VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

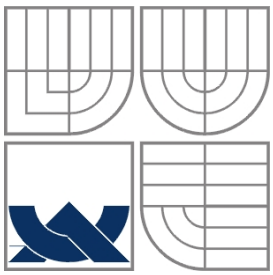
NÁVRH A IMPLEMENTACE KODÉRU A DEKODÉRU IWADARIHO KÓDU

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

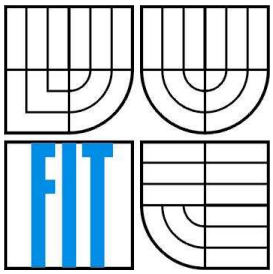
AUTOR PRÁCE
AUTHOR

JAN KŘIVÁNEK

BRNO 2007



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NÁVRH A IMPLEMENTACE KODÉRU A DEKODÉRU IWADARIHO KÓDU

DESIGN AND IMPLEMENTATION OF ENCODER AND DECODER OF IWADARI CODE

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — JAN KŘIVÁNEK
AUTHOR

VEDOUCÍ PRÁCE — Doc. Ing. SEKANINA LUKÁŠ, Ph.D.
SUPERVISOR

BRNO 2007

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2006/2007

Zadání bakalářské práce

Řešitel: **Křivánek Jan**
Obor: Informační technologie
Téma: **Návrh a implementace kodéru a dekodéru Iwadariho kódu**
Kategorie: Návrh číslicových systémů

Pokyny:

1. Seznamte se s kódováním a dekódováním Iwadariho kódu.
2. Seznamte se s výukovou platformou FitKit.
3. Navrhněte a simulujte obvodovou realizaci kodéru a dekodéru pro Iwadariho kód.
4. Implementujte navržený kodér a dekodér na platformě FitKit.
5. Ověřte funkčnost implementace.
6. Shrňte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Bod 1 a 2.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním paměťovém médiu (disketa, CD-ROM), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Sekanina Lukáš, doc. Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2006
Datum odevzdání: 15. května 2007

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami

1. Pan

Jméno a příjmení: **Jan Křivánek**
Id studenta: 84275
Bytem: Bzenecká 4, 628 00 Brno
Narozen: 04. 05. 1985, Ružomberok
(dále jen "autor")

a

2. Vysoké učení technické v Brně

Fakulta informačních technologií
se sídlem Božetěchova 2/1, 612 66 Brno, IČO 00216305
jejímž jménem jedná na základě písemného pověření děkanem fakulty:

.....
(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
bakalářská práce

Název VŠKP: Návrh a implementace kodéru a dekodéru Iwadariho kódu
Vedoucí/školitel VŠKP: Sekanina Lukáš, doc. Ing., Ph.D.
Ústav: Ústav počítačových systémů
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

tištěné formě počet exemplářů: 1
elektronické formě počet exemplářů: 2 (**1** ve skladu dokumentů, 1 na CD)

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizování výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti:
   - ☐ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/ 1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: .....

.....	.....
Nabyvatel	Autor

Abstrakt

Náplní předkládané práce je návrh, implementace a následné otestování obvodů realizujících kodér a dekodér Iwadariho kódu obecné zabezpečující schopnosti. Návrh vychází z nastíněné teorie protichybových kódových systémů a je podložen odvozeným matematickým aparátem. Popis implementace je zaměřen na využití jazyka *VHDL*. Ověření správné činnosti implementovaného modelu kodéru i dekodéru byla provedena simulací ve specializovaných vývojových prostředích. Finální demonstraci funkce poskytuje praktická realizace testovacího zapojení kodéru a dekodéru na experimentální výukové platformě FIT-kit.

Klíčová slova

zabezpečovací kódování, Iwadariho kodér, Iwadariho dekodér, FPGA, VHDL, FIT-kit

Abstract

This thesis deals with the design, the implementation and oncoming testing of circuits implementing the encoder and the decoder of the Iwadari code with universal correction ability. The design according to the theory of forward error correction systems is well-founded by mathematic modeling. Describing of the implementation prefers *VHDL* language. The implemented encoder and decoder were simulated in specialized development environments to verify their correct operation. Final demonstrational physical implementation of testing connection of the encoder and the decoder in experimental educational platform FIT-kit was made to show their function and mechanism.

Keywords

forward error correction, Iwadari´s encoder, Iwadari´s decoder, FPGA, VHDL, FIT-kit

Citace

Jan Křivánek: Návrh a implementace kodéru a dekodéru Iwadariho kódu, bakalářská práce, Brno, FIT VUT v Brně, 2007

Návrh a implementace kodéru a dekodéru Iwadariho kódu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Doc. Ing. Lukáše Sekaniny, Ph.D.
Další informace mi poskytl Ing. Zdeněk Vašíček
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jan Křivánek
15. 5. 2007

Poděkování

Rád bych poděkoval Doc. Ing. Lukáši Sekaninovi, Ph.D. za laskavé vedení mnou zvolené práce, věnovaný čas a podnětné náměty na zaměření a cíl práce. Dále pak Ing. Zdeňkovi Vašíčkovi za zajímavé návrhy na optimalizaci vytvořeného díla.

Obsah

1 Úvod

V současné době velkého rozmachu informačních a komunikačních technologií je naprosto nezbytné zabývat se otázkou zabezpečování informací proti chybám a odstraňování případných chyb. S velkým rozšířením rekonfigurovatelného hardwaru a tím značného zlevňování procesu výroby specializovaných hardwarových obvodů může vznikat otázka, zda se nepokoušet implementovat systémy zabezpečení proti chybám také přímo jako specializovaný hardware. Snahou této práce je tedy zrealizovat implementaci konkrétního kodeku do programovatelného hradlového pole na platformě FIT-kit. Na případných rozšířeních této práce je pak ponecháno vlastní posouzení rentability využití tohoto způsobu implementace některých zabezpečovacích kódů jako alternativy k nejčastěji používané softwarové implementaci v digitálních signálových procesorech.

Vlastním obsahem této práce je teoretické odvození činnosti Iwadariho kódu a praktická realizace kodeku Iwadariho kódu. Druhá kapitola, *Teorie informací, datová komunikace*, obeznamuje s některými nejdůležitějšími pojmy a zdůrazňuje nutnost existence zabezpečovacích kódů. Zabezpečovacími kódy obecně se podrobněji zabývá třetí kapitola s názvem *Zabezpečovací kódy*. Ta jednak začleňuje zabezpečovací kódy do širšího okruhu kódů, uvádí princip jejich činnosti a také je klasifikuje, aby bylo možné přesně zařadit zkoumaný Iwadariho kód. *Iwadariho kód* je pak tématem celé čtvrté kapitoly. V této části se zavádí jeho formální definice a za pomocí jednoduchého matematického aparátu navrhuje způsob hardwarové implementace kodéru a dekodéru Iwadariho kódu s obecnou korekční schopnost.

Obecnými možnostmi *návrhu a realizace číslicových systémů* se zabývá kapitola pátá. V ní jsou uvedeny stupně a alternativy návrhu číslicových systému a možné využitelné cílové technologie pro tyto systémy. Vyzdvižen je především moderní návrh za pomocí programovacího jazyka *VHDL* a cílová technologie programovatelného hradlového pole - *FPGA*. Je popsána platforma FIT-kit (obsahující programovatelné hradlové pole), určené k názorné praktické výuce tvorby hardwaru na Fakultě informačních technologií.

Popis postupu implementace kodéru a dekodéru Iwadariho kódu v jazyce *VHDL* i následné testování zvolené implementace je náplní šesté kapitoly, *Implementace a simulace kodeku*. Jsou v ní analyzovány problémy spojené s praktickou realizací kodeku a jsou navrženy postupy jejich řešení. Vlastní implementace byla provedena dvěmi odlišnými způsoby popisu pro možnost zhodnocení jejich rozdílů a zvolení vhodnější cílové podoby.

Závěrečnou fázi práce - vlastní fyzickou realizaci kodeku a zhodnocení jeho parametrů - popisuje sedmá kapitola, *Syntéza a praktická realizace kodeku*. Uvedená kapitola dokazuje a demonstruje správnou funkčnost kodeku. Zhodnocením všech dosažených výsledků pak navazuje kapitola osmá, *Závěr*.

2 Teorie informace, datová komunikace

Chceme-li se zabývat přenosem informací a jejich zabezpečováním, je třeba si nejprve definovat jednotlivé používané pojmy a ucelit si teoretický podklad problému.

Základní pojem, s nímž se v informačních technologiích setkáváme, je *informace*. Podstatou informace je schopnost ji interpretovat. To mimo jiné zahrnuje proces jejího přenášení (z praktického hlediska se jedná nejen přenos informace mezi dvěmi výpočetními stroji, ale také mezi jednotlivými prvky jednoho výpočetního stroje - například načtení dat z pevného disku do operační paměti). Z tohoto důvodu patří k nejdůležitějším odvětvím teorie informace právě teorie přenosu informací. Dalším důležitým pojmem tedy pro nás bude *datová komunikace*.

2.1 Informace

Výklad tohoto pojmu není jednoznačný. Podle zdroje [7] se různé vědy k pojmu informace stavějí různě. Ale vědy exaktnější používají méně abstraktních definic tohoto pojmu. Pokusíme se tedy vystihnout to nejpodstatnější z různých pohledů na tento pojem.

Například podle [5] je informace množství neurčitosti nebo nejistoty o nějakém náhodném ději, odstraněná realizací tohoto děje a získáním výsledků. Každá realizace náhodně probíhajícího děje odstraňuje část nejistoty o něm a poskytuje určité množství znalostí, jejichž kvantitativní mírou je informace.

Za pomoci jiné literatury (viz [6]) je možné vyslovit podobnou, avšak pro naše účely lépe formulovanou, definici pojmu informace:

V nejobecnějším slova smyslu se informací chápe údaj o reálném prostředí, o jeho stavu a procesech v něm probíhajících. Informace snižuje (nebo v ideálním případě odstraňuje) neurčitost systému (například systému příjemce informace). Množství informace je dáno mírou neurčitosti systému (entropie), o který se neurčitost systému přijetím zprávy zmenšila. V informační vědě se informací rozumí především komunikovatelný poznatek, který má význam pro příjemce nebo údaj usnadňující volbu mezi alternativními rozhodovacími možnostmi.

Z výše uvedeného je patrná charakteristická vlastnost informace a to, že je z místa vzniku přenášena do místa využití. V této souvislosti velmi často hovoříme o *komunikaci*.

2.2 Datová komunikace

Datovou komunikaci (sdělování) můžeme definovat jako přenos informace mezi místem jejího vzniku (zdrojem) a místem jejího využití (spotřebičem) prostřednictvím přenosového prostředí (přenosového kanálu) podle dohodnutých pravidel. Což znázorňuje schéma (Obr. 2.1).

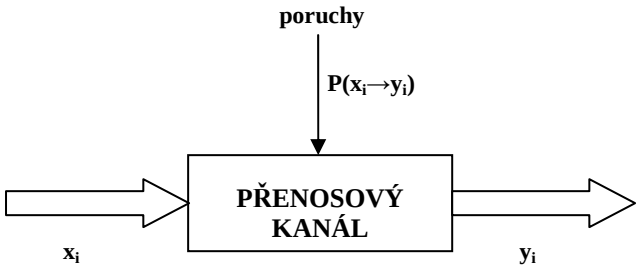
Obr. 2.1: Blokové schéma přenosu informace

Podle počtu stavů, jichž může přenosový kanál v průběhu přenosu nabývat, rozlišujeme dva typy kanálů: *diskrétní* a *analogový*. Pokud ve výše uvedeném schématu představují zdroj a spotřebič informace dva počítačové systémy (nebo obecněji i číslicové systémy), budou i přenášená data diskrétní. Pro účely této práce tedy budeme dále uvažovat již pouze přenos diskrétních dat pomocí digitálních signálů po diskrétním kanálu.

Uvažujeme-li pod pojmem datové komunikace v oboru informačních technologií prostorové a časové přemísťování informace pomocí elektrických a elektromagnetických prostředků, musíme v neideálním případě počítat s náchylností těchto prostředků k chybám. V uvedeném blokovém schématu přenosu informace je místem nejnáchylnějším k chybám právě přenosový kanál. Abychom dokázali problému výskytu chyb čelit, bude vhodné si nejprve formalizovat pojem *diskrétní kanál s poruchami*.

2.2.1 Komunikace přes diskrétní kanál s poruchami

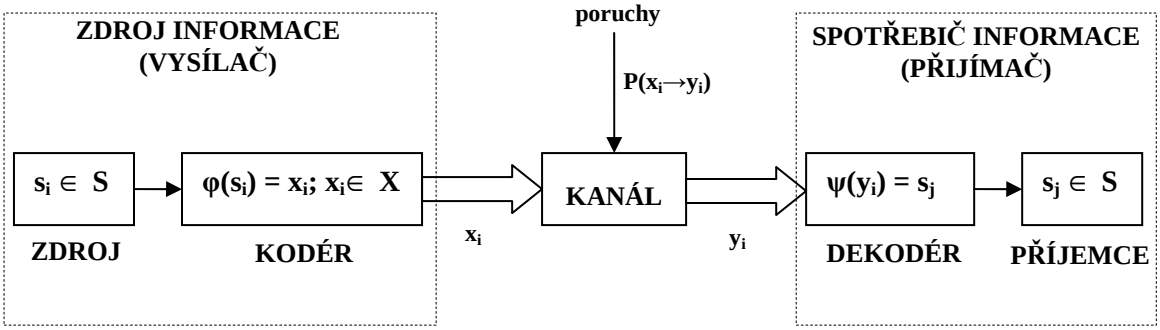
Pro diskrétní kanál s poruchami je možné uvažovat schéma (Obr. 2.2).



Obr. 2.2: Model diskrétního kanálu s poruchami

Vstup poruch do přenosového kanálu způsobí, že při vysílání prvku $x_i \in X$ se může tento prvek změnit na prvek $y_j \in Y$ různý od x_i. Prvkem můžeme označovat samostatný symbol (v našem případě jeden bit) - pak je vstupní abeceda X přenosového kanálu i abeceda Y výstupních prvků stejná (rovná {0; 1}). Avšak lze jím také označovat i souvislý blok symbolů - pak se mohou abecedy X a Y lišit, což umožňuje interpretaci přenesené informace správně i v případě vstupu chyb do přenosového kanálu.

K tomu, abychom mohli této skutečnosti využít pro zjištění a případně opravení chyb způsobených přenosem informace přes diskrétní kanál s poruchami, je třeba chápat schéma přenosu informace (Obr. 2.1) podrobněji.



Obr. 2.3: Blokové schéma systému diskrétního přenosu informace přes kanál s poruchami

Zavedeme tedy detailnější schéma (Obr. 2.3), pro jehož použité pojmy platí [4]:

- **Zdroj** - je charakterizován množinou prvků *S* zpráv, do níž náleží i aktuálně přenášený prvek s_i generované zprávy.
- **Kodér** - přiřazuje prvky abecedy *X* (zakódované posloupnosti) prvkům abecedy *S* prostřednictvím funkce přiřazení φ.
- **Kanál** - je určen vstupní abecedou *X* kódových posloupností, výstupní abecedou *Y* přenesených posloupností a pravděpodobností poruchy $P_{(x\rightarrow y)}$.
- **Dekodér** - vstupní posloupnost y_i (prvek *Y*) transformuje na nejvíce odpovídající prvek s_j množiny prvků zpráv (tato činnost může zahrnovat detekci případně korekci chyb přenosu).
- **Příjemce** - zpracovává přenesenou informaci. Při testovacím zapojení může vyhodnocovat platnost rovnosti $s_i = s_j$ jednotlivých přenášených a přenesených prvků zpráv, a tím vyhodnocovat vlastnosti použitého kódování.

Obecně lze vyslovit hypotézu, že pokud přenášíme prvky zpráv konstantní délky *n* po dvoustavovém kanálu, je celkový počet výstupních posloupností, které se mohou vyskytnout na výstupu kanálu 2^n (v našem případě $|Y| = 2^n$). Chyba v přenosu je zjistitelná (a případně opravitelná) v případě, kdy je počet kódových posloupností (posloupností používaných k přenosu prvků zpráv) značně menší než počet všech možných posloupností (tedy $|X| << |Y|$ a zároveň $X \subset Y$).

Způsob, jakým se volí tyto charakteristické vlastnosti přenosového systému a jakým způsobem transformují funkce φ a ψ informační posloupnosti na kódové a zpět, závisí na použitém kódu. Jednotlivé skupiny kódů mají ale tyto vlastnosti podobné. Je tedy smysluplné si *zabezpečovací kódy* klasifikovat.

3 Zabezpečovací kódy

Dle literatury je možné zabezpečovací kódy rozlišovat na dvě hlavní podskupiny [4]:

- Zabezpečující kódy pro ochranu informace proti chybám
- Zabezpečující kódy pro ochranu informace před zcizením (šifrování)

Nadále budeme pod pojmem zabezpečovací kódy uvažovat první skupinu kódů.

3.1 Princip protichybových kódů

V běžném dorozumívání (myšleno komunikace v soustavě člověk–člověk) obsahuje sdělovaná zprávy značnou míru *redundance (nadbytečnosti)*. Redundancí myslíme tu část obsahu zprávy, která se nikterak nepodílí na vlastním informačním obsahu zprávy. Vlastní informační obsah zprávy pak můžeme označit jako *relevantní informaci*.

Pro komunikaci v systému stroj-stroj (lépe mezi dvěmi číslicovými systémy) se snažíme přenášenou informaci zbavit redundance typické právě pro přirozený jazyk. Tento proces je označován jako *komprese zprávy*. V případě absence jakékoliv redundance v přenášené zprávě však jakákoliv chyba způsobí ztrátu celého relevantního informačního obsahu. Je tedy účelné redundanci do zprávy znovu uměle zavádět.

Původní redundance ve zprávě vyjádřené přirozeným jazykem (případně jiným, pseudonáhodným způsobem) měla příliš složitá pravidla vytváření. Pro korekční kódy je smysluplné nalézt předpis pro vytváření nadbytečnosti jednoduchým algoritmickým způsobem. Pokud takovýto algoritmus implementujeme do systému vysílače s kodérem i přijímače s dekodérem, bude v závislosti na míře přidání nadbytečnosti možné jistou část chyb v přenosu rozpoznat a případně část z těchto chyb i opravit.

3.2 Dělení protichybových kódů

Podle umístění bitů v nezabezpečeném toku bitů, které jsou používány k zabezpečení aktuálně přenášeného bitu, se rozlišují dva základní typy zabezpečovacích kódů [3]:

- **Blokové kódy** - zabezpečovací proces využívá pouze bity jednoho úseku nezabezpečeného toku bitů (jednoho bloku nezabezpečených bitů).
- **Btromové kódy** - zabezpečovací proces využívá navíc oproti blokovým kódům bity z předcházejících bloků nezabezpečených bitů (jako na schématu (Obr. 3.3)).

Blokové a stromové kódy se podle zabezpečovací schopnosti dělí na kódy, které zabezpečují proti [4]:

1. *Nezávislým chybám*.

2. *Shlukujícím se chybám* (tj. proti chybám, které mají v určitých úsecích přenášených dat nápadně zvýšenou četnost. Bývají způsobené například poškrábáním kompaktního disku, výbojem blesku, slunečními erupcemi atp.)
3. Nezávislým a shlukujícím se chybám.

Při zabezpečování proti shlukovým chybám je také třeba si stanovit, maximální délku shluku chyb *b* a kódem požadovanou minimální délku bezchybného intervalu *A* mezi jednotlivými shluky chyb. Tento interval se nazývá *Ochranný interval A*.

Kritériem dalšího dělení stromových kódů je splnění podmínky linearity:

$$ad_1 + bd_2 \rightarrow G(ad_1 + bd_2) = aG(d_1) + bG(d_2) \tag{3.1}$$

kde d_1 a d_2 jsou dvě vstupní posloupnosti a $G(d_1)$ a $G(d_2)$ jsou dvě výstupní posloupnosti. Pro dvoustavové signálové prvky se při zabezpečovací procesu používá operace *XOR* (v některé literatuře též nazývaná *sčítačka modulo 2*) umožňující tuto podmínku splnit. Proto se při implementaci kodérů dvoustavových stromových kódů používají především paměťové buňky a logické členy *XOR*.

Stromové kódy se tedy dle této podmínky dělí do dvou podskupin:

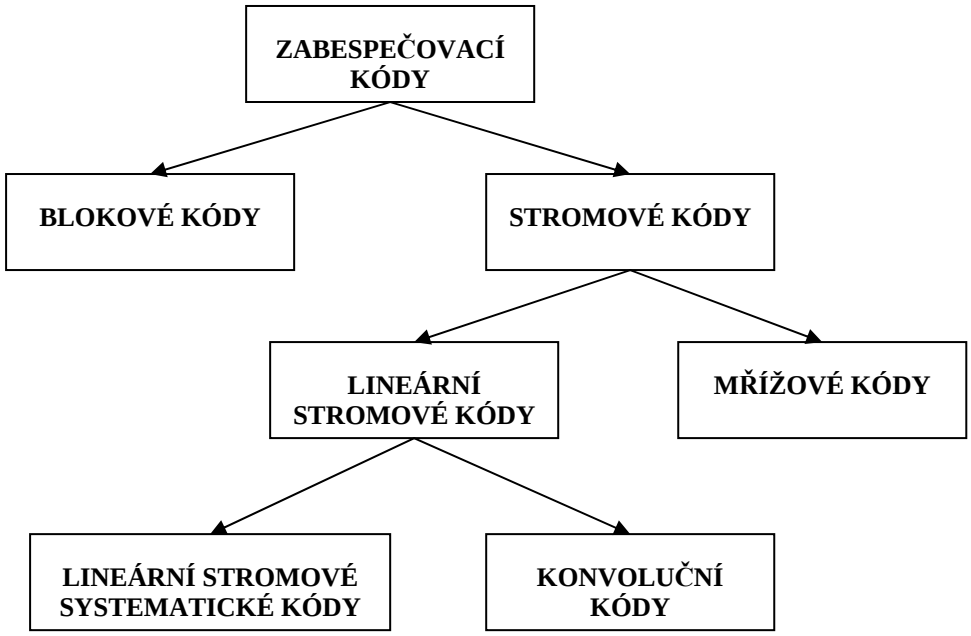
- **Lineární stromové kódy** - pro tyto kódy platí podmínka lineárnosti.
- **Mřížové kódy** - nebo-li nelineární stromové kódy, nevyhovují podmínce lineárnosti.

Lineární stromové kódy se dělí dle jejich systematičnosti (viz [4]). Systematičnost je vlastnost stromových kódů, kdy je úsek zabezpečené zprávy n_0 složen ze dvou částí

$$n_0 = k_0 + r_0 \tag{3.2}$$

kde k_0 jsou informační prvky a r_0 jsou zabezpečovací prvky. Uvedenou podmínku splňují:

- **Lineární stromové systematické kódy.**
- **Konvoluční kódy** - budou rozebrány v následující kapitole.



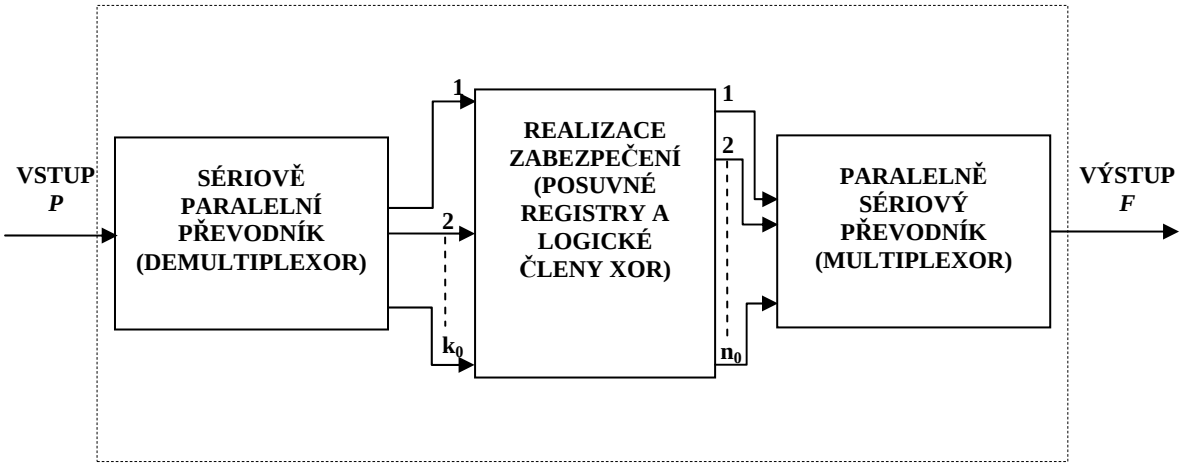
Obr. 3.1: Zařazení konvolučních kódů v klasifikaci zabezpečovacích kódů

3.3 Konvoluční kódy

Svůj název odvozují konvoluční kódy od latinského slova konvolut - propletenec. Z obrázku (Obr. 3.1) je zřejmé, že konvoluční kódy vychází z lineárních stromových kódů, které patří mezi kódy zabezpečovací [4].

