FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Návrh spolehlivých systémů v FPGA s použitím bezpečnostních kódů
Student:	Bc. Vojtěch Pail
Vedoucí:	Ing. Pavel Kubalík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Návrh a programování vestavných systémů
Katedra:	Katedra číslicového návrhu
Platnost zadání:	Do konce letního semestru 2020/21

Pokyny pro vypracování

1) Prozkoumejte existující metody řešení.
2) Analyzujte vlastnosti různých typů kombinačních obvodů pomocí sady benchmarků z hlediska odolnosti proti poruchám. Využijte simulační software dostupný na KČN.
3) Na základě takto získaných dat nalezněte vhodný bezpečnostní kód, který bude schopen tyto poruchy detekovat, popř. i opravovat tak, aby redundance (area overhead) byla co nejmenší.
4) Specifikujte požadavky na úpravu simulačního softwaru tak, aby obsahoval podporu pro výběr nejvhodnějšího kódu.
5) Navržený způsob řešení ověřte na několika příkladech konkrétních obvodů a kódů.
6) Vytvořte klasifikaci obvodů s hlediska možností opravy/detekce poruch.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 5. února 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Návrh spolehlivých systémů v FPGA s použitím bezpečnostních kódů

Bc. Vojtěch Pail

Katedra číslicového návrhu
Vedoucí práce: Ing. Pavel Kubalík, Ph.D.

7. ledna 2021

Poděkování

Děkuji své rodině a svému vedoucímu práce za poskytnutou podporu, pomoc a rady. Dále děkuji Jiřímu Němcovi za pomoc při typografické kontrole a Jaroslavu Boreckému za simulaci poruch v obvodech.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisu, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programu, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli zpusobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelum). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným zpusobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. ledna 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2021 Vojtěch Pail. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Pail, Vojtěch. *Návrh spolehlivých systémů v FPGA s použitím bezpečnostních kódů*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Cílem této práce je analýza poruchového chování obvodů pomocí simulačního softwaru KČN. Pro obvody jsou nalezeny vhodné bezpečnostní kódy, které jsou schopné detekovat, případně opravovat tyto poruchy s co nejmenší redundancí. Tato práce specifikuje požadavky na úpravu simulačního software pro možnost výběru nejvhodnějšího kódu.

Klíčová slova FPGA, bezpečnostní kód, porucha, chyba, kombinační obvod

Abstract

The aim of this diploma thesis is to analyze the fault behavior of circuits using the simulation software of the Department of Digital Design. Suitable safety codes capable of detecting or correcting these faults with the least possible redundancy are found for the circuits. This thesis specifies the requirements for modifications of the simulation software for the possibility of selecting the most suitable code.

Keywords FPGA, error control code, fault, error, combinational circuit

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Výrobní procesy digitálních obvodů se neustále zlepšují. Avšak spolu s tím, jak roste hustota integrace, roste i náchylnost obvodů k poruchám a následným chybám až selháním. V současnosti jsou velmi populární využívat jako implementační platformu pro hardware FPGA, především díky své nízké ceně, snadnému návrhu a testování obvodů, či flexibilitě — navržený obvod je možné později znovu upravit. FPGA jsou tvořeny převážně SRAM pamětmi, které jsou obzvláště náchylné k poruchám způsobeným zářením, například zásahem nabité částice z kosmu. Takový zásah paměti může způsobit i změnu struktury navrženého obvodu, a tím omezit nebo úplně přerušit jeho správnou činnost. Zabezpečení digitálních obvodů se proto stává stále důležitějším. Již při návrhu je nutné počítat s výskytem poruch a návrh obvodu přizpůsobit tak, aby v maximální míře zamezil jejich dopadu na činnost výsledného produktu. Tato práce se zabývá zabezpečením kombinační logiky pro FPGA pomocí bezpečnostních kódů s co nejmenší přidanou plochou. Pro co nejlepší zabezpečení využívá simulační a emulační software pro testování vlivu poruch na kombinační obvody, který je dostupný na KČN FIT ČVUT. Dále pak analyzuje různé druhy kombinačních obvodů z hlediska možnosti zabezpečení navrženou metodou a navrhuje rozšíření simulačního softwaru o výběr optimálního kódu pro zabezpečení testovaného obvodu.

Kapitola 1

Cíl práce

Cíle práce jsou:

- Prozkoumání existujících řešení testování poruch v kombinačních obvodech zabezpečených bezpečnostními kódy;
- Analyzování vlastností různých typů kombinačních obvodů pomocí sady benchmarků z hlediska odolnosti proti poruchám za využití simulačního softwaru dostupného na KČN;
- Nalezení vhodného bezpečnostního kódu pro detekci, případně opravování těchto poruch na základě získaných dat tak, aby redundance (area overhead) byla co nejmenší;
- Specifikování požadavků na úpravu simulačního softwaru tak, aby obsahoval podporu pro výběr nejvhodnějšího kódu;
- Ověření navrženého způsobu řešení na příkladech konkrétních obvodů a poruch;
- Vytvoření klasifikace obvodů z hlediska možností opravy/detekce poruch.

