FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Nástroje pro podporu výuky samoopravných kódů v prostředí Wolfram Mathematica
Student:	Helena Linhartová
Vedoucí:	Ing. Pavel Kubalík, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Prozkoumejte existující řešení nástrojů vhodných k výuce samoopravných kódů.
Analyzujte problémy studentů při výuce kódů.
Zaměřte se zejména na tyto kódy: sudá parita, křížová parita, Hammingův kód, rozšířený i zkrácený Hammingův kód, cyklický kód, součinový kód a RM kód.
Navrhněte vlastní nástroje vhodné k výuce těchto kódů.
Zaměřte se zejména na oblast generování, dekódování a opravy těchto kódů.
Nástroje budou splňovat tyto požadavky:
 - generování souborů pro program Wolfram Mathematica umožňující studentům samoopravné kódy procvičit,
 - generování VHDL kódů a testbenche pro lepší představu jejich implementace v hardware,
 - vkládání chyb a jejich oprava.
Navržené řešení realizujte a řádně otestujte.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 9. února 2023 v Praze.

Bakalářská práce

NÁSTROJE PRO PODPORU VÝUKY SAMOOPRAVNÝCH KÓDŮ V PROSTŘEDÍ WOLFRAM MATHEMATICA

Helena Linhartová

Fakulta informačních technologií
Katedra číslicových obvodů
Vedoucí: Ing. Pavel Kubalík, Ph.D.
11. května 2023

Odkaz na tuto práci: Linhartová Helena. *Nástroje pro podporu výuky samoopravných kódů v prostředí Wolfram Mathematica*. Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěla bych srdečně poděkovat především vedoucímu této bakalářské práce Ing. Pavlu Kubalíkovi, Ph.D. za jeho ochotu konzultovat postup a problémy, které se po cestě vyskytly.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 citovaného zákona.

V Praze dne 11. května 2023 .....

Abstrakt

Tato bakalářská práce se věnuje tvorbě podpůrných materiálů pro výuku detekčních a samoopravných kódů. Na základě zvoleného kódu a jeho parametrů je prostřednictvím balíčků Wolfram Mathematica možné vygenerovat kodér, soubor vkládající chybu, dekodér, testbench a testovací data. Všechny tyto soubory jsou realizovány v jazyce VHDL pro lepší ilustraci principu kódování. V rámci práce se věnujeme těmto kódům: sudá parita, křížová parita, Hammingův kód, rozšířený Hammingův kód, zkrácený Hammingův kód, cyklický kód, součinový kód a RM kód.

Klíčová slova bezpečnostní kód, samoopravný kód, VHDL, sudá parita, křížová parita, Hammingův kód, cyklický kód, součinový kód, RM kód

Abstract

This bachelor thesis is devoted to creation of supporting materials for the teaching of detection and self-correcting keys. Based on the selected code and its parameters, the Wolfram Mathematica packages can generate an encoder, an error embedding file, a decoder, a testbench, and test data. All these files are implemented in VHDL to better illustrate the coding principle. Work is dedicated to the following codes: even parity, crossed parity, Hamming code, extended Hamming code, shortened Hamming code, cyclic code, product code and RM code.

Keywords security code, self-correcting code, VHDL, Even Parity, Cross Parity, Hamming code, Cyclic code, Product code, RM code

Seznam zkratek

ČVUT	České vysoké učení technické
FIT	Fakulta informačních technologií
VHDL	VHSIC Hardware Description Language
BI-JPO	Jednotky počítače (bakalářský předmět FIT)
WM	Wolfram Mathematica
RM	Reed-Muellerovy kódy
HK	Hammingův kód

Úvod

V informačních technologiích často nastane situace, kdy je třeba zahájit komunikaci mezi dvěma či více samostatnými entitami. Jedny z obecných základních požadavků na informační systémy jsou přesnost, rychlost a spolehlivost – tyto nároky se tedy automaticky rozšiřují i na jakékoliv spojení. Spojením můžeme rozumět přenos dat drátovým i bezdrátovým způsobem. Při obou variantách může dojít k šumu, tedy k rušení, jež může změnit posílaná data na chybná. Aby byly všechny nezbytnosti dodrženy, začínají do problematiky komunikace vstupovat takzvané detekční, či samoopravné kódy. Jak jejich název napovídá, tyto kódy mají za cíl buďto detekovat chybu nebo ji opravit a získat tak původně zasílanou informaci. S jejich použitím je po přijetí ihned zřejmé, zda jsou přijatá data validní, a zda s nimi můžeme pracovat.

Detekční i samoopravné kódy svůj úkol realizují tak, že data před zasláním modifikují a rozšíří o nadbytečné kousky informací, pomocí kterých snáze pozorují změny a určují následky šumu na přenášených datech.

Jedinci, kteří se věnují informatice na hardwarové úrovni, velmi pravděpodobně někdy přišli či přijdou do styku s principem realizace komunikace a jejím zabezpečením. Tato skupina jedinců se nepochybně do jisté míry překrývá i se studenty ČVUT FIT. Studenti bakalářského programu se těmto kódům (v jejich pravé podstatě jakožto nástroje komunikace) věnují zejména na předmětu Jednotky počítače, uváděného pod zkratkou BI-JPO.

Jednotky počítače je předmět pokrývající základy fungování procesoru včetně jeho komunikace s okolím. Trvá jeden semestr a v jeho druhé polovině je studentům představena problematika lineárních a cyklických kódů. Pro reálné použití je nedostačující umět ověřit, zda přijímaná zakódovaná zpráva je validní či nikoliv. Je nutné umět vytvořit svůj vlastní kód, který bude na míru odpovídat našim požadavkům. V tomto bodě se vyskytuje největší překážka výuky kódů na BI-JPO – tvora generující matice a kontrola její správnosti je zdlouhavé maticové počítání, které si student za omezený čas nezvládne řádně procvičit. Časový problém byl z části vyřešen promítáním nevyplněné matice, kterou vyučující doplňuje s doprovodným komentářem – výklad postupu je mnohem rychlejší. I přesto však přetrvává problém časové náročnosti u studentů. Dovednost správně zvolit parametry kódu se nejlépe naučí praxí, tedy opakováním pokusů, které buď selžou, nebo uspějí.

Tato práce by měla sloužit jako podpůrný materiál pro výuku vybraných detekčních a samoopravných kódů na předmětu BI-JPO ale i mimo něj. Zpřístupněním materiálů studentům, kde si mohou realizovat generující matice v řádu vteřin a ihned sledovat jejich výstup, jim nabídne možnost velmi rychle si osvojit náležitosti bezpečnostních kódů. Do cvičných přenosů bude možné vložit chybu, a sledovat následující detekci či opravu. Pomůcky pro výuku bezpečnostních kódů jakožto téma bakalářské práce jsem si zvolila s ohledem na vlastní zkušenosti z absolvování předmětu BI-JPO, kde kódování zpráv představovalo nemalou překážku.

Chci takto navázat na práce Stanislava Koleníka a Timura Ganeeva, kteří se v minulých letech bezpečnostním kódům také věnovali.

Cíle práce

Cílem této bakalářské práce je vytvořit sadu nástrojů, které budou figurovat jako podpůrné materiály výuky bezpečnostních kódů. V návaznosti na skutečnost, že cíleným příjemcem je student předmětu BI-JPO, bude nutné zanalyzovat problémy a postoje studentů k dané problematice.

Tato práce bude rozdělena na teoretickou a praktickou část.

Cílem teoretické části je shrnout poznatky o bezpečnostních kódech – tyto poznatky zahrnují definice nutných pojmů, princip jednotlivých kódů, postup generování, způsob určení chyby a její následné opravení. Konkrétně se budu věnovat sudé paritě, křížové paritě, Hammingovu kódu, zkrácenému Hammingovu kódu, rozšířenému Hammingovu kódu, cyklickému kódu, součinovému kódu a RM kódu.

Cílem praktické části je vytvořit výukové nástroje pro výše uvedené kódy, které by studentovi poskytli náhled na jejich obecné fungování. Výstupem této části budou nástroje generující VHDL skripty simulující jednotlivé kódy.

Kapitola 1

Teoretický úvod

Kód je způsob zobrazení informace A na sekvenci symbolů či znaků B (lze zapsat $A \rightarrow B$).

Pro bezpečnostní kódy je původní informace A uspořádaným vektorem $\mathbf{a} = (a_1, \ldots, a_k)$ o k bitech (k-tice), a B uspořádaným vektorem $\mathbf{b} = (b_1, \ldots, b_n)$ o n bitech (n-tice). Indexy k a n jsou přirozená nenulová čísla, splňující podmínku $n > k$. Pokud vektoru $\mathbf{b}$ pro daný kód náleží vektor $\mathbf{a}$, pak o vektoru $\mathbf{b}$ mluvíme jako o kódovém slově kódu K. O všech ostatních n-bitových slovech hovoříme jako o nekódových slovech.

Bezpečnostní kód zajišťuje spolehlivý přenos dat v podobě posloupnosti bitů, jež mohou být poničena během přenosu. Poničení se projevuje inverzí jednoho či více bitů na vektoru $\mathbf{b}$ (kódové slovo 0101 může být přenosem poničeno na 0001). [1, s. 8]

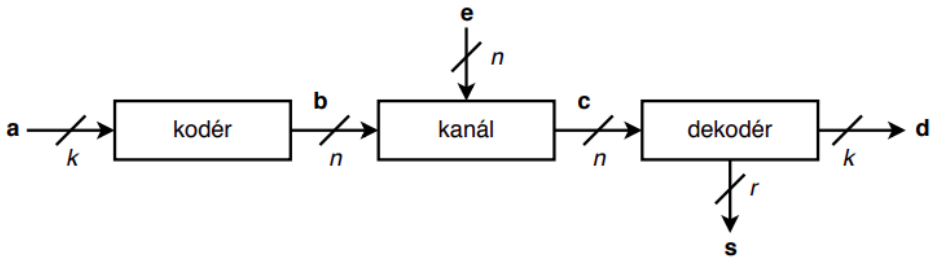
Účel bezpečnostních kódů je detekovat chybu a v některých případech ji i opravit. Proto lze bezpečnostní kódy rozdělit na dvě kategorie:

(a) detekční, kde dojde ke zjištění, že přijaté slovo je či není kódovým slovem, a

(b) samoopravné, kde přijaté slovo s chybou bude nahrazeno „nejpodobnějším“ kódovým slovem.

V rámci digitálního bezpečnostního kódování informací se často vyskytuje termín kanál, kterým se označuje prostor, v němž může dojít k chybě na přenášeném slově. Může to být skutečný komunikační kanál nebo třeba i paměť.

V procesu kódování se kanálem rozumí meziprostor kodéru a dekodéru. Kodér je jednotka zodpovědná za převod původní informace $\mathbf{a}$ na kódové slovo $\mathbf{b}$. Dekodér pak tvoří komplement ke kodéru, neboť z kanálu přijímá n-bitové slovo $\mathbf{c}$, které buď určí za neporušené, nebo chybné. Indikátorem přenosu zatíženého chybou je r-bitový syndrom $\mathbf{s} = (s_1, \ldots, s_r)$. Výstupem dekodéru je k-bitové slovo $\mathbf{d}$, které pro nulové $\mathbf{s}$ odpovídá původní informaci $\mathbf{a}$. Ve všech ostatních případech (nenulový syndrom) je jeho tvar neurčitý. Toto je základní princip detekčních kódů. Samoopravné kódy navíc využívají syndrom k určení konkrétního n-bitového vektoru chyby $\mathbf{e}$, a tím i vyslaného kódového slova $\mathbf{b}$, které pak jen převedou na správný výstup $\mathbf{d}$ (Obrázek 1.1). [2] [3]



Obrázek 1.1 Proces kódování (převzato z [1, s. 7])

Pro kód K definujeme kódovou vzdálenost (kvzd) jako počet bitů, ve kterých se dvě různá kódová slova kódu K odlišují (tabulka 1.1). Návazně na to definujeme také Hammingovu vzdálenost $\mu(K)$, neboli minimální kódovou vzdálenost, minimální počet indexů, na kterých se bity libovolných dvou kódových slov nerovnají. Všechna slova jsou od sebe oddělena minimálně počtem chyb (invertovaných bitů), které musí nastat, aby slovo u bylo mylně přijato jako slovo v (u a v jsou různá kódová slova). (tabulka 1.2)

Při použití samoopravných kódů se jejich očekávaný výsledek (schopnost detekovat a následně opravit chyby) zakládá právě na kódové vzdálenosti. Platí vztah $kvzd = dch + och + 1$, kde platí podmínka $dch \geq och$. Přirozené proměnné značí detekovatelné chyby dch a opravitelné chyby och. [1, s. 8-9] Pro K s kvzd = 2 lze detekovat jednu chybu. Pro kvzd = 3 lze buď detekovat dvě chyby, nebo detekovat jednu a jednu opravit. Pro kvzd = 4 lze buď detekovat tři chyby, nebo detekovat dvě chyby a jednu opravit. Nelze však detekovat jednu chybu a dvě opravit, neboť by to porušilo podmínku $dch \geq och$.

S myšlenkou Hammingovy vzdálenosti lze kódy geometricky interpretovat. Pro vyobrazení n-bitových kódových slov je využita n-rozměrná krychle. Hrany jsou značeny souřadnicemi, z nichž jsou některé shodné s prvky kódu K. Pro tato vyobrazení značí $\mu(K)$ minimální počet hran, přes které je možné přejít od jednoho kódového slova k druhému. [3]

Tabulka 1.1 Kódová vzdálenost dvou různých slov.

slovo u	slovo v	kvzd
01100	01000	1
11100	01011	4
01010	10101	5

Tabulka 1.2 Hammingova vzdálenost kódu K.

Kódy	Kódová slova	Hammingova vzdálenost $\mu(K)$
K_1	000, 011, 101, 110	2
K_2	111, 001, 010, 100	2

Chybový n-bitový vektor **e** je buďto nulový (bezchybný přenos), obsahuje jeden nenulový bit (jedna chyba) nebo obsahuje více nenulových bitů (dvě a více chyb). Pro třetí zmíněný případ lze množinu označit jako shluk chyb. Série bitů délky l je podmíněna výskytem nenulových bitů na jejích koncových pozicích. Všechny bity mezi nimi tvoří vlastní sérii délky l-2 a mohou nabývat jakékoliv hodnoty. Od jedné chyby k druhé může mít shluk chyb minimální délku 2, a maximální délku n, pro případ, že se chyba vyskytuje na 0. a n-1. indexu. Shluky mohou být do sebe vnořené (tabulka 1.3). [2]

Tabulka 1.3 Do sebe vnořené shluky chyb různých délek.

Shluky v $\mathbf{e} = (0\ 1\ 1\ 0\ 1\ 0\ 0\ 1\ 0)$	Délka shluku l
1 1 0 1 0 0 1	7
1 0 1 0 0 1	6
1 1 0 1	4
1 0 0 1	4
1 0 1	3
1 1	2

Těleso T je struktura, na níž jsou definovány dvě binární operace - sčítání $\oplus$ a násobení $\odot$ (splňující axiómy asociativity, neutrálního prvku, inverzního prvku). Pro dvouprvkové těleso lze samotné prvky značit jako bity, které nabývají hodnot 0 nebo 1. Operace součtu na bitových prvcích je realizována jako funkce XOR, zatímco součin jako funkce AND (tabulka 1.4). [4]

Tabulka 1.4 Funkce XOR a AND (bitový součet a součin).

Vstup a	Vstup b	XOR	Vstup a	Vstup b	AND
0	0	0	0	0	0
0	1	1	0	1	0
1	0	1	1	0	0
1	1	0	1	1	1

Vektorový (lineární) prostor V nad komutativním tělesem T s vektorovými operacemi součtu a součinu platí za předpokladu splnění těchto axiomů: [5]

(i) $\forall a, b \in V : a \oplus b = b \oplus a$

(ii) $\forall a, b, c \in V : (a \oplus b) \oplus c = a \oplus (b \oplus c)$

(iii) $\forall \alpha, \beta \in T, \forall a \in V : \alpha \odot (\beta \odot a) = (\alpha\beta) \odot a$

(iv) $\forall \alpha \in T, \forall a, b \in V : \alpha \odot (a \oplus b) = (\alpha \odot a) \oplus (\alpha \odot b)$

(v) $\forall \alpha, \beta \in T, \forall a \in V : (\alpha + \beta) \odot a = (\alpha \odot a) \oplus (\beta \odot a)$

(vi) $\forall v \in V : 1 \odot v = v$

(vii) $\exists \vec{0} \in V, \forall v \in V : 0 \odot v = \vec{0}$

Lineární kódy jsou typem bezpečnostních kódů pro efektivní detekci i opravu chyb. Kód K kódující zprávy délky k na slova délky n je lineárním (n, k)-kódem, jestliže K je podprostor daného konečného tělesa T^n dimenze k. Pro dvě a více kódových slov tohoto typu kódu platí, že jejich lineární kombinace je taktéž kódové slovo.

Kodér lineárních kódů je realizován jako generující matice $G_K \in T^{k,n}$. Její řádky jsou tvořeny bází podprostoru K, tedy lineárně nezávislým souborem vektorů jejichž lineární obal je roven kódu K. Pro zakódování původní informace $\mathbf{a}$ na kódové slovo $\mathbf{b}$ je využit vztah $b = a * G_K^{k,n}$.

Dekodér je realizován s logikou pro určení syndromu, kontrolní maticí $H_K \in T^{n-k,n}$. Pro tuto matici je množina $x \in K$ soustavou řešení $H_K * x = \vec{0}$. Pro přijatou zprávu $\mathbf{c}$ je syndrom $\mathbf{s}$ určen vztahem $s = c * H_K^T$.

Kód lze někdy klasifikovat jako systematický. Tedy jako kód $K \subseteq T^k$ s k informačními a $r = n-k$ kontrolními znaky a s bijektivním (vzájemně jednoznačném) zobrazením $\varphi : T^k \to K$. U lineárních systematických kódů je možné dosáhnout generující matice tvaru $G_K = (E^{k,k} B^{k,n-k})$, kde $E^{k,k}$ je jednotková matice s k řádky a k sloupci. Z této matice je možné odvodit kontrolní matici ve tvaru $H_K = ((B^{k,n-k})^T E^{n-k,n-k})$ (ukázka 1.2). [2]

$$G_K = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Generující matice

$$H_K = \begin{pmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Kontrolní matice

Obrázek 1.2 Ukázka dvojice generující a kontrolní matice systematického kódu.

Cyklické lineární kódy umožňují generovat kód K mnohočlenem $G(x)$. Jsou zapisovány ve tvaru

$$a_n * x^n + a_{n-1} * x^{n-1} + \cdots + a_1 * x + a_0,$$

s koeficienty a_i, $i \in (0, \ldots, n)$, a kde $a_n \neq 0$, je polynom stupně n, což lze zapsat jako $degP(x) = n$.

Kapitola 2
Rešerše

Mezi již existující pomůcky pro podporu výuky lineárních kódů se řadí diplomová práce Stanislava Koleníka a bakalářská práce Timura Ganeeva - obě napsané v rámci studia na FIT ČVUT.

2.1 Diplomová práce Stanislava Koleníka 2021

Stanislav Koleník se ve své diplomové práci z roku 2021 [6] věnoval tvorbě Wolfram Mathematica balíčků (tabulka 2.1), které pokrývají funkčnosti širokého spektra bezpečnostních kódů: Fireův kód, RM kód, Goppa kód, binární BCH kód, nebinární BCH kód, součinový kód, konvoluční kód a RS kód. Pro všechny zkoumané kódy nabízejí balíčky mimo jiné možnost kódovat a dekódovat zadanou zprávu, vytvořit generátor kódu (nejčastěji ve formě generující matice), detekovat a případně opravit chybu na přijatém slově, apod.

Páteří všech souborů je pomocný balíček *CommonCode.wl*, který obsahuje ty nejzákladnější funkce nevztahující se ke konkrétnímu kódu - superponuje chybu na kódové slovo, konvertuje generující matici na kontrolní a naopak, generuje náhodné vektory zadané délky, a mnoho dalších.

Pro každý balíček je paralelně součástí výstupu také notebook, který popisuje na konkrétních ukázkách využití všech veřejných funkcí dostupných po načtení souboru .wl. V těchto notebookách jsou jednotlivé ukázky doplněny o popis vykonávané operace. Ukázky příkladů poslouží při vyplňování příkladů na samotném konci souboru.

Tabulka 2.1 Balíčky vystupující z diplomové práce Stanislava Koleníka 2021.

Jména souborů	
BasicCode.wl	*BCHCode.wl*
CommondCode.wl	*CommonGFCode.wl*
ConvolutionalCode.wl	*FireCode.wl*
GoppaCode.wl	*HammingCode.wl*
InterleavedCode.wl	*LDPCCode.wl*
PolynomialCode.wl	*ProductCode.wl*
RMCode.wl	*RSCode.wl*

2.2 Bakalářská práce Timura Ganeeva 2021

Bakalářská práce Timura Ganeeva z roku 2021 [1] byla zaměřena na tvorbu Wolfram Mathematica souborů, které umožní generovat VHDL kodéry, dekodéry a testbenche pro danou skupinu

kódů: sudá parita, rozšířená sudá parita, křížová parita, Hammingův kód, rozšířený Hammingův kód, zkrácený Hammingův kód a cyklický kód.

WM generátory VHDL skriptů byly realizovány jako celkem 8 notebooků (tabulka 2.2). V nich byly do připravených buněk uživatelem zadány konkrétní parametry kódu, s nimiž postupným vyhodnocováním buněk, nebo vyhodnocením celého souboru najednou vznikla trojice skriptů pro zvolený kód - kodér, dekodér a testbench. Pokud dojde k vyhodnocení buněk v nesprávném pořadí, výsledné soubory mohou být nespustitelné.

Logika ve skriptech je většinou realizována kombinačně s výjimkou cyklického kódu, kde je logika sekvenční. Výstupní soubory jsou přehledné a snadno čitelné, doplněné o vysvětlující komentáře. Pro případnou simulaci musí být skripta doplněna o zastřešující entitu pro kodér, dekodér i testbench, vstupní data musí simulaci zadat uživatel.

Tabulka 2.2 Soubory vystupující z bakalářské práce Timura Ganneva z roku 2021.

Jméno souboru
parity.nb
cross_parity.nb
ext_cross_parity.nb
hamming.nb
shortened_hamming.nb
ext_hamming.nb
cyclic.nb

Kapitola 3
Analýza

3.1 Sudá parita

Sudá parita spočívá v přidání jednoho redundantnímu bitu k původnímu k-bitovému vektoru **a** čímž vznikne k+1-bitový vektor **b**, platí tedy $n = k + 1$. Vektor $\mathbf{a} = (a_1, \ldots, a_k)$ nabývá tvaru $\mathbf{b} = (a_1, \ldots, a_k, p)$, kde p značí redundanci. Přidaný bit doplní originální informaci **a** tak, že celkový počet jedničkových bitů ve vektoru **b** je sudý. Pokud počet jedniček ve vektoru **a** je již sudý, přidaný bit bude nula, jinak postupujeme opačně a přidaný bit je nenulový. Kódové slovo sudé parity pro operaci XOR na všechny jeho členy musí dát nulový výsledek, $a_1 \oplus a_2 \oplus \cdots \oplus a_{k-1} \oplus a_k \oplus p = 0$. [7, s. 15]

Existuje také lichá parita, která paritním redundantním bitem zajišťuje lichý počet nenulových bitů ve slově. Není to však lineární kód, neboť lineární kombinací dvou slov kódu liché parity, vznikne slovo se sudým počtem nenulových bitů (slovo mimo množinu K).

Sudá parita je velmi jednoduchý kód, který odhalí pouze přítomnost chyby **e** s lichým počtem nenulových bitů. Bez ohledu na to, zda přijatý vektor **c** je či není kódové slovo kódu K, pokud nastane sudý počet chyb, jednobitový syndrom **s** bude nulový, neboť bude zachována sudá parita (Hammingova vzdálenost je rovna 2).

Paritní kód lze generovat maticí tvořenou jednotkovou maticí o rozměrech $k \times k$ zprava doplněnou o sloupec bez nulových bitů. Pro určení syndromu pak stačí primitivní kontrolní matice o jediném řádku délky $k + 1$. Ukázku realizace kódu sudé parity lze vidět v tabulce 3.1. [2] [7, s. 16]

$$G_K = \left(\begin{array}{cccc|c} 1 & 0 & \ldots & 0 & 1 \\ 0 & 1 & \ldots & 0 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \ldots & 1 & 1 \end{array}\right)$$

$$H_K = \left(\begin{array}{ccc|c} 1 & 1 & \ldots & 1 & 1 \end{array}\right)$$

Tabulka 3.1 Sudá parita délky $4 + 1$ bitů.

4-bitový vektor **a**	$(4+1)$-bitový vektor **b**
0110	0110**0**
1001	1001**0**
1000	1000**1**
1110	1110**1**

3.2 Křížová parita

Křížová parita (alternativně příčná a podélná) je kód, který pro zabezpečení zapíše informaci **a** o $k = k_r \times k_s$ bitech do matice s k_r řádky a k_s sloupci. Každé dvojici řádku a sloupce přidělí jeden bit sudé parity. V důsledku je každý informační bit zabezpečen unikátní dvojicí paritních bitů. Bity v matici společně s přidanými paritními bity zapsanými do řádku vytvoří kódové slovo **b**. Křížová parita zaručuje Hammingovu vzdálenost rovnou 3, neboť liší-li se jediný informační bit původní zprávy **a**, pak se budou lišit i dva jeho paritní bity. [2]

Pro zprávu **a** o délce $k = 4$, by prvním krokem bylo vepsání do matice k_r řádků a k_s sloupců, jejichž součin je roven k. Matice může být typu 2×2 nebo 1×4.

Zprávu liché délky je možné zarovnat přidáním nulového bitu. Původní informace **a**$= (a_1, \dots a_4) =$ 1011 může být zakódována křížovou paritou jako **b**$= (a_1, \dots a_4, p_1, \dots p_4) =$ 10111001. (Tabulka 3.2) (Tabulka 3.3)

Tabulka 3.2 Křížová parita - matice s prvky a_1, a_2, a_3 a a_4 je zprava doplněna paritními bity pro řádky ($a_1 \oplus a_2 = p_1$, apod.), zdola paritními bity pro sloupce.

a_1	a_2	p_1
a_3	a_4	p_2
p_3	p_4	

Tabulka 3.3 Rozšířená křížová parita - má jeden paritní bit navíc, který je definován jako $p_1 \oplus p_2 = p_5$ nebo $p_3 \oplus p_4 = p_5$.

a_1	a_2	p_1
a_3	a_4	p_2
p_3	p_4	p_5

Generující matice kódu lze opět nejsnadněji dosáhnout doplněním jednotkové matice o paritní bity. Hammingova vzdálenost křížové parity jsou 3, je tedy možné detekovat dva chybné bity, nebo detekovat jeden a opravit ho. (Rozšířená křížová parita má Hammingovu vzdálenost rovnou 4, může tedy detekovat tři chybové bity, nebo dva a jeden opravit.) Každý informační bit je zabezpečený dvěma paritními bity. Ty jsou dále zabezpečeny dalšími dvěma bity. Křížová parita je systematický kód, proto je kontrolní matice odvoditelná z té generující. [1, s. 13]

$$G_K = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

$$H_K = \begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$$

3.3 Hammingův kód

Hammingův kód lze použít k detekci dvou chybných bitů, ale spíš k opravě jednoho bitu. Je to takzvaný perfektní kód. Pokud v dekodéru přijatý syndrom je nenulový, pak přísluší právě jedné chybě (můžeme s jistotou určit do kanálu odesílané kódové slovo). Hammingova vzdálenost kódu jsou 3.

Pro libovolný počet redundantních bitů $r = (n-k) > 1$ lze vytvořit kontrolní matici s rozměry r řádků a $n = 2r - 1$ sloupců. Pro připomenutí k je počet bitů původní informace (vektoru **a**), n je počet bitů kódového slova (vektor **b**) a r je počet redundantních bitů, o které je k rozšířeno na n. Pokud disponujeme pouze délkou původní informace k, můžeme potřebný počet bitů r určit nerovností $2^r \geq r + k + 1$. [2]

Kontrolní matici H_K pro různá r náleží konkrétní nejefektivnější hodnoty n a k udávající její rozměr dle tabulky 3.4. Od nich je vždy odvíjen název kódu, zapisuje se Hammingův kód HK(n, k). Chceme-li zabezpečit 4 informační bity původní zprávy, použijeme HK(7, 4), obdobně pro zabezpečení 11 informačních bitů použijeme HK(15, 11). Chceme-li zabezpečit počet informačních bitů, který není zastoupen v tabulce a stále využívat perfektní kód pro opravu jedné chyby, je možné zprávu doplnit nulovými bity na nejbližší tabulkové k.

Tabulka 3.4 Hodnoty n a k pro příslušný počet redundantních bitů r.

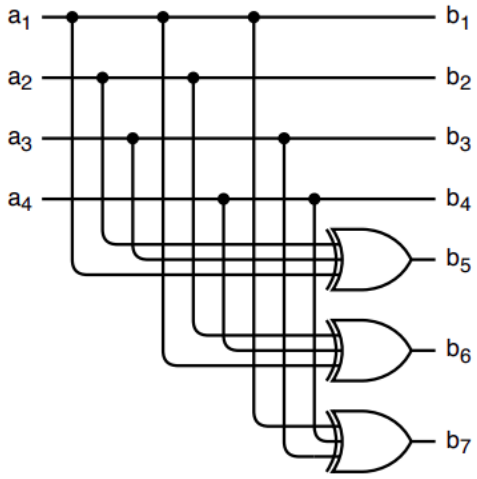
r	$n = 2^r - 1$	$k = 2^r - r - 1$
2	3	1
3	7	4
4	15	11
5	31	26
6	63	57
7	127	120
8	255	247
...	...	...

Nejčastější variantou Hammingova kódu je HK(7, 4). Generující matice pro tento kód je opět složena z jednotkové matice 4×4 a 3 sloupců redundantních bitů. Jejich množství odpovídá našim požadavkům na detekci a opravu jedné chyby ($kvzd = dch + och + 1 \rightarrow 3 = 1 + 1 + 1$). Počet sloupců tedy odpovídá délce kódových slov n a počet řádků délce zpráv k - $G_K^{4,7}$. K ní patřící kontrolní matice s množstvím sloupců také odpovídá délce kódových slov n, ale její počet řádků odráží, kolik jsme přidali redundantních bitů $r = n - k$ - $H_K^{3,7}$.

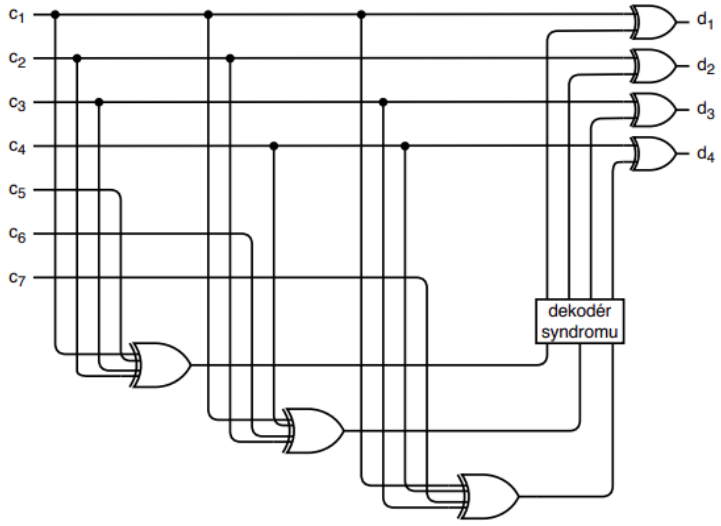
$$G_K = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$H_K = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Realizace kodéru zprávy a dekodéru syndromu přijatého slova dle výše uvedených matic G_K a H_K pro Hammingův kód HK(7, 4) na obrázcích 3.1 a 3.2.



Obrázek 3.1 Implementace generující matice jako kodéru pro HK(7, 4) - (převzato z [1, s. 32])



Obrázek 3.2 Implementace kontrolní matice k určení syndromu pro HK(7, 4) - (převzato z [1, s. 33])

3.4 Zkrácený Hammingův kód

Zkrácený Hammingův kód maximalizuje možnost detekce dvou chyb, není však již perfektním kódem. Pro zkrácený Hammingův kód je typické nedodržení hodnot n a k pro dané r. Obvykle se odvíjí od stanovené k-bitové zprávy k zakódování. Pokud její délka nevyhovuje žádné trojici v řádcích tabulky 3.4, požadované parametry dopočítáme přes vztahy $2^r \geq r + k + 1$ a $r = n - k$.

Pro zakódování 5-bitové zprávy, si lze do vztahu dosadit $2^r \geq r + 5 + 1$ a $r = n - 5$, z čehož můžeme určit $r = 4$ a $n = 9$. Proto vystavíme generující matici $G_K^{5,9}$ založenou na jednotkové matici 5×5, kterou na každém řádku doplňuje unikátní čtveřice redundantních bitů. Z ní pak určíme i kontrolní matici $H_K^{4,9}$. [1, s. 15]

$$G_K^{5,9} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$H_K^{4,9} = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

3.5 Rozšířený Hammingův kód

Bezpečnost Hammingova kódu si můžeme pojistit rozšířením o sudou paritu. K redundantním bitům tedy přidáme jeden paritní a tím zvedneme Hammingovu vzdálenost kódových slov ze 3 na 4. Umožní nám detekovat tři chyby, nebo dvě a jednu opravit. Pro vytvoření generující matice $G_K^{k,n+1}$ si stačí vzít její nerozšířenou obdobu $G_K^{\prime k,n}$ pro HK(n,k) a vynásobit jí maticí $G_K^{\prime\prime n,n+1}$, jež je tvořena jednotkovou maticí doplněnou o sloupec s každým bitem nenulovým. Kontrolní matici $H_K^{n+1-k,n+1}$ získáme převodem z té generující. [1, s. 16] [7, s. 19]

$$G_K^{k,n+1} = G_K^{\prime k,n} * G_K^{\prime\prime n,n+1}$$

$$G_K^{4,8} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} * \left(\begin{array}{ccccccc|c} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}\right) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$H_K^{4,8} = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

3.6 Cyklický kód

Cyklický kód je takový lineární kód, kde pro každé kódové slovo $(b_0, b_1, \ldots, b_{n-2}, b_{n-1})$ je $(b_{n-1}, b_0, b_1, \ldots, b_{n-2})$ také kódovým slovem.

Původní informace k zakódování **a** je interpretována jako mnohočlen $A(x)$ stupně menšího než k ($degA(x) < k$), a také kódové slovo **b** je interpretováno jako mnohočlen $B(x)$ stupně menšího než n ($degB(x) < n$). Obdobně se značí slovo **c** přijaté z kanálu - $C(x)$, chyba **e** superponovaná přenosem - $E(x)$, a z dekodéru výstupní informace **d** - $D(x)$.

Informace **a** = 010011 (zapisováno zleva od nejvýznamnějšího bitu) se pro cyklický kód přepíše jako mnohočlen $A(x) = 0 * x^5 + 1 * x^4 + 0 * x^3 + 0 * x^2 + 1 * x^1 + 1 * x^0 = x^4 + x + 1$, $degA(x) = 4$.

Mezi mnohočleny stupně menšího než n vzniká množina kódových slov kódu K. Platí, že jsou dané unikátní dvojice kódových slov a příslušných informačních mnohočlenů, tedy jedno kódové slovo není reprezentováno dvěma polynomy a naopak.

Na rozdíl od výše zmíněných kódů není logika kodéru a dekodéru nutně reprezentována maticí. Funkci kodéru vykonává generující mnohočlen $G(x)$, jež obsahuje absolutní člen pro zamezení násobnosti s x. Kódové slovo $B(x)$ vznikne ze vztahu

$$B(x) = A(x) * G(x).$$

Zdařilý přenos polynomu $C(x)$ do dekodéru bude mít nulový zbytek po dělení, nulový syndrom

$$S(x) = C(x)\%G(x).$$

Nenulový zbytek $S(x)$ (nenulový syndrom) značí, že se v kanálu na přenášené slovo $B(x)$ superponovala chyba $E(x)$,

$$C(x) = E(x) + B(x).$$

$E(x)$ popisuje shluk chyb, jeho stupeň značí délku shluku l. Dekódovaná informace vychází ze vztahu

$$D(x) = C(x)/G(x).$$

Známe-li hodnoty n a k, můžeme mnohočleny zapisovat pro přehlednost jako bitovou posloupnost od nejvýznamnějšího bitu. Tedy mnohočlen $A(x) = x^2 + 1$ pro $k = 4$ (v celém tvaru $A(x) = 0 * x^3 + 1 * x^2 + 0 * x^1 + 1$) se zapíše jako 0101. [2] [8]

Přehled maximálních stupňů mnohočlenů cyklického kódu je v tabulce 3.5.

Tabulka 3.5 Limity pro stupně mnohočlenů cyklického kódu.

Mnohočlen	Stupeň
$A(x)$, $D(x)$	$deg < k$
$B(x)$, $C(x)$, $E(x)$	$deg < n$
$G(x)$	$deg = r = n - k$
$S(x)$	$deg < r$

V rámci bezpečnostních kódů uvažujeme cyklické kódy, které jsou zároveň lineární kódy. Množina kódových slov je uzavřená na operacích cyklického posuvu a dále $\odot$ a $\oplus$ definovaných pro těleso T, na kterém konáme. Kód je lineární pokud lineární kombinace libovolných kódových slov dá vzniknout také kódovému slovu, a kód je cyklický pokud cyklickým posuvem na libovolném kódovém slově vznikne také kódové slovo. Ukázka možného kódování původní informace **a** cyklickým a necyklickým kódem následuje níže (tabulka 3.6). [3]

Tabulka 3.6 Možnosti zakódování původní informace **a** s lineárním cyklickým kódem K_1 a lineárním necyklickým kódemK_2.

Původní informace **a**	K_1	K_2
00	000	00000
01	011	01011
10	101	10100
11	110	11110

Cyklického kódu je možné dosáhnout i tradičnější metodou mezi lineárními kódy, a to sice generující maticí G_K. Tato matice musí být tvořena bází cyklického kódu, stačí tedy do každého řádku vložit generátor s posuvem, tak aby žádné dva řádky nebyly stejné a byly zastoupeny všechny možnosti posunu. Vhodnou generující maticí cyklického kódu (6, 3) je

$$G_K^{3,6} = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Přestože je metoda generování maticí dostupná, je v případě cyklických kódu nadbytečná, neboť stačí si vzít jediné kódové slovo a začít generovat. Z matice G_K výše si vezmeme její řádek 001001 jako generátor (toto slovo lze zapsat jako mnohočlen $G(x) = x^3 + 1$). Postup generování je shrnut v tabulce 3.7 a přehled kódování původní informace je v tabulce 3.8. [7, s. 21-22] [9, s. 192-193]

Tabulka 3.7 Generování cyklického kódu z jediného kódového slova 001001, toto slovo bylo vybráno jakožto nenulové slovo s nejnižším stupněm mnohočlenu.

	Kód K	Postup vzniku
1.	001001	1. kódové slovo a generátor
2.	010010	posuv 1. slova o jednu pozici
3.	100100	posuv 1. slova o dvě pozice
4.	011011	součet 1. a 2. slova
5.	110110	posuv 4. slova o jednu pozici
6.	101101	posuv 4. slova o dvě pozice
7.	111111	součet 3. a 4. slova
8.	000000	součet 7. slova sama se sebou

Tabulka 3.8 Přehled všech možností 3-bitové původní informace a jejího zakódování do 6-bitových kódových slov s $G(x) = x^3+1$. Hammingova vzdálenost tohoto kódu je $kvzd = 3$, což umožňuje detekovat jednu chybu.

	Kódové slovo	Příslušný mnohočlen	Původní informace	Zakódování
1.	001001	$x^3 + 1$	001	$G(x) * (1)$
2.	010010	$x^4 + x$	010	$G(x) * (x)$
3.	100100	$x^5 + x^2$	100	$G(x) * (x^2)$
4.	011011	$x^4 + x^3 + x + 1$	011	$G(x) * (x + 1)$
5.	110110	$x^5 + x^4 + x^2 + x$	110	$G(x) * (x^2 + x)$
6.	101101	$x^5 + x^3 + x^2 + 1$	101	$G(x) * (x^2 + 1)$
7.	111111	$x^5 + x^4 + x^3 + x^2 + x + 1$	111	$G(x) * (x^2 + x + 1)$
8.	000000	0	000	$G(x) * (0)$

3.7 Součinový kód

Součinový kód K vznikne součinem dvou kódů K_1 a K_2. K_1 kóduje k_1-bitové zprávy na n_1-bitová slova s $kvzd_1$, obdobně K_2 kóduje k_2-bitové zprávy na n_2-bitová slova s $kvzd_2$. Součinový kód K přijímá zprávy délky $k = k_1 * k_2$ a v kodéru je kóduje na slova délky $n = n_1 * n_2$. Jeho kódová vzdálenost je $kvzd = kvzd_1 * kvzd_2$.

Všechny úkony kódování a dekódování jsou založeny na opakovaném provádění těch samých úkonů pro kódy K_1 a K_2.

Původní informace **a** je k-bitový vektor, kódové slovo kódu K je n-bitový vektor **b**. Z kanálu přijímáno n-bitové slovo **c**, které dekódujeme na k-bitovou informaci **d**. Při přenosu se na slovo mohla superponovat n-bitová chyba **e**.

Pro zakódování zapíšeme informaci $\mathbf{a} = (a_0, \ldots, a_{k-1})$ po řádcích do matice $k_2 \times k_1$. Pro $K_1(5, 4)$ a $K_2(3, 2)$ se $(4*2)$-bitová zpráva **a** k zakódování zapíše do matice tvaru $k_2 \times k_1 = 2 \times 4$.

$$\begin{pmatrix} a_0 & a_1 & a_2 & a_3 \\ a_4 & a_5 & a_6 & a_7 \end{pmatrix}$$

Řádky této matice o délce k_1 budou nejprve zakódovány podle kódu K_1, obdobně budou druhotně sloupce o délce k_2 kódovány kódem K_2. Po zakódování řádků vznikne matice $k_2 \times n_1$.

Zakódováním sloupců vzniká již matice W tvaru $n_2 \times n_1$. Tato matice je tvořená z kódových slov K_1 zapsaných v řádcích a kódových slov K_2 zapsaných ve sloupcích. Kódové slovo $\mathbf{b} = (b_0, \ldots, b_{n-1})$ se zapisuje po řádcích. Pro $K_1(5,4)$ a $K_2(3,2)$ bude matice W tvaru $n_2 \times n_1 = 3 \times 5$ vyplněna $(3 * 5)$-bitovým kódovým slovem.

$$W = \begin{pmatrix} b_0 & b_1 & b_2 & b_3 & b_4 \\ b_5 & b_6 & b_7 & b_8 & b_9 \\ b_{10} & b_{11} & b_{12} & b_{13} & b_{14} \end{pmatrix}$$

Slovo **b** je kódovým slovem, pokud po jeho vepsání do matice W po řádcích platí, že její řádky jsou stále prvky K_1 a sloupce prvky K_2. Při dekódování přijaté slovo **c** zapíšeme do matice W po řádcích. Postupujeme tentokrát od K_2 ke K_1. Sloupce W dekódujeme podle K_2 a vytvoříme matici $k_2 \times n_1$. Řádky nově vzniklé matice dekódujeme podle K_1 a získáme matici tvaru $k_2 \times k_1$ kde je po řádcích vepsaná přijatá informace **d**. [6, s. 52-56] [2]

3.8 RM kód

Reed-Mullerovy (RM) kódy umožňují opravit libovolné množství chyb. Detekce a opravy chyb jsou snadno implementovatelné. Generující matice tohoto kódu je založená na principu prostých součinů.

Pro přirozené číslo μ a uspořádanou μ-tici $i_1, \ldots i_\mu$, kde ke každému $j \in (1, \ldots, \mu)$ máme $i_j \in \{0, 1\}$, existuje funkce $f(x_1, \ldots, x_\mu) = {x_1}^{i_1} \odot \cdots \odot {x_\mu}^{i_\mu}$ zobrazení $f : {\mathbb{Z}_2}^\mu \to \mathbb{Z}_2$ (ukázka v tabulce 3.9). Tento logický součin je nazýván prostý, neboť žádná z proměnných $x_1, \ldots, x_\mu$ není negována, jejich exponent tomu zamezuje. Počet exponentů $i_1, \ldots i_\mu$, které jsou nenulové, je nazýván řád l prostého součinu. Každá funkce prostého součinu pro μ proměnných má 2^μ možností uspořádaných μ-tic $i_1, \ldots i_\mu$.

Tabulka 3.9 Prostý součin pro $\mu = 3$ se všemi 8 možnostmi.

Řád l	i_1, i_2, i_3	$f(x_1, x_2, x_3) = {x_1}^{i_1} \odot {x_2}^{i_2} \odot {x_3}^{i_3}$
0	000	1
1	100, 010, 001	x_1, x_2, x_3
2	110, 101, 011	$x_1 \odot x_2$, $x_1 \odot x_3$, $x_2 \odot x_3$
3	111	$x_1 \odot x_2 \odot x_3$

Prostý součin $f(x_1, \ldots, x_\mu)$ můžeme zapsat binárně jako 2^μ-tici $(f_0, \ldots, f_{2^\mu - 1}) \in {\mathbb{Z}_2}^{2^\mu}$. Spodní index, dle zvyklosti psán v desítkové soustavě, značí ohodnocení proměnných $x_1, \ldots, x_\mu$. V binárním přepisu spodního indexu značí bit na 0. pozici hodnotu x_1 a bit na $(\mu-1)$. pozici značí hodnotu x_μ (ukázka v tabulce 3.10). Binární zápis prostého součinu o μ proměnných má délku 2^μ. Pro každý řád l existuje $\binom{\mu}{l}$ různých prostých součinů. [6, s. 28-29] [2]

Tabulka 3.10 Pro prostý součin $f(x_1, x_2, x_3) = x_1 \odot x_2 \odot x_3$ třetího řádu (tedy $i_1, i_2, i_3 = 1, 1, 1$) je binární zápis ve tvaru $(f_0, f_1, f_2, f_3, f_4, f_5, f_6, f_7) = (0, 0, 0, 0, 0, 0, 0, 1)$.

	$f(x_1, x_2, x_3)$	$x_1 \odot x_2 \odot x_3$
f_0	$f(0, 0, 0)$	0
f_1	$f(0, 0, 1)$	0
f_2	$f(0, 1, 0)$	0
f_3	$f(0, 1, 1)$	0
f_4	$f(1, 0, 0)$	0
f_5	$f(1, 0, 1)$	0
f_6	$f(1, 1, 0)$	0
f_7	$f(1, 1, 1)$	1

Generující matice kódu RM(ρ, μ), kde $\mu \geq \rho > 1$, je složena z ρ podmatic pro jednotlivé řády l prostého součinu. Tyto podmatice jsou tvořeny řádky binárních zápisů prostého součinu pro každou kombinaci $i_1, \ldots, i_\mu$ odpovídající příslušnému řádu l - řazení je lexikografické (od x_1 po x_μ). Rozměry ${}^lG$ jsou $\binom{\mu}{l} \times 2^\mu$. Složení podmatic jednotlivých řádů do jedné generující je tvaru

$$G = \begin{pmatrix} {}^0G \\ {}^1G \\ \vdots \\ {}^\rho G \end{pmatrix}.$$

■ **Tabulka 3.11** Kompletní výpis prostých součinů pro $\rho = 3$ a $\mu = 3$. Tato tabulka může posloužit i pro $\rho < 3$.

Řád l	Prosté součiny	Kombinace vstupů	Podmatice řádu
0	1 1 1 1 1 1 1 1	1	${}^0G$
1	0 1 0 1 0 1 0 1	x_1	${}^1G$
	0 0 1 1 0 0 1 1	x_2	
	0 0 0 0 1 1 1 1	x_3	
2	0 0 0 1 0 0 0 1	$x_1 \odot x_2$	${}^2G$
	0 0 0 0 0 1 0 1	$x_1 \odot x_3$	
	0 0 0 0 0 0 1 1	$x_2 \odot x_3$	
3	0 0 0 0 0 0 0 1	$x_1 \odot x_2 \odot x_3$	${}^3G$

Pro vytvoření generující matice RM$(1, 3)$ je vhodné si vypsat tabulku prostých součinů pro všechny řády a kombinace vstupu (ukázka v tabulce 3.11). Všechny binární zápisy prostých součinů pro daný řád l tvoří podmatici ${}^lG$ - jednotlivé možnosti vstupů vystupují jako samostatné řádky. Podmatice jsou skládány od nejnižšího řádu na prvním řádku.

Pro RM(1,3) je dle tabulky 3.11 generující matice tvaru

$$G = \begin{pmatrix} {}^0G \\ {}^1G \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

Délka kódového slova kódu RM(ρ, μ) je $n = 2^\mu$ a délka původní informace je $k = \sum_{l=0}^{\rho} \binom{\mu}{l}$. Kódová vzdálenost kvzd určuje počet chyb t, které lze opravit skrz vztahy $kvzd = 2^{\mu-\rho}$ a $t = \lfloor \frac{kvzd-1}{2} \rfloor$. [2]

Původní informace **a** o k bitech je zakódována na kódové slovo **b** o n bitech pomocí rovnice

$$b = a * G.$$

Pro RM(1,3) můžeme kódování rozepsat i takto

$$b = a * G = ({}^0a, {}^1a) * \begin{pmatrix} {}^0G \\ {}^1G \end{pmatrix} = ((a_0), (a_1, a_2, a_3)) * \begin{pmatrix} {}^0G \\ {}^1G \end{pmatrix}.$$

Dekódováním přijatého slova **c** z kanálu vznikne informace **d**. Dekódování probíhá majoritně (nejčastější výsledek bude vybrán), proto informaci rozdělíme do frakcí ${}^\rho d, ..., {}^0d$ jež odpovídají podmaticím ${}^\rho G, ..., {}^0G$. Opět pro RM(1,3) může mít přijatá informace rozdělená do frakcí tvar

$$d = ({}^0d, {}^1d) = (d_0, d_1, d_2, d_3)$$

Při dekódování frakce ${}^ld$, jejíž počet bitů je roven počtu řádků v příslušné podmatici ${}^lG$, odvodíme z generující matice celkem $2^{\mu-l}$ vzájemně nezávislých rovnic pro každý bit v ${}^ld$ (každou

neznámou). Každá taková rovnice je tvořena součtem 2^l bitů ze slova **c**. Níže uvedená ukázka soustavy rovnic je odvozena od generující matice RM(1,3) - první rovnice c_0 je odvozena od prvního sloupce, druhá rovnice c_1 je odvozena od druhého sloupce, pro ostatní rovnice je postup stejný. Ze soustavy osmi rovnic $c_0, \ldots, c_7$ je následně odvozena také soustava rovnic $d_0, \ldots, d_3$ (tato soustava je definována pro d_1, d_2, d_3 více vztahy, které nabydou důležitosti při zjišťování chyby).

$$\begin{aligned}
c_0 &= d_0 \\
c_1 &= d_0 + d_1 \\
c_2 &= d_0 + d_2 \\
c_3 &= d_0 + d_1 + d_2 \\
c_4 &= d_0 + d_3 \\
c_5 &= d_0 + d_1 + d_3 \\
c_6 &= d_0 + d_2 + d_3 \\
c_7 &= d_0 + d_1 + d_2 + d_3
\end{aligned}$$

$$\begin{aligned}
d_3 &= c_0 + c_4 = c_1 + c_5 = c_2 + c_4 = c_3 + c_7 \\
d_2 &= c_0 + c_3 = c_1 + c_3 = c_4 + c_6 = c_5 + c_7 \\
d_1 &= c_0 + c_1 = c_2 + c_3 = c_4 + c_5 = c_6 + c_7 \\
d_0 &= c_0
\end{aligned}$$

Pokud se na **c** nesuperponoval chybný vektor **e**, měly by tyto rovnice vždy dát stejný výsledek pro daný bit frakce ${}^l d$. Jestli se superponovalo na **c** nejvýše t chyb a máme různá řešení rovnic, bude vybrána taková hodnota bitu, která se vyskytuje nejčastěji. Z tohoto důvodu se tento postup nazývá majoritní dekódování.

Konečným krokem je odečtení násobku ${}^l d * {}^l G$ od přijatého slova **c**. Celý postup je sestupně opakován od ${}^\rho d$, dokud nebude zpracována frakce ${}^0 d$. [6, s. 31-34]

3.9 VHDL

VHDL (VHSIC Hardware Description Language, VHSIC = Very High Speed Integrated Circuits) je v informatice hardwarový popisový jazyk, který je využíván k návrhu a k simulaci integrovaných obvodů. Jazyk byl vytvořen s cílem jednotně dokumentovat a popisovat chování obvodů pro Ministerstvo obrany USA.

Soubor s koncovkou .vhd s obvodem popsaným ve VHDL lze vytvořit ve WM notebooku .nb či balíčku .wl. K vytvoření funkčního skriptu, který popisuje kodér, dekodér, testbench (skript pro testování funkčnosti kodéru a dekodéru), stačí VHDL prvky popsané níže.

Entita nepopisuje chování, figuruje jako jednotka neznámého obsahu, ke které pouze známe její vstupy a výstupy (porty) a funkčnost. S její pomocí definujeme menší celek pro použití ve větším projektu (např.: nechceme-li neustále tvořit 4-bitovou sčítačku, je možné ji vytvořit jen jednou jako architekturu konkrétní entity názvu ADDER, a tu pak jako komponentu opakovaně využívat). [10, s. 18-21, 23, 169]

Výpis kódu 3.1 VHDL entita

```
entity ENTITY is
    port(
        x, y    : in STD_LOGIC;
        z       : out STD_LOGIC
    );
end ENTITY;
```

Architektura definuje chování entity. Jedna entita může mít více interpretací chování skrz více architektur. Chování může být definováno behaviorálně (s vysokou úrovní abstrakce) či strukturálně (popis je hardwarově závislý). Prováděno pak může být sekvenčně nebo souběžně.

Výpis kódu 3.2 VHDL architektura

```
architecture ARCHITECTURE_BODY of ENTITY is
    -- signály
begin
    -- procesy nebo kontinuální rozkaz
end ARCHITECTURE_BODY;
```

V momentě, kdy existuje entita a k ní alespoň jedna architektura, jež definuje její chování, je možné ji použít v rámci většího obvodu jako komponentu.

Výpis kódu 3.3 VHDL komponenta

```
component ENTITY is
    port(
        x, y    : in STD_LOGIC;
        z       : out STD_LOGIC
    );
end component ENTITY;
```

3.10 Wolfram Mathematica

Wolfram Mathematica je počítačový program určený pro vykonávání výpočtů různé obtížnosti z oblasti matematiky, statistiky, techniky a dalších věd. Rozsah funkcí je velmi široký a neomezuje ani výpočty s parametry. Pro práci s bezpečnostními kódy je důležité, že program umožňuje pracovat s maticemi, vektory, polynomy, rovnicemi, jejich soustavami a mnoha dalšími entitami.

WM je používáno také jako vývojové prostředí pro práci v programovacím jazyce Wolfram Language. Tento jazyk začala v roce 1988 vyvíjet společnost Wolfram Research. Samotný výpočetní systém lze rozdělit na jádro a front end. Jádro (tzv. kernel) zodpovídá za vyhodnocování kódů a realizaci výpočtů. Front end pak zprostředkovává komunikaci s uživatelem. Výchozím pracovním prostředím WM je notebook (soubor s koncovkou .nb). V rámci notebooku je kód členěn dle sestupné hierarchie do buněk (tzv. cells). Vstup je uživatelem psán do vstupních buněk (In[*pořadové číslo*]:=*vstup*), při vyhodnocování se pod vstupními buňkami tvoří příslušné výstupní buňky (Out[*pořadové číslo*]=*výstup*) s výstupem z jádra. Jednotlivé buňky lze seskupovat v jednu i rozdělovat na více částí. [11]

Výpis kódu 3.4 Vstupní a výstupní buňky ve WM

```
In[1]:= var = 12.5 + 7
Out[1]= 19.5

In[1]:= Expand[(x^2 + 1)*(x^4 + x^3 + 1)]
Out[1]= 1 + x^2 + x^3 + x^4 + x^5 + x^6
```

Pro rozšíření funkcionalit lze využít balíčků (tzv. package, soubor s koncovkou .wl). Lze o nich uvažovat jako o obdobě knihoven v jiných programovacích jazycích (například C nebo C++). V balíčcích jsou definovány veřejné funkce, které po načtení onoho balíčku do notebooku, je možné s patřičnými parametry zavolat. Zároveň je možné definovat i soukromé funkce, které pro ty veřejné vykonávají dílčí úkony.

3.11 Vyhodnocení studijních požadavků studentů BI-JPO

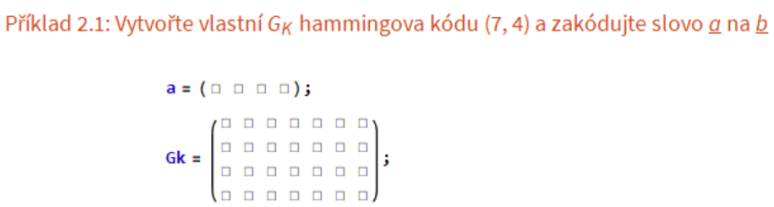
Během zimního semestru akademického roku 2022/2023 byl studentům předmětu BI-JPO na cvičení zaměřeném na lineární kódy předložen pracovní list s dotazníkem. Cílem tohoto průzkumu bylo zjistit, kde mají studenti největší problémy při využívaní kódů pro zabezpečení zpráv.

Pracovní list je zaměřen na tzv. perfektní kód pro opravu jedné chyby - na Hammingův kód. Pro realizaci pracovního listu byly využity tři WM balíčky Stanislava Koleníka z roku 2021: *HammingCode.wl*, *BasicCode.wl* a *CommonCode.wl*.

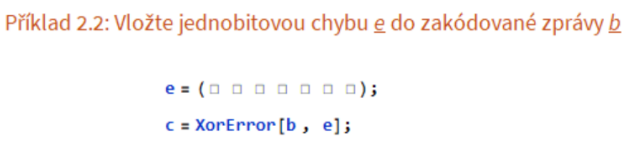
Balíček *HammingCode.wl* tvoří generující či kontrolní matici na základě validní dvojice parametrů n a k nebo parametru r. Pro neplatné parametry nelze požadované matice vytvořit. Balíček *BasicCode.wl* se nevěnuje Hammingovu kódu ani obecným funkčnostem kódování, nýbrž pokrývá paritní a repetiční kódy. Je vyžíván pro podpůrné funkce prvního zmíněného balíčku. Balíček *CommonCode.wl* nabízí funkce určení syndromu, konverze mezi H_K a G_K, aplikování chyby na kódové slovo, určení Hammingovy vzdálenosti a mnoho dalších. Utility tohoto souboru jsou k využití pro všechny úkony samotného procesu kódování, kromě zjištění generující a kontrolní logiky.

Pracovní list předložený studentům je realizován jako WM notebook ve stylu prezentace se stránkami. Pro jeho vyplnění není třeba žádných materiálů zvenčí, předpokládá se, že vyplňující si zcela vystačí se souhrnem teorie k obecnému lineárnímu kódování a teorie Hammingova kódu na prvních stránkách.

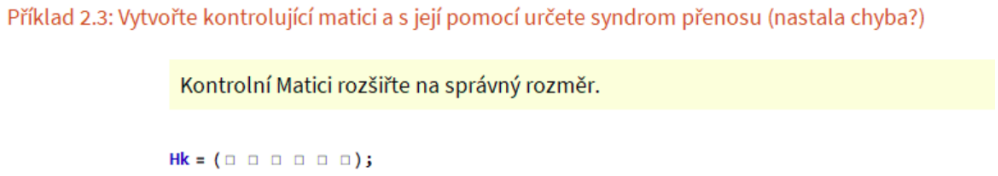
Celý pracovní list byl koncipován jako ilustrační nástroj, kde není konkrétní zadání ale ani konkrétní správná odpověď. Úkoly jsou založeny na tom, že vyplňující musí vytvořit validní generující matici, a s ní následně vhodně nakládat. Vlastní informaci je možné zakódovat, poškodit chybou, opravit a zjistit syndrom chyby, je-li vůbec nějaká přítomna. Volnou ruku vyplňujícímu dává připravená prázdná generující matice, kterou lze zvětšit či zmenšit, dle potřeby. Způsob zadání informace a a generující matice G_K pro příklad zaměřený na HK(7, 4) je v ukázce 3.3. Zadané hodnoty jsou zpracovány výpočtem ve vstupních buňkách (nejsou viditelné na ukázce), které musí vyplňující pouze spustit. V následující ukázce 3.4 je navazující příklad jež zadává vložení jednobitové chyby, tento požadavek pramení ze snahy získat nenulový syndrom. Výpočet, který na ukázce není viditelný je zcela funkční pro nulovou i vícebitovou chybu (nedojde k opravě, neboť HK opraví nanejvýš jednu chybu). Třetí ukázka 3.5 zadává vyplňujícímu výzvu v tvorbě kontrolní matice H_K. Její rozměr již nemůže být volitelný, ale musí doplňovat G_K. Způsob konverze generující matice na kontrolní je obsažen ve shrnutí teorie na začátku pracovního listu, a způsob rozšíření prázdné matice v prostředí WM je uveden pod příkladem.



Obrázek 3.3 Příklad z pracovního listu bez následného výpočtu - kódování informace a



Obrázek 3.4 Příklad z pracovního listu - superponování chyby e na slovo c



Obrázek 3.5 Příklad z pracovního listu - tvorba vhodné kontrolní matice

Cílem příkladů, jako jsou ty na ukázkách výše, je dát vyplňujícímu možnost vytvořit vlastní kód skrz G_K. Není-li kód validní, nebude výpočet vykonán díky funkcím z balíčku *Hamming-Code.wl*. Je-li však kód validní, stačí spustit všechny vstupní buňky a výsledek operací a mezikroků lze sledovat bez závislosti na ručních maticových výpočtech (které jsou náchylnější na selhání a pomalejší). Pokud není vyplňující příliš obeznámen s procesem kódování, tento typ příkladů je vhodným ilustračním nástrojem problematiky. Pro tři ukázky výše je kupříkladu vhodné měnit informaci a a chybu e a sledovat, jak se mění slova b a c.

Studenti po vyzkoušení pracovního listu vyplnili stručný dotazník, který měl za cíl zjistit, jakým směrem směřovali jejich problémy s touto tématikou.

Pracovní list s dotazníkem vyplnilo celkem 18 respondentů (studentů BI-JPO).

12 z 18 respondentů uvedlo, že jim tento čistě ilustrační přístup s podporou balíčku Stanislava Koleníka zcela vyhovoval a porozuměli probíranému tématu. 2 z 18 respondentů neposkytli konkluzivní odpověď (nebyli si jistí přínosem vyplnění pracovního listu), a zbylý 4 z 18 respondentů uvedli, že si stále nejsou jistí svými znalostmi v příkladech na Hammingův kód. (tabulka 3.12)

Na dotaz, zda předpokládají, že pracovní list, jehož cílem je ilustrovat téma Hammingových kódů, využijí při přípravě na zkoušku k zakončení předmětu BI-JPO, 13 z 18 respondentů uvedlo ANO, 4 z 18 respondentů uvedli NE, 1 z 18 respondentů uvedl, že by pracovní list opět využil, ale prostředí WM není pro jeho potřeby vhodné (nebyla poskytnuta konkrétní povaha problému s prostředím WM). (tabulka 3.13)

Tabulka 3.12 Výsledek dotazníku: Vyhovuje ilustrační přístup pracovního listu?

Respondentů	Pracovní list vyhovoval	Pracovní list nevyhovoval	Neurčitý závěr
Celkem 18	12	4	2

Tabulka 3.13 Výsledek dotazníku: Využijete pracovní list při přípravě na zkoušku?

Respondentů	ANO	NE	Nevyhovuje prostředí
Celkem 18	13	4	1

Kapitola 4

Návrh řešení

4.1 Generování VHDL skriptů

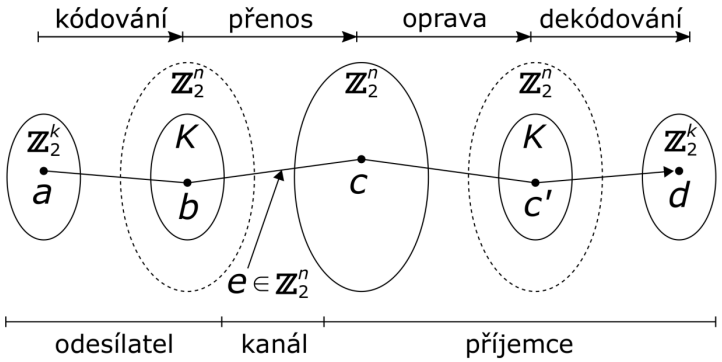
Generování VHDL skriptů bude realizováno pomocí Wolfram Mathematica balíčků, které zapouzdří proces kódování zvoleného kódu (obrázek 1.1) do kombinačního obvodu.

Výstupní VHDL skripty budou realizovat celý proces kódování. Tento proces vždy začne s informací **a**, kterou je třeba zakódovat na validní kódové slovo zvoleného kódu K. Tento úkon tedy bude potřebovat patřičnou generující logiku - tím lze rozumět například generující matici nebo generující mnohočlen. Z této logiky s jediným vstupem **a** bude výstupem hledané kódové slovo **b**. Informace **a** a slovo **b** tvoří unikátní dvojici mezi svými disjunktními množinami. Tato modifikace je označována jako průchod původní informace kodérem.

Zakódovaná informace **b** je následně podrobena podmínkám, kvůli kterým byla primárně modifikována. V rámci působení rušivých podmínek se může, ale nemusí, promítnout chyba na kódové slovo. Taková chyba může zasáhnout jeden až všechny bity slova a tím znehodnotit jeho validitu. Protože obvykle nejsme schopni s jistotou určit, zda slovo vystupující z rušivých podmínek bylo modifikováno, preemptivně jej označíme jako slovo **c**. Proces možné modifikace v důsledku rušivých podmínek je znám jako průchod slova informačním kanálem.

Po obdržení slova **c** z kanálu je nutné zjistit, zda takovou informaci smíme použít, zda je validní. Nelze ji jen převést na slovo z množiny původní informace **a**, protože bychom nikdy neměli jistotu, že pracujeme právě s **a**. Pokud nenastalo takové množství chyb s takovým rozložením, které by jedno kódové slovo změnilo na jiné kódové slovo stejného kódu, je možné určit, zda došlo ke změně. Z původní generující logiky, lze odvodit kontrolní logiku, která určí příznak chyby (syndrom). Příznak je nenulový, pokud se na slovo v kanálu superponovala chyba. Někdy je tato kontrola validity rozšířena o možnost opravy omezeného rozměru. Jisté kódy skrze svou kontrolní logiku vytvoří unikátní dvojice příznaků a konkrétních chyb. Máme-li znalost chyby, můžeme ji bezpečně odstranit a získat dekódovanou informaci **d**.

Podrobněji je kódování demonstrováno na ukázce 4.1, kde je výše popsaný proces interpretován jako série zobrazení. Informace z Z_2^k se bijektivně vyobrazí na podmnožinu (kódová slova) kódu K z Z_2^n (Z_2^k a Z_2^n jsou stejně velké množiny). Na kódové slovo se může promítnout chyba nabývající tvaru jakéhokoliv prvku množiny Z_2^n. Analogicky chybou zasažené slovo je uvažováno jako prvek z podmnožiny kódových slov ze Z_2^n, dokud se neprokáže jeho invalidita bez možnosti opravy.



Obrázek 4.1 Proces kódování jako zobrazení (převzato z [12, s. 9])

Celý proces kódování lze popsat jak sekvenčně, tedy v závislosti na řídících signálech, tak kombinačně, v závislosti na vstupu samotném.

4.1.1 Formát výstupních souborů

Před strukturováním generátoru je třeba vědět, co bude očekávaný výstup - soubory realizující celý proces kódování. Budeme tedy potřebovat soubor pro zakódování (encode), soubor pro vložení chyby (error) a soubor pro dekódování (decode). Všechny tyto části se budou muset odkazovat na parametry kódu, proto bude třeba, aby měli jednotně definované proměnné, k tomu se hodí vytvořit soubor jen pro konstanty (constants), který bude přikládán k projektu. Za účelem ověření funkčnosti doposud uvedených souborů se nám bude hodit soubor pro otestování (testbench nebo-li TB). Testovací soubor kontroluje proces kódování jako celek, tedy nevěnuje se pouze kódování či dekódování, ale všem částem procesu. Proto je třeba první tři soubory (encode, decode a error) zabalit do jednoho projektu pro další použití. K tomu bude stačit zastřešující soubor (top), který v sobě definuje propojení a vzájemné navázání tří částí. Soubor top je pak při simulaci v hierarchii zdrojových souborů roven souboru testbench.

Při generování bude pro jednotlivé VHDL skripty dodržována jmenná konvence podle tabulky 4.1.

Tabulka 4.1 Výstupní VHDL soubory pro kódy

Kód	Název VHDL souboru
kodér	*codeName*_encode.vhd
vložená chyby	*codeName*_error.vhd
dekodér	*codeName*_decode.vhd
konstanty	*codeName*_constants.vhd
zastřešující soubor	*codeName*_top.vhd
testovací soubor	*codeName*_TB.vhd

Skripta kodéru, vložení chyby a dekodéru budou realizována kombinačně - entita bude definována jedinou architekturou, která bude mít vždy jediný proces. Proces lze pojmenovat a určit mu citlivostní seznam (uveden v kulatých závorkách za klíčovým slovem *process*), který definuje skupinu signálů na jejichž změnu bude proces reagovat. Dojde-li ke změně předávané hodnoty na jakémkoliv signálu, který je uveden v onom seznamu, dojde k vykonání procesu samotného, pokud jsou signály neměnné, nic se nekoná.

Kodér bude realizován procesem ENCODE v entitě encode_*codeName*, jež přijímá vstup *message* a vrací výstup *code_word*. Proces reaguje na změnu signálu *message*.

Výpis kódu 4.1 Návrh procesu zakódování

```
-- this sample is based on an even parity encoding process
ENCODE : process (message)
begin
    -- Each bit of the output signal 'code_word' is defined separately.
    code_word(0) <= message(0);
    ...
    code_word(k) <= message(0) xor ... xor message(k-1);
end process ENCODE;
```

Soubor pro vložení chyby bude realizován procesem ER v entitě error_*codeName*, jež přijímá vstup *error* a *code_word_IN* a vrací výstup *code_word_OUT*. Proces reaguje na změnu signálů *error* a *code_word_IN*.

Výpis kódu 4.2 Návrh procesu vložení chyby

```
-- this sample is based on an even parity error process
ER : process (error, code_word_IN)
begin
    -- The XOR operation is performed bit by bit.
    -- Each bit of the output signal 'code_word_OUT' is defined separately.
    code_word_OUT(0) <= code_word_IN(0) xor error(0);
    ...
    code_word_OUT(k) <= code_word_IN(k) xor error(k);
end process ER;
```

Dekodér bude realizován procesem DECODE v entitě decode_*codeName*, jež přijímá vstup *code_word* a vrací výstup *message* a *syndrom*. Proces reaguje na změnu signálu *code_word*.

Výpis kódu 4.3 Návrh procesu dekódování bez opravy chyby

```
-- this sample is based on an even parity decoding process
-- the input is word received from the information channel
DECODE : process (code_word)
begin
    -- Each bit of the output signals 'message' and 'syndrom' is defined
    -- separately.
    message(k-1) <= code_word(k);
    ...
    syndrom <= code_word(0) xor ... xor code_word(k);
end process DECODE;
```

Výpis kódu 4.4 Návrh procesu opravy chyby bez dekódování

```
-- this sample is based on an cross parity decoding process
DECODE : process (code_word)
    -- variable s_syndrom,  s_code_word
begin
    s_syndrom(0) := code_word(n-1) xor ... xor code_word(0);
    ... -- definition of the remaining syndrome bits
    -- The detected syndrome indicates the control value for the case.
    -- Case will allow to invert the bit that the syndrome has flagged as
    -- erroneous.
    case s_syndrom is
        when "..." => s_code_word(0) := not(code_word(0));
        ... -- remaining options for syndrome values
        when others => null;
    end case;
end process DECODE;
```

4.1.2 Tvorba balíčků

Wolfram Mathematica balíčky umožňují definovat funkční celky, které jsou chráněné proti nesprávnému pořadí vyhodnocování vstupních buněk uživatelem v notebooku, zajišťují kontinuální vykonání úlohy bez vyrušení či opomenutí nějaké části, a jsou využitelné pro více různých projektů najednou. To je jejich hlavní výhoda v porovnání s uložením funkčních celků přímo v noteboocích. K jejich použití je třeba být obeznámen s jejich náplní a jak využít jejich nabídku. Po naimportování balíčku jsou uživateli v notebooku přístupné veřejné (public) funkce. Ty velmi často volají pomocné soukromé (private) funkce, které jsou však uživateli neviditelné a nemůže je přímo využít z prostředí notebooku.

Univerzální struktura pro každý z osmi balíčků osmi kódů bude obsahovat veřejnou funkci *codeName*VHDL[*parameters*], kde *codeName* bude nahrazeno názvem konkrétního kódu a *parameters* nahradí parametry definující požadovaný kód. Tato funkce bude jako jediná veřejná ze souboru všech funkcí generujících VHDL skripty. Každý skript bude generován právě jednou soukromou funkcí, která může volat další pomocné funkce, ale pouze v jejím těle bude docházet ke generování textového obsahu (přehled stěžejních funkcí pro generování skriptu je v tabulce 4.2). Výsledný text skriptu je na konci každé z této funkcí zapsán do souboru, který je buďto vytvořen nebo přepsán, existoval-li již dříve. Zavoláním *codeName*VHDL[*parameters*] budou volány také všechny soukromé funkce zastupující jednotlivé skripty a mnoho dalších podpůrných funkcí (mezi něž se budou řadit také veřejné funkce z naimportovaných balíčku Stanislava Koleníka z roku 2021, které zastupují logiku samotných kódů).

Tabulka 4.2 Veřejná funkce *codeName*VHDL a jí podřízené funkce pro generování jednotlivých skript

Kód	Status viditelnosti
*codeName*VHDL[*parameters*]	veřejná
*codeName*ConstantsVHDL[*parameters*]	soukromá
*codeName*EncodeVHDL[*parameters*]	soukromá
*codeName*ErrorVHDL[*parameters*]	soukromá
*codeName*DecodeVHDL[*parameters*]	soukromá
*codeName*TopVHDL[*parameters*]	soukromá
*codeName*TBVHDL[*parameters*]	soukromá

Pro každý z osmi kódů vznikne balíček (4.3), který bude obsahovat veřejné a soukromé funkce generující VHDL skripta.

Tabulka 4.3 Výstupní balíčky pro kódy

Kód	Název WM balíčku
Sudá parita	wsParity.wl
Křížová parita	wsCross.wl
Hammingův kód	wsHamming.wl
Zkrácený hammingův kód	wsShortened.wl
Rozšířený Hammingův kód	wsExtended.wl
Cyklický kód	wsCyclic.wl
Součinový kód	wsProduct.wl
RM kód	wsRM.wl

4.2 Generátor pracovního listu

Pracovní listy pro procvičení kódů budou zaměřené na výuku látky předmětu BI-JPO (Jednotky počítače). Jejich charakteristika by se mohla zakládat na pracovním listu, který byl předložen

studentům tohoto předmětu. Z dotazníku, který tento list následoval, lze zobecněně odvodit závěr, že čistě ilustrativní přístup k probírané látce pomohl pochopit kódování informací. Proto bude tento princip aplikován i zde.

Cílené použití pracovních listů je pro předmět BI-JPO, kde stěžejní látkou lineárních kódů je základní Hammingův kód. Pro jeho pochopení jsou v učebních materiálech předloženy vlastnosti jednodušších kódů, např.: sudé parity nebo křížové parity. Těmto kódům bude také věnována pozornost. Ostatní kódy nejsou po praktické stránce hluboce probírané.

Pracovní listy budou soustředěné na trojici kódů sudá parita, křížová parita a Hammingův kód. Nebudou obsahovat konkrétní zadání, pouze nabídnou šablonu pro každý aspekt procesu kódování (zakódování, aplikování chyby na kódové slovo, určení syndromu, a je-li to možné oprava a dekódování). Jejich formát bude umožňovat, aby se vyplňující vždy vrátil k předchozímu kroku, upravil zadání daného momentu a pokračoval ve výpočtu dále.

Jejich formát bude prezentace s více stránkami. Tento formát umožňuje zřetelné oddělení sekcí vysvětlující teorii, načítání balíčků a procvičení kódů. Prezentace nebudou nikterak dlouhé, jejich rozsah bude držen pod 6 stránek. Tato limitace je důsledkem požadavku možnosti rychlého procvičení kódu na co nejmenším množství příkladů s co největším efektem.

Nástrojem generování WM pracovních listů bude projekt C++ zvaný Generator. Tento projekt bude obsahovat třídu Generator, která nabídne tři veřejné metody: workSheetEvenParity(), workSheetCrossParity() a workSheetHamming().

Tvorba pracovních listů bude probíhat jako tvorba textového souboru, avšak přesto se bude jednat o WM notebook. Pracovní list bude přepsán nebo vytvořen, neexistuje-li doposud. Každá metoda pro generování pracovního listu pro jim přidělený kód vytvoří objekt fileOUT výstupního proudu std::ofstream. Do výstupního proudu bude možné zapisovat za použití operátoru pro výstup «. [13]

Kapitola 5
Řešení

Jako součást praktické části práce bylo vytvořeno osm Wolfram Mathematica balíčků pro osm kódů, jejichž implementaci se věnuje tato kapitola. Dále byl vytvořen C++ projekt, který obsahuje tři metody pro tvorbu tří pracovních listů k naučení či procvičení látky předmětu BI-JPO.

5.1 Obecná struktura balíčků

Balíčky jsou pojmenovány podle názvu kódu s předponou *ws* (označení pro pracovní pomůcku *work supplement*). Univerzální tvar jejich jmen je *wsCodeName*. Každý z osmi WM balíčků je realizován jako samostatný .wl soubor (wolfram language soubor), a jeho definice je zahájena, respektive ukončena klíčovými slovy BeginPackage[”*wsCodeName*“”], respektive EndPackage[]. Tyto dvě klíčová slova ohraničují jmenný prostor *wsCodeName*, který obsahuje všechny funkce definované pro balíček, vytvářející tak nový kontext. Uvnitř balíčku je běžné definovat také soukromou sekci, kterou lze zahájit, respektive ukončit klíčovými slovy Begin[„Private“”], respektive End[]. V jejich rozmezí se vytvoří další kontext vnořený do toho prvního. Při zahajování balíčku je možné importovat potřebné podpůrné balíčky přímo s BeginPackage[”*wsCodeName*“”,{”*importedPackage*“”}] vypsáním jejich jmen do uvozovek s apostrofou na konci ve složených závorkách. [14] [15]

V rámci soukromé sekce (alternativně *private section*) jsou definovány všechny funkce balíčku, soukromé či veřejné. Aby byla funkce definována v soukromé sekci a přesto byla přístupná mimo ohraničení balíčku a byla tedy veřejná, musí být svým názvem deklarována před zahájením sekce. Prostor pro deklaraci veřejných funkcí, který se prostírá mezi zahájením balíčku a zahájením soukromé sekce, se nazývá dokumentační sekce (alternativně *documentation section*).

Výpis kódu 5.1 Sekce viditelnosti .wl souborů

```
BeginPackage["wsCodeName`",{"importedPackage1`","importedPackage2`"}];

(* documentation section *)
(* declarations of public functions *)

Begin["`Private`"];

(* private section *)
(* definitions of both public and private functions *)

End[];
EndPackage[];
```

Všechny příkazy psané ve WM je povolené zakončit středníkem, který slouží jako oddělovač mezi výrazy. Jeho použití je dobrovolné, pokud jeho opomenutí neovlivní výpočet. Průběžné komentáře k psanému kódu jsou psány mezi dvojici hvězdiček uvnitř dvojice kulatých závorek.

Balíčky pro generování VHDL skriptů jednotlivých kódů nabízí pro uživatele vždy jen jednu veřejnou funkci, a to sice *codeName*VHDL. Při deklarování veřejné funkce v dokumentační sekci stačí pouze uvést její jméno. To je vhodné doplnit o zprávu o způsobu jejího využití. Zprávy jsou definovány symbolem :: (v dokumentaci WM je uváděn jako MessageName). Zprávu o použití funkce definuje zápis *codeName*VHDL::usage.

Výpis kódu 5.2 Ukázka deklarace veřejné funkce se zprávou o tvaru a způsobu využití

```
codeNameVHDL::usage=
"codeNameVHDL[parameter1, parameter2] is generating VHDL scripts.";
```

Definice funkcí v soukromé sekci jsou doplněny o parametry potřebné pro její zavolání. Všechny parametry jsou uvedené v hranatých závorkách a jejich názvy jsou následovány podtržítkem. To zajišťuje párování vzorů parametrů. Tělo funkce je k hlavičce přiřazeno symbolem := (v dokumentaci WM je uváděn jako SetDelayed). Tělo funkce je vyjma momentu zavolání drženo ve stavu nevyhodnoceno.

Výpis kódu 5.3 Ukázka definice veřejné funkce

```
codeNameVHDL[parameter1_, parameter2_]:= (* function body *)
```

Veřejná funkce *codeName*VHDL má vcelku neměnné tělo napříč všemi generátory. Prostřednictvím této funkce se postupně volají soukromé funkce jež tvoří jednotlivé části projektu, které dohromady tvoří funkční celek (částí projektu se zde rozumí předně jeden členský soubor nebo sada testovacích dat).

Výpis kódu 5.4 Definice funkce codeNameVHDL

```
codeNameVHDL[parameters_]:= Module[{},
    (codeName)ConstantsVHDL[parameters];
    (codeName)EncodeVHDL[parameters];
    (codeName)ErrorVHDL[parameters];
    (codeName)DecodeVHDL[parameters];
    (codeName)TopVHDL[parameters];
    (codeName)TBVHDL[parameters];
    (codeName)TestData[parameters];
]
```

První, třetí, pátá a šestá volaná funkce z *codeName*VHDL - *codeName*ConstantsVHDL, *codeName*ErrorVHDL, *codeName*TopVHDL a *codeName*TBVHDL - generují skripta, které jsou si velmi podobné s ekvivalentními funkcemi jiných balíčků, nezáleží u nich na tom k jakému kódu se vztahují. Jejich vzájemná podoba umožňuje věnovat pozornost jejich účelu bez upřesnění kódu. Proto je řešení výše uvedených čtyř funkcí a skript jimi generovaných rozebráno obecně.

Druhá funkce *codeName*EncodeVHDL a čtvrtá funkce *codeName*DecodeVHDL jsou závislé na konkrétním kódů a jeho logice, neboť zprostředkovávají zakódování a dekódování daným kódem. Jsou napříč kódy velmi variabilní a proto jsou podrobněji rozebrány níže pro každý kód zvlášť. Využívají balíčky Stanislava Koleníka 2021 jako zdroj tvorby logických struktur.

Sedmá funkce *codeName*TestData generující testovací data se nijak nepodílí na funkčnosti projektu, pouze usnadňuje simulaci. Její postup je podrobně rozebrán níže.

5.2 Tvorba a modifikace výstupního souboru

Zápis textových řetězců do výstupních souborů je v každé funkci každého balíčku realizován pomocí výstupního proudu do vytvořeného či otevřeného souboru.

Pro jednotlivé kódy jsou definovány proměnné držící textové řetězce jmen souborů. Ty označují o jaký kód se jedná a jakou část procesu kódování zastupuje. Počet proměnných odpovídá obvyklému počtu využívaných souborů při simulaci kódu K. Vedle souborů zajišťující zakódování, vložení chyby, dekódování, zastřešení, jednotnou definici konstant a testování jsou k této sadě očekávaných souborů přidány ještě soubory pro usnadnění onoho testování - jeden se vstupními hodnotami (náhodné vektory délky k), jeden s chybovými hodnotami (jedná se vektory délky n, které jsou nulové nebo mají jediný kladný bit) a v některých případech jeden s očekávanými syndromy pro samostatnou kontrolu uživatelem.

Výpis kódu 5.5 Proměnné zastoupené v každém balíčku držící jména proměnných

```
(* file names variables *)
constantsFileName = "(codeName)_constants.vhd";
encodeFileName = "(codeName)_encode.vhd";
errorFileName = "(codeName)_error.vhd";
decodeFileName = "(codeName)_decode.vhd";
topFileName = "(codeName)_top.vhd";
TBFileName = "(codeName)_TB.vhd";
inputDataFileName = "(codeName)_TB_inputs.vec";
errorDataFileName = "(codeName)_TB_errors.vec";
syndromDataFileName = "(codeName)_TB_syndroms.vec";
```

Při tvorbě či otevírání cílového souboru je nutné znát jeho celé jméno. To kromě přehledu výše uvedených jmen obsahuje i absolutní cestu k souboru. Jako cesta k souboru bude využita absolutní cesta k notebooku, ve kterém bude zavolána funkce *codeName*VHDL. Výstupní soubory budou vždy generovány v lokaci, odkud byly zavolány. Celé jméno *FileName* pro generovaný soubor získáme spojením výstupního textového řetězce z NotebookDirectory[] udávajícího lokaci adresáře, z něhož byl zavolán, a patřičné proměnné s držící název skriptu. Pro spojení je nejvýhodnější použít další funkci FileNameJoin[]. S hotovým jménem je již možné vytvořit objekt výstupního proudu *FileStream* zavoláním OpenWrite[].

Výpis kódu 5.6 Otevření výstupního proudu pro zápis do souboru konstant

```
FileName = FileNameJoin[{NotebookDirectory[], constantsFileName}];
FileStream = OpenWrite[FileName,PageWidth->Infinity,FormatType->OutputForm];
```

Je-li soubor již otevřen, stačí připravit textový řetězec, který do něj vložíme. Všechny balíčky zapisují výstupy do souborů pouze jednou na konci patřičných funkcí, kdy je soubor napsán proudem z textového řetězce v proměnné.

Textový řetězec nesmí být v průběhu své tvorby nikdy přepsán, jinak hrozí poškození kvality funkčnosti VHDL skriptu. Kvůli tomu jsou průběžné podřetězce přidávány zřetězením. Toho je docíleno použitím symbolu <> (v dokumentaci WM je uváděn jako StringJoin) mezi řetězci, které chceme spojit. Tento způsob je limitován skutečností, že může zřetězit pouze textové řetězce a každá proměnná jiného typu musí být nejprve konvertována funkcí ToString[]. Ukázka níže demonstruje možnosti při kombinování podřetězců do výstupu "alfa beta 20".

Výpis kódu 5.7 Ukázka zřetězení textu

```
text = "alfa ";
text = text <> " beta " <> ToString[20];
```

Dokončený textový řetězec je následně zapsán do již dříve vytvořeného objektu výstupního proudu *FileStream* funkcí Write[]. Po zapsání je proud uzavřen funkcí Close[]. [16]

Výpis kódu 5.8 Zapsání do výstupního proudu z proměnné constantsTEXT a jeho uzavření

```
Write[FileStream, constantsTEXT];
Close[FileStream];
```

5.3 Hlavička VHDL souborů

Soubory obsahující VHDL skript se neobejdou bez knihovny. Knihovna je soubor již kompilovaných funkčních jednotek (architektury, entity, balíčky, konfigurace). Fyzicky existují jako adresáře, na něž je odkazováno názvem knihovny. Její obsah je použit klíčovým slovem *use*. Koncovka *.ALL* dává pokyn, aby byl k dispozici celý obsah. V rámci rozsahu kombinačně definovaného kódování je pro každý kód dostačující *IEEE.STD_LOGIC_1164* (např.: základní datové struktury), a *IEEE.NUMERIC_STD* (např.: početní operace).

V hlavičce každého souboru, který tvoří aktivní část projektu, je

Výpis kódu 5.9 Zpřístupnění obsahu knihovny IEEE skriptu

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
```

5.4 Definování konstant

První volaná funkce z veřejné funkce *codeName*VHDL je vždy *codeName*ConstantsVHDL, jež sjednotí parametry kódu do proměnných, které jej následně budou definovat. Tyto parametry se týkají především délky bitových či logických vektorů. Výstupním souborem této funkce je VHDL package. VHDL package je jednotka, která sdružuje skupinu funkcí, metod, proměnných, konstant a mnoha dalších. Pro naše potřeby sdružuje pouze konstanty definující délku původní informace, délku kódového slova.

Výpis kódu 5.10 Tvar VHDL souboru pro definici konstant HK(7, 4)

```
package CONSTANTS is
        constant k : integer := 4;
        constant n : integer := 7;
end package;
```

Není to aktivní článek kódu a pro jeho využití ve skriptech musí být načten v hlavičce kódu.

Výpis kódu 5.11 Zpřístupnění obsahu package

```
use work.constants.all;
```

5.5 Vložení chyby

Třetí volaná funkce z veřejné funkce *codeName*VHDL je vždy *codeName*ErrorVHDL. Během jejího vyhodnocování je vytvořen soubor pro vložení chyby. Přestože je toto velká součást procesu kódování, je její řešení jednoduché a zcela uniformní bez ohledu na typ kódu. Skript vzniklý touto funkcí pracuje se dvěma vstupními logickými vektory, na nichž provede operaci XOR, bit po bitu, a výstup uloží do výstupního logického vektoru. Při generování textového řetězce, který reprezentuje obsah skriptu, je pro opakovaný výpis operace XOR použit výpis ve smyčce s funkcí For[] a zřetězením textových řetězců <>.

Výpis kódu 5.12 Zřetězení textu souboru pro vložení chyby ve smyčce

```
For[i = 0, i < n, i++,
    errorTEXT = errorTEXT<>
    "code_word_OUT("<>ToString[i]<>")
    <= code_word_IN("<>ToString[i]<>") xor error("<>ToString[i]<>");
"];
```

Výpis kódu 5.13 Tvar VHDL souboru pro vložení chyby pro n = 7

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.constants.all;

entity error_codeName is
    port (
        error           : in    std_logic_vector((n - 1) downto 0);
        code_word_IN    : in    std_logic_vector((n - 1) downto 0);
        code_word_OUT   : out   std_logic_vector((n - 1) downto 0)
    );
end error_hamming;

architecture error_codeName_BODY of error_codeName is
begin
    ER : process (error, code_word_IN)
    begin
        code_word_OUT(0) <= code_word_IN(0) xor error(0);
        code_word_OUT(1) <= code_word_IN(1) xor error(1);
        code_word_OUT(2) <= code_word_IN(2) xor error(2);
        code_word_OUT(3) <= code_word_IN(3) xor error(3);
        code_word_OUT(4) <= code_word_IN(4) xor error(4);
        code_word_OUT(5) <= code_word_IN(5) xor error(5);
        code_word_OUT(6) <= code_word_IN(6) xor error(6);
    end process ER;
end error_codeName_BODY;
```

5.6 Sjednocení procesu kódování

Pátou volanou funkcí z veřejné funkce *codeName*VHDL je *codeName*TopVHDL. Tato funkce tvoří VHDL skript popisující TOP entitu, která zastřešuje a propojuje komponenty pro zakódování, vložení chyby a dekódování. Její definice je uniformní a je vždy generována jako celistvý text bez nutnosti úprav vztahujících se ke konkrétním parametrům. Na entity jiných VHDL celků je zde nahlíženo jako na komponenty, recyklovatelné kousky kódu, jejichž porty jsou namapovány mezi sebou, mezi pomocnými signály, nebo mezi porty zastřešující entity, to vše v instancích procesů komponent (EN_INST, ER_INST, DE_INST).

Namapování tvoří řetězec, kdy vstup TOP entity je vstupem komponenty zakódování. Výstup první komponenty je vstupem druhé komponenty vložení chyby. Výstup druhé komponenty je vstup poslední komponenty dekódování. Výstup TOP entity je zároveň výstup třetí komponenty. Pro propojení mezi komponentami jsou použity signály. Propojení krajních komponent s entitou využívá její rozhraní.

Top entita reprezentuje kódování jako celek, proto má vždy pouze dva vstupy. Prvním je informace k zakódování (message_IN), druhým je chyba (error). Prvním výstupem je dekódovaná informace (message_OUT). Ta může být defektní, opravená nebo nepoškozená. Druhým výstupem je příznak chyby (syndrom). Pokud došlo k chybě během kódování, bude nenulovou hodnotu. Jakmile je zjištěn syndrom přijatého slova do dekodéru, jeho hodnota zůstane neměnná, bez ohledu na možnou opravu chyby. Soubory pro dekódování jsou zaměřené na opravu nanejvýš jedné chyby, a to pouze v případě, že se jedná o sebeopravný a nikoliv detekční kód. Kdyby se v informačním kanále projevilo více chyb, není možné je všechny opravit. Tento limit je stanoven, neboť výstup této bakalářské práce je určen jako studijní materiál, kde jednobitová chyba je nejjednodušší ilustrací problému.

Výpis kódu 5.14 Tvar VHDL zastřešující soubor pro kodér, dekodér a soubor vložení chyby

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.constants.all;

entity top_codeName is
    port (
        message_IN     : in     std_logic_vector((k - 1) downto 0);
        error          : in     std_logic_vector((n - 1) downto 0);
        message_OUT    : out    std_logic_vector((k - 1) downto 0);
        syndrom        : out    std_logic_vector((n - k - 1) downto 0)
    );
end top_codeName;

architecture top_codeName_BODY of top_codeName is
    component encode_codeName is
        port (
            message     : in    std_logic_vector((k - 1) downto 0);
            code_word   : out   std_logic_vector((n - 1) downto 0)
        );
    end component encode_codeName;
    component error_codeName is
        port (
            error           : in    std_logic_vector((n - 1) downto 0);
            code_word_IN    : in    std_logic_vector((n - 1) downto 0);
            code_word_OUT   : out   std_logic_vector((n - 1) downto 0)
        );
    end component error_codeName;
    component decode_codeName is
        port (
            code_word   : in  std_logic_vector((n - 1) downto 0);
            message     : out std_logic_vector((k - 1) downto 0);
            syndrom     : out std_logic_vector((n - k - 1) downto 0)
        );
    end component decode_codeName;
    signal  code_word_EN, code_word_DE : std_logic_vector((n - 1) downto 0);
begin
    EN_INST : encode_codeName port map (
        message => message_IN,
        code_word => code_word_EN
    );
    ER_INST : error_codeName port map (
        error => error,
        code_word_IN => code_word_EN,
        code_word_OUT => code_word_DE
    );
    DE_INST : decode_codeName port map (
        code_word => code_word_DE,
        message => message_OUT,
        syndrom => syndrom
    );
end top_codeName_BODY;
```

5.7 Testovací soubor

Šestou volanou funkcí z veřejné funkce *codeName*VHDL je *codeName*TBVHDL, která generuje soubor pro testování zvaný testbench (často označován zkratkou TB). Tento soubor testuje funkčnost kodéru, souboru vložení chyby a dekodéru najednou. Aby to bylo možné, byl již vytvořen soubor *codeName*TopVHDL, který realizuje spojení tří entit v roli komponent do jedné entity. Nad tou je možné spustit testování.

Testování je doplněno o testovací data, které jsou generována při vyhodnocování funkce *codeName*TestData, uložená ve dvou, respektive ve třech samostatných souborech ve stejném adresáři jako všechna ostatní nově vygenerovaná VHDL skripta. Při simulaci bude nutné k testovacím datům přistoupit a přečíst. Proto ve skriptu musí být uveden celý název souboru i s absolutní cestou. Umístění lze snadno získat s funkcí NotebookDirectory[]. Pro uživatele operačního systému Windows však může nastat problém s lomítky v cestě k souboru, protože prostředí Windows na rozdíl od většiny ostatních operačních systémů používá v určení lokace zpětná lomítka. Ta mohou pro mnoho simulátorů představovat problém a proto je vhodné si cestu k souboru nejprve modifikovat a s funkcí StringReplace[] zpětná lomítko změnit na klasická. Zpětné lomítko má speciální funkci - rušení vlastnosti. Proto musí být uvedeno dvakrát za sebou, aby mohlo být přečteno jako pouhý znak.

Výpis kódu 5.15 Změna lomítek v cestě k souboru

```
location = ToString[NotebookDirectory[]];
location = StringReplace[location,"\\"->"/"];
```

Po získání cesty k souboru s testovacími daty a jejím zřetězení s obsahem proměnné držící název výstupního souboru, je možné tento variabilní článek zřetězit také se zbytkem textového souboru. Další úpravy relativní ke konkrétnímu kódu - rozdílné definice délky logických vektorů a podobně - jsou neměnné a v balíčku již zadané bez nutnosti aktualizace výpočtu při každém vyhodnocení funkce *codeName*TBVHDL.

Testbench je definován entitou bez rozhraní. Nedochází ke komunikaci s většími celky a je proto zbytečné. Testovací proces pracuje s komponentou TOP entity. Na její rozhraní mapuje signály v instanci procesu DUT (*Design Under Test*, tedy testovaný návrh). těmto signálům bude připisovat hodnoty získané čtením vstupních dat v procesu STIMULI_GEN. V tom jsou postupně zpracovávány vstupy z testovacích. Do proměnných jsou načítány celé řádky, dokud soubory nejsou celé přečtené. Načtené řádky jsou následně interpretovány jako bitové vektory, a ty jsou okamžitě reinterpretovány jako logické vektory. Získané logické vektory je již možné přiřadit k signálům namapovaným na testovaný návrh.

Proces testování je zakončen kontrolou nenulového syndromu. Pokud bylo z informačního kanálu přijato slovo obsahující chybu, bude proveden na konzoli simulátoru výpis ohlašující chybu a původně očekávanou hodnotu message_IN. Syndrom se i přes případnou opravu slova nemění, proto k výpisu dojde i v případě, že slovo bylo korektně opraveno.

Výpis kódu 5.16 Tvar VHDL testovacího souboru

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use STD.TEXTIO.all;
use work.constants.all;

entity TB_codeName is
end TB_codeName;

architecture TB_codeName_BODY of TB_codeName is
    component top_codeName is
```

```
        port (
            message_IN     : in     std_logic_vector((k - 1) downto 0);
            error          : in     std_logic_vector((n - 1) downto 0);
            message_OUT    : out    std_logic_vector((k - 1) downto 0);
            syndrom        : out    std_logic_vector((n - k - 1) downto 0)
        );
    end component top_codeName;
    signal TB_message_IN, TB_message_OUT : std_logic_vector((k - 1) downto 0);
    signal TB_error : std_logic_vector((n - 1) downto 0);
    signal TB_syndrom : std_logic_vector((n - k - 1) downto 0);
begin
    DUT : top_codeName port map (
        message_IN => TB_message_IN,
        error => TB_error,
        message_OUT => TB_message_OUT,
        syndrom => TB_syndrom
    );
    STIMULI_GEN : process
        file INPUTS     :  TEXT is in " location/codeName_TB_inputs.vec";
        file ERRORS     :  TEXT is in " location/codeName_TB_errors.vec";
        file SYNDROMS   :  TEXT is in " location/codeName_TB_syndroms.vec";
        variable LINE_input, LINE_error, LINE_syndrom : LINE;
        variable in_BV : BIT_VECTOR ((k - 1) downto 0);
        variable er_BV : BIT_VECTOR ((n - 1) downto 0);
        variable sy_BV : BIT_VECTOR ((n - k - 1) downto 0);
        variable in_LV : STD_LOGIC_VECTOR ((k - 1) downto 0);
        variable er_LV : STD_LOGIC_VECTOR ((n - 1) downto 0);
        variable sy_LV : STD_LOGIC_VECTOR ((n - k - 1) downto 0);
    begin

        while not ENDFILE(INPUTS) loop
            readline(INPUTS, LINE_input);
            readline(ERRORS, LINE_error);
            readline(SYNDROMS, LINE_syndrom);

            read(LINE_input, in_BV);
            read(LINE_error, er_BV);
            read(LINE_syndrom, sy_BV);
            in_LV := To_StdLogicVector(in_BV);
            er_LV := To_StdLogicVector(er_BV);
            sy_LV := To_StdLogicVector(sy_BV);

            TB_message_IN ((k - 1) downto 0) <= in_LV((k - 1) downto 0);
            TB_error ((n - 1) downto 0) <= er_LV((n - 1) downto 0);

            if not ENDFILE(INPUTS) then
                wait for 20 ns;
            end if;

            assert to_integer(unsigned(TB_syndrom)) /= 0
            report " expected="
                & integer'image(TO_INTEGER(UNSIGNED(TB_message_IN)))
                severity error;
        end loop;
        report "  **  SIMULATION ENDED  **  " severity failure;
    end process STIMULI_GEN;
end TB_codeName_BODY;
```

5.8 Testovací data

Pro efektivnější simulaci jsou kromě VHDL skriptů generovány také dva, respektive tři soubory s testovacími daty - *codeName*_TB_inputs.vec, *codeName*_TB_errors.vec a občas také *codeName*_TB_syndroms.vec. Tyto soubory vznikají hromadně v těle soukromé funkce *codeName*TestData, která je volána z veřejné funkce *codeName*VHDL. Množství generovaných dat je nastaveno na 10 simulačních vstupů (počet generovaných vstupů je nastaven v proměnné testDataAmount) a není možné to modifikovat bez editování balíčku.

Ve smyčce se při každém z 10 průchodů zavolá funkce RandomBinaryWord[] z knihovny *CommonCode*, jež vygeneruje konkrétní bitový vektor o zadané délce k k zakódování. K ní se následně vygeneruje bitový vektor chyby o zadané délce n, který má jediný nebo žádný kladný bit. Z deseti průchodů smyčkou se při každém lichém průchodu vygeneruje nulová chyba, která nijak neovlivní informaci v informačním kanálu, a při každém sudém průchodu se vygeneruje nenulová chyba.

Vstupy a chyby jsou generovány a syndromy dopočítávány jako bitové vektory. K tomu náleží pomocná soukromá funkce ClearString[], která obdrží vektor a interpretuje jej jako textový řetězec. Ten je s funkcí StringDelete[] postupně redukován pouze na znaky 1 a 0, které jsou vráceny volající funkci. Vrácené textové řetězce tvořené sekvencí jedniček a nul jsou následně zřetězené s výstupním řetězcem, který je nahrán na výstupního proudu testovacích dat. Redukce nadbytečných znaků je nutná pro simulaci, která data interpretuje jako bitové vektory, a jiné znaky obsažené ve čtených vstupech by nemuseli být zpracovány korektně. Funkce ClearString[] je využívaná v každém balíčku bez jakékoliv modifikace.

Výpis kódu 5.17 Funkce ClearString pro redukci nadbytečných znaků

```
ClearString[vec_] := Module[{},
    src=ToString[vec];
    src=StringDelete[src,","];
    src=StringDelete[src," "];
    src=StringDelete[src,"{"];
    src=StringDelete[src,"}"];
    src
]
```

5.9 Kodéry a dekodéry

Všechny generující balíčky se v nemalém rozsahu spoléhají na nabídku balíčku *CommonCode*. Především při zajišťování logiky jednotlivých kódů, balíček funkcí nabízí široký rozsah operací s vektory a s maticemi, jenž jsou specifické pro práci s kódy. Využívá-li generující balíček další balíčky Stanislava Koleníka z roku 2021, jsou uvedeny u konkrétního kódu.

5.9.1 Sudá parita

Sudá parita využívá balíček *BasicCode*. Paritní kód je definován jediným parametrem a to délkou původní informace, jež se značí k. Paritní kód pouze požaduje, aby počet informačních bitů informace k zakódování byl nenulový a kladný. Požadavek je kontrolován na počátku veřejné funkce podmínkou, která, není-li splněna, ukončí vyhodnocování funkce a vypíše zprávu s popisem, jaký formát má vstup mít.

Obdobně, jako bylo uvedeno v kapitole Návrh řešení, definuje každý bit kódového slova samostatně. Generování textových podřetězců probíhá pomocí dvou smyček. V první se přepíše zpráva na kódové slovo, a v druhé se vypíše na poslední nedefinovaný bit výsledek operace XOR mezi všemi bity původní zprávy.

Výpis kódu 5.18 Kodér sudé parity - definování bitů pro k = 3

```
code_word(3) <= message(2);
code_word(2) <= message(1);
code_word(1) <= message(0);
code_word(0) <= message(0) xor message(1) xor message(2);
```

Při dekódování se postupuje analogicky, kromě jednoho bitu přijatého slova se ostatní bity přepíšou na zprávu. Následně se určí hodnota syndromu pomocí operace XOR mezi všemi bity přijatého slova. Paritní kód neumožňuje provést opravu chyby, proto je zde pouze zjištěn syndrom pro validaci zprávy.

Výpis kódu 5.19 Dekodér sudé parity - definování bitů a syndromu pro k = 3

```
message(2) <= code_word(3);
message(1) <= code_word(2);
message(0) <= code_word(1);
syndrom <= code_word(0) xor code_word(1) xor code_word(2);
```

5.9.2 Křížová parita

Obdobně jako sudá parita i druhý paritní kód využívá balíček *BasicCode*. A stejně jako předchozí kód, je i křížová parita definována jediným parametrem k, délkou původní informace. Tato hodnota je očekávána nenulová a kladná. Proto po zavolání veřejné funkce je zkontrolována tato podmínka. V případě jejího nenaplnění, je vyhodnocování ukončeno a je vypsána zpráva obsahující požadovaný formát vstupu.

Pro zakódování je využitá funkce CrossParityG[k], která jako parametr přijímá právě k. Tato funkce vrací generující matici. Proto definování jednotlivých bitů musí být kombinační zápis maticového násobení informačního vektoru a generující matice. To je docíleno smyčkou, jež iteruje po sloupcích matice. V ní je vnořena další smyčka, jež iteruje po řádcích. Prostředí WM indexuje vždy od jedničky, proto jsou indexy o jednotku posunuty. Poslední je kontrola, zda prvek na právě procházené pozici generující matice je kladný. Pokud ano, bude daný prvek přidán do definice pro právě probíraný bit kódového slova.

Výpis kódu 5.20 Vnořené smyčky tisknoucí výsledek maticového násobení

```
GK = CrossParityG[k];

For[i = 1, i < n + 1, i++,
    ...
        For[j = 1, j < k + 1, j++,
                If[GK[[j,i]] == 1, ... , ... ]
        ];
];
```

Při dekódování se opět určí generující matice s funkcí CrossParityG[k], tentokrát se však k ní ještě určí kontrolní matice s pomocí funkce ConvertToH[G_K], která jako parametr přijme již vytvořenou generující matici. Pro určení syndromu chyby se používá stejná konstrukce dvou vnořených smyček s kontrolou, zda právě kontrolovaný bit kontrolní matice je kladný. Se zjištěným syndromem nastává možnost opravit chybu. Syndrom je porovnáván se sloupci kontrolní matice. Je-li nalezena shoda, bude invertován ten bit přijatého slova, kterému sloupci kontrolní matice byl syndrom roven. Toto porovnávání je generováno v samostatné smyčce.

Výpis kódu 5.21 Dekódování přijaté zprávy určením syndromu a invertováním postiženého bitu

```
s_syndrom(3):=code_word(6) xor code_word(5) xor code_word(4) xor code_word(3);
s_syndrom(2):=code_word(6) xor code_word(2);
```

```
s_syndrom(1):=code_word(5) xor code_word(1);
s_syndrom(0):=code_word(4) xor code_word(0);

s_code_word := code_word;

case s_syndrom is
    when "1100" => s_code_word(6) := not(code_word(6));
    when "1010" => s_code_word(5) := not(code_word(5));
    when "1001" => s_code_word(4) := not(code_word(4));
    when "1000" => s_code_word(3) := not(code_word(3));
    when "0100" => s_code_word(2) := not(code_word(2));
    when "0010" => s_code_word(1) := not(code_word(1));
    when "0001" => s_code_word(0) := not(code_word(0));
    when others => null;
end case;
```

5.9.3 Hammingův kód

Perfektní kód pro opravu jedné chyby využívá podpůrné funkce z balíčků *BasicCode* a *HammingCode*. Generující balíček přijímá dva parametry, délku původní informace k a délku kódového slova n. Před vykonáním jakéhokoliv dílčího úkonu veřejné funkce je provedena kontrola podmínky parametrů. Pokud není splněno $n = 2^r - 1$ a $k = 2^r - r - 1$, kde $r = n - k$, pak bude vyhodnocování ukončeno a bude vypsána zpráva informující o požadovaném formátu parametrů.

Pro zakódování je zjištěna generující matice skrze funkci HammingCodeG[n,k], která jako jeden parametr přijímá dvouprvkový list n a k ve složených závorkách. Vygenerovaná maice je systematická, dodržuje tedy formát sloupců jednotkové matice následované sdruženými sloupci redundantních bitů. Bity kódového slova jsou definovány jako kombinační zápis maticového násobení informace a generující matice - stejně jako u kódování křížové parity. Z toho důvodu je proces kódování zcela identický, včetně dvou vnořených smyček a kontroly kladného bitu.

Níže je ukázka přepisu generující matice pro zakódování 4-bitové zprávy na 7-bitové kódové slovo na definice jednotlivých bitů za použití HK(7, 4).

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

$$\downarrow$$

$$\begin{aligned} code\_word(6) &= message(3); \\ code\_word(5) &= message(2); \\ code\_word(4) &= message(1); \\ code\_word(3) &= message(0); \\ code\_word(2) &= message(0) \oplus message(1) \oplus message(2); \\ code\_word(1) &= message(0) \oplus message(1) \oplus message(3); \\ code\_word(0) &= message(0) \oplus message(2) \oplus message(3); \end{aligned}$$

Pro dekódování je postup opět stejný jako u křížové parity. Získáme s pomocí funkce ConvertToH[G_K] kontrolní matici a generujeme syndrom. Syndrom používáme jako klíč pro strukturu *case* k opravě jednoho bitu přijatého slova. Možnosti k porovnání tvoří množina sloupců kontrolní matice. Je-li nalezena shoda syndromu s nějakou nabízenou možností, dojde k inverzi patřičného bitu kódového slova. Tento kód bez ohledu na rozměry je schopen opravit pouze jednu chybu.

Níže je ukázka přepisu kontrolní matice kódu HK(7, 4) na syndrom k určení chyby.

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

$$\downarrow$$

$$syndrom(2) = code\_word(2) \oplus code\_word(3) \oplus code\_word(4) \oplus code\_word(5);$$
$$syndrom(1) = code\_word(1) \oplus code\_word(3) \oplus code\_word(4) \oplus code\_word(6);$$
$$syndrom(0) = code\_word(0) \oplus code\_word(3) \oplus code\_word(5) \oplus code\_word(6);$$

Při určení původního slova zpět do výstupní zprávy message_OUT se překopíruje k počátečních bitů. Blízké seskupení bitů zprávy v kódovém slově bylo umožněno generováním systematickou maticí.

5.9.4 Zkrácený Hammingův kód

Zkrácený Hammingův kód stejně jako jeho základní verze Hammingův kód využívá nabídku balíčků *BasicCode* a *HammingCode*. Oba kódy jsou si velmi podobné, a proto bude jejich implementace v mnoha aspektech totožná. Jediná hmatatelná změna v procesu kódování a dekódování je využití funkce HammingShortenedCodeG[n,k], jež přijímá opět dvouprvkový list z n a k. Ten je podroben kontrole podmínek $n = 2^r - 1 - i$ a $k = 2^r - r - 1 - i$, kde $r >= 2$, $i >= 0$ a $r = n - k$. V případě jejich nesplnění je vyhodnocování funkce okamžitě ukončeno.

Zkrácený Hammingův kód nenabízí k tvorbě generující matice systematický tvar. Zápis kódování je stejný jako u klasického Hammingova kódu, pouze generující matice nabývá odlišných rozměrů. Zkrácený Hammingův kód je nevhodný používat pro účely opravy chyby na kódovém slově, protože nedokáže pokrýt všechny možnosti výskytu chybového bitu. Přestože je tento kód obecně neopravitelný, byl mu ponechán proces dekódování pro ilustrační účely. Případně vykonané opravy nebudou validní, pokud chyba nebyla nepromítnuta na redundantní bit.

Proces dekódování (ač zbytečného) je rozšířen o speciální převod kódového slova zpět na informaci. Zde je informační zpráva definovaná po bitech, nikoliv skupinou sousedících bitů. Toho je dosaženo postupným procházením generující matice. Ta je procházena po sloupcích, mezi nimiž se snažíme nalézt sloupce jednotkové matice. Bity sloupců jsou sčítány do sumy. Pokud je součet jedničkových bitů po průchodu celým sloupcem roven jedné, spárujeme pozici tohoto sloupce s bitem kódového slova, které bude předáno bitu informace s indexem, který odpovídá indexu osamělé jedničky v nalezeném sloupci.

5.9.5 Rozšířený Hammingův kód

Rozšířený Hammingův kód, jenž je rozšířený o redundantní bit sudé parity, opět využívá nabídku balíčků *BasicCode* a *HammingCode*. Z počátku je vykonána kontrola podmínek $n-1 = 2^{n-1-k}-1$ a $k = 2^{n-1-k} - n - k - 2$. Není-li podmínce vyhověno, bude konání programu ukončeno.

Podobnost s výchozím Hammingovým kódem je natolik rozsáhlá, že implementace neobsahuje téměř žádné rozdíly. Stejně jako u zkráceného Hammingova kódu je hlavní změnou generující matice. Zde ji získám operací maticového násobení mezi funkcemi HammingCodeG[n-1,k] a EvenParityG[n-1] (v tomto pořadí). Výsledek takového násobení musí být držen na dvouprvkovém tělese, proto je na něj aplikována operace modus 2 (zbytek po dělení dvou) zprostředkovaná funkcí Mod[]. Proces kódování a dekódování probíhá analogicky se stejnými procesy Hammingova kódu. Tento kód opravuje jednu chybu. Je-li chyba objevena, bude hodnota jejího bitu invertována ve struktuře *case*.

Níže je ukázka přepisu generující matice pro zakódování 8-bitové zprávy na 4-bitové kódové slovo na definice jednotlivých bitů za použití Rozšířený HK(8, 4).

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{pmatrix}$$

$$\downarrow$$

$$\begin{aligned} code\_word(7) &= message(3); \\ code\_word(6) &= message(2); \\ code\_word(5) &= message(1); \\ code\_word(4) &= message(0); \\ code\_word(3) &= message(0) \oplus message(1) \oplus message(2); \\ code\_word(2) &= message(0) \oplus message(1) \oplus message(3); \\ code\_word(1) &= message(0) \oplus message(2) \oplus message(3); \\ code\_word(0) &= message(1) \oplus message(2) \oplus message(3); \end{aligned}$$

Níže je ukázka přepisu kontrolní matice kódu Rozšířený HK(8, 4) na syndrom k určení chyby.

$$\begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

$$\downarrow$$

$$\begin{aligned} syndrom(3) &= code\_word(3) \oplus code\_word(4) \oplus code\_word(5) \oplus code\_word(6); \\ syndrom(2) &= code\_word(2) \oplus code\_word(5) \oplus code\_word(6) \oplus code\_word(7); \\ syndrom(1) &= code\_word(1) \oplus code\_word(4) \oplus code\_word(6) \oplus code\_word(7); \\ syndrom(0) &= code\_word(1) \oplus code\_word(4) \oplus code\_word(6) \oplus code\_word(7); \end{aligned}$$

5.9.6 Cyklický kód

Cyklický kód využívá balíčky *BasicCode* a *PolynomialCode* Stanislava Koleníka z roku 2021 a balíček *FiniteFields*. [17] Přijímá dva parametry, generující polynom p s neznámou proměnnou x a délku kódového slova n. Před zavoláním soukromých funkcí generujících jednotlivá skripta, musí být potvrzena cykličnost kódu funkcí CyclicQ[] a dopočítána délka zprávy k zakódování k. Ta je následně dílčím funkcím předávána pro zamezení opakovaných výpočtu.

Cyklický kód je soustředěn kolem počtů s mnohočleny. V prostředí automatizace výpočtů se však hodí polynomiální generátor, zprávy i kódová slova interpretovat jako logické vektory.

Proces zakódování je v teoretickém postupu uskutečněn vynásobením mnohočlenů původní informace a generátoru. Místo tohoto přístupu však byla využita alternativa zakódování generující maticí, jako tomu je u všech ostatních kódů. Matice je tvořena z bitového zápisu generátoru v jednom řádku a dále jeho bitovými posuvy doprava na zbylých $k-1$ řádcích. Jednotlivé řádky je vhodné seskládat od prvního řádku k poslednímu s klesajícími stupni mnohočlenů. Této matice je možné docílit funkcí GeneratorMatrix[]. Balíček také obsahuje lokální nepoužitou funkci CyclicG[], která taktéž umožňuje vytvořit z generujícího polynomu generující matici. Toho je docíleno opakovaným posuvem generujícího polynomu doprava, který vytvoří nový vektor, jež je přidán jako další řádek matice.

V momentě vytvoření generující matice je ve funkci CyclicEncodeVHDL[] další postup definování jednotlivých bitů výstupního kódového slova. Toho je opět docíleno strukturu dvou

vnořených smyček s podmínkou kontroly, že právě zkoumaný bit je kladný a musí být tedy přidán do definice onoho bitu.

Pro zadaný generující polynom $p = x^3 + x + 1$ a $n = 7$ vznikne matice, s jejímž přepisem lze nadefinovat postupně každý bit kódového slova. Generující matice G_K polynomu p má tvar

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{pmatrix}.$$

Dekódování v cyklickém kódu je realizováno polynomiálním dělením. Přijaté slovo nejvýše stupně $deg = n$ je vyděleno původním generujícím polynomem nejvýše stupně $deg = r$. Výsledek dělení, který nabývá nejvýše stupně $deg = k = n - r$, je dekódovanou informací. Pokud je zbytek po dělení nenulový, pak je dekódovaná informace nevalidní. Došlo-li však k nejvýše jedné chybě, může být opravena.

Polynomiální dělení je realizováno kombinačně ve VHDL s pomocí operace XOR a pomocných struktur logických vektorů. Přijaté slovo je přepsáno jako n-bitový logický vektor, kde jsou nulové členy v pozici svého řádu zastoupeny nulou. Stejně jsou zapsány také generátor a výsledek.

$$(x^5 + x^4 + x^3 + x) : (x^3 + x + 1) = x^2 + x$$

$$\downarrow$$

```
 6.  5.  4.  3.  2.  1.  0.      3.  2.  1.  0.      3.  2.  1.  0.
(0,  1,  1,  1,  0,  1,  0)  :  (1,  0,  1,  1)  =  (0,  1,  1,  0)
```

Dělení je rozepsáno do k podprocesů, ve kterých se hodnota děleného vektoru vždy sníží o řád. První krok dělení je uveden níže. Prvek na pozici 6. řádu děleného vektoru a prvek nejvyššího řádu generátoru jsou vyhodnoceny operací AND, a její výsledek je zapsán 3. pozici výstupní informace (od dělené pozice je odečten stupeň generujícího polynomu). Následně je provedena zpětná kontrola dělení polynomu, nově získaného bit zprávy a zbylé bitů generátoru jsou podrobeny operaci AND a její výsledek je zapsán na součet pozic obou aktérů. Volná místa jsou doplněna nulami. Do dalšího kroku dělení vstoupí pouze zbytek po prvním dělení. Tím je výstup operace XOR mezi již děleným logickým vektorem a kontrolním vektorem zapsaným pod ním (na ukázce níže je tento výstup oddělen přerušovanou čárou).

```
 6.  5.  4.  3.  2.  1.  0.      3.  2.  1.  0.      3.  2.  1.  0.
(0,  1,  1,  1,  0,  1,  0)  :  (1,  0,  1,  1)  =  (0,  _,  _,  _)
 0   0   0   0   0   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 0   1   1   1   0   1   0
```

Druhý krok dělení postupuje analogicky pro 5. pozici aktuálně děleného logického vektoru.

```
 6.  5.  4.  3.  2.  1.  0.      3.  2.  1.  0.      3.  2.  1.  0.
(0,  1,  1,  1,  0,  1,  0)  :  (1,  0,  1,  1)  =  (0,  1,  _,  _)
 0   0   0   0   0   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 0   1   1   1   0   1   0
 0   1   0   1   1   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
 0   0   1   0   1   1   0
```

Třetí krok dělení, jenž je rozepsán níže, se věnuje dělení pro prvek na 4. pozici děleného logického vektoru, jehož výsledek ze zpětné kontroly dá k dalšímu dělení nulový vektor, není již tedy žádný zbytek. V prostředí VHDL ta poslední, zde nevykonaná, instance dělení proběhne s hodnotou zbytku (i nulového).

```
  6.  5.  4.  3.  2.  1.  0.      3.  2.  1.  0.      3.  2.  1.  0.
 (0,  1,  1,  1,  0,  1,  0) :  (1,  0,  1,  1) =  (0,  1,  1,  0)
  0   0   0   0   0   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  0   1   1   1   0   1   0
  0   1   0   1   1   0   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  0   0   1   0   1   1   0
  0   0   1   0   1   1   0
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
  0   0   0   0   0   0   0
```

Realizace polynomiálního dělení je ukázána na jedné instanci níže. Po ukončení poslední instance dělení je příznak chyby, nebo-li zbytek po dělení, získán z rozsahu $(n-k-1)$. až 0. pozice s_code_word (tento vektor uchovává aktuální hodnotu děleného slova v průběhu dělení).

Výpis kódu 5.22 První instance dělení pro generátor $p = x^3 + x + 1$, neboli p = (1, 0, 1, 1)

```
--  DIVISION
    s_message(3) := s_code_word(6) and generator(3);
--  REMAINDERS
    s_remainder := std_logic_vector(to_unsigned(0, n));
    s_remainder(6) := s_message(3) and generator(3);
    s_remainder(5) := s_message(3) and generator(2);
    s_remainder(4) := s_message(3) and generator(1);
    s_remainder(3) := s_message(3) and generator(0);
--  UPDATE s_code_word
    s_code_word(0) := s_code_word(0) xor s_remainder(0);
    s_code_word(1) := s_code_word(1) xor s_remainder(1);
    s_code_word(2) := s_code_word(2) xor s_remainder(2);
    s_code_word(3) := s_code_word(3) xor s_remainder(3);
    s_code_word(4) := s_code_word(4) xor s_remainder(4);
    s_code_word(5) := s_code_word(5) xor s_remainder(5);
    s_code_word(6) := s_code_word(6) xor s_remainder(6);
```

Získaný zbytek je použit jako klíčová hodnota pro *case*, kde označí chybný bit, jenž by měl být invertován. Pro některé generující polynomy zde nastává problém - nejsou schopné pro každou možnou chybu zajistit unikátní zbytek po dělení. Tedy dvě či více různých chyb mohou sdílet stejný příznak. Generující matice vytvořená z polynomu $p = x^5 + x^2$ pro $n = 6$, nabízí pouze tři možné příznaky pro šest možných chyb.

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}.$$

Chyby x^4 a x budou mít stejný příznak x. V případě, že by se jedna z těchto chyb pro takto zadaný kód objevila, nebudeme schopni ji jednoznačně eliminovat. Simulace neumožní dvě totožné možnosti pro *case*, proto je syndromová tabulka dekódujícího VHDL skriptu konstruována postupně. Syndrom je nejprve zkontrolován, že se již nevyskytuje v kontrolní množině již použitých variant, pokud je již přítomen, nový nebude přidán, a naopak. Toto způsobuje, že proti chybě je chráněn pouze bit na pozici, který byl přidán do syndromové tabulky. To může poškodit zprávu, pokud bude opraven na místo jiného bitu, který vyžadovanou opravu nedostane.

Jakmile byla zjištěna chyba a došlo k opravě na kódovém slově, probíhá nové dělení pro získání validní informace.

5.9.7 Součinový kód

Součinový kód využívá balíčky *BasicCode* a *ProductCode* Stanislava Koleníka z roku 2021. Přijímá dva parametry, dvě generující matice dvou kódů K_1 a K_2 (mohou být odlišné i totožné). U tohoto

kódu neprobíhá žádná kontrola validity parametrů. Z generujících matic jsou zjištěny parametry jednotlivých kódů n_1, k_1, n_2 a k_2, před zavoláním dílčích funkcí tvorby VHDL projektu.

Zakódování zprávy je u součinového kódu dosaženo s pomocí matice, která na jednotlivých pozicích obsahuje indexy bitů původní zprávy, kterým se rovná bit kódového slova na právě té pozici, po jeho vepsání do řádků matice. Při generování VHDL skriptu je tato matice procházena po řádkách.

Generující matice je vytvořena v soukromé funkci ProductG. Zde je nejprve vytvořen zápis indexů původní informace pouze podle kódu K_1, tím vzniká poloviční generující matice. Ta je na kompletní generující matici přetvořena vynásobením s maticí kódu K_2.

Pro kódy $K_1 = HK(7,4)$ a $K_2 = HK(3,1)$ je nejprve vytvořena poloviční generující matice tvaru

$$\begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 1+2+3 \\ 0+2+3 \\ 0+1+3 \end{pmatrix}.$$

Po vynásobení této matice generující maticí kódu K_2, je výstup generující matice součinového kódu pro zadané parametry $K_1 = HK(7,4)$ a $K_2 = HK(3,1)$ uveden níže.

$$G_{K_{K_1,K_2}} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 1+2+3 & 1+2+3 & 1+2+3 \\ 0+2+3 & 0+2+3 & 0+2+3 \\ 0+1+3 & 0+1+3 & 0+1+3 \end{pmatrix}$$

Výpis kódu 5.23 Definice bitů kódového slova pro součinový kód s $K_1 = HK(7,4)$ a $K_2 = HK(3,1)$

```
code_word(0) <= message(0);
code_word(1) <= message(0);
code_word(2) <= message(0);
code_word(3) <= message(1);
code_word(4) <= message(1);
code_word(5) <= message(1);
code_word(6) <= message(2);
code_word(7) <= message(2);
code_word(8) <= message(2);
code_word(9) <= message(3);
code_word(10) <= message(3);
code_word(11) <= message(3);
code_word(12) <= message(1) xor message(2) xor message(3);
code_word(13) <= message(1) xor message(2) xor message(3);
code_word(14) <= message(1) xor message(2) xor message(3);
code_word(15) <= message(0) xor message(2) xor message(3);
code_word(16) <= message(0) xor message(2) xor message(3);
code_word(17) <= message(0) xor message(2) xor message(3);
code_word(18) <= message(0) xor message(1) xor message(3);
code_word(19) <= message(0) xor message(1) xor message(3);
code_word(20) <= message(0) xor message(1) xor message(3);
```

Dekódování je velmi obsáhlé, neboť dekódujeme dva různé kódy. Přijaté slovo vepíšeme do řádků matice o rozměrech $n_2 \times n_1$. Z této matice si následně bereme její sloupce. Pro každý z nich určíme dle kódu K_2 syndrom a slovo s pomocí syndromové tabulky ve formátu *case* případně opravíme. Dekódovaná slova vepisujeme do sloupců matice $k_2 \times n_1$, která je zkrácena o paritní bity kódu K_2. Z nové menší matice budeme postupně vybírat řádky. Těm je určen syndrom dle kódu K_1 a jsou podle něj případně opraveny. Opravené řádky jsou zapsány bez paritních bitů kódu K_1 do třetí a poslední matice $k_2 \times k_1$, kde se čtením po řádcích získá dekódovaná informace. Dekódování každého sloupce první matice a každého řádku nové matice je uvedeno ve skriptu samostatně. Projekt kódování součinovým kódem neobsahuje v rozhraní výstupní hodnotu syndromu, protože kódové slovo nemá svůj samostatný syndrom, tím disponují pouze jeho části.

Generátor součinového kódu je v momentě dekódování omezen svými možnostmi. Při dekódování dochází k okrajování paritních bitů z právě používané matice, z toho důvodu se předpokládá, že k zakódování byly použity systematické generující matice, jejichž začínající sloupce tvoří jednotkovou matici následované sloupci paritních bitů. Návrh dekódování také nepředpokládá cyklické kódy.

Generátor přijme a vhodně zpracuje kódy, které jsou schopny opravit minimálně jednu chybu, jsou lineární necyklický a jsou zadány systematickou maticí.

5.9.8 RM kód

RM kód využívá balíčky *BasicCode* a *RMCode* Stanislava Koleníka z roku 2021. Druhý uvedený balíček není obsažen v samotné implementaci, neboť jeho funkce jsou zaměřené na vykreslení výpočtů než na dosažení jejich výsledku, přestože postupy k tomu potřebné obsahuje. Dva takové postupy se staly základem pro výpočetní postup dvou soukromých funkcí.

Tento balíček přijímá dva parametry ρ a μ, nejvyšší řád podmatice generující matice a počet proměnných tvořící prosté součiny, které jsou před vykonáním soukromých dílčích funkcí podrobeny kontrole. Není-li naplněna podmínka $1 <= \rho <= \mu$, pak je běh okamžitě přerušen.

Informace jsou kódovány generující maticí, která je specifická pro každou dvojici ρ a μ. Je vytvořena v soukromé funkci RMG (její princip je založen na řešení tvorby generující matice v balíčku *RMCode.wl*). Nejprve jsou postupně vytvořeny všechny podmatice řádů menších nebo rovných ρ, které jsou poté sloučeny do jedné generující matice. Zakódování probíhá jako u jiných lineárních kódů - bity kódového slova jsou definovány maticovým součinem původní informace a nové generující matice.

Proces dekódování je založen na principu majority. Z generující matice jsme schopni vypsat soustavu rovnic, kde jednu stranu tvoří samostatné bity přijatého slova a druhou stranu kombinace neznámých bitů dekódované informace. Ze soustavy je možné odvodit rovnice definující právě ty neznámé bity. Soustavu těchto rovnic pro patřičný bit dekódovaného slova je získána zavoláním funkce Equations (princip se zakládá na řešení dekódující funkce v balíčku *RMCode.wl*). Rovnice neurčí jednoznačnou kombinaci bitů slova z informačního kanálu, která se musí rovnat konkrétnímu bitu dekódované informace. Nabídne však více alternativ, a nejčastější výsledek všech alternativ se stane hodnotou bitu dekódované informace.

Proces dekódování byl, v rozporu s ostatními řešeními, realizován za použití více procesů. Tyto procesy se také liší tím, že nemají citlivostní seznam signálů, ale využívají wait příkazy. Pro každý bit dekódované zprávy jsou v kódu přítomny dva procesy, z nichž první určuje dílčí výstupy všech alternativních výsledků a druhý zjišťuje majoritní hodnotu, která bude nakonec připsána bitu. Poslední proces se nevěnuje hodnotě jednotlivých bitů, ale zapsání výsledků do výstupního signálu dekódované zprávy.

5.10 Generátor pracovních listů

Projekt pro generování až tří pracovních listů, každý o pěti stránkách, pro studenty předmětu BI-JPO je složen ze tří souborů. Hlavičkového souboru Generator.h, implementačního souboru Generator.cpp a souboru s funkcí main(), main.cpp.

Soubory Generator.h a Gnerator.cpp tvoří třídu Generator, která deklaruje a definuje tři veřejné metody workSheetEvenParity(), workSheetCrossParity() a workSheetHamming().

Výpis kódu 5.24 Vzor veřejné metody pro generování pracovního listu kro kód s názvem CodeName

```
bool Generator::workSheetCodeName() {
    std::ofstream fileOUT( CodeNameNameFile );
    if ( ! fileOUT ) return false;

    headerCodeName( fileOUT );
    middleSlides ( fileOUT );
    slideCodeName ( fileOUT );

    fileOUT.close();
    return true;
}
```

Implementace požadavku generování byla provedena velmi jednoduše, až primitivně. Veřejné metody mají k dispozici soukromé funkce. Pro každý kód existují dvě soukromé funkce header*codeName* (generující hlavičku WM souboru a část prvního stránky) a slide*codeName* (generující patu souboru včetně čtvrté a páté stránky WM notebooku obsahujícího zadání příkladů specifických pro daný kód), a pak jedna soukromá funkce middleSlides, kterou sdílí všechny kódy, neboť generuje společnou část souboru (dokončuje první stránku a tvoří celou druhou a třetí stránku). Všechny soukromé metody přijímají jako parametr referenci na výstupní proud (std::ofstream & out). Do toho proudu jsou zapisovány dlouhé fixní řetězce, které při správném poskládání vytvoří funkční textový soubor reprezentující WM soubor formátu s koncovkou .nb. Pracovní listy jsou zamýšleny jako čistě ilustrační pomůcky, proto neobsahují konkrétní hodnoty pro zadání, které by bylo možné při generování změnit pro větší variabilitu zadání. Součástí příkladů jsou předvyplněná data, která nemají nutně potřebnou délku či dimenzi, to dává vyplňujícímu nutnost najít jednotné řešení pro jím zamýšlený výsledek. Styl zadání úloh neposkytuje ochranu proti špatným vstupům, ty jsou pouze detekovány a je vypsána zpráva, která o nich informuje. Pracovní listy pro různé kódy se liší v jejich uvedení na první stránce. Dalším rozdílem jsou poslední dvě stránky, které již nabízejí rámcové příklady, které se vztahují ke kódu, jemuž se pracovní list věnuje.

Doplňkem k projektu je soubor Makefile, který zajišťuje kompilaci a odstranění souborů vytvořených pro spuštění programu. Po zadání příkazu *make* dojde ke kompilaci všech částí generátoru. Pro spuštění je pak k dispozici kompilací vytvořený objekt *generator*. Po kompilaci a spuštění programu je vhodné také zadat *make clean* (dojde ke smazání již nepotřebných souborů).

Výpis kódu 5.25 Obsah souboru Makefile

```
generator: main.o Generator.o
        mkdir -p build
        g++ main.o Generator.o -o generator
main.o: main.cpp Generator.h
        mkdir -p build
        g++ -g -Wall -pedantic -std=c++11 -c main.cpp -o main.o
generator.o: Generator.cpp Generator.h
        mkdir -p build
        g++ -g -Wall -pedantic -std=c++11 -c Generator.cpp -o Generator.o
```

```
clean:
        rm generator
        rm -r build

.PHONY: clean
```

Kapitola 6
Testování

Výstupní osmice generátorů byla v průběhu svého vývoje a po něm podrobena testům, které složitostí odpovídají možným zadáním ve výukovém prostředí ČVUT FIT. Tyto nástroje mají primárně sloužit jako ilustrační pomůcka pro výuku bezpečnostních kódů, a proto rozsah a způsob jejich testování byl zaměřen na menší rozsahy hodnot.

Vygenerované VHDL projekty byly podrobeny testování v programu Vivado verze 2018.3. Výstup simulace je ve formátu waveform, který umožňuje sledovat změny na signálech v časovém průběhu. Hodnoty jednotlivých vln jsou do nich vepsány v hexadecimálním formátu. Legenda je uvedena na levé straně - jméno signálu (značeno Name) společně s hodnotou v momentě ukončení simulace (značeno Value).

Výsledky simulací jsou k dispozici v přiložených materiálech ve složce **Simulations** v osmi souborech formátu PDF. Tyto soubory sdružují výsledky simulace pro různé parametry definující kód a různá testovací data. Kromě PDF souborů je obsažen ještě soubor snímků výsledků jednotlivých simulací ve složce **ResultImages**.

Testování je níže členěno do samostatných osmi sekcí. Každá jedna sekce je určena pro zhodnocení generátoru VHDL projektu jednoho kódu. Každému kódu byl vytvořen notebook s předvyplněnými vstupy, který usnadňuje použití příslušného balíčku.

6.1 Sudá parita - wsParity.wl

Sudá parita je velmi jednoduchý detekční kód, který nejen že neumožňuje opravu chyby, ale i samotné detekování je limitované na jednu chybu. Testovací data nabízejí pouze nulové nebo jednobitové chybové stupy. Výsledky simulace ukazují, že jednobitová vlna syndromu nabývá nenulové hodnoty, pokud je ve stejný časový okamžik hodnota chyby také nenulová. Výstupní zpráva (message_OUT) je pro všechny časové rámce s nenulovou chybou rozdílná od vstupní zprávy (message_IN). Jedinou výjimkou je případ, kdy chyba postihla paritní bit, a dekódovaná informace je i přes chybu schodná s originálem.

Tabulka 6.1 Simulované testy v Parity_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$k = 3$
2.	$k = 5$
3.	$k = 12$

Součástí přiložených materiálů je soubor Parity.nb ve složce **Notebooks**, který načítá balíček wsParity.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.1.

V rámci testování byly vygenerovány tři projekty pro relativně malý počet vstupních bitů, pro 3, 5 a 12 bitů. Výstup simulace se naplnil očekávaní průběhu a výstupu. Záznam průběhu simulace je v přiloženém souboru ve složce **Simulations** pod názvem Parity_simulation_waveform_result.pdf.

6.2 Křížová parita - wsCross.wl

Křížová parita je paritní kód, který již umožňuje provést korekci přijatého slova z informačního kanálu. Toho je využito během testování. Projekt byl testován třemi vstupy relativně malých hodnot, 2, 4 a 9. Všechny aplikované jednobitové chyby byly odstraněny během dekódování a ze záznamu průběhu simulace je patrné, že vstupní zprávy (message_IN) jsou vždy ve shodě s výstupní zprávou (message_OUT). Stejně tak je dle očekávání na výstupu nulový syndrom, nenastala-li žádná chyba, a nenulový syndrom, pokud chyba nastala. To se nezmění ani s opravou dané chyby (syndrom nadále reprezentuje chybu, jež byla detekována, přestože již není přítomna).

Tabulka 6.2 Simulované testy v Cross_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$k = 2$
2.	$k = 4$
3.	$k = 9$

Součástí přiložených materiálů je soubor Cross.nb ve složce **Notebooks**, který načítá balíček wsCross.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.2. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem Cross_simulation_waveform_result.pdf.

6.3 Hammingův kód - wsHamming.wl

Hammingův kód byl v rámci testování podroben nejvyšší zátěži v podobě jedné chyby, která je pro tento kód zcela adekvátní. Na vstupu byly střídány nulové a jednobitové chyby, ze záznamu průběhu simulace je však patrné že vstupní a výstupní informace se nikdy nelišili.

Tabulka 6.3 Simulované testy v Hamming_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$n = 3, k = 1$
2.	$n = 7, k = 4$
3.	$n = 15, k = 11$

Součástí přiložených materiálů je soubor Hamming.nb ve složce **Notebooks**, který načítá balíček wsHamming.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.3. Testovány byly tři nejjednodušší verze Hammingova kódu, a to sice pro počet redundantních bitů rovný 2, 3, a 4. Výstup simulace byl jednoznačně pozitivní, v případě výskytu jednobitové chyby došlo vždy k nápravě. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem Hamming_simulation_waveform_result.pdf.

6.4 Zkrácený Hammingův kód - wsShortened.wl

Zkrácený Hammingův kód je silně vybaven pro detekci chyby, ale pro její opravu již tolik ne. Z výstupu simulace je patrné, že kódová slova, která nebyla modifikována chybou jsou korektně

dekódována zpět. Ovšem slova postižená jednobitovou chybou jsou dekódovány do původního tvaru informace pouze, pokud chyba postihla redundantní bit. Ve výstupu simulace je patrné, že jen některá slova po aplikaci chyby byla opravena. Testování naplnilo očekávání jeho o průběhu a výsledku.

Tabulka 6.4 Simulované testy v Shortened_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$n = 3, k = 1$
2.	$n = 9, k = 5$
3.	$n = 31, k = 26$

Součástí přiložených materiálů je soubor Shortened.nb ve složce **Notebooks**, který načítá balíček wsShortened.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.4. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem Shortened_simulation_waveform_result.pdf.

6.5 Rozšířený Hammingův kód - wsExtended.wl

Rozšířený Hammingův kód nabízí jeden redundantní bit navíc oproti jeho základní verzi. Kód zaručuje detekování a opravu jednobitové chyby. Z přehledu simulace je zřejmé, že vstupní a výstupní informace ne nikdy neliší, bez ohledu na to, zda ve stejném časovém okamžiku byla hodnota chyby nenulová.

Tabulka 6.5 Simulované testy v Extended_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$n = 4, k = 1$
2.	$n = 8, k = 4$
3.	$n = 16, k = 11$

Součástí přiložených materiálů je soubor Extended.nb ve složce **Notebooks**, který načítá balíček wsExtended.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.5. Testovány byly hodnoty navazující na testy pro klasický Hammingův kód, pro počet redundantních bitů rovný 3, 4, a 5, jedná se o jejich rozšíření o jeden bit parity. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem Extended_simulation_waveform_result.pdf.

6.6 Cyklický kód - wsCyclic.wl

Cyklický kód nabízí velkou versatilitu ve výběru platných parametrů. V rámci testování byl podroben takovým zadáním, které zaručují jednoznačnost syndromové tabulky pro každou jednobitovou chybu. Cyklické kódy této charakteristiky zvládají nalézt jednobitovou chybu a opravit ji. Ve výstupu simulace je možné vidět výsledky pro kód generovaný polynomy třetího stupně a polynomem pátého stupně. Výsledky testování naplnily očekávání, kdy všechny výstupní (message_OUT) informace se shodují se vstupními (message_IN).

Tabulka 6.6 Simulované testy v Cyclic_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$p = x^3 + x + 1, n = 7$
2.	$p = x^3 + x^2 + 1, n = 7$
3.	$p = x^5 + x^4 + x^2 + 1, n = 15$

Součástí přiložených materiálů je soubor Cyclic.nb ve složce **Notebooks**, který načítá balíček wsCyclic.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.6. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem Cyclic_simulation_waveform_result.pdf.

6.7 Součinový kód - wsProduct.wl

Součinový kód se věnuje dvěma kódům zvlášť, to platí zejména pro dekódování. Balíček wsProduct.wl požaduje na vstupu necyklické lineární kódy, zadané systematickou generující maticí. Výsledky simulace naplnily očekávání, a výstupní hodnota se vždy shoduje se vstupní informací k zakódování.

Tabulka 6.7 Simulované testy v Product_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$K_1 = HK(3,1), K_2 = HK(7,4)$
2.	$K_1 = HK(7,4), K_2 = HK(3,1)$
3.	$K_1 = HK(7,4), K_2 = HK(7,4)$
4.	$K_1 = HK(7,4), K_2 = Cross(k = 5)$

Součástí přiložených materiálů je soubor Product.nb ve složce **Notebooks**, který načítá balíček wsProduct.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.7. Testování bylo zaměřeno na vstupy ze skupiny Hammingových kódů, a pak také na křížovou paritu, která k dekódování též využívá syndromovou tabulku. Výsledky simulace naplnili očekávání. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem Product_simulation_waveform_result.pdf.

6.8 RM kód - wsRM.wl

RM kód umožňuje detekci a opravu libovolného množství chyb. V rámci simulace byl však podroben, stejně jako všechny ostatní kódy, pouze jednobitovým chybám. Ačkoliv generované VHDL skripty pro zakódování informace a pro vložení chyby fungují bez jediného pochybení, není možné to samé říci o generovaných skriptech pro dekódování, neboť nenaplňují očekávání. Kódová slova, ať už modifikovaná chybou či nikoliv, nejsou průchodem procesů dekódování vrácena do původního stavu. Tento nedostatek pravděpodobně pramení z nevhodného návrhu implementace VHDL řešení.

Tabulka 6.8 Simulované testy v RM_simulation_waveform_result.pdf.

Stránka	Parametry testovaného kódu
1.	$\rho = 1, \mu = 3$
2.	$\rho = 2, \mu = 3$
3.	$\rho = 3, \mu = 4$

Součástí přiložených materiálů je soubor RM.nb ve složce **Notebooks**, který načítá balíček wsRM.wl ze složky **Notebooks/Packages** a obsahuje zadání pro testy uvedené v tabulce 6.6. Záznam průběhu simulace je přístupný ve složce **Simulations** pod názvem RM_simulation_waveform_result.pdf.

6.9 Generování pracovních listů

Po zadání příkazu *make* byly vygenerovány všechny tři pracovní listy pro lineární bezpečnostní kódy. Jejich forma (členění na stránky a hierarchické řazení) i obsah naplňovali očekávání. Při spuštění nově vygenerovaných souborů program Wolfram Mathematica, verze 12, preventivně informuje uživatele o skutečnosti, že soubor nebyl vytvořen prostředím WM. Soubory projektu pro generování pracovních listů jsou přístupné ve složce **WorkSheets**. Tato složka však neobsahuje knihovny potřebné pro použití pracovních listů, ty jsou umístěny v lokaci **Notebooks/Packages** a nebyly kopírovány k projektu ze snahy vyhnout se duplikování přiložených souborů.

Závěr

Jako cíl této bakalářské práce bylo stanoveno shrnout poznatky o bezpečnostních kódech a vytvořit výukové pomůcky pro předmět BI-JPO. Proto se značná část práce věnuje tvorbě generátorů v podobě balíčků prostředí Wolfram Mathematica. Generátory tvoří pro zvolený kód celkem osm, respektive sedm souborů. Tvoří mimo jiné kodér, kde je informace kombinační logikou přeměněna na kódové slovo, soubor superponující chybu, kde je simulováno narušení přenášené informace, dekodér, kde se detekuje narušení integrity kódového slova a případně dojde i k opravě. Pro jednotné využití těchto komponent a jejich otestování jsou generovány dva pomocné soubory společně se třemi, respektive dvěma soubory vstupních dat.

Generátory úspěšně využívají sadu nástrojů z výstupu práce Stanislava Koleníka z roku 2021 a jsou na ně odkázané při tvorbě logických struktur využívaných pro samotné generování.

Testování vygenerovaných skriptů pro různé parametry a vstupy dosáhlo očekávání pro sedm z osmi kódů. Dekodér RM kódu nenaplňuje své funkční požadavky. Výsledky simulace potvrzují úspěšnost u všech ostatních výstupů, včetně detekce a opravy chyby, je-li to možné.

Práce může být v budoucnu dále rozšiřitelná o chybějící funkční řešení dekodéru, další lineární kódy a sekvenční realizaci skriptů pro nahrání do přípravku.

Příloha A

Příloha

Součástí práce je soubor Solutions.zip s výstupem praktické části práce.

Bibliografie

1. GANEEV, Timur. *Nástroj pro generování bezpečnostních kódů ve VHDL pomocí programu Wolfram Mathematica*. ČVUT v Praze, Fakulta informačních technologií, Katedra počítačových systémů, 2021.
2. ADÁMEK, Jiří. *Foundations of Coding: Theory and Applications of Error-Correcting Codes with an Introduction to Cryptography and Information Theory* [online]. 1991. ISBN 978-0-471-62187-4. Dostupné také z: `https://pg024ec.files.wordpress.com/2013/09/adc3a1mek-j-for-foundations-of-coding-wiley-1991.pdf`.
3. MOREIRA, Jorge Castiñeira; FARRELL, Patrick Guy. *Essentials of Error-Control Coding*. Wiley, 2006. ISBN 047002920X.
4. LIN, Shu; JR., Daniel J. Costello. *Error Control Coding*. 2. vyd. Prentice Hall, 2004. ISBN 0130426725.
5. OLŠÁK, Petr. *Lineární algebra* [online]. 2007. Dostupné také z: `http://petr.olsak.net/ftp/olsak/linal/linal.pdf`.
6. KOLENÍK, Stanislav. *Pokročilé bezpečnostní kódy v programu Wolfram Mathematica*. ČVUT v Praze, Fakulta informačních technologií, Katedra počítačových systémů, 2021.
7. DOUBEK, Jakub. *Knihovna funkcí pro program Wolfram Mathematica umožňující snadný návrh bezpečnostních kódů ve výuce*. ČVUT v Praze, Fakulta informačních technologií, Katedra počítačových systémů, 2016.
8. PURSER, Michael. *Introduction to Error-Correcting Code*. Artech House Publishers, 1994. ISBN 978-0890067840.
9. GHOSH, Debopam; MITRA, Arijit; MUKHOPADHYAY, Arijit; DAWN, Aniket; GHOSH, Devopam. *A generalized code for computing cyclic redundancy check* [online]. 2013. Dostupné také z: `https://core.ac.uk/download/pdf/268005925.pdf`.
10. MEALY, Bryan; TAPPERO, Fabrizio. *Free Range VHDL* [online]. 2013. Dostupné také z: `https://faculty-web.msoe.edu/johnsontimoj/EE3921/files3921/Book_FreeRangeVHDL.pdf`.
11. WOLFRAM, Stephen. *Celebrating a Third of a Century of Mathematica, and Looking Forward* [online]. 2021. Dostupné také z: `https://writings.stephenwolfram.com/2021/10/celebrating-a-third-of-a-century-of-mathematica-and-looking-forward/`.
12. KOLENÍK, Stanislav. *Podpora výuky bezpečnostních kódů v programu Wolfram Mathematica*. ČVUT v Praze, Fakulta informačních technologií, Katedra počítačových systémů, 2019.
13. CPLUSPLUS.COM. *std::ofstream* [online]. 2023. Dostupné také z: `https://cplusplus.com/reference/fstream/ofstream/`.

14. WOLFRAM. *Create a Package File* [online]. 2023. Dostupné také z: https://reference.wolfram.com/language/workflow/CreateAPackageFile.html.
15. WOLFRAM. *Load a Package* [online]. 2023. Dostupné také z: https://reference.wolfram.com/language/workflow/LoadAPackage.html.
16. WOLFRAM. *File Operations* [online]. 2023. Dostupné také z: https://reference.wolfram.com/language/guide/FileOperations.html.
17. WOLFRAM. *Finite Fields Package* [online]. 2023. Dostupné také z: https://reference.wolfram.com/language/FiniteFields/guide/FiniteFieldsPackage.html.

Obsah přiloženého média

- `Solution`
  - `Literature` — Literární zdroje
  - `Notebooks` — Výstup praktické části
    - `Packages` — Adresář s WM balíčky
      - `Internal` — Adresář s balíčky doplňující soubory *Code.wl
      - `ws*.wl` — Výstupní balíčky práce
      - `*Code.wl` — Balíčky Stanislava Koleníka z roku 2021
    - `*.nb` — WM notebooky pro zpracované kódy
  - `Simulations` — Adresář s výsledky simulací
    - `ResultImages` — Adresář se snímky výsledků simulací
    - `*.pdf` — PDF soubory s výsledky simulací
  - `WorkSheets` — Adresář nástroje generování pracovních listů
    - `Generator.cpp` — .cpp soubor generátoru pracovních listů
    - `Generator.h` — .h soubor generátoru pracovních listů
    - `main.cpp` — Soubor funkce main()
    - `Makefile` — Makefile