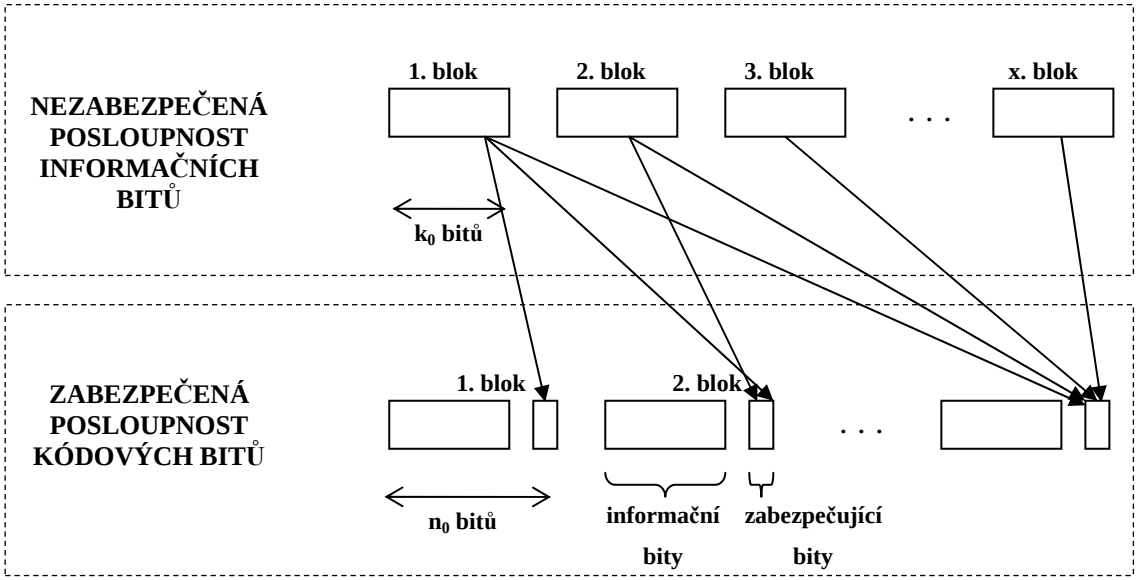
3.3.1 Kódování konvolučních kódů

Vznik zabezpečené posloupnosti z nezabezpečené je znázorněn na (Obr. 3.2). Při kódování se vstupní sériový tok nezabezpečených bitů, který je označen jako vektor *P*, rozdělí do k_0 dílčích vstupních toků. Zabezpečení se uskutečňuje pomocí kódovacích obvodů v bloku *Realizace zabezpečení* (posuvné registry) tak, že vzniká n_0 dílčích výstupních toků, které jsou pak převedeny na výstupní sériový tok zabezpečených bitů, označený jako vektor *F* [3].



Obr. 3.2: Schéma kódování obecného konvolučního kodéru

Zabezpečovací (kontrolní) bity jsou vytvářeny jednak pomocí informačních bitů právě přenášeného bloku a také pomocí informačních bitů bloků předcházejících. Schématické znázornění situace je na (Obr. 3.3). Tedy každé výstupní kódové slovo délky n_0 bitů obsahuje kromě informačního slova délky k_0 bitů také zabezpečovací slovo délky r_0 bitů. Z vlastnosti systematičnosti konvolučních kódů vyplývá, že kódovaná zpráva se může rozdělit na informační a zabezpečující část, čímž se dosáhne snadnějšího technického řešení kodéru i dekodéru [8].



Obr. 3.3: Schematické znázornění principu kódování kovolučními kodéry

Pro zadání kódu obecně je zapotřebí vyjádřit vztah mezi vstupními a výstupními dílčími bitovými toky kodéru. Na teoretické úrovni (tedy neuvažujeme-li možnost zadat kód schématem zapojení kodéru a dekodéru) existují dvě metody. Zadávání kódu souborem *vytvářecích mnohočlenů* a zadávání *vytvářecí maticí*. O prvním způsobu informuje literatura [4].

3.3.2 Zadání konvolučních kódů vytvářecí maticí

Funkce vytvářecí matice je následující. Vstupní bitový tok označíme opět jako vektor *P*, výstupní bitový tok jako vektor *F*. Vztah mezi uvedenými vektory lze vyjádřit maticovou rovnicí:

$$F = P \times \mathbf{G} \tag{3.3}$$

Zde **G** je vytvářecí matice konvolučního kódu a je polonekonečná. Jednu stranu má ohraničenou začátkem kódování a druhou stranu má ohraničenou délkou zprávy, která může být v krajním případě nekonečná.

Po přenosu *F* komunikačním kanálem se potřebujeme přesvědčit o správnosti přenosu. K tomu využijeme takzvanou *kontrolní matici* **H**. Ta se odvozuje z vytvářecí matice **G** pomocí operace transponace nad specifickou podmaticí matice **G**. Podrobně je tento problém popsán v literatuře [9].

Vynásobením přeneseného bitového vektoru *F* kontrolní maticí **H** získáme vektor syndromu *S*. Pokud je tento vektor nulový, pak byl přenos bezchybný. Pokud obsahuje vektor alespoň jednu jedničku, došlo při přenosu k proniknutí chyby. Vektor *S* se v dekodéru případně převádí na chybový vektor *E*, pomocí kterého lze přesně lokalizovat chybu v přenesené posloupnosti a následně ji korigovat.

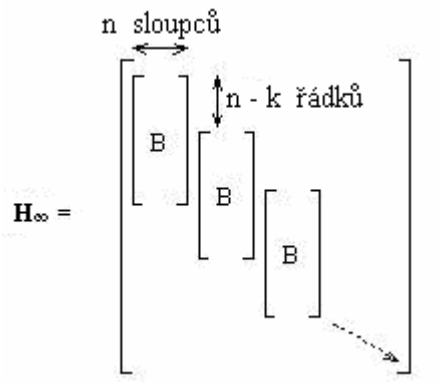
Podmínkou správného výsledku však je, aby při přenosu nebyly překročeny mezní zabezpečovací schopnosti kódu vyplývající z jeho Hammingovy vzdálenosti d_{min}, která je vyjádřena následujícími vztahy [9]:

Detekční schopnost: $$d_{\min} \geq t+1 \tag{3.4}$$

Korekční schopnost: $$d_{\min} \geq 2t+1 \tag{3.5}$$

kde *t* je maximální délka chyby proti níž je schopen kód přenášené bity zabezpečit. Z této vzdálenosti se pak u stromových kódů zabezpečujících proti shlukovým chybám odvozuje *ochranný interval A*, což je minimální počet bitů mezi dvěmi nezávislými shluky chyb vyžadovaný daným kódem pro bezchybný přenos.

Kontrolní matice **H** je opět polonekonečná. Je možno v ní najít určitou pravidelnou strukturu. Skládá se z podmatic, kterým říkáme blokové matice **B** (viz (Obr. 3.4)).



Obr. 3.4: Schéma kontrolní matice **H**

Tuto blokovou matici **B** lze tedy použít u některých kódů k jejich zadání místo vytvářecí matice. Výhodou tohoto způsobu je snazší hledání zapojení kodéru a dekodéru pro některé významné druhy kódů.

4 Iwadariho kód

Iwadariho kód je systematický konvoluční kód. Jeho velmi dobrou vlastností je, že nezpůsobuje nekonečný průnik chyby do vlastní informace, dojde-li k překročení mezní zabezpečovací schopnosti.

Přesněji, v přenosu se může vyskytnout shluková chyba o délce větší než je maximální délka shlukové chyby, proti níž konkrétní realizace Iwadariho kódu přenášenou posloupnost zabezpečuje. Případně se v přenosu mohou vyskytnout dvě nezávislé shlukové chyby ve vzdálenosti menší než je minimální ochranný interval této realizace Iwadariho kódu. V takovémto případě dojde při pokusu o opravu vzniklých chyb dekodérem Iwadariho kódu k potenciální ztrátě informace o maximální délce odpovídající ochrannému intervalu *A*. Naproti tomu u jiných konvolučních i stromových kódů obecně může při překročení těchto dvou parametrů kódu docházet k soustavnému narušování informace dekodérem i v dalších úsecích bezchybně přenesené zprávy.

4.1 Definice Iwadariho kódu

Jak již bylo řečeno výše, jednou z možností popisu konvolučního kódu je zadání pomocí *vytvářecí matice*, v některých případech pomocí *blokové vytvářecí matice*. Pro blokovou vytvářecí matici popisující Iwadariho kód platí následující vztahy (odvození v [1]):

$$\mathbf{B_0} = \left[m \cdot n_0 ; m \cdot k_0 \right] \tag{4.1}$$

Příslušná matice má *m* řádků, n_0 sloupců a její podmatice o *m* řádcích a k_0 sloupcích určuje vlastní zapojení kodéru. Zde *m* a k_0 jsou definovány:

$$m = \frac{n_0 \cdot \left(n_0 - 1 \right)}{2} + 2n_0 - 1 \tag{4.2}$$

$$k_0 = n_0 - 1 \tag{4.3}$$

Kde k_0 je počet informačních bitů v jednom přenášeném bloku a n_0 počet všech bitů v přenášeném bloku - tedy k_0 informačních a jeden korekční, viz schéma (Obr. 3.3).

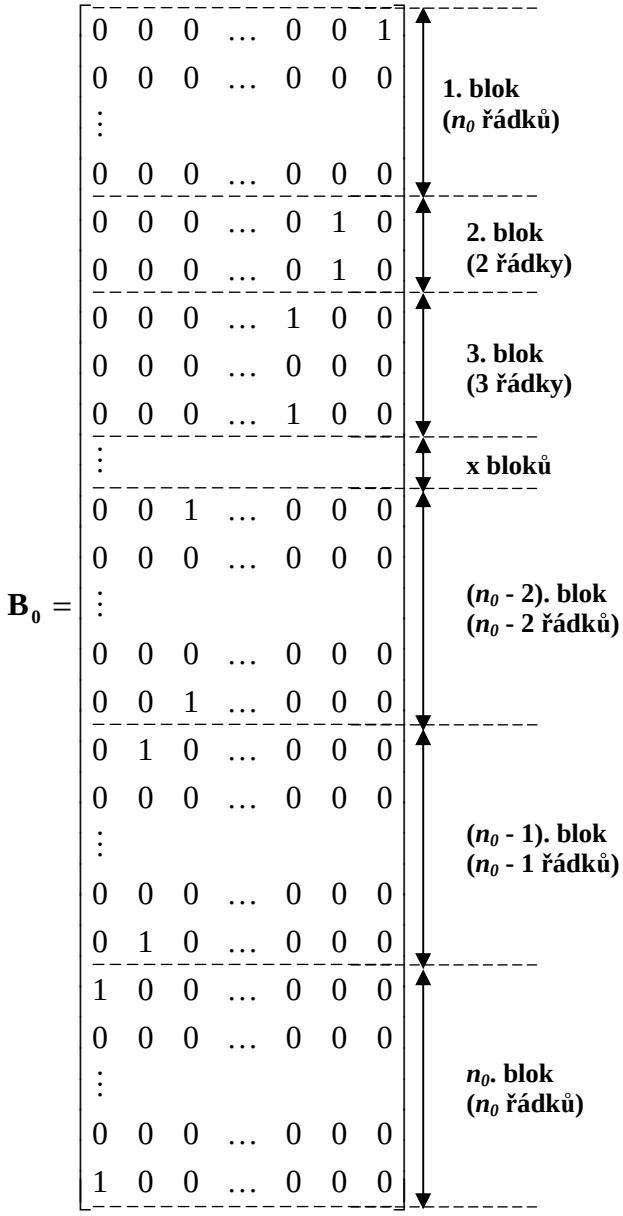
Dalšími definovanými parametry jsou *korekční schopnos*t *b* a *ochranný interval A*:

$$b \leq n_0 \tag{4.4}$$

$$A \geq n_0 \cdot m - 1 \tag{4.5}$$

Konkrétní realizace Iwadariho kódu bude schopna opravovat shluky chyb o délce až *b* bitů, ale pouze v případě, že tyto shluky budou od sebe vzdálené minimálně *A* bitů.

Vlastní bloková vytvářecí matice $\mathbf{B_0}$ je znázorněna na obrázku (Obr. 4.1).



Obr. 4.1: Bloková vytvářecí matice Iwadariho kódu s obecnou korekční schopností

4.1.1 Popis blokové vytvářecí matice Iwadariho kódu

Matice má m (tedy $\frac{n_0}{2}(n_0-1)+2n_0-1$) řádků a n_0 sloupců. Celkový počet řádků matice si rozdělíme do n_0 bloků, kde první bude mít n_0 řádků, druhý dva řádky, třetí tři řádky až n_0-tý blok bude mít n_0 řádků. Každý blok budou tvořit samé nuly, pouze ve druhém bloku ve druhém sloupci zprava na prvním a posledním řádku, ve třetím bloku ve třetím sloupci zprava na prvním a posledním řádku až v n_0-tém bloku v n_0-tém sloupci na prvním a posledním řádku bude jednička. A také v prvním bloku v prvním sloupci zprava na prvním řádku bude jednička.

Z této blokové vytvářecí matice je dále možné odvodit délku zpožďovacího registru a umístění logických členů *XOR* v zapojení kodéru Iwadariho kódu. Zmíněná problematika bude podrobněji rozebrána v kapitole zabývající se konstrukcí kodéru Iwadariho kódu.

4.1.2 Rozšiřitelnost Iwadariho kódu

Ze vzorců (4.2) až (4.5) vyplývá, že při požadované korekční schopnosti Iwadariho kódu je pevně daný ochranný interval, který je třeba pro správnou funkčnost kódování důsledně dodržovat. V praxi se však u určitého typu přenosového kanálu v prostředí můžeme setkat s výskytem shlukových chyb, jejichž maximální délka je odhadnutelná, stejně tak je odhadnutelná i pravděpodobnost výskytu těchto shlukových chyb (tedy průměrná vzdálenost mezi nezávislými chybami).

Aby se v takovýchto případech nemuselo kódování Iwadariho kódem doplňovat dalšími metodami a také, aby bylo možné zjistit nejlépe vyhovující zapojení kodéru a dekodéru, pro známou délku shlukových chyb a vzdálenost mezi jejich výskytem, byla odvozena také *rozšířená varianta Iwadariho kódu*. Rozšířený kód je možné zadat tak, aby korekční schopnost *b* i ochranný interval *A* odpovídal požadavkům z praxe.

Stupeň rozšíření značíme *i* a do uvedených vztahů se promítne následovně (odvozeno v [1]):

$$m = \frac{n_0 \cdot (n_0 - 1)}{2} + (2n_0 - 1) \cdot i \tag{4.6}$$

$$k_0 = n_0 - i \tag{4.7}$$

korekční schopnost: $$b \leq n_0 \cdot i \tag{4.8}$$

ochranný interval: $$A \geq n_0 \cdot m - 1 \tag{4.9}$$

Vlastní bloková vytvářecí matice se od blokové matice nerozšířeného kódu liší pouze tím, že za všechny bloky, na než je možné matici rozčlenit, se vsune *i* nulových řádek. Toto pozmění i indexy členů v syndromových rovnicích a tím vlastní pozice *XOR*ů v zapojení kodéru a dekodéru Iwadariho kódu. Logika odvození rovnic a zkonstruování kodéru a dekodéru je však naprosto stejná. Pro zjednodušení tohoto komplexního problému se tedy rozšířeným kódem nebudeme dále zabývat.

4.2 Konstrukce kodéru

Dle návrhu obvodové realizace v literatuře [1] se Iwadariho kodér skládá z registrů o celkové šířce $m \cdot k_0$ paměťových buněk, několika logických členů *XOR* napojených na specifická místa v registrech, jednoho multiplexoru a jednoho demultiplexoru. Multiplexor realizuje paralelně sériový převodník kódu, obsahuje n_0 vstupních a jeden výstupní signál. Demultiplexor realizuje sériově paralelní převodník kódu, obsahuje jeden vstupní a k_0 výstupních signálů.

Pozice paměťových buněk v posuvných registrech jejichž výstupy jsou přivedeny na vstupy logických členů *XOR* (počítající zabezpečení) jsou odvoditelné z blokové vytvářecí matice.

4.2.1 Označování prvků v blokové vytvářecí matici

Pro odvození pozic *XORů* je nutné nejprve odvodit takzvané *syndromové rovnice* kódu. Zpravidla se odvozují syndromové rovnice a obvodové zapojení kodéru pro určitou konkrétní hodnotu korekční schopnosti *b*. Cílem této práce je odvodit a realizovat obvodové zapojení komponent kodéru a dekodéru Iwadariho kódu o generické šířce, parametrizovatelné právě hodnotou požadované korekční schopnosti. Ke snadnějšímu řešení tohoto problému bude účelné zavést si vlastní způsob označení prvků blokové matice $\mathbf{B_0}$.

V dostupné literatuře ([1] až [9]) se označují jednotlivé prvky blokové matice následovně - jednotlivé sloupce jsou postupně zleva označeny písmeny latinské abecedy, jednotlivé členy matice jsou pak označeny písmenem podle sloupce, ve kterém se nachází, s indexem řádku tohoto prvku v matici, přičemž řádky se pro tento účel číslují odspodu počínaje jedničkou.

Takto označené prvky pak vystupují v jednotlivých syndromových rovnicích, což může působit u odvozování zapojení kodéru (a následně dekodéru) obecné zabezpečovací schopnosti značné problémy. Vzorce a algoritmy, jež by počítaly počty prvků mezi některými dvěmi takto označenými prvky (takto zjistíme šířku jednotlivých registrů), by musely přepočítávat písmenné označení prvků a jejich indexování do některé konvenční číselné soustavy (nejlépe desítkové nebo dvojkové), což by byl v tomto případě netriviální problém.

Pro účely naší práce budeme prvky v blokové vytvářecí matici označovat následovně - všechny prvky budou mít stejné označení, například *R*, a index bude udávat pořadí prvku v matici, přičemž prvky v matici budeme číslovat zprava prvním řádkem počínaje. Nebudeme ovšem číslovat žádny prvek v prvním sloupci zprava, takže při číslování bude vždy po nejlevějším prvku v jednom řádku následovat druhý nejpravější prvek následujícího řádku tak, jak to znázorňuje obrázek (Obr. 4.2). Výhoda takovéhoto značení se projeví v odvozování obecných syndromových rovnic.

$$\left[\begin{array}{cccc|c} R_4 & R_3 & R_2 & R_1 & 0 \\ R_8 & R_7 & R_6 & R_5 & 0 \\ R_{12} & R_{11} & R_{10} & R_9 & 0 \\ R_{16} & R_{15} & R_{14} & R_{13} & 0 \\ & & \ddots & & \end{array}\right]$$

Obr. 4.2: Naznačení zavedené konvence označování prvků v blokové vytvářecí matici

4.2.2 Výpočet pozice logických členů XOR

Zabezpečený vektor *F* získáme násobením vektoru *P*, představujícího vstupní bitový tok, vytvářecí maticí **G** (viz vzorec (3.3)), jež je složena z blokových vytvářecích matic (viz obrázek (Obr. 4.1)).

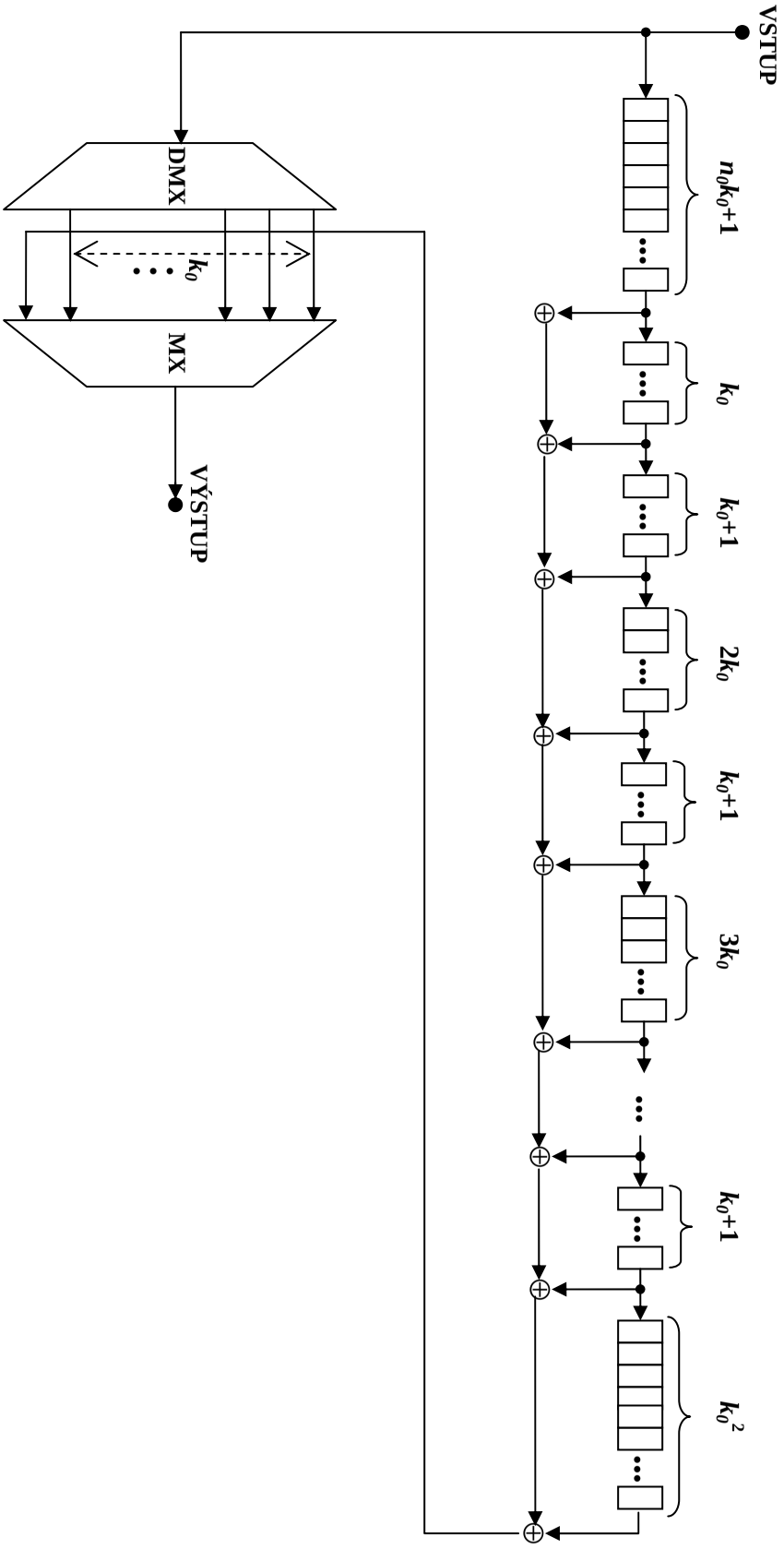
Zabezpečující bity výstupního bitového vektoru *F* budou tvořeny logickou nonekvivalencí vybraných informačních bitů vektoru *P* zvolených právě podle pozice jedniček v blokové vytvářecí matici $\mathbf{B_0}$.

Tento výpočet zabezpečení v kodéru prezentuje první syndromová rovnice. Její hodnota je tedy logickou nonekvivalencí všech členů blokové vytvářecí matice, jež mají hodnotu jedna. Označíme si tuto rovnici písmenem *s* a indexem 1:

$$s_1 = R_{n_0k_0+1} \oplus R_{n_0k_0+1+k_0} \oplus R_{n_0k_0+1+k_0+n_0} \oplus R_{n_0k_0+1+k_0+n_0+2k_0} \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0} \oplus \\ \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0} \oplus \ldots \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+k_0^2} \tag{4.10}$$

Indexy členů jsou odvozeny z obecné blokové matice (Obr. 4.1) za využití teorie řad.

Pro obvodovou realizaci odvozeného zabezpečujícího členu tedy budeme potřebovat registry o celkové šířce $m.k_0$ (tolik má členů bloková matice bez nejpravějšího sloupce). Logické členy *XOR* budou připojeny na výstup paměťových buněk, jejichž indexy jsou totožné s indexy v rovnici (4.10). Pomocí prezentovaného odvození a s přihlédnutím ke schématům obvodové realizace kodéru (uvedených ovšem pro konkrétní hodnoty korekční schopnosti - viz [1], [4], [8]) můžeme odvodit následující návrh obvodové realizace kodéru (Obr. 4.3).



Obr. 4.3: Schéma obvodového zapojení kodéru Iwadariho kódu

4.3 Konstrukce dekodéru

Dekodér se skládá z těchto bloků: multiplexor, demultiplexor, kodér přijímače (ten se opět skládá z registrů a logických členů *XOR*), generátor syndromu (což je vlastně jeden dvouvstupý logický člen *XOR*) a převodník $S \rightarrow E$ (převodník syndromového vektoru na chybový vektor - obsahuje k_0 dvouvstupých hradel *AND* a syndromových paměťových buněk - tolik existuje dalších syndromových rovnic potřebných k základní syndromové rovnici) a korekčních členů *XOR* v registrech kodéru přijímače.