Kapitola 2

Existující řešení

V této kapitole jsou popsána existující řešení problematiky zabezpečení kombinační logiky pomocí bezpečnostních kódů. Základní pojmy problematiky zabezpečení a testování digitálních obvodů jsou definované a popsané v sekcích 3.1, 3.2 a 3.3.

2.1 Zabezpečení sčítačky pomocí TMR a sudých parit

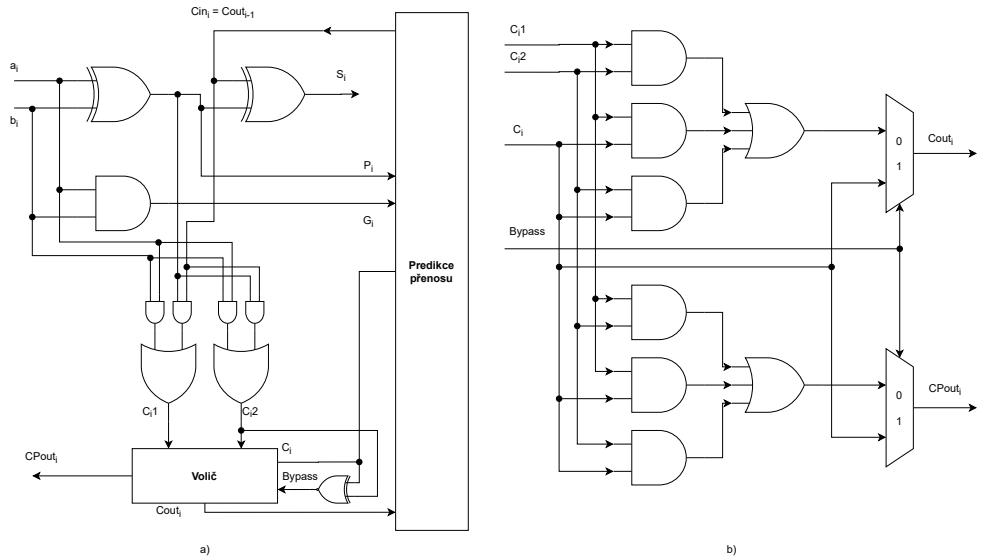
Autor v článku **A Novel Self-Checking Carry Lookahead Adder with Multiple Error Detection/Correction**[1] popisuje zabezpečení úplné sčítačky s predikcí přenosu. tak, aby byla schopná opravovat, případně alespoň detekovat všechny jednobitové chyby a s určitou pravděpodobností i chyby postihující více bitů. Zabezpečení je docíleno kombinací dvou aspektů.

1. **Predikce parity** — detekce chyb pomocí predikované hodnoty sudé parity výstupu a sudé parity skutečného výstupu. Prediktor parity pro úplnou sčítačku určuje očekávanou paritu následujícím způsobem:

$$P_S = P_A \bigoplus P_B \bigoplus P_C \tag{2.1}$$

   kde P_S je predikovaná parita, P_A a P_B parita vstupních operandů A, B a P_C je parita vnitřních přenosů. Pro výpočet parity vnitřních přenosů jsou použity redundantní přenosy, které slouží pouze pro výpočet predikované parity a ne pro samotnou činnost sčítačky. Redundantní přenosy nejsou počítány nezávisle, ale kvůli snížení plochy sdílí jedno hradlo XOR v každém řádu sčítačky s výpočtem součtu a parity, viz obr. 2.1.

2. **TMR** — zabezpečuje přenosy mezi řády sčítačky. Protože TMR přináší velkou redundanci, autor zvolil několik optimalizací, aby tuto plochu co



Obrázek 2.1: a) Sdílení hradla XOR pro výpočet přenosu, redundantního přenosu a součtu v i-tém bitu úplné sčítačky. b) Struktura voliče s bypass signálem.[1]

nejvíce zredukoval. Jako jeden vstup do majoritního voliče je zaveden běžný přenos. Protože je nutné počítat redundantní přenos kvůli predikci parity, je i tento redundantní přenos použit jako vstup do voliče a výpočet obou přenosů sdílí hradlo XOR (obr. 2.1 a)). Protože autor používá sčítačku s predikcí přenosu, třetím vstupem do majoritního voliče je predikovaný přenos z prediktoru přenosů. Volič má ještě jeden vstupní signál — bypass. Ten v případě, že predikovaný přenos je stejný jako jeden z vypočtených nabývá hodnoty log. 1, pro zrychlení obejde funkci voliče a propaguje jeden z přenosů rovnou na výstup voliče. V případě, že se přenosy liší, pravděpodobně došlo k chybě, bypass signál zůstane v log. 0 a volič případnou chybu vymaskuje. Princip je zobrazen na obrázku 2.1.

Autor obvod rozdělil na dvě části: **Detekční část** — majoritní volič, hradla XOR generující součet a přenos v každém řádu a hradla XOR generující predikovanou a skutečnou hodnotu parity výstupu. A **opravnou část** — logika pro generování hlavního přenosu, hradla AND produkující signál G_i pro prediktor přenosu, hradla produkující dva redundantní přenosy v každém řádu a XNOR hradlo generující signál bypass pro volič. Autor dokázal, že navržená architektura má tyto vlastnosti:

- TMR je schopné opravit všechny chyby vzniklé v důsledku poruchy v ko-

rekční části.

- Zabezpečení pomocí parity a její predikce detekuje všechny chyby vzniklé poruchou postihující hradlo XOR, které generuje signál P_i pro prediktor přenosu.

- Parita a její predikce detekují všechny chyby vzniklé poruchou postihující hradla, která generují oba redundantní přenosy.

- Takto zabezpečená sčítačka opravuje nebo alespoň detekuje všechny jednobitové chyby. Dokáže opravit/detekovat i některé chyby postihující více bitů až do velikosti n, kde n je šířka sčítačky, avšak pravděpodobnost detekce/opravy n chyb s rostoucím n rychle klesá. Aby bylo možné opravit více chyb, musí všechny nastat v korekční části architektury a navíc každá v jiném řádu sčítačky.

V tabulkách 2.1 a 2.2 je zobrazeno porovnání navržené architektury s DMR a TMR z hlediska hardwarové redundance a možností detekce/opravy chyb. Můžeme vidět, že navržená architektura je větší než DMR, ale má lepší vlastnosti a dokáže určité procento jednobitových chyb kromě detekce i opravit.

Tabulka 2.1: HW redundance navržené architektury porovnané s DMR a TMR.[1]

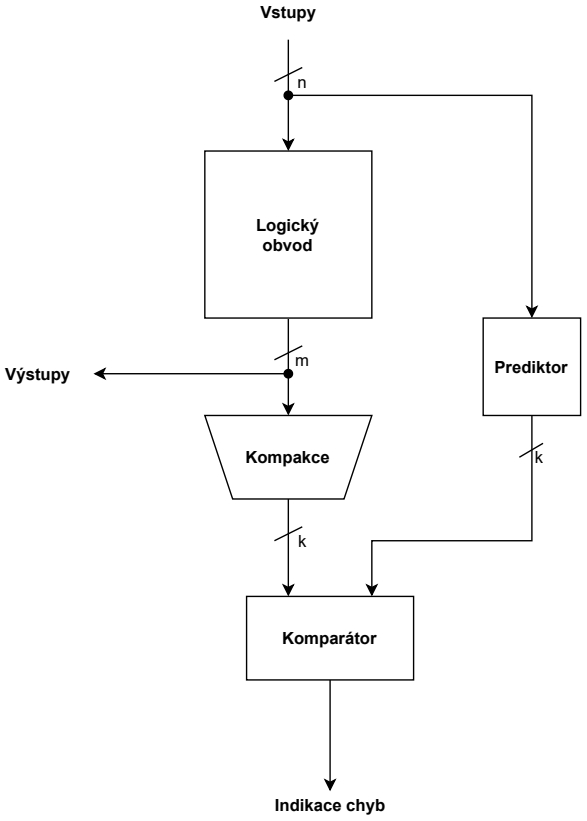
Sčítačka	**Předst. arch.**	**DMR**	**TMR**
Velikost	**HW redundance**		
8-bitová	137,5%	112,1%	227,3%
16-bitová	136,9%	112,5%	227,3%
32-bitová	136,6%	112,5%	227,3%
64-bitová	136,5%	112,5%	227,3%

Tabulka 2.2: Porovnání detekčních a korekčních vlastností navržené architektury, DMR a TMR.[1]

Sčítačka	**Předst. arch.**		**DMR**		**TMR**	
Velikost	P_{cor}%	P_{det}%	P_{cor}%	P_{det}%	P_{cor}%	P_{det}%
8-bitová	53,6	46,4	0	100	91,7	0
16-bitová	53,7	46,3	0	100	91,7	0
32-bitová	53,8	46,2	0	100	91,7	0
64-bitová	53,8	46,2	0	100	91,7	0

2.2 Zabezpečení kombinační logiky pomocí upravených detekčních a korekčních kódů

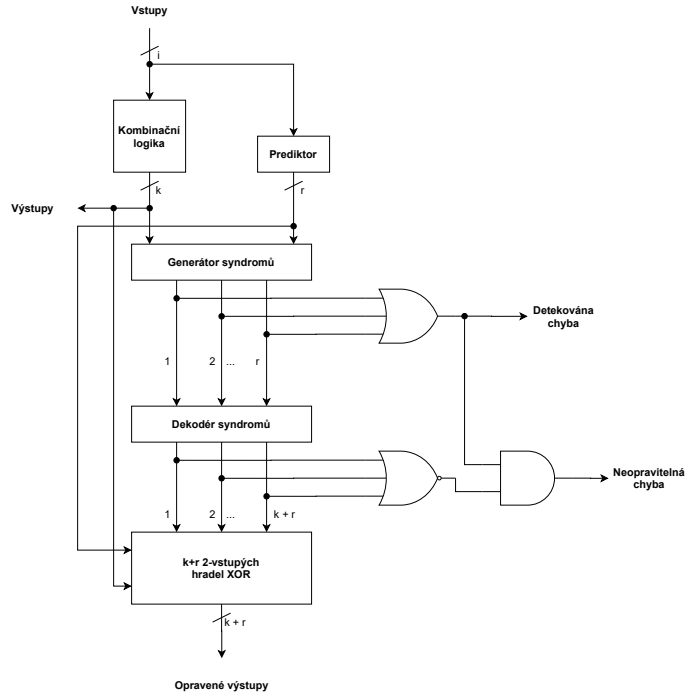
Autoři v článku **Combinational Logic Circuit Protection Using Customized Error Detecting and Correcting Codes**[2] popisují možnosti zabezpečení obecné kombinační logiky pomocí detekčních/opravných kódů. Jako základní přístup k detekci chyb pomocí bezpečnostních kódů uvádějí následující architekturu [3], znázorněnou na obr. 2.2. Zde vidíme kombinační obvod o n vstupech a m výstupech. Na výstup je připojen redukční obvod, který počet výstupů sníží z m na k. Dále je součásti architektury prediktor, který těchto k bitů nezávisle predikuje v závislosti na vstupech obvodu. Těchto k bitů z redukčního obvodu a k bitů z prediktoru je porovnáváno a při nesouladu hodnot je detekována chyba na výstupu obvodu. Jedním z možných přístupů je použití operace XOR na různé podmnožiny výstupů kombinačního obvodu, tedy využití parit.



Obrázek 2.2: Základní architektura využívající bezpečnostní kódů pro detekci chyb na výstupu kombinační logiky.[2]

Autoři článku tuto architekturu rozšířili o možnost opravy nejčastějších

chyb. Takto upravená architektura je zobrazená na obr. 2.3. Pro vytvoření architektury je zapotřebí kombinační obvod, seznam poruch podle zvoleného poruchového modelu a počet simulací (L) k provedení u každého modelu poruchy. Proces zabezpečení pak probíhá následujícím způsobem.



Obrázek 2.3: Navržená architektura pro opravu nejčastějších chyb na výstupu kombinační logiky.[2]

- Náhodně se simuluje L vstupních vektorů pro každou poruchu v seznamu poruch. Výstupem jsou chybové vektory, které získáme jako XOR správné a chybné odezvy obvodu, a které jsou řazeny dle četnosti výskytu.
- Vytvoří se bezpečnostní kód schopný opravovat nejčastější získané chybové vektory. Sestaví se kontrolní matice tohoto kódu a z matice se získají lineární rovnice pro každý paritní bit.
- Vytvoří se prediktor. Vznikne XORem příslušných výstupů původního kombinačního obvodu podle rovnic získaných v předchozím kroku tak, aby prediktor generoval paritní bity zvoleného kódu.
- Z výstupu kombinačního obvodu a predikovaných paritních bitů se vypočte syndrom. V případě nenulového syndromu je detekována chyba. Syndromy dále vstupují do dekodéru syndromů, jehož výstup vede do logiky opravující chybný výstup a detekující neopravitelnou chybu.

V tabulce 2.3 jsou zobrazeny výsledky aplikace této architektury na vybrané obvody z benchmarkové sady MCNC. U každého obvodu autoři použili dva až tři různé bezpečnostní kódy a porovnali jejich detekční a korekční vlastnosti, a také hardwarovou redundanci příslušného prediktoru.

Tabulka 2.3: HW redundance prediktoru a detekční/korekční schopnosti pro různé kódy pro vybrané obvody z MCNC benchmarkové sady.

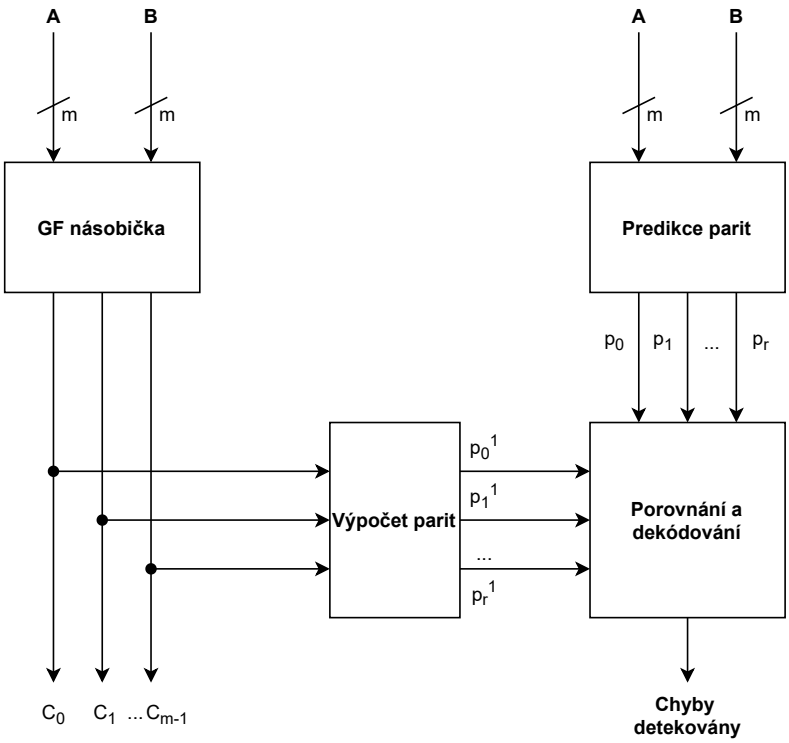
Obvod	Pokrytí v %		HW redund. prediktoru
	Detekce	Korekce	
sao2	99,0	00,0	62,0
	99,8	88,3	84,1
cu	91,9	00,0	78,9
	90,1	50,7	143,4
b12	97,1	00,0	20,5
	89,1	36,2	48,9
	90,1	84,6	75,0
misex1	95,9	00,0	44,1
	97,6	83,6	84,4

2.3 Použití bezpečnostních kódů pro zabezpečení násobiček v aritmetice GF

Článek **Multiple bit Error Detection and Correction in GF Arithmetic Circuits**[4] pojednává o možnostech zabezpečení násobiček provádějících násobení prvků z tělesa $GF(p^m)$ (konkrétně $GF(2^m)$), které se využívají například v kryptografickém hardwaru pro výpočty s eliptickými křivkami. Autoři uvádějí dva přístupy. Jeden pro detekci chyb a druhý i pro korekci chyb v těchto násobičkách.

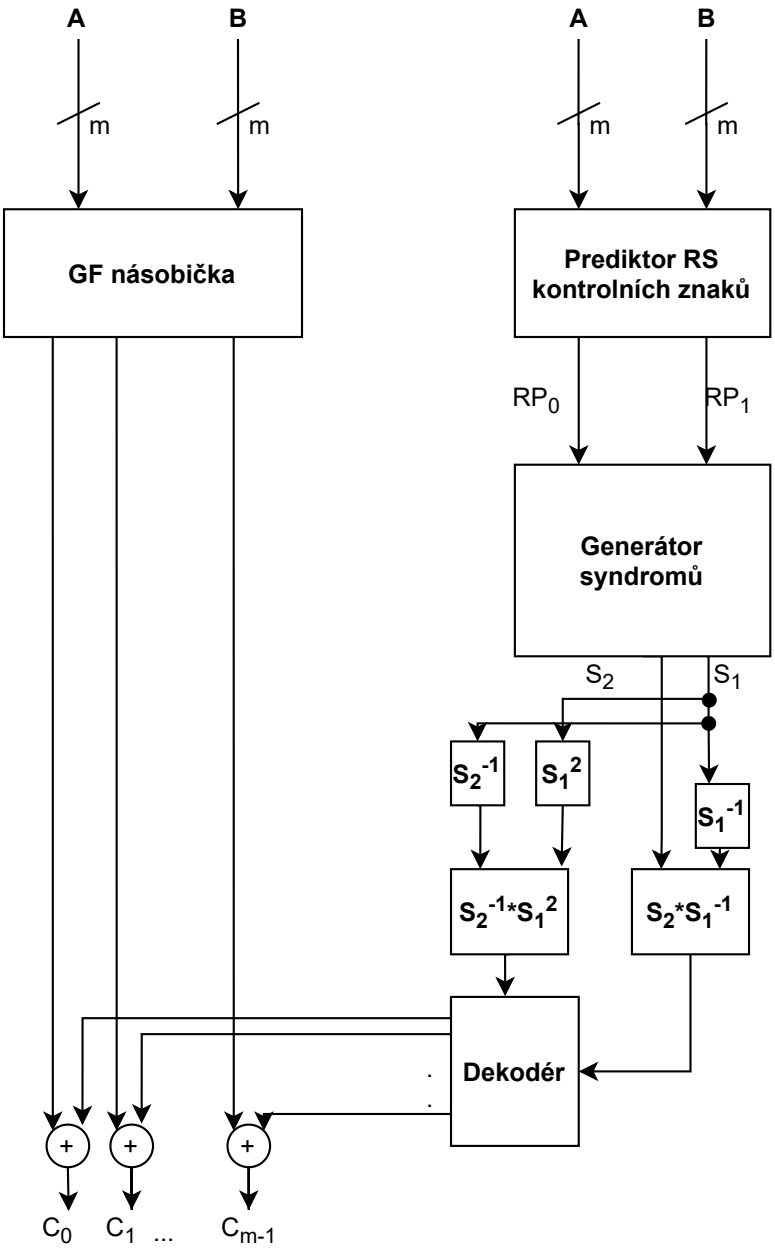
Vícenásobná detekce chyb — autoři uvádějí příklad zabezpečení paralelních násobiček proti vícenásobným chybám. Zabezpečení je založeno na paritních bitech a jejich predikci. Z výstupu paralelní násobičky se vypočítají paritní bity a obvod prediktoru tyto parity určuje nezávisle ze vstupů obvodu. U prediktoru autoři využili rovnic pro výpočet součinu v tělese $GF(2^m)$ a pro predikci jednotlivých parit odvodili uzavřené výrazy, ze kterých prediktor následně sestavili. Navržená architektura je zobrazena na obr. 2.4.

Oprava vícenásobných chyb — Pro architekturu schopnou opravovat vícenásobné chyby se autoři rozhodli použít Reed-Solomonovy (RS) kódy, což jsou blokové (n, k) kódy. Kódová slova RS kódů jsou symboly z tělesa $GF(2^m)$. Platí $n = 2^m - 1$, k je počet informačních znaků, $r = n - k = 2t$



Obrázek 2.4: Architektura pro detekci vícenásobných chyb pro násobičku prvků tělesa $GF(2^m)$.[4]

a $d_{min} = 2t + 1$. Architektura pro detekci a opravu chyb je zobrazená na obr. 2.5. Stejně jako u architektury pro detekci vícenásobných chyb obsahuje prediktor, který tentokrát nezávisle predikuje ze vstupů dva symboly RS kódu. Dále architektura obsahuje generátor syndromů, který z kontrolních symbolů RS kódu a výstupu násobičky určí syndromy. Tyto syndromy dále projdou logikou pro opravu, která určí pozici a délku shluku chyb na výstupu násobičky a provede korekci.



Obrázek 2.5: Struktura architektury pro opravu chyb pro násobičku prvků tělesa $GF(2^m)$[4].

Kapitola 3

Analýza a návrh

V této kapitole jsou definovány základní pojmy oblasti testování digitálních obvodů, spolehlivosti a bezpečnostních kódů. Dále je zde na základě analýzy existujících řešení navržena vlastní architektura pro zabezpečení kombinačních obvodů založená na bezpečnostních kódech.

3.1 Vymezení základních pojmů

V této kapitole jsou definovány základní pojmy týkající se problematiky spolehlivosti a testování hardwarových zařízení.

Defekt je fyzikální událost způsobující nechtěný rozdíl mezi zamýšleným designem zařízení a jeho implementací, který může způsobit, že se zařízení nebude chovat podle svojí specifikace a nebude správně vykonávat svojí funkci. K výskytu defektu může dojít kdekoliv v hierarchii zařízení. Defekty mohou vzniknout při výrobě nebo v průběhu životního cyklu zařízení. Defekty typicky vznikají při výrobním procesu např. chybějícími kontakty nebo špatným pájením. Mimo výrobní proces stárnutím materiálů, degradací kontaktů, či vlivy prostředí — teplotou, vlhkostí, otřesy, radiací apod.[5][6]

Porucha (fault) modeluje defekt na logické úrovni. Defekt v zařízení může způsobit i více poruch. Porucha může vést k chybě.[5][6]

Chyba (error) zařízení se nechová tak, jak by se podle specifikace chovat mělo. Jde o důsledek poruchy, avšak porucha nemusí vždy způsobit chybu.[5][6]

Délka chyby — mějme chybu popsanou jako binární slovo. Délkou chyby se v této práci rozumí vzdálenost mezi jedničkovým bitem nejvyššího a nejnižšího řádu, který se v této chybě vyskytuje. Bude-li chyba zapsaná jako binární slovo mít tvar 00**1**1010**1**00, má tato chyba délku pět. Chyba

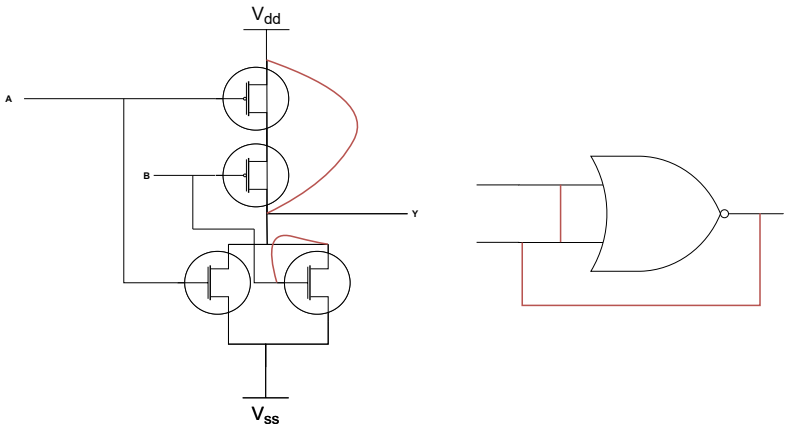
která ve svém binárním zápisu má jen jeden jedničkový bit bude v této práci chápána jako chyba délky nula.

3.1.1 Modely poruch

Za účelem testování zařízení se běžně používá několik různých modelů poruch.

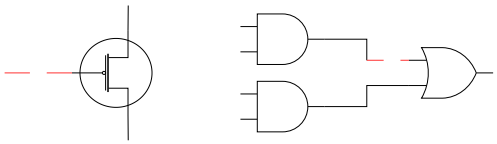
Stack-at model porucha je modelována připojením logických hodnot 0 nebo 1 na signál v designu zařízení. Podle připojené hodnoty mluvíme o stack-at 0 nebo o stack-at 1. Je dokázáno, že tento model pokrývá většinu defektů.[7] Přesto například v technologii CMOS existují defekty, které tento model neodhalí. Proto se v současnosti používají i pokročilejší modely.[6]

Zkrat (bridging fault) pokrývá defekty, které mohou vést k spojení (zkratu) různých částí obvodu. Tyto poruchy lze rozdělit na dvě skupiny. Pokud jde o zkrat mezi vstupy/výstupy hradel, pak hovoříme o zkratech vně hradel (inter-gate shorts). Zkrat mezi tranzistory se nazývá zkrat uvnitř hradla (intra-gate short). Na obrázku 3.1 jsou zobrazeny různé zkraty vně a uvnitř hradla NOR (červeně). Většina defektů tohoto modelu je pokryta stack-at fault modelem.[5]



Obrázek 3.1: Zkraty uvnitř a vně hradla NOR (červeně).

Rozpojení (opens) defekt způsobí rozpojení části obvodu na dvě části. Lze modelovat přerušením signálu uvnitř zařízení. Toto rozpojení se může objevit kdekoliv v hierarchii zařízení. Například na úrovni signálů mezi hradly nebo na úrovni tranzistorů přerušením signálu mezi tranzistory uvnitř logických hradel. Tyto dvě možnosti jsou zobrazeny na obrázku 3.2.[5]



Obrázek 3.2: Přerušené signály na úrovni tranzistorů a na úrovni hradel

Poruchy zpoždění (delay faults) jde o časové poruchy v obvodu. Zpoždění na výstupu hradla, na cestě uvnitř designu nebo změna signálu z log. 0 na 1, či naopak, trvá déle, než se očekává.[5]

3.2 Metody zvyšování spolehlivosti

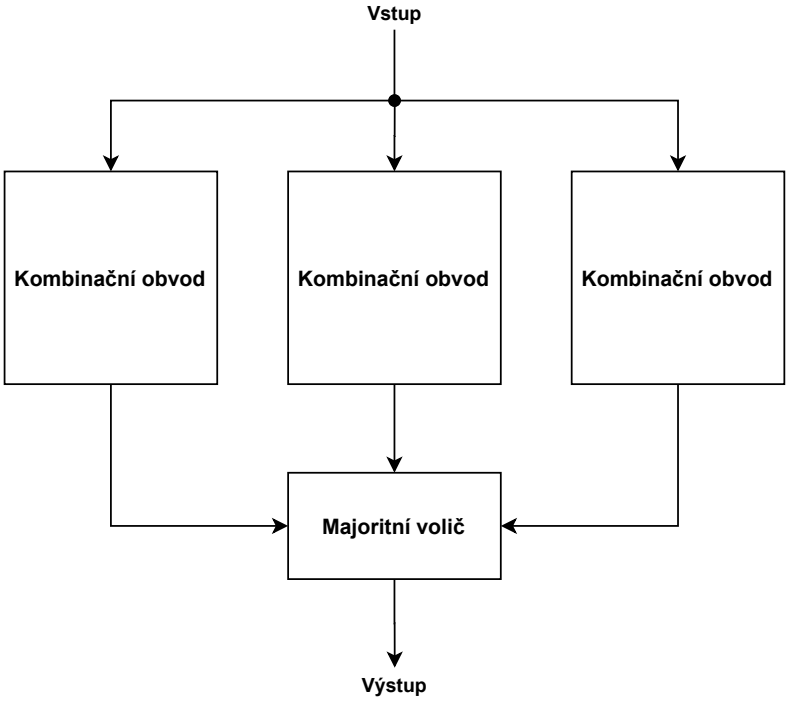
Pro zvýšení spolehlivosti proti poruchám je nutné použít redundantní prostředky.Pro zvýšení spolehlivosti se využívá redundance. Jde o přidání dodatečných zdrojů, času, či informace ke stávajícímu systému, jejichž použití by bylo zbytečné (redundantní), pokud by daný systém fungoval bezporuchově. Redundanci můžeme rozdělit na hardwarovou, programovou, časovou a informační. Redundantní prostředky lze využít pro detekci poruchy, její zamaskování, nebo pro zotavení po poruše. Detekování poruchy je klíčové, protože bez korektního detekování nastalé poruchy není možné provádět další kroky vedoucí k vypořádání se s následky této poruchy.[8]

3.2.1 Hardwarová redundance

Pokud do systému přidáme dodatečné zdroje nebo replikujeme ty stávající, hovoříme o hardwarové redundanci. Typickým příkladem je **TMR** (triple modular redundancy). Jde o systém schopný spolehlivě detekovat a zamaskovat jednu chybu na výstupu zálohované jednotky. V principu jde o použití tří identických bloků a majoritního voliče, který porovnává jejich výstup. Pokud jeden z bloků produkuje jiný výstup, než ostatní dva, je tento chybový výstup ignorován a zamaskován většinovou (majoritní) hodnotou výstupu ostatních bloků. Princip fungování TMR je znázorněn na obrázku 3.3. Takto je docíleno maskování jedné chyby s 200% přidanou plochou oproti původnímu návrhu.[8]

Zobecněním TMR získáme **NMR** (N-modular redundancy), kde je stejně jako u TMR porovnáván výstup N identických bloků majoritním voličem. N bývá typicky liché číslo, protože funkce majority není definovaná pro sudý počet členů. Čím větší je N, tím více chyb dokážeme zamaskovat, ovšem za cenu velkého nárůstu plochy navíc.

Pokud jsou použity dva identické bloky, mluvíme o **DMR** (double modular redundancy), kde je možné pomocí porovnávání výstupů detekovat výskyt chyby ale nelze provést maskování.



Obrázek 3.3: Princip fungování TMR.

3.2.2 Časová redundance

Časová redundance spočívá v opakování výpočtu. Pokud je chyba pouze dočasná, je možné ji tímto způsobem detekovat. Problém nastává u chyb trvalých, kde dostaneme pokaždé stejně chybný výsledek. Pro detekci této skutečnosti je možné výpočet opakovat v jiné části HW, případně se redundantní výpočet provede jiným způsobem. Výhodou je nulový nárůst plochy, avšak za cenu delšího výpočtu. Výpočet lze opakovat v SW i HW.

3.2.3 Informační redundance

Jde o přidání dodatečné informace k datům, která má umožnit v případě jejich poškození tuto skutečnost detekovat nebo i opravit. Za tímto účelem se používají bezpečnostní kódy.

3.3 Bezpečnostní kódy

Princip kódování spočívá v přiřazení slovu ze vstupní množiny (množina vzorů) slova z množiny výstupní (množina obrazů). V oblasti bezpečnostních kódů v HW zařízeních budeme slovem rozumět slovo binární, tedy slovo vyjádřené pomocí dvojkových bitů. Množinu obrazů budeme nazývat kódem a prvky

této množiny kódovými slovy. Kódování informace pomocí bezpečnostních kódů přidává ke vstupním datům dodatečnou informaci, díky které lze detekovat/opravovat chyby, ke kterým v těchto datech došlo. Bezpečností kódy lze rozdělit do dvou skupin. Na kódy **detekční** a **opravné**.[8] Před vyjmenováním základních bezpečnostních kódů je nutné definovat několik základních pojmů:

Hammingova váha je číslo udávající počet jedniček v binárním slově V. Značíme w(V).[8]

Hammingova vzdálenost je číslