlejší. V dalších sloupcích v Tabulce 2 jsou parametry počet DFT a počet taktů hodinového signálu potřebných pro výpočet jednoho bloku DFT.

2.1 Koncept architektury FFT modulu

Návrh obvodu implementujícího algoritmus FFT do hardwarové podoby vychází z motýlkového diagram algoritmu FFT Radix-2. V motýlkovém diagramu je celý výpočet rozdělen na jednotlivé stupně, přičemž výpočty v navazujícím stupni využívají data z předchozího stupně. Data z předchozího stupně, s kterými už bylo počítáno v navazujícím stupni, lze okamžitě zahodit, protože už ve výpočtu nejsou dále použity. V takovém případě je výhodné použít takto uvolněné místo v paměti a uložit do něj data z aktuálního stupně. Stejně tak si můžeme povšimnout, že matematické operace jsou v každém kroku prováděny podle stejného vzoru. Pro výpočet se bere vždy dvojice čísel, jedno toto číslo se vynásobí otáčecím činitelem a následně se sečte (resp. odečte) s druhým číslem. Získáme tak dvojici výsledků, součet a rozdíl obou čísel. Přestože v motýlkovém diagramu nebývá zobrazeno násobení v prvním stupni výpočtu, ve skutečnosti lze toto chápat jako násobení jedničkou a proto i pro první stupeň je postup výpočtu stejný.

Z předchozí úvahy plyne, že pro implementaci algoritmu FFT je třeba paměťová část uchovávající data, aritmetická část provádějící matematické výpočty a řídící logika určující data, která jsou určena pro aktuální krok výpočtu. Právě řídící logika bude jakýmsi mozkem celého modulu, bude mít na starost adresaci data podle aktuálního kroku výpočtu a její návrh také bude nejobtížnější. Bude vhodné si takto navrženou architekturu a především postup řídící logiky při generování adres ověřit modelováním v Matlabu.

2.2 Referenční model v Matlabu

Před začátkem samotné implementace algoritmu FFT do hardwaru je vhodné ověřit funkčnost algoritmu v zamýšlené architektuře v prostředí Matlab. Toto ověření se provádí z prostého důvodu, popsat tento algoritmus v Matlabu je podstatně jednodušší a časově méně náročné než popis v jazyce VHDL a pokud by se v plánované architektuře vyskytla nějaká zásadní chyba, byla by odhalena ještě před začátkem implementace a došlo by k výrazné úspoře času návrhu. Zároveň tento referenční model bude sloužit jako vzor pro referenční model popsaný v SystemVerilogu, který bude na konci návrhového procesu použit pro verifikaci.

Zdrojový kód pro ověření algoritmu FFT je možné nalézt v příloze A. Kód je rozdělen na šest částí. V první části kódu je nastavena velikost FFT, jsou generovány vstupní hodnoty a připraveny velikosti použitých matic (alokování místa v paměti tím,

že jsou do nich zapsány samé nuly). Vstupní hodnoty jsou generovány pomocí generátoru náhodných čísel (funkce Random). Pro generování je použito normální rozdělení pravděpodobnosti (známé také jako Gaussovo rozložení) se střední hodnotou v nule a se směrodatnou odchylkou 1. Nicméně, toto nastavení není pro testování algoritmu nijak důležité.

V další části zdrojového kódu je provedeno seřazení vstupních dat v bitově reverzovaném pořadí. Generátor adresy nejdřív převádí číselnou hodnotu na binární číslo, následně toto číslo přečte zezadu a opět konvertuje do celých čísel. Tento proces není plně automatický, při změně velikosti FFT je třeba doplnit případně smazat chybějící nebo přebívající prvky podle vzoru. Následně jsou data přeházena podle nově vygenerovaných adres v bitově reverzovaném pořadí.

Další částí je generátor konstant, který generuje konstanty (otáčecí činitele) z komplexní exponenciály podle předpisu v rovnici (8).

Nejdůležitější částí zdrojového kódu je pak samotný algoritmus FFT, který je napsán tak, aby se co nejlépe přiblížil zamýšlené architektuře v obvodu. Nejdříve jsou vybrány konstanty z banky konstant, následně je provedeno násobení dat vybranou konstantou (v případě implementace FFT do mikroprocesoru by bylo samozřejmě možné spoustu těchto operací násobení vynechat vzhledem k faktu, že dochází často k násobení číslem jedna, chceme-li však věrně simulovat chování v obvodu, je třeba tyto operace provést, protože v hardwarové podobě budou prakticky prováděny taky), sečteny dvojice dat a uloženy do paměti. Hlavním prvkem, kvůli kterému vůbec provádíme modelování v Matlabu (a který by měl co nejlépe simulovat činnost řídící logiky FFT modulu), je však kombinace tří cyklů (jeden cyklus *for* a dva cykly *while*) pomocí kterých jsou adresována data. Tyto cykly budou v obvodu realizovány kombinací čítačů a právě správná funkčnost čítačů, především pak jejich spouštění a nulování, bude pro funkčnost výsledného obvodu kritická.

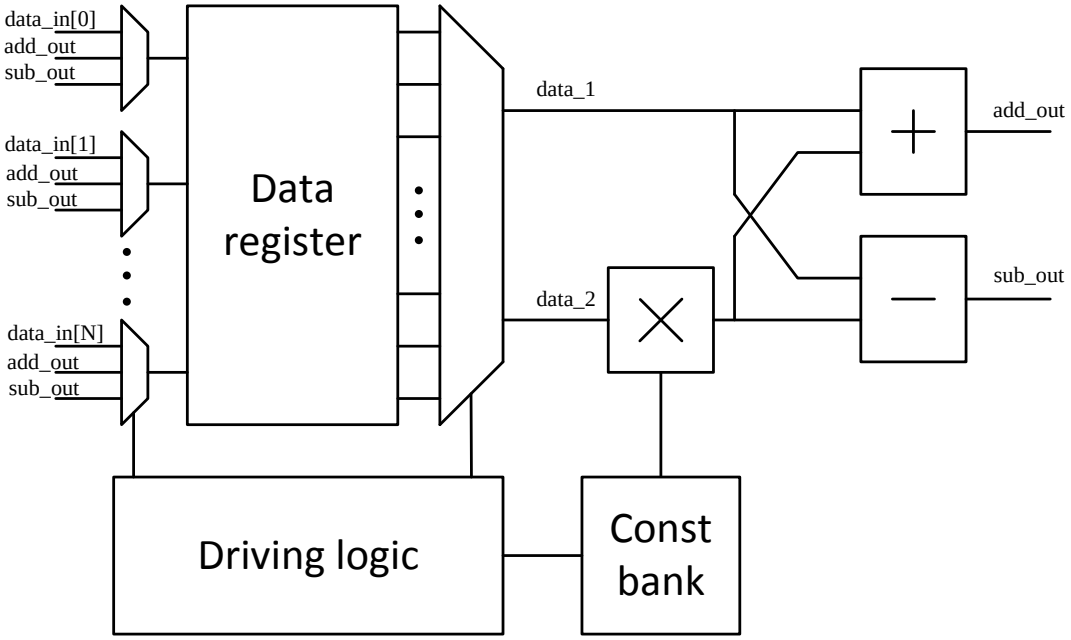
Posledním blokem v referenčním modelu v Matlabu je kontrola vypočtených dat. Je prováděna s pomocí funkce *fft*, která spočítá DFT pro stejná vstupní data a výsledek je srovnáván pomocí vzájemeného odčítání hodnot referenčního modelu a funkce *fft*. V ideálním případě by měl být rozdíl nulový, vzhledem k jistým odlišnostem mezi implementací funkce *fft* a referenčního modelu je však ve výsledku nepatrný rozdíl s odchylkou v řádu 10^{-14}.

2.3 Obecný popis architektury FFT modulu

Na začátku popisu architektury bude vhodné upozornit na některé zásady psaní zdrojových kódů dodržované v této práci. Vzhledem k faktu, že výstupem práce má být univerzální popis FFT modulu, zdrojové kódy jsou psány tak, aby bylo možné změnu každého parametru provést jen změnou jedné konstanty na jednom místě. Všechny tyto parametry jsou uloženy v jednom souboru, kterým je knihovní balík *fft_module_pkg*. Kromě nastavitelných parametrů FFT modulu obsahuje také další konstanty a datové typy, které jsou z nich odvozeny.

Veškeré signály vyskytující se v FFT modulu nebo hlavním modulu jsou tzv. „active high“, což znamená, že pokud se zde bude hovořit a jejich aktivní hodnotě, pak je tím vždy myšlena hodnota ‚1‘.

Podle předchozích dvou odstavců pak bude popsána architektura v této kapitole. Pro velikost algortimu FFT bude používána proměnná *N* (ve zdrojovém kódu je však pro lepší přehlednost použita konstanta *c_FFT_SIZE*), pro celkový počet stupňů výpočtu bude používána proměnná *stage_cnt* (tato hodnota je odvozena z proměnné *N*, ve zdrojovém kódu je tato konstanta nazvána *c_STAGE_COUNT*) a pro bitovou šířku vstupních a výstupních dat proměnná *w* (ve zdrojovém kódu se jedná o konstantu *c_DATA_WIDTH*).



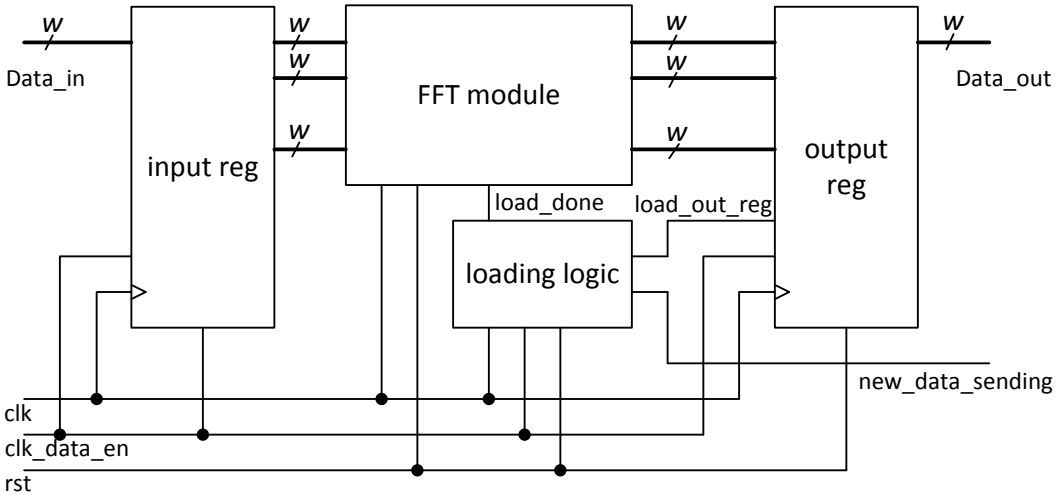
Obrázek 4: Bloková struktura architektury pro implementaci algoritmu FFT

V kapitole 2.1 jsme uvažovali tři hlavní části potřebné pro implementaci algoritmu FFT do hardwarové podoby, nyní se však podívejme trošku blíž na architekturu, kterou bude algoritmus implementován. Bloková struktura samostatného FFT modulu je zakreslena na Obrázku 4.

Na tomto místě je dobré upozornit na strukturu vstupních dat pro takový FFT modul. Data jsou do hlavního registru nahrávána paralelně a výpočet je spuštěn náběžnou hranou na signálu *load_en*. Výsledek je možné získat z hlavního datového registru v paralelním formátu. Pokud je třeba data přijímat a odesílat sériově, což je případ, který v praxi nastane nejčastěji, FFT modul musí být vybaven také registry pro převod sériových dat na paralelní (pro příjem dat) a naopak pro převod dat z paralelního formátu na sériový (pro odesílání dat). Do takového hlavního modulu s registry (*top_fft*) je FFT modul (*fft_module*, oba názvy korespondují s názvy entit ve VHDL kódu) vložen. Schéma zapojení FFT modulu do hlavního modulu je na Obrázku 5.

Vstupními piny hlavního modulu je *w*-bitový datový vstup (*Data_in*), pin pro hodinový signál použitý na čipu (*clk*), pin pro povolení načtení validních dat (*clk_data_en*) a pin pro signál reset (*rst*), jehož aktivací je modul uveden do výchozího stavu. Výstupní piny jsou pro odesílání vypočtených dat (*Data_out*), stejně jako u datového vstupu se jedná o *w*-bitové slovo, a pin oznamující začátek odesílání nového cyklu dat (*new_data_sending*). *w*-bitové slovo, ať už u vstupních nebo výstupních dat,

reprezentuje jednu naměřenou, respektive vypočtenou hodnotu. Struktura vstupních a odesílaných dat bude rozebrána později v kapitole 2.6.



Obrázek 5: Zapojení FFT modulu se vstupním a výstupním registrem

Datová cesta zpracovávaných dat začíná ve vstupním registru, do něhož jsou data načítána ze sériového vstupu. Důležitý je fakt, že vstupní registr je řízen hodinovým signálem *clk*, povolení změny obsahu však probíhá s aktivní hodnotou signálu *clk_data_en* (pro správnou funkčnost je tedy nutné, aby byl tento signál aktivní pouze po dobu jednoho hodinového taktu). Po načtení dat do vstupního registru řídící logika rozhodne o jejich načtení do datového registru, kde jsou data zpracovávána podle algoritmu FFT. Výsledky jsou postupně ukládány do datového registru a dle potřeby přepisovány výsledky z dalšího stupně. Po provedení všech výpočtů řídící logika (modul *driving logic* na Obrázku 5) rozhodne o odeslání dat do výstupního registru. Z výstupního registru odcházejí data po sériové lince pryč. Zde je třeba zmínit, že i výstupní registr pracuje se stejným hodinovým signálem jako registr pro vstupní data, nová data jsou nastavena aktivní hodnotou signálu *clk_data*_en. Tím je zaručeno, že data odcházejí se stejnou frekvencí, s jakou přicházejí do hlavního modulu. Podrobněji budou rozebrány jednotlivé bloky v následujících kapitolách, nejdříve se však podívejme na předpokládanou plochu na čipu, která bude třeba pro implementaci zvolené architektury.

2.3.1 Rozbor předpokládané plochy na čipu

Jak už bylo napsáno v úvodu, pro obvody ASIC je velice důležitá velikost použité plochy na čipu. Proto je nejdříve vhodné analyzovat potřebnou plochu na čipu vzhledem k velikosti FFT. Tabulka 3 obsahuje očekávanou využitou plochu na čipu reprezentovanou počtem použitých registrů, požadovaný hodinový signál vzhledem k frekvenci vstupních dat a počet matematických operací.

První sloupec v Tabulce 3 obsahuje počet vzorků, s kterými je prováděna rychlá Fourierova transformace. Druhý sloupec obsahuje informaci, kolik stupňů je potřeba pro výpočet FFT, zároveň si můžeme všimnout vztahem mezi prvním a druhým sloupcem, kdy počet vzorků odpovídá mocnině čísla 2, přičemž touto mocninou je právě počet stupňů. Další dva sloupce obsahují počet matematických operací, ať už

sčítání nebo násobení, v závislosti na počtu vstupních vzorků. Vztah mezi počtem operací sčítání a počtem vstupních vzorků byl pospán v rovnici (13), počet operací násobení byl popsán v rovnici (14).

Tabulka 3: Tabulka závislosti plochy a hodinového signálu na počtu vstupních vzorků pro šířku slova 14 bitů

Počet vstupních vzorků	Počet stupňů	Počet operací sčítání	Počet operací násobení	minimální frekvence clk / data	Počet sčítaček	Počet násobiček	Počet buněk paměti
2	1	2	1	0,5	9	3	112
4	2	8	4	1	9	3	224
8	3	24	12	1,5	9	3	448
16	4	64	32	2	9	3	896
32	5	160	80	2,5	9	3	1792
64	6	384	192	3	9	3	3584
128	7	896	448	3,5	9	3	7168
256	8	2048	1024	4	9	3	14336
512	9	4608	2304	4,5	9	3	28672
1024	10	10240	5120	5	9	3	57344

Parametry popsané v předchozím odstavci jsou nezávislé na implementaci algoritmu FFT, jedná se o hodnoty, které vyplívají přímo z vlastností tohoto algoritmu. Další sloupce v Tabulce 3 už jsou vztaženy ke zde navržené implementaci tohoto algoritmu.

V pátém sloupci Tabulky 3 je dán poměr mezi frekvencí hodinového signálu na čipu, která je použita pro výpočty, a mezi frekvencí, s jakou přicházejí vstupní data. Tento poměr je určen z počtu kroků výpočtu (tato hodnota není v Tabulce 3 uvedena, nicméně číselně se jedná o stejné hodnoty jako počet operací násobení) a z počtu vstupních vzorků, kdy je požadováno, aby všechny výpočty proběhly v době, kdy přicházejí nová vstupní data. Jedná se o minimální teoretickou hodnotu, skutečný poměr musí být vždy minimálně nejbližší vyšší celé číslo než hodnota udávaná v tabulce. To znamená, že například i pro velikost 16 (čtvrtý řádek v Tabulce 3) je nejnižší reálný poměr frekvencí roven hodnotě 3.

Další dva sloupce informují o počtu použitých násobiček a sčítaček. Co se týče těchto posledních tří parametrů, se vzrůstajícím počtem vstupních vzorků se poměr mezi frekvencí hodinového signálu a frekvencí vstupních dat zvyšuje jen relativně málo a počet sčítaček i násobiček je konstantní. Překážkou pro efektivní využití FFT vyššího řádu je právě množství potřebných paměťových buněk. Jejich počet je určen počtem vstupních vzorků a bitovou šířkou slova, je také počítáno s tím, že vstupní data budou mít jen reálnou složku. V takovém případě vstupní i výstupní registr obsahuje počet paměťových buněk odpovídající součinu velikosti FFT a šířce slova, datový registr v FFT modulu obsahuje dvojnásobek tohoto součinu (je třeba uchovat reálnou i imaginární část komplexního čísla). Počet paměťových buněk (PPB) lze tedy vypočítat z rovnice (15).

$$PPB = N * w * 4 \tag{15}$$

V Tabulce 3 je pro výpočet použita konstantní bitová délka vstupních dat, zvolená hodnota byla 14-bitová šířka slova, která je dostatečná i pro největší zde uváděnou velikost algoritmu FFT. Jak vidíme, hodnoty v posledním sloupci se vzrůstajícím počtem vstupních dat prudce rostou, pro počet vstupních vzorků $N = 1024$ a dané bitové šířce vstupních dat je potřeba použít přes 50 000 registrů, což by zabralo významnou plochu na čipu. Zároveň je třeba počítat s tím, že se jedná jen o odhad, další registry budou třeba pro čítače a případnou synchronizaci signálů. Reálný počet registrů bude oproti hodnotám v Tabulce 3 ještě vyšší, registry pro uchování dat však budou stále nejvýznamnější položkou celkového počtu registrů na čipu.

2.4 Hlavní modul

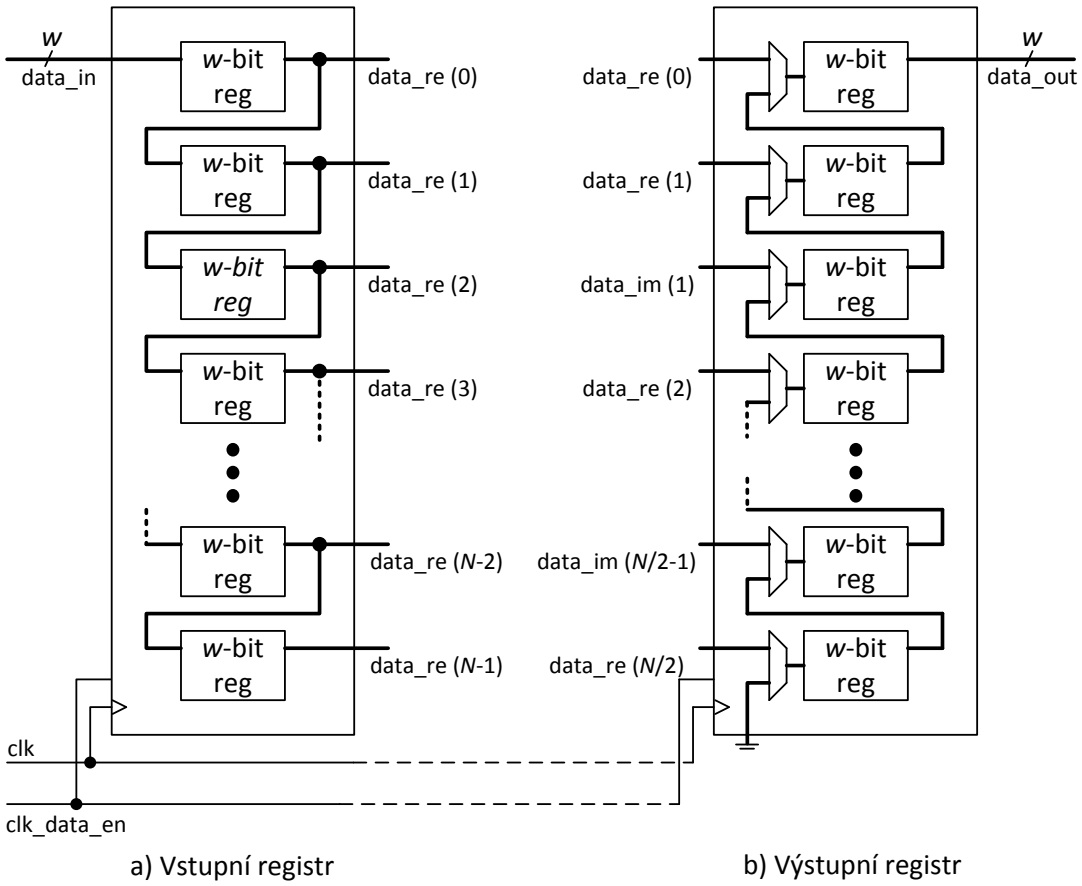
Detailní pohled na výše popsanou architekturu nejdříve začněme u práce s příchozími daty, tedy u hlavního modulu *top_fft*. V tomto modulu jsou dva registry určeny pro načítání příchozích dat a odesílání vypočtených hodnot z modulu. Zároveň je v tomto modulu implementován čítač čítající příchozí data (blok *loading_logic* na Obrázku 4**)**, který zároveň generuje signály pro načtení nových dat do hlavního datového registru v FFT modulu a načtení vypočtených hodnot do výstupního registru. Tyto tři části (dva registry a čítač) nejsou nezbytnou součástí výpočtu rychlé Fourierovy transformace, slouží pouze pro převod sériových vstupních dat na paralelní data v případě vstupního registru nebo naopak pro převod paralelních dat na sériová pro přenos po sériové lince v případě výstupního registru. Proto jsou tyto části umístěny v hlavním modulu a jejich použití je volitelné (viz kapitola 5).

Zdrojový kód hlavního modulu je uložen v souboru *top_fft.vhd* a nalezneme ho i v příloze B.1. V hlavním modulu jsou použity dvě komponenty, první z nich je instance FFT modulu provádějící výpočet algoritmu FFT, druhou z nich instance logiky načítání sériových dat; dále jsou zde vytvořeny instance vstupního a výstupního registru pomocí procesů *input_reg_inst* a *output_reg_inst* a pomocí příkazů *for-generate* jsou oba registry zapojeny jako posuvné registry.

2.4.1 Vstupní a výstupní datový registr

Vstupní registr načítá pouze reálná data ze sériové linky a je zapojen jako posuvný registr. Příchozí data jsou načítána do registru s indexem N - 1 a jsou posouvána do registrů s nižší adresou. Při načtení všech dat jsou tedy první příchozí data uložena v registru s indexem ‚0' a poslední příchozí data v registru s indexem N - 1.

Vstupní registr je řízen hodinovým signálem na čipu *clk*, avšak načítání nových dat a posouvání načtených dat je povoleno aktivní hodnotou signálu *clk_data_en*, který by měl mít délku jedné periody hodinového signálu *clk*. Pokud dojde k načtení všech hodnot, odpovídajících velikosti FFT, řídící logika pro načítání dat pošle příkaz FFT modulu k načtení těchto dat do svého datového registru, data ve vstupním registru však nejsou ihned vymazána, ale jsou zahozena až na výstupu posledního registru. Vstupní registr zapojený jako posuvný registr i s datovými vodiči pro odeslání dat do datového registru je nakreslen na Obrázku 6a.



Obrázek 6: Schéma vstupního a výstupního registru hlavního modulu

Výstupní registr má stejnou velikost jako vstupní, rozdíl je ovšem v tom, že data v něm obsažená nejsou pouze reálná čísla – výsledkem diskrétní Fourierovi transformace jsou vždy komplexní čísla. V případě, že vstupní data jsou reálná čísla, je výsledné spektrum signálu symetrické a stačí odeslat pouze jeho polovinu a velikost výstupního registru je tedy dostatečná. Struktura odchozích dat je popsána podrobněji v kapitole 2.6.

Účelem výstupního registru je převádět data z paralelního formátu v datovém registru FFT modulu na sériová data za účelem odesílání po sériové lince. Proto je výstupní registr stejně jako vstupní registr zapojen jako posuvný registr. Rozdíl je však v tom, že před vstupem do každého registru je multiplexor, který dle dat z řídící logiku přiřazuje na vstup každého registru buď výstup následujícího registru, nebo odpovídající data z datového registru v FFT modulu. Na tomto místě je vhodné se zmínit o tom, že z FFT modulu přicházejí všechna vypočtená data a výběr dat k odeslání probíhá před načtením do výstupního registru až v hlavním modulu. Správné připojení výstupů z FFT modulu do multiplexorů na vstup výstupního registru je ve zdrojovém kódu provedeno pomocí už zmiňovaného příkazu *for-generate* (ve zdrojovém kódu v příloze B.1 je to blok s návěstím *output_reg_load*).

Multiplexory na vstupech výstupního registru jsou vytvořeny pomocí příkazu *for-generate* a jsou všechny řízeny signálem *load_out_reg* generovaným v ovladači načítání nových dat. Při jeho aktivní hodnotě dochází k nastavení nových dat na vstupy výstupního registru.

Stejně jako vstupní register, i výstupní register je řízen hodinovým signálem *clk* a posun dat a tím i odesílání je povolováno aktivní hodnotou na signálu *clk_data_en*. Výstupní registr zapojený jako posuvný registr i s multiplexory je nakreslen na Obrázku 6b.

2.4.2 Ovladač načítání příchozích dat

Ovladač pro načítání příchozích dat (blok *loading_logic* na Obrázku 5, soubor se zdrojovým kódem je nazván *loading_logic.vhd* a je možné ho nalézt v příloze B.2) je realizován jako binární čítač čítající od 0 po N - 1. Jedná se o čítač s úplným cyklem, protože počet vstupních vzorků algoritmu FFT je vždy mocnina čísla 2. Počet klopných obvodů v tomto čítači je tedy roven hodnotě $\log_2(N)$.

Blok *loading_logic* generuje dva signály. První z nich (ve zdrojovém kódu nazván *load_out_reg*) slouží k nastavení multiplexorů výstupního registru tak, aby do výstupního registru byla načtena vypočtená data při příchodu následujícího pulsu na signálu *clk_data_en*. Aktivní hodnota je vygenerována při hodnotě čítače $N - 2$, značí tedy příchod předposledního vzorku dat. Tak je zaručeno, že do výstupního registru jsou nahrána vypočtená data z datového registru v FFT modulu ještě předtím, než v tomto registru dojde k jejich přepsání novými daty po příchodu posledního vzorku.

Druhý signál (ve zdrojovém kódu nazván *load_done*) je generován při příchodu posledního vzorku dat a slouží také jako příkaz pro FFT modul, aby nahrál do svého datového registru nová data a spustil výpočet algoritmu FFT. Tento signál je také přiveden na výstup hlavního modulu a slouží jako informace o začátku odesílání cyklu nových dat (jeho aktivní hodnota značí hodnotu prvního vzorku na výstupu, tedy stejnosměrné složky spektra). Na výstupu hlavního modulu je tento signál přejmenován na *new_data_sending*, aby lépe vystihoval význam tohoto výstupního signálu pro další bloky.

2.5 Modul Rychlé Fourierovy transformace

Modul rychlé Fourierovy transformace zajišťuje výpočet algoritmu FFT. Jeho blokové schéma je nakresleno na Obrázku 4. Struktura a funkce tohoto modulu byla nastíněna už v kapitole 2.3, nicméně protože se jedná o stěžejní část celého návrhu, bude třeba tento blok rozebrat detailněji. Pro popis funkčnosti je vhodné tento modul rozdělit na tři hlavní části dle činnosti, kterou vykonávají. Nalezneme zde tedy paměťovou část, aritmetickou část a řídící logiku.

Zdrojový kód modulu FFT je uložen v souboru *fft_module.vhd* a lze ho nalézt v Příloze B.3. Každý ze tří jmenovaných základních bloků je popsán jiným způsobem, paměťová část je zapsána pomocí příkazů *for-generate* a *process*, aritmetická část je tvořena procedurami uloženými v knihovním balíku *fft_module_pkg.vhd* (ten je možné nalézt v Příloze 6) a řídící logika je vložena jako komponenta.

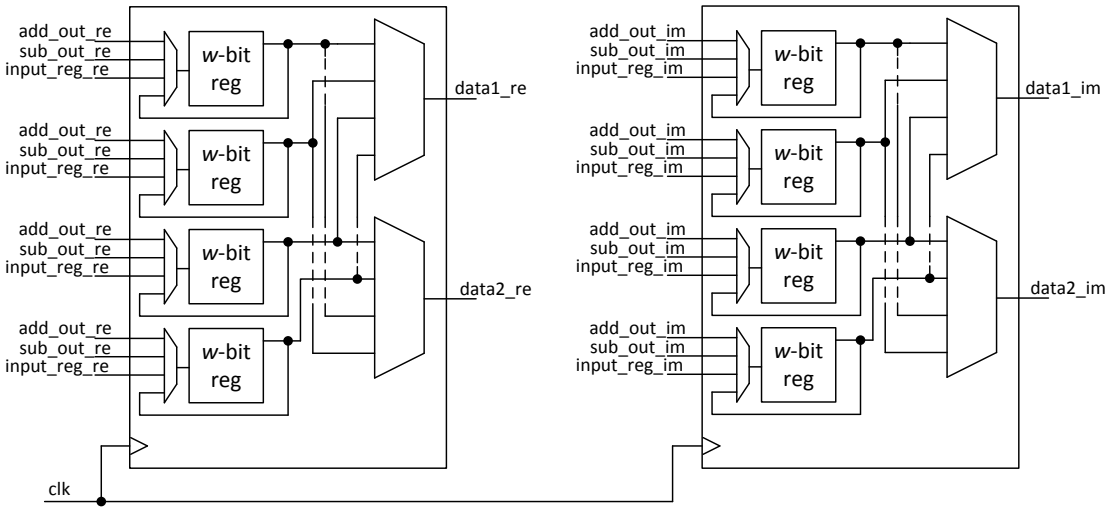
Pokud se podíváme na Obrázku 5, FFT modul je do hlavního modulu připojen pomocí pole signálů o bitové šířce w. Protože FFT modul může být použit i bez hlavního modulu, je třeba upozornit na strukturu tohoto pole. Jedná se o pole signálů dvojnásobku velikosti algoritmu FFT – to znamená, že vstupem *DATA_in* jsou do FFT modulu přivedena komplexní čísla. Pokud chceme do FFT modulu nahrát jen reálná

čísla, všechny vstupy s lichým indexem je třeba nastavit do nuly.

Vstupy signálu *DATA_in* se sudým indexem jsou přivedeny do části datového registru pro reálnou část čísel. Přeskládání dat do bitově reverzovaného pořadí je zajištěno až v FFT modulu, na vstup je třeba přivést hodnoty v normálním pořadí. První vzorek (na časové ose by to byl ten nejstarší vzorek) je připojen na vstup s indexem ‚0', poslední vzorek (na časové ose ten nejnovější) je pak připojen na poslední vstup s indexem 2*(*N*-1).

2.5.1 Paměťová část modulu FFT

Hlavní datový registr v FFT modulu slouží k uchovávání výsledků jednotlivých kroků výpočtu algoritmu FFT. Vzhledem k tomu, že ukládá jak reálnou tak i imaginární část komplexních čísel, je jeho velikost dvojnásobná oproti vstupnímu registru. Na Obrázek 7 je nakreslen datový registr a to včetně multiplexorů na vstupních a výstupních portech datového registru, na levé části obrázku se jedná o registry ukládající reálnou část čísla a registry na pravé části uchovávají imaginární část čísla.



Obrázek 7: Datový registr FFT modulu s multiplexory na vstupu i výstupu

Multiplexory na vstupních portech jsou určeny k řízení dat, které jsou do registru zapsány. Každý vstup registru má svůj multiplexor, který určuje data, která budou zapsána.

Multiplexory na výstupních portech určují data, která budou odesílána do aritmetické části. Do aritmetické části vždy vstupuje dvojice komplexních hodnot, proto jsou tyto multiplexory čtyři – dva pro reálnou část dat a dva pro imaginární část dat.

Signály generované řídící logikou jsou pro multiplexory na vstupních i výstupních portech stejné (ve zdrojovém kódu jsou to signály *data1_addr* a *data2_addr*). Použít stejné signály je možné z toho důvodu, že data jsou ukládána do stejných registrů, z kterých byla vyčtena. Pokud jsou v aritmetické části použity registry kvůli zvýšení rychlosti (viz kapitola 2.5.2), je třeba také oba řídící signály pro multiplexory na vstupních portech datového registru zpozdit o jeden takt hodinového signálu pomocí klopných obvodů.

VHDL popis hlavního datového registru je umístěn přímo do zdrojového kódu FFT modulu (Příloha B.3). Vytvoření registru je provedeno v procesu *main_data_reg*. Pole multiplexorů na vstupu datového registru je vytvořeno pomocí příkazu *for-generate* a multiplexory na výstupu datového registru jsou vytvořeny jako funkce popsané v knihovním balíku *fft_module_pkg*.

Velikost pole multiplexorů na vstupu datového registru je dána velikostí algoritmu FFT a ve zdrojovém kódu bylo nazváno *data_reg_input_mul*. Každý multiplexor vybírá ze čtyř možných datových signálů na základě tří řídících signálů, přičemž nejvyšší prioritu mají data ze vstupního registru (v kódu je možné si povšimnout, že ty jsou přivedeny v bitově reverzovaném pořadí, tento přepočet adres zajišťuje funkce *bit_reverse* popsaná v knihovním balíku *fft_module_pkg*), následně jsou to data z aritmetické části (vyšší prioritu mají výsledky ze sčítačky, ale to je jen formální záležitost, protože nemůže dojít k situaci, kdy by se do stejného registru měly zapsat oboje data z aritmetické části) a nejnižší prioritu mají data z výstupu datového registru, tedy hodnoty, které jsou v datovém registru už uloženy a v daném kroku výpočtu se nemění.

Ovládání pole vstupních multiplexorů při zápisu dat z aritmetické části je realizováno pomocí binárního dekodéru. Dekodér *data_addr_decoder* převádí binárně kódovanou adresu dat generovanou v řídící logice na kód 1 z N, pomocí kterého je možné řídit multiplexory na vstupech datového registru. V případě použití registrů pro rozdělení výpočtu v aritmetické části (viz. kapitola 2.5.2 a kapitola 5) je třeba i adresu na vstupních multiplexorech zdržet o jeden hodinový takt, aby byla vypočtená data uložena do stejného registru. Toto zpoždění je kvůli úspoře klopných obvodů realizováno už na signálu z řídící logiky před dekódováním na kód 1 z N. Pokud je výpočet ukončen a čeká se na nová data, dekodér všechny řídící signály multiplexory na vstupech datového registru do hodnoty ‚0'. Díky tomu je zaručeno, že až do příchodu příkazu k načtení nových dat je na vstup datového registru přiveden jeho výstup a všechna data jsou uchována.

Multiplexory na výstupu datového registru jsou vytvořeny jako funkce *get_data_from_reg* popsané v knihovním balíku *fft_module_pkg*. Tato funkce má dva vstupní parametry, výstupní signály datového registru a adresu požadovaných dat. Na základě této adresy funkce vrací hodnotu uloženou v datovém registru. I při použití registrů pro rozdělení výpočtu v aritmetické části je použita adresa aktuální (tedy bez zpoždění o jeden hodinový takt). Funkce je použita dvakrát, protože z datového registru je vždy vybírána dvojice dat.

Do paměťové části FFT modulu patří také paměť ROM obsahující hodnoty otáčecího činitele. Paměť ROM je ve VHDL popisu vytvořena pomocí dvou konstant pro reálnou a imaginární část otáčecího činitele. Tyto konstanty jsou uloženy v samostatném souboru *const_bank.vhd* (Příloha B.6, konstanty pro $N = 16$, $w = 8$) a aktuální potřebné hodnoty jsou vybírány signálem *const_addr* generovaném v řídící logice.

2.5.2 Aritmetická část

Aritmetická část zahrnuje logické bloky provádějící matematické operace v pevné řádové čárce. Modul FFT obsahuje dvě komplexní sčítačky a komplexní násobičku dat. Návrh obvodu implementujícího algoritmus FFT na čip byl proveden se snahou ušetřit co nejvíce místa na čipu a z toho důvodu jsou použity právě dvě komplexní sčítačky (ačkoliv se může zdát, že vzhledem k ploše by bylo vhodnější použít pouze jednu komplexní sčítačku, ve skutečnosti tomu tak není, bylo by nezbytné implementovat do obvodu další registry a doplňkovou logiku a došlo by ke zvětšení použité plochy na čipu a zároveň zpomalení výpočtu na poloviční rychlost).

Komplexní sčítačky jsou určeny ke sčítání komplexních čísel. U komplexních čísel se sčítá zvlášť reálná a zvlášť imaginární část, komplexní sčítačka tedy obsahuje dvě *w*-bitové sčítačky, každá operuje buďto s reálnou nebo imaginární částí čísla. Při sčítání vak může dojít k přetečení, to je ošetřeno zahozením posledního bitu (LSB) a posunutím čísla o jeden bit doprava. Bitový posun probíhá pro každá data jednou v každém stupni, pro výstupní signál to znamená, že je oproti vstupnímu signálu zmenšen právě 2^{stage}-krát, kde *stage* odpovídá počtu stupňů výpočtu. Zdrojový kód obou sčítaček (jedna pracuje v módu odčítání) je uložen v knihovním balíku *fft_module_pkg* jako procedury *cmpl_adder* a *cmpl_subtractor*. Všechny sčítačky v návrhu jsou vygenerovány s pomocí knihovny IEEE.NUMERIC_STD.

Násobení dat koeficienty zajišťuje násobička komplexních čísel. Násobení komplexních čísel je komplikovanější než sčítání – nelze násobit pouze reálné části a imaginární části komplexních čísel zvlášť, při násobení se navzájem ovlivňují. Násobení komplexních čísel v součtovém tvaru probíhá podle vzorce

$$(a+bi)*(c+di)=(a*c-b*d)+i*(a*d+b*c) \tag{16}$$

V rovnici (16) vidíme, že násobení komplexních čísel probíhá pomocí čtyř operací násobení reálných čísel, dvou operací sčítání a jedné operace odčítání. Použití násobičky na čipu je však nežádoucí, plocha zabraná násobičkou je podstatně větší než plocha zabraná sčítačkami. Proto je vhodné si rovnici (16) upravit do formy s méně operacemi násobení [1]. Vyjděme z rovnice (16) a nejprve se zaměřme na reálnou část výsledku.

$$a*c-b*d=a*c+b*c-b*d-b*c=(a+b)*c-(c+d)*b \tag{17}$$

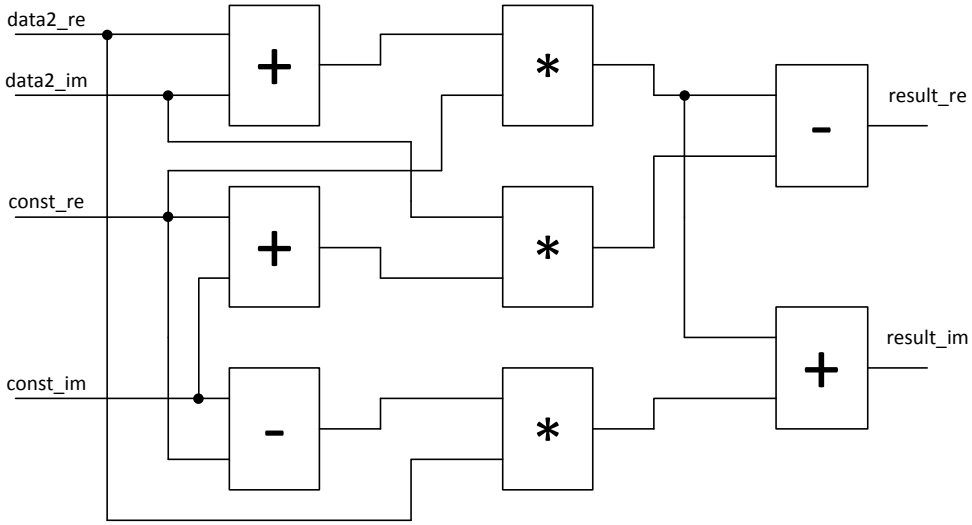
V rovnici (17) jsme reálnou část výsledku rozšířili o $b*c - b*c$, čímž se výsledek nijak nezměnil, ale umožnilo nám to vytknout z prvního součtu c a z druhého součtu b. Na první pohled se zdá, že nejde o nijak užitečnou úpravu, při podobné úpravě imaginární části výsledku násobení komplexních čísel jako v rovnici (18) se výhoda už objeví.

$$a*d+b*c=a*d-a*c+b*c+a*c=(d-c)*a+(a+b)*c \tag{18}$$

V rovnici (17) a rovnici (18) se vyskytuje stejný člen $(a + b) * c$, čímž jsme eliminovali jednu operaci násobení a výsledná úprava je shrnuta v rovnici (19)

$$m_1 = (a+b)*c \qquad m_2 = (d-c)*a \qquad m_3 = (c+d)*a$$
$$(a+bi)*(c+di) = (m_1 - m_3) + i*(m_1 + m_2) \tag{19}$$
$$(a+bi)*(c+di) = (a+b)*c - (c+d)*a + i*((a+b)*c + (d-c)*a)$$

Po použití úprav z rovnice (19) vypadá výsledný obvod pro násobení komplexních čísel jako na Obrázku 8. Výše popsané úpravy v rovnici (19) pomohla dosáhnout úspory jedné násobičky za cenu použití tří dalších sčítaček a prodloužení datové cesty. Pokud by byla násobička popsána podle rovnice (16), nejdelší datová cesta by vedla přes sériovou kombinaci sčítačky a násobičky. V případě úspory jedné násobičky se datová cesta prodloužila o jednu sčítačku. V rámci celého návrhu se jedná o nejdelší datovou cestu a právě v závislosti na použité technologii je tato část kritická pro volbu frekvence hodinového signálu.



Obrázek 8: Schéma násobičky komplexních čísel

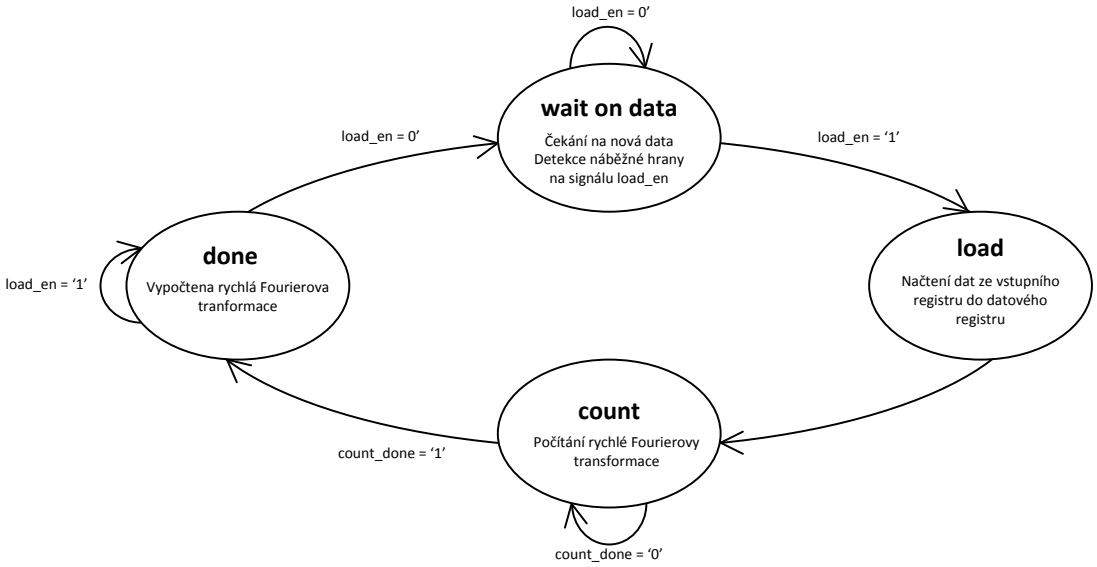
Zdrojový kód násobičky komplexních čísel odpovídá popisu v rovnici (19). Stejně jako sčítačky komplexních čísel i tato násobička je popsána jako procedura. Její kód ve VHDL nalezneme v Příloze B.5 v knihovním balíku *fft_module_pkg* jako proceduru s názvem *cmpl_multiplier*.

Pokud se podíváme na Obrázku 4, data z násobičky komplexních čísel jsou přivedena přímo na vstup obou sčítaček komplexních čísel. Takové sériové spojení násobičky a sčítaček však vytváří dlouhou datovou cestu, která je omezující pro frekvenci hodinového signálu použitelnou na čipu. Při použití FFT modulu jako součást složitějšího obvodu pro zpracování digitálního signálu toto může být výrazný problém, proto je v architektuře možnost použít registry zapojené mezi násobičku a obě sčítačky, které nejdelší datovou cestu znatelně zkracují. Rozdělení datové cesty v aritmetické části mezi dva takty hodinového signálu je možné volbou konstanty *c_SYNC_ASYNC_ARITHMETIC* v knihovním balíku *fft_module_pkg* (hodnota ‚1' znamená, že registry budou použity). Rozdíl mezi maximální frekvencí hodinového signálu *clk* bude ukázán při syntéze navržené architektury do obvodů FPGA v kapitole 4.

2.5.3 Řídící logika

Řídící logika je určena k ovládání FFT modulu a tím i k řízení výpočtu algoritmu FFT. Problematiku implementace algoritmu FFT lze chápat jako záležitost adresace správných dat a správného otáčecího činitele v závislosti na výpočetním kroku. Hlavním úkolem řídící logiky je právě výpočet adres dat v datovém registru a adres otáčecího činitele v paměti pro konstanty. Kromě toho také kontroluje stav, ve kterém se modul nachází, ovládá načítání a odesílání dat. Zdrojový kód je uložen v souboru *driving_logic.vhd* a je možné ho nalézt i v příloze B.4.

FFT modul pracuje ve čtyřech stavech, nejprve čeká na nová data, následně nová data načítá a zpracovává je. Pokud je po výpočtu signál *load_en* stále aktivní, čeká na jeho nulovou hodnotu. Tímto je zajištěno, že načítání nových dat je spuštěno jen náběžnou hranou tohoto signálu. Graf stavového automatu je na Obrázku 9.

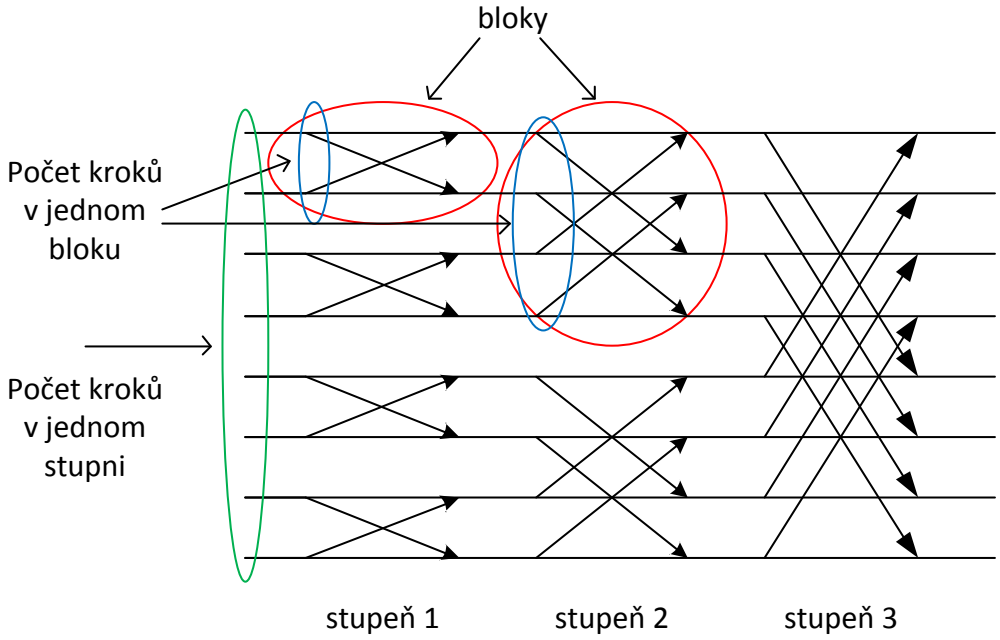


Obrázek 9: Stavový automat řídící logiky

Po spuštění FFT modulu nebo jeho resetování je stavový automat nastavený do počátečního stavu *done* a čeká na náběžnou hranu na signálu *load_en* značící povolení načtení nových dat. Nová data jsou načtena v jednom taktu hodinového signálu *clk* (stav *load*). Následně stavový automat řídící logiky přechází do stavu *count*, během kterého je prováděn výpočet algoritmu FFT. Výpočet je dokončen, pokud proběhl poslední výpočet v posledním stupni – tyto hodnoty jsou určeny dvěma čítači *stage counter* a *progress counter* které budou popsány později v této kapitole. Stavový automat nyní přechází do stavu *done*, ve kterém kontroluje hodnotu signálu *load_en*. Pokud je signál *load_en* v tomto okamžiku v aktivní hodnotě, je předpokládáno, že na vstup nepřicházejí nová data a stavový automat bude čekat ve stavu *done*, dokud se na signálu *load_en* neobjeví sestupná hrana. Pokud je signál *load_en* nastaven do hodnoty ‚0', stavový automat přechází do stavu *wait on data* a v něm čeká na aktivní hodnotu (resp. náběžnou hranu) signálu *load_en*, po které jsou načtena nová data do datového registru.

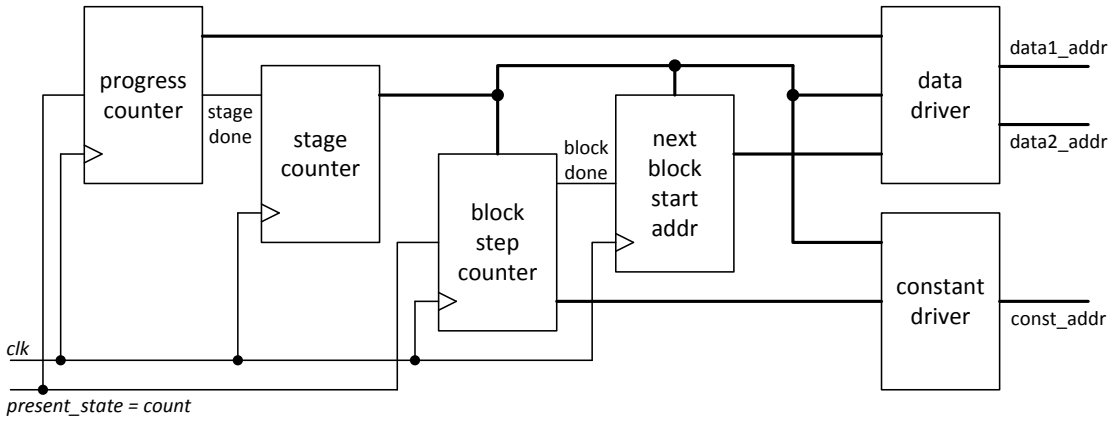
K implementaci algoritmu rychlé Fourierovy transformace bylo použito tří čítačů, z jejichž vzájemné kombinace lze vždy určit správnou adresu dat a koeficientů v paměti. Všechny tyto čítače jsou řízeny hodinovým signálem na čipu.

Pro vysvětlení funkce jednotlivých čítačů je vhodné se nejprve podívat na motýlkový diagram graficky znázorňující algoritmus rychlé Fourierovy transformace. Na Obrázku 10 je tento algoritmus rozkreslen spolu s několika klíčovými pojmy pro pochopení navrženého algoritmu výpočtu adres. Postup transformace je rozdělen do jednotlivých stupňů, v každém z nich můžeme najít imaginární bloky výpočtů, které postupně probíhají s daty, jejichž adresa se zvyšuje vždy o hodnotu ‚1'. Po skončení výpočtu v tomto imaginárním bloku se adresa následujících dat v novém bloku zvýší podle aktuálního stupně a začíná se počítat další blok. Po vypočtení poslední dvojice dat výpočet přechází do dalšího stupně a data se berou opět z prvního registru.



Obrázek 10: Motýlkový diagram se znázorněnými bloky a stupni výpočtu

Výhodou takového rozdělení motýlkového diagramu jako na Obrázku 10 je snadná algoritmizace výpočtu adresy jen pomocí aktuálních hodnot v čítačích. Každá z výše popsaných části motýlkového diagramu odpovídá jednomu konkrétnímu čítači. V návrhu tedy nalezneme čítač, který počítá současný stupeň výpočtu, další, který počítá celkový počet kroků výpočtu v jednom stupni a čítač počítající aktuální krok v každém bloku. S hodnotami v čítačích pak pracuje logika, která z nich určuje adresu aktuálně požadovaných dat v datovém registru a adresu konstant v paměti. Zapojení čítačů a dalších bloků v řídící logice je na Obrázku 11.



Obrázek 11: Zapojení čítačů pro výpočet adresy dat a konstant v řídící logice FFT modulu

První použitý čítač je nazván *progress counter*. Jeho úkolem je počítat aktuální krok výpočtu v rámci jednoho stupně. Čítá od hodnoty 0 do hodnoty $N/2$ - 1, v jednom taktu hodinového signálu je vypočtena vždy dvojice nových dat, počet kroků v jednom stupni je tedy $N/2$. Velikost čítače je z této hodnoty odvozena, počet klopných obvodů potřebných pro jeho realizaci je tedy $\log_2(N)$-1. Čítač *progress counter* prakticky funguje jako časovač (počítá tedy hodinové pulsy), jeho čítání je povoleno pouze tehdy, pokud je stavový automat ve stavu *count*. Výstupem čítače je hodnota aktuálního kroku, kterou využívá kombinační logika pro výpočet adresy dat a také signál *stage_done*, jehož aktivní hodnota je vygenerována při maximu čítače, tedy po posledním kroku v současném stupni a značí dokončení všech kroků v aktuálním stupni. Tímto signálem je řízen další čítač nazvaný *stage counter*.

Další použitý čítač je nazván *stage counter* a slouží k určení aktuálního stupně výpočtu. Jeho rozsah je tedy dán počtem stupňů výpočtu odvozených z velikosti FFT. V tomto případě se jedná o čítač čítající pulsy na signálu *stage_done*, jeho čítání je povoleno neustále, protože aktivní hodnota signálu *stage_done* může být vygenerována čítačem *progress counter* pouze ve stavu *count*. Výstupním signálem čítače *stage counter* je signál *stage* obsahující informaci o aktuálním stupni výpočtu. Čítač *stage counter* pracuje v rozsahu od 0 do $\log_2(N)$-1, první stupeň výpočtu odpovídá hodnotě signálu *stage* = 0, druhý stupeň pak hodnotě *stage* = 1 atd. Pokud bude v následujícím textu použit výraz *stage*, je tím tedy myšlena hodnota tohoto signálu, která odpovídá aktuálnímu stupni výpočtu zmenšenému o jedničku. Tento signál využívají zbývající bloky pro výpočty.

Třetím čítačem je *block step counter*, který počítá krok výpočtu v aktuálním bloku. Podobně jako čítač *progress counter* funguje jako časovač, čítá takty hodinového signálu *clk* a je povolen, jen pokud je stavový automat ve stavu *count*. Zásádní rozdíl mezi těmito čítači je nastavení maximální hodnoty, do které čítání probíhá; zatímco čítač *progress counter* čítá v celém rozsahu, rozsah čítače *block step counter* je omezen podle aktuálního stupně výpočtu signálem *stage*. Čítač pracuje v dynamickém rozsahu od 0 do $2^{(stage)}$- 1, přičemž interval čítání se mění podle aktuálního stupně výpočtu a odpovídá velikosti aktuálního bloku výpočtu. Počet klopných obvodů pro realizace tohoto čítače je dán největším rozsahem pro čítání hodnot. Největší rozsah je stejný jako u čítače *progress counter*, protože v posledním stupni je vždy jeden blok a počet kroků v něm tedy odpovídá počtu kroků v jednom stupni.

Čítač *block step counter* generuje dva výstupní signály, prvním z nich je hodnota kroku v aktuálním bloku výpočtu, což je vlastně aktuální hodnota tohoto čítače. S touto hodnotou pracuje kombinční logika pro výpočet adresy otáčecího činitele v paměti pro konstanty.

Druhým generovaným signálem je signál *block_done*, kterým je řízen další blok pro výpočet počáteční adresy dalšího bloku. Tento blok je nazván *next block start addr* a je realizován pomocí registrů a kombinační logiky. K počáteční adrese předchozího bloku, která je uložena v registrech tohoto bloku, je při příchodu aktivní hodnoty signálu *block_done* přičtena hodnotu odpovídající velikosti aktuálního bloku. Velikost aktuálního bloku je stejně jako u rozsahu čítače *block step counter* rovna $2^{(stage)}$. S počáteční adresou nového bloku následně pracuje kombinační logika pro výpočet adresy dat.

K samotnému vypočtení adresy potřebných dat v registru a konstant v paměti slouží dva bloky – *data driver* a *constant driver*. Tyto bloky jsou tvořeny pouze kombinační logikou a pracují s aktuálními hodnotami v čítačích a vzájemnou kombinací jejich výstupních signálů určují aktuální adresu, ať už dat nebo konstant.

Hodnoty otáčecího činitele pro výpočet algoritmu FFT jsou uloženy v paměti ROM jako konstanty (viz. kapitola 2.5.1), přičemž jejich adresa v paměti odpovídá proměnné *k* z rovnice (8). Ukolem bloku *constant driver* je generování adresy pro čtení aktuálně potřebné konstanty z paměti ROM. Pokud se podíváme na motýlkový diagram, je jasně vidět, že hodnota otáčecího činitele je závislá na velikosti počítaného bloku a aktuálním kroku výpočtu v takovém bloku. Kombinační logika *constant driver* proto pracuje s aktuální hodnotou čítače *block step counter* a aktuálním stupněm výpočtu, který jasně určuje velikost aktuálního bloku výpočtu. Řešení adresace je provedeno pomocí bitového posunu signálu z čítače *block step counter*. K bitovému posunu dochází v závislosti na stupni výpočtu, s vyšším stupněm výpočtu se počet posunutých bitů snižuje. V rovnici (20) je pro zjednodušení použit operátor pro bitový posun z jazyka C.

$$const\_addr = block\_step\_cnt << \left(stage_{\max} - stage + 1\right) \tag{20}$$

kde: *const_addr* je adresa konstanty v paměti ROM

block_step_cnt je signál z čítače *block step counter*

$stage_{max}$ je poslední stupeň výpočtu

stage je aktuální stupeň výpočtu

Popis pomocí bitového posunu je užitečný pro pochopení, jak jsou adresy konstant v paměti určeny, při syntéze však taková konstrukce může způsobovat problémy. Ve zdrojovém kódu *driving_logic* v příloze B.4 je pro generování adresy konstant použita funkce *get_const_addr* z knihovního balíku *fft_modul_pkg*. Tato funkce je tvořena příkazem *case*, přičemž jednotlivé hodnoty v tomto příkazu jsou vytvořeny generátorem FFT modulu, který bude popsán v kapitole 5.

Blok *data driver* slouží k výpočtu adres dat v hlavním datovém registru pro aktuální krok výpočtu. Kvůli potřebě dvojice dat pro každý výpočet generuje tento blok dvě adresy současně, adresa první z dvojice dat je vypočtena jako součet signálů z bloku *next block start addr* a čítače *progress counter*. Adresa druhých dat má vždy hodnotu právě o velikost poloviny bloku vyšší oproti adrese prvních dat a je z ní

odvozena s pomocí hodnoty čítače *stage counter*. Stupeň výpočtu už není třeba uvažovat při výpočtu první adresy dvojice dat, protože tato informace je obsažena už v signálu z bloku *next block start counter* a ovlivňuje výpočet této adresy nepřímo. Předpis pro výpočet obou adres je zapsán v rovnici (21), ve zdrojovém kódu řídící logiky v příloze B.4 je to proces nazvaný *data_driver*.

$$\begin{aligned} data1\_addr &= progress\_cnt + next\_block\_start\_addr \\ data2\_addr &= data1\_addr + 2^{stage} \end{aligned} \tag{21}$$

2.6 Struktura vstupních a výstupních dat

Matematické operace při zpracovávání dat FFT modulem jsou prováděny v pevné řádové čárce, tím je také omezena přesnost výpočtu a je třeba dbát na správný formát vstupních dat pro dosažení co nejpřesnějších výsledků. Nejdříve bude vhodné vysvětlit termín váhový koeficient při počítání v pevné řádové čárce.

Váhový koeficient je poměr mezi celočíselnou reprezentací nějakého čísla a jeho skutečnou hodnotou. Je to tedy koeficient, kterým je třeba vynásobit reálné číslo, abychom získali jeho reprezentaci v pevné řádové čárce a naopak, když hodnotu v pevné řádové čárce tímto koeficientem vydělíme, získáme jeho reálnou hodnotu. S tímto je také spojen problém zaokrouhlení. Je zřejmé, že ne vždy je možné reálné číslo vynásobit takovým váhovým koeficientem, aby byl výsledek celočíselný, proto se desetinná místa po vynásobení váhovým koeficientem zahazují.

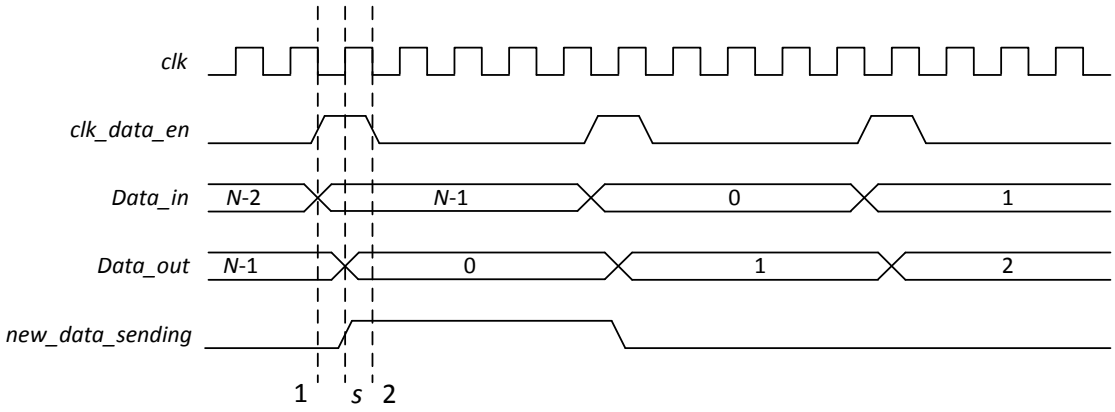
Na tomto místě je vhodné se zmínit ještě o jedné vlastnosti DFT, výsledek DFT dává informaci o spektru signálu, o amplitudě a fázi jednotlivých harmonických složek, nikoliv však přímo hodnotu amplitudy. Pokud bychom například uvažovali stejnosměrný signál (např. napětí 1 V) a provedli transformaci pro 64 vstupních vzorků, výsledkem bude hodnota 64 pro stejnosměrnou složku spektra, což jak vidíme, není velikost vstupního signálu. Výsledek DFT je tedy třeba ještě vydělit počtem vstupních vzorků.

Při výpočtu algoritmu FFT v pevné řádové čárce je však nutné při sčítání dat zahodit poslední bit, aby šířka signálu zůstala stejná. Tím dojde k bitovému posunu výsledku, což odpovídá dělení číslem 2. Toto se děje se všemi daty v každém stupni výpočtu, na konci výpočtu jsou všechna výsledná data vydělena hodnotou $2^{(stage\_cnt)}$, to je ovšem velikost FFT, kterou bychom dělili výsledek, abychom získali hodnoty amplitud harmonických složek. Tento fakt je třeba vzít na zřetel při dalším zpracování vypočtených dat.

Přesnost vypočtených dat je dána dvěma faktory, velikostí algoritmu FFT a bitovou šířkou dat. Je logické, že při větší bitové šířce budou vypočtená data přesnější, výsledek však také nepříznivě ovlivňuje velikost algoritmu FFT. V každém stupni výpočtu dochází k zaokrouhlení výsledků a toto zaokrouhlení se může projevit na některých výstupních hodnotách. Je proto nutné volit dostatečnou bitovou šířku, aby vliv zaokrouhlení na výsledek byl minimální. Bitová šířka dat by měla být vždy volena alespoň o dva bity vyšší než je počet stupňů. Dobré přesnosti výsledků je pak dosaženo při bitové šířce, která je větší než počet stupňů výpočtu alespoň o čtyři bity.

Váhový koeficient vstupních dat by měl být volen podle maximálního rozsahu vstupních hodnot tak, aby byla plně využita bitová šířka slova. První bit je rezervován pro znaménkový bit, nejvyšší hodnota je tedy reprezentována nulou následovanou samými jedničkami a nejnižší hodnota je jednička následovaná samými nulami.

Při odesílání dat do FFT modulu a při čtení dat z modulu je třeba dbát na správné časování jednotlivých signálů, aby byla vstupní data vzorkována FFT modulem ve správném okamžiku a stejně tak aby byla čtena správná data z FFT modulu. Při návrhu bylo předpokládáno, že příchozí signály (ať už signál se vstupními daty *Data_in* nebo povolovací signál *clk_data_en*) jsou synchronizovány s hodinovým signálem. Časování jednotlivých signálů z pohledu vstupu a výstupu hlavního modulu je na Obrázku 12.



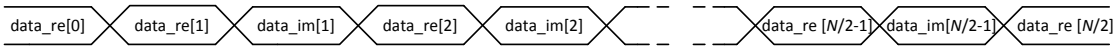
Obrázek 12: Časování vstupních a výstupních signálů z pohledu vstupu a výstupu FFT modulu

Načítání dat do FFT modulu je povoleno aktivní hodnotou signálu *clk_data_en*. Vzhledem k tomu, že celý FFT modul je synchronní s hodinovým signálem *clk,* jsou i nová data do registrů načtena při jeho náběžné hraně. Pro správné načtení příchozích dat je tedy třeba splnit dvě podmínky. První podmínkou je, aby doba aktivní hodnoty povolovacího signálu *clk_data_en* trvala maximálně jednu periodu hodinového signálu *clk* (resp. během aktivní hodnoty signálu *clk_data_en* může nastat jen jedna náběžná hrana hodinového signálu *clk*) a druhou podmínkou je ustálená hodnota signálu vstupních dat *Data_in* v okamžiku načítání.

Na Obrázku 12 jsou vyznačené tři časové okamžiky 1, 2 a *s*. První dva jmenované časové okamžiky byly použity při verifikaci pro nastavení nových hodnot na vstup (okamžik 1) a pro vzorkování odchozích dat (okamžik 2). Časový okamžik *s* (sample) značí okamžik, kdy jsou data načtena do hlavního modulu a kdy se objeví nová data na výstupu. V zásadě je tedy nutné, aby byla vstupní data stabilní pouze v tomto časovém okamžiku, vhodnější však bude zajistit jejich stabilitu alespoň po dobu trvání aktivní hodnoty signálu *clk_data_en.*

Nová výstupní data se na výstupu *Data_out* objevují při aktivní hodnotě povolovacího signálu *clk_data_en* a současně náběžné hraně hodinového signálu *clk*. Ve výsledku výpočtu DFT pro reálná vstupní data se vyskytuje symetrie, díky které je možné odesílat pouze polovina vypočtených hodnot bez ztráty informací[1]. Reálná část spektra signálu je symetrická podle poloviční hodnoty vzorkovací frekvence, imaginární část spektrálních složek signálu odpovídá zase liché funkci se středem v polovině

vzorkovací frekvence. Dvě hodnoty fáze spektra signálu jsou vždy nulové – imaginární část stejnosměrné složky signálu (první vypočtený vzorek na Obrázku 13, jsou to data s adresou ‚0') a imaginární část kmitočtu odpovídající polovině vzorkovací frekvence (data s adresou ‚*N*/2'). Pořadí odesílaných dat je nakresleno na Obrázku 13. Vidíme, že ze stejnosměrné složky je odesílána pouze reálná část a stejně je tomu pro hodnotu poloviny vzorkovací frekvence.

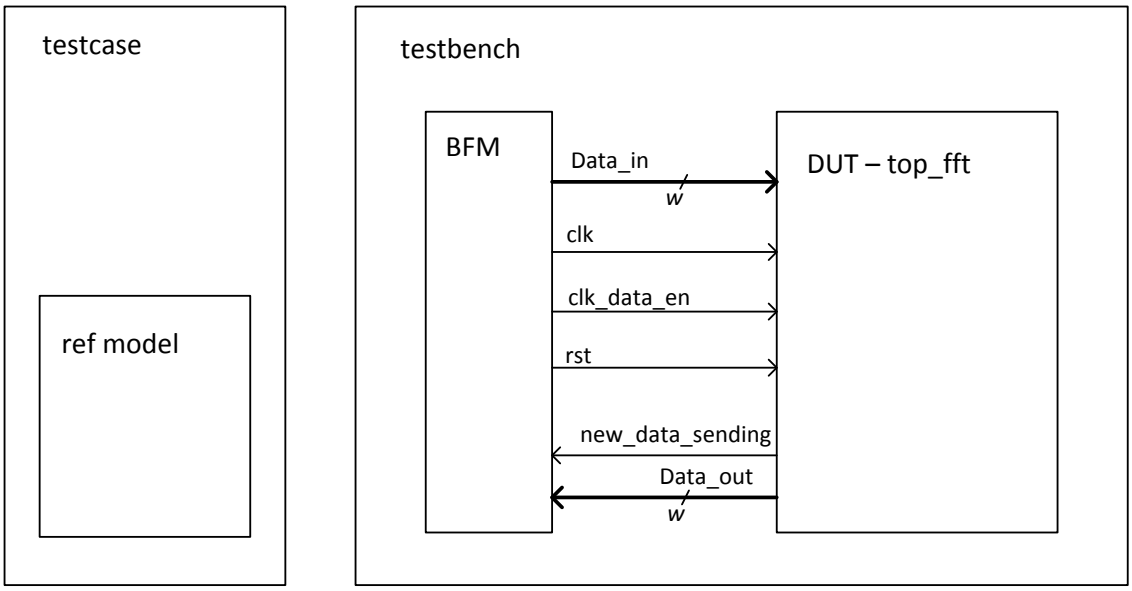


Obrázek 13: Pořadí odesílaných dat

Při zpracovávání odchozích dat je třeba upozornit ještě na jeden fakt. V kapitole 2.4.2 byl popsán čítač čítající příchozí data, který generoval příkaz k odeslání vypočtených hodnot při příchodu předposledního vzorku dat. Z toho plyne, že na výstupu z FFT modulu jsou nová vypočtená data při příchodu posledního vzorku vstupních dat a zpoždění dat ze vstupu na výstup tedy odpovídá velikosti FFT snížené o jedničku. Ke ztrátě vypočtených dat samozřejmě nedochází, jsou vždy odeslána všechna potřebná data.

3 VERIFIKACE NAVRŽENÉ ARCHITEKTURY

Proces verifikace slouží k ověření správnosti návrhu, tedy zda navržený obvod splňuje zadání. Verifikace je nedílnou součástí každého návrhu, její význam vzrůstá s komplexností navrhovaných obvodů. Pro správnou verifikaci je dnes popsáno několik různých metodologií, z kterých vychází i verifikační prostředí pro verifikaci zde navržené architektury. Schéma tohoto verifikačního prostředí je na Obrázku 14.



Obrázek 14: Schéma verifikačního prostředí pro testovaní FFT modulu

Verifikační prostředí obsahuje dvě hlavní část, modul *testbench* a program *testcase*, a je kódováno v jazyce SystemVerilog. Jazyk SystemVerilog vychází ze syntaxe jazyku Verilog HDL a přináší několik nových rozšíření vycházejících z vyšších programovacích jazyků (především použití objektově orientované programování), která naleznou své uplatnění při verifikaci návrhu.

3.1 Modul testbench

Modul *testbench* vytváří prostředí, ve kterém je modul FFT simulován. Obsahuje dvě části – *DUT* (Design Under Test), což je testovaný obvod, v tomto případě se jedná o instanci hlavního modulu FFT, a *BFM* (Bus Function Model), který ovládá vstupy a čte výstupy testovaného obvodu. Zdrojový kód modulu *testbench* je uložen v souboru *tb.sv* a je možné ho nalézt i v příloze C.1.

Modul *BFM* obsahuje generátor hodinového signálu *clk*, ovladač globálního resetu *rst*, generátor povolovacího signálu pro příjem dat *clk_data_en* a ovladač datových portů. Zdrojový kód tohoto modulu je umístěn do souboru *BFM.sv* a je možné ho nalézt také v příloze C.2.

Generátor hodinového signálu je realizován jako *always* blok, generování signálu *clk* je povoleno signálem *clk_en*. Ovládání tohoto signálu z vnějšku BFM modulu je možné jen pomocí procedury *run_clk*. Druhou procedurou pro ovládání generátoru hodinového signálu je *set_clk_per*, jehož pomocí je možné nastavit periodu hodinového signálu *clk* a také počet period hodinového signálu *clk*, po kterých jsou nastovavána nová data na vstup FFT modulu.

Ovladač globálního resetu je realizován jako procedura *perform_reset*, jejímž zavoláním je vygenerován asynchronní reset, který je uvolněn po jedné periodě hodinového signálu *clk*, případně s další sestupnou hranou hodinového signálu *clk*.

Generátor povolovacího signálu *clk_data_en* je spouštěn z ovladače datových portů. Generuje aktivní hodnotu signálu při sestupné hraně hodinového signálu *clk*. Aktivní hodnota signálu *clk_data_en* trvá právě jednu periodu hodinového signálu *clk*. Po vygenerování pulsu čítá náběžné hrany hodinového signálu *clk* podle konstanty uložené v souboru *test_pkg.sv* nebo hodnoty nastavené procedurou *set_clk_per*. Následně opět vygeneruje puls při sestupné hraně hodinového signálu *clk*, generování signálu *clk_data_en* je ukončeno opět ovladačem datových portů.

Ovladač datových portů v modulu *BFM* je proveden jako procedura *send_and_receive_data*. Vstupním parametrem pro tuto proceduru je pole hodnot o velikosti *N*, které mají být odeslány do testovaného modulu. Tato procedura následně vrací pole hodnot přijatých z testovaného FFT modulu. Odesílání a přijímání dat probíhá ve třech fázích. V první fázi je nejprve aktivován generátor povolovacího signálu *clk_data_en* pro příjem dat a na výstup *Data_in* modulu *BFM* jsou přivedena první data. Další data jsou na vstup nastavena vždy s náběžnou hranou signálu *clk_data_en*. Po odeslání všech vstupních dat se čeká na provedení výpočtu v FFT modulu. Během čekání, což je druhá fáze této procedury, je stále generován signál *clk_data_en* a počítá se počet pulzů na tomto signálu. Počet pulzů odpovídá velikosti FFT snížené o jedničku (viz. kapitola 2.6). Následně se vstoupí do poslední fáze této procedury, kdy jsou ukládána data odesílaná z FFT modulu. Vzorkování odchozích dat je prováděno při sestupné hraně signálu *clk_data_en*. Po přijetí všech dat je ukončeno generování povolovacího signálu *clk_data_en* a procedura vrátí přijatá data.

3.2 Program testcase

Druhou částí verifikačního prostředí je program *testcase*, který slouží ke generování testovacích dat a jejich kontrolu. Na první pohled se může zdát, že verifikační prostředí by mělo obsahovat jen jeden hlavní modul, v kterém je umístěn testovaný obvod i generátor testovacích vektorů. Jazyk SystemVerilog však obsahuje novou konstrukci, kterou je blok *program*. Tento blok je přímo určen pro generování testovacích dat a jejich případnou kontrolu. Použití bloku *program* umožňuje urychlení simulace a snížení nároků na paměť. Testovací data jsou pak do testovaného obvodu odesílána pomocí volání procedury *send_and_receive_data* modulu *BFM*. Zdrojový kód programu *testcase* nalezneme v souboru *testcase.sv* a v příloze C.3.

Jak bylo zmíněno výše, program *testcase* slouží ke generování testovacích dat a kontrole dat vypočtených v FFT modulu. Testovací data jsou vytvářena generátorem náhodných čísel a převedena do binární podoby. Tuto činnost zajišťuje funkce *randomize_data*. Náhodné hodnoty (ve skutečnosti to náhodné hodnoty nejsou, funkce

pro generování náhodných hodnot je totiž algoritmus, který je definovaný ve standartech jazyka SystemVerilog) jsou generovány ve formě celých čísel a následně jsou přetypovány do reálné a binární hodnoty. Data v bínární hodnotě jsou následně odeslána do testovaného obvodu pomocí výše popsané procedury pro odesílání a přijímání dat v modulu *BFM*. Data ve formě reálných čísel slouží jako vstupní data pro referenční model.

Referenční model je popsán jako samostatná třída a algoritmus je shodný s algoritmem popsaným v prostředí Matlab (viz. kapitola 2.2) a v této kapitole už nebude podrobněji rozebírán. Použití třídy pro popis referenčního modelu má svůj důvod; pod pojmem třída si můžeme představit datový typ, pro který jsou definovány nějaké funkce, které můžou být při vytvoření proměnné tohoto datového typu použity. Důležitý je však fakt, že po vytvoření objektu (jak je správně nazývána proměnná vytvořená z dané třídy), je možné provést relativně složitý výpočet, ale pokud není tento objekt dále používán, je uložena jen výsledná hodnota a zbytek dat (tzn. všechny proměnné, které byly v průběhu výpočtu použity) je zahozen. Toto výrazně šetří místo v paměti a snižuje celkovou výpočetní náročnost simulace. Zdrojový kód referenčního modelu v SystemVerilogu je možné nalézt v příloze C.4 a pro verifikaci je uložen v souboru *ref_fft.sv*. Referenční model je do programu *testcase* připojen pomocí příkazu *`include "ref_fft.sv"*, to v podstatě znamená, že při kompilaci verifikačního prostředí je do souboru *testcase.sv* vložen kód ze souboru *ref_fft.sv*. Pro bezproblémovou verifikaci je pak třeba soubor s referenčním modelem vložit do složky, ze které je spuštěna simulace nebo ve které je uložen soubor s nastavením simulačního projektu.

Odeslání dat do ovladače portů a referenčního modelu probíhá v proceduře *send_receive_data* v programu *testcase*. Nejdříve je v této proceduře vytvořen objekt s referenčním modelem algoritmu FFT a následně jsou v něm provedeny výpočty pomocí funkce *run_fft* (tato funkce spustí výpočet se zadanými vstupními daty a vrátí vypočtené hodnoty). Následně je zavolána procedura *send_and_receive_data* z modulu *BFM*, která vrátí data vypočtená v testovaném obvodu.

Kontrola přijatých dat probíhá v další proceduře programu *testcase*, kterou je procudera *check_data*. V této proceduře je třeba nejprve přeskupit data přijatá z testovaného modulu, protože jsou ve formátu popsaném na Obrázku 13. Data jsou převáděna do formátu, v jakém byla uložena v hlavním datovém registru v FFT modulu, v tomto formátu totiž vrací výsledky referenční model, s nimiž jsou přijatá data po konverzi na reálná čísla porovnávána. Tolerance je nastavena pomocí konstanty *c_TOLERANCE* v souboru *test_pkg.sv* a porovnání je provedeno přičtením resp. odečtením přijatých dat a této konstanty a je ověřováno, že v takto vzniklém intervalu je i hodnota referenčního modelu. Procedura vypíše chybovou hlášku, pokud jsou data přijatá z testovaného obvodu odlišná o více než povolený interval tolerance oproti datům z referenčního modelu.

3.3 Výsledky verifikace

Verifikace byla provedena pro různé velikosti FFT a různé bitové šířky vstupních a výstupních dat. Všechny tyto vybrané kombinace jsou zapsány v Tabulce 4 a byly testovány 50 běhy s náhodnými vstupními daty. Stejné kombinace velikosti FFT a bitové šířky jako v Tabulce 4 byly použity pro verifikaci FFT modulu s registry v aritmetické části i bez nich. FFT modul byl verifikován jako tzv. „Black box“, což znamená, že nejsou kontrolovány vnitřní signály a ověřuje se jen správná odpověď na vstupní data.

Tabulka 4: Vybrané kombinace parametrů FFT modulu pro verifikaci

Velikost FFT	Počet stupňů	Bitová šířka slova *w*								
		4	5	6	7	8	9	10	11	12
4	2	x		x						
8	3		x		x					
16	4			x		x				
32	5						x		x	
64	6							x		x
128	7						x		x	

Při porovnávání dat z testovaného modulu a referenčního modulu byla zvolena tolerance odpovídající hodnotě dvou posledních bitů. Tato hodnota je však menší než možná odchylka vzniklá nepřestností výpočtu v aritmetické části. Pokud by však byla nastavena větší tolerance, je možné, že by byla takto zakryta i skutečná chyba v obvodu. Problém nedostatečné tolerance se objevuje především u větších velikostí algoritmu FFT spolu s použitím malé bitové šířky. V takovém případě mohou být výsledky výrazně zaokrouhleny a potřebná tolerance na výstupu by byl příliš velká. Proto je třeba i při výpisu chyby verifikačním prostředím zkontrolovat, zda se jedná o skutečnou chybu či zda vypočtená hodnota jen mírně překročila nastavenou toleranci. Tento problém nastal například při verifikaci modulu FFT pro velikost $N = 128$ a $w = 9$, kdy při několika testech byly stejnosměrné složky mimo toleranci. Vzhledem k tomu, že referenční model počítá v plovoucí řádové čárce s přesností na 64 bitů, je jeho přesnost výrazně lepší a při výrazně menší datové šířce, která byla pro testování FFT modulu použita, lze očekávat odlišné výsledky.

4 SYNTÉZA NAVRŽENÉ ARCHITEKTURY

Syntéza FFT modulu navrženého v diplomové práci byla provedena jak pro obvody FPGA, tak i pro obvody ASIC. Je zřejmé, že vzhledem k velkému množství kombinací velikosti algoritmu FFT a šířky vstupních dat nebylo možné provést syntézu pro všechny možné kombinace. Pro syntézu návrhu do obvodů FPGA i ASIC byly vybrány velikosti algoritmu FFT 8 až 256 a bitové šířky vstupních slov 8 bitů a 12 bitů. Byly syntetizovány dvě varianty navrženého FFT modulu – s rozdělením datové cesty v aritmetické části pomocí registrů a bez tohoto rozdělení. Rozdíl ve velikosti výsledného obvodu nebude pravděpodobně nijak velký, lze však očekávat znatelný rozdíl mezi maximálními frekvencemi hodinového signálu *clk* v obou případech.

Syntéza pro obvody FPGA byla provedena v návrhovém prostředí Xilinx ISE Project Navigator 14.2. Jako cílový obvod byl vybrán obvod FPGA Virtex 4 XC4VFX12 od firmy Xilinx. Rodina FPGA obvodů Virtex je určena především pro zpracování signálu a implementaci rozsáhlých systémů do jednoho čipu, její varianta FX pak obsahuje i komponenty pro komunikaci pomocí rozhraní Ethernet a procesor PowerPC. Tento obvod byl zvolen kvůli přiměřenému množství buněk, které jsou na čipu obsaženy a jsou dostatečné pro menší velikosti algoritmu FFT, a zároveň přítomnosti komponentu pro komunikaci po Ethernetovém rozhraní, které by mohlo být použito pro odesílání dat do modulu. Výsledky syntézy nalezneme ve dvou tabulkách, v Tabulce 5 a Tabulce 6.

Tabulka 5: Výsledky syntézy FFT modulu do obvodu Virtex 4 - registry v aritmetické části

Velikost algoritmu FFT	Bitová šířka dat	Počet buněk	Počet klopných obvodů	Počet 4-vstupých LUT	$f_{max}(clk)$ [MHz]
8	8	365	313	552	106,3
	12	511	460	769	102,3
16	8	667	574	980	99,2
	12	977	853	1413	97,1
32	8	1251	1105	1822	95,0
	12	1838	1637	2664	92,5
64	8	2389	2130	3502	81,9
	12	3517	3168	5116	79,1
128	8	4670	4180	6759	76,6
	12	6927	6246	9940	71,5
256	8	9196	8286	13281	66,1
	12	13713	12402	19626	63,7

Tabulka 6: Výsledky syntézy FFT modulu do obvodu Virtex 4 - bez registrů v aritmetické části

Velikost algoritmu FFT	Bitová šířka dat	Počet buněk	Počet klopných obvodů	Počet 4-vstupých LUT	$f_{max}(clk)$ [MHz]
8	8	319	275	550	87,9
	12	450	404	755	84,0
16	8	579	535	979	78,5
	12	839	789	1413	75,8
32	8	1083	1068	1869	77,9
	12	1589	1583	2714	75,4
64	8	2063	2085	3579	67,9
	12	3042	3107	5223	65,8
128	8	4025	4130	6765	61,5
	12	6063	6289	9949	60,1
256	8	8059	5245	13306	55,2
	12	11985	12349	19630	53,5

V obou předchozích tabulkách je možné si všimnout, že počet použitých buněk v FPGA obvodu Virtex 4 narůstá s velikostí algoritmu při stejné šířce dat vždy na asi dvojnásobek předchozí hodnoty. To je předpokládané chování, algoritmus FFT dvojnásobné velikosti potřebuje také dvojnásobnou velikost paměti. Rozdíl mezi Tabulkou 5 a Tabulkou 6 je především v maximální použitelné frekvenci hodinového signálu. Pro nejmenší velikosti FFT je tento rozdíl až 20 MHz, při velikosti algoritmu FFT 256 už tento rozdíl klesá na asi 10 MHz. Je také patrný nárůst počtu potřebných klopných obvodů, nicméně tato hodnota je u větších velikostí algoritmu FFT jen velmi malá část z celkové plochy, naproti tomu u malých velikostí algoritmu FFT už se může jednat o významnou část plochy a je třeba zvážit, zda je žádoucí vyšší frekvence hodinového signálu nebo menší plocha.

Syntéza pro obvody ASIC probíhala v programu Encounter RTL Compiler od společnosti Cadance. Byla zvolena technologická knihovna amis350uaascc. Výsledky syntézy pro obvody ASIC jsou významné především kvůli využité ploše na čipu, frekvence hodinového signálu *clk* nebyla v tomto případě zásadní kritérium při syntéze a byla ponechána velká rezerva v periodě hodinového signálu. Výsledky syntézy pro obvody ASIC najdeme ve dvou tabulkách, v Tabulce 7 a Tabulce 8.

Rozdíl mezi oběma následujícími tabulkami obsahující výsledky syntézy pro obvody ASIC je opět dán volbou použití či nepoužití registrů v aritmetické části kvůli zvýšení maximální frekvence signálu. Rozdíl v ploše je znát především u sekvenční části logiky, protože kombinační část logiky se mění jen minimálně. Z tabulek je patrné, že i při syntéze do obvodů ASIC dvojnásobná velikost algoritmu FFT znamená asi dvojnásobnou plochu. Stejně tak lze postupovat u bitové šířky slova, pokud použijeme pro stejnou velikost algoritmu FFT dvojnásobnou bitovou šířku, plocha obvodu bude dvojnásobná. Pokud budeme vycházet z této úvahy, lze odhadnout i plochu pro další velikosti algoritmu FFT s různými bitovými šířkami dat.

Tabulka 7: Výsledky syntézy FFT modulu do obvodu ASIC - registry v aritmetické části

Velikost algoritmu FFT	Bitová šířka dat	Plocha sekvenční logiky [μm^2]	Plocha kombinační logiky [μm^2]	Celková plocha [μm^2]	Celková plocha [mm^2]
8	8	80079	84006	164085	0,16
	12	120735	133008	253743	0,25
16	8	147510	140149	287659	0,29
	12	221758	231828	453586	0,45
32	8	287219	223663	510882	0,51
	12	425113	356425	781538	0,78
64	8	553819	389175	942994	0,94
	12	829206	594928	1424134	1,42
128	8	1098969	706501	1805470	1,81
	12	1647332	1054944	2702276	2,70
256	8	2148016	1358389	3506405	3,51
	12	3260450	1978460	5238910	5,24

Tabulka 8: Výsledky syntézy FFT modulu do obvodu ASIC - bez registrů v aritmetické části

Velikost algoritmu FFT	Bitová šířka dat	Plocha sekvenční logiky [μm^2]	Plocha kombinační logiky [μm^2]	Celková plocha [μm^2]	Celková plocha [mm^2]
8	8	70139	83527	153666	0,15
	12	103602	132684	236286	0,24
16	8	138049	135600	273649	0,27
	12	201605	230817	432422	0,43
32	8	270423	222354	492777	0,49
	12	400489	327579	728068	0,73
64	8	535235	384160	919395	0,92
	12	805062	583161	1388223	1,39
128	8	1069316	693372	1762688	1,76
	12	1630782	1056913	2687695	2,69

Obecně lze tvrdit, že menší velikosti algoritmu FFT je vhodnější použít bez rozdělení výpočtu v aritmetické části na dva hodinové takty při zachování relativně velké maximální frekvence hodinového signálu. Zároveň jsou menší velikosti algoritmu FFT kvůli relativně malé ploše a velké frekvenci hodinového signálu vhodné pro rychlé zpracování dat s malým zpožděním. Jejich nevýhodou je samozřejmě menší rozlišení spektrálních složek.

Naopak pokud chceme dosáhnout dobrého rozlišení spektrálních složek, je třeba použít větší velikost algoritmu FFT, což přináší nevýhodu v podobě větší zabrané plochy na čipu a možnosti použít menší frekvenci hodinového signálu. Jak je z předchozích tabulek zřejmé, registry v aritmetické části tvoří u větších velikostí FFT

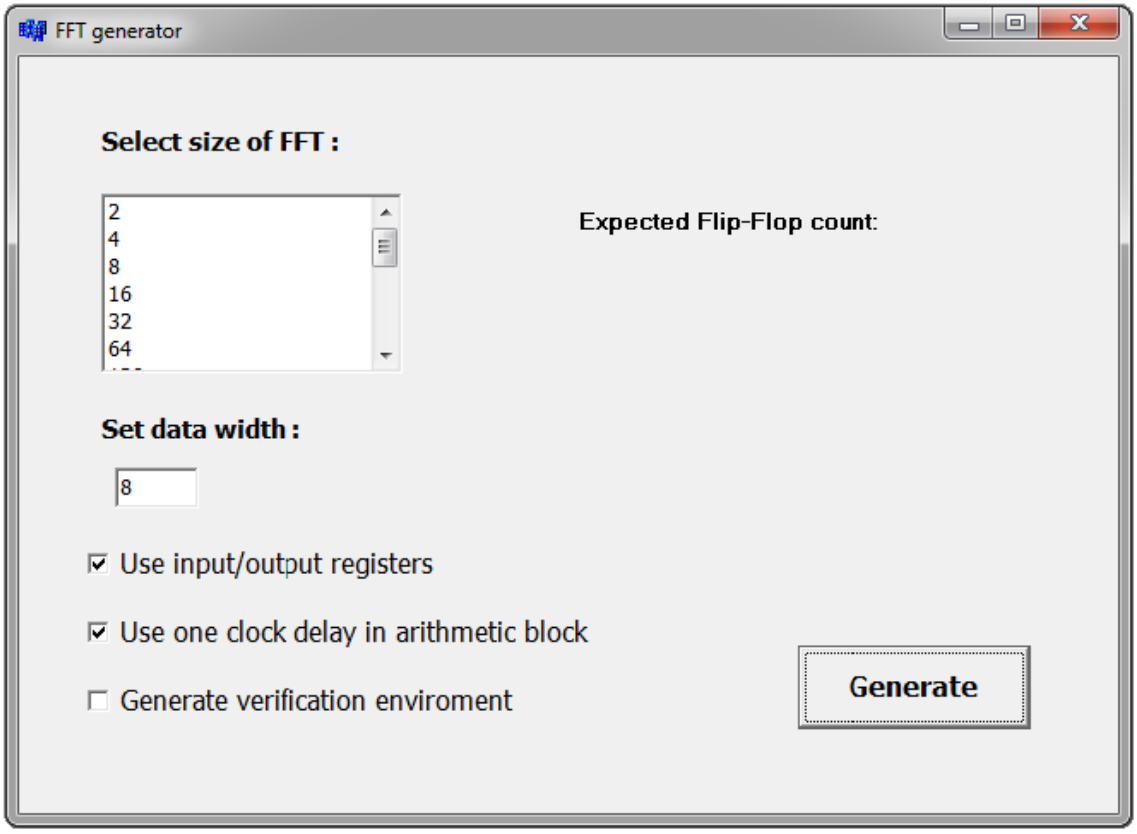
jen malý díl z celkové využité plochy, ale mohou zvýšit maximální frekvenci hodinového signálu. U větších velikostí algoritmu FFT je tedy vhodné tyto registry využít.

Syntéza obvodu byla provedena ze dvou důvodů. Prvním z nich bylo ověření syntetizovatelnosti zápisu ve VHDL kódu, především pak správnou syntézu různých funkcí popsaných v knihovním balíku *fft_module_pkg.vhd*. Druhým důvodem bylo zjištění výsledné plochy pro různé parametry obvodu, aby bylo možné alespoň odhadnout velikost plochy ještě před začátkem procesu syntézy obvodu, dle očekávané plochy zvolit parametry FFT modulu a tomuto případně uzpůsobit další bloky na čipu.

5 GENERÁTOR ZDROJOVÝCH KÓDŮ FFT MODULU

V poslední části diplomové práce bude popsán generátor zdrojových kódů FFT modulu. Důvodem vytvoření tohoto generátoru je snaha o usnadnění práce při použití navržené architektury. Není tedy třeba pracně přepisovat části kódu při změně velikosti FFT nebo při změně bitové šířky vstupních dat a je možné snadno simulovat chování FFT modulu s různými parametry a vybrat tu nejlepší variantu pro navrhovaný obvod.

Generátor byl vytvořen s grafickým prostředím v programovacím jazyce C++ v návrhovém studiu Borland C++ Builder 6. Grafické prostředí generátoru je na Obrázku 15.



Obrázek 15: Grafické prostředí FFT generátoru

Před vygenerováním samotných zdrojových kódů je zde možnost nastavit parametry výsledného modulu FFT. Jako první je třeba nastavit velikost algoritmu FFT. Výběr dostupných velikostí FFT je v okně se seznamem označeným názvem „*Select size of FFT*“. Maximální velikost algoritmu FFT je v programu omezena do velikosti 65 536 (2^{16}). Pokud nebude zvolena žádná velikost algoritmu FFT, v programu je automaticky nastavena hodnota 64 (2^{6}).

Druhým parametrem, který je třeba v FFT modulu nastavit, je šířka vstupních a výstupních dat. Šířka dat je nastavovena pomocí pole pro vstup textu označeném názvem „*Set data width*“. I v případě této proměnné je nastavena jeji počáteční hodnota, tentokrát do hodnoty 8, a její maximální hodnota, která omezuje bitovou šířku slova na 32 bitů.

Další volitelné parametry jsou nastavovány pomocí prvků pro výběr. První z nich je nazván „*Use input/output registers*“ slouží k výběru, zda chceme vygenerovat celý FFT modul včetně jeho hlavního modulu (entity *top_fft* a *fft_module*) nebo pouze vnitřní část FFT modulu obstarávající výpočet algoritmu (pouze entita *fft_module*). S touto informací pracuje generátor, který při výběru nepoužít vstupní a výstupní registry zkopíruje jen část souborů.

Další výběrový prvek je určen pro volbu, zda chceme výpočet v aritmetické části rozdělit pomocí registrů na dva hodinové takty a zvýšit tak maximální použitelnou frekvenci.

Poslední možnou volbou je současně s vygenerováním kompilovatelných HDL souborů vygenerovat i verifikační prostředí pro FFT modul s danými parametry.

Po nastavení zvolených parametrů je automaticky na pravé straně zobrazován odhad použitého počtu klopných obvodů vzhledem ke zvoleným parametrům. Hodnoty jsou vypsány dvě. První hodnota obsahuje informaci o počtu klopných obvodů použitých jako pamět pro data a výpočet je proveden podle rovnice (15). Druhá hodnota udává očekávaný počet klopných obvodů použitých v řídící logice a pro synchronizaci signálu. Tuto hodnotu je třeba brát s rezervou, jedná se jen o odhad a skutečná hodnota bude (byť třeba jen napatrně) jiná. HDL soubory jsou vygenerovány stisknutím tlačítka „*Generate*“.

Architektura popsaná v této práci je rozdělena do několika souborů pro lepší přehlednost a především kvůli možnosti použít komponenty ve zdrojových kódech, protože při psaní zdrojových kódů v jazyce VHDL musí být každá entita umístěna v samostatném souboru. Vzorové soubory, s kterými generátor pracuje, musí být uloženy ve složce *temp_rtl*, která pak musí být ve stejné složce jako generátor FFT modulu. Kompilovatelné soubory obsahující kompletní popis FFT modulu pak nalezneme ve složce *rtl*, která musí být taktéž ve stejné složce jako generátor FFT modulu.

5.1 Funkce pro generování kódu FFT modulu

Pro vygenerování souborů VHDL popisem FFT modulu bylo vytvořeno několik funkcí, které jsou uloženy v souboru *generate_hdl_sources.cpp* a je možné je nalézt v příloze D.1. Tyto funkce převážně pracují se vzorovými soubory, které upravují podle zadaných parametrů. Výhodou tohoto přístupu je možnost jednoduché úpravy vzorových souborů při jejich odlaďování a případné úpravě funkčnosti FFT modulu. Je však doporučeno do těchto vzorových souborů nijak nezasahovat, neboť i malá změna (například přidání prázdného řádku) může způsobit nefunkčnost vygenerovaných zdrojových kódů. Vzorové soubory jsou umístěny ve složce *temp_rtl* a vygenerované zdrojové kódy ve složce *rtl*. Obě tyto složky se musí být umístěny ve stejné složce jako generátor zdrojových kódů.

První volaná funkce při spuštění generování HDL kódu je generátor hodnot konstant (hodnoty otáčecího činitele pro počítání v pevné řádové čárce), které jsou zapsány do souboru *const_bank.vhd* ve složce *rtl*. Pro tento soubor neexistuje žádný vzorový soubor a je tedy vytvořen přímo programem. Vstupními parametry pro generování obsahu paměti ROM je velikost algoritmu FFT (z této hodnoty je určeno, kolik konstant v paměti ROM bude) a bitová šířka slova (konstanty mají stejnou bitovou šířku jako vstupní a výstupní data).

Funkce pro generování obsahu paměti ROM nejdříve vytvoří nový soubor a zapiše do něj hlavičku a příkazy pro použití knihoven. Pokud by nebylo možné vytvořit nový soubor (například kvůli tomu, že by neexistovala cílová složka *rtl*), bude tato chyba při generování zapsána do souboru *report* ve složce generátoru. Po úspěšném vytvoření souboru je naopak do tohoto souboru zaspána informace o úspěšném generování souboru *const_bank.vhd* a s použitými parametry.

Po vytvoření souboru *const_bank.vhd* jsou následně v cyklu *for* postupně generovány reálné části hodnoty otáčecího činitele podle rovnice (8) funkcí *cos*. Hodnoty jsou vypočteny v reálných číslech, vynásobeny váhovým koeficientem $N/2$ a konvertovány do celočíselného typu. Z celočíselného typu jsou hodnoty konvertovány pomocí funkce *fce_int2bin* vytvořené ve stejném souboru do binární podoby zapsané jako řetězec znaků ve formátu *char* a tento řetězec je zapsán do cílového souboru pomocí funkce *fprintf*. Generování imaginární části otáčecího činitele je proveden obdobným způsobem, jen pro výpočet jeho hodnot je použita funkce *sin*. Po zápisu všech dat do souboru *const_bank.vhd* je soubor uzavřen.

Druhou volanou funkcí je generátor definičního souboru FFT modulu *fce_gen_def_pkg_file*. Vygenerovaný soubor s knihovním balíkem je uložen ve složce *rtl* pod názvem *fft_module_pkg.vhd*. Při generování definičního souboru tato funkce vychází ze vzorového souboru *fft_module_pkg_temp.vhd* ve složce *temp_rtl*. V tomto souboru jsou uloženy především definice různých datových typů, funkcí a procedur použitých v popisu FFT modulu. Kvůli univerzálnosti návrhu a snadné změně parametrů jsou ve všech HDL souborech použity konstanty pro šířky signálů případně velikosti datových registrů a právě tyto konstanty jsou zapsány do definičního souboru podle hodnot nastavených v generátoru.

Funkce *fce_gen_def_pkg_file* nejprve vytvoří nový soubor *fft_module_pkg.vhd* a následně otevře vzorový definiční soubor *fft_module_pkg_temp.vhd*. Ze vzorového souboru jsou postupně kopírovány všechny řádky, přičemž se kontroluje, zda na řádku není předdefinovaný řetězec znaků, místo kterého jsou zapsány nové řetězce. Tyto identifikátory jsou dva, místo prvního jsou zapsány konstanty i s jejich komentáři a místo druhého, který je ve funkci *get_const_addr*, je vytvořen příkaz *case* kvůli bezproblémové syntéze. K zápisu do souboru se opět používá funkce *fprintf*.

Podobně jako funkce pro generování obsahu paměti ROM zapisuje funkce generátoru definičního souboru informaci o úspěchu provedené operace do souboru *report*. Pokud tedy není možné otevřít jeden ze souborů, bude vypsána chyba.

Poslední funkce pro generování VHDL kódu kopíruje vzorové soubory do složky *rtl*. Funkce je nazvána *fce_copy_files*. Nejdříve otevře vzorový soubor, následně cílový soubor a obsah vzorového souboru je beze změny nakopírován do cílového souboru. Funkce kopíruje vždy jeden soubor, pro kopírování všech souborů je tato funkce volána několikrát, přičemž se mění jen názvy vzorových a cílových souborů.

Názvy těchto souborů jsou definovány v makru na začátku souboru *generate_hdl_sources.cpp*. I tato funkce vypíše do souboru *report* chybu v případě neúspěšné operace.

V grafickém prostředí generátoru FFT je možnost zvolit použití či nepoužití vstupních a výstupních registrů. Pomocí této volby lze vybrat, které soubory budou kopírovány. Pokud tedy není třeba použít vstupní a výstupní registry, funkce pro kopírování souborů je zavolána jen pro soubory *driving_logic.vhd* a *fft_module.vhd*. Výběr je proveden ve funkci *fce_gen_hdl_sources*, pomocí které jsou spuštěny všechny předchozí funkce popsané v této kapitole. Návratovou hodnotou této funkce je informace o úspěšnosti provedených operací, pokud tedy při generování nějakého souborunastane chyba, bude tato informace vypsána v okně grafického prostředí FFT generátoru.

5.2 Funkce pro generování verifikačního prostředí

Funkce pro generování verifikačního prostředí pracují podobným způsobem jako funkce pro generování VHDL kódu FFT modulu popsané v kapitole 5.1. Tyto funkce jsou zapsány v souboru *generate_ver_env.cpp* a můžeme je nalézt i v příloze D.2. Generovaní verifikačního prostředí je volitelné a slouží především k usnadnění verifikace FFT modulu v rámci této diplomové práce. Vzorové soubory verifikačního prostředí jsou uloženy ve složce *temp_models*. Soubory jsou generovány do složky *models* a obě tyto složky musí být umístěny ve složce s generátorem FFT modulu.

První použitá funkce je pro generování definičního balíčku pro verifikaci *test_pkg.sv*. Funkce nejprve vytvoří nový soubor ve složce *models*, následně otevře vzorový soubor *test_pkg_temp.sv* a kopíruje jeho obsah do cílového souboru. Ve vzorovém souboru je umístěn identifikátor, místo něhož jsou do cílového souboru zapsány zvolené parametry FFT modulu, bitovou šířku vstupních a výstupních dat a velikost algoritmu FFT.

Druhá volaná funkce slouží k vytvoření hodnot otáčecího činitele. Tato funkce byla vytvořena především kvůli problémům při použití funkcí *sin* a *cos* v simulátoru Modelsim PE Student Edition 10.2a. Generování hodnot je prováděno stejným způsobem jako u obdobné funkce pro generování obsahu paměti ROM v kapitole 5.1. Vypočtené hodnoty jsou však uloženy jako reálná čísla.

Poslední funkce pro generování verifikačního prostředí kopíruje obsah vzorových souborů ze složky *temp_models* do cílových souborů ve složce *models*. Názvy kopírovaných souborů jsou definovány jako makro na začátku souboru *generate_ver_env.cpp*. Tato funkce je volána vícekrát, přičemž při každém zavolání kopíruje jiný soubor.

Pokud by při otvírání jednoho ze souborů došlo k chybě, do souboru *report* bude tato informace zapsána a v okně grafického prostředí FFT generátoru se objeví informace o chybě. V případě, že během otvírání souborů nenastane žádná chyba, je do souboru *report* zapsána informace o jejich úspěšném generování. Tato funkce je implementována do všech funkcí určených pro generování verifikačního prostředí.

6 ZÁVĚR

V rámci této diplomové práce byl navržen digitální obvod provádějící výpočet rychlé Fourierovi transformace. Algoritmus použitý pro zpracování signálu byl nejprve modelován v prostředí Matlab, především proto, aby byla ověřena správnost výpočtu adres dat v paměti. Výsledný obvod byl popsán v jazyce VHDL. Ve zdrojových kódech bylo dbáno na snadnou změnu parametrů navrženého obvodu pomocí několika konstant definovaných v knihovním balíku tohoto návrhu.

Pro simulaci navrženého obvodu bylo vytvořeno verifikační prostředí v jazyce SystemVerilog obsahující referenční model, které umožňuje snadné testování modulu rychlé Fourierovi transformace pro různé parametry obvodu s pomocí automaticky generovaných testovacích signálů. Pro vybrané velikosti algoritmu FFT a bitové šířky vstupních signálů byla provedena verifikace. Vstupní data pro verifikaci byla vytvořena generátorem náhodných čísel a přijatá data byla porovnávána s očekávanou odezvou vypočtenou referenčním modelem.

Obvod byl pro vybrané velikosti algoritmu FFT a bitové šířky vstupních signálů syntetizován v programu Encounter RTL Compiler do technologické knihovny amis350uaascc. Syntézou byla zjištěna plocha FFT modulu, kterou lze pro různé velikosti algoritmu FFT v technologii 0,35 μm očekávat. Zároveň byla provedena syntéza pro FPGA obvod Virtex-4 od firmy Xilinx. Výsledek syntézy pro obvod FPGA dává především informaci o počtu klopných obvodů, které jsou pro různé velikosti FFT modulu potřeba.

V poslední části práce byl navržen generátor HDL kódu. Pro generování kódu bylo vytvořeno grafické prostředí, ve kterém lze snadno měnit důležité parametry obvodu a usnadňuje práci při implementaci FFT modulu do obvodu provádějící zpracování signálu. FFT generátor zároveň umožňuje generování verifikačního prostředí pro usnadnění verifikace různých velikostí FFT modulu.

LITERATURA

[1] LYONS, Richard G. *Understanding Digital Signal Processing*. 2nd ed. New Jersey : Bernard Goodwin, 2004. 665 s. ISBN 0-13-108989-7.

[2] MEYER-BAESE, Uwe. *Digital Signal Processing with Field Programmable Gate Arrays*. 3rd. Berlin: Springer, 2007. ISBN 978-3-540-72612-8.

[3] CHU, Eleanor a Alan GEORGE. *Inside the FFT black box*: *serial and parallel fast Fourier transform algorithms*. Boca Raton: CRC Press, 2000, 312 s. ISBN 0-8493-0270-6.

[4] Cooley-Tukey FFT algorithm. In *Wikipedia : the free encyclopedia*[online]. St. Petersburg (Florida) : Wikipedia Foundation, 30. 10.2003, last modified on 4. 12. 2011 [cit. 2011-12-15]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm>.

[5] Butterfly diagram. In *Wikipedia : the free encyclopedia* [online]. St. Petersburg (Florida) : Wikipedia Foundation, 19. 9. 2005, last modified on 17. 8. 2011 [cit. 2011-12-15]. Dostupné z WWW: <http://en.wikipedia.org/wiki/Butterfly_diagram>.

[6] *Always Learn* [online]. 2005 [cit. 2011-12-15]. DFT and FFT tutorial. Dostupné z WWW: http://www.alwayslearn.com/DFT%20and%20FFT%20Tutorial/DFTandFFT_BasicIdea.html

[7] PINKER, Jiří a Martin POUPA. *Číslicové systémy a jazyk VHDL*. 1. vydání. Praha: BEN, 2009. ISBN 80-7300-198-5.

[8] SystemVerilog Tutorial. *ASIC world* [online]. 2013 [cit. 2013-05-29]. Dostupné z: http://www.asic-world.com/systemverilog/tutorial.html

SEZNAM PŘÍLOH

A REFERENČNÍ ALGORITMUS FFT V MATLABU

```
% FFT algorithm

%% generate random input data
 N = 64; % set FFT size
 stage_cnt = log(N)/log(2);
 xin = random ('Normal',0,1,1,N);

 address(N) =zeros();
 data (N)= zeros();
 const(N/2)= zeros();
 addr_cnst(N/2)= zeros();

%% reverse address bits - not automated !!!
 %addr2 = [addr(10) addr(9) addr(8) addr(7) addr(6) addr(5) addr(4) addr(3) addr(2) addr(1)];
 for i = 0 : N-1;
     addr =  dec2bin(i,stage_cnt);
     addr2 = [addr(6) addr(5) addr(4) addr(3) addr(2) addr(1)];
     address(i+1) = bin2dec(addr2);
 end
 % order data in reverse bit order address
 for i = 1 : N
     pos = address(i)+1;
     data (i) = xin (pos);
 end

%% generate coeficients
 for c = 1 : N/2
     const(c)= exp(pi/(N/2)*-1i*(c-1));
 end

%% FFT algorithm

 for stage = 0 : stage_cnt-1     % stage counter
    i = 1;                       % address in data register
   while (i <= N)
    a = 0;                       % block counter

       while (a < 2^stage)
       %constant selector
        cnst = const(2^(stage_cnt-stage-1)*a+1);
       % complex multiplier
        mull = cnst * data(i + 2^stage);
       % adder and subtractor
        add_out = data (i) + mull;
        sub_out = data (i) - mull;
       % save data into register
        data(i) = add_out;
        data(i + 2^stage) = sub_out;

       % DEBUG - save all values to compare with ref model in SystemVerilog
        data_reg(i,stage+1)=add_out;
        data_reg(i + 2^stage,stage+1) = sub_out;

       i=i+1;
       a=a+1;
       end

      i=i+2^stage;
    end
 end

%% check data
 xout(N)=zeros();
 xout = data;
 xout_ref = fft(xin);
 check_data = (xout-xout_ref)

% EOF
```

B ZDROJOVÉ KÓDY V JAZYCE VHDL

B.1 Hlavní modul – *top_fft.vhd*

```
--------------------------------------------------------------------------------
---------- Top FFT module with serial input/output data format --------------
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.fft_module_pkg.all;

entity top_fft is
 Port (
    -- input data serial - time domain
       DATA_in        : in  std_logic_vector (c_DATA_WIDTH-1 downto 0);
    -- clock and reset signals
       clk            : in  std_logic;
       clk_data_en    : in  std_logic;
       rst            : in  std_logic;
    -- output data serial - frequency domain
       DATA_out       : out std_logic_vector (c_DATA_WIDTH-1 downto 0);
       new_data_ready : out std_logic
      );

end top_fft;

architecture Structural of top_fft is

--------------------------------------------------------------------------------
-------------------------- Signal declaration ----------------------------------
--------------------------------------------------------------------------------

  signal load_done          : std_logic;
  signal load_out_reg       : std_logic;

 --signals - input register
  signal input_reg_in       : t_LOAD_REG;
  signal input_reg_out      : t_LOAD_REG;

  signal input_reg_out_temp : t_DATA_IN_OUT_FFT;

 --signals - output register
  signal output_reg_in      : t_LOAD_REG;
  signal output_reg_out     : t_LOAD_REG;

  signal output_reg_in_temp : t_DATA_IN_OUT_FFT;

 --signals - output register multiplexors - load data or shift out data
  signal output_reg_mul1    : t_LOAD_REG;
  signal output_reg_mul2    : t_LOAD_REG;

--------------------------------------------------------------------------------
------------------------- Component declaration -------------------------------
--------------------------------------------------------------------------------

component loading_logic
  port(
      -- clock and reset signals
       clk             : in  std_logic;
       clk_data_en     : in  std_logic;
       rst             : in  std_logic;
      -- load done
       load_done       : out std_logic;
       load_out_reg    : out std_logic
       );
end component;
```

```
component fft_module
  port(
      -- input data serial - time domain
       DATA_in     : in  t_DATA_IN_OUT_FFT;
      -- clock and reset signals
       clk         : in  std_logic;
       rst         : in  std_logic;
      -- load enable
       load_en     : in  std_logic;
      -- output data serial - frequency domain
       DATA_out    : out t_DATA_IN_OUT_FFT
       );
end component;

begin

--------------------------------------------------------------------------------
------------------------- Component connection -------------------------------
--------------------------------------------------------------------------------


 input_reg: process (clk, clk_data_en, rst)
  begin
   if (rst = '1') then
     input_reg_out <= (others => (others =>'0'));
   elsif (rising_edge(clk) and clk_data_en = '1') then
     input_reg_out <= input_reg_in;
   end if;
  end process;

 output_reg: process (clk, clk_data_en, rst)
  begin
   if (rst = '1') then
     output_reg_out <= (others => (others =>'0'));
   elsif (rising_edge(clk) and clk_data_en = '1') then
     output_reg_out <= output_reg_in;
   end if;
  end process;


 loading_logic_1: loading_logic
  port map(
       -- clock and reset signals
        clk             => clk,
        clk_data_en     => clk_data_en,
        rst             => rst,
       -- load done pulse after load all data
        load_done       => load_done,
       -- load data to output register enable
        load_out_reg    => load_out_reg
          );

 fft_module_1: fft_module
  port map(
       -- input data serial - time domain
        DATA_in     => input_reg_out_temp,
       -- clock and reset signals
        clk         => clk,
        rst         => rst,
       -- load enable
        load_en     => load_done,
       -- output data serial - frequency domain
        DATA_out    => output_reg_in_temp
         );

--------------------------------------------------------------------------------
--------- Generate input register and output register connections -------------
--------------------------------------------------------------------------------

-----  INPUT REGISTER ------

 -- serial input register - set as FIFO
  input_reg_load:
   for addr in 1 to c_FFT_SIZE - 1 generate
     input_reg_in (addr-1)  <= input_reg_out (addr);
   end generate;

 -- connect input pin with first input register
 input_reg_in (c_FFT_SIZE - 1) <= DATA_in;
```

```
 -- generate connection with FFT_module component
  input_reg_send:
   for addr in 0 to c_FFT_SIZE- 1 generate
     -- real part
     input_reg_out_temp(addr * 2) <= input_reg_out(addr);
     -- imaginary part
     input_reg_out_temp(addr * 2 + 1) <= (others => '0');
   end generate;


 -----  OUTPUT REGISTER ------

 -- serial output register - set as FIFO
  output_reg_serial:
   for addr in 0 to c_FFT_SIZE - 2 generate
    output_reg_mul2 (addr) <= output_reg_out (addr+1);
   end generate;

 -- when shifting data out, zeros are loaded into empty registers
  output_reg_mul2(c_FFT_SIZE-1) <= (others=>('0'));


 -- data from data register to output register multiplexor
 -- when only real input number, output is complex but symmetric
 -- data format: Re(0) Re(1) Im(1) Re(2) Im(2) ...
 -- ... Re(c_FFT_SIZE/2 -1) Im(c_FFT_SIZE/2 -1) Re(c_FFT_SIZE/2)
   output_reg_load: for addr in 1 to c_FFT_SIZE - 1
    generate
     output_reg_mul1 (addr)   <= output_reg_in_temp(addr + 1);
    end generate;
     output_reg_mul1 (0)      <= output_reg_in_temp(0);

 -- output register multiplexer
  output_reg_mul:
   for addr in 0 to c_FFT_SIZE - 1 generate
     output_reg_in (addr) <= output_reg_mul1 (addr)
       when load_out_reg = '1' else
                           output_reg_mul2 (addr);
   end generate;

 -- drive output pin from output register
 DATA_out <= output_reg_out(0);

 new_data_ready <= load_done;

end Structural;
```

B.2 Ovladač načítání dat – *loading_logic.vhd*

```
----------------------------------------------------------------------------------
------------------------------ Data load driver ----------------------------------
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.fft_module_pkg.all;

entity loading_logic is
  Port(
      -- clock and reset signals
       clk           : in  std_logic;
       clk_data_en   : in  std_logic;
       rst           : in  std_logic;
      -- load done
       load_done     : out std_logic;
       load_out_reg  : out std_logic
       );
end loading_logic;

architecture Behavioral of loading_logic is

 -- signal declaration
  signal load_cnt_s  : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
  signal load_cnt_c  : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
```

```
 begin

  load_counter_s: process (clk, clk_data_en, rst)
   begin
    if rst = '1' then
     load_cnt_s <= (others => '1');
    elsif (rising_edge(clk) and clk_data_en = '1') then
     load_cnt_s <= load_cnt_c;
    end if;
   end process;

  load_counter_c: process (load_cnt_s)
   begin
     load_done <= '0';
     load_out_reg <= '0';
     load_cnt_c <= load_cnt_s + 1;
    if load_cnt_s = c_FFT_SIZE - 1 then
       load_cnt_c <= (others => '0');
       load_done <= '1';
    end if;
    if load_cnt_s = c_FFT_SIZE - 2 then
      load_out_reg <= '1';
    end if;
   end process;

end Behavioral;
```

B.3 FFT modul – *fft_module.vhd*

```
----------------------------------------------------------------------------------
-------------- FFT module - implement Fast Fourier Transform ----------------
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

use work.fft_module_pkg.all;
use work.const_bank_pkg.all;

entity fft_module is
 Port (
      -- input data - time domain
       DATA_in     : in  t_DATA_IN_OUT_FFT;
      -- clock and reset signals
       clk         : in  std_logic;
       rst         : in  std_logic;
      -- load enable
       load_en     : in  std_logic;
      -- counting of FFT done, do not use with top_fft
      -- do not forget uncomennt also signal assigment on line 235
--       count_done  : out std_logic;
      -- output data - frequency domain
       DATA_out    : out t_DATA_IN_OUT_FFT
       );

end fft_module;

architecture Structural of fft_module is

-----------------------------------------------------------------------------------
--------------------------- Signal declaration --------------------------------
-----------------------------------------------------------------------------------

 -- selected data from main register (into multiplier, adder)
  signal data1           : t_COMPLEX;
  signal data2           : t_COMPLEX;

 -- signal from constant bank (into multiplier)
  signal const           : t_COMPLEX;

 -- signals from complex multiplier
  signal mull_out        : t_COMPLEX;

 -- signals from complex adder
  signal add_out         : t_COMPLEX;

 -- signals from complex subtractor
  signal sub_out         : t_COMPLEX;
```

```
 -- main data register signal array
  signal data_reg_out    : t_COMPLEX_ARRAY;
  signal data_reg_in     : t_COMPLEX_ARRAY;
  signal data_reg_load   : t_COMPLEX_ARRAY;

 -- select data to write into data register (into multiplexer_data_reg_in)
  signal data_reg_in_add : std_logic_vector (c_FFT_SIZE - 1    downto 0);
  signal data_reg_in_sub : std_logic_vector (c_FFT_SIZE - 1    downto 0);

 -- trigger for loading new data from input register
  signal data_reg_load_en: std_logic;

 -- disable writing new data into main data register, active when count ends
  signal data_reg_hold   : std_logic;

 -- select data from data register (drives data1, data2)
  signal data1_addr     : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
  signal data2_addr     : std_logic_vector (c_STAGE_COUNT - 1 downto 0);

-- constant address (into constant bank)
  signal const_addr      : std_logic_vector (c_STAGE_COUNT - 2 downto 0);

 -- one clock delay in arithmetic part between multiplier and adder/subtractor
 -- + data register input multiplexors driving signals one clock delay
  signal arith_delay_c   : t_SYNC_ASYNC_ARITHMETIC;
  signal arith_delay_s   : t_SYNC_ASYNC_ARITHMETIC;


--------------------------------------------------------------------------------
------------------------- Component declaration -------------------------------
--------------------------------------------------------------------------------

  -- driving logic, implements FFT algorithm
  component driving_logic
    Port (
         -- clock, reset and load enable from higher module
          clk            : in  std_logic;
          rst            : in  std_logic;
          load_en        : in  std_logic;
         -- selecting data from main data register
          data1_addr     : out std_logic_vector (c_STAGE_COUNT-1 downto 0);
          data2_addr     : out std_logic_vector (c_STAGE_COUNT-1 downto 0);
         -- load data form input register
          data_reg_load  : out std_logic;
         -- hold ceonted data in main data register until new count cycle starts
          data_reg_hold  : out std_logic;
         -- select constant form constant bank
          const_addr     : out std_logic_vector (c_STAGE_COUNT-2 downto 0)
          );
  end component;

 begin

--------------------------------------------------------------------------------
----------------- Structural part + component connection -----------------------
--------------------------------------------------------------------------------

 -- complex multiplier - procedure defined in fft_module_pkg
  cmpl_multiplier(data2, const, mull_out);

 -- complex adder - procedure defined in fft_module_pkg
  cmpl_adder (arith_delay_s.data1, arith_delay_s.data2, add_out);

 -- complex subtractor - procedure defined in fft_module_pkg
  cmpl_subtractor (arith_delay_s.data1, arith_delay_s.data2, sub_out);

 -- main data register
  main_data_reg :  process (rst, clk)
   begin
    if (rst = '1') then
     -- if reset, set 0 in all registers
     data_reg_out <= (others => (others =>(others => '0')));
     -- if rising edge, data are loaded
    elsif (rising_edge(clk)) then
     data_reg_out <= data_reg_in;
    end if;
   end process;
```

```
-- select data to write into main register, array of multiplexors
 data2data_reg_selector:
  for addr in 0 to c_FFT_SIZE - 1 generate
                      -- load new data - highest priority
  data_reg_in(addr) <= data_reg_load(bit_reverse(addr))
                         when data_reg_load_en = '1'      else
                       -- load data from adder
                        add_out
                         when data_reg_in_add(addr) = '1' else
                       -- load data from subtractor
                        sub_out
                         when data_reg_in_sub(addr) = '1' else
                       -- default data are data from main reg output
                        data_reg_out(addr);
 end generate;

 -- driving logic generates address of data and drives loading/sending
 -- new/counted data.
 fft_algorithm: driving_logic
  port map(
      -- clock and reset signals
       clk             => clk,
       rst             => rst,
       load_en         => load_en,
      -- driving data from register
       data1_addr      => data1_addr,
       data2_addr      => data2_addr,
      -- load data from input register
       data_reg_load   => data_reg_load_en,
       data_reg_hold   => data_reg_hold,
      -- select constant form constant bank
       const_addr      => const_addr
          );


-- One clock delay in arithmetic part

 arith_delay_c.dr_hold   <= data_reg_hold;
 arith_delay_c.addr1     <= data1_addr;
 arith_delay_c.addr2     <= data2_addr;
 arith_delay_c.data1     <= data1;
 arith_delay_c.data2     <= mull_out;

 process (arith_delay_c.dr_hold, arith_delay_c.data1, arith_delay_c.data2,
          arith_delay_c.addr1, arith_delay_c.addr2,  clk, rst)
  begin
  -- if latches in artih part are not enabled, signals are connected
  -- without clock delay.
   if (c_SYNC_ASYNC_ARITHMETIC = 0) then
     arith_delay_s <= arith_delay_c;
   else
    if (rst = '1') then
     arith_delay_s.dr_hold <= '0';
     arith_delay_s.addr1   <= (others => '0');
     arith_delay_s.addr2   <= (others => '0');
     arith_delay_s.data1   <= (others =>(others => '0'));
     arith_delay_s.data2   <= (others =>(others => '0'));
    elsif rising_edge(clk) then
     arith_delay_s <= arith_delay_c;
    end if;
   end if;
  end process;

  -- encoder from c_STAGE_COUNT size to c_FFT_SIZE size (one-hot)
 encoder: process (arith_delay_s, data_reg_load_en, arith_delay_s)
    variable data_reg_in_add_var : std_logic_vector (c_FFT_SIZE - 1 downto 0);
    variable data_reg_in_sub_var : std_logic_vector (c_FFT_SIZE - 1 downto 0);
   begin
  -- if all data are counted, disable loading data to main reg from arith part
    if (arith_delay_s.dr_hold = '1') then
     data_reg_in_add_var := (others => '0');
     data_reg_in_sub_var := (others => '0');
  -- one - hot encoding is used to drive one multiplexor in multiplexor array
    else
     data_reg_in_add_var := (others => '0');
     data_reg_in_add_var(to_integer(unsigned(arith_delay_s.addr1))) := '1';
     data_reg_in_sub_var := (others => '0');
     data_reg_in_sub_var(to_integer(unsigned(arith_delay_s.addr2))) := '1';
    end if;
      data_reg_in_add <= data_reg_in_add_var;
      data_reg_in_sub <= data_reg_in_sub_var;
 end process;
```

```
-- Constant selector
  const.re <= c_CONST_RE(to_integer(unsigned(const_addr)));
  const.im <= c_CONST_IM(to_integer(unsigned(const_addr)));

 -- data 1 and data 2 from data reg selector
  data1 <= get_data_from_reg(data_reg_out, data1_addr);
  data2 <= get_data_from_reg(data_reg_out, data2_addr);

-- incoming data ordering - here Re Im Re Im Re Im ... Re Im
 order_loaded_data : for addr in 0 to c_FFT_SIZE - 1 generate
  data_reg_load(addr).re <= signed(DATA_in (addr*2));
  data_reg_load(addr).im <= signed(DATA_in (addr*2 + 1));
 end generate;

--  data from data reg to output reg - Re Im Re Im ... Re Im Re
   output_reg_load: for addr in 0 to c_FFT_SIZE - 1
    generate
     DATA_out (addr * 2)     <= std_logic_vector(data_reg_out(addr).re);
     DATA_out (addr * 2 + 1) <= std_logic_vector(data_reg_out(addr).im);
    end generate;

-- uncomment if count done output signal is needed
--  count_done  <= data_reg_hold;

end Structural;
```

B.4 Řídící logika – *driving_logic.vhd*

```
--------------------------------------------------------------------------------
--------------- Driving logic - implements algorithm of FFT -----------------
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

use work.fft_module_pkg.all;

entity driving_logic is
  Port(
      -- clock and reset signals
       clk             : in  std_logic;
       rst             : in  std_logic;
       load_en         : in  std_logic;     -- rising edge starts FFT count
      -- driving data from register - address in data register
       data1_addr      : out std_logic_vector (c_STAGE_COUNT - 1 downto 0);
       data2_addr      : out std_logic_vector (c_STAGE_COUNT - 1 downto 0);
      -- load data from input register when loading in Data in shift register is finished
       data_reg_load   : out std_logic;
      -- hold values in Data register until they are loaded into Data out shift register
       data_reg_hold   : out std_logic;
      -- select constant form constant bank
       const_addr      : out std_logic_vector (c_STAGE_COUNT - 2 downto 0)
       );
end driving_logic;

architecture Behavioral of driving_logic is

--------------------------------------------------------------------------------
--------------------------- Signal declaration ---------------------------------
--------------------------------------------------------------------------------

 -- state machine signals
  signal present_state       : t_DRIVING_LOGIC_STATE;
  signal next_state          : t_DRIVING_LOGIC_STATE;
  -- signal count_done          : std_logic;

 --progress counter signals
  signal progress_cnt_s      : std_logic_vector (c_STAGE_COUNT - 2 downto 0);
  signal progress_cnt_c      : std_logic_vector (c_STAGE_COUNT - 2 downto 0);

 -- stage counter signals
  signal stage_s             : std_logic_vector(c_STAGE_COUNTER_SIZE - 1 downto 0);
  signal stage_c             : std_logic_vector(c_STAGE_COUNTER_SIZE - 1 downto 0);
  signal stage_done          : std_logic;
```

```
-- address counter
  signal data1_addr_sig      : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
  signal data2_addr_sig      : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
  signal block_step_count_c  : std_logic_vector (c_STAGE_COUNT - 2 downto 0);
  signal block_step_count_s  : std_logic_vector (c_STAGE_COUNT - 2 downto 0);
  signal block_done          : std_logic;
  signal next_block_addr_c   : std_logic_vector (c_STAGE_COUNT - 2 downto 0);
  signal next_block_addr_s   : std_logic_vector (c_STAGE_COUNT - 2 downto 0);

 -- constant address
  signal const_addr_sig      : std_logic_vector (c_STAGE_COUNT - 2 downto 0);

 begin

--------------------------------------------------------------------------------
------------------------------ State machine -----------------------------------
--------------------------------------------------------------------------------

 -- state machine - sequential part
  state_machine_s: process (clk, next_state, rst)
   begin
    if rst= '1' then
      present_state <= e_DONE;
    elsif rising_edge(clk) then
      present_state <= next_state;
    end if;
   end process;

 -- state machine - combinational part
  state_machine_c: process (present_state, load_en, stage_s, progress_cnt_s)
   begin
     data_reg_load <= '0';
     data_reg_hold <= '0';
    case present_state is
    -- load state - save data on input ports into data register
    -- duration one clock period, next state is counting in progress
     when e_LOAD =>
       next_state <= e_COUNT;
       data_reg_load <= '1';
    -- counting in progress state - perform Fast Fourier Transform
    -- duration depends on FFT size, next state is counting in progress
    -- or counting done
     when e_COUNT =>
      next_state <= e_COUNT;
     -- in last stage
      if (stage_s = c_STAGE_COUNT - 1) then
      -- and last step
       if (progress_cnt_s  = (c_FFT_SIZE/2 - 1)) then
       -- counting is finished, next state is counting done
        next_state <= e_DONE;
       end if;
      end if;
    -- done state - counting is done, check if load_en is 0 ->
    -- -> detector of falling edge on load_en
    -- if no FE -> still old data on input -> wait until FE on load_en,
    -- else next state is wait on data
     when e_DONE =>
     -- hold counted data in data register
      data_reg_hold <= '1';
      if load_en = '1' then
       next_state <= e_DONE;
      else
       next_state <= e_WAIT_ON_DATA;
      end if;
    -- wait on data state - wait until new data are set on input
    -- next state is load state - set after rising edge on load_en
     when e_WAIT_ON_DATA =>
      data_reg_hold <= '1';
      if load_en = '1' then
       next_state <= e_LOAD;
      else
       next_state <= e_WAIT_ON_DATA;
      end if;
     when others =>
      next_state <= e_DONE;
    end case;
   end process;
```

```
--------------------------------------------------------------------------------
------------------------------ FFT algotithm -----------------------------------
--------------------------------------------------------------------------------

--------------------------- progress counter ---------------------------------
-- count actual step of FFT counting in each stage,determine when stage is done

 -- progress counter - sequential part
  progress_counter_s: process(clk,rst,progress_cnt_c)
   begin
    if rst = '1' then
     progress_cnt_s <= (others => '0');
    elsif rising_edge(clk) then
     progress_cnt_s <= progress_cnt_c;
    end if;
   end process;

 -- progress counter - combinational part
  progress_counter_c: process(progress_cnt_s, stage_s, present_state)
   begin
     stage_done <= '0';
   -- counting is enabled by FSM only in count state
    if present_state = e_COUNT then
    -- stage is done after (FFT size)/2 clk periods (2 results counted
    -- in arithmetic in one clk period
     if progress_cnt_s = (c_FFT_SIZE/2 - 1) then
       progress_cnt_c <= (others => '0');
       stage_done <= '1';
     else
       progress_cnt_c <= progress_cnt_s + 1;
     end if;
    else
       progress_cnt_c <= (others => '0');
    end if;
   end process;

------------------------------ stage counter ---------------------------------
-- stage counter determine which stage of FFT is counted
 -- stage counter - sequential part
  stage_counter_s: process (clk, stage_c, rst, present_state)
   begin
    if rst = '1' then
     stage_s <= (others => ('0'));
    elsif rising_edge(clk) then
     stage_s <= stage_c;
    end if;
   end process;

 -- stage counter - combinational part
  stage_counter_c: process (stage_s, stage_done)
   begin
   -- count pulses on stage_done, maximal stage set in fft_module_pkg
    if (stage_done = '1') then
     if (stage_s = c_STAGE_COUNT - 1) then
        stage_c <= (others => ('0'));
     else
        stage_c <= stage_s + 1;
     end if;
    else
     stage_c <= stage_s;
    end if;
   end process;


---------------------------- block step counter --------------------------------
-- count actual step of FFT counting in each block,determine when block is done

 -- block step counter - sequential part
  block_step_counter_s: process (clk, rst, block_step_count_c)
   begin
    if rst = '1' then
      block_step_count_s <= (others => '0');
    elsif rising_edge(clk) then
      block_step_count_s <= block_step_count_c;
    end if;
   end process;
```

```
 -- block step counter - combinational part
  block_step_counter_c: process (block_step_count_s, stage_s, present_state)
   begin
     block_done <= '0';
   -- counting is enabled by FSM only in count state
    if present_state = e_COUNT then
    -- block size depends on actual stage, in the following stage is block
    -- size doubled, counter works in full range, but range of counter is dynamic
     if (block_step_count_s = (2**(to_integer(unsigned(stage_s))))-1) then
       block_done <= '1';
       block_step_count_c <= (others => '0');
     else
       block_step_count_c <= block_step_count_s + 1;
     end if;
    else
      block_step_count_c <= (others => '0');
    end if;
   end process;

----------------------------- next block start --------------------------------
-- generate starting address of next block, add size of block to last value
 -- next block start - sequential part
  next_block_start_s: process (rst, clk, block_done, next_block_addr_c)
   begin
    if rst = '1' then
     next_block_addr_s <= (others => '0');
    elsif rising_edge(clk) and block_done = '1' then
     next_block_addr_s <= next_block_addr_c;
    end if;
   end process;

 -- next block start  - combinational part
  next_block_start_c: process (stage_s, next_block_addr_s, stage_done)
   begin
     if stage_done = '1' then
      next_block_addr_c <= (others => '0');
     else
      next_block_addr_c <= next_block_addr_s +
                           (2**(to_integer(unsigned(stage_s))));
     end if;
   end process;


-------------- data address driver and constant address driver ----------------

 -- generate address of data in main data register
  data_driver: process (next_block_addr_s, stage_s, progress_cnt_s)
    variable data1_addr_var : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
    variable data2_addr_var : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
   begin
     data1_addr_var := ('0' & progress_cnt_s) + next_block_addr_s;
     data2_addr_var := data1_addr_var + (2**(to_integer(unsigned(stage_s))));
  -- data address to out
     data1_addr_sig <= data1_addr_var;
     data2_addr_sig <= data2_addr_var;
   end process;

 -- generate constant address
  constant_drive: process (stage_s,block_step_count_s)
   begin
     const_addr_sig <= get_const_addr (stage_s, block_step_count_s);
   end process;

-- output assignment
  data1_addr  <= data1_addr_sig;
  data2_addr  <= data2_addr_sig;
  const_addr  <= const_addr_sig;

 end Behavioral;
```

B.5 Knihovní balík FFT modulu – *fft_module_pkg.vhd*

```
--------------------------------------------------------------------------------
---------------------- FFT module definition package -----------------------
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

 package fft_module_pkg is
--------------------------------------------------------------------------------
---------------- global constants - define design parameters ----------------
--------------------------------------------------------------------------------
 -- stage count - derived from size of FFT - stage count = log2(N)
 constant c_STAGE_COUNT      : integer    := 6;

 -- accuracy of input data - bit size
 constant c_DATA_WIDTH       : integer    := 8;

  -- switch for implementation of delay in aritmetic block
 --  1 - latch is inserted between multiplier and adders, higher clk frequency
 --  0 - no latch inserted, result is load in data reg with next clk rising edge
 constant c_SYNC_ASYNC_ARITHMETIC : integer := 1;

 -- bit size of stage counter - derived from c_STAGE_COUNT
 constant c_STAGE_COUNTER_SIZE : integer := 3;

 -- size of FFT (= N) - derived from parameter c_STAGE_COUNT
 constant c_FFT_SIZE         : integer    := 2**c_STAGE_COUNT;

 -- size of constant bank, derived from size of FFT, always N/2
 constant c_CONST_COUNT      : integer    := 2**(c_STAGE_COUNT - 1);

--------------------------------------------------------------------------------
-------------- Type definition - define types used in design ----------------
--------------------------------------------------------------------------------

 -- Input and output data type (TOP -> FFT_MODULE -> TOP)
 type t_DATA_IN_OUT_FFT is array (2 * c_FFT_SIZE - 1 downto 0)
             of std_logic_vector (c_DATA_WIDTH - 1 downto 0);

 type t_LOAD_REG is array (c_FFT_SIZE - 1 downto 0)
             of std_logic_vector (c_DATA_WIDTH - 1 downto 0);

 -- complex number type, array of real and imaginary part
 type t_COMPLEX  is record
      re        : signed  (c_DATA_WIDTH - 1 downto 0);
      im        : signed  (c_DATA_WIDTH - 1 downto 0);
 end record;

 -- array of complex numbers
 type t_COMPLEX_ARRAY is array (c_FFT_SIZE  - 1 downto 0)
      of t_COMPLEX;

 -- constant bank data type
 type t_CONST_BANK is array (0 to c_CONST_COUNT - 1)
      of  signed  (c_DATA_WIDTH - 1  downto 0);

 -- latched data when synch aritmetic is used
 type t_SYNC_ASYNC_ARITHMETIC is record
      dr_hold   : std_logic;
      addr1     : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
      addr2     : std_logic_vector (c_STAGE_COUNT - 1 downto 0);
      data1     : t_COMPLEX;
      data2     : t_COMPLEX;
 end record;

 -- state machine, states: wait for new data, loading data and counting data
 type t_DRIVING_LOGIC_STATE is (e_DONE, e_LOAD, e_COUNT, e_WAIT_ON_DATA);
```

```
------------------------------------------------------------------------------
---------------------------- Function declaration ----------------------------
------------------------------------------------------------------------------

-- FFT module arithmetic functions -------------------------------------------
-- Complex multiplication - get 2 complex number and return their product
 procedure cmpl_multiplier (signal cmpl_data_1 : in  t_COMPLEX;
                            signal cmpl_data_2 : in  t_COMPLEX;
                            signal mult_result : out t_COMPLEX);

-- Complex adding - get 2 complex number and return their sum
 procedure cmpl_adder (signal cmpl_data_1 : in  t_COMPLEX;
                       signal cmpl_data_2 : in  t_COMPLEX;
                       signal add_result  : out t_COMPLEX);

-- Complex subtracting - get 2 complex number and return their difference
 procedure cmpl_subtractor (signal cmpl_data_1 : in  t_COMPLEX;
                            signal cmpl_data_2 : in  t_COMPLEX;
                            signal sub_result  : out t_COMPLEX);

-- FFT module functions used for selecting and ordering data ------------------
-- Reverse bit order, gets value and return its bit reverse order value
 function bit_reverse(sig_in : integer) return integer;

-- choose which data are send to arithmetic part
 function get_data_from_reg (
                     data_reg  : t_COMPLEX_ARRAY;
                     data_addr : std_logic_vector (c_STAGE_COUNT - 1 downto 0))
                          return t_COMPLEX;

-- Driving logic functions ----------------------------------------------------
-- generate constant address according to actual stage & step in block
 function get_const_addr (
            stage      : std_logic_vector (c_STAGE_COUNTER_SIZE - 1 downto 0);
            block_step : std_logic_vector (c_STAGE_COUNT        - 2 downto 0))
                 return std_logic_vector;

end fft_module_pkg;

package body fft_module_pkg is

------------------------------------------------------------------------------
---------------------------- Function definition -----------------------------
------------------------------------------------------------------------------

-- FFT module arithmetic functions -------------------------------------------
 procedure cmpl_multiplier (signal cmpl_data_1 : in  t_COMPLEX;
                            signal cmpl_data_2 : in  t_COMPLEX;
                            signal mult_result : out t_COMPLEX) is
   variable temp1       : signed (2*c_DATA_WIDTH downto 0);
   variable temp2       : signed (2*c_DATA_WIDTH downto 0);
   variable temp3       : signed (2*c_DATA_WIDTH downto 0);
   variable res_temp_re : signed (2*c_DATA_WIDTH downto 0);
   variable res_temp_im : signed (2*c_DATA_WIDTH downto 0);
   variable cmpl_multiplier : t_COMPLEX;

  begin
   temp1 := (resize (cmpl_data_1.re, c_DATA_WIDTH + 1)
           + resize (cmpl_data_1.im, c_DATA_WIDTH + 1)) * cmpl_data_2.re;
   temp2 := (resize (cmpl_data_2.re, c_DATA_WIDTH + 1)
           + resize (cmpl_data_2.im, c_DATA_WIDTH + 1)) * cmpl_data_1.im;
   temp3 := (resize (cmpl_data_2.im, c_DATA_WIDTH + 1)
           - resize (cmpl_data_2.re, c_DATA_WIDTH + 1)) * cmpl_data_1.re;

   -- cmpl_data_2(constants) is =< 1, result of multiplication cannot overflow
   res_temp_re := (temp1 - temp2);
   res_temp_im := (temp1 + temp3);

   cmpl_multiplier.re := res_temp_re (2 * (c_DATA_WIDTH - 1) - 1
                                      downto (c_DATA_WIDTH - 1) - 1);
   cmpl_multiplier.im := res_temp_im (2 * (c_DATA_WIDTH - 1) - 1
                                      downto (c_DATA_WIDTH - 1) - 1);
   mult_result <= cmpl_multiplier;
  end cmpl_multiplier;
```

```
 procedure cmpl_adder (signal cmpl_data_1 : in  t_COMPLEX;
                       signal cmpl_data_2 : in  t_COMPLEX;
                       signal add_result  : out t_COMPLEX) is
  begin
   add_result.re <= resize(shift_right((
                    resize(cmpl_data_1.re, c_DATA_WIDTH+1)
                  + resize(cmpl_data_2.re, c_DATA_WIDTH+1)),
                           1), c_DATA_WIDTH);
   add_result.im <= resize(shift_right((
                    resize(cmpl_data_1.im, c_DATA_WIDTH+1)
                  + resize(cmpl_data_2.im, c_DATA_WIDTH+1)),
                           1), c_DATA_WIDTH);
  end cmpl_adder;

 procedure cmpl_subtractor (signal cmpl_data_1 : in  t_COMPLEX;
                            signal cmpl_data_2 : in  t_COMPLEX;
                            signal sub_result  : out t_COMPLEX) is
  begin
   sub_result.re <= resize(shift_right((
                         resize(cmpl_data_1.re, c_DATA_WIDTH+1)
                       - resize(cmpl_data_2.re, c_DATA_WIDTH+1)),
                                1), c_DATA_WIDTH);
   sub_result.im <= resize(shift_right((
                         resize(cmpl_data_1.im, c_DATA_WIDTH+1)
                       - resize(cmpl_data_2.im, c_DATA_WIDTH+1)),
                                1), c_DATA_WIDTH);
  end cmpl_subtractor;

-- FFT module functions used for selecting and ordering data ------------------
 function bit_reverse(sig_in : integer) return integer is
     variable sig_out_logic : unsigned (c_STAGE_COUNT-1 downto 0);
     variable sig_in_logic  : unsigned (c_STAGE_COUNT-1 downto 0);
  begin
    sig_in_logic := to_unsigned(sig_in, c_STAGE_COUNT);
    for i in c_STAGE_COUNT-1 downto 0 loop
      sig_out_logic(i) := sig_in_logic(c_STAGE_COUNT - 1 - i);
    end loop;
    return to_integer(sig_out_logic);
  end bit_reverse;

  function get_data_from_reg(
             data_reg  : t_COMPLEX_ARRAY;
             data_addr : std_logic_vector (c_STAGE_COUNT - 1 downto 0))
                 return t_COMPLEX is
    variable addr : integer;
    variable addr_un : unsigned (c_STAGE_COUNT - 1 downto 0);
    variable data : t_COMPLEX;
   begin
      addr_un := unsigned(data_addr);
      addr := to_integer(addr_un);
      data := data_reg(addr);
     return data;
   end get_data_from_reg;

-- Functions for Driving logic module -----------------------------------------

  function get_const_addr (
             stage      : std_logic_vector (c_STAGE_COUNTER_SIZE - 1 downto 0);
             block_step : std_logic_vector (c_STAGE_COUNT       - 2 downto 0))
                   return std_logic_vector is

    variable const_addr : std_logic_vector (c_STAGE_COUNT       - 2 downto 0);
   begin
    case stage is
     when "000"  => const_addr := "00000";
     when "001"  => const_addr := block_step( 0 downto 0) & "0000";
     when "010"  => const_addr := block_step( 1 downto 0) & "000";
     when "011"  => const_addr := block_step( 2 downto 0) & "00";
     when "100"  => const_addr := block_step( 3 downto 0) & "0";
     when others => const_addr := block_step;
    end case;
    return const_addr;
   end get_const_addr;

end fft_module_pkg;
```

B.6 Paměť ROM pro konstanty – *const_bank.vhd*

```
--------------------------------------------------------------------------------
------------------------------ Constant bank -----------------------------------
--------------------------------------------------------------------------------

library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

use work.fft_module_pkg.all;

package const_bank_pkg is

constant c_CONST_RE  : t_CONST_BANK := (
             "01000000",
             "00111011",
             "00101101",
             "00011000",
             "00000000",
             "11101000",
             "11010011",
             "11000101"
             );


constant c_CONST_IM  : t_CONST_BANK := (
             "00000000",
             "11101000",
             "11010011",
             "11000101",
             "11000000",
             "11000101",
             "11010011",
             "11101000"
             );

end const_bank_pkg;
```

C ZDROJOVÉ KÓDY V JAZYCE SYSTEMVERILOG

C.1 Testbench – *tb.sv*

```
`timescale 1 ns/1 ps
// testbench - verification top module
import test_pkg::*;

module FFT_top_testbench ();

    t_DATA   DATA_out;       // output from FFT_top - serial data
    t_DATA   DATA_in;        // input to FFT_top - serial data
    logic    clk;            // main clock domain, used for counting FFT
    logic    clk_data_en;    // drives DATA_in sending, active one clk period
    logic    rst;            // system reset
    logic    new_data_ready; // new sending cycle started

 top_fft FFT(.*);
 FFT_top_BFM BFM(.*);
// testcase tc();
endmodule
```

C.2 Ovladač portů – *BFM.sv*

```
`timescale 1 ns/1 ps
import test_pkg::*;

module FFT_top_BFM
      (
       input  t_DATA   DATA_out,      // output from FFT_top - serial data
       output t_DATA   DATA_in,       // input to FFT_top - serial data
       output wire     clk,           // main clock domain, used for counting FFT
       output wire     clk_data_en,   // clock signal, drives DATA_in sending
       output wire     rst            // system reset
      );

  logic    clk_en = 0;               // enable clk generator
  logic    clk_var = 0;              // clk signal
  logic    clk_data_en_en = 0;       // enable clk_data_en generator
  logic    clk_data_en_var = 0;      // clk_data_en signal
  logic    load_en_var = 0;          // load_en signal
  logic    reset = 0;                // rst signal
  t_DATA   DATA_in_var = 0;          // data_in signal

  time     clk_per = c_CLK_PER;      // period of clk signal
  int      data_en_per = c_DATA_EN_AFTER_CLK_PER;
                                     // number of periods of clk_data_en signal


 // enable clk generator
 task run_clk (integer clk_run); // 1 - enable
                                 // 0 - disable
  if (clk_run) clk_en = 1;
  else  clk_en = 0;
 endtask


 // set clk and clk_data_en period
 task set_clk_per (time clk_period,
                   int  clk_data_en_period);
  clk_per = clk_period;
  data_en_per = clk_data_en_period;
 endtask
```

```
// perform reset, reset is active for 1 clk period
task perform_reset (int synch_to_negedge); // 1 - synch to clk negedge
                                           // 0 - asynch release after 2 clk_per
  reset = 0;
 #(clk_per)
  reset = 1;
 #(clk_per)
 if (synch_to_negedge == 1)
  @ (negedge clk_var);
  reset = 0;
endtask

// send and receive data to/from FFT module
task send_and_receive_data(
      input  t_DATA_ARRAY send_data_re_array,
      input  t_DATA_ARRAY send_data_im_array,
      output t_DATA_ARRAY received_data_array[1:0]);

 // 1. enable clk_data_en generator on falling edge of clk

  @ (posedge(clk_var))
   clk_data_en_en = 1;
 // 2. sending data
 for (int i = 0; i < c_FFT_SIZE; i++)
  begin
 // set Data_in defined in send_data_array on rising edge of clk_data_en
    @ (posedge(clk_data_en_var));
    DATA_in_var = send_data_re_array [i];
  end

 // 3. wait one count cycle
 for (int i = 0; i < c_FFT_SIZE -1; i++)
  begin
     @ (posedge(clk_data_en_var));
     DATA_in_var = 'd0; //send_data_re_array [i];
  end

 // 4. receiving data from FFT module
 for (int i = 0; i < c_FFT_SIZE; i++)  //i = 1
  begin
 //  sample output data when rising edge of clk_data_en
 //  and save them into received_data_array
 @ (posedge(clk_data_en_var));
     DATA_in_var = 'd0;
 @ (posedge clk_var);
 @ (negedge clk_var);
     received_data_array [0][i] = DATA_out;
  end

 // 5. disable clk_data_en
    @(negedge(clk_data_en_var))
    clk_data_en_en = 0;

 endtask

// clk generator
always
 begin
  if (clk_en) begin
     clk_var = 1'b1;
    # (clk_per/2);
     clk_var = 1'b0;
    # (clk_per/2);
  end else begin
     clk_var = 1'b0;
   @ (clk_en);
  end
 end

// clk_data_en generator, synchronize with rising edge of clk
always
 begin
 @ (posedge(clk_var));
  while (clk_data_en_en)
   begin
    @ (negedge(clk_var));
     clk_data_en_var = 1'b1;
    # (clk_per);
     clk_data_en_var = 1'b0;
    repeat (data_en_per - 1)
      @ (posedge clk_var);
   end
    clk_data_en_var = 1'b0;
 end
```

```
 // output assignment, generated signals are assgined with output signals
 assign rst = reset;
 assign clk = clk_var;
 assign clk_data_en = clk_data_en_var;
 assign DATA_in = DATA_in_var;
endmodule
```

C.3 Testcase – *testcase.sv*

```
`timescale 1 ns/1 ps
import test_pkg::*;
`include "ref_fft.sv"
program testcase;
// module testcase;

task init;
 FFT_top_testbench.BFM.set_clk_per(100ns, 10);
 FFT_top_testbench.BFM.run_clk(1);
 FFT_top_testbench.BFM.perform_reset(1);
endtask : init

task randomize_data(
     output t_DATA_ARRAY data_bit_re,
     output t_DATA_ARRAY data_bit_im,
     output t_COMPLEX data_compl [c_FFT_SIZE - 1 : 0]
                 );
     int rand_re;
     int rand_im;

 for (int a = 0; a < c_FFT_SIZE; a++)
  begin
  // generate real part of input data
   rand_re = $random%(2**(c_DATA_WIDTH - 1));
   data_compl[a].r = real'(rand_re)/real'(2**(c_DATA_WIDTH - 1));
   data_bit_re [a] =  t_DATA'(rand_re);

  // generate imaginary part of input data
  if(c_COMPLEX_EN == 1)
   rand_im = $random%(2**(c_DATA_WIDTH - 1));
  else
   rand_im = 0;

   data_compl[a].i = real'(rand_im)/real'(2**(c_DATA_WIDTH - 1));
   data_bit_im [a] =  t_DATA'(rand_im);
  end

  if (c_VERBOSE > 1)
   begin
    for (int i =0; i< c_FFT_SIZE; i++)
     begin
       $write("DEBUG randomize: Data randomization: pos:%%2d.: model stimuli: re =%%5f,im= %%5f,
design stimuli: re =%%5d, im=%%5d.\n",
         i, data_compl[i].r, data_compl[i].i, int'(data_bit_re[i]), int'(data_bit_im[i]));
     end
  end
 endtask

// task send_receive_data
// 1.  Build refernece model object (call constructor new in ref_fft class)
// 2.  Generate reference output in real type
// 3.  Send generated data into DUT (call task send_and_receive_data in BFM module)
task send_receive_data( input  t_COMPLEX data_real_in  [c_FFT_SIZE - 1 : 0],
                        output t_COMPLEX data_real_out [c_FFT_SIZE - 1 : 0],
                        input  t_DATA_ARRAY data_bit_re_in,
                        input  t_DATA_ARRAY data_bit_im_in,
                        output t_DATA_ARRAY data_bit_out[1:0]);

  automatic ref_fft ref_model = new();
  ref_model.run_FFT (data_real_in, data_real_out);
  FFT_top_testbench.BFM.send_and_receive_data (data_bit_re_in,
                                               data_bit_im_in,
                                               data_bit_out[1:0]);

endtask
```

```
// Check received data - compare with reference model result
task check_data (input int run,
                 input t_COMPLEX data_real_out [c_FFT_SIZE - 1 : 0],
                 input t_DATA_ARRAY data_bit_out[1:0]);

 t_COMPLEX data_bit2real [c_FFT_SIZE - 1 : 0];
 automatic int ind = 0;

 // transform received data
 if (c_DEBUG == 0) // do not use, used only for ver env debug
  begin
    data_bit2real[0].r = conv_bit2real(data_bit_out[0][0]);
    data_bit2real[0].i = 0.0;
   for (int i = 1; i < c_FFT_SIZE/2; i++)
    begin
     data_bit2real[i].r = conv_bit2real(data_bit_out[0][i + ind    ]);
     data_bit2real[i].i = conv_bit2real(data_bit_out[0][i + ind + 1]);

     data_bit2real[c_FFT_SIZE - i].r = data_bit2real[i].r;
     data_bit2real[c_FFT_SIZE - i].i = -data_bit2real[i].i;
     ind++;
    end
   data_bit2real[c_FFT_SIZE/2].r = conv_bit2real(data_bit_out[0][c_FFT_SIZE-1]);
   data_bit2real[c_FFT_SIZE/2].i = 0.0;
  end
 else
  begin
   for (int i = 0; i < c_FFT_SIZE-1; i++)
    begin
     data_bit2real[i].r =conv_bit2real(data_bit_out[i][0]); //FIXME
     data_bit2real[i].i =conv_bit2real(data_bit_out[i][1]); //FIXME
    end
 end

 // compare data
  if (c_VERBOSE == 0)
   begin
   for (int position = 0; position < c_FFT_SIZE; position ++)
    begin
     if ((data_real_out[position].r > data_bit2real[position].r + c_TOLERANCE) ||
         (data_real_out[position].r < data_bit2real[position].r - c_TOLERANCE))
       begin
         $write("ERROR check_data: %%2d. run, %%3d position, wrong real data received, reference
model result: %%f, received data: %%f.\n",
                run, position, data_real_out[position].r, data_bit2real[position].r);
       end
     if ((data_real_out[position].i > data_bit2real[position].i + c_TOLERANCE) ||
         (data_real_out[position].i < data_bit2real[position].i - c_TOLERANCE))
       begin
         $write("ERROR check_data: %%2d. run, %%3d position, wrong complex data received, reference
model result: %%f, received data: %%f.\n",
                run, position, data_real_out[position].i, data_bit2real[position].i);
       end
    end
   end

  // Print out received data and expected data.
  if (c_VERBOSE > 0)
   begin
   for (int position = 0; position < c_FFT_SIZE; position ++)
    begin

         $write("DEBUG check_data: %%2d. run, position: %%3d, real data received, ref model result:
%%f, received data: %%f.\n",
                run, position, data_real_out[position].r, data_bit2real[position].r);

         $write("DEBUG check_data: %%2d. run, position: %%3d, cmpl data received, ref model result:
%%f, received data: %%f.\n",
                run, position, data_real_out[position].i, data_bit2real[position].i);
    end
   end

endtask
```

```
task run_random_data_test (int run);
 t_DATA_ARRAY data_bit_re_in;
 t_DATA_ARRAY data_bit_im_in;
 t_DATA_ARRAY data_bit_out[1:0];
 t_COMPLEX data_real_in  [c_FFT_SIZE - 1 : 0];
 t_COMPLEX data_real_out [c_FFT_SIZE - 1 : 0];

   randomize_data(data_bit_re_in, data_bit_im_in, data_real_in);
   send_receive_data (data_real_in,    data_real_out,
                      data_bit_re_in,  data_bit_im_in,
                      data_bit_out[1:0]);

   check_data (run, data_real_out, data_bit_out[1:0]);
endtask

initial
 begin
 init;
 for (int run = 0; run < 50; run ++)
  begin
    #10us;
    run_random_data_test (run+1);
    #10us;
  end
  FFT_top_testbench.BFM.run_clk(0);
 end

endprogram
// endmodule
```

C.4 Referenční model – *ref_fft.sv*

```
`timescale 1 ns/1 ps
import test_pkg::*;
import const_bank_for_ver::*;
class ref_fft;

 // complex adder
 local function t_COMPLEX add(t_COMPLEX a, b);
     add.r = a.r + b.r;
     add.i = a.i + b.i;
 endfunction


 // complex subtractor
 local function t_COMPLEX sub(t_COMPLEX a, b);
     sub.r = a.r - b.r;
     sub.i = a.i - b.i;
 endfunction


 // complex multiplier
 local function t_COMPLEX mul(t_COMPLEX a, b);
     mul.r = (a.r * b.r) - (a.i * b.i);
     mul.i = (a.r * b.i) + (a.i * b.r);
 endfunction


 //reverse addres bits
 local task get_data_addr (output integer   addr_data  [c_FFT_SIZE-1 : 0]);
  logic  [c_STAGE_COUNT-1 : 0] addr;
  for (int i = 0;i <c_FFT_SIZE; i++)
   begin
     addr         = i;               // convert address in bit values
     addr         = {<<{addr}};      // bit reverse
     addr_data[i] = integer'(addr);  // convert bit reversed address to integer
   end
 endtask


 // data order is reversed acording to bit reversed address
 local task reverse_bit_data_order(input t_COMPLEX data_in   [c_FFT_SIZE-1 : 0],
                                   output t_COMPLEX data      [c_FFT_SIZE-1 : 0],
                                   input integer   addr_data [c_FFT_SIZE-1 : 0]);
  for (int i = 0; i < c_FFT_SIZE; i++)
   begin
     data[i] = data_in [addr_data[i]];
   end
  endtask
```

```
 // count constants for FFT and save them into constant bank
 local task count_const (output t_COMPLEX const_bank [c_FFT_SIZE/2 - 1 : 0]);
  real cmpl_angle;
  // real const_r;
  // real const_i;
  for (int addr = 0; addr < c_FFT_SIZE/2; addr++)
   begin
    // cmpl_angle = 3.14159 / real'(c_FFT_SIZE/2) * real'(addr);
    // const_r =  cos(cmpl_angle);
    // const_i = -sin(cmpl_angle);
    const_bank[addr] = '{c_CONST_VER_RE[addr],c_CONST_VER_IM[addr]};
    if (c_VERBOSE > 1)
     begin
      $write("DEBUG: Const gen: Addr: %%2d, const_re = %%4f, const_im = %%4f \n",
             addr, const_bank[addr].r, const_bank[addr].i);
     end
   end
 endtask


// FFT algorithm
 local task count_FFT (input  t_COMPLEX data_in    [c_FFT_SIZE-1    : 0],
                       output t_COMPLEX data_out   [c_FFT_SIZE-1    : 0],
                       input  t_COMPLEX const_bank [c_FFT_SIZE/2 -1 : 0]);
  t_COMPLEX      constants;
  t_COMPLEX      temp1,temp2;
  t_COMPLEX      data  [c_FFT_SIZE-1 : 0];
  int block;

  data = data_in;
  for (int stage = 0; stage < c_STAGE_COUNT; stage++)
   begin
    for ( int addr = 0; addr < c_FFT_SIZE; addr++)
      begin
        block = 0;
        while (block < (2**stage))
         begin
          constants = const_bank[2**(c_STAGE_COUNT-stage-1) * block];
          temp1 = add(data [addr], mul(constants, data[addr + 2**stage]));
          temp2 = sub(data [addr], mul(constants, data[addr + 2**stage]));

          data[addr] = temp1;
          data[addr + 2**stage] = temp2;
        if (c_VERBOSE > 1)
         begin
          $write("DEBUG, ref_model: stage = %%1d addr = %%3d, data = %%4f.\n",
                 stage, addr, temp1);
          $write("DEBUG, ref_model: stage = %%1d addr = %%3d, data = %%4f.\n",
                 stage, addr + 2**stage, temp2);
         end
          addr  = addr  + 1;
          block = block + 1;
         end
       addr=addr+2**stage - 1;
      end
   end
   data_out = data;
endtask

// run FFT counting, task run previous local task in right order and return complex output
task run_FFT (input  t_COMPLEX data_in  [c_FFT_SIZE-1 : 0],
              output t_COMPLEX data_out [c_FFT_SIZE-1 : 0]);

  t_COMPLEX data       [c_FFT_SIZE-1 : 0];
  t_COMPLEX const_bank [c_FFT_SIZE/2 -1 : 0];
  integer   addr_data  [c_FFT_SIZE-1 : 0];

    get_data_addr(addr_data);
    reverse_bit_data_order(data_in, data, addr_data);
    count_const (const_bank);
    count_FFT(data, data_out, const_bank);
endtask

endclass
```

C.5 Knihovní balík pro verifikaci – *test_pkg.sv*

```
`timescale 1 ns/1 ps
package test_pkg;

 // function sin and cos, C function, imported into simulation enviroment
 // import "DPI-C" pure function real sin (input real rTheta);
 // import "DPI-C" pure function real cos (input real rTheta);

  // verbose parameter for verification: -1 -> no report
  //                                     0 -> report error in received data
  //                                     1 -> print out received & expected data
  //                                     2 -> print all debug messages
  parameter c_VERBOSE = 0;

  // use complex/real data on input/output
  localparam c_COMPLEX_EN  = 0;   // 0 - real
                                  // 1 - complex
  localparam c_DEBUG = 0;
  // design parameters
  localparam c_DATA_WIDTH  = 8;  // bit size of input/output data
  localparam c_STAGE_COUNT = 6;   // stage count = log2(c_FFT_SIZE);
  localparam c_FFT_SIZE    = 2**c_STAGE_COUNT;  // FFT input/output samples
  localparam c_TOLERANCE = 1.0/(2.0**real'(c_DATA_WIDTH-1-c_STAGE_COUNT));
  // clock period
  localparam c_CLK_PER               = 100ns;
  localparam c_DATA_EN_AFTER_CLK_PER = 10;

  // data type definition
  typedef logic [c_DATA_WIDTH-1 : 0] t_DATA;
  typedef logic [c_DATA_WIDTH-1 : 0] t_DATA_ARRAY [c_FFT_SIZE - 1 : 0];

  // complex numbers type
  typedef struct {real r,i;} t_COMPLEX;

 //
  function real conv_bit2real(input t_DATA data_logic);
   real data_real;
  // if negative value
   if (data_logic[c_DATA_WIDTH - 1] == 1'b1)
    data_real = real'(data_logic) - real'(2**c_DATA_WIDTH);
   else
    data_real = real'(data_logic);
   conv_bit2real = data_real/(2.0**real'(c_DATA_WIDTH-1-c_STAGE_COUNT));
  endfunction : conv_bit2real


endpackage : test_pkg
```

D ZDROJOVÉ KÓDY GENERÁTORU FFT

D.1 Funkce pro generování kódu FFT modulu

```
//---------------------------------------------------------------------------
#pragma hdrstop

#include "generate_hdl_sources.h"
#include <stdio.h>
#include <string.h>
#include <math.h>
//---------------------------------------------------------------------------

#pragma package(smart_init)

#define FILE_COUNT 4
#define MAX_BIT_WIDTH 32
#define MAX_STAGE_CNT 16
#define TEMP_HDL_SOURCES_ADDR "temp_rtl/"
#define HDL_SOURCES_ADDR "rtl/"
#define HDL_FILE_NAMES {"driving_logic",       \
                        "fft_module",          \
                        "loading_logic",    \
                        "top_fft"};


 int fce_gen_hdl_sources (int stage_count,      // number of stages, log2(N)
                          int bit_size,         // bit width of input data
                          int arith_synch,      // use one clk delay in arith part
                          int not_use_load_reg, // generate only fft_module
                          FILE *report)
  {
   int file_ind;
   int _error = 0;

  // generate constant bank file and definition package
   _error = _error + fce_gen_const_bank_file(pow(2, stage_count-1),bit_size,report);
   _error = _error + fce_gen_def_pkg_file(stage_count, bit_size, arith_synch, report);

  // copy templates file into hdl directory
   for (file_ind = 0; file_ind < (FILE_COUNT - 2 * not_use_load_reg); file_ind ++)
    {
     _error = _error + fce_copy_files(file_ind,report);
    }
   return _error;
  }

  int fce_gen_const_bank_file(int const_count, int bit_size, FILE *report)
  {
   FILE *const_bank;
   int   const_addr;
   int   const_int;
   int   conv;
   int   __error = 0;
   float const_float;
   char  _string [MAX_BIT_WIDTH];

   const_bank = fopen(HDL_SOURCES_ADDR"/const_bank.vhd", "w");
   if (const_bank == NULL)
    {
     __error = __error + 1;
     fprintf(report, "Error during opening const_bank.vhd file!\n");
    }
   else
  {
    // print header and libraries
   fprintf(const_bank, "--------------------------------------------------------------------------------
\n");
   fprintf(const_bank, "------------------------------ Constant bank -------------------------------
\n");
   fprintf(const_bank, "--------------------------------------------------------------------------------
\n\n");

   fprintf(const_bank, "library IEEE; \n");
   fprintf(const_bank, "use IEEE.STD_LOGIC_1164.all; \n");
```

```
  fprintf(const_bank, "use IEEE.NUMERIC_STD.all; \n\n");

  fprintf(const_bank, "use work.fft_module_pkg.all; \n\n");

  fprintf(const_bank, "package const_bank_pkg is \n\n");

  fprintf(const_bank, "constant c_CONST_RE  : t_CONST_BANK := ( \n");

 // ------------------ generate constant values --------------------
  for (const_addr = 0; const_addr < const_count; const_addr++)
   {
   // generate constants (real part) in real numbers and transform into integer
    const_float = (cos(3.14159/const_count*const_addr))*pow(2,bit_size-1-1);
    const_int = (int)const_float;

   // convert integer to binary values
    fce_int2bin(const_int, bit_size, _string);

    fprintf(const_bank, "             \"%s\"",_string);
    if (const_addr < const_count -1)
     {
      fprintf(const_bank, ", ");}
      fprintf(const_bank, "\n");
     }
  fprintf(const_bank, "             ); \n\n\n");

  fprintf(const_bank, "constant c_CONST_IM  : t_CONST_BANK := ( \n");

  for (const_addr = 0; const_addr < const_count; const_addr++)
   {
   // generate constants (imag part) in real numbers and transform into integer
    const_float = -(sin(3.14159/const_count*const_addr))*pow(2,bit_size-1 -1);
    const_int = (int)const_float;

   // convert integer to binary values
    fce_int2bin(const_int, bit_size, _string);

    fprintf(const_bank, "             \"%s\"",_string);
    if (const_addr < const_count -1)
      fprintf(const_bank, ", ");

    fprintf(const_bank, "\n");
   }
    fprintf(const_bank, "             ); \n\n");

    fprintf(const_bank, "end const_bank_pkg; \n\n");

   // report succesful constant bank generating
    fprintf(report, "Constant bank generated for FFT size %d, data width is %d.\n",
                     2 * const_count, bit_size);

    fclose(const_bank);
  }
  return __error;
 }


// convert integer to binary numbers
void fce_int2bin(int const_int, int bit_size, char* _string)
 {
   int i;
   _string[bit_size] = '\0';
   for (i = bit_size - 1; i >= 0; --i, const_int >>= 1)
    {
     _string[i] = (const_int & 1) + '0';
    }
 }

// generate definition package
int fce_gen_def_pkg_file(int stage_count,
                         int bit_size,
                         int arith_sync,
                         FILE *report)
 {
  FILE *def_pkg;
  FILE *def_pkg_template;
  char line [85];
  char _string [MAX_STAGE_CNT];
  int  __error = 0;
  int  index;
  int  stage_cnt_size; // FFT maximal size is here limited by 2**16

        if (stage_count < 2)    stage_cnt_size = 0;
   else if (stage_count < 3)    stage_cnt_size = 1;
```

```
    else if (stage_count < 5)    stage_cnt_size = 2;
    else if (stage_count < 9)    stage_cnt_size = 3;
    else                         stage_cnt_size = 4;

   // open template and target file
    def_pkg = fopen(HDL_SOURCES_ADDR"fft_module_pkg.vhd", "w");
    def_pkg_template = fopen (TEMP_HDL_SOURCES_ADDR"fft_module_pkg_temp.vhd", "r");

      if (def_pkg_template == NULL)
    {
     fprintf(report, "Error during opening fft_module_pkg_temp.vhd file!\n");
     __error = __error + 1;
    }
     else if (def_pkg == NULL)
    {
     fprintf(report, "Error during opening fft_module_pkg.vhd file!\n");
     __error = __error + 1;
    }
     else
   {
    while (!feof(def_pkg_template))
     {
      fgets(line,sizeof(line) - 1,def_pkg_template);
      if (strstr(line,"--insert design parameters--"))
       {
        fprintf(def_pkg, " -- stage count - derived from size of FFT - stage count = log2(N)\n");
        fprintf(def_pkg, " constant c_STAGE_COUNT      : integer    := %d;\n\n", stage_count);

        fprintf(def_pkg, " -- accuracy of input data - bit size\n");
        fprintf(def_pkg, " constant c_DATA_WIDTH       : integer    := %d;\n\n",bit_size);

        fprintf(def_pkg, "  -- switch for implementation of delay in aritmetic block\n");
        fprintf(def_pkg, " --  1 - latch is inserted between multiplier and adders, higher clk
frequency\n");
        fprintf(def_pkg, " --  0 - no latch inserted, result is load in data reg with next clk rising
edge\n");
        fprintf(def_pkg, " constant c_SYNC_ASYNC_ARITHMETIC : integer := %d;\n\n", arith_sync);

        fprintf(def_pkg, " -- bit size of stage counter - derived from c_STAGE_COUNT\n");
        fprintf(def_pkg, " constant c_STAGE_COUNTER_SIZE : integer := %d;\n\n", stage_cnt_size);
       }
      else
      if (strstr(line,"-- insert const addr case --"))
      {
        fprintf(def_pkg, "    case stage is\n");
         fce_int2bin (0, stage_cnt_size, _string);
         fprintf(def_pkg, "     when \"%s\"  =>", _string);
         fce_int2bin (0, stage_count-1, _string);
         fprintf(def_pkg, " const_addr := \"%s\";\n", _string);

        for (index = 1; index < stage_count-1; index ++)
        {
         fce_int2bin (index, stage_cnt_size, _string);
         fprintf(def_pkg, "     when \"%s\"  =>", _string);
         fce_int2bin (0, stage_count - index - 1, _string);
         fprintf(def_pkg, " const_addr := block_step(%2d downto 0) & \"%s\";\n",index-1, _string);
        }
         fprintf(def_pkg, "     when others =>");
         fprintf(def_pkg, " const_addr := block_step;\n");
         fprintf(def_pkg, "    end case;\n");
      }
      else
       fprintf(def_pkg, line);
     }

     fprintf(report, "FFT design definition package generated.\n");
     fclose(def_pkg);
     fclose(def_pkg_template);
   }
  return __error;
}

 int fce_copy_files ( int file_ind, FILE *report)
  {
   FILE *file;
   FILE *file_out;
   char temp_file_name[50];
   char hdl_file_name[50];
   char line [85];
   char file_list[FILE_COUNT] [20] = HDL_FILE_NAMES
   int  __error = 0;

   strcpy(temp_file_name,TEMP_HDL_SOURCES_ADDR);
   strcat(temp_file_name,file_list[file_ind]);
```

```
   strcat(temp_file_name,".vhd");

   strcpy(hdl_file_name,HDL_SOURCES_ADDR);
   strcat(hdl_file_name,file_list[file_ind]);
   strcat(hdl_file_name,".vhd");

   file = fopen(temp_file_name,"r");
   file_out = fopen(hdl_file_name,"w");

   if (file == NULL || file_out == NULL)
    {
     __error = __error + 1;
     fprintf(report, "Error during copying source file %s into rtl/ directory.\n \
                      Check if template file and rtl/ directory exist.\n",
             hdl_file_name);
    }
   else
    {
     fprintf(report, "Source file %s was copied into rtl/ directory.\n",
             temp_file_name);
     while (!feof(file))
      {
       fgets(line,sizeof(line),file);
       fprintf(file_out, line);
      }

     fclose(file);
     fclose(file_out);
    }
  return __error;
 }
```

D.2 Funkce pro generování kódu verifikačního prostředí

```
//---------------------------------------------------------------------------
#pragma hdrstop

#include "generate_ver_env.h"
#include <stdio.h>
#include <string.h>
#include <math.h>
//---------------------------------------------------------------------------

#pragma package(smart_init)

#define FILE_COUNT 4
#define TEMP_VER_SOURCES_ADDR "temp_models/"
#define VER_SOURCES_ADDR "models/"
#define VER_FILE_NAMES {"bfm",       \
                        "ref_fft",   \
                        "tb",        \
                        "testcase"};


 int fce_gen_ver_env (int stage_count, int bit_size, FILE *report)
  {
   int file_ind;
   int _error = 0;

   _error = _error + fce_gen_ver_def_pkg_file(stage_count, bit_size, report);
   _error = _error + fce_gen_ver_const_bank(pow(2, stage_count-1), report);
   for (file_ind = 0; file_ind < FILE_COUNT; file_ind ++)
    {
     _error = _error + fce_copy_ver_files(file_ind,report);
    }
    return _error;
  }

 int fce_gen_ver_def_pkg_file(int  stage_count,
                              int  bit_size,
                              FILE *report)
  {
   FILE *def_pkg;
   FILE *def_pkg_template;
   char line [85];
   int __error = 0;


    def_pkg = fopen(VER_SOURCES_ADDR"test_pkg.sv", "w");
    def_pkg_template = fopen (TEMP_VER_SOURCES_ADDR"test_pkg_temp.sv", "r");
```

```
    if (def_pkg_template == NULL)
    {
     fprintf(report, "Error during opening test_pkg_temp.sv file!\n");
     __error = __error + 1;
    }
     else if (def_pkg == NULL)
    {
     fprintf(report, "Error during opening test_pkg.sv file!\n");
     __error = __error + 1;
    }
     else
   {
    while (!feof(def_pkg_template))
     {
      fgets(line,sizeof(line) - 1,def_pkg_template);
      if (strstr(line,"/*insert verification parameters*/"))
       {
        fprintf(def_pkg, "  localparam c_DATA_WIDTH  = %d;  // bit size of input/output
data\n",bit_size);
        fprintf(def_pkg, "  localparam c_STAGE_COUNT = %d;   // stage count = log2(c_FFT_SIZE);\n",
stage_count);

       }
      else
       fprintf(def_pkg, line);
     }

     fprintf(report, "FFT verification parameters package generated.\n");
     fclose(def_pkg);
     fclose(def_pkg_template);
   }
   return __error;
}

 int fce_gen_ver_const_bank (int const_count, FILE *report)
  {
   FILE *const_bank;
   int  const_addr;
   float const_float;
   int __error = 0;

   const_bank = fopen(VER_SOURCES_ADDR"/const_ver_bank.sv", "w");
   if (const_bank == NULL)
    {
     fprintf(report, "Error during opening verification const_bank.sv file!\n");
     __error = __error + 1;
    }
    else
   {
   fprintf(const_bank, "//-------------------------------------------------------------------------------
\n");
   fprintf(const_bank, "//------------------- Constant bank for verification ------------------------
\n");
   fprintf(const_bank, "//-------------------------------------------------------------------------------
\n\n");

   fprintf(const_bank, "`timescale 1 ns/1 ps \n\n");

   fprintf(const_bank, "package const_bank_for_ver; \n\n");

   fprintf(const_bank, "typedef real  t_CONST_ARRAY [0 : %d]; \n\n",const_count-1);

   fprintf(const_bank, "const t_CONST_ARRAY c_CONST_VER_RE  = { \n");

   for (const_addr = 0; const_addr < const_count; const_addr++)
    {
     const_float = cos(3.14159/const_count*const_addr);

     fprintf(const_bank, "             %12f",const_float);
     if (const_addr < const_count -1)
      {
       fprintf(const_bank, ", ");}
       fprintf(const_bank, "\n");
      }
   fprintf(const_bank, "             }; \n\n\n");

   fprintf(const_bank, "const t_CONST_ARRAY c_CONST_VER_IM = {  \n");

   for (const_addr = 0; const_addr < const_count; const_addr++)
    {
     const_float = -(sin(3.14159/const_count*const_addr));

     fprintf(const_bank, "             %12f",const_float);
```

```
    if (const_addr < const_count-1)
      fprintf(const_bank, ", ");

    fprintf(const_bank, "\n");
   }
    fprintf(const_bank, "             }; \n\n");

    fprintf(const_bank, "endpackage : const_bank_for_ver; \n\n");

    fprintf(report, "Verification constant bank generated\n");

    fclose(const_bank);
  }
  return __error;
 }

int fce_copy_ver_files (int file_ind, FILE *report)
 {
  FILE *file;
  FILE *file_out;
  char temp_file_name[50];
  char hdl_file_name[50];
  char line [85];
  char file_list[FILE_COUNT] [20] = VER_FILE_NAMES
  int __error = 0;

  strcpy(temp_file_name,TEMP_VER_SOURCES_ADDR);
  strcat(temp_file_name,file_list[file_ind]);
  strcat(temp_file_name,".sv");

  strcpy(hdl_file_name,VER_SOURCES_ADDR);
  strcat(hdl_file_name,file_list[file_ind]);
  strcat(hdl_file_name,".sv");

  file = fopen(temp_file_name,"r");
  file_out = fopen(hdl_file_name,"w");

  if (file == NULL || file_out == NULL)
   {
    __error = __error + 1;
    fprintf(report, "Error during copying source file %s into models/ directory.\n \
                     Check if template file and models/ directory exist.\n",
            hdl_file_name);
   }
  else
   {
    fprintf(report, "Source file %s was copied into models/ directory.\n",
            temp_file_name);
    while (!feof(file))
     {
      fgets(line,sizeof(line),file);
      fprintf(file_out, line);
     }

    fclose(file);
    fclose(file_out);
   }
 return __error;
}
```