FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

**Bakalářská práce**

# Akcelerátor ztrátové komprese na hradlovém poli

| | |
|---|---|
| *Studijní program:* | B0613A140005 Informační technologie |
| *Studijní obor:* | Aplikovaná informatika |
| | |
| *Autor práce:* | **Karel Najman** |
| *Vedoucí práce:* | Ing. Martin Rozkovec, Ph.D.<br>Ústav informačních technologií a elektroniky |

Liberec 2023

FAKULTA MECHATRONIKY,
INFORMATIKY A MEZIOBOROVÝCH
STUDIÍ TUL

## Zadání bakalářské práce

# Akcelerátor ztrátové komprese na hradlovém poli

| | |
|---|---|
| *Jméno a příjmení:* | **Karel Najman** |
| *Osobní číslo:* | M20000218 |
| *Studijní program:* | B0613A140005 Informační technologie |
| *Specializace:* | Aplikovaná informatika |
| *Zadávající katedra:* | Ústav informačních technologií a elektroniky |
| *Akademický rok:* | 2022/2023 |

**Zásady pro vypracování:**

1. Seznamte se základními a pokročilými metodami ztrátové komprese obrazu. Dále se seznamte s platformou APSoC Zynq a balíkem vývojových nástrojů Xilinx Vitis.
2. Algoritmus enkodéru naprogramujte pro vnořená jádra ARM v Zynq.
3. Vyberte části algoritmu vhodné pro implementaci v HW a popište je v jazyce VHDL.
4. Zvolené řešení implementujte v FPGA obvodu a porovnejte s referenčním

| | |
|---|---|
| *Rozsah grafických prací:* | dle potřeby dokumentace |
| *Rozsah pracovní zprávy:* | 30-40 stran |
| *Forma zpracování práce:* | tištěná/elektronická |
| *Jazyk práce:* | Čeština |

**Seznam odborné literatury:**


[1] Khalid Sayood. Indtroduction to Data Compresssion, 5th Edition, 2017, Morgan Kaufmann, ISBN: 9780128094747

[2] Xilinx Inc., Vivado Design Suite User Guide: High-Level Synthesis (ug902), 2019, online <https://bit.ly/3428vjl>, 24. 10. 2109

[3] Topiwala, P.N.: Wavelet Image and Video Compression, 1998, Springer, ISBN: 978-0792381822

[4] Machat, Jáchym: Ztrátové komprese obrazu v embedded zařízeních, Bakalářská práce, TUL, 2021


| | |
|---|---|
| *Vedoucí práce:* | Ing. Martin Rozkovec, Ph.D.<br>Ústav informačních technologií a elektroniky |
| *Datum zadání práce:* | 24. října 2022 |
| *Předpokládaný termín odevzdání:* | 22. května 2023 |

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 24. října 2022

# Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

22. května 2023 Karel Najman

# Akcelerátor ztrátové komprese na hradlovém poli


## Abstrakt

Tato bakalářská práce se zabývá implementací ztrátové komprese na FPGA, konkrétně na vývojové desce ZedBoard. Práce začíná seznámením se vývojovými nástroji balíku Xilinx Vitis a pokročilými metodami ztrátové komprese, jako jsou diskrétní kosinová transformace, diskrétní vlnková transformace, kvantizace, RLE (Run-Length Encoding), Huffmanovo kódování. Na základě této teoretické analýzy je navrhnut implementovatelný postup ztrátové komprese, který je odvozen od standardu JPEG.

Pro implementaci ztrátové komprese na FPGA je realizován výpočet 2D DCT (diskrétní kosinové transformace), který je klíčovým krokem pro získání frekvenční reprezentace obrazu. Dále je provedena kvantizace a cikcak uspořádání, které slouží k redukci dat a vytvoření spojitého toku. Procesory platformy ZedBoard, konkrétně ARM Cortex-A9, jsou využity k provádění RLE (Run-Length Encoding). Tento krok slouží k efektivní kompresi dat a minimalizaci jejich velikosti.

Výsledná implementace dosahuje akceptovatelného kompresních výsledků při vysokém výpočetním výkonu. Výsledky experimentů potvrzují funkcionalitu a výkonnost navržené implementace a naznačují její potenciální využití pro praktické aplikace ve zpracování obrazových dat.



**Klíčová slova:** ztrátová komprese obrazu, FPGA, VHDL, knihovna fixed point package, DCT, diskrétní kosinová transformace kvantizace, cikcak, RLE

# Lossy compression accelerator on FPGA


## Abstract

This bachelor's thesis deals with the implementation of lossy compression on FPGA, specifically on the ZedBoard development board. The work begins with an introduction to the development tools of the Xilinx Vitis software and advanced methods of lossy compression, such as discrete cosine transform, discrete wavelet transform, quantization, RLE (Run-Length Encoding), Huffman coding. Based on this theoretical analysis, an implementable lossy compression procedure is proposed, which is derived from the JPEG standard.

To implement lossy compression on the FPGA, a 2D DCT (discrete cosine transform) calculation is implemented, which is a key step for obtaining the frequency representation of the image. Next, quantization and zigzag arrangement are performed, which serve to reduce data and create a continuous flow. ZedBoard platform processors, specifically ARM Cortex-A9, are used to perform RLE (Run-Length Encoding). This step serves to effectively compress the data and minimize its size.

The resulting implementation achieves acceptable compression results at high computing power. The results of the experiments confirm the functionality and performance of the proposed implementation and indicate its potential use for practical applications in image data processing.



**Keywords:** Lossy image compression, FPGA, VHDL, fixed point package, DCT,discrete cosine transform quantization, zigzag, RLE

# Poděkování

Rád bych vyjádřil své upřímné díky vedoucímu mé bakalářské práce, panu Ing. Martinovi Rozkovci, Ph.D, za jeho odborné vedení, časově náročné konzultace, odpovědnost za sestavení IP jádra a cenné rady při psaní práce. Děkuji také mé rodině, přítelkyni a přátelům, kteří mi poskytli podporu, inspiraci a motivaci a zázemí při mém bakalářském studiu. Dále bych rád poděkoval své alma mater, Technické univerzitě v Liberci, která mi poskytla rozhled do velké škály oblastí které spadají pod rámec studia informatiky. Možnost vzdělání a rozvíjet své schopnosti a znalosti. V neposlední řadě bych rád poděkovala všem pedagogům, spolužákům a kolegům, kteří mi pomáhali v průběhu studia a při psaní této práce. Vaše rady, názory a podněty mi velmi pomohly a obohatily můj pohled na danou problematiku.

# Obsah


Seznam zkratek . . . 10

**1 Úvod** **11**

**2 Použité prostředky** **12**
2.1 Xilinx Vitis 2022.2 . . . 12
2.2 MATLAB R2022b . . . 12
2.3 ZedBoard . . . 13

**3 Ztrátová komprese obrazu** **14**
3.1 Diskrétní kosinová transformace . . . 15
3.2 Diskrétní vlnková transformace . . . 17
3.3 Datové typy s řádovou čárkou . . . 19
3.3.1 Datové typy s pevnou řádovou čárkou . . . 19
3.3.2 Datové typy s plovoucí řádovou čárkou . . . 20
3.4 JFIF/JPEG . . . 21
3.4.1 Převedení obrazových dat z RGB do YCbCr . . . 21
3.4.2 Podvzorkování barvonosných složek . . . 22
3.4.3 Kvantizace . . . 22
3.4.4 Cik-cak . . . 23
3.4.5 RLE kódování . . . 23
3.4.6 Huffmanovo kódování . . . 24

**4 Návrh řešení** **26**
4.1 DCT . . . 26
4.1.1 Kvantizace . . . 27
4.1.2 Cik-cak . . . 28
4.1.3 RLE . . . 28
4.2 Implementace kódu v jazyku C pro vnořené jádro Zynq . . . 29
4.2.1 DCT . . . 29
4.2.2 Transpozice . . . 30
4.2.3 Kvantizace . . . 30
4.2.4 Cik-cak . . . 30
4.2.5 RLE . . . 31
4.3 Implementace ve VHDL . . . 32
4.3.1 Určení numerické přesnosti . . . 32

4.3.2 DCT . . . 34
4.3.3 Transpozice matice . . . 35
4.3.4 Kvantizace a cik-cak . . . 36
4.3.5 RLE . . . 37
4.4 Testování . . . 39
4.4.1 Test DCT . . . 40
4.4.2 Testování DCT2 kvantizace s cik-cak reindexací . . . 40
4.5 Výsledky z FPGA . . . 42
4.6 Porovnání rychlosti . . . 43

**5 Závěr** **44**

**Použitá literatura** **46**

**A Přílohy** **47**
A.1 Zdrojové kódy . . . 47

# Seznam zkratek

**ARM** Advanced RISC Machine
**DCT** Discrete Cosine Transform
**FPGA** Field Programmable Gate Array
**JFIF** JPEG File Interchange Format
**JPEG** Joint Photographic Experts Group
**RLE** Run-Length Encoding
**RGB** Red-Green-Blue
**VHDL** Very High-Speed Integrated Circuit Hardware Description Language

# 1 Úvod

Tato bakalářská práce se zabývá problematikou akcelerace ztrátové kompresní metody na hradlovém poli. V dnešním digitálním světe, zejména ve oblasti tvorby vizuálního abraze jsou za hodinu vytvářeny miliony snímků v čím dál vyšším rozlišení, pro příklad 1 snímek v RGB o velikosti 4K (4 096 × 2 160 px) bez jakékoliv komprese zabírá 26,54 MB, v případě videa o frekvenci 60 snímků za vteřinu to je 1,6 GB, to je obrovské množství dat které je potřeba ukládat. Právě proto je potřeba dělat kompresi rychle aby nedocházelo k bottleneckingu, efektivně aby se neplýtvalo paměťovými ale i výpočetními zdroji a v případě ztrátové komprese je nutné ještě dát si pozor na kritérium výsledné kvality dat. Bude zajímavé prozkoumat, jaké možnosti v řešení nastíněného problému může přinést specializovaný hardware.

Vývojová platforma je předem určená, jedná se o desku ZedBoard. Jedná se o vývojovou desku kombinující ARM procesor s FPGA programovatelnou částí. Na počátku je nutné se seznámit s jazykem VHDL pro popis a návrh hardwaru, softwarového balíku Vitis, který slouží pro syntézu a testování VHDL kódu a následné naprogramování desky. Následně se seznámíme s principy ztrátové komprese, které metody jsou používány, jaké algoritmy a postupy se používají pro redukci dat v obraze. Z poznatků vybereme postup ztrátové komprese, který je možné uskutečnit v hardwaru a navrhnout vlastní řešení ztrátového algoritmu. Nejprve v MATLABU kde je jednoduché vše otestovat, pak postup přenést do nízkoúrovňového popisu v jazyku C a nakonec vytvořit hardwarový enkodér ztrátové komprese ve VHDL. Nakonec výsledky jednotlivých implementací vůči sobě porovnat v rychlosti provedené komprese a přesnosti výpočtu.

Mým cílem a motivací pro řešení tohoto problému je vyzvat sám sebe a využít nabyté znalosti ze svého studijního oboru, abych mohl přispět k rozvoji této oblasti. Kromě toho mě velmi zajímá, jaký bude výsledek v porovnání s tradičním řešením na procesoru a jak velký bude rozdíl v rychlosti. Věřím, že tato práce bude přínosem nejen pro mě, ale i pro ostatní, kteří se chtějí dozvědět víc, nebo využít ztrátovou kompresi obrazu ve vlastní práci.

Očekává se, že výsledky práce poskytnou cenné poznatky pro další vývoj a optimalizaci ztrátové komprese obrazu na hradlových polích, přispěje k možnostem efektivnějšího využití hardwarových prostředků a zrychlí zpracování obrazových dat.

# 2 Použité prostředky

## 2.1 Xilinx Vitis 2022.2

Integrované vývojové prostředí pro vývoj softwaru a k programování na FPGA a SoC zařízeních od společnosti Xilinx. Nástroj umožňuje psát software v jazyku VHDL nebo Verilog a optimalizovat jej pro výkon na FPGA nebo SoC platformách. Vitis také nabízí možnost využití knihoven a API pro přístup k IP knihovnám, které obsahují genericky navržené bloky kódu.

Díky Vitis je možné dosáhnout rychlejšího vývoje a optimalizace programu pro FPGA a SoC platformy, což významně zvyšuje efektivitu a produktivitu vývojového procesu. Xilinx Vitis je také podporován Xilinx Runtime (XRT), což umožňuje vývojářům využít akcelerace na úrovni jádra pro zpracování dat a získání nejvyššího výkonu z platformy.

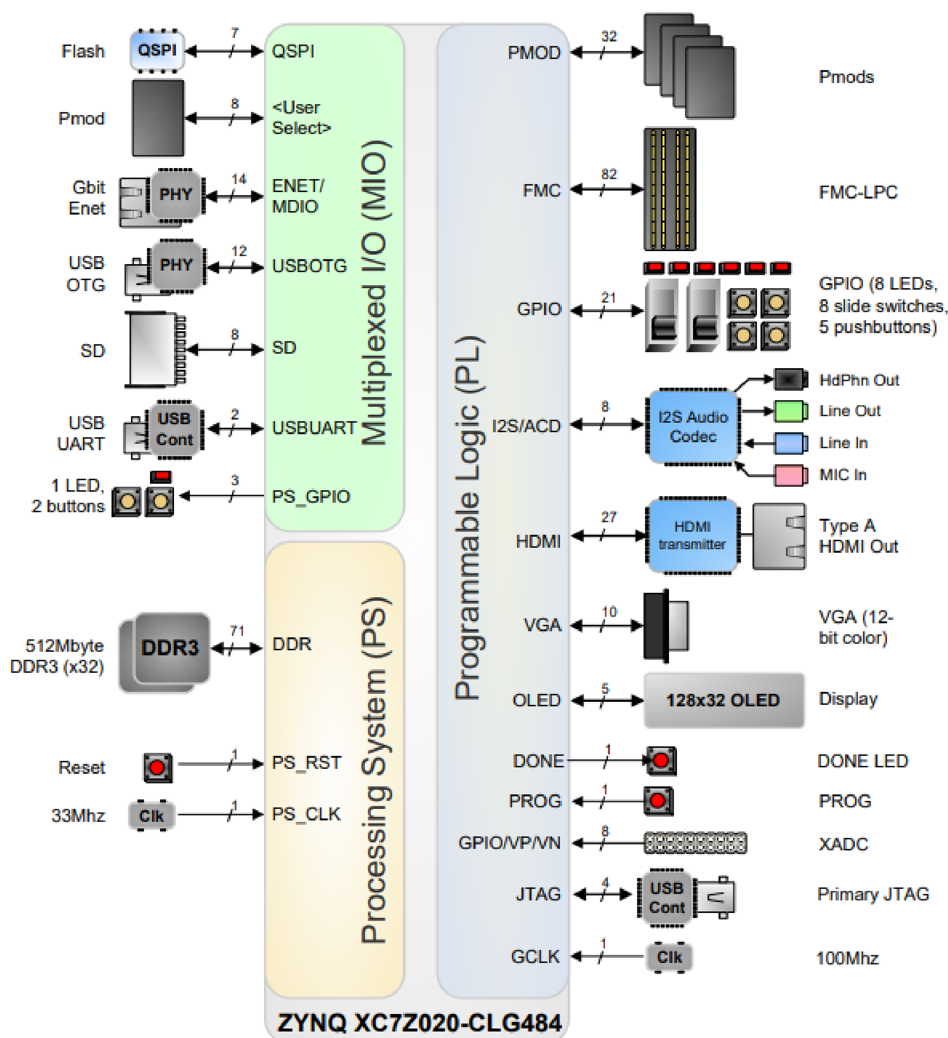
Vitis dále poskytuje nástroje pro debugování, profilování a optimalizaci kódu, což zlepšuje vývojový proces a umožňuje vývojářům lépe porozumět výkonu aplikace a případným problémům. Celkově lze říci, že Xilinx Vitis je výkonný a uživatelsky přívětivý vývojový nástroj, který umožňuje vývojářům snadno a efektivně využít potenciál FPGA a SoC platform.

## 2.2 MATLAB R2022b

Výpočetní programovací prostředí, které je specializované na numerické výpočty a zpracování signálů. Program je navržen tak, aby usnadnil vědecké a technické výpočty v oblastech jako matematika, inženýrství, fyzika, chemie a další přírodní vědy. Prostředí obsahuje mnoho funkcí a operátorů pro výpočty a práci s maticemi a vektory, což usnadňuje manipulaci s velkými datovými soubory a řešení složitých matematických rovnic. [1, 2]

## 2.3 ZedBoard

Výuková, vývojová a testovací deska postavená na procesoru Xilinx Zynq-7000 AP-SoC neboli All Programmable System-On-Chip řešení. Vyvinutá spoluprací společností Digilent, Avnet a Xilinx, kombinuje systém zpracování na dvoujádrovém procesoru ARM Cortex-A9 a 85 000 programovatelných logických buněk řady 7, které jsou integrované v jediném čipu. Deska taktéž obsahuje rozsáhlou sadu periférií, včetně 512 MB DDR3 paměti, USB, Ethernet, HDMI, VGA, audio, SD karty a dalších. Také má rozšiřující konektory, jako jsou Pmod a FMC, které umožňují připojení dalších periférií a rozšíření funkčnosti desky. Vývojáři mohou také využít integrovaný programátor pro jednoduché a rychlé programování.




Obrázek 2.1: ZedBoard Block Diagram

# 3 Ztrátová komprese obrazu

Pojmem komprese obrazu se rozumí proces při kterém se zmenšuje množství dat potřebných k reprezentaci obrazu, aby bylo možné snížit velikost souboru pro jeho uložení či další zpracování. Takový proces se děje za použití vhodných matematických algoritmů, které odstraňují redundanci v datech a vytvářejí tak komprimovanou podobu obrazu.

Oproti tomu ztrátová komprese obrazu využívá nedokonalosti lidského zraku a ukládá data s určitou ztrátou informace, díky čemuž dosahuje daleko lepších výsledků než neztrátové komprese. Lidský zrak například vnímá intenzivněji velké změny v obraze než jemné detaily nebo je daleko citlivější na změnu jasu než barvy. Toho se využívá ke zmenšení objemu barevných informací přepočítáním barevných složek na menší rozlišení (podvzorkování). Následně se data rozdělí na menší čtvercové bloky, což zjednodušuje kompresní procesy jako přeskupení nebo transformace. K transformaci obrazu jsou nejčastěji používané: diskrétní kosinová transformace 3.1 nebo diskrétní vlnková transformace 3.2 Ty se používají pro vytvoření frekvenční domény obrazu a cílenější kvantizaci – potlačení frekvencí, které nejsou informačně zajímavé ve výsledném obrazu. Na závěr se data zkomprimují některým z bezztrátových algoritmů

Výsledný obraz má menší velikost než původní, což umožňuje ukládání velkého množství dat na relativně malém prostoru, což je zvláště důležité v situacích, kdy je velikost souboru kritická, jako například při přenosu dat přes internet nebo ukládání dat na paměťové karty. Nicméně, ztrátová komprese obrazu má i své nevýhody. Jelikož se odstraňují určité informace, může při nesprávně zvolených parametrech komprese dojít k přílišnému zhoršení kvality obrazu, což je důležité vzít v potaz zejména u aplikací, kde je kvalita obsahu klíčová, jako například u lékařských obrazů nebo archivaci snímků.

## 3.1 Diskrétní kosinová transformace

Jedná se metodu, která vyjadřuje konečnou sekvenci datových bodů jako součet kosinových funkcích o různé frekvenci. Na ní je založena část kompresních algoritmů pro zpracování obrazu, kupříkladu standardy jako JPEG, WebP, HEIF pro kompresi videa pak MPEG, H.261 nebo AV1. [3]

Formálně se definuje jako lineární invertibilní funkce $f : \mathbb{R}^N \rightarrow \mathbb{R}^N$(kde $\mathbb{R}$ vyjadřuje konečnou množinu reálných čísel) tento vztah může být nahrazen čtvercovou maticí $N * N$ [4]

Standardních předpisů, pro jednorozměrnou dopřednou diskrétní kosinovou transformaci (1D FDCT) z N prvků, existuje 8 a liší se rozdílnými předpoklady podmínek symetrie. To je způsobeno tím že DCT v reálných podmínkách může pracovat pouze na konečných, diskrétních intervalech. Tak vznikají dva problémy, které pro spojitou transformaci neplatí. Za prvé je třeba určit, zda je funkce lichá nebo sudá na hranicích definičního oboru. Za druhé, určit kolem kterého bodu je funkce sudá nebo lichá.[5] Nejběžněji používaná forma DCT je označována jako DCT-II neboť je svou definicí nejméně složitá pro algoritmický výpočet a existuje značné množství modifikací. [6]. Ve zbylém textu se právě proto bude forma DCT-II označovat pouze jako DCT.

Následující forma předpisu FDCT je rozšířená o normalizaci prvků, předpis je následující:

$$F(u) = \sqrt{\frac{2}{N}} C(u) \sum_{x=0}^{N-1} f(x) \cos\left[\frac{\pi(2x+1)u}{2N}\right]$$

pro $u = 0, 1, ...N - 1$ kde

$$C(u) = \begin{cases} \frac{1}{\sqrt{2}} \text{ pro } u = 0 \\ 1 \text{ jinak} \end{cases}$$

kde funkce $f(x)$ reprezentuje hodnotu $x$-tého prvku vstupního signálu a $F(u)$ vyjadřuje DCT koeficienty $u = 0, 1, ...N-1$. Inverzní jednorozměrná diskrétní kosinová transformace (1D IDCT) k 1D FDCT je dle stejných pravidel předpisu definována:

$$f(x) = \sqrt{\frac{2}{N}} \sum_{x=0}^{N-1} C(u) F(u) \cos\left[\frac{\pi(2x+1)u}{2N}\right]$$

pro $x = 0, 1, ..., N - 1$.

Pro zpracování dvourozměrných dat je však nutné rozšířit definici o druhý rozměr. DCT je oddělitelná lineární transformace; to znamená, že dvourozměrná transformace je ekvivalentní jednorozměrné DCT spočtené podél jedné dimenze následované jednorozměrnou DCT provedou přes druhou dimenzi. Je proto irelevantní zda výpočet proběhne nejprve horizontálně a pak vertikálně, nebo vertikálně a následně horizontálně, obě kombinace vyústí ve stejný výsledek. [4, 7]

Předpis dvourozměrné dopředné diskrétní kosinové transformace (2D FDCT) je následující [4]:

$$F(u,v) = \frac{2}{\sqrt{MN}} C(u)C(v) \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} f(x,y) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2M}\right]$$

pro $u = 0, 1, ...N-1$ a $v = 0, 1, ...M-1$ kde

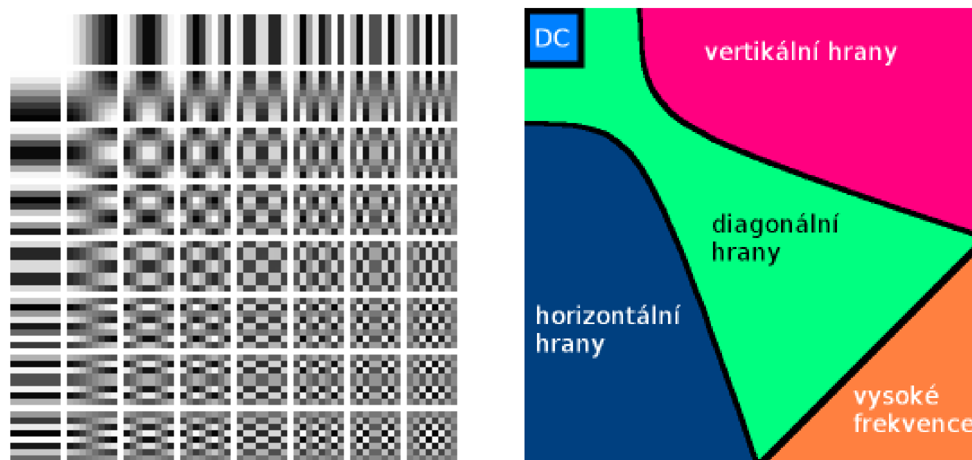
$$C(k) = \begin{cases} \frac{1}{\sqrt{2}} \text{ pro } k = 0 \\ 1 \text{ jinak} \end{cases}$$

pro $k$ reprezentující $u, v$

Dvourozměrná inverzní diskrétní kosinová transformace( 2D IDCT) je dle stejných pravidel předpisu definována[4]:

$$f(x,y) = \frac{2}{\sqrt{MN}} \sum_{x=0}^{N-1} \sum_{y=0}^{M-1} C(u)C(v)F(u,v) \cos\left[\frac{\pi(2x+1)u}{2N}\right] \cos\left[\frac{\pi(2y+1)v}{2M}\right]$$

pro $x = 0, 1, ...N-1$ a $y = 0, 1, ...M-1$




(a) Frekvenční rozložení po 2D DCT v bloku.[8]

(b) Zastoupení jednotlivých typů obrazových struktur v bloku.[9]

Obrázek 3.1: Frekvenční

## 3.2 Diskrétní vlnková transformace

Vlnková transformace je integrální transformace, která umožňuje časově-frekvenční popis signálu. Je proto možné rozklad signálu na nezávislé bloky. Předpis spojité vlnkové transformace pro mateřskou vlnku označovanou $\psi(t)$

$$\psi_{a,b}(t) = \frac{1}{\sqrt{a}}\psi\left(\frac{t-b}{a}\right) \tag{3.1}$$

Než je vůbec možné definovat, co je diskrétní vlnková transformace, je nezbytné definovat vlnky diskrétních hodnot z hlediska dilatace a translace parametrů $a$ a $b$, které nebudou kontinuální. Je mnoho způsobů jak diskretizovat $a$ a $b$ a podle nich reprezentovat diskrétní vlnky. Nejvyužívanější způsob je následující:

$$a = a_0^m, b = nb_0a_0^m \tag{3.2}$$

kde $m$ a $n$ jsou z oboru $\mathbb{Z}$. Substitucí $a$ a $b$ podle 3.2 do předpisu 3.1 , bude diskrétní vlnková transformace reprezentována 3.3

$$\psi_{m,n}(t) = a_0^{-m/2}\psi(a_0^{-m}t - nb_0) \tag{3.3}$$

Je mnoho možností hodnot, které lze dosadit za $a_0$ a $b_0$. Nejtypičtěji používané hodnoty jsou: $a_0 = 2$ a $b_0 = 1$; proto $a = 2^m$ a $b = n2^m$. To odpovídá vzorkování (diskretizaci) $a$ a $b$ takovým způsobem, že po sobě jdoucí diskrétní hodnoty $a$ a $b$ stejně jako intervaly vzorkování se liší druhým faktorem. Tento způsob vzorkování je všeobecně známý jako dyadické vzorkování a odpovídající rozklad signálů se nazývá dyadický rozklad. Pomocí těchto hodnot může reprezentovat diskrétní vlnky jako v 3.4, který tvoří rodinu ortonormálních základních funkcí. [4]

$$\psi_{m,n}(t) = 2^{-m/2}\psi(2^{-m}t - n) \tag{3.4}$$

Obecně jsou vlnkové koeficienty pro funkci $f(t)$ dány vztahem,

$$c_{m,n}(f) = a_0^{-m/2}\int f(t)\psi(a_0^{-m}t - nb_0)dt \tag{3.5}$$

a proto pro dyadický rozklad lze vlnkové koeficienty odvodit podle toho:

$$c_{m,n}(f) = 2^{-m/2}\int f(t)\psi(2^{-m}t - n)dt \tag{3.6}$$

To nám umožňuje rekonstruovat signál $f(t)$ z diskrétních vlnkových koeficientů:

$$f(t) = \sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty} c_{m,n}(f)\psi_{m,n}(t) \tag{3.7}$$

Transformace zobrazená v 3.5 se nazývá vlnková řada, která je analogická s Fourierovou řadou, protože vstupní funkce $f(t)$ je spojitá funkce, zatímco transformační koeficienty jsou diskrétní. Je označovaná jako diskrétní časová vlnková transformace (DTWT). Pro aplikace zpracování digitálního signálu nebo obrazu prováděné

digitálním počítačem musí mít vstupní signál $f(t)$ diskrétní povahu kvůli digitálnímu vzorkování původních dat, která jsou reprezentována konečným počtem bitů. Vstupní funkce $f(t)$ potřebuje mít diskrétní povahu, protože jde o digitální vzorkování původních data ta jsou reprezentována konečným počtem bitů. Když je vstupní funkce $f(t)$ stejně jako vlnkové parametry $a$ a $b$ reprezentována v diskrétní formě, transformace se běžně označuje jako diskrétní vlnková transformace (DWT) signálu $f(t)$.[4]

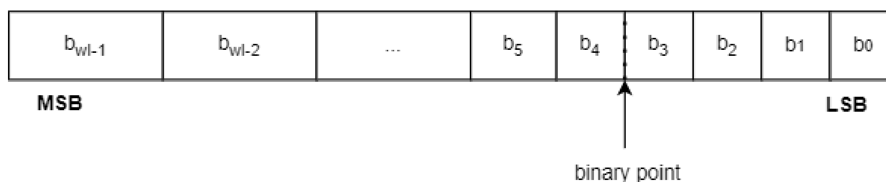
Diskrétní vlnková transformace (DWT) se stala velmi všestranným nástrojem pro zpracování signálu poté, co Mallat[10] navrhl multirozkladnou reprezentaci signálů založenou na vlnkové dekompozici. Výhoda DWT spočívá oproti Fourierově transformaci v tom, že provádí multirozkladnou analýzu signálů s lokalizací. Výsledkem DWT je rozklad digitálního signálu do různých dílčích pásem tak, že dílčí pásma nižší frekvence mají jemnější frekvenční rozlišení a hrubší časové rozlišení ve srovnání s dílčími pásmy vyšší frekvence. DWT se stále více používá pro kompresi obrazu kvůli skutečnosti, že DWT podporuje funkce jako progresivní přenos obrazu (podle kvality, rozlišení), kódování oblasti zájmu a snadnou manipulaci s komprimovaným obrazem. DWT je základem pro kompresi obrazu formátu JPEG2000.[4]

## 3.3 Datové typy s řádovou čárkou

V digitálním hardwaru jsou čísla ukládána jako binární slovo. Binární slovo je sekvence bitů (1 a 0) s pevně danou délkou (povětšinou o velikostech $2^N$). Způsob, jak hardwarové komponenty nebo softwarové funkce interpretují bitovou sekvenci je definován datovým typem. Binární čísla jsou reprezentována jako datové typy s pevnou nebo pohyblivou desetinnou čárkou.

### 3.3.1 Datové typy s pevnou řádovou čárkou

Datový typ s pevnou řádovou/desetinou čárkou je charakterizován délkou bitového slova, velikostí desetinné části a zda je znaménkový nebo nikoliv. Poloha binární čárky slouží ke správné interpretaci a škálování hodnot s pevnou desetinou čárkou.[11]




Obrázek 3.2: Příklad binární reprezentace zobecněného čísla s pevnou řádovou čárkou (se znaménkem nebo bez)

[11]

kde

- $b_i$ je i-tý bit,
- $wl$ je délka bitového slova,
- $b_{wl-1}$ je nejvýznamnější/nejvyšší bit (MSB - **m**ost **s**ignificant **b**it),
- $b_0$ je nejméně-významný/nejnižší bit (LSB - **l**east **s**ignificant **b**it),
- Binární bod je zobrazen čtyři místa doleva od LSB. To v uvedeném příkladu znamená, že číslo má čtyři desetinné bity.

Jak už bylo zmíněno, datové typy s pevnou řádovou čárkou mohou být se znaménkem nebo bez znaménka. Zda je hodnota s pevnou řádovou čárkou se znaménkem nebo ne, není obvykle v binárním slově explicitně určeno; Místo toho je informace o znaménku implicitně definována v rámci hardwarové/softwarové architektury. Obvykle je zvolen jeden z následujících způsobů[11]:

- Znaménkový kód – první bit binárního slova je vždy vyhrazený znaménku, zatímco zbývající bity slova kódují velikost čísla. Tato reprezentace není výhodná pro matematické operace, je nutné definovat odlišné algoritmy pro sčítání a odčítání a také existuje ”kladná” a ”záporná” reprezentace nuly.

  *Příklad: číslo 7 je v 8 bitové reprezentaci 0000 0111, pak -7 se vyjádří kódem 1000 0111*

- Jedničkový doplněk – Kladná čísla se vyjadřují klasickým způsobem naopak záporná čísla se vyjadřují binární negací. To znamená, že všechny 0 jsou převráceny na 1 a všechny 1 jsou převráceny na 0. V jedničkovém doplňku existují dva způsoby, jak reprezentovat nulu. Binární slovo všech 0 představuje ”kladnou” nulu, zatímco binární slovo všech 1 představuje ”zápornou” nulu.

  *Příklad: pokud 0000 0110 je binární vyjádření čísla 6 , pak* $-6$ *se vyjádří kódem 1111 1001.*

- Dvojkový doplněk – Záporná čísla se vytváří bitovou inverzí (převod na jedničkový doplněk) a následným binárním přičtením jedničky.

  *Příklad: pokud 00001101 je binární vyjádření čísla 13, pak* $-13$ *se vypočte jako NOT(0000 1101) + 1 = 1111 0011.*

### 3.3.2 Datové typy s plovoucí řádovou čárkou

Čísla s pevnou řádovou čárkou mají jedno velké omezení. Současně není možné reprezentovat velmi velká a velmi malá čísla s rozumnou velikosti bitového slova. Toto problematiku lze překlenout použitím vědecké notace. S vědeckou notací lze umístit desetinou čárku (binární bod) dynamicky na kteroukoliv pozici. Tak je možné reprezentovat rozsah velkých a malých čísel zároveň vyjádřených několika číslicemi.

Binární číslo s plovoucí desetinnou čárkou je vědeckým zápisem reprezentováno jako

$$f2^e$$

- kde $f$ je zlomek (nebo mantisa),
- 2 je radix nebo základ (v tomto případě binární, může být i desítkový, šestnáctkový, dle vybrané soustavy)
- $e$ je exponent ze základu.

Základ je vždy kladné číslo, zatímco $f$ a $e$ mohou být jak kladná tak záporná čísla. Nejčastěji používaná reprezentace formátu je definována standardem IEEE 754.

Při provádění aritmetických operací s pohyblivou řádovou čárkou na hardwaru, je nutné vzít v úvahu, že znaménko, exponent a zlomek jsou zakódovány ve stejném binárním slově. Výsledkem takových operací jsou složité logické obvody ve srovnání s obvody pro binární operace s pevnou řádovou čárkou. [12]

## 3.4 JFIF/JPEG

Formát JFIF (**J**PEG **F**ile **I**nterchange **F**ormat), nesprávně, ale častěji, nazýván JPEG dle konsorcia Joint Photographic Experts Group. Formát vytvořený v roce 1992 (poslední verze 1994), jako standard pro úsporné kódování fotografických obrazových dat.[13] Cílem standardu je aplikační variabilita komprese souvislých tónovaných obraz (Continuous-tone still images), se velikosti obrazu do 65,535 px libovolného barevného prostoru a rozsáhlou možností pro implementací. [4]

Kroky pro kompresi do JFIF jsou následující:

1. Převedení obrazových dat do barevného modelu YCbCr
2. Podvzorkování barvonosných složek
3. 2D FDCT
4. Kvantizace
5. Cik-cak vyčítání
6. RLE kódování
7. Huffmanovo případně Aritmetické kódování

Míra zachování detailů je volitelná a nastavuje se koeficientem kvality v rozsahu 1 – 100. Vyšší hodnoty znamenají menší zkreslení a větší výsledný soubor, menší hodnoty se projeví menším objemem dat za cenu větší ztráty kvality. Parametry JPEG komprese ovlivňují pouze kvalitu obrazu a velikost souboru, velikost původního obrázku (počet obrazových bodů) zůstává nezměněn.

### 3.4.1 Převedení obrazových dat z RGB do YCbCr

Převedení z RGB do YCbCr je využíváno pro oddělení informace o jasové složce od složek barevných

Převod se provádí pomocí lineární transformace, která vypočítá hodnoty Y (luminance), Cb (chrominance blue) a Cr (chrominance red) na základě hodnot složek R, G a B. Konkrétně jsou vztahy pro výpočet YCbCr hodnot následující:

$$Y = 0.299R + 0.587G + 0.114B$$

$$Cb = -0.1687R - 0.3313G + 0.5B + 128$$

$$Cr = 0.5R - 0.4187G - 0.0816B + 128$$

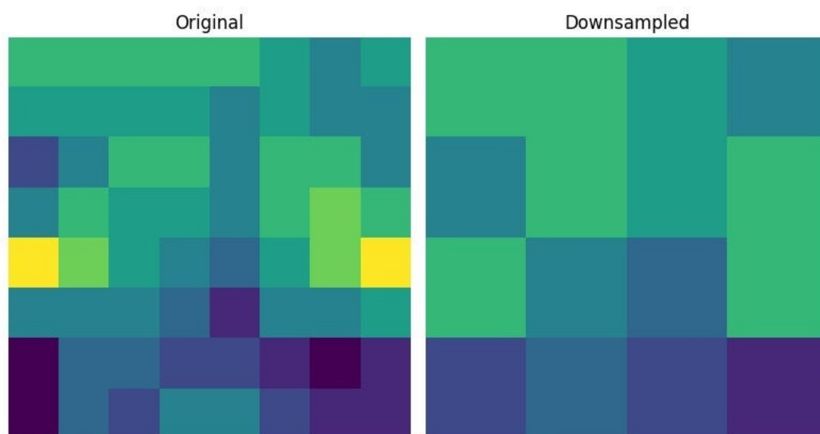
Koeficienty v těchto vztazích jsou zvoleny tak, aby co nejvíce odpovídaly vnímání lidského oka. Hodnota Y představuje jasovou složku obrazu, zatímco Cb a Cr reprezentují chrominanční složky, které určují rozdíly v barevnosti. Výsledné YCbCr hodnoty jsou obvykle reprezentovány na 8 bitů, což umožňuje 256 různých úrovní pro každou složku.

### 3.4.2 Podvzorkování barvonosných složek

Podvzorkování barvonosných složek snižuje množství dat, nutných pro přenos a uložení barevných informací. Vzhledem k tomu, že lidské oko je méně citlivé na změny v barevných složkách než na změny v jasové složce, lze redukovat počet vzorků pro chrominanční složky oproti jasové složce.

V algoritmu JPEG se nejčastěji používá podvzorkování 4:2:0, což znamená, že každý blok 8×8 pixelů je rozdělen na 4 podsoubory 2 bloky pro jasovou složku a jeden podsoubor pro každou chrominanční složku. To znamená, že pro každý blok 8×8 pixelů pro chrominanční složky využito pouze 16 vzorků, zatímco pro jasovou složku je použito všech 64 vzorků.

Podvzorkování umožňuje dosáhnout významného snížení velikosti datového toku při zachování přijatelné kvality obrazu. Nicméně je důležité mít na paměti, že při dekompresi je nutné provést inverzní operaci, tj. interpolaci vzorků chrominančních složek, aby bylo možné obnovit plnou barevnou informaci. Tento proces může vést k mírné ztrátě kvality obrazu, zejména při vyšší míře komprese.




Obrázek 3.3: Grafické znázornění podvzorkování barvonosných složek [14]

### 3.4.3 Kvantizace

Stěžejní část ztrátové komprese JPEG, blok vypočtených koeficientů z 2D DCT jsou prvek po prvku vyděleny koeficienty v kvantizační matici a zaokrouhleny na celá čísla.

Míra zachování detailů je volitelná a nastavuje se koeficientem kvality v rozsahu 1-100. Vyšší hodnoty znamenají menší zkreslení a větší výsledný soubor, menší hodnoty se projeví nižší kvalitou a menší velikostí souboru při stejném počtu obrazových bodů.

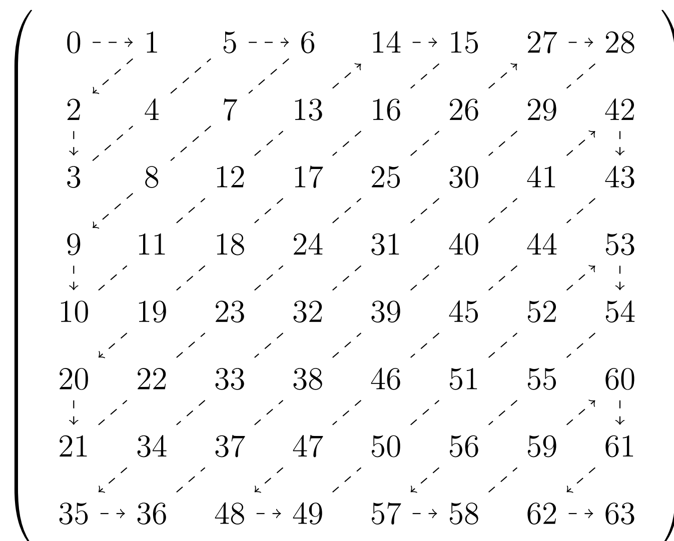
Kvantizační tabulky jsou vytvořeny tak, aby obsahovaly nízké hodnoty u nízkofrekvenčních a stejnosměrných složek a naopak vysoké hodnoty pro vysokofrekvenční.

$$
\begin{bmatrix}
16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\
12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\
14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\
14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\
18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\
24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\
49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\
72 & 92 & 95 & 98 & 112 & 100 & 103 & 99
\end{bmatrix}
$$

Obrázek 3.4: Jasová kvantizační matice s koeficientem zachování kvality 50

### 3.4.4 Cik-cak

Protože kvantování dvourozměrné matice vede k významné koncentraci energie v nízkých frekvenčních složkách obrazu, a hodnoty vyšších frekvenčních složek mají tendenci být nulové. Je výhodné přeuspořádat data do vektoru ve specifickém pořadí, střídavého procházení dle diagonál, nazývaného cik-cak (zig-zag) a tím připravit data pro následující krok kterým je RLE kódování. [4] Vzor je následující:




Obrázek 3.5: Matice hodnot se vzorem průchodu cik-cak

### 3.4.5 RLE kódování

Celým názvem **R**un **--L**ength **E**ncoding je neztrátová metoda kódování dat, používaná pro opakujících se sérií hodnot v datovém řetězci. Princip metody spočívá v nahrazení opakujících se symbolů na uspořádanou dvojici symbolu a čísla, které udává, kolikrát se symbol opakuje. RLE je zejména vhodné pro kompresi bitmapových obrázků, kde jsou často vyskytující se barvy uloženy jako opakující se série pixelů. Jedná se jednoduchý algoritmus pro bezztrátovou kompresi dat. Používá se k redukci redundance v datech nahrazením opakujících se bloků stejných symbolů

kratšími zápisem skládajícím se ze symbolu a počtu výskytů. Je zejména výhodné při kompresi binárních obrazů, zde můžou nastat jen 2 možnosti (bílá/černá), právě proto budou sousední pixely spíše stejné.

RLE algoritmus při procházení datového proudu funguje následovně:

1. Detekuje symbol a inkrementuje čítač dokud nepřeteče nebo se symbol nezmění,
2. Vrátí dvojici [symbol, počet symbolů] pokračuje dalším symbolem.

Například, řetězec "AAABBBCCC" může být zakódován jako "A3B3C3", což znamená tři A, tři B a tři C. To značně snižuje délku dat, která musí být uložena nebo přenesena, což může být užitečné zejména u velkých opakujících se objemů dat.

Přestože je RLE kódování efektivní a jednoduché na implementaci, existují situace, ve kterých může vést k nežádoucím výsledkům. Jedním z takových rizik je vznik dvojnásobné délky dat. Toto se může stát, pokud v datovém řetězci nejsou žádné opakující se symboly nebo pokud se symboly opakují velmi vzácné. V těchto případech je každý symbol kódován jako dvojice symbolu, což může způsobit zvýšení délky datového řetězce až na dvojnásobek.

Při použití RLE je tedy důležité vzít v úvahu povahu dat a předpokládanou frekvenci výskytu opakujících se symbolů. Pokud jsou v datech časté opakování, RLE může poskytnout významné úspory. Nicméně, pokud jsou symboly vzácné nebo se neopakují, RLE nemusí být vhodnou metodou komprese a může vést k neefektivnímu využití paměti nebo přenosového kanálu.

### 3.4.6 Huffmanovo kódování

Huffmanovo kódování bylo vyvinuto Davidem Huffmanem roku 1952 v rámci školního projektu na MIT. Jedná se o metodu bezztrátového kódování dat, používanou k efektivnímu snížení velikosti datových souborů. Kódování využívá statistické informace o frekvenci výskytu jednotlivých symbolů v datovém souboru. Na základě frekvence je sestavena stromové struktura. Kódová slova pro jednotlivé symboly jsou následně vytvořena jako cesta v Huffmanově stromu od kořene ke konkrétnímu symbolu.

Huffmanovo kódování využívá dva základní principy, které platí pro optimální kódování prefixových kódů.

1. Symboly s větší pravděpodobností výskytu mají kratší kódová slova než symboly s menší pravděpodobností.
2. Dva symboly s nejmenší pravděpodobností mají stejně dlouhá kódová slova a ta se liší pouze na pozici nejméně významného bitu.

Kód pro každý symbol je pak uložen v tabulce kódů. Kódování a dekódování pak probíhá tak, že se postupně čtou vstupní symboly a kódují se podle tabulky

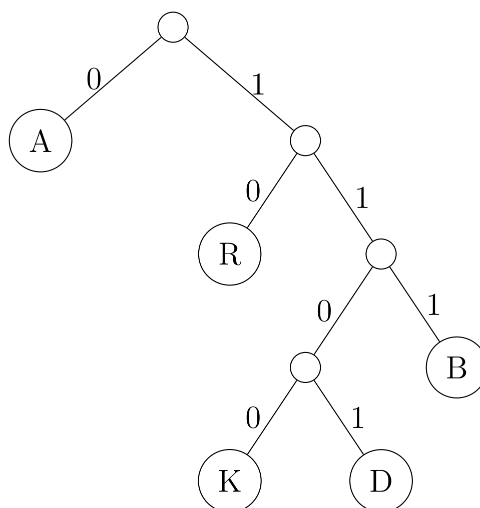
kódů. Při dekódování se pak postupně čtou kódová slova a dekódují se na základě Huffmanova stromu.

Postup pro generování Huffmanových kódů lze shrnout následovně:

1. Určete pravděpodobnost výskytu každého symbolu v abecedě vstupu.
2. Uspořádejte symboly podle neklesající pravděpodobnosti.
3. Kombinujte dva symboly s nejnižšími pravděpodobnostmi tak, aby vytvořily nový symbol s pravděpodobností rovnající se součtu pravděpodobností původních symbolů.
4. Nahraďte dva symboly novým symbolem v uspořádaném seznamu a opakujte kroky 3 a 4, dokud nejsou všechny symboly sloučeny do jediného symbolu.
5. Přiřaďte každému symbolu binární kód na základě cesty, kterou je nutné projet v binárním stromě, aby se dostal k tomuto symbolu. Přiřaďte 0 každému levému větvení a 1 každému pravému větvení.
6. Výsledné binární kódy jsou Huffmanovy kódy.

Příklad pro znakový řetězec **ABRAKADABRA**

| **Symbol** | Četnost | **Huffmanův kód** | **Délka kódu** |
|---|---|---|---|
| A | 0.46 | 0 | 1 |
| R | 0.18 | 1 | 1 |
| B | 0.18 | 00 | 2 |
| K | 0.09 | 01 | 2 |
| D | 0.09 | 10 | 2 |



# 4 Návrh řešení

Jako optimální volba ztrátového formátu vhodného pro hardwarovou akceleraci byl vybrán formát JFIF, nebo alespoň přiblížení se postup podle kterého je obraz dle předpisu JFIF kódován. Důvodů pro výběr formátu JFIF je hned několik:

- účinný a snadno škálovatelný poměr komprese obrazu,
- jedná se o jeden z nejpoužívanějších formátů, je dobře dokumentován, a zároveň podporován prakticky jakýmkoliv zařízením nebo platformou,
- postup komprese je vhodně přenositelný do hardwaru.

## 4.1 DCT

Před výpočtem DCT bloku 8×8 jsou jeho hodnoty posunuty z kladného rozsahu. U 8bitového obrázku spadá každý záznam v původním bloku do rozsahu $[0, 255]$. Střed rozsahu (v tomto případě hodnota 128) se odečte od každé položky, aby se vytvořil rozsah dat, který je vycentrován na nulu, takže upravený rozsah je[**-128 127**]. Tento krok snižuje požadavky na dynamický rozsah ve fázi kvantizace.[13]

Pokud DCT 3.1 reprezentujeme jako matici pro $N = 8$, $k = (0, 1, ..., N-1)$, a $c_k = \cos(\frac{k\pi}{16})$ kde

$$\alpha_k = \begin{cases} \frac{1}{\sqrt{N}} \text{ pro } k = 0 \\ \sqrt{\frac{2}{N}} \text{ jinak} \end{cases}$$

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \alpha_k \begin{bmatrix} c_0 & c_0 & c_0 & c_0 & c_0 & c_0 & c_0 & c_0 \\ c_1 & c_3 & c_5 & c_7 & -c_7 & -c_5 & -c_3 & -c_1 \\ c_2 & c_6 & -c_6 & -c_2 & -c_2 & -c_6 & c_6 & c_2 \\ c_3 & -c_7 & -c_1 & -c_5 & c_5 & c_1 & c_7 & -c_3 \\ c_4 & -c_4 & -c_4 & c_4 & c_4 & -c_4 & -c_4 & c_4 \\ c_5 & -c_1 & c_7 & c_3 & -c_3 & -c_7 & c_1 & -c_5 \\ c_6 & -c_2 & c_2 & -c_6 & -c_6 & c_2 & -c_2 & c_6 \\ c_7 & -c_5 & c_3 & -c_1 & c_1 & -c_3 & c_5 & -c_7 \end{bmatrix} * \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} \quad (4.1)$$

Ze 4.1 lze vidět, že pro výpočet jednoho vektoru transformovaných hodnot $y$, je nutno provést 64 násobení a 56 součtů. V případě 8 vektorů vstupních hodnot počet operací nutných k vyčíslení transformovaných hodnot, pro ně charakteristických to činí 512 násobení a 448 sčítání.[15] Počet provedených násobení lze snížit na 32 díky

symetrii matice koeficientů $c_k$ a tím zároveň počet sčítání/odčítání klesne na 32. Upravená forma je následující:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \alpha_k \begin{bmatrix} c_0 & c_0 & c_0 & c_0 & 0 & 0 & 0 & 0 \\ c_2 & c_6 & -c_6 & -c_2 & 0 & 0 & 0 & 0 \\ c_4 & -c_4 & -c_4 & c_4 & 0 & 0 & 0 & 0 \\ c_6 & -c_2 & c_2 & -c_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -c_1 & c_3 & c_5 & c_7 \\ 0 & 0 & 0 & 0 & c_3 & -c_7 & -c_1 & -c_5 \\ 0 & 0 & 0 & 0 & c_5 & -c_1 & c_7 & c_3 \\ 0 & 0 & 0 & 0 & c_7 & -c_5 & c_3 & -c_1 \end{bmatrix} * \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \\ x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix} \tag{4.2}$$

Existují úpravy, které počet operací sníží ještě více. Takové úpravy spadají do kategorie rychlých výpočetních metod DCT. Mezi známe a hojně užívané patří Chenův algoritmus [16] , který redukuje počet operací na 16 násobení a 26 sčítání/odčítání, a Loefflerův algoritmus [17], který redukuje počet operací na 11 násobení a 29 sčítáni/odčítání.

Upravený vztah založený na úpravách dle Lofflera vypadá následovně [17, 6] kde $c_k = \sqrt{2}\cos(\frac{k\pi}{16})$ *pro* $k = 1, 2, ...N - 1$ kde $\alpha_k = \frac{1}{\sqrt{N}}$

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \alpha_k \begin{bmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ c_2 & c_6 & -c_6 & -c_2 & 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 & 0 & 0 & 0 & 0 \\ c_6 & -c_2 & c_2 & -c_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -c_1 & c_3 & c_5 & c_7 \\ 0 & 0 & 0 & 0 & c_3 & -c_7 & -c_1 & -c_5 \\ 0 & 0 & 0 & 0 & c_5 & -c_1 & c_7 & c_3 \\ 0 & 0 & 0 & 0 & c_7 & -c_5 & c_3 & -c_1 \end{bmatrix} * \begin{bmatrix} x_0 + x_7 \\ x_1 + x_6 \\ x_2 + x_5 \\ x_3 + x_4 \\ x_0 - x_7 \\ x_1 - x_6 \\ x_2 - x_5 \\ x_3 - x_4 \end{bmatrix} \tag{4.3}$$

Tento návrh obsahuje 28 součtů a 20 násobení a je nutné ho vykonávat na 4 fáze, které nemohou být z důvodu datové závislostí vykonávány paralelně. Loeffler postup v těchto fázích pro snížení multiplikativní složitosti provádí rotace vedoucí k snížení počtu násobení ku sčítání, což je pro hardwarový návrh zbytečné dle [6] proto bude pro implementaci postačující návrh 4.3 . Výsledný tok signálu bude odpovídat 4.4.

### 4.1.1 Kvantizace

V paměti bude uložená kvantizační tabulka 3.4 vstupní blok 8×8 dat z 2D DCT bude prvek po prvku vynásoben $\frac{1}{Q_n}$ a zaokrouhlen k nejbližšímu celému číslu.

### 4.1.2 Cik-cak

Pro uspořádání dat do cikcak vzoru dle 3.4.4 lze vytvořit tabulku indexů 4.4 a dle ní jednoduše přeuspořádat jednotlivé prvky v datovém bloku 8×8 přeuspořádat.

$$\begin{bmatrix} 0 & 1 & 5 & 6 & 14 & 15 & 27 & 28 \\ 3 & 2 & 4 & 7 & 13 & 16 & 26 & 29 \\ 4 & 3 & 8 & 12 & 17 & 25 & 30 & 41 \\ 5 & 9 & 11 & 18 & 24 & 31 & 40 & 44 \\ 6 & 10 & 19 & 23 & 32 & 39 & 45 & 52 \\ 7 & 20 & 22 & 33 & 38 & 46 & 51 & 55 \\ 8 & 21 & 34 & 37 & 47 & 50 & 56 & 59 \\ 9 & 35 & 36 & 48 & 49 & 57 & 58 & 62 \end{bmatrix} \tag{4.4}$$

### 4.1.3 RLE

Kódování cik-cak frekvence bude upraveno oproti naivní implementaci RLE 3.4.5. Sestavení kódového slova je následující:

- Jako **x** označíme nenulovou hodnotu z cik-cak vektoru.
- DÉLKABĚHU je počet nul, který předchází nenulové hodnotě.
- VELIKOST je počet bitů potřebných k reprezentaci x.
- HODNOTA je bitová reprezentace x.

Kódování vytváří ustálenou dvojici 8bitových symbolů.

| Symbol 1 | Symbol 2 |
|---|---|
| (DÉLKABĚHU, VELIKOST) | (HODNOTA) |

Tabulka 4.1: RLE kódové slovo

DÉLKABĚHU A VELIKOST sdílí stejný bajt, což znamená rozdělení na 4 informační bity. Vyšší bity značí počet nul, zatímco nižší bity označují nezbytný počet bitů pro uložení hodnoty x. To má okamžitý dopad na to, že Symbol 1 může uchovávat informace pouze o prvních 15 nulách předcházejících nenulové hodnotě. Nicméně, jsou definována 2 speciální kódová slova, řešící krajní stavy. Jedno je pro předčasné ukončení sekvence, kdy zbývající koeficienty jsou nulové (nazýváno "End-of-Block" nebo "EOB"), a další, když běh nul přesáhne počet 15, než dosáhne nenulové hodnoty. V případě, když je před danou nenulovou hodnotou nalezeno 16 nul, je Symbol 1 "speciálně" zakódován jako: (15, 0)(0). Celkový proces pak pokračuje, dokud se nedosáhne "EOB" – označeno jako (0, 0).[13]

## 4.2 Implementace kódu v jazyku C pro vnořené jádro Zynq

### 4.2.1 DCT

Vstupem do funkce je blok 8×8 obrazových dat, následně jsou deklarovány proměnné pro uložení výsledků transformace, následuje smyčka pro každý sloupec matice $Mt$, v každé iteraci se provede výpočet DCT, normalizace a uložení na deklarované pozice výstupní matice. Po dokončení smyčky funkce vrací výslednou matici.

```
float **fdct(float **M){
    float x0, x1, x2, x3, x4, x5, x6, x7;
    float x0x3, x1x2, x1_x2, x0_x3;
    float **Y = alloc_float_matrix(8, 8);
    const float norm = sqrt(1.0 / 8.0);
    const float c[8] = {1, cos(PI / 16) * sqrt(2), cos(PI / 8) * sqrt(2), cos(3 * PI / 16) * sqrt(2), cos(PI / 4) * sqrt(2), cos(5 * PI / 16) * sqrt(2), cos(3 * PI / 8) * sqrt(2), cos(7 * PI / 16) * sqrt(2)};

    for (int i = 0; i < 8; i++){
        x0 = M[0][i] + M[7][i];
        x1 = M[1][i] + M[6][i];
        x2 = M[2][i] + M[5][i];
        x3 = M[3][i] + M[4][i];
        x4 = M[0][i] - M[7][i];
        x5 = M[1][i] - M[6][i];
        x6 = M[2][i] - M[5][i];
        x7 = M[3][i] - M[4][i];

        x0x3 = x0 + x3;
        x1x2 = x1 + x2;
        x1_x2 = x1 - x2;
        x0_x3 = x0 - x3;

        Y[0][i] = (x0x3 + x1x2) * norm;
        Y[1][i] = (c[1] * x4 + c[3] * x5 + c[5] * x6 + c[7] * x7) * norm;
        Y[2][i] = (c[2] * x0_x3 + c[6] * x1_x2) * norm;
        Y[3][i] = (c[3] * x4 - c[7] * x5 - c[1] * x6 - c[5] * x7) * norm;
        Y[4][i] = (x0x3 - x1x2) * norm;
        Y[5][i] = (c[5] * x4 - c[1] * x5 + c[7] * x6 + c[3] * x7) * norm;
        Y[6][i] = (-c[2] * x1_x2 + c[6] * x0_x3) * norm;
        Y[7][i] = (c[7] * x4 - c[5] * x5 + c[3] * x6 - c[1] * x7) * norm;
    }
    return Y;
}
```

Listing 4.1: DCT

### 4.2.2 Transpozice

Pro výpočet 2D DCT je nutné data po prvním průchodu transformovat . Vstupem do funkce je proto matice která bude transponována.

```
    void transpose_matrix(float **A, int matrix_size)
{
    float temp;
    for (int i = 0; i < matrix_size - 1; i++)
    {
        for (int j = i + 1; j < matrix_size; j++)
        {
            temp = A[i][j];
            A[i][j] = A[j][i];
            A[j][i] = temp;
        }
    }
}
```

Listing 4.2: Transpozice jednotlivých prvků

### 4.2.3 Kvantizace

Vstupem do funkce je matice a kvantizační tabulka, hodnoty v matici jsou prvek po prvku vyděleny složkou kvantizační matice a zaokrouhleny k nejbližšímu celému číslu.

```
void quantization(float **M, int (*q_table)[8]){
    for (int i = 0; i < 8; i++){
        for (int j = 0; j < 8; j++){
            M[i][j] = round((M[i][j] / q_table[i][j]));
        }
    }
}
```

Listing 4.3: Kvantizace jednotlivých prvků

### 4.2.4 Cik-cak

Vstupem do funkce je matice kvantizovaných hodnot. Ty budou přeuspořádány do pole podle indexační cik-cak matice.

```
int *zig_zag(int **M, int size){
    const int coeficients[8][8] = {{0, 1, 5, 6, 14, 15, 27, 28},
                                  {2, 4, 7, 13, 16, 26, 29, 42},
                                  {3, 8, 12, 17, 25, 30, 41, 43},
                                  {9, 11, 18, 24, 31, 40, 44, 53},
                                  {10, 19, 23, 32, 39, 45, 52, 54},
                                  {20, 22, 33, 38, 46, 51, 55, 60},
                                  {21, 34, 37, 47, 50, 56, 59, 61},
                                  {35, 36, 48, 49, 57, 58, 62, 63}};

    int *zz = alloc_int_array(size * size);
```

```
    for (int i = 0; i < size; i++){
        for (int j = 0; j < size; j++){
            zz[coeficients[i][j]] = M[i][j];
        }
    }
    return zz;
}
```

### 4.2.5 RLE

Vstupem do funkce je pole po cic-cak reindexaci. Pak je alokována dvojnásobně velká paměť pro RLE v případě že by se ani jedna hodnota neopakovala. Poté už zbytek operací probíhá v cyklu, zjišťuje se počet nul, velikost nenulové hodnoty a amplituda nenulové hodnoty, z této kombinace je vytvořena uspořádaná dvojice hodnot dle předpisu a uložena do výstupního pole, výstupní pole je před návratovou hodnotou funkce realokováno na odpovídající velikost.

```
    uint8_t *rle = malloc(2*input_size * sizeof(uint8_t));
    int count = 0, index = 0;
    int zero_count = 0;
    uint8_t runlength, size;
    uint8_t amplitude;

    while (index < input_size){
        if (zigzag[index] == 0){
            zero_count++;
            index++;
        }else{
            if (zero_count > 15){
                runlength = 15;
                size = 0;
            }else{
                runlength = zero_count;
                size = (uint8_t)log2(abs(zigzag[index])) + 1;
            }
            rle[count++] = runlength << 4 | size;
            amplitude = zigzag[index];
            rle[count++] = (uint8_t)amplitude;
            zero_count = 0;
            index++;
        }
    }
    // end of block
    rle[count++] = 0;
    rle[count++] = 0;
    *rle_print_size = count;
    rle = realloc(rle, count * sizeof(uint8_t));
    return rle;
}
```

Listing 4.4: RLE

## 4.3 Implementace ve VHDL

### 4.3.1 Určení numerické přesnosti

Při použití datového typu s pevným řádem 3.3.1 je důležité zvolit adekvátně velké bitové slovo, pro reprezentaci všech hodnot, s kterými bude vnitřně bude nakládáno. Pro správnou volbu je nutné zvážit :

1. Velikostí bitového slova ovlivňuje rozsah čísel, které je možné reprezentovat. Čím větší je bitové slovo, tím větší rozsah čísel lze zahrnout. Použitím příliš malého bitového slova je omezena možnost reprezentace velkých čísel, nebo je nutné provádět speciální operace, aby se s takovými hodnotami zacházelo správně.

2. Čím větší je desetinná část slova, tím vyšší numerické přesnosti lze dosáhnout. Větší bitové slovo nám umožňuje a zpracovávat více číselných hodnot s vyšší přesností. Pokud bychom naopak použili malé bitové slovo, mohlo by docházet k zaokrouhlovacím chybám a ztrátě důležitých číselných informací.

3. Velikosti bitového slova ovlivňuje výkon a velikost návrhu. Čím větší je bitové slovo, tím více paměti a výpočetního výkonu je potřeba pro manipulaci s ním. Správně zvolená velikosti bitového slova by mělo zajistit dostatečný výkon, aniž by se plýtvalo systémovými prostředky.

Vhodným datovým typem pro implementaci je **sfixed** (signed fixed-point). Ve standardu VHDL 2008 - IEEE P1076 [18] je pro reprezentaci tohoto datového typu zahrnuta knihovna Fixed point package[19].

Když navrhujeme s VHDL, ve skutečnosti navrhujeme přizpůsobený hardware. To znamená, že nejsme vázáni počítáním v žádných konkrétních délkách slov. Z tohoto důvodu si můžeme vybrat libovolný počet bitů, které odpovídají přesnosti a dynamickému rozsahu potřebnému pro naše výpočty. Existují dva způsoby, jak zvýšit délku slova. Můžeme buď zvýšit počet celých bitů nebo délku zlomkové části.[20]

Pro rozhodování o správné rozsahu bitového slova byla provedena analýza přesností v MATLABU pro velikosti sfixed od 8 do 32 bitů s automatizovaným zvolením nejvýhodnějšího rozsahu řádové čárky. Výsledek je překvapující vhodná velikost bitového slova s kompromisem přesností a velikostí vychází na 16 bitů pro prvky matice, koeficienty i normalizační konstantu.

Ideální prvek v matici bude mít velikost 16 bitů a 6 bitů z toho bude pro vyjádření desetinné části. M<10,6> bude ve VHDL specificky zapsáno:

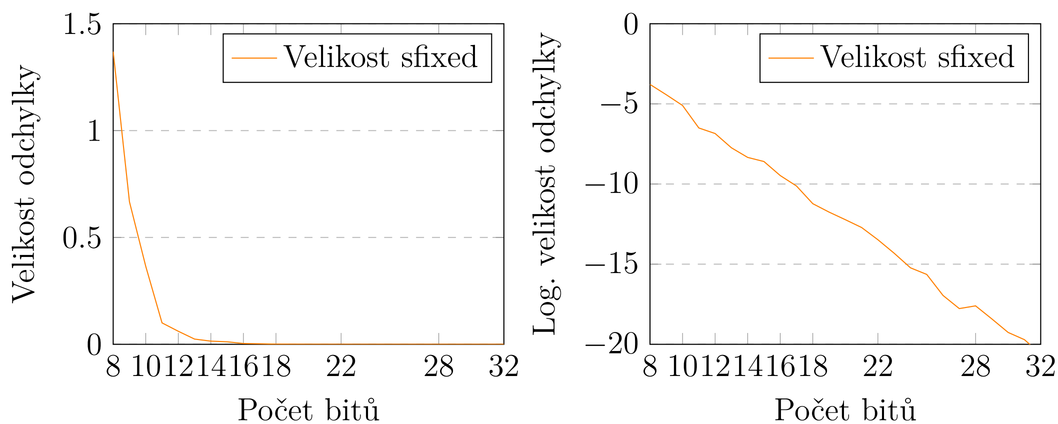
```
sfixed(10 downto -5)
```

Jako optimální velikost pro koeficienty $c_k$ vychází sfixed s rozsahem 16 bitů <2, 14>

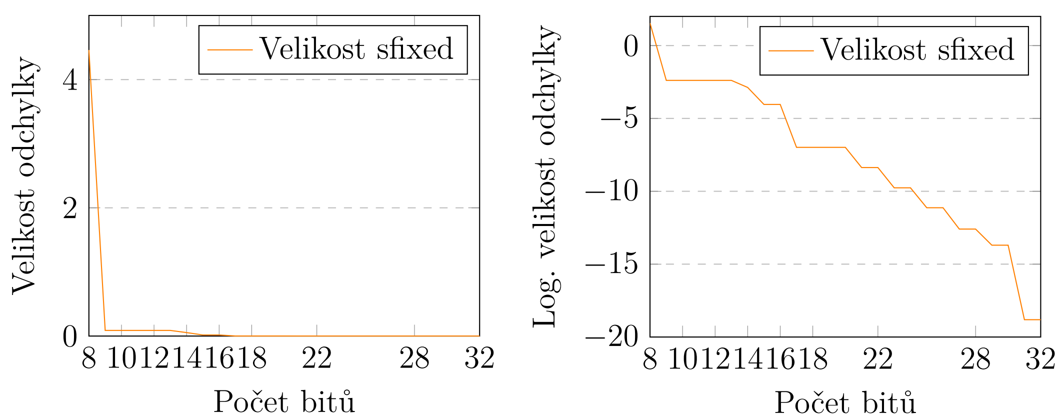
```
sfixed(2 downto -13)
```

Pro normalizační konstantu vychází optimální velikost pro sfixed rozsah <0,16>

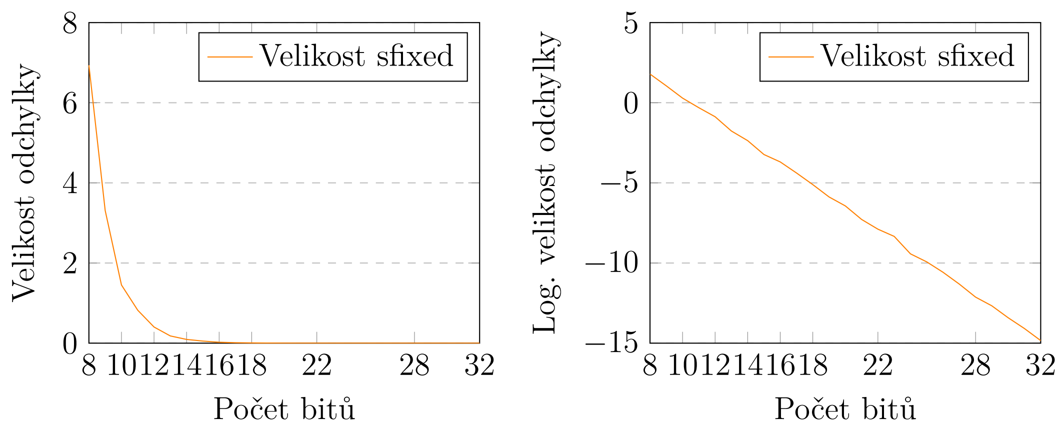
```
sfixed(0 downto -15)
```




Obrázek 4.1: Analýza velikosti odchylky koeficientů




Obrázek 4.2: Analýza velikosti normalizační konstanty




Obrázek 4.3: Analýza velikosti prvků matice po DCT

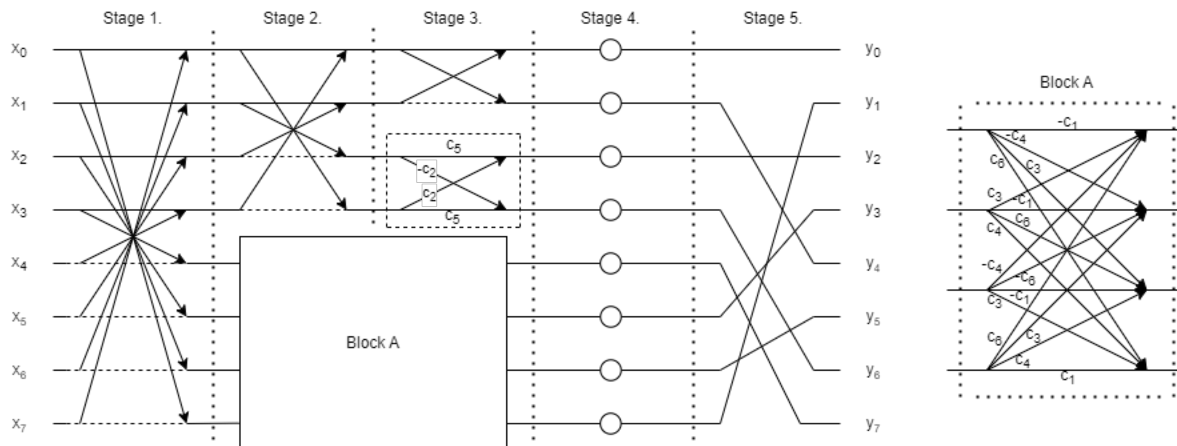
### 4.3.2 DCT

Schéma implementace dle 4.3 . Výpočet samotného DCT probíhá na 5 fází (1-3 fáze jsou čistě výpočetní, signály jsou buď sčítány, nebo násobeny koeficienty, 4. fáze je normalizační, v 5. fázi probíhá zaokrouhlení a uspořádání dat do vektoru výstupu.

Modul implementuje komunikační protokol AXI, [21] používaný pro SoC komunikaci mezi periferiemi a procesorem, který poskytuje efektivní a standardizovaný způsob přenosu dat. Jednotlivé moduly si mezi sebou komunikuji jako master a slave, přenáší informaci zda jsou data validní *data_valid*, zda je komponenta volna nebo zaneprázdněná *master_ready slave_ready* a samotná data.

DCT pracuje s 16 bitovými slovy typu sfixed . Na vstupu jsou ale data 8 bitová. Proto je vytvořen vedlejší modul, který tato data převede do 16 bitů a také provede subtrakci 128 od vstupních hodnot. Při provádění aritmetických operací, jako je sčítání a násobení, je nutné zvětšit velikosti těchto slov tak, aby se zachovala numerická přesnost výsledků. Konkrétně se velikost slova zvětšuje o hodnotu +1 při sčítání a 2x při násobení. V knihovně sfixed je pro zjištění optimální velikosti připravena funkce a to jak pro horní tak dolní mez, ukázka nastavení pro výpočet $y = a + b$:

```
signal y: sfixed(sfixed_high(a, '+', b) downto sfixed_low(a '+',b);
```




Obrázek 4.4: Graf toku signálu Loefflerovy modifikované DCT transformace. Přerušované čáry představují násobení $(-1)$. Fáze 1, 2, 3, 4, 5 představují jednotlivé kroky, které se provedou v jednom hodinovém taktu.

#### Velikost návrhu DCT

Protože každá provedení aritmetické operace zvětší výsledný vektor, hodnot, které jsou při přenosu stejně oříznuty na 16bitů, nabízí se vždy po výpočtu zaokrouhlit a saturovat výsledek na desetinná místa. Takto provedené zaokrouhlování je z hlediska ztráty přesnosti o $10^{-3}$ a následující kvantizaci, vůči výsledné velikosti návrhu výhodné implementovat. Porovnání velikosti návrhu v tabulce 4.2

| Bloky | Se zaokrouhlování a saturací | Bez zaokrouhlování a saturace |
|---|---|---|
| LUT | 1469 | 651 |
| FF | 445 | 354 |
| DSP | 25 | 28 |
| IO | 262 | 262 |

Tabulka 4.2: Počet komponent vytvořených po syntéze na FPGA

### 4.3.3 Transpozice matice

K výpočtu 2D DCT je nutné po prvním průchodu DCT data uložit, transponovat a znovu předat do DCT modulu. K tomu lze přistoupit dvěma způsoby:

1. Polem registrů. V případě použití pole registrů by bylo nutné umístit modul výpočtu DCT před i za, výhodou toho návrhu spočívají v nižší složitosti a nižší režijní řízení datového toku, nicméně nevýhody jsou mnohem markantnější: lze pouze pracovat s jedním blokem dat, výrazně se zvýší velikost návrhu o druhý DCT modul a pro blok dat o velikosti 8×8 s 16bitovými slovy návrh zabere dalších 1024 LUT.

2. Paměti RAM. Paměťové adresové prostory mohou být dynamicky přidělovány a uvolňovány, což umožňuje práci s různými velikostmi datových bloků a optimalizaci paměťového využití. Oproti poli registrů se do RAM hůře přistupuje, špatně se z ní paralelně čte.

Výhody použití paměti RAM jsou jasné, proto bude použita pro návrh.

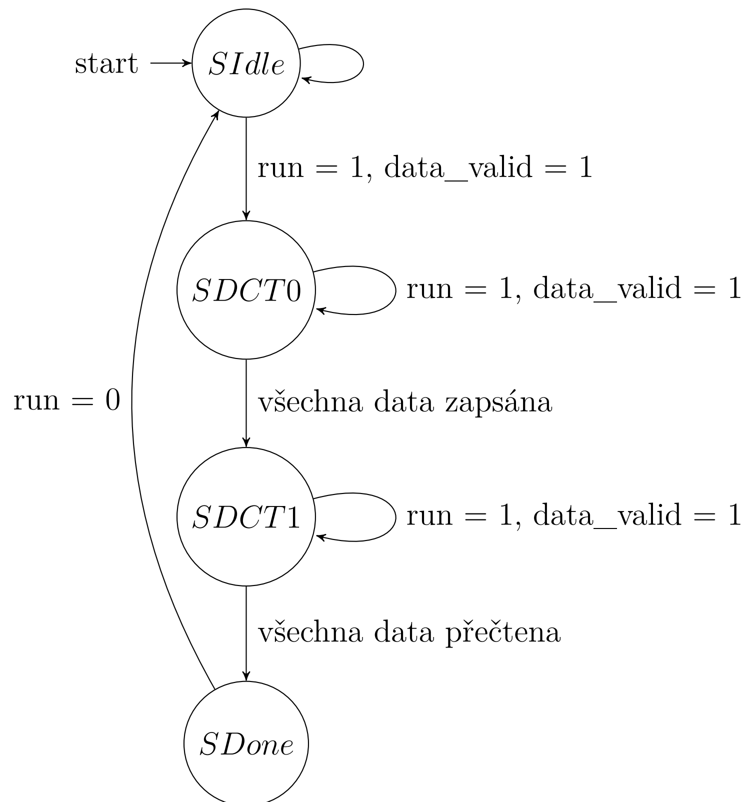
O celý proces se stará sekvencer fungující jako stavový automat adresující uložení vektor sfixed hodnot.

Kde jednotlivé stavy značí: DCT0: ze sběrnice jdou data do DCT modulu a informace o tom zda jsou data validní, modul po $x$ taktech vrací vektory diskretizovaných hodnot, ty jsou zapisovány "do sloupců", dokud není paměť RAM naplněna jedním blokem kvantizovaných hodnot z DCT modulu. DCT1: do DCT modulu jdou data z RAM "po řádcích" a z DCT jdou data do modulu kvantizace a cik-cak.

1. SIdle (Nečinný stav):
   Výchozí stav sekvenceru po resetu, kdy je nečinný, adresa a čítače jsou vynulovány. Čeká se na aktivační signál *run* pro přechod do stavu SDCT0.

2. SDCT0 (Stav výpočtu 1D DCT a zápisu vypočtených dat do paměti):
   Přechod do stavu: Při aktivaci signálu *run*. V přítomnosti validních dat z DCT modulu, sekvencer data příjme a inkrementuje čítač sloupců 8x podle kterého se posouvá *write enable*. Každý i-ty pixel je tak replikován 8x pro každou i-tou řádkovou paměť. Po zápisu jednoho celého bloku dat se přechází do stavu SDCT1.

3. SDCT1 (Stav čtení dat modulem DCT pro výpočet 2D DCT a přenos dat do modulu kvantizace a cikcak):




Obrázek 4.5: Stavový automat sekvenceru

Přechod do stavu: Po zápisu všech dat z modulu DCT do paměti RAM. Sekvencer inkrementuje 8x čítač řádků a každý prvek v řádku z paměti RAM a je předán do DCT modulu. Zároveň se v tomto stavu provádí přenos vypočtených dat z modulu DCT dále pipelinou do modulu kvantizace a cikcak. Přenos dat do modulu DCT2 a vypočtených dat probíhá současně. Po vyčtení všech dat z paměti RAM se přechází do stavu SDone.

4. SDone (Dokončení operace):
   Přechod do stavu: Po přečtení všech dat z paměti RAM. Sekvencer dokončil operaci a je připraven na další cyklus. V tomto stavu může být, dokud není deaktivován signál "run" a nejsou obdržena nová validní data z modulu DCT což způsobí přechod zpět do stavu SDCT0.

Pro řízení komunikace je vytvořen multiplexor, který přepíná dle vstupního signálu, která data půjdou do modulu DCT.

### 4.3.4 Kvantizace a cik-cak

Modul, který provádí kvantizaci vstupních dat pomocí kvantizační matice 3.4 uložené v paměti a přeskupuje kvantovaná data ve specifickém pořadí definované maticí

cik-cak indexů 4.4 Kvantizace probíhá pro vektor 8 hodnot zároveň.

$$A(8 * i + j) = M(i, j) * \frac{1}{Q(i, j)}$$

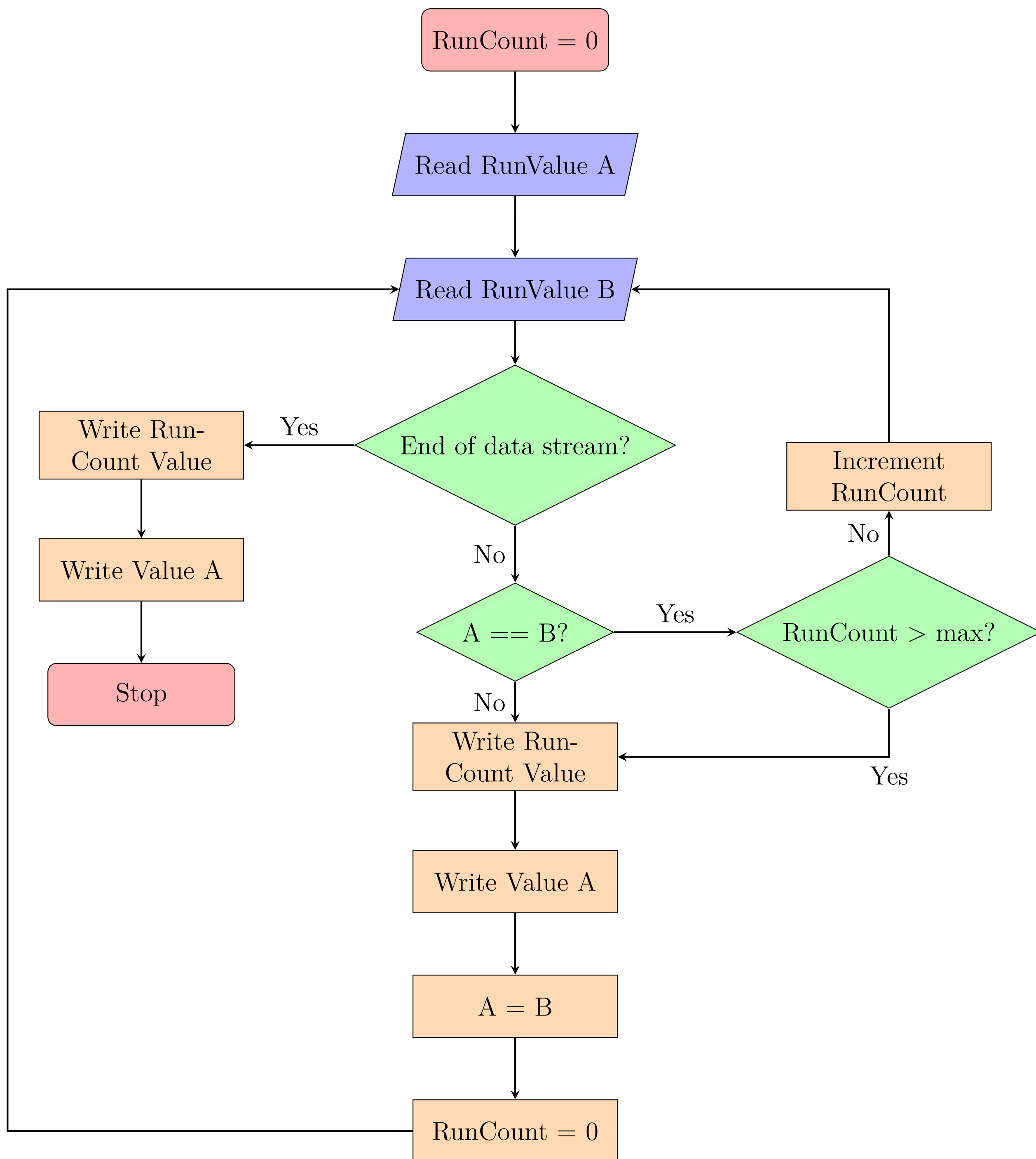
Modul taktéž implementuje komunikační rozhraní AXI, pokud jsou data validní tak se adresuje vektor kvantizačních prvků typu ufixed<1,7> ten je třeba konvertovat do sfixed<0,8> pro provedení aritmetických operací se vstupním vektorem hodnot v rozsahu sfixed<10,-5>. Hodnoty jsou mezi sebou pronásobeny, zaokrouhleny a uloženy do 8bitového vektoru podle matice.

### 4.3.5 RLE

RLE lze řešit jako konečný automat, jak je znázorněno na navrženém řešení pomocí vývojového diagramu 4.6, bohužel takové řešení není vhodné protože při výpočtu RLE je nutné sledovat délku opakovaných sekvencí a ukládat je do výstupního řetězce. Postup je ale závislý na předchozích hodnotách a nelze jej efektivně provádět současně s ostatními fázemi výpočtu. Musíme tedy přepnout na sekvenční výpočet, kde postupně zpracováváme vstupní data. Tímto přechodem dochází k zpomalení celkového procesu. Veškerý čas, který byl ušetřen díky paralelnímu zpracování v předchozích fázích pipeliningu, je prakticky zbytečné, protože musíme čekat na dokončení sekvenčního výpočtu RLE. To vytváří bottleneck. Hardwarově tedy RLE řešeno nebude a bude se řešit se až poté co FPGA vykoná akcelerovaný výpočet.

V důsledku toho je nutné přejít na sekvenční výpočet, kde se postupně zpracovávají vstupní data. Tento přechod vede k zpomalení celkového procesu. Veškerý čas, který byl ušetřen díky paralelnímu zpracování v předchozích fázích pipeliningu, je prakticky ztracen, protože musíme čekat na dokončení sekvenčního výpočtu RLE. Tím vzniká tzv. bottleneck, kdy je celý proces zpomalen v důsledku jednoho procesu.

Vzhledem k těmto omezením nelze RLE efektivně implementovat v rámci hardwaru Namísto toho je vhodné přistoupit k výpočtu této operace na procesoru. Je třeba poznamenat, že samotné RLE bude implementováno až poté, co FPGA dokončí akcelerovaný výpočet. Tímto způsobem lze dosáhnout optimálního výkonu a minimalizovat negativní dopad na celkový výpočetní proces.




Obrázek 4.6: Návrh RLE řešení jako konečný automat

## 4.4 Testování

Zde se zaměříme na ověření správnosti a efektivnosti implementovaných algoritmů. Konkrétně DCT, kvantizace a RLE. Bude provedeno detailní testování s důrazem na přesnost výpočtu a odchylky od očekávaných výsledků. Současně bude také hodnocena rychlost výpočtů, abychom získali informace o efektivitě navrženého řešení. Cílem je poskytnout objektivní analýzu a vyhodnocení implementovaných algoritmů z hlediska jejich funkcionality, přesnosti a výkonu.

Pro referenční porovnání dat použijeme blok hodnot z Wikipedie [13]

$$\begin{bmatrix} 52 & 55 & 61 & 66 & 70 & 61 & 64 & 73 \\ 63 & 59 & 55 & 90 & 109 & 85 & 69 & 72 \\ 62 & 59 & 68 & 113 & 144 & 104 & 66 & 73 \\ 63 & 58 & 71 & 122 & 154 & 106 & 70 & 69 \\ 67 & 61 & 68 & 104 & 126 & 88 & 68 & 70 \\ 79 & 65 & 60 & 70 & 77 & 68 & 58 & 75 \\ 85 & 71 & 64 & 59 & 55 & 61 & 65 & 83 \\ 87 & 79 & 69 & 68 & 65 & 76 & 78 & 94 \end{bmatrix} \tag{4.5}$$

Referenční 2D DCT výpočet zaokrouhlený na 3 desetinná místa

$$\begin{bmatrix} -415,38 & -30,19 & -61,20 & 27,24 & 56,12 & -20,10 & -2,39 & 0,46 \\ 4,47 & -21,86 & -60,76 & 10,25 & 13,15 & -7,09 & -8,54 & 4,88 \\ -46,83 & 7,37 & 77,13 & -24,56 & -28,91 & 9,93 & 5,42 & -5,65 \\ -48,53 & 12,07 & 34,10 & -14,76 & -10,24 & 6,30 & 1,83 & 1,95 \\ 12,12 & -6,55 & -13,20 & -3,95 & -1,87 & 1,75 & -2,79 & 3,14 \\ -7,73 & 2,91 & 2,38 & -5,94 & -2,38 & 0,94 & 4,30 & 1,85 \\ -1,03 & 0,18 & 0,42 & -2,42 & -0,88 & -3,02 & 4,12 & -0,66 \\ -0,17 & 0,14 & -1,07 & -4,19 & -1,17 & -0,10 & 0,5 & 1,68 \end{bmatrix} \tag{4.6}$$

Po kvantizaci a cik-cak

$$\begin{bmatrix} -26 & -3 & 0 & -3 & -2 & -6 & 2 & -4 \\ 1 & -3 & 1 & 1 & 5 & 1 & 2 & -1 \\ 1 & -1 & 2 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{4.7}$$

### 4.4.1 Test DCT

Po provedení testování navrženého řešení ve VHDL v porovnání s upraveným VHDL kódem pracujícím s přesností reálných čísel, bylo možné získat zajímavé výsledky. Upravený kód není syntetizovatelný, ale lze ho simulovat na behaviorální úrovni a využít pro srovnání s řešením s pevnou přesností. Pro testování byl jako referenční datový blok zvolen 4.5, a následně byla generována náhodná data pro vstup. Test probíhal po dobu 10 ms, což představuje 1 000 000 vstupních vektorů, z nichž každý obsahuje osm hodnot. Během vyhodnocování byly sledovány dva parametry: průměrná odchylka a maximální naměřená odchylka mezi sumou vypočtených a referenčních hodnot.4.3.

| Počet uskutečněných testů | Průměrná odchylka ze sumy výpočtu | Maximální naměřená odchylka |
|---:|---:|---:|
| 100 | 0,12829 | 0,21875 |
| 1 000 | 0,12353 | 0,25 |
| 10 000 | 0,12354 | 0,25 |
| 100 000 | 0,12689 | 0,28125 |
| 1 000 000 | 0,12549 | 0,28125 |

Tabulka 4.3: Měření velikosti odchylky

Tabulka 4.3 prezentuje výsledky měření velikosti odchylky. Bylo provedeno několik testů s různým počtem iterací. Jak je patrné, průměrná odchylka se pohybuje v rozmezí 0,12353 až 0,12829, zatímco maximální naměřená odchylka se pohybuje od 0,21875 do 0,28125. Tyto výsledky jsou více než uspokojivé, jelikož rozdíl mezi vypočtenými a referenčními hodnotami je pouze na úrovni 1 bitu. Lez konstatovat že implementované řešení tedy dosahuje přesných výsledků s minimálními odchylkami.

### 4.4.2 Testování DCT2 kvantizace s cik-cak reindexací

Testování DCT2 kvantizace s cik-cak reindexací přináší mnohem větší rozdíly co se týče numerické přesnosti. Po provedení 2D DCT lze pozorovat rozdíly ve výsledných hodnotách matice mezi implementací s plovoucí řádovou čárkou v jazyce C a simulovanými hodnotami ve VHDL než v případě testování DCT 4.4.1 s pevnou desetinou již na místě desetin, což ale vůbec nevadí protože hodnoty při kvantizaci celočíselně dělíme a zaokrouhlujeme, přeindexujeme podle cik-cak vzoru. Hodnoty po kvantizaci jsou navíc převedeny na 8bitů proto může docházet k drobným odchylkám hodnot o $\pm 1$, toho si lze všimnout při porovnání 4.7c s 4.7d na druhém řádku u posledního prvku.

```
2D DCT:
-415.3750 -30.1857 -61.1971  27.2393  56.1250 -20.0952  -2.3876   0.4618
   4.4655 -21.8574 -60.7580  10.2536  13.1451  -7.0874  -8.5354   4.8769
 -46.8345   7.3706  77.1294 -24.5620 -28.9117   9.9335   5.4168  -5.6490
 -48.5350  12.0684  34.0998 -14.7594 -10.2406   6.2960   1.8312   1.9459
  12.1250  -6.5534 -13.1961  -3.9514  -1.8750   1.7453  -2.7872   3.1353
  -7.7347   2.9055   2.3798  -5.9393  -2.3778   0.9414   4.3037   1.8487
  -1.0307   0.1831   0.4168  -2.4156  -0.8778  -3.0193   4.1206  -0.6619
  -0.1654   0.1416  -1.0715  -4.1929  -1.1703  -0.0978   0.5013   1.6755
```

(a) Výstup 2D DCT z Vitis implementace na procesoru ARM Cortex-A9

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -415.40625 | -30.3125 | -61.28125 | 27.125 | 56.0625 | -20.21875 | -2.46875 | 0.34375 |
| 4.40625 | -21.9375 | -60.78125 | 10.21875 | 13.125 | -7.125 | -8.5625 | 4.84375 |
| -46.84375 | 7.34375 | 77.09375 | -24.59375 | -28.9375 | 9.90625 | 5.375 | -5.6875 |
| -48.5625 | 12.03125 | 34.0625 | -14.8125 | -10.28125 | 6.25 | 1.78125 | 1.90625 |
| 12.125 | -6.5625 | -13.21875 | -3.96875 | -1.875 | 1.75 | -2.8125 | 3.125 |
| -7.78125 | 2.875 | 2.34375 | -5.96875 | -2.40625 | 0.9375 | 4.25 | 1.78125 |
| -1.0625 | 0.1875 | 0.375 | -2.4375 | -0.90625 | -3.03125 | 4.09375 | -0.6875 |
| -0.21875 | 0.09375 | -1.125 | -4.25 | -1.21875 | -0.125 | 0.46875 | 1.65625 |

(b) Výstup 2D DCT z behaviorální simulace ve Vivadu

```
Zig zag:
-26  -3   0  -3  -2  -6   2  -4
  1  -3   1   1   5   1   2  -1
  1  -1   2   0   0   0   0   0
 -1  -1   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
  0   0   0   0   0   0   0   0
```

(c) Kvantizovaný výstup po cikcak reindexaci na procesoru

```
-26,-3,0,-3,-2,-6,2,-4
1,-3,1,1,5,1,2,0
1,-1,2,0,0,0,0,0
-1,-1,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
```

(d) Kvantizovaný výstup po cikcak reindexaci z behaviorální simulace

Obrázek 4.7: Implementace v C pro procesor a behaviorální simulace

## 4.5 Výsledky z FPGA

Vedoucí práce Ing. Martin Rozkovec, Ph.D. sestavil wrapper celého řešení, které je tak možné ve Vitisu nahrát na desku a porovnat tak výpočet na hardwaru s výpočtem simulovaným a také změřit skutečnou rychlost výpočtu. V setupu procesor posílá referenční data do periférií desky a následně probíhá test datové propustnosti.

Testovací výstup z desky je následující:

```
pseudoJPEGenc test application
source macroblock:
0: 0052 0055 0061 0066 0070 0061 0064 0073
1: 0063 0059 0055 0090 0109 0085 0069 0072
2: 0062 0059 0068 0113 0144 0104 0066 0073
3: 0063 0058 0071 0122 0154 0106 0070 0069
4: 0067 0061 0068 0104 0126 0088 0068 0070
5: 0079 0065 0060 0070 0077 0068 0058 0075
6: 0085 0071 0064 0059 0055 0061 0065 0083
7: 0087 0079 0069 0068 0065 0076 0078 0094
Setup done in 837 ticks, run in 304 ticks

Computed zig-zagged a quantized 2D DCT:
0: -026 -003 0000 -003 -002 -006 0002 -004
1: 0001 -003 0001 0001 0005 0001 0002 0000
2: 0001 -001 0002 0000 0000 0000 0000 0000
3: -001 -001 0000 0000 0000 0000 0000 0000
4: 0000 0000 0000 0000 0000 0000 0000 0000
5: 0000 0000 0000 0000 0000 0000 0000 0000
6: 0000 0000 0000 0000 0000 0000 0000 0000
7: 0000 0000 0000 0000 0000 0000 0000 0000
RLE:
(0,5)(-26); (0,2)(-3); (1,2)(-3); (0,2)(-2);
(0,3)(-6); (0,2)(2); (0,3)(-4); (0,1)(1);
(0,2)(-3); (0,1)(1); (0,1)(1); (0,3)(5);
(0,1)(1); (0,2)(2); (0,1)(-1); (0,1)(1);
(0,1)(-1); (0,2)(2); (5,1)(-1); (0,1)(-1);
SPEED TEST
HW:
Image 1024*8 repeated 96 times
Image done in 1566813 ticks.
Setup done in average 524 ticks, run in average 15796 ticks.
C:
C code macroblock computed in 6089 ticks
For (1024*768/64) block in 10461760 ticks
```

Vypočtená data na FPGA odpovídají těm simulovaným 4.7d.

## 4.6 Porovnání rychlosti

V tabulce 4.4 je rychlostní porovnání jednotlivých implementací. Porovnání je provedeno mezi implementací na FPGA, ARM Cortex-A9 procesorem a vlastním laptopu s procesorem Intel i5-10210U.

Kdy 1 tik = ( 1/333.3 MHz systémový čítač) = 3 ns

| Implementace | Počet taktů pro testovací makro blok | Počet taktů pro obraz o velikosti 1024x768 | Čas pro testovací makro blok | Čas pro obraz o velikosti 1024x768 |
|---|---|---|---|---|
| FPGA | 837 | 1 566 816 | 2,5 $\mu s$ | 4,7 $ms$ |
| ARM Cortex-A9 | 6 089 | 10 461 760 | 18,26 $\mu s$ | 265,42 $ms$ |
| Laptop s Intel i5-10210U ve Visual Studiu | – | – | 72,10 $\mu s$ | 206,10 $ms$ |

Tabulka 4.4: Porovnání rychlosti mezi FPGA, ARM procesorem a laptopem

Z výsledků je patrné, že FPGA implementace výrazně převyšuje ostatní implementace z hlediska rychlosti výpočtů. FPGA je schopná provést DCT za velmi krátký časový úsek díky paralelním zpracováním schopnostem a optimalizaci algoritmu. Tento výsledek potvrzuje výhody a efektivitu FPGA technologie pro aplikace, které vyžadují vysoký výkon a rychlou reakci.

# 5 Závěr

V této bakalářské práci byl proveden návrh a implementace ztrátové komprese na FPGA, konkrétně na platformě ZedBoard s využitím vývojových nástrojů balíku Xilinx Vitis. Cílem práce bylo implementovat postup ztrátové komprese vhodný pro hardwarovou implementaci, přičemž výpočetně náročné operace byly realizovány na FPGA a proces RLE kódování na existujícím procesoru ARM Cortex-A9.

V rámci práce byla provedena analýza a seznámení se s pokročilými metodami ztrátové komprese, včetně Diskrétní kosinové transformace, Diskrétní vlnkové transformace, kvantizace, RLE, Huffmanova kódování . Byly zkoumány jejich vlastnosti a vhodnost pro implementaci na FPGA.

Na základě analýzy byl navržen a implementován postup ztrátové komprese, který zahrnuje výpočet 2D DCT, kvantizaci a cik-cak transformaci na FPGA. Tento postup byl implementován a otestován na testovacích datech. Výsledky byly porovnány s referenčními výsledky standardu JPEG, aby byla zhodnocena kvalita a efektivita implementace.

Výsledky ukázaly, že implementace ztrátové komprese na FPGA dosáhla výrazného zrychlení a optimalizace výpočtů ve srovnání s CPU implementací. FPGA poskytuje dostatečný výkon pro efektivní zpracování výpočetně náročných operací ztrátové komprese. Bylo dosaženo výrazného snížení velikosti dat a zachování přijatelné kvality obrazu.

Tato práce představuje praktický příklad implementace ztrátové komprese na FPGA s využitím platformy ZedBoard. Poskytuje užitečné poznatky o optimalizaci a výkonu FPGA implementace a může sloužit jako výchozí bod pro další výzkum a vývoj v oblasti ztrátové komprese na FPGA.

V budoucím výzkumu se samozřejmě nabízí práci doplnit o Huffmanovo nebo jiné entropické kódování na procesoru a přidání hlavičky a přiblížit se tak ke standardizovanému JPEG enkodéru případně rozšířit o vhodnou hardwarovou implementaci RLE kódování metody ztrátové komprese. V potaz přidá i dodatečná optimalizace již existující implementace pro dosažení ještě lepšího výkonu. Dále je možné zkoumat možnosti výpočtu na větší blocích dat a distribuce výpočtů na více komponent nebo využití novějších FPGA technologií s vyššími výpočetními kapacitami.

# Použitá literatura


[1] *MATLAB what is matlab* [online]. United States: The MathWorks, Inc., 2023 [cit. 2023-05-02]. Dostupné z: https://www.mathworks.com/discovery/what-is-matlab.htm.

[2] *MATLAB solutions* [online]. United States: The MathWorks, Inc., 2023 [cit. 2023-05-02]. Dostupné z: https://www.mathworks.com/solutions.html%5C#capabilities.

[3] MATHWORKS. *Discrete cosine transform - Wikipedia* [https://en.wikipedia.org/wiki/Discrete_cosine_transform]. [B.r.]. (Accessed on 05/13/2023).

[4] ACHARYA, Tinku; RAY, Ajoy K. *Image processing: principles and applications*. 2. vyd. Hoboken, N.J.: Wiley, 2005. ISBN 04-717-1998-6.

[5] GONZALEZ, Rafael C.; WOODS, Richard E. *Digital image processing*. Fourth edition. New York: Pearson, 2018. ISBN 12-922-2304-9.

[6] COELHO, Diego F. G. et al. Low-Complexity Loeffler DCT Approximations for Image and Video Coding. *Journal of Low Power Electronics and Applications*. 2018, roč. 8, č. 4. ISSN 2079-9268. Dostupné z DOI: 10.3390/jlpea8040046.

[7] MATHWORKS. *2-D diskrétní kosinusová transformace - MATLAB dct2* [https://www.mathworks.com/help/images/ref/dct2.html]. [B.r.]. (Accessed on 05/12/2023).

[8] DEVCORE. *8x8 DCT (discrete cosine transformation)* [https://upload.wikimedia.org/wikipedia/commons/2/24/DCT-8x8.png]. (Accessed on 05/18/2023).

[9] *jpeg5_5-116845032625000.png (256×256)* [https://www.root.cz/clanky/programujeme-jpeg-kvantizace-dct-koeficientu/#k02]. (Accessed on 05/18/2023).

[10] MALLAT, S.G. A theory for multiresolution signal decomposition: the wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1989, roč. 11, č. 7, s. 674–693. ISSN 1939-3539. Dostupné z DOI: 10.1109/34.192463.

[11] *Fixed-Point Data Types - MATLAB & Simulink* [https://www.mathworks.com/help/fixedpoint/gs/fixed-point-data-types.html]. [B.r.]. (Accessed on 05/16/2023).

[12] *Floating-Point Numbers - MATLAB & Simulink* [https://www.mathworks.com/help/fixedpoint/ug/floating-point-numbers.html]. [B.r.]. (Accessed on 05/16/2023).

[13] *JPEG - Wikipedia* [https://en.wikipedia.org/wiki/JPEG]. [B.r.]. (Accessed on 05/20/2023).

[14] *Downsampling image* [https://towardsdatascience.com/whats-lost-in-jpeg-e5e6e80b1f94]. [B.r.]. (Accessed on 05/20/2023).

[15] DĄBROWSKA, Agnieszka; WIATR, Kazimierz. CHEN AND LOEFFLER FAST DCT MODIFIED ALGORITHMS IMPLEMENTED IN FPGA CHIPS FOR REAL-TIME IMAGE COMPRESSION. In: *Computer Vision and Graphics: International Conference, ICCVG 2004, Warsaw, Poland, September 2004, Proceedings*. Ed. WOJCIECHOWSKI, K. et al. Dordrecht: Springer Netherlands, 2006, s. 768–773. ISBN 978-1-4020-4179-2. Dostupné z DOI: 10.1007/1-4020-4179-9_111.

[16] CHEN, Wen-Hsiung; SMITH, C.; FRALICK, S. A Fast Computational Algorithm for the Discrete Cosine Transform. *IEEE Transactions on Communications*. 1977, roč. 25, č. 9, s. 1004–1009. Dostupné z DOI: 10.1109/TCOM.1977.1093941.

[17] LOEFFLER, C.; LIGTENBERG, A.; MOSCHYTZ, G.S. Practical fast 1--D DCT algorithms with 11 multiplications. In: *International Conference on Acoustics, Speech, and Signal Processing,* 1989, 988–991 vol.2. ISSN 1520-6149. Dostupné z DOI: 10.1109/ICASSP.1989.266596.

[18] *Standard for VHDL Language Reference Manual*. IEEE, 2009. Č. P1076. Dostupné také z: https://standards.ieee.org/ieee/1076/3666/.

[19] BISHOP, David. *Fixed point package user's guide*. 2.0. vyd. IEEE, 2011.

[20] *Aritmetika s pevnou čárkou s vysokou úrovní VHDL - Hardware Descriptions* [https://hardwaredescriptions.com/elementor-fixed-point-arithmetic-in-synthesizable-vhdl/]. [B.r.]. (Accessed on 05/20/2023).

[21] *Learn the architecture - An introduction to AMBA AXI* [https://developer.arm.com/documentation/102202/0300/AXI-protocol-overview]. [B.r.]. (Accessed on 05/20/2023).

[22] SAYOOD, Khalid. *Introduction to data compression*. 5th edition. Cambridge, MA: Elsevier, 2017. ISBN 978-012-8094-747.

[23] HUFFMAN, David A. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE*. 1952, roč. 40, č. 9, s. 1098–1101. Dostupné z DOI: 10.1109/JRPROC.1952.273898.

# A Přílohy

## A.1 Zdrojové kódy

Tabulka A.1: Struktura dodaného .zip souboru se zdrojovými kódy

| **Složka** | **Soubory** | **Komentář** |
| --- | --- | --- |
| VHDL | DCT_computing.vhd<br>DCT_data_preparation.vhd<br>DCT_stream_switch.vhd<br>project_type_def.vhd<br>quantiZZation.vhd<br>sq2.vhd | Výpočet DCT,<br>úprava vstupních dat,<br>mux pro vstup do DCT,<br>definované vlastní typy a funkce,<br>kvantizace a cik-cak,<br>sekvencer. |
| C | functions.c<br>main.c<br>print_functions.c | Implementované funkce,<br>soubor s testy,<br>funkce pro výpis. |
| MATLAB | chen_loeffler_22.mlx<br>MATLAB_vs_VHDL.mlx<br>mereni_presnosti.mlx<br>porovnani_DCT_metod.mlx<br>RLE.mlx<br>test_img.bmp<br>zigzag.mlx | Skripty a soubory pro analýzu a porovnání v MATLABu |