4.3.1 Odvození zapojení převodníku $S \rightarrow E$

Zmíněný převodník zajišťuje vlastní korekci chybných bitů. K opravě bude využívat syndromové rovnice a jejich počet musí splnit požadavek existence alespoň dvou ortogonálních součtů k jednomu opravovanému bitu [12] (tedy v obou těchto rovnicích se opravovaný bit vyskytuje, zatímco ostatní bity se nesmí vyskytnout v obou rovnicích). Z teoretického hlediska můžeme další požadované syndromové rovnice k první odvozené (viz (4.10)) získat pouhým vhodným "posouváním v čase", tedy snížením všech indexů v první rovnici o stejnou hodnotu. Pro tento účel byla také vhodně navržena bloková vytvářecí matice.

Iwadariho kód je systematický, tudíž po k_0 informačních bitech následuje v přenosu jeden bit zabezpečující, odvozený právě pomocí první syndromové rovnice (4.10). Nabízí se tak možnost posouvat první syndromovou rovnici v čase právě o k_0 přenosových okamžiků. Takovouto myšlenku podporuje i bloková vytvářecí matice $\mathbf{B_0}$. V každém z jejích k_0 levých sloupců (a sloupec představuje skupinu prvků s indexy zvětšujícími se o k_0 - nebo-li jejich přenos je posunutý v čase právě o k_0 okamžiků) jsou vždy dvě jedničky pod sebou (viz (Obr. 4.1)). Znamená to, že posouvání postupně o k_0 okamžiků způsobí, že některá jednička se dostane na pozici jiné jedničky ve stejném sloupci. Získáváním nových syndromových rovnic tak, že zvýšíme všechny indexy předchozí rovnice o k_0, budou tyto rovnice spolu s první syndromovou rovnicí (4.10) použitelné pro korekci bitů za splnění požadované podmínky ortogonality.

Pro první snydromovou rovnici a z ní odvozenou druhou, jenž je zpožděná o k_0 členů, platí:

$$s_1 = R_{n_0k_0+1} \oplus R_{n_0k_0+1+k_0} \oplus R_{n_0k_0+1+k_0+n_0} \oplus R_{n_0k_0+1+k_0+n_0+2k_0} \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0} \oplus \\ \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0} \oplus \ldots \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+k_0^2} \tag{4.10}$$

$$s_2 = R_{n_0k_0+1+k_0} \oplus R_{n_0k_0+1+2k_0} \oplus R_{n_0k_0+1+2k_0+n_0} \oplus R_{n_0k_0+1+2k_0+n_0+2k_0} \oplus R_{n_0k_0+1+2k_0+n_0+2k_0+n_0} \oplus \\ \oplus R_{n_0k_0+1+2k_0+n_0+2k_0+n_0+3k_0} \oplus \ldots \oplus R_{n_0k_0+1+2k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+k_0^2} \tag{4.11}$$

Rovnice mají společný člen $R_{n_0k_0+1+k_0}$. Podle těchto syndromových rovnic se počítá zabezpečující bit v kodéru (označíme si jej s apostrofem) i dekodéru (označíme si jej bez apostrofu).

$$\left(s_1 \oplus s_1^{'}\right) \cdot \left(s_2 \oplus s_2^{'}\right) = x \tag{4.12}$$

V rovnici (4.12) je *x* rovno 1 právě v případě, že je bit odpovídající členu $R_{n_0k_0+1+k_0}$ přenesen do dekodéru chybně nebo nebyla dodržena ochranná vzdálenost. Při dodržení ochranné vzdálenosti totiž můžeme být chybný pouze jeden přenesený zabezpečující bit. Každý ze zabezpečujících bitů se počítá ze členů, které jsou od sebe vzdáleny méně než je ochranná vzdálenost *A* (protože všechny jsou odvozeny z jedné blokové matice, jejíž počet členů odpovídá právě ochrannému intervalu). Tudíž, aby se v rovnici (4.12) odlišovaly členy s_1 a s_1' a členy s_2 a s_2', musí být při splnění podmínek ortogonality (ty zajišťuje bloková vytvářecí matice) chybný právě člen $R_{n_0k_0+1+k_0}$. Pro jeho opravu pak platí vztah:

$$\left(R_{n_0k_0+1+k_0}\right)_{opravený} = \left(R_{n_0k_0+1+k_0}\right)_{prenesený} \oplus \left(s_1 \oplus s_1^{'}\right)\cdot\left(s_2 \oplus s_2^{'}\right) \tag{4.13}$$

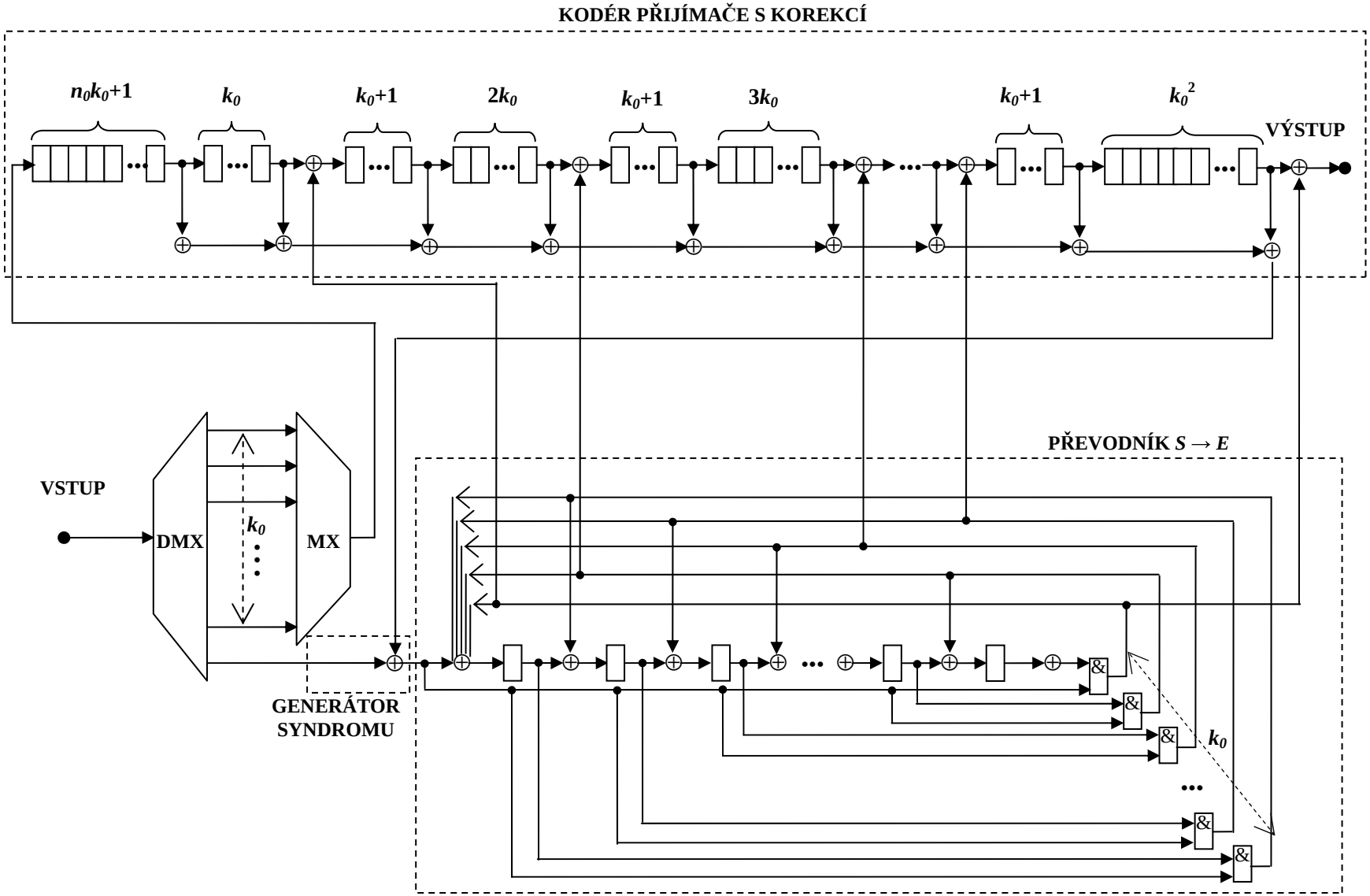
Pokračováním v postupném zvyšování všech indexů první syndromové rovnice bychom dostali celkově k_0+1 syndromových rovnic. Každá z nich (kromě první) spolu s první rovnicí určuje korekci některého z bitů, analogicky ke vztahům (4.12) a (4.13). Například pro poslední opravovaný bit (bit opravovaný až na výstupu registrů dekodéru, viz schéma dekodéru (Obr. 4.4)), bychom použili syndromovou rovnici k_0-krát zpožděnou o k_0 členů od první rovnice:

$$\begin{aligned} s_{k_0+1} = {} & R_{n_0k_0+1+k_0^2} \oplus R_{n_0k_0+1+k_0+k_0^2} \oplus R_{n_0k_0+1+k_0+n_0+k_0^2} \oplus R_{n_0k_0+1+k_0+n_0+2k_0+k_0^2} \oplus \\ & \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+k_0^2} \oplus \ldots \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+k_0^2} \oplus \\ & \oplus R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+2k_0^2} \end{aligned} \tag{4.14}$$

$$\begin{aligned} & \left(R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+k_0^2}\right)_{opravený} = \\ & \qquad = \left(R_{n_0k_0+1+k_0+n_0+2k_0+n_0+3k_0+\cdots+n_0+k_0^2}\right)_{prenesený} \oplus \left(s_1 \oplus s_1^{'}\right)\cdot\left(s_2 \oplus s_2^{'}\right) \end{aligned} \tag{4.15}$$

Rozdíl přeneseného zabezpečení a zabezpečení spočítaného v dekodéru (tedy $\left(s_a \oplus s_a^{'}\right)\ \forall a \in \{1\ldots n_0\}$) zajistí generátor syndromu tvořený jedním dvojvstupým členem *XOR*. Vlastní převodník $S \rightarrow E$ (tedy $\left(s_1 \oplus s_1^{'}\right)\cdot\left(s_a \oplus s_a^{'}\right)\ \forall a \in \{2\ldots n_0\}$) bude tvořen dvojvstupými členy *AND* (bude provádět součin syndromů), zpožďovacími buňkami (pomocí nich dosáhneme postupného zpožďování syndromové rovnice) a členy *XOR*, jež budou nulovat syndromy, které již iniciovaly opravu některého bitu. Oprava jednotlivých bitů (tedy $\left(R_b\right)_{opravený} = \left(R_b\right)_{prenesený} \oplus \left(s_1 \oplus s_1^{'}\right)\cdot\left(s_a \oplus s_a^{'}\right)$) pak bude tvořena logickými členy *XOR* v posuvném registru dekodéru řízenými výstupy součinů z převodníku $S \rightarrow E$.

Na základě tohoto odvození je možné uvést schéma (Obr. 4.4) obvodové realizace obecného dekodéru Iwadariho kódu (jehož ekvivalenty pro konkrétní korekční schopnosti je možné nalézt v literatuře [1], [4], [8]).



Obr. 4.4: Schéma obvodového zapojení dekodéru Iwadariho kódu

4.4 Synchronizace kodéru a dekodéru

V protichybových kódových systémech se obecně setkáváme s požadavkem na alespoň dvě přenosové rychlosti (a tedy i na různou synchronizační frekvenci určitých prvků): Jedné na rozhraních kodéru, dekodéru a koncových zařízení komunikačních systémů pro něž zprostředkovávají zabezpečení a druhé na rozhraní kodéru, dekodéru a přenosového kanálu, který je spojuje (viz [2]).

Existují dvě možnosti, jak řešit problém s požadavkem na různé přenosové rychlosti [2]:

- Rozčlenit vstupní tok bitů protichybového kódového systému na přerušované úseky - pakety. V tomto případě je možné použít ve všech prvcích stejnou synchronizační frekvenci. Je ovšem třeba implementovat určitý komunikační protokol, který by řešil různé stavy zařízení (vysílající, připravené, vytížené apod.)
- V případě zachování nepřetržitého sériového toku informací se v systému vyskytnou minimálně dvě různé výše popsané přenosové rychlosti. Jednotlivé prvky zabezpečovacího systému je tedy třeba synchronizovat (respektive řídit) odlišnými frekvencemi, tato situace se řeší implementací speciálního bloku *řízení*.

5 Návrh a realizace číslicových systémů

Číslicovým systémem míníme zpravidla elektromagnetický systém, který pracuje s informacemi reprezentovanými v číselné formě - nejčastěji ve dvojkové soustavě a to za využití booleovy algebry. Typickým příkladem je běžný personální počítač, ale také se s číslicovými systémy můžeme setkat ve formě mikrokontrolerů a vestavěných systémů v běžně používaných spotřebičích a zařízeních (v pračkách, mikrovlnných troubách, automobilech apod.)

V posledních desetiletích se dramaticky zvýšila využitelnost těchto systémů v nejrůznějších oborech, a tudíž je snaha co nejvíce zefektivnit a zlevnit proces jejich návrhu a realizace. Měnily a zdokonalovaly se postupy pro návrh a realizaci těchto systémů tak, aby bylo potřebné co nejméně úsilí, znalostí a zkušeností experta a naopak, aby byl proces návrhu co nejvíce automatizovaný a spoluřešený pomocí moderních návrhových systémů. Stále je ovšem možné vysledovat několik stupňů (popřípadě možností), kterými může proces návrhu číslicového systému projít.

5.1 Způsoby návrhu číslicových systémů

Obecně existuje řada způsobů návrhu číslicových systému. Mezi nejvýznamnější patří tyto [11]:

- **Slovní popis** - jsou v něm obsaženy veškeré podklady nezbytné pro přesnou definici činnosti obvodu a mělo by tedy být možné jej použít jako podklad pro fyzickou implementaci příslušného obvodu. Další zpracování takovéhoto návrhu ovšem není v současné době zautomatizovatelné a zpravidla slouží pouze jako zadání požadavku pro člověka-návrháře hardwaru.
- **Matematický popis** - je to formalizovaný slovní popis funkcí systému. Jeho největší význam spočívá ve verifikaci a validaci navrhovaného systému. V současné době ovšem neexistuje plná automatizace pro převedení takovéhoto stupně popisu ve fyzickou realizaci systému.
- **Obvodové schéma** - pro tento způsob návrhu existují pokročilé návrhové prostředí podporující návrh číslicového systému pomocí zakreslování schématu (například free CAD nástroj *Alliance* dostupný na URL http://www-asim.lip6.fr/recherche/alliance nebo *gEDA* dostupný na URL http://www.geda.seul.org). Výhodou je rychlé osvojení si návrhu, bez nutnosti učit se programovací či matematické postupy návrhu. Velkou nevýhodou ovšem je snížená přehlednost u složitějších systému a nutnost, aby návrhář měl od počátku konkrétní představu o cílové architektuře navrhovaného systému.
- **Programovací jazyk** - v současnosti je nejvyužívanějším způsobem návrhu. Vytvoří se popis chování obvodu ve vybraném programovacím jazyce pomocí vhodných jazykových konstrukcí jako jsou např. přiřazení, porovnání, smyčky, ap. Tyto jazyky se označují jako *HDL* (Hardware Description Language). Mezi nejpoužívanější HDL patří:

- ***Verilog,***
- ***Abel,***
- ***Handel C,***
- ***VHDL.***

První tři stupně popisu číslicového systému (slovní popis, matematický popis, obvodové schéma) jsme při návrh kodéru a dekodéru Iwadariho kódu již použili ve čtvrté kapitole. Pro konstruktivní simulaci a syntézu těchto komponent bude ovšem nejvhodnější využít poslední popsaný způsob - popis pomocí některého vhodného jazyka pro popis hardwaru. Pro naše účely zvolíme jazyk *VHDL*.

5.2 VHDL

Jazyk *VHDL* (*VHSIC* Hardware Description Language) vznikl v USA v rámci stejnojmenného programu, který navazoval na program *VHSIC* (Very High Speed Integrated Circuits), který byl zaměřen na výzkum a vývoj technologie vysoce rychlých integrovaných obvodů. Již v počátcích řešení tohoto programu se totiž ukázala potřeba standardního jazyka pro popis obvodů, který by umožňoval snadnou komunikaci návrhových dat [13]. S přihlédnutím k zjištěnému požadavku a dalším, které požadovali, aby nový jazyk používal co nejvíce konstrukcí převzatých z jazyku Ada, vznikl nový jazyk, ve kterém je možné popsat jednak jednotlivé hardwarové prvky, tak obecné algoritmy běžně používané při tvorbě softwaru.

Jazyk byl velmi brzy přijat i jako standart mezinárodní organizace pro elektrické a elektronické inženýrství, IEEE. Standardizování *VHDL* mimo jiné zapříčinilo rozšíření jazyka a jeho velkou přenositelnost.

Základními používanými pojmy jazyka *VHDL* je *komponenta* a *návrhová entita*. Komponenta slouží pro práci s navrhovanými systémy a podsystémy bez nutnosti znalosti jejího popisu. K popisu komponenty slouží právě návrhová entita. Návrhová entita modeluje číslicový systém libovolné složitosti. Může jít o hradlo, klopný obvod, ale i počítačový systém. Každá návrhová entita je charakterizována svým rozhraním a jedním nebo několika alternativními těly. Rozhraní obsahuje definice společné všem tělům a zahrnuje především informace viditelné zvnějšku. Jsou to vstup/výstupní porty definující kanály, kterými entita komunikuje s okolím, a tzv. generické parametry v případě, že entita je generická, tj. když se některý parametr určující její strukturu, chování nebo prostředí může lišit u jednotlivých výskytů entity.

Pro popis těla entity (někdy označeného jako architektura komponenty) je možné využít dva základní styly popisu [13]: *Strukturní* - popisuje především propojení prvků a popis chování (někdy též označovaný jako *behaviorální popis*). Popis chování má dvě základní formy - první formou je popis algoritmu, kdy transformace vstupů na výstupy prováděné komponentou popíšeme za pomocí

algoritmu a druhou formou je popis toku dat (neboli *data flow popis*), který popisuje prováděné transformace dat pomocí příkazů odpovídajících jazykům meziregistrových přenosů.

Obecně poskytuje behaviorální popis nejvyšší míru abstrakce, a tudíž pohodlný styl programování. Výsledná funkce navrhované entity je z jejího popisu snadno pochopitelná. Vysoká míra abstrakce (typická pro navrhování softwaru) ovšem nechává největší část práce na syntetizátoru výsledného hardwaru, který musí provést konkretizaci popisované entity a optimalizovat její strukturu. To ovšem nemusí dnešní syntetizační prostředky dokázat, a tudíž behaviorálně popsaná entita nemusí být ani syntetizovatelná - převeditelná do fyzické realizace. Tento styl popisu se tedy především hodí využívat pro návrhy určené k simulaci.

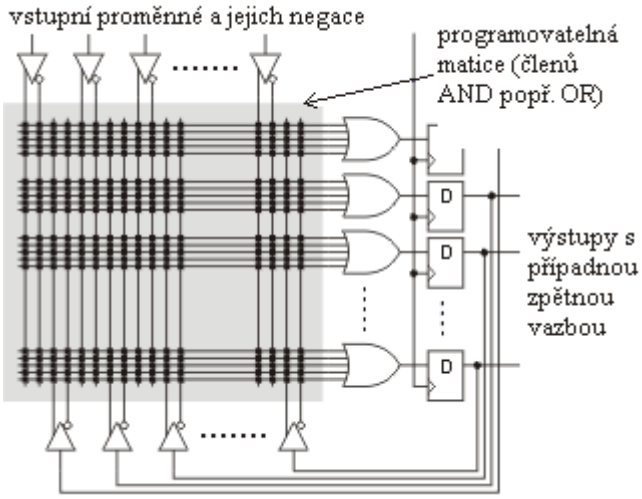
Naproti tomu strukturní popis je třeba tvořit s detailní znalostí vnitřní struktury výsledné entity a výsledná podoba popisu je pro neobeznámeného návrháře velmi obtížně čitelná. Popis detailně definující vnitřní strukturu entity dává jen velmi málo prostoru optimalizátoru a syntetizátoru pro rozsáhlejší zásahy do architektury výsledné komponenty. A tedy pro kvalitní návrh určený pro fyzickou realizaci číslicového systému je vhodnější využít propracovaný strukturální popis entity, který bude bez problému syntetizovatelný, a proces syntézy ovlivní podobu entity jen minimálně.

5.3 Technologie rekonfigurovatelného hardwaru

Spolu se skutečností, že je možné hardware navrhovat technikami zažitými pro navrhování software, je dalším důležitým faktorem, zlevňujícím a zefektivňujícím proces návrhu a realizace číslicových systému, existence technologie *rekonfigurovatelného* (někdy též označovaného jako programovatelného) hardwaru. Tato technologie umožňuje jednoduše měnit funkci součástek určených jako cíl návrhu číslicového systému a tím usnadňuje testování, úpravy na vyšší verzi a dokonce samoopravování číslicových systémů. Souhrnně se takovéto součástky obvykle označují jako *PLD* (Programmable Logic Device). Podle jejich vnitřní struktury (především způsobu jejich programování) a historického vývoje je možné je rozdělit do tří hlavních skupin[17]: klasické *PLD*, *CPLD* a obvody typu *FPGA*.

5.3.1 Klasické PLD

Společným rysem programovatelných obvodů této kategorie, je že programování se dosahovalo odpojováním spojů (realizováno proudovým impulzem na podobném principu jako přepalování pojistek) v propojené matici logických členů. Obvody v této skupině mají vnitřní strukturu analogickou se strukturou na obrázku (Obr. 5.1).

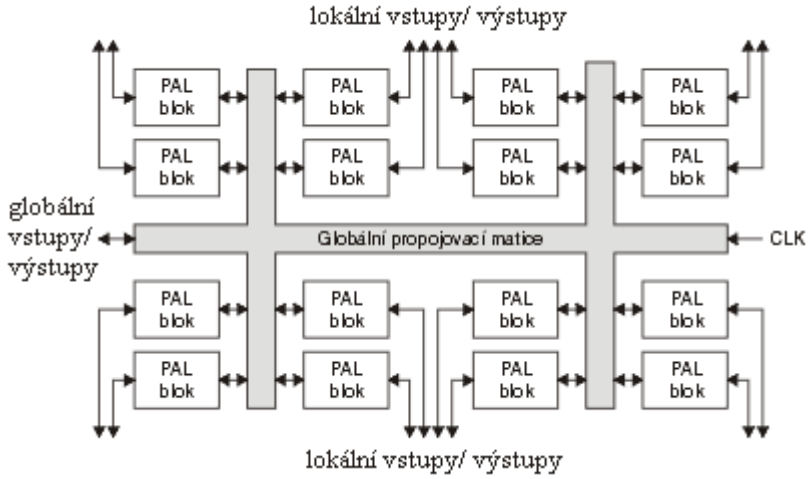


Obr. 5.1: Schematická struktura obvodů typu PAL (převzato z [17] a upraveno)

Rozlišujeme dva typy matic logických členů jež je možné odpojováním spojů programovat: matice *OR* (slouží k realizaci konjunktní formy - tedy součtu logických členů) a matice *AND* (slouží k realizaci disjunktní formy - tedy součinu logických členů). Struktura uvedená na obrázku (Obr. 5.1), tedy pouze s programovatelným polem *AND*, je typická pro konfigurovatelné součástky *PAL* (Programmamble Array Logic). Novější konfigurovatelné součástky označované jako *PLA* (Programmamble Logic Array) mají obecnější strukturu než *PAL*. Mají totiž programovatelnou nejenom matici logických součinů, ale i následující matici logických součtů. Díky výskytu přepalovacích pojistek také v matici *OR* jsou pomalejší a dražší než obvody *PAL*.

5.3.2 Komplexní PLD (CPLD)

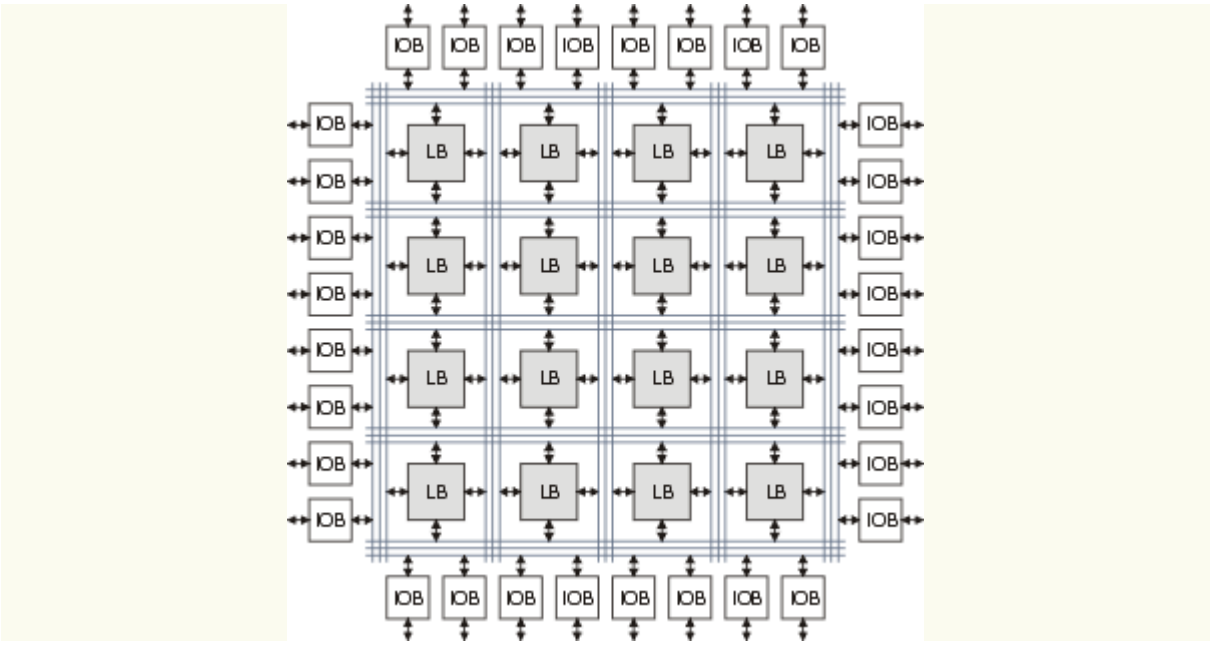
Klasické obvody *PLD* mají velmi omezené zdroje, proto umožňují realizovat pouze jednodušší funkce. Avšak výrobci začali sdružovat více takovýchto obvodů na jednom čipu spolu s nutnými prostředky pro propojení. Nově vzniklé obvody se většinou označují jako *CPLD* (Complex Programmable Logic Device). K vlastnímu naprogramování těchto součástek se používá naprogramování paměťových buněk *EEPROM*. Typická struktura obvodu *CPLD* je znázorněna na obrázku (Obr. 5.2).



Obr. 5.2: Schématická struktura obvodů typu CPLD (převzato z [17] a upraveno)

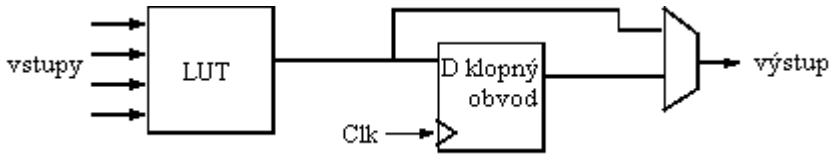
5.3.3 Obvody typu FPGA

Z poměrně striktní a tím omezující vnitřní struktury *CPLD* obvodů se v polovině osmdesátých let za spoluúčasti firmy Xilinx vyvinuly obvody typu *FPGA* (Field Programmable Gate Array). *FPGA* mají z programovatelných obvodů nejobecnější strukturu a obsahují nejvíce logiky. Současné největší obvody *FPGA* obsahují až 6 milionů ekvivalentních hradel (což je až tisíckrát více než tomu bylo u typických obvodů *CPLD*). Velkou odlišností od součástek *CPLD* je v jejich programování. Konfigurace *FPGA* je vždy při zapnutí napájení nahrávána z paměti (nejčastěji typu *SRAM*) a tedy funkce obvodu není okamžitě dostupná. Naproti tomu tato vlastnost umožňuje konfiguraci za běhu měnit a to i samotným hardwarem, který tak získává schopnost samoopravitelnosti a seberegulace.



Obr. 5.3: Schématická struktura obvodů typu FPGA (převzato z [17] a upraveno)

Typická struktura obvodů *FPGA* je na obrázku (Obr. 5.3). Bloky označené jako *IOB* (Input/Output Block) poskytují obousměrné rozhraní mezi vstupně výstupními piny *FPGA* a vnitřní výpočetní logikou. Popisované bloky nabízejí tři cesty pro tok signálů - vstupní, výstupní a třístavovou - a obvykle obsahují registr, budič, multiplexor a ochranné obvody. Bloky *LB* (Logic Block) představují vlastní programovatelné logické bloky. Všechny bloky mohou být různě propojeny globální propojovací maticí. Nejpoužívanější struktura konfigurovatelného logického bloku je znázorněna na obrázku (Obr. 5.4). Obsahuje blok *LUT* (Look-Up Table) poskytuje možnost pro sestavení obecné funkce čtyř proměnných, D klopný obvod, využitelný jako jednobitový registr a multiplexor pro přepínání výstupů (U různých výrobců se může tato struktura lišit - a to především v počtu zmíněných bloků).



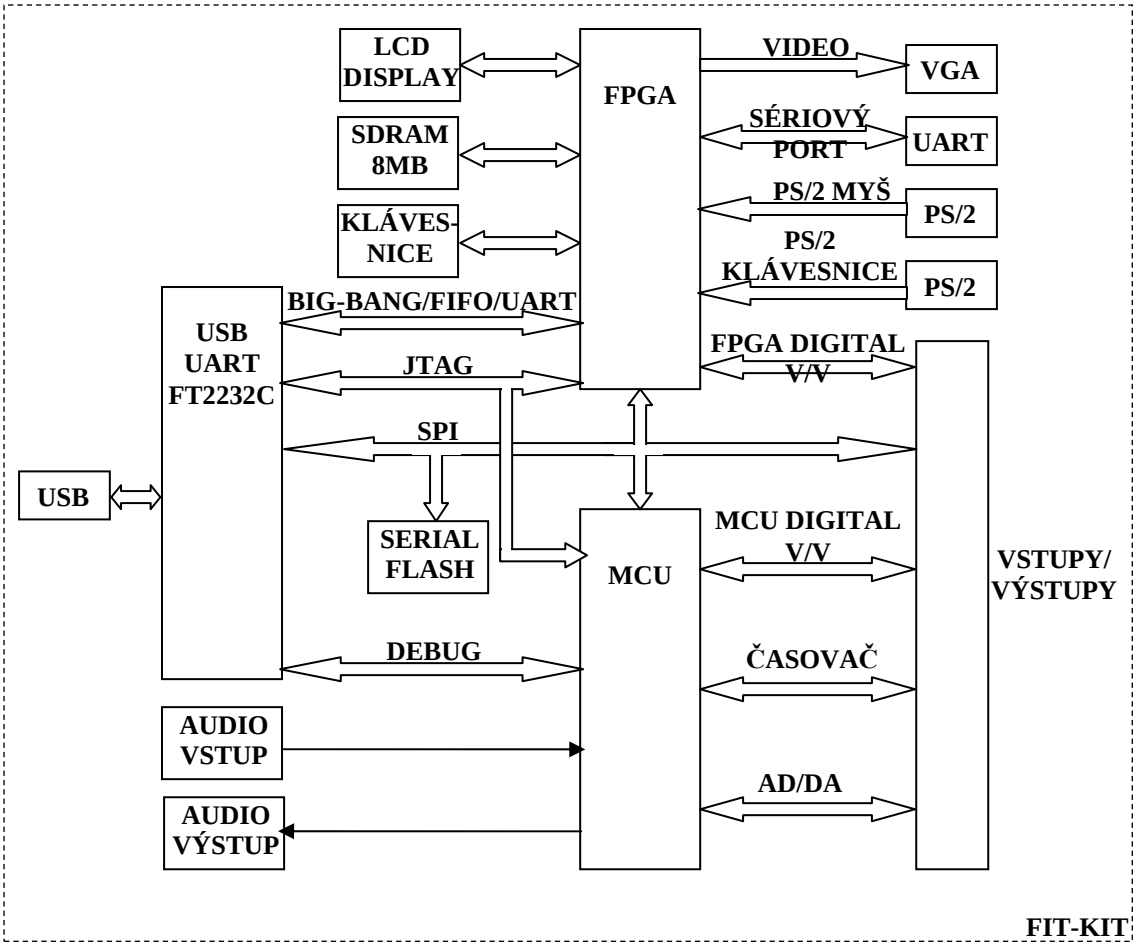
Obr. 5.4: Schéma typického logického bloku obvodu FPGA

Kromě bloků znázorněných na předchozích obrázcích integrují výrobci do *FPGA* další prvky. Většina moderních *FPGA* obsahuje několik bloků rychlé synchronní statické paměti *RAM* (označované jako *SRAM*). Velmi často obvody *FPGA* obsahují *PLL* (Phase Locked Loop) nebo *DLL* (Delay Locked Loop) pro obnovení charakteristik hodinového signálu, případně pro násobení nebo dělení jeho frekvence.

V současné době se vývojem obvodů *FPGA* na trhu zaobírá řada velkých firem - například Altera, Atmel, Actel, Xilinx. Patrně nejproduktivnějším dodavatelem je firma Xilinx (ta také stála u zrodu obvodů *FPGA*). Od této firmy také pochází také *FPGA* obvod typu *XC3S50* z rodiny *Spartan-3*, na němž je možné pracovat na platformě FIT-kit.

5.3.4 FIT-kit

FIT-kit je vývojová deska jejíž součástí je výše zmíněný *FPGA* čip spolu s mikrokontrolérem a periferiemi jako *LCD* display, maticová klávesnice, audio vstup/výstup a podobně (podrobnější schéma nabízí blokové schéma FIT-kitu, viz (Obr. 5.5) [14]).



Obr. 5.5: Blokové schéma platformy FIT-kit

Struktura *FPGA* na čipu *Spartan-3* odpovídá schématickému obrázku (Obr. 5.3), jeden logický blok ovšem poskytuje více elementárních obvodů než jak je zobrazuje schéma (Obr. 5.4). Jeden logický blok (*LB*) obvodů *FPGA* ve FIT-kitu se skládá ze čtyř menších struktur - takzvaných *slice*. Každý slice je analogií logického bloku na schématickém obrázku (Obr. 5.4) a konkrétně nabízí dva čtyřvstupé bloky *LUT* (využitelné také jako funkční generátory libovolné funkce čtyř proměnných), dva D klopné obvody a dva multiplexory.

Kromě logických bloků a vstupně-výstupních bloků (kterých je na *Spartanu-3* 124 s podporou různých používaných komunikačních standardů) tak jak je vyobrazeno na schématu (Obr. 5.3) obsahuje *FPGA* čip navíc také dva bloky *DCM* (Digital Clock Manager) určené pro úpravu hodinového signálu (například jeho dělení, násobení, fázového posouvání), čtyři bloky pamětí *RAM* o velkosti 18kB a čtyři násobičky s dvěma 18 bitovými operandy a jedním 36 bitovým výstupem (násobičky je ovšem možné zapojit též do kaskády, jejich obvodové řešení v rámci těchto bloků poskytuje mnohem rychlejší a méně prvků zabírající řešení násobení, než by tomu bylo za použití běžných logických bloků) [15].

FPGA FIT-kitu je možné konfigurovat naprogramováním *MCU* mikrokontroléru, který po restartu nahraje konfiguraci z paměti *FLASH* do *FPGA* (popřípadě je i možné konfigurovat *FPGA*

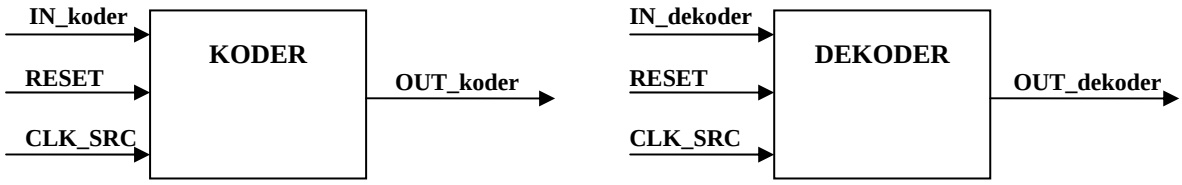
přímo za využití sériové komunikace přes vhodný terminál). Je to tedy vhodná cílová platforma pro realizaci demonstračních výukových číslicových systémů, jako například kodéru a dekodéru Iwadariho kódu. Závěrečnou etapou této práce tedy bude - po implementaci a úspěšném odsimulování komponent ve vhodném vývojovém prostředí - zrealizovat tyto komponenty v *FPGA* čipu FIT-kitu.

6 Implementace a simulace kodeku

Analýza problému a částečný návrh řešení byly provedeny v předchozích kapitolách teoretickým odvozením kodeku a schématu jeho obvodového zapojení. V této kapitole se budeme zabývat návrhem implementace a vlastní implementací komponent kodéru a dekodéru Iwadariho kódu v programovacím jazyce *VHDL*. Z předchozí kapitoly je zřejmé, že je to nejvhodnější způsob modelování rozsáhlejších číslicových systémů. Také z předchozí kapitoly vyplývá, že ve *VHDL* existují dvě možnosti jak komponenty popsat: pomocí behaviorálního anebo strukturálního popisu. A protože může mít jedna entita více alternativních těl (lišících se konkrétní podobou popisu) - zvolíme oba způsoby popisu a v závěru vyhodnotíme jejich přínosy a nedostatky.

6.1 Rozhraní a univerzálnost komponent

I když budou mít komponenty *kodér* a *dekodér* dva různé popisy chování, musí být jejich rozhraní jednotné. K popisu rozhraní ve *VHDL* slouží konstrukt *entity* (viz kapitolu 5.2 *VHDL*). Rozhraní entity obsahuje vstupní a výstupní porty. Podle navržených obvodových schémat (viz (Obr. 4.3) a (Obr. 4.4)) je vstupem a výstupem obou komponent sériový bitový tok určený k zakódování popřípadě dekódování. Kromě toho by mělo být možné započít přenos z jistého definovaného vnitřního stavu komponent - tento vnitřní stav nejlépe vyvoláme signálem *RESET*. Také je nutné vstupní sériový bitový tok vhodně vzorkovat, minimálně z tohoto důvodu je potřebné zajistit komponentám synchronizační signál (tento signál bude také použit v bloku *řízení*, jak bude uvedeno dále). Blokové schéma komponent kodér a dekodér (naznačující pouze jejich rozhraní) zobrazuje blokové schéma (Obr. 6.1).



Obr. 6.1: Blokové schéma komponent kodéru a dekodéru Iwadariho kódu

Rozhraní entity ovšem také obsahuje takzvané generické parametry, pomocí kterých lze odlišit strukturu různých výskytů entity. Toto vystihuje obvodový návrh kodéru a dekodéru odvozený v předchozích kapitolách. V nich bylo uvedeno obecné schéma zapojení kodéru a dekodéru zabezpečující komunikaci proti zvolené délce shlukových chyb. Ve schématech byla použita proměnná *k*. Tato proměnná by byla tedy i vhodným generickým parametrem. Jelikož proměnná *k* zároveň udává i délku bloku informačních bitů zabezpečeného proti shlukové chybě, zvolíme si pro

generický parametr uživatelsky přívětivější pojmenování, které po uživateli komponenty nevyžaduje detailní znalost souvislostí kódování pomocí Iwadariho kódu. Popis komponent s rozhraním uvedeným na obrázku (Obr. 6.1) a s odvozeným generickým parametrem je v jazyce *VHDL* následující:

```
entity IwKoder is GENERIC (block_length: integer := 4);
    port (
        IN_koder  : in  std_logic;     -- seriovy vstup koderu
        OUT_koder : out std_logic;     -- seriovy vystup koderu
        RESET     : in  std_logic;     -- reset (aktivni v 1)
        CLK_SRC   : in  std_logic      -- synchronizacni siganl, je
                                            entitou dale delen
    );
end IwKoder;

entity IwDekoder is GENERIC (block_length: integer := 4);
     port (
        IN_dekoder  : in  std_logic; -- seriovy vstup dekoderu
        OUT_dekoder : out std_logic; -- seriovy vystup dekoderu
        RESET       : in  std_logic; -- reset (aktivni v 1)
        CLK_SRC     : in  std_logic  -- synchronizacni siganl, je
                                         entitou dale delen
    );
end IwDekoder;
```

6.2 Strukturální popis

Výsledkem teoretického odvození kodeku je schématický návrh obvodové realizace kodéru i dekodéru Iwadariho kódu o obecné korekční schopnosti. Obvodové schéma je ideální pro přímý přepis do strukturálního kódu ve *VHDL*. Před vlastní implementací je ovšem potřeba navrhnout řešení několika dílčích problémů: Jak popsat opakující se části obvodové struktury s měnícími se parametry v závislosti na korekční schopnosti kodeku - v obvodovém schématu jsou zaznačeny třemi tečkami. Jak přesně použít multiplexory a demultiplexory ve funkcích sériově-paralelních a paralelně-sériových převodníků a také jak řešit problém odlišnosti vstupní a výstupní rychlosti komponent (viz kapitola 4.4 Synchronizace kodéru a dekodéru).

6.2.1 Popis opakujících se struktur

Podle navržených schémat zapojení (viz (Obr. 4.3) a (Obr. 4.4)) se vynechané opakující se struktury skládají z navzájem propojených registrů a logických členů *XOR*, jejichž počet a velikost (u registrů) je odvoditelná z parametru *k* (jenž udává korekční schopnost zabezpečujícího kódu). U strukturálního popisu komponent využíváme pro popis opakování příkaz *GENERATE*, což je jistá forma makra. Příkaz *GENERATE* se často používá právě ve spojení s příkazy sloužícími k propojování jednotlivých komponent.

Záhlaví příkazu *GENERATE* je analogické záhlavím počítaných cyklů běžně používaných při programování softwaru. K popisu určitých odlišností jednotlivých struktur je tedy využitelná řídicí

proměnnou počítaného cyklu *GENERATE*. Pro vyjádření obecné struktury registrů počítajících zabezpečení v kodéru (ale též i v dekodéru), kde je třeba propojovat za sebe registry o určitých šířkách a jejich jednotlivá propojení připojit na logické členy *XOR*, je možný tvar tohoto příkazu následující:

```
FOR i IN pocatecni_hodnota TO konecna_hodnota GENERATE
 posuvny_registr(sirky f(i)):     vstupem je vystup předchozího registru
                                  vystup pripoj na XOR
                                  vystup pripoj na nasledujici registr
END GENERATE;
```

Tento pseudokód je třeba doplnit o konkrétní počáteční a koncovou hodnotu řídicí proměnné, o konkrétní podobu funkce *f(i)*, jež bude udávat šířku registrů a v neposlední řadě o vlastní realizaci propojení prvků. První dva požadavky je možné odvodit z vytvářecí matice (viz (Obr. 4.1)) anebo schématu zapojení (viz (Obr. 4.3)). Z obou obrázků je zřejmé, že délka prvního registru je nepravidelná a délky ostatních registrů jsou střídavě konstantní (k+1) a střídavě členy aritmetické posloupnosti s koeficientem k a s k členy (tedy k, 2k, 3k … k^2). Vzrůstající délku registrů bude možné popsat právě pomocí řídicí proměnné - její rozsah by měl odrážet počet členů zmíněné aritmetické řady (tedy k). Dále by mělo být v jednotlivých cyklech vystihnuté střídání registrů konstantní délky a vzrůstající délky a také nepravidelnost délky prvního registru. Uvedený pseudokód by s ohledem na popsané požadavky bylo možné upřesnit například následovně:

```
FOR i IN 1 TO k GENERATE
 První cyklus:
         posuvny_registr(sirky n*k+1)
         posuvny_registr(sirky k)
         vzajemne propojeni registru a pripojeni na XOR
 Ostatní cykly:
         posuvny_registr(sirky k+1)
         posuvny_registr(sirky i*k)
         vzajemne propojeni registru a pripojeni na XOR
END GENERATE;
```

Vlastní provedení propojení registrů je implementačním detailem, který je podrobně zdokumentovaný v přiložených zdrojových kódech, jejichž výčet bude uveden dále. V komponentě dekodér se další opakování struktur ve schématu objevilo v části převodníku syndromového vektoru na chybový vektor - zde se k-krát za sebou opakuje stejná struktura zpožďování syndromu, jeho násobení se vstupním syndromem a korekce příslušných bitů zpožděných v posuvném registru. Bude tedy velmi účelné i tuto strukturu namodelovat pomocí příkazu *GENERATE*.

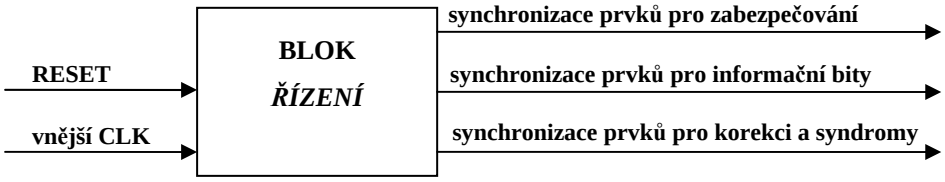
6.2.2 Synchronizace prvků kodéru a dekodéru

Podle kapitoly 4.4 musíme při implementaci vyřešit problém různého objemu dat na vstupu a výstupu kodéru i dekodéru (u každé z těchto komponent je buď vstup anebo výstup obohacen o zabezpečení, které oproti nezabezpečenému vstupu/výstupu zvětšuje počet zpracovávaných dat). Existují dvě

řešení - generovat pro vnitřní prvky kodéru a dekodéru minimálně dva různé synchronizační signály pomocí prvku *řízení*, anebo využít přerušování přenosového toku dat, a tedy zavést do komunikace nějaký protokol. Pro naše účely jsme zvolili první možnost - kodér a dekodér byly totiž navrženy pro nepřerušovaný přenos dat. Navíc také nebude nutné vytvářet komunikační protokol, který tak může být plně v režii nadřazených vrstev komunikačního zařízení využívající takto navržené zabezpečující komponenty.

Nyní je třeba analyzovat, kolik bude v komponentách kodéru a dekodéru potřeba různých synchronizačních signálů (popřípadě řídicích signálů - záleží na úhlu pohledu, zda budeme generované signály blokem *řízení* považovat za řídicí signály nebo synchronizační signály). Určitě budou potřebné synchronizace pro práci se zabezpečeným tokem dat (to se týká prvků generujících výstup kodéru a zpracovávajících vstup dekodéru) a s nezabezpečeným, čistě informačním, tokem dat (to se týká prvků zpracovávajících vstup kodéru a tvořících výstup dekodéru). K těmto dvěma synchronizacím (které se obecně musí vyskytnout u všech protichybových kódových systému pracujících s nepřerušovaným tokem dat) v našem případě ještě přibude synchronizace pro prvky pro práci se syndromy a korekci bitů. Ze 4. kapitoly totiž vyplývá, že k přenosu zabezpečujícího bitu, výpočtu syndromu chyby a případné korekci chyby v dekodéru dochází vždy po přenesení *k* informačních bitů.

Budeme tedy potřebovat tři různé synchronizační (řídicí) signály. Komponenta blok řízení, kterou budeme tyto signály generovat, bude pravděpodobně k výpočtu časování těchto signálu používat určitý hodinový signál. Navíc bude vhodné mít možnost určit počáteční stav řízení - tedy možnost resetovat komponentu. Tím je dáno rozhraní potřebné komponenty (Obr. 6.2).

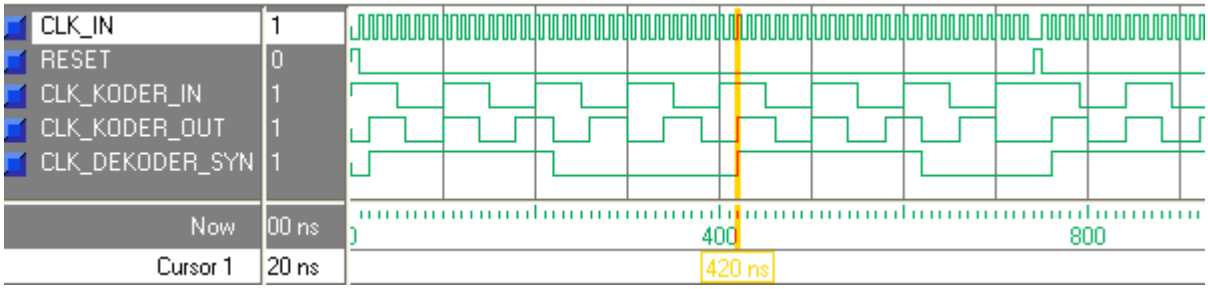


Obr. 6.2: Blokové schéma komponenty blok řízení

Dále je nutno vzít v úvahu generičnost kodéru a dekodéru, které bude blok *řízení* řídit. S ohledem na generičnost pak takovémuto rozhraní v popisu v jazyce *VHDL* odpovídá následující konstrukce:

```
entity Control_block is GENERIC (block_length: integer := 4);
  port (
    RESET           : in std_logic;  -- reset (aktivni v 1)
    CLK_IN          : in std_logic;  -- synchronizace na vstupu
    CLK_KODER_IN    : out std_logic; -- synchronizace vstupnich dat
                                        vstupujicich do koderu
    CLK_KODER_OUT   : out std_logic; -- synchronizace zabezpecenych dat
                                        mezi koderem a dekoderem
    CLK_DEKODER_SYN : out std_logic; -- synchronizace zabezpecujicich
  );                                    bitu
end Control_block;
```

Pro délku bloku 4 informační bity by časový průběh signálů výše popsané komponenty měl vypadat jako je uvedeno na obrázku (Obr. 6.3). Bloky řízení podobného charakteru jsou tvořeny čítači realizujícími dělení vstupního synchronizačního signálu. Nejsnazší ovšem bude popsat potřebnou komponentu behaviorálně za pomocí proměnných inkrementovaných podle hodnoty vstupní synchronizace. V takovémto popisu syntetizátor rozpozná zamýšlené zapojení čítačů. Uvedený signálový průběh byl vytvořen simulováním takto popsané komponenty (kompletní, plně okomentovaný kód je součástí elektronických příloh této práce).



Obr. 6.3: Časový průběh signálů komponenty blok řízení

Pro úplnost je třeba dodat, že pro naši komponentu jsme použili techniku dělení hodinového signálu. Tímto způsobíme, že komponenty kodeku budou pracovat na nižší frekvenci než bude frekvence synchronizace dodaná bloku řízení. Tato technika je tedy snadno použitelná, pokud cílová architektura disponuje dostatečně rychlou synchronizací na to, aby dělení neovlivnilo maximální pracovní frekvenci cílových obvodů. V opačném případě je možné použít takzvané bloky *PLL* (viz kapitolu 5.3.3) určené pro násobení hodinového signálu. Tato technika je ovšem náročnější na realizaci a také je méně přenositelná (je třeba přítomnost obvodu *PLL*).

Vlastní odvození behaviorálního popisu této komponenty je nastíněno v kapitole 6.3.3.

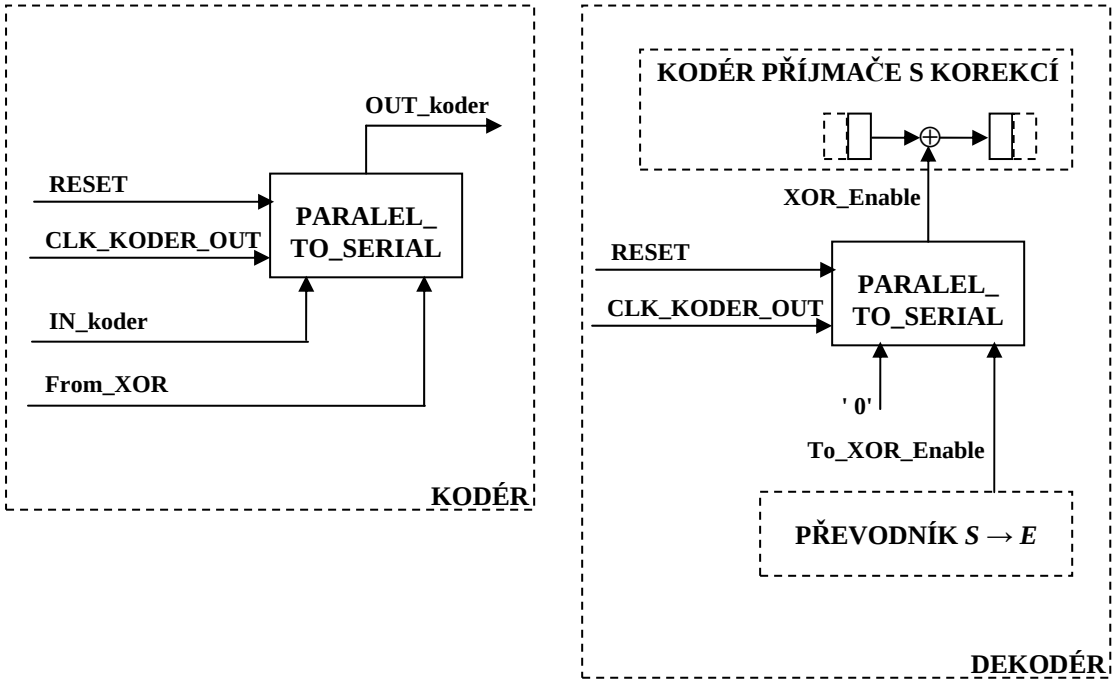
6.2.3 Práce se vstupem a výstupem

Přizpůsobení vstupu a výstupu kodéru a dekodéru Iwadariho kódu je v obvodové realizaci navržené podle literatury [1] řešené použitím multiplexoru a demultiplexoru ve funkcích sériově/paralelního a paralelně/sériového převodníku. Podrobnější analýzou tohoto problému je ovšem možné nalézt efektivnější řešení.

U vstupů obou komponent je dostačující, aby prvky pracující se vstupem jej vzorkovaly s odpovídající frekvencí. Pro toto je již z předchozí kapitoly odpovídajícím způsobem navržena komponenta blok řízení. Pak tedy budou zpožďovací posuvné registry kodéru načítat bity ze vstupu s frekvencí určenou signálem *CLK_KODER_IN*, se stejnou frekvencí budou načítat vstup zpožďovací posuvné registry dekodéru a s frekvencí *CLK_DEKODER_SYN* budou načítat vstup zpožďovací syndromové buňky dekodéru.

Výstup dekodéru je tvořen výstupem poslední paměťové buňky zpožďovacího posuvného registru (viz schéma obvodového zapojení na schématu (Obr. 4.4)). Není jej tedy třeba dále upravovat

žádnou komponentou. Naproti tomu výstup kodéru je tvořen vstupními informačními bity obohacenými o zabezpečující bity. Na výstup je tedy potřebné přepínat bity informační a zabezpečující, a to s takovou frekvencí, aby se v sériovém přenosu vyskytly všechny. Komponenta by tedy měla tyto bitové toky přepínat na výstup s frekvencí určenou signálem *CLK_KODER_OUT*. Navrhneme-li tuto komponentu tak, aby měla dva vstupy (jeden pro informační bity druhý pro zabezpečující bity) a druhý vstup přepínala na výstup jednou po *k* taktech, bude tato komponenta využitelná kromě přepínaní výstupu kodéru také pro povolení opravy bitů v dekodéru (zde totiž prvky iniciující opravu pracují s frekvencí *CLK_DEKODER_SYN* opravit je ovšem třeba pouze jeden bit v posuvném registru pracujícím na frekvenci *CLK_KODER_IN* - tyto frekvence také odpovídají frekvencím přepínaných vstupů v kodéru). Povolení opravy realizované pomocí logického členu *XOR* je totiž snadno realizovatelné přepínáním mezi signálem iniciujícím opravu a logickou hodnotou 0 (výsledkem výrazu s operaci *XOR* pak bude vždy druhý operand - nedojde tedy k jeho opravě). Blokové schéma této komponenty zapojené v kodéru pro přepínání výstupu a zapojené v dekodéru pro povolování oprav je naznačeno na schématu (Obr. 6.1).



Obr. 6.4: Blokové schéma zapojení komponenty *Paralel_To_Serial* v kodéru a dekodéru

Tato komponenta je popsatelná například pomocí čítače na základě jehož přetečení/nepřetečení se na výstup přepne první/druhý vstup. Nejvhodnější bude takovouto komponentu popsat behaviorálně za pomocí inkrementované proměnné, ve které syntetizátor snadno rozpozná zamýšlený čítač.

6.2.4 Komponenty využité pro strukturální popis kodeku

Cílové komponenty kodéru a dekodéru byly popsány pomocí vzájemného propojování dílčích komponent podle schémat (Obr. 4.3) a (Obr. 4.4) s úpravami, jež byly popsány výše. Tato kapitola obsahuje seznam všech komponent, které byly namodelovány a použity.

- **IwNKoder_struct** - generická komponenta kódující vstupní sériový bitový tok pro jeho ochranu proti násobné chybě jejíž délka je vybrána generickým parametrem (ve skutečnosti zabezpečuje proti chybě o jeden bit delší - a to samotným vkládáním bitů, které nenesou užitečnou informaci).
  Vstupy: *IN_koder* - sériový bitový tok určený k zabezpečení.
  RESET - resetovací signál uvádějící kodér do počátečního stavu.
  CLK_SRC - synchronizace pro zajištění funkčnosti bloku řízení (Control_block).
  Výstup: *OUT_koder* - zabezpečený sériový bitový tok určený k přenosu.
- **IwNDekoder_struct** - generická komponenta dekódující přenesený vstupní sériový bitový tok opravující případné chyby přenosu.
  Vstupy: *IN_dekoder* - sériový bitový tok určený k dekódování.
  RESET - resetovací signál uvádějící dekodér do počátečního stavu.
  CLK_SRC - synchronizace pro zajištění funkčnosti bloku řízení (Control_block).
  Výstup: *OUT_dekoder* - dekódovaný a opravený sériový bitový tok určený ke zpracování.
- **And_beh** - dvojvstupý logický člen AND.
  Vstupy: *IN_1, IN_2* - operandy určené k logickému násobení.
  Výstup: *AND_OUT* - výsledek logického součtu.
- **Control_block** - generická komponenta řídící funkci prvků kodéru a dekodéru.
  Vstupy: *CLK_IN* - synchronizace ze vstupu pro výpočet řídicích signálů.
  RESET - resetovací signál uvádějící blok řízení do počátečního stavu.
  Výstupy: *CLK_KOD_DEKOD_IN* - řídicí signál pro prvky kodéru a dekodéru pracující s informačními bity.
  CLK_KODER_OUT - řídicí signál pro prvek přepínající výstup kodéru.
  *CLK_DEKODER_SY*N - řídicí signál pro prvky dekodéru pracující se syndromy a korekcí.
  CLK_DATA_IN - řídicí signál pro pomocnou komponentu simulující vstup kodéru.
- **Nxor_beh** - generická komponenta představující n-vstupý logický člen XOR.
  Vstupy: *IN_vect* - vektor operandů určených k logické operaci XOR.
  Výstup: *XOR_OUT* - výsledek logické operace XOR n operandů.
- **Paralel_to_serial** - generická komponenta přepínající informační a zabezpečující bity na výstup dekodéru a povolující opravu informačních bitů v zpožďovacím posuvném registru dekodéru.
  Vstupy: *Paralel_IN* - dvoubitový vektor, který bude přepínán na výstup.
  RESET - resetovací signál uvádějící komponentu do počátečního stavu.

CLK - řízení přepínání.

Výstup: *Serial_OUT* - výstupní sériový bitový tok.

- **Reg_struct** - generická komponenta představující n-bitový posuvný registr se sériovým vstupem a výstupem a asynchronním nulováním.

  Vstupy: *Reg_IN* - sériový bitový tok vstupující do registru.

  RESET - resetovací signál asynchronně nulující registr.

  CLK - signál řídící posouvání bitů v registru.

  Výstup: *Reg_OUT* - sériový bitový tok vysouvaný z registru.

- **Xor_beh** - dvojvstupý logický člen *XOR*.

  Vstupy: *IN_1, IN_2* - operandy určené k logické operaci *XOR*.

  Výstup: *XOR_OUT* - výsledek logického operace *XOR*.

Pomocné komponenty použité pro testovací účely:

- **My_input** - generická komponenta modelující vstup kodéru a vstup chyb do přenosového kanálu.

  Vstupy: *IN_buffer* - bitový vektor jež bude vysouván jako sériový bitový tok.

  RESET - resetovací signál asynchronně nulující vnitřní registr.

  CLK - signál řídící vysouvání bitů z vnitřního registru na výstup.

  Výstup: *Input* - vysouvaný sériová bitový tok (představuje vstup pro testované komponenty).

- **My_lfsr** - komponenta generující pseudonáhodný vstup kodéru.

  Vstupy: *RESET,CLK* - resetovací a synchronizační signál - stejné jako výše.

  Výstup: *DATA_OUT* - sériový pseudonáhodný výstup.

- Testovací testbench komponenty pro každou z výše popsaných komponent.

6.2.5 Simulace strukturního popisu kodeku

Simulace popsaných komponent byla provedena v produktu *Modelsim XE II*. Účelem byla verifikace modelovaných komponent (na odpovídající vstupy dostaneme očekávané výstupy - neboli komponenta je přesným obrazem formální definice, kterou jsme si pro ni zavedly), ale také validace (především u cílových komponent kodéru a dekodéru - ověřování zda celá funkčnost modelu odpovídá původním neformálním popisům funkce. Například zjišťování jakým způsobem dochází ke korekci chyb a zda jsou při korekcích splněny očekávané předpoklady korekční schopnosti). Více o pojmech verifikace a validace simulačních modelů pojednává literatura [13].

Každá komponenta má v elektronických přílohách testovací komponentu, která vytváří testované komponentě různé kombinace vstupů (pokud je to možné tak všechny možné kombinace) a na základě zobrazených výstupů je pak možné posoudit očekávanost chování (verifikovat model). Takto bylo provedeno například testování komponenty Control_block - posouzením signálových diagramů podobných tomu na obrázku (Obr. 6.3) (testování bylo samozřejmě prováděno pro různé

hodnoty generického parametru). Popsaný způsob testování mohl být proveden u všech dílčích komponent provádějících jednoduché logické nebo jiné funkce.

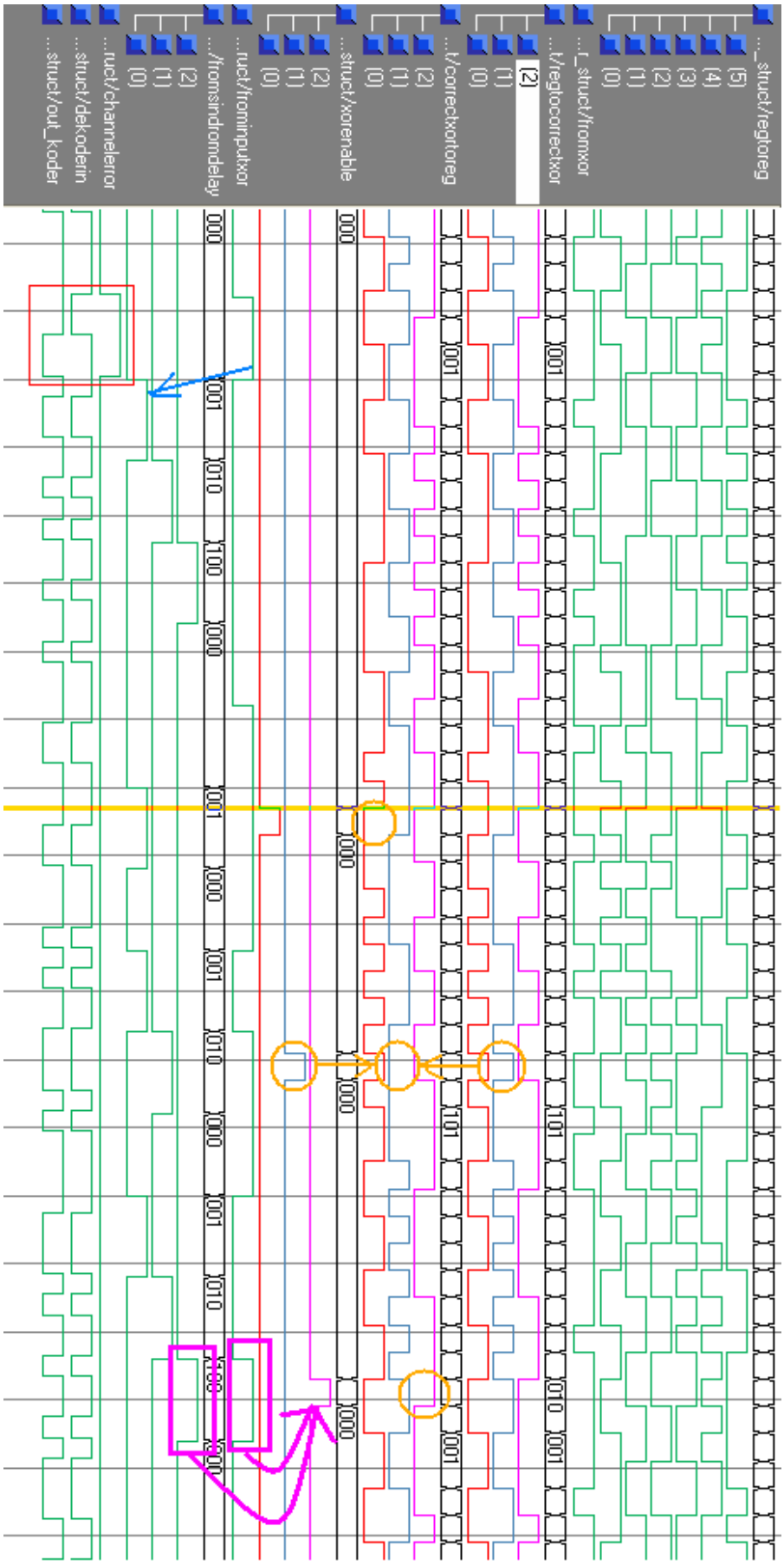
Pro posouzení funkcionality kodéru a dekodéru byla provedena jednak verifikace - tedy kontrola korespondence s abstraktním, matematickým, modelem (tedy zda výsledné signálové toky odpovídají transformacím vstupů podle rovnic (4.10) až (4.14)), ale také validace - posouzení zda signálové toky prokazují, že dochází ke korekcím poškozených bitů. Při tomto způsobu testování byly posuzovány grafy signálových toků podobných tomu na obrázku (Obr. 6.5).

6.2.5.1 Příklad opravy násobné chyby

Obrázek (Obr. 6.5) představuje diagram toku části vnitřních signálů instance pro zabezpečování proti shluku čtyř chyb generické komponenty dekodéru podílejících se na detekci a korekci chyb. Instance byla vytvořena nastavením generického parametru *block_length* na 3, což znamená že při přenosu mohou být poškozeny tři informační bity a jeden zabezpečující, aby byl dekodér schopen zprávu správně interpretovat. Červeně orámovaná část v (Obr. 6.5) představuje propagaci chyby do přenosu (signál *ChannelError* představuje chybový vektor při přenosu, na něm byla vygenerována čtyřnásobná shluková chyba, která znehodnotila bity přenášené z kodéru signálem *OUT_koder* do vstupu dekodéru modelovaného signálem *DekoderIN*).

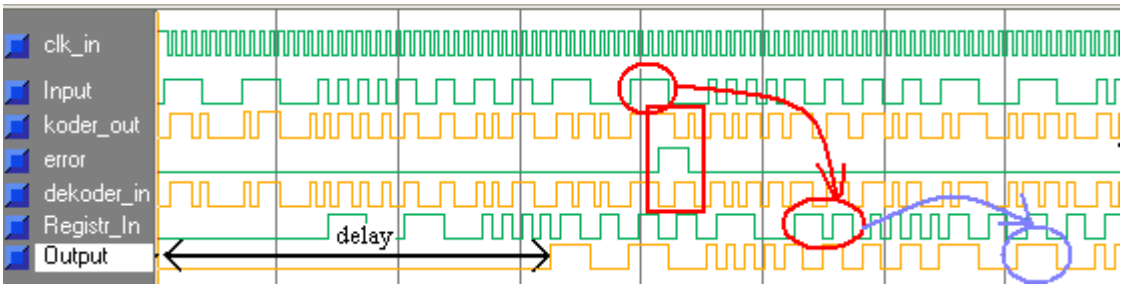
Chyby v přenosu se postupně projeví v generátoru syndromu, který porovnává lokálně vypočtené zabezpečující bity s přenesenými. Zjištěné chyby se projeví jako impulzy signálu *FromInputXor*. Z generátoru syndromu postupují zjištěné příznaky chyb přenosu do zpožďovacích syndromových buněk (to je v jednom z případů v diagramu (Obr. 6.5) znázorněno modrou šipkou). Zpožďovacími buňkami se pak bit příznaku chyby posouvá dokud generátor syndromu nevygeneruje další příznak. To způsobí iniciaci signálu *XorEnable* určeného k opravě chybného bitu a také vynulování zpožďovacích syndromových buněk (jeden případ realizované iniciace je v grafu zaznačen růžovou šipkou). Anebo se bit příznaku chyby ze zpožďovacích syndromových buněk vysune aniž by inicioval opravu (k tomu dojde v případě poškození zabezpečovacího bitu - tedy v případě, kdy není třeba opravovat žádný informační bit. K popsanému případu došlo v grafu v místě, kde je modrou šipkou zaznamenáno zpoždění vstupního syndromu - ten postupně projde všemi třemi buňkami bez iniciace signálu *XorEnable*).

Signál *XorEnable* pak řídí vlastní opravu chybných bitů pomocí logické funkce *XOR*. Vektory signálů *RegToCorrectXor* a *CorrectXorToReg* představují bitový tok vstupující do a vystupující z opravných logických členů *XOR* v posuvných registrech dekodéru. Na nich je viditelná oprava iniciovaná vektorem signálů *XorEnable*. V grafu je vždy stejnou barvou zakreslen signál vektoru *RegToCorrectXor* vstupující do opravného *XOR*u, signál vektoru *CorrectXorToReg* vystupující z opravného XORu a signál vektoru *XorEnable*, který opravu řídí. Opravené bity jsou označené žlutými kroužky, v jednom případě je žlutými kroužky označen i bit opravovaný a bit řídící opravu a šipkami je vyznačen vznik opraveného bitu.



Obr. 6.5: Diagram signálových toků dekodéru

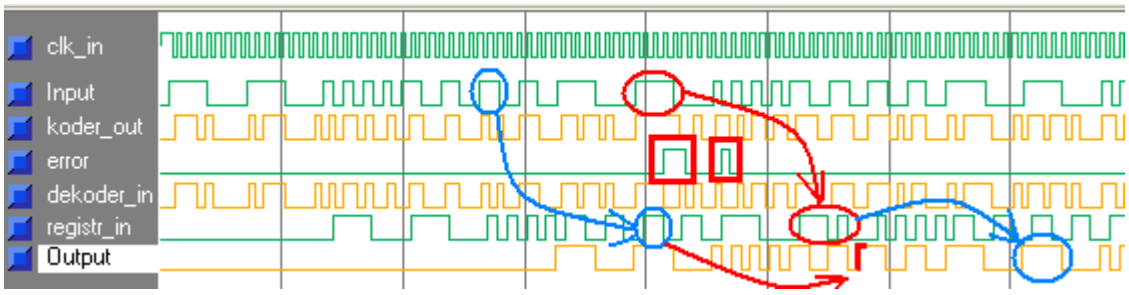
Pro názornost je na obrázku (Obr. 6.6) stejný diagram signálových průběhů, zobrazený s větším rozlišením a se signály, na nichž je nejnázornější vlastní průběh opravy. Je na něm znázorněn vstup a výstup kodéru (*Input*, *koder_out*), chybový vektor (*error*) znehodnocující vstup dekodéru (*dekoder_in*), poškozený bitový informační tok vstupující do posuvného registru (*Registr_in*) a opravený zpožděný bitový informační tok vystupující z posuvného registru dekodéru (*Output*). V diagramu je opět červeným rámečkem znázorněna propagace chyby do přenosu, červenými ovály a šipkou výskyt chyby v informačních bitech (shlukovou čtyřchybou byly poškozeny tři informační a jeden zabezpečující bit), modrou šipkou a oválem oprava násobné chyby a bílou šipkou je naznačeno informační zpoždění způsobené soustavou kodér-dekodér. Zpoždění je rovno době potřebné pro projití všemi buňkami posuvného registru dekodéru - ten má *A* paměťových buněk - viz vzorec (4.5).



Obr. 6.6: Stručný diagram signálových toků demonstrující korekci bitů

6.2.5.2 Příklad nedodržení ochranné vzdálenosti

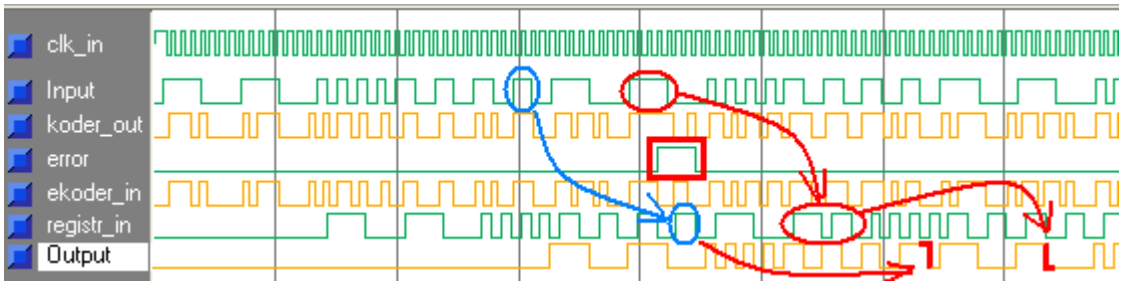
Na obrázku (Obr. 6.7) je diagram signálových toků do a z komponent kodér a dekodér pro stejnou zabezpečovací schopnost jako výše. Zde ovšem nebyl dodržen jeden ze základních požadavků pro bezchybnou funkčnost protichybového kódového systému - ochranná vzdálenost. V signálu *error* bylo vygenerováno více úrovní představujících chyby tak, aby nemohly dohromady tvořit jednu chybu shlukovou (tedy vzdálenost mezi nimi překračuje zvolenou korekční schopnost kódu). Dekodér poté sice nejen správně opravil vzniklou chybu v přenosu (znázorněnou červenou šipkou od signálu *Input* k signálu *dekoder_in*), ale také nesprávnou opravou poškodil jiný, správně přenesený, bit (modrou šipkou od signálu *Input* k *dekoder_in* je naznačen bezchybný přenos, červenou šipkou k *Output* pak nesprávná oprava).



Obr. 6.7: Stručný diagram signálových toků demonstrující nedodržení ochranné vzdálenosti

6.2.5.3 Příklad překročení maximální délky násobné chyby

Na obrázku (Obr. 6.8) je další ze signálových toků, nyní pro demonstraci překročení zabezpečovací schopnosti kódu. Se zachováním označování signálů a chybných a správných přenosů jako v předchozích diagramech je zde zřetelný vliv příliš dlouhé chyby v přenosu. Opět dochází k pokusu o opravu na nesprávném místě a navíc k neúplné opravě přenosem poškozených bitů.



Obr. 6.8: Stručný diagram signálových toků demonstrující překročení korekční schopnosti

Poslední dvě možnosti ukazující chování kodeku při nedodržení potřebných parametrů přenosu se prolínají (násobnou chybu lze požadovat za dlouhou shlukovou chybu a naopak). Společné pro ně je, že dochází k chybným opravám správně přenesených bitů (v případě, že bylo poškozeno více zabezpečujících bitů a ty pak nesprávně iniciují opravy) anebo k nedostatečným opravám poškozených bitů (v případě, kdy je poškozeno příliš mnoho informačních bitů a dekodér není ze zabezpečení schopen rozlišit, které bity má opravovat), vždy je však průnik chyby konečný.

6.3 Behaviorální popis

Pro behaviorální styl popisu je typické používání speciálních bloků příkazů - takzvaných procesů. Pro správné rozdělení příkazů do procesů je třeba odlišit, v závislosti na kterých vstupech a signálech vnitřního stavu systému se mění výstupy a vnitřní stav popisovaného systému.

V behaviorálním popisu již není nezbytně nutné modelovat zvlášť všechny prvky kodeku (logické členy *XOR* a *AND*, posuvné registry a zpožďovací buňky a převodníky signálů o různé synchronizaci). Je tedy třeba analyzovat způsob modelování těchto prvků. A stejně jako u strukturálního popisu potřebujeme navrhnout řešení popisu opakujících se struktur a požadavku různých synchronizací pro vstupy a výstupy kodeku.

6.3.1 Popis prvků kodéru a dekodéru

Logické členy pro výpočet operací *XOR* a *AND* je nejsnadnější namodelovat pomocí stejnojmenných vestavěných logických operátorů jazyka *VHDL*. Další potřebnou komponentou je posuvný registr. Typický behaviorální popis této komponenty se sestává z vektoru signálů, měněných v těle procesu, který způsobí změnu stavu až při změně potřebného (obvykle synchronizačního) signálu. Vlastní činnost posuvu v registru je popsatelná pomocí operátoru konkatenace (&) anebo logického posunu či

rotace (*SLL*, *SRL*, *SLA*, *SRA*, *ROL* a *ROR*) - více o použití těchto operátorů pojednává literatura [10]. Samotný registr je možné popsat následujícím kódem:

```
signal BUFFER: std_logic_vector(X downto Y); -- registr potrebne sirky

process(RST, CLK)                        -- zmena jen pri zmene CLK a RST
begin
   if (RST='1') then                     -- resetovani
      BUFFER <= (others => '0');
   elsif (CLK'event) and (CLK='1') then -- při nastupne hrane hodin
      -- pro posuv buď
      BUFFER <= INPUT & BUFFER(7 downto 1);
      -- anebo
      -- BUFFER <= BUFFER SRL 1;
     -- BUFFER(BUFFER'length - 1) <= INPUT;
   end if;
end process;
```

Poslední z používaných komponent je převodník signálů a vstupů/výstupů s různou synchronizací. V kapitole 6.2.3 bylo odvozeno, že vstupní převodníky nejsou potřebné - postačuje, aby odpovídající prvky vzorkovaly vstup se správnou frekvencí. V behaviorálním popisu není třeba na výstup připojit nastálo výstup jediné komponenty, naopak můžeme v čase přepínat výstupy více komponent. Přesně pro tento účel jsme ve strukturálním popisu museli vytvořit komponentu *PARALEL_TO_SERIAL*. V behaviorálním popisu bude ovšem potřeba pouze pomocí správně řízeného procesu posílat na výstup kodéru střídavě informační a zabezpečující bity z různých částí kodéru. Tedy stačí výstup vytvářet na základě hodnot řídicích čítačů (bude popsáno v kapitole 6.3.3).

6.3.2 Popis opakujících se struktur

Behaviorální styl popisu je charakteristický tím, že je v něm možné používat konstrukce typické pro jazyky pro popis software - tedy rozhodovací bloky, cykly a podobně. A právě opakující se struktury znázorněné na schématech (Obr. 4.3) a (Obr. 4.4) pomocí tří teček, by v běžném programovacím jazyce vedly k popisu využívajícímu počítaný cyklus. Posuvné registry počítající zabezpečení by popisoval cyklus odpovídající tomuto pseudokódu:

```
-- deklarace pomocnych promennych, pro moznost vyuziti cyklu
pomocna_promenna;
pomocny_index;

-- pred cyklem popiseme nepravidelnosti
pomocna_promenna := BUFFER(X) XOR BUFFER(Y)

-- dalsi hodnoty jsou v registru rozmisteny pravidelne
FOR i IN 2 TO k LOOP
    pomocny_index := pomocny_index + f(i);
    pomocna_promenna := pomocna_promenna XOR BUFFER(pomocny_index);
END LOOP;

-- vysledne zabezpeceni posleme na vystup
OUT <= pomocna_promenna;
```

Závislost *f(i)*, jež určuje rozmístění bitů v registru určených pro výpočet zabezpečení, jsme již odvodili v kapitole 6.2.1. Naprosto obdobně popíšeme v komponentě dekodér opakující se struktury v prvku převodník $S \rightarrow E$, a pro korekci bitů v registru na určitých pozicích.

6.3.3 Realizace různých synchronizací

V závěru kapitoly 6.2.2 bylo u odvození komponenty generující řídicí (synchronizační) signály pro jednotlivé prvky kodeku uvedeno, že nejvhodnější bude popsat zmíněnou komponentu behaviorálně a to za pomocí čítačů. Příslušně nastavované čítače můžeme využít i přímo v popisu funkce kodéru popřípadě dekodéru, kde budeme na základě hodnot těchto čítačů propouštět vstup do určitých prvků popřípadě propouštět výstup z určitých prvků na výstup celé komponenty.

Nejdříve je třeba odvodit, jakým způsobem budeme vstupní hodinový signál dělit na signály o potřebných frekvencích. Z odvození kodeku (a z podrobnějšího odvozování potřeb vnitřní synchronizace v kapitolách 4.4 a 6.2.2) je zřejmé, že jsou potřebné synchronizace pro vstup kodéru, výstup kodéru a výstup zabezpečujících bitů z kodéru (dekodér pak použije stejné synchronizace). Zabezpečenou zprávu je možno rozdělit do bloků o *k* informačních a jednom zabezpečeném bitu. Tedy pokaždé, když do kodéru vstoupí postupně *k* bitů, vystoupí z něj *k* + 1 bitů (z toho jeden zabezpečující). Synchronizační signály vstupu, výstupu a zabezpečujících bitů se musí tedy po zmíněných počtech period ocitnout ve výchozím stavu (všechny budou mít například opět společnou nástupnou hranou).

Ze vstupního periodického synchronizačního signálu tedy potřebujeme odvodit signály o frekvencích:

pro vstup kodéru: $$f_{IN\_koder} = \frac{k}{t} \tag{6.1}$$

pro výstup kodéru: $$f_{OUT\_koder} = \frac{k+1}{t} \tag{6.2}$$

pro prvky pracující se zabezpečením: $$f_{SYN\_dekoder} = \frac{1}{t} \tag{6.3}$$

Kde *t* je doba potřebná pro zpracování jednoho bloku bitů nezabezpečeného vstupu do kodéru. Doba *t* tedy odpovídá *k* periodám synchronizace vstupu, *k*+1 periodám synchronizace výstupu a jedné periodě synchronizace zabezpečení. Lze snadno odvodit, že nejmenší počet period vstupní synchronizace, již budeme dělit, odpovídá nejmenšímu společnému násobku všech počtů period, které se do této doby vejdou. V našem případě pak můžeme psát:

počet period děleného signálu: $$t = NSN(k; k+1; 1) * T_{divided} \tag{6.4}$$

Také platí:

$$NSN(k; k+1; 1) = NSN(k; k+1) = k * (k+1) \tag{6.5}$$

Ze vzorců (6.1) až (6.5) je nyní již snadné odvodit periody potřebných signálů vyjádřené v počtech period dělené synchronizace:

pro vstup kodéru: $T_{IN\_koder} = \frac{1}{f_{IN\_koder}} = \frac{t}{k} = \frac{k*(k+1)}{k} * T_{divided} = (k+1) * T_{divided}$ (6.6)

pro výstup kodéru: $T_{OUT\_koder} = \frac{1}{f_{OUT\_koder}} = \frac{t}{k+1} = \frac{k*(k+1)}{k+1} * T_{divided} = k * T_{divided}$ (6.7)

pro prvky pracující se zabezpečením:

$$T_{SYN\_dekoder} = \frac{1}{f_{SYN\_dekoder}} = \frac{t}{1} = \frac{k*(k+1)}{1} * T_{divided} = k*(k+1)*T_{divided} \quad (6.8)$$

Příslušné synchronizační (respektive řídicí) signály tedy budeme měnit vždy po vypočteném počtu změn signálu dělené synchronizace. Vlastnímu dělení synchronizací může odpovídat diagram (Obr. 6.3) pro konkrétní hodnotu parametru k = 4. V behaviorálním popisu komponent stačí použít inkrementované čítače, na základě jejichž hodnoty vyvoláme příslušné akce. Pomocí části pseudokódu by odvozené řízení činnosti kodéru či dekodéru mohlo vypadat následovně:

```
-- deklarace promennych predstavujicich jednotlive citace
counter_in;
counter_out;
counter_syn;

process (CLK, RESET)
begin
  if RESET = '1' then             -- pri aktivite signalu reset
     vynuluj vsechny citace;
  elsif CLK='1' and CLK'event then -- pri nabezne hrane vstupnich hodin
      if (counter_in = k+1)       -- pokud citac dosahl hranicni hodnotu
         vynuluj counter_in;      --   nuluj
         nacitej vstup;           --   a vykonej prislusne akce
      else
         inkrementuj counter_in;  -- jinak citac jen inkrementuj
      endif;

      if (counter_out = k)        -- obdobne pro všechny citace
      ...
      ...
end process;
```

Hodnoty čítačů řídí funkci celého kodéru i dekodéru. Z příkladu je zřejmé vyčítání vstupních bitů ze sériového proudu o předpokládané frekvenci. Stejně ovšem budou pomocí všech čítačů řízené i ostatní činnosti, jako například posílání informačních a zabezpečujících bitů na výstup (řízených čítači *counter_out* a *counter_syn*) anebo oprava chybných bitů v dekodéru (řízená čítačem *counter_syn*). Tímto je plně nahrazena funkce prvku *PARALEL_TO_SERIAL*, který tyto zmíněné činnosti řídil v kodéru a dekodéru popsaném strukturálně.

Vlastní čítač je možné jednoduše modelovat pomocí celočíselné proměnné (typu integer), pak ovšem bude mít po syntéze čítač vždy pevnou šířku 32 bitů. Vhodnější je popsat čítač pomocí bitového vektoru o šířce odpovídající nejbližší vyšší celočíselné hodnotě dvojkového logaritmu

požadované maximální hodnoty čítače. V tomto případě dosáhneme stejného výsledného efektu s menší spotřebovanou plochou výsledné obvodové realizace:

```
signal counter_in : std_logic_vector(log2(k+1) downto 0);
```

Funkce *log2* není vestavěnou funkcí jazyka *VHDL* - bylo ji třeba naprogramovat (viz zdrojový kód *math.vhd* ve složce *src* na přiloženém CD).

6.3.4 Komponenty využité pro behaviorální popis kodeku

Behaviorálním popisem komponent kodéru a dekodéru, bylo možné přehledně popsat činnost v rámci jednoho souboru pro každou komponentu. Byly tedy vytvořeny tyto komponenty:

- **IwNKoder_beh**
- **IwNDekoder_beh**

Popis činnosti, vstupy a výstupy těchto komponent jsou identické s komponentami uvedenými v části o strukturálním popisu (s tímto záměrem byly také dvě alternativní těla jedné entity navrhovány).

6.3.5 Simulace behaviorálního popisu kodeku

Testování behaviorálně popsaných komponent jejich simulací bylo analogické výše popsanému postupu u strukturálně modelovaných komponent (viz kapitola 6.2.5). Při validaci popsaných modelů byl mimo jiné použit také diagram některých signálových toků v systému zapojení kodér - chybový kanál - dekodér, který je na obrázku (Obr. 6.9).

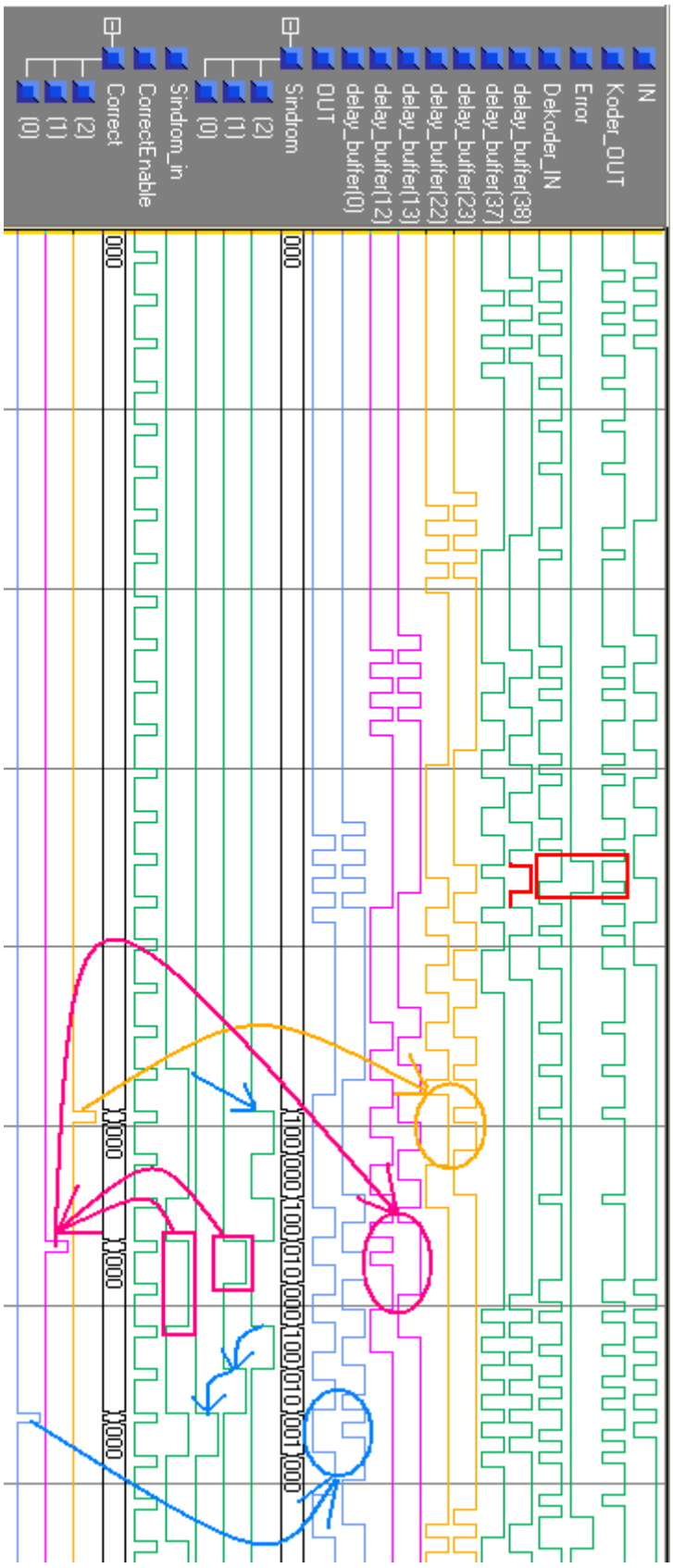
6.3.5.1 Příklad opravy násobné chyby

Obrázek (Obr. 6.9) představuje diagram toku části vnitřních signálů instance pro zabezpečování proti shluku čtyř chyb generické komponenty dekodéru podílejících se na detekci a korekci chyb. Mechanizmus vzájemné iniciace jednotlivých signálů a následné korekci chyb je analogický jako je tomu na obrázku (Obr. 6.5) s popisem v kapitole 6.2.5.1. V této kapitole bude tedy jen velmi stručný popis zaměřující se převážně na odlišnosti oproti situaci se strukturálně popsaným systémem.

V diagramu (Obr. 6.9) je červeně vyznačen průnik chyby do přenosu, ten v dekodéru v jistých okamžicích vyvolá vznik vstupních syndromů (*Syndrom_in*), které se zachytávají v syndromových buňkách (*Syndrom*). Syndromové buňky spolu se vstupním syndromem vyvolávají opravu příslušných paměťových buněk (signály *Correct*). Oprava je ovšem povolena jen v jistých časových okamžicích - aby při zjištění chyby nebyl opraven více než jeden informační bit (jejichž frekvence posunu ve zpožďovacích buňkách je *k*-krát větší než frekvence posunu syndromů viz vzorce (6.6) a (6.8)). Povolení opravy se děje za pomocí signálu *CorrectEnable*.

V diagramu byly zobrazeny jen ty zpožďovací paměťové buňky, v nichž proběhne oprava iniciovaná příslušným signálem *Correct* a buňky jim bezprostředně předcházející, aby byla oprava

zřetelná. Stejnou barvou jsou zobrazeny signály *Correct*, jež invertováním chybu opravují a paměťové buňky, kterých se oprava týká.



Obr. 6.9: Diagram signálových toků dekodéru

I přes průnik chyby (*Error*) jsou signály vstupující do zabezpečujícího systému (*IN*) a z něj vystupující (*OUT*) totožné (pouze signál *OUT* je zpožděný). Došlo tedy k úspěšnému zabezpečení přenášené informace proti shlukové chybě.

Příklady demonstrující korekční schopnosti modelovaného protichybového kódového systému, které by zobrazovaly pouze vstup a výstup systému a propagaci chyby, jsou naprosto identické s příklady uvedenými v kapitole 6.2.5. Behaviorálně a strukturálně popsané komponenty se liší pouze vnitřními signály a aktuálním vnitřním stavem signálů. Na totožné vstupy však reagují totožnými výstupy, mají tedy ekvivalentní chování (blíže o problematice modelů a jejich vztahů pojednává literatura [13]). Další příklady již tedy nebudou znovu uváděny.

7 Syntéza a praktická realizace kodeku

Zdrojové kódy popisující kodek byly syntetizovány a binární konfigurační soubory pro naprogramování platformy FIT-kit byly získány pomocí vývojového prostředí Xilinx ISE 9.1 a také pomocí překladového systému, vyvinutého jako podpůrná součást projektu FIT-kit, využívajícího *Makefile* soubory.

Pro porovnání výsledných komponent popsaných strukturálně a behaviorálně byly využity charakteristické parametry a výsledné obvodové zapojení komponent vygenerované v průběhu syntézy.

7.1 Charakterizující parametry komponent

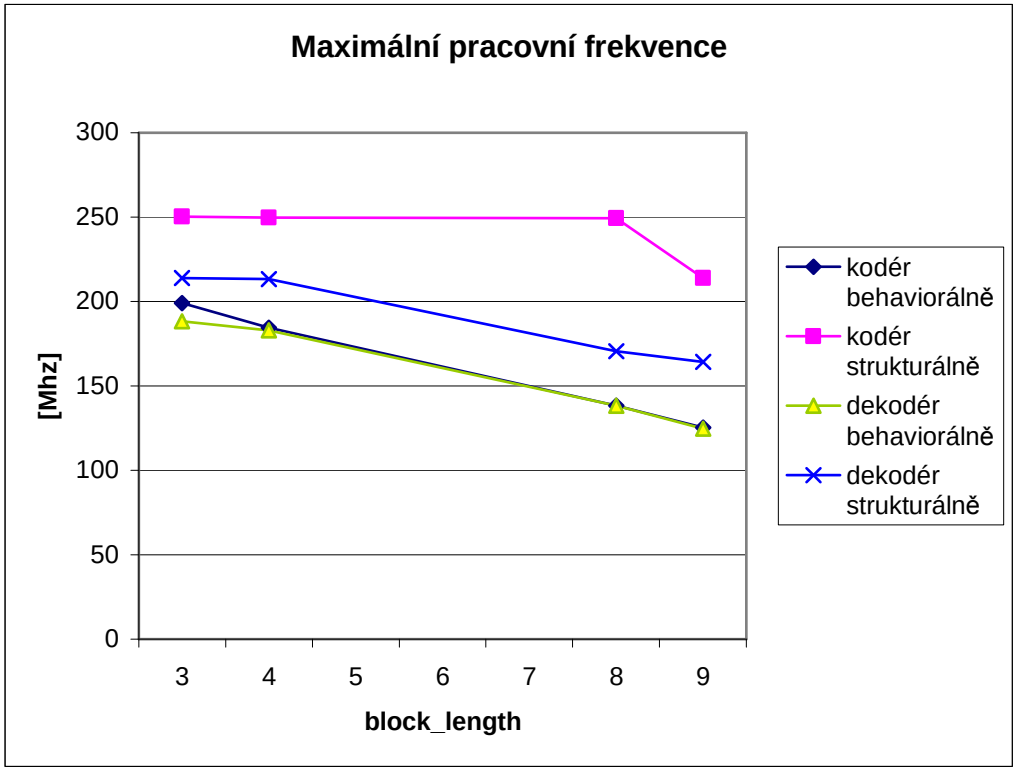
Mezi nejvýznamnější kritéria pro hodnocení a porovnávání číslicových systému patří jejich rychlost a plocha. Při hodnocení behaviorálně a strukturálně popsaného kodeku byly rovněž použity tyto kritéria a to konkrétně maximální dosažitelné pracovní frekvence a rozsah obvodu přepočítaný do takzvaných *ekvivalentních hradel*. Závislost těchto parametrů na vzrůstající hodnotě generického parametru *block_lenght* jsou znázorněny v tabulce (Tab. 7.1) a (Tab. 7.1) a grafu (Obr. 7.2) a (Obr. 7.2).

		FREKVENCE [MHz]			
		kodér behaviorálně	kodér strukturálně	dekodér behaviorálně	dekodér strukturálně
block_length	3	198,958	250,25	188,395	213,995
	4	184,574	249,626	182,815	213,311
	8	138,358	249,377	138,281	170,532
	9	125,293	213,904	124,701	164,285

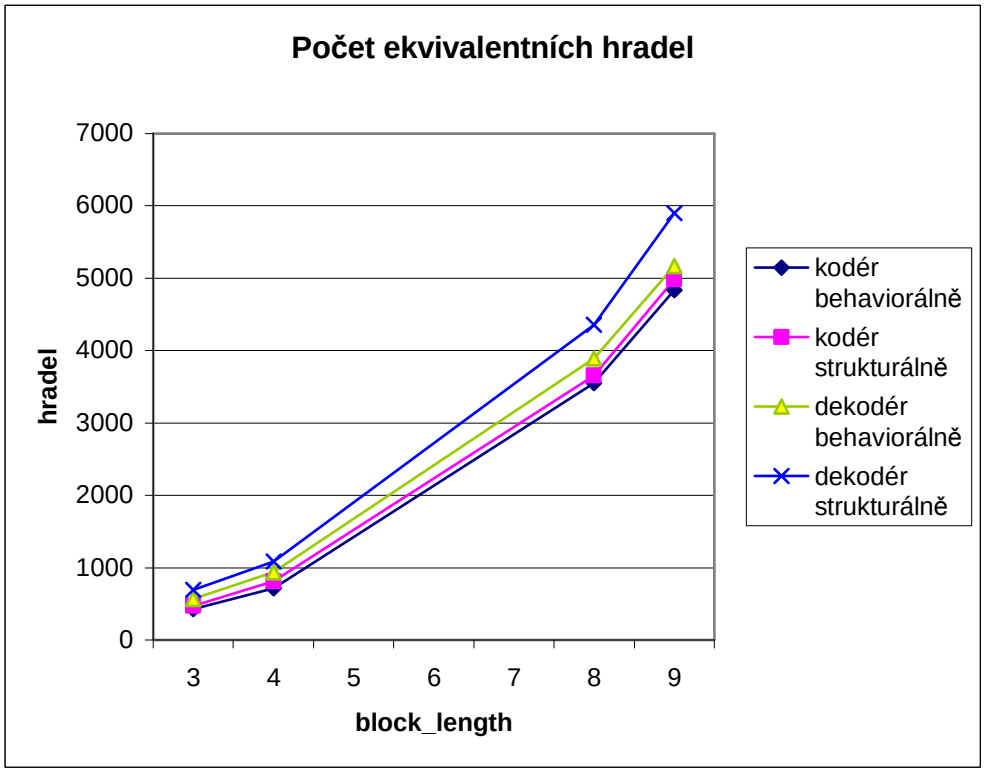
Tab. 7.1: Hodnoty maximální pracovní frekvence kodeku

		EKVIVALENTNÍCH HRADEL			
		kodér behaviorálně	kodér strukturálně	dekodér behaviorálně	dekodér strukturálně
block_length	3	427	475	565	691
	4	713	814	940	1083
	8	3549	3652	3891	4356
	9	4838	4985	5168	5899

Tab. 7.2: Hodnoty objemu použité logiky v ekvivalentních hradlech



Obr. 7.1: Graf závislosti maximální pracovní frekvence kodeku na hodnotě generického parametru



Obr. 7.2: Graf závislosti objemu použité logiky kodeku na hodnotě generického parametru

Z hodnot (Tab. 7.2) a především z grafu (Obr. 7.2) je zřejmé, že plocha na čipu spotřebovaná logikou tvořící kodér či dekodér Iwadariho kódu se zvyšující se hodnotou generického parametru (a tedy korekční schopností) prudce roste (přesněji narůstá kubicky, jak bylo odvozeno ve čtvrté kapitole). Objem spotřebované logiky není příliš závislý na tom, zda je komponentou kodér, či dekodér a zda byla syntetizována z behaviorálního či strukturálního popisu. Ovšem o něco méně logiky spotřebovává kodér než dekodér a nepatrně lepší vlastnost v tomto ohledu mají behaviorálně popsané komponenty.

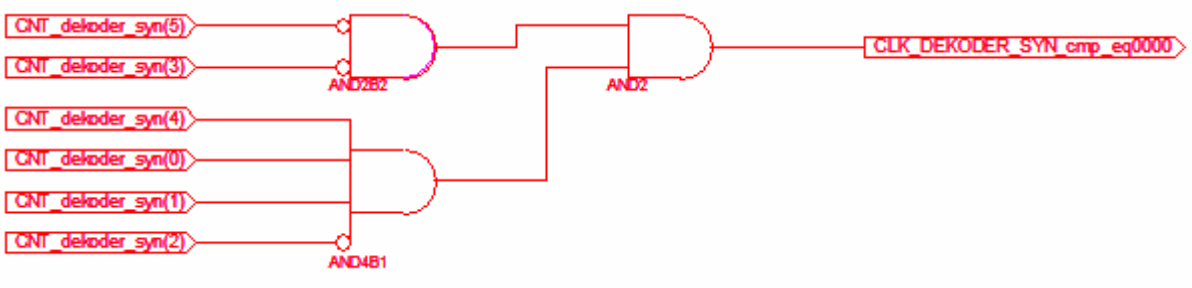
Tabulka (Tab. 7.1) a graf (Obr. 7.1) zaznamenávají závislost maximální dosažitelné pracovní frekvence komponent na vzrůstající hodnotě generického parametru. Je zřejmé, že hodnoty pracovní frekvence klesají se zvyšující se korekční schopností a že strukturálně popsané komponenty v tomto ohledu poskytují ve zkoumaném vzorku výrazně lepší vlastnosti. Z tabulky ani grafu ovšem nelze pro zvolený vzorek hodnot odvodit konkrétní podobu funkční závislosti (ani její přibližné aproximace) pracovní frekvence na korekční schopnosti kodeku. K tomuto účelu by bylo nutné mnohem podrobnější (nejlépe automatizované) testování.

7.2 Výsledná obvodová podoba komponent

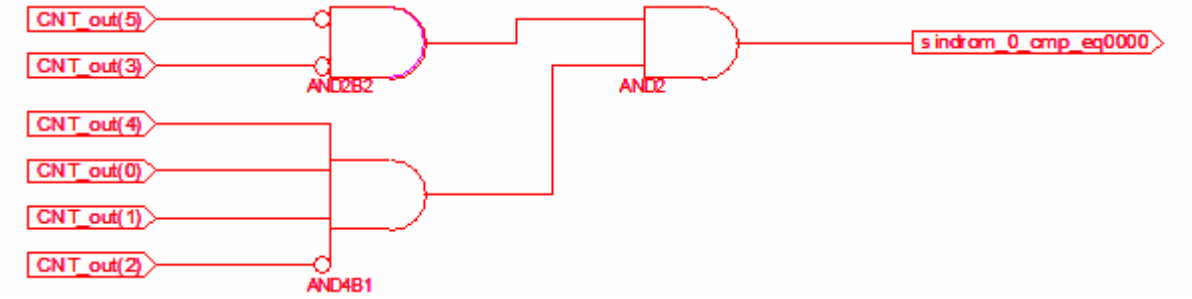
I přestože se rozsah a způsob popisů cílových komponent v behaviorálním a strukturálním stylu velice liší, vycházely oba tyto popisy ze stejných předloh (viz (Obr. 4.3) a (Obr. 4.4)) a jejich cílem bylo popsat ekvivalentní činnost výsledného obvodu. Ekvivalence činnosti komponent byla ověřena a potvrzena v předchozí kapitole - Implementace a simulace kodeku. V předchozí podkapitole bylo ukázáno, že i charakteristické parametry komponent popsané těmito dvěma styly jsou velmi blízké. Definitivní posouzení odlišností nám poskytne porovnání *RTL (register transfer level)* schémat vygenerovaných při syntéze (Zobrazují modelovanou komponentu jako propojení registrů, čítačů, multiplexorů a další logiky této úrovně. Při syntéze modelované komponenty nástroji firmy Xilinx jsou tato schémata obvykle uložena jako soubory s příponou .ngr).

Vygenerovaná schémata se zdají být po zběžném prohlédnutí značně odlišná (viz Příloha 1 až Příloha 4 a na CD ve formě příslušných .ngr souborů). To je ovšem způsobeno odlišným členěním prvků do takzvaných hierarchických bloků. Po podrobném prozkoumání schémat je zřejmé, že jsou si výsledné obvody velmi podobné, v jistých částech jsou často naprosto identické. Pro celé schéma odkazujeme na zmíněné přílohy, zde uvedeme jen některé reprezentativní části.

Například obvody synchronizace (či řízení) jsou pro oba různé popisy naprosto identické - viz (Obr. 7.3) a (Obr. 7.4)

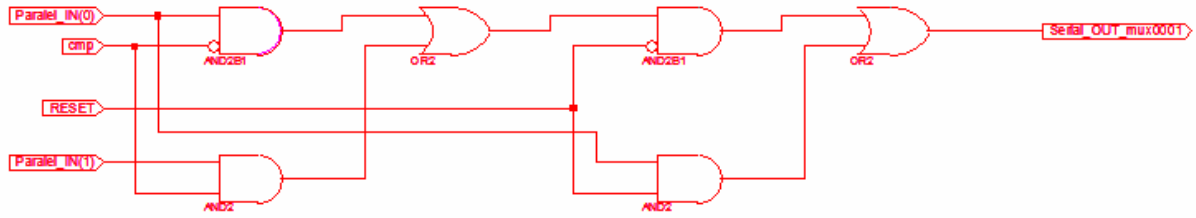


Obr. 7.3: Vygenerované RTL schéma obvodu synchronizace pro zabezpečení (strukturálním popis)

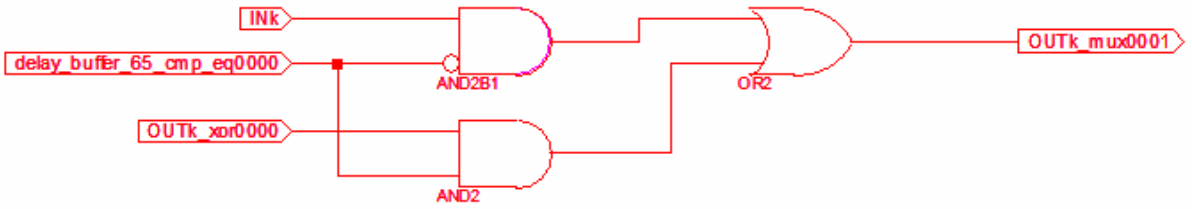


Obr. 7.4: Vygenerované RTL schéma obvodu synchronizace pro zabezpečení (behaviorální popis)

Dalším problémem jenž byl tentokrát značně odlišně řešen v obou popisech bylo přepínání informačních a zabezpečujících bitů na výstup dekodéru (viz kapitoly 6.2.3 a 6.3.1). Obvodová schémata obvodů realizujících vlastní přepínání jsou zobrazena v (Obr. 7.5) a (Obr. 7.6).



Obr. 7.5: Vygenerované RTL schéma obvodu přepínajícího výstup kodéru (strukturální popis)



Obr. 7.6: Vygenerované RTL schéma obvodu přepínajícího výstup kodéru (behaviorální popis)

Různé popisy zde opět vedly k obdobnému schématu. Obě schémata představují přepínání mezi vstupem a výstupem ze zabezpečujícího N-vstupého *XOR*u na základě hodnoty čítače. Liší se pouze řešením *RESET*u, kdy u strukturálního popisu bylo nutné definovat výstup přepínacího obvodu i při aktivním signálu *RESET*, z důvodů použití tohoto obvodu v dekodéru pro povolování oprav (viz schéma (Obr. 6.4)).

Pravděpodobně jediným místem, kde se výsledná schémata popsaná v různých stylech liší jsou obvody realizující vlastní korekci bitů. Po automatizované optimalizaci při syntéze behaviorálně

popsaného dekodéru je z *RTL* schématu jen velmi těžko rozpoznatelné, které obvody se přesně na této činnosti podílejí. Oproti dobře čitelnému a logicky strukturovanému *RTL* schématu strukturálně popsaného dekodéru však právě tato část způsobuje dosažení menšího skoku v dosažené maximální pracovní frekvenci oproti dekodéru - viz grafy (Obr. 7.1) a (Obr. 7.2).

7.3 Testovací obvod

Závěrečnou fází této práce je vlastní fyzická realizace kodeku v nějakém vhodném testovacím zapojení. Pro správnou činnost kodeku je nezbytné správné zasynchronizování komponenty kodéru a dekodéru. Tento problém obvykle řeší komunikační protokol, tedy vyšší vrstva abstraktního modelu komunikačního zařízení.

Máme dvě hlavní možnosti, jak se vyhnout nutnosti navrhnout způsob korektní synchronizace zařízení - kodér, dekodér, zdroj zprávy a příjemce zprávy (viz schéma (Obr. 2.3)) se budou fyzicky nacházet na stejném místě (ve společném zapojení na jednom chipu) a tím budou mít k dispozici společné řízení a synchronizaci, anebo za využití některého již existujícího komunikačního protokolu (například SPI) budeme nejprve přenášet data ze zdroje do kodéru a z něj do nějaké mazipaměti a poté z použité mezipaměti do kodéru a z něj do příjemce zprávy. Zde popíšeme pouze první možnost, pro její názornost a možnost neřešit problémy netýkající se přímo protichybového kódování, což je pro demonstraci činnosti velmi vhodné.

7.3.1 Komponenty testovacího obvodu

Pro názornou demonstraci činnost kodeku, je nezbytná komponenta, které bude do sériového přenosu mezi kodérem a dekodérem vnášet shlukové chyby. Generování pseudonáhodných shlukových chyb by mělo podléhat určitým kritériím jako ochranná vzdálenost mezi chybami a maximální délka shlukové chyby. Hodnoty těchto parametrů je možné odvodit z korekční schopnosti kódu - viz vzorce (4.2) až (4.5) - které odpovídá námi používanému parametru *block_length*.

```
component Nerror is GENERIC (block_length: integer := 4);
    port (
        INerr     : in std_logic;           -- vstup
        OUTerr    : out std_logic;          -- zasumneny vystup
        RESETerr  : in std_logic;           -- reset (aktivni v 1)
        CLKerr    : in std_logic            -- synchronizace
    );
end component;
```

Komponenta si na základě hodnoty generického parametru určí ochrannou vzdálenost a délku shlukové chyby, jež nemůže překročit a na základě těchto kritérií překlápí některé bity vstupu a simuluje tak vznik chyby. Pozměněním maximálních hodnot jejích vnitřních čítačů můžeme simulovat i situaci vzniku neopravitelné chyby - viz Příloha 6, Manuál k použití přiložených zdrojových kódů.

Též je nutné simulovat zdroj zprávy. Z důvodu nutnosti určité synchronizace vstupu bude pravděpodobně nejjednodušší a nejméně prostorově náročné řešení využívající generování pseudonáhodných čísel. Pro tento účel vytvoříme jednoduchý generátor pseudonáhodných čísel typu *LFSR* (této problematice se podrobněji věnuje například literatura [16]).

```
component my_lfsr
port(
      CLK        : in std_logic; -- synchronizace vystupu
      RESET      : in std_logic; -- asynchronnni reset
      DATA_OUT   : out STD_LOGIC -- pseudonahodny vystup
   );
end component;
```

Pro posouzení úspěšnosti korekcí je třeba uchovávat - zpožďovat - generovaný vstup, poté jej porovnávat s přenesenou a dekódovanou zprávou a počítat počet rozdílných bitů - tedy neúspěšných korekcí. Pro zpožďování postačí běžný posuvný registr použitý v komponentách kodér i dekodér a popsaný v kapitole 6.2.4. Zpožděný vstup a dekódovanou zprávu je možné porovnávat běžným logickým členem *XOR*, kdy hodnota logická jedna naznačuje chybu v přenosu. Výstup z tohoto členu tedy můžeme využít jako povolovací vstup do běžného čítače, který nám tak poskytne funkci čítání počtu chybně přenesených a dekódovaných bitů.

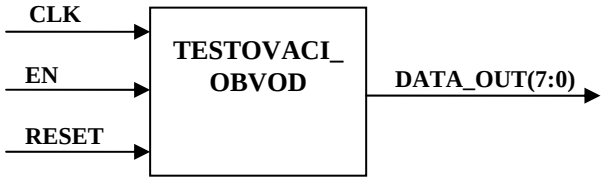
```
component counter
port(
      CLK       : in std_logic;       -- synchronizace citani
      RESET     : in std_logic;       -- asynchronnni reset
      EN        : in std_logic;       -- povoleni citani
      DATA_OUT  : out STD_LOGIC_VECTOR (7 downto 0) -- vystup
   );
end component;
```

Popis těl jednotlivých komponent a jejich vzájemné propojení je v podobě zdrojových kódů součástí elektronické přílohy této práce. Vygenerované schéma testovacího obvodu skládajícího se ze zmíněných komponent je součástí této práce jako Příloha 5.

7.3.2 Rozhraní testovacího obvodu

Pro bezproblémový průběh testování, je nezbytné vhodně navrhnout rozhraní komponenty testovacího obvodu. Jistě bude nutné poskytnout testovacímu obvodu synchronizaci pro blok řízení, dále by měla být tato komponenta vybavena signálem *RESET* pro uvedení do definovaného výchozího stavu a také by měla komponenta mít určitý výstup (výstupy), pomocí kterého by bylo možné sledovat průběh testování. Nejpodstatnějšími ukazateli demonstrace funkce kodéru a dekodéru jsou hodnoty počtu přenesených bitů a počtu chybně přenesených a dekódovaných bitů. Počet přenesených bitů je možné snadno odvodit na základě poskytované synchronizace a povolování činnosti, jež bude diskutována dále. Proto není nutné opatřovat komponentu dalším vnitřním čítačem a výstupní branou. Postačující bude tedy jediný výstup, který bude přímo připojen na výstup výše zmíněného čítače počtu chyb.

Jestliže se rozhodneme zmíněné charakteristické ukazatele zobrazovat například na LCD displeji platformy FIT-kit, je třeba si uvědomit, že změny hodnot těchto ukazatelů při nepřetržité práci testovacího obvodu jsou tak rychlé, že jsou prakticky nepostřehnutelné a tudíž velmi málo průkazné. Řešením je opatřit pro tyto případy komponentu testovacího obvodu povolovacím vstupem, jež lze periodicky s dostatečným zpožděním nastavovat tak, aby byly změny hodnot postřehnutelné. Odvozené rozhraní testovací komponenty znázorňuje schéma (Obr. 7.7).



Obr. 7.7: Blokové schéma komponenty testovacího obvodu kodéru a dekodéru Iwadariho kódu

7.3.3 Implementace testovacího obvodu do platformy FIT-kit

V závěrečné fázi je potřebné navrhnout vhodnou takzvanou *top-level* entitu, jejíž signály představují jednotlivé piny fyzického *FPGA* obvodu. Dále také vhodný řídicí program, který bude z *MCU* řídit činnost obvodu. V těchto činnostech je výhodné využít předpřipravené šablony pro práci a propracované zdokumentování potřebných činností volně dostupných v rámci projektu FIT-kit (viz literaturu [14]).

Z již předpřipravených entit byla použita top-level entita *tlv_bare_ifc*, která obsahuje nejmenší počet signálů a zároveň signály představující všechny potřebné piny (tedy pro práci s *LCD* displejem pomocí SPI). Při tvoření programu pro MCU, bylo možné efektivně využít vytvořené ukázkové aplikace pro řízení činnosti čítače, který měl velmi podobnou funkci a rozhraní jako navržený testovací obvod. Kompletní potřebné zdrojové kódy a soubor *Makefile* potřebný pro překlad jsou součástí elektronických příloh tohoto dokumentu (Příloha 7). Návod pro přeložení a spuštění testovací aplikace je prezentován v dokumentu Příloha 6.

7.4 Průběh testování

Komponenty kodér a dekodér Iwadariho kódu byly na platformě FIT-kit odzkoušeny za pomocí testovacího obvodu pro několik hodnot generického parametru udávajícího korekční schopnost kódu. Testování prokázalo jednak bezchybný přenos dat při dodržení potřebných parametrů (ochranný interval a maximální délka shlukové chyby), ale také izolovaný průnik chyby (tj. bez nekonečné propagace do přenášené posloupnosti) při nedodržení některého z požadovaných parametrů.

8 Závěr

Výsledkem předkládané práce je jednak teoreticky odvozený abstraktní model kodéru a dekodéru Iwadariho kódu pro obecnou korekční schopnost (tedy matematický model a obvodové schéma). Dále pak implementace navrženého modelu ve *VHDL*, čímž byl vytvořen simulační model a po následné syntéze této implementace i fyzická realizace zkoumaného protichybového kódového systému. Verifikace a validace (či obecněji testování a ladění) bylo prováděno simulací *VHDL* modelu ve specializovaných vývojových prostředích a implementací a provozem navrženého testovacího obvodu na platformě FIT-kit.

Značnou náročnost práce způsoboval fakt nedostatku dostupných informaci o Iwadariho kódu. V podstatě nedostupnost jakýchkoliv hlubších informací o hardwarové realizaci tohoto kódu i korekčních kódů obecně. Část z uvedené literatury se sice zabývá i problematikou hardwarové realizace korekčních kódu, ovšem opět pouze z teoretického hlediska - tedy bez jakéhokoliv nástinu možných obvodových realizací potřebných komponent v hradlových polích). Bylo tedy nutné jednak mírně pozměnit v literatuře uvedený matematický model analyzovaného kódu pro možnost teoretického odvození kodeku obecné korekční schopnosti. Dále před vlastní implementací analyzovat kodek a navrhnout vhodné řešení některých dílčích problémů, jako například potřeba existence více synchronizací (vyřešena pomocí navržení bloku řízení), zpracovávání vstupu a výstupu komponentami (kdy bylo v analýze zavrženo použití řady multiplexorů, jejichž použití navrhují některé prameny použité literatury) a vlastní způsob komunikace kodéru a dekodéru (jejíž řešení bylo navrženo podle literatury s ohledem na nenáročnost a snadnou funkčnost).

Všechny popsané vyskytnuvší se problémy byly vyřešeny a zkoumaný kodek byl implementován dvěmi odlišnými způsoby popisu číslicových systému (strukturálně a behoaviorálně). Po provedení optimalizací a syntéze zvolených odlišných popisů vznikly dvě různé obvodové realizace kodeku, které si jsou velmi podobné, v některých detailech se ovšem odlišují. Obě varianty vykazují podobné a uspokojivé hodnoty charakterizujících parametrů (jako objem spotřebované logiky, maximální pracovní frekvence). Není ovšem možné jednoznačně rozhodnout, která z výsledných obvodových realizací je celkově lepší.

Jistou nevýhodou této práce je, že se zabývá Iwadariho kódem, který je poměrně zastaralý (byl odvozen v 70. letech minulého století). V dnešní době tudíž neposkytuje zcela uspokojivé vlastnosti (překonaný je především kubický růst objemu potřebné logiky s lineárním růstem korekční schopnosti a tím i nezanedbatelné informační zpoždění zprávy způsobené jejím dekódováním). Patří ovšem do skupiny konvolučních kódů, které se v současné době běžně používají v digitální komunikaci (satelitní komunikace, digitální televize, vysokorychlostní modemy, mobilní telefonie). Většina používaných kodeků jsou ovšem součástí firemního tajemství. Princip jejich hardwarové

realizace je ale velmi podobný, a provedená analýza i návrh implementace je z velké částí použitelný i na ostatní konvoluční kódy.

Hlavní možnost navázání na vytvořenou práci tedy spočívá v použití odvozených postupů a dílčích výsledků k realizaci jiného, v praxi lépe použitelného, konvolučního kódu. Dále by ovšem bylo nutné provést velký objem testování a analýzu rentability průmyslové výroby takového kodeku v podobě specializovaného hardwaru například oproti jeho implementaci na digitálním signálovém procesoru.

Ve výsledném efektu je současný přínos této práce především naučný. Demonstruje činnost protichybových kódových systémů, konkrétně prezentované na Iwadariho kódu. A také způsob návrhu, implementace a testování číslicových systémů.

Literatura

[1] Kasami, T., Tokura, N., Iwadari, J., Inagaki, J.: Těorija kodirovanija. Mír, Moskva 1978

[2] Němec, K.: Řešení požadavků kladených na protichybový kódový systém. 2006, č. 17, str. 1-8, ISSN 1213-1539. Dokument dostupný na URL http://147.229.144.23/clanky/06017/ (duben 2007)

[3] Vlček, K.: Teorie informace, kódování a kryptografie. VŠB – Technická univerzita Ostrava, Ostrava 1998.

[4] Němec, K.: Kódové zabezpečovací systémy. Učební text pro kurs Kódové zabezpečovací systémy. ÚTKO FEI VUT Brno, Brno 1995.

[5] Nečas, J. a kol.: Oborové encyklopedie - Aplikovaná matematika I. Praha, SNTL 1977.

[6] KTD, Česká terminologická databáze knihovnictví a informační vědy (TDKIV). Praha, Národní knihovna České republiky, 2003. Dostupné na URL http://sigma.nkp.cz/cze/ktd (duben 2007)

[7] Hruška, T.: Informační systémy, studijní opora. 2007, FIT VUT Brno. Dokument dostupný na URL https://www.fit.vutbr.cz/study/courses/WAP/private/opory/OporaIIS2PISPojemDataProcesy Transakce.pdf (duben 2007)

[8] Křivánek, V.: Návrh kodérů a dekodérů umožňující opravu shlukových chyb a jejich simulace. Elektrorevue, 2006, č. 8., str. 1-9, ISSN 1213-1539. Dokument dostupný na URL http://147.229.144.23/clanky/06008/ (duben 2007).

[9] Němec, K.: Simulace protichybových kódů v prostředí visicode. VUT FEKT Brno, 2003.

[10] Kolouch, J.: Programovatelné logické obvody a modelování číslicových systémů v jazycích ABEL a VHDL. VUT Brno, 2000.

[11] Fučík, O.: Návrh číslicových systémů: Studijní opora. FIT VUT v Brně, 2007. Dokument dostupný na URL https://www.fit.vutbr.cz/study/courses/INC/private/inc_text_20070206.pdf

[12] Sekanina, L.: Návrh počítačových systémů - Řešené a neřešení příklady. 2006, FIT VUT Brno. Dokument dostupný na URL https://www.fit.vutbr.cz/study/courses/INP/private/cvic/inp_cv_opora.pdf (duben 2007).

[13] Rábova, Z., Češka, M., Zendulka, J., Peringer, P., Janoušek, V.: Modelování a simulace - studijní opora. 2005, FIT VUT Brno. Dokument dostupný na URL https://www.fit.vutbr.cz/study/courses/IMS/private/SkriptaMS.pdf (duben 2007)

[14] FIT-kit: Domovská web stránka FIT-kitu, 2007. Dostupné na URL http://merlin.fit.vutbr.cz/FITkit/ (duben 2007).

[15] Xilinx: Spartan-3 FPGA Family - Complete Data Sheet. 2005, p. 198. Dokument dostupný na URL http://merlin.fit.vutbr.cz/FITkit/doc/hardware/datasheet/Spartan-3.pdf (duben 2007).

[16] Wikipedia: Linear feedback shift register. 2007, p. 7. Dokument dostupný na URL http://en.wikipedia.org/wiki/Linear_feedback_shift_register (duben 2007).

[17] Šusta, R.: Logické řízení, studijní opora. 2006, Katedra řídící techniky ČVUT Praha. Dokument dostupný na URL http://dce.felk.cvut.cz/lor/prednasky/LOR8.pdf (duben 2007).

Seznam příloh

Příloha 1. vygenerované RTL schéma strukturálně popsaného kodéru (block_lenght = 4)

Příloha 2. vygenerované RTL schéma behaviorálně popsaného kodéru (block_lenght = 4)

Příloha 3. vygenerované RTL schéma strukturálně popsaného dekodéru (block_lenght = 3)

Příloha 4. vygenerované RTL schéma behaviorálně popsaného dekodéru (block_lenght = 3)

Příloha 5. vygenerované RTL schéma testovacího obvodu implementovaných komponent

Příloha 6. manuál k použití přiložených zdrojových kódů

Příloha 7. CD se složkami doc - elektronická forma textu bakalářské práce

src - všechny vytvořené a použité zdrojové kódy

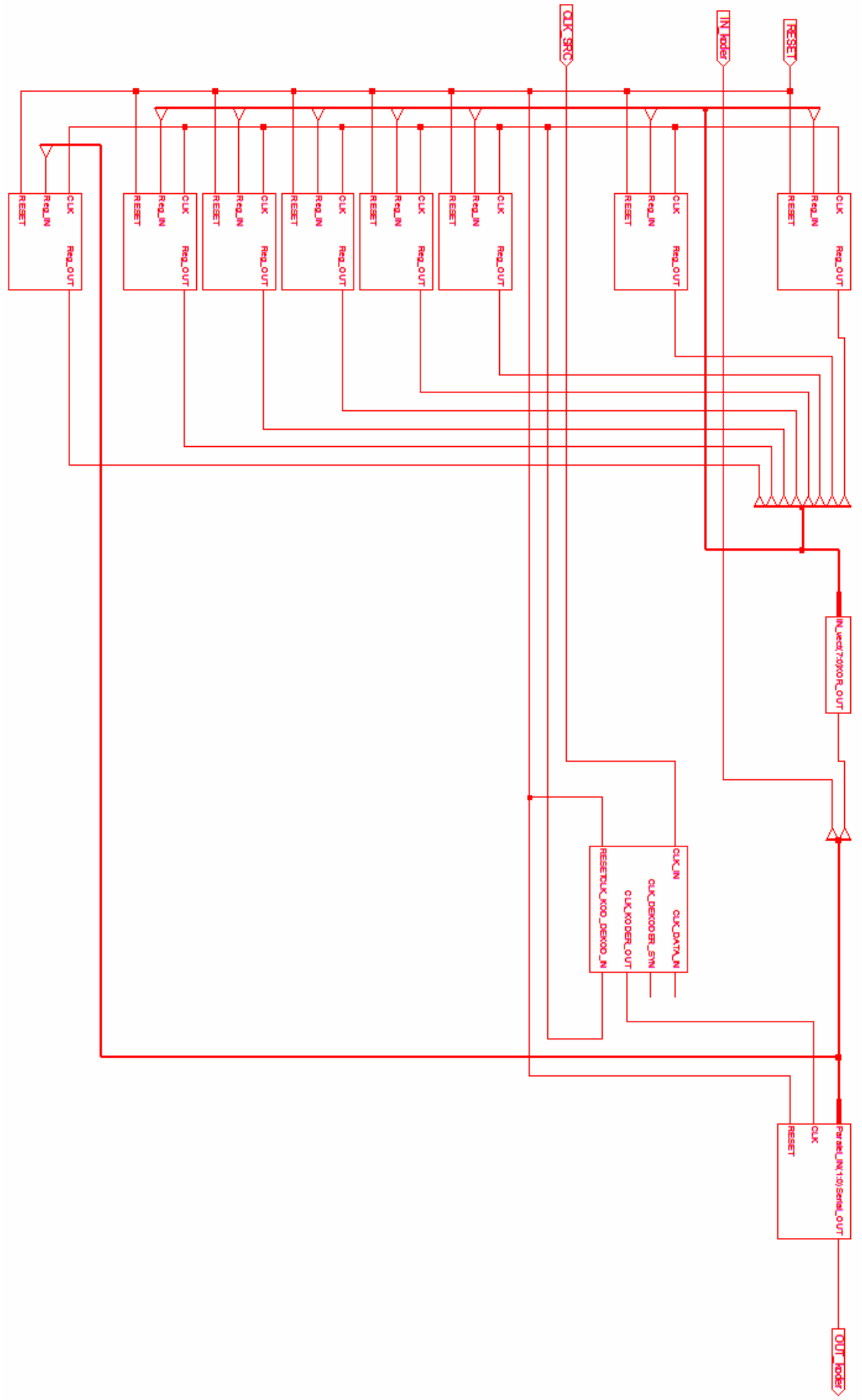
behavioral - zdrojové kódy behaviorálního popisu kodeku

structural - zdrojové kódy strukturálního popisu kodeku

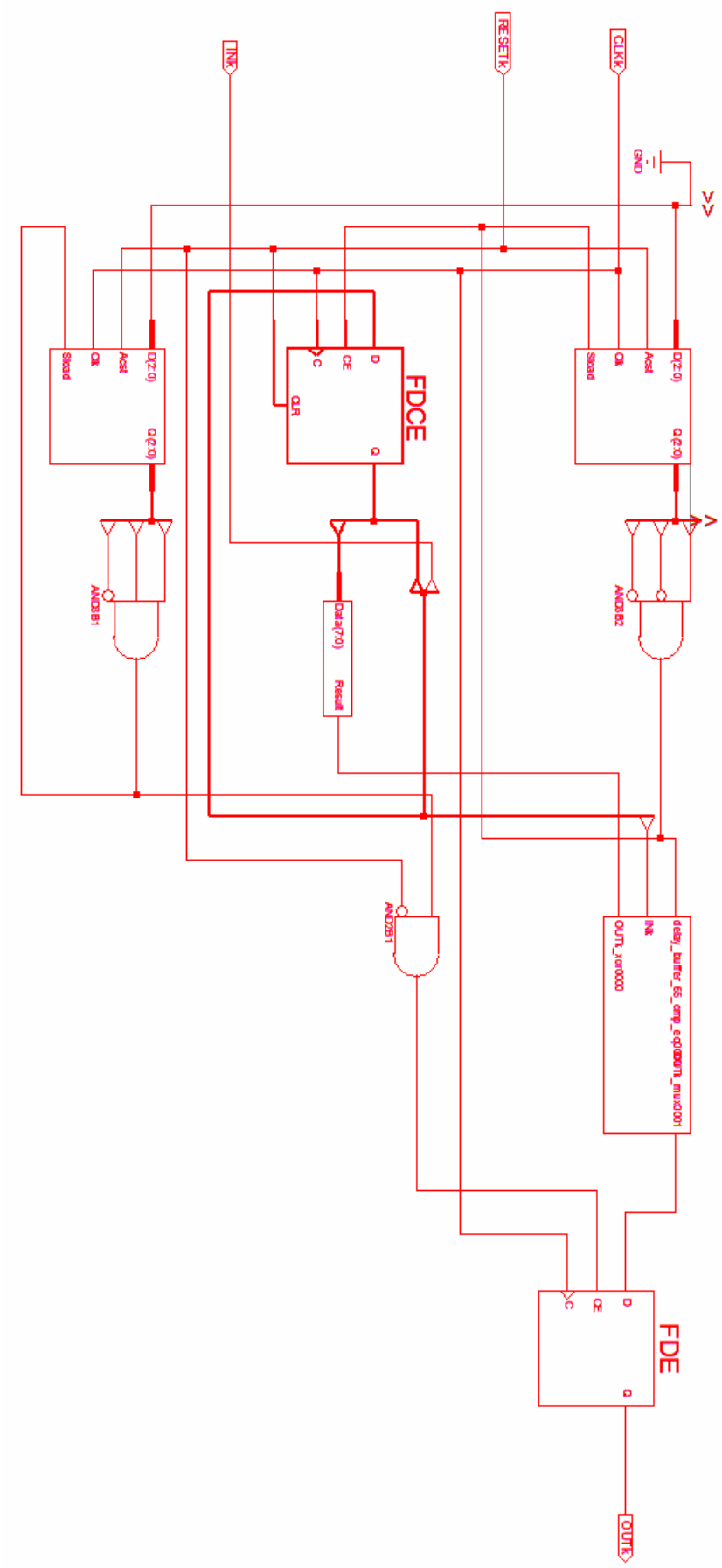
FITkit.rev195 - kopie úložiště zdrojových kódů projektu FIT-kit

testovaci_obvod - zdrojové kódy testovacího obvodu

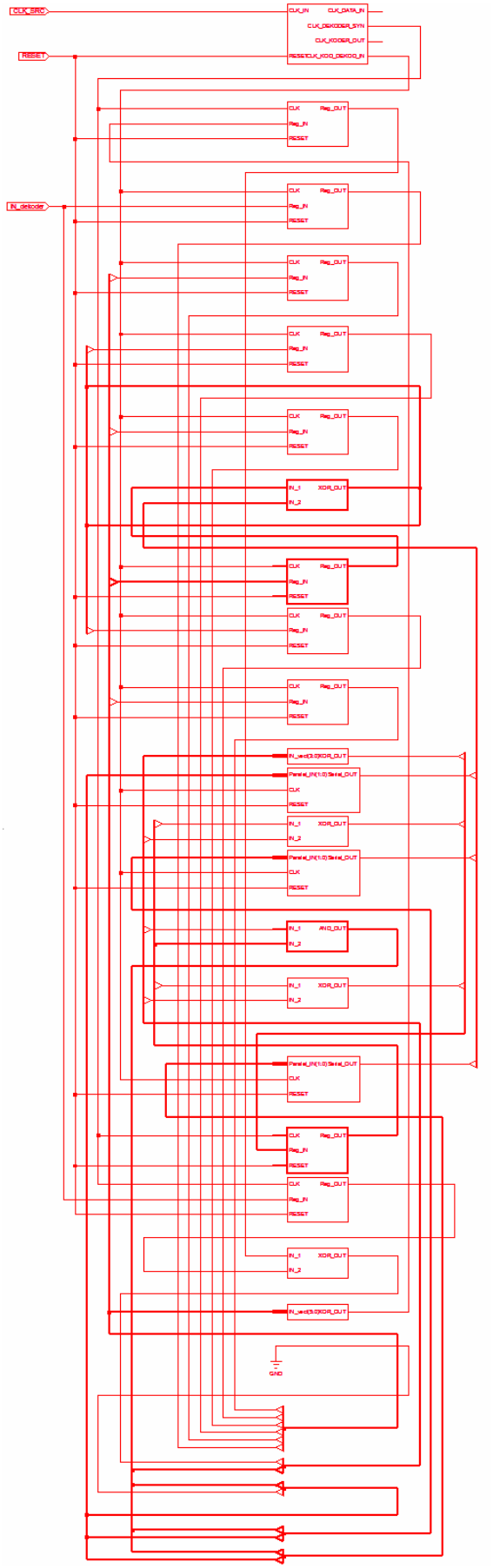
testovaci_obvod_SW - program pro MCU řídící testovací obvod



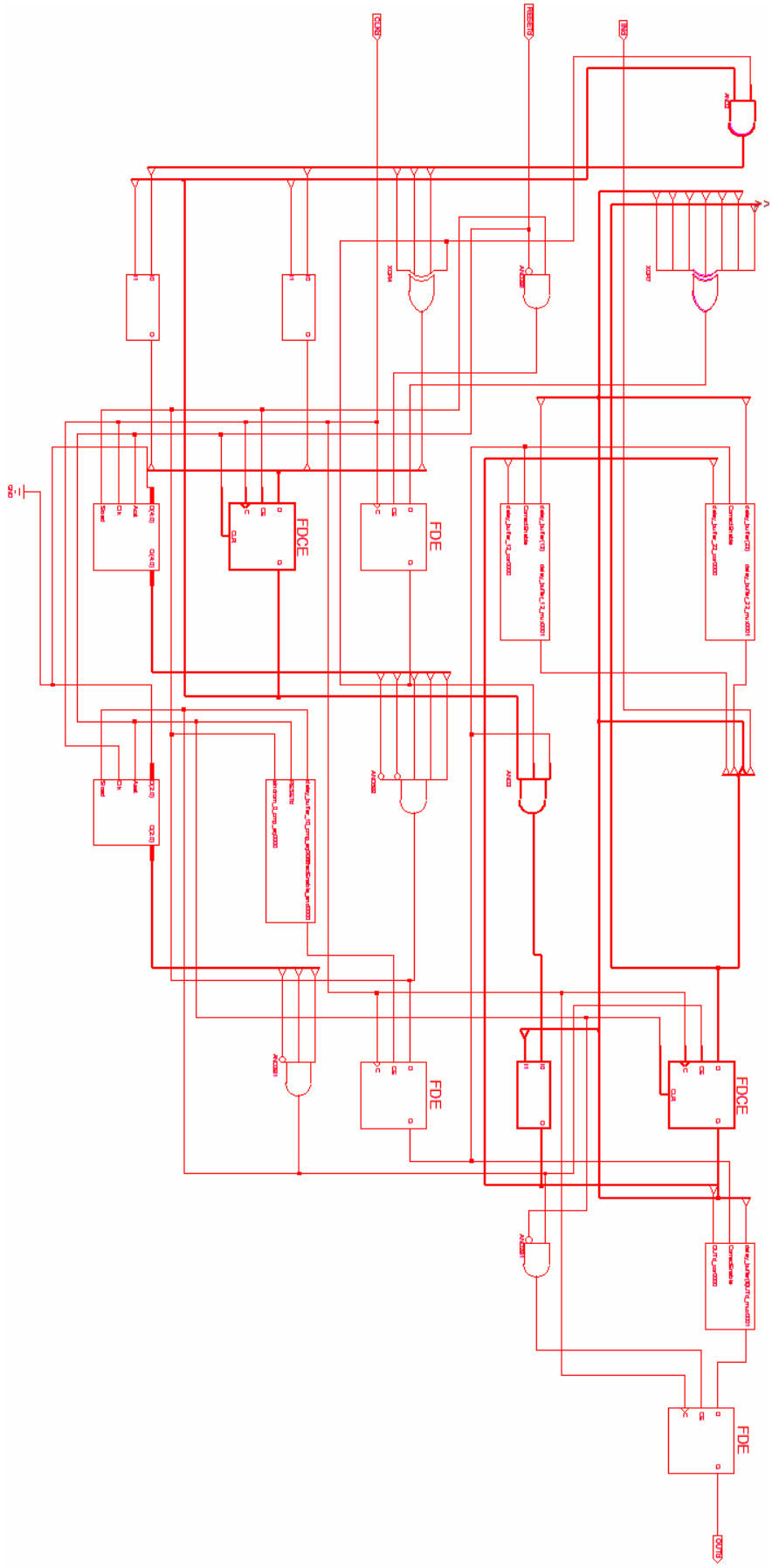
Příloha 1: vygenerované RTL schéma strukturálně popsaného kodéru (block_lenght = 4)



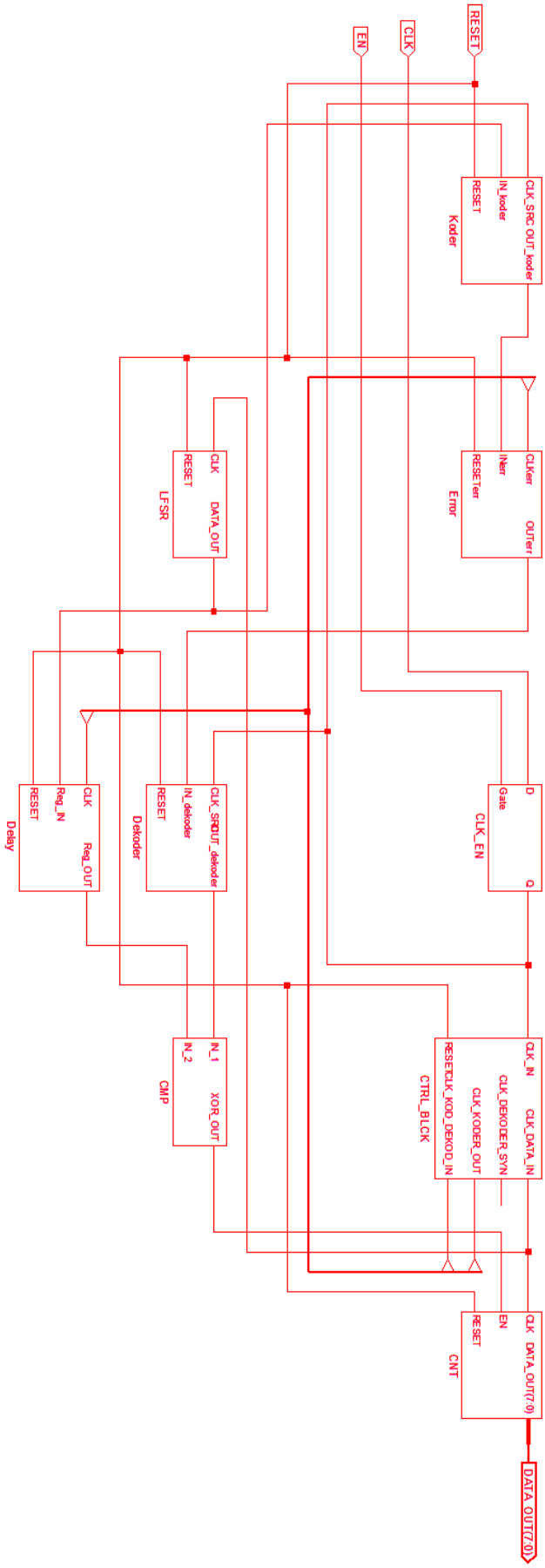
Příloha 2: vygenerované RTL schéma behaviorálně popsaného kodéru (block_lenght = 4)



Příloha 3: vygenerované RTL schéma strukturálně popsaného dekodéru (block_lenght = 3)



Příloha 4: vygenerované RTL schéma behaviorálně popsaného dekodéru (block_lenght = 3)



Příloha 5: vygenerované RTL schéma testovacího obvodu implementovaných komponent

Manuál k použití zdrojových kódů

Všechny vytvořené zdrojové kódy jsou na přiloženém CD v adresáři *src*. Adresář *src* obsahuje podadresáře *FITkit.rev195* (s poslední revizí zdrojových kódů k platformě FIT-kit), *behavioral*, *structural* (se zdrojovými kódy pro dva alternativní způsoby popisu kodeku Iwadariho kódu), *testovaci_obvod* a *testovaci_obvod_SW* (se zdrojovými kódy pro testování komponent na platformě FIT-kit).

Postup simulace komponent

Každý z alternativních popisů těl (strukturální a behaviorální) entit kodeku mají v příslušné složce podsložku *BenchTests*. Ta obsahuje testovací pomocné komponenty pro otestování každé z popsaných komponent (i pomocných komponent, které byly využity pouze jako součást kodeku). V podsložce ModelSim_Models se nacházejí projekty vývojového prostředí *ModelSim XE II*, pomocí kterých je možné simulovat příslušná zapojení testovacích komponent a komponent testovaných. Projekty jsou nakonfigurované tak, aby je bylo možné spouštět přímo z CD (vývojové prostředí používá absolutní cesty ke zdrojovým kódům, ty by bylo tedy nutné po přemístění projektů upravit).

Postup syntézy a realizace komponent v FPGA

Pro úspěšné otestování komponent na platformě FIT-kit musíme nakonfigurovat *FPGA* a naprogramovat mikrokontrolér programem podporujícím činnost obvodu v *FPGA*.

Program pro mikrokontrolér se nachází ve složce *testovaci_obvod_SW*. Zde je i soubor *Makefile*, pomocí něhož je možné pohodlně program přeložit a nahrát do *MCU* FIT-kitu následovně:

```
...\src\testovaci_obvod_SW> make
...\src\testovaci_obvod_SW> make load
```

Ve složce *testovaci_obvod* jsou zdrojové texty komponent a soubor *Makefile*, pomocí nichž je možné vytvořit binární konfigurační soubor testovacího obvodu s behaviorálně nebo strukturálně popsanými komponentami kodeku. Pomocí příkazového řádku je možné vytvořit konfigurační soubory následovně:

```
...\src\testovaci_obvod> make ARCHITECTURE=structural
```

anebo:

```
...\src\testovaci_obvod> make ARCHITECTURE=behavioral
```

První volba je implicitní - vyvolá se i zadáním příkazu make bez parametrů. Po vyvolání překladu se v aktuální složce vytvoří soubory spojené se syntézou komponent. Podle zvolené architektury budou mít soubory jméno *output_struct* popřípadě *output_beh* a upřesňující koncovky. Pro nakonfigurování *FPGA* FIT-kitu použijeme příslušný binární konfigurační soubor (*output_struct.bin* nebo *output_beh.bin*) . Postup jak FIT-kit konfigurovat nalezneme například v literatuře [14].

Po úspěšném naprogramování *MCU* a nakonfigurování *FPGA* FIT-kitu se na LCD obrazovce budou zobrazovat údaje o celkovém počtu dosud přenesených bitů a celkovém počtu chybně dekódovaných bitů (z důvodu nedostatku zobrazitelných znaků na displeji jsou ve skutečnosti zobrazovány jen poslední tři číslice počtu přenesených bitů a poslední dvě číslice počtu chybně dekódovaných bitů. Pro demonstrační účely je toto ovšem plně postačující). Údaj na LCD obrazovce pak může vypadat následovně:

```
send:596 errs:00
```

Je možné testovat i různé instance generických komponent (tj. kodér a dekodér různých zabezpečovacích schopností). Ve zdrojovém souboru *src\testovaci_obvod\top_level.vhd* se změnou implicitní hodnoty generického parametru entity *Tb_KoderDekoder* docílí změny generického parametru v instancích kodéru a dekodéru zapojených v testovacím obvodu.

```
...\src\testovaci_obvod> nl top_level.vhd
...
209 tb: Tb_KoderDekoder GENERIC MAP (block_length => 3)
            -- <----Zmenou tohoto parametru budeme testovat syntezu
210      port map (  --  instance kodeku o ruzne zabezp. schopnosti
211         CLK       => clk,
212         RESET     => rst,
213         EN        => cnt_write_en,
214         DATA_OUT  => cnt_data_in
215      );
...
```

Pro otestování překročení zabezpečovacích schopností kódu je možné modifikovat komponentu *error_provider* v souboru *src\testovaci_obvod\error_provider.vhd*. Tato komponenta slouží k zašumění přenášené posloupnosti bitů shlukovými chybami o specifickém rozmístění a délce. Komponenta poškodí shluk $k + 1$ bitů, vždy po průchodu A bitů (což odpovídá korekční schopnosti a ochrannému intervalu - viz vzorce (4.2) až (4.5)). Pokud tedy ve zmíněném souboru na vyznačeném místě pozměníme porovnávání vnitřních čítačů tak, že bude vytvářet delší chyby nebo v kratších vzdálenost (i když by se jednalo o jediný bit), nebude již dekodér schopen chyby opravovat. Zmíněný úsek kódu:

```
...\src\testovaci_obvod> vim error_provider.vhd
...
--         ZDE MUZEME PREKROCIT ZABEZPECOVACI SCHOPNOST KODU          --
-------------------------------------------------------------------------
if counter = interval then         -- dosahli jsme ochranneho intervalu A
 if errs = block_length + 1 then -- dokud neni zhluk. chyba k+1 bitu
-------------------------------------------------------------------------
...
```

Můžeme opravit například na kód:

```
...
if counter = interval then         -- dosahli jsme ochranneho intervalu A
   if errs = block_length + 2 then -- dokud neni zhluk. chyba k+2 bitu
...
```

Nebo na kód:

```
...
if counter = interval - 2 then     -- dosahli jsme ochranneho intervalu A
   if errs = block_length + 1 then -- dokud neni zhluk. chyba k+2 bitu
...
```

Vzniklé překročení zabezpečovací schopnosti kódu se na výstupu testovací komponenty projeví postupnou inkrementací výstupu čítajícího počet chybně dekódovaných bitů. Na LCD obrazovce FIT-kitu zaznamenáme tuto inkrementaci zobrazením průběžného výsledku, jako je například tento:

```
send:896 errs:03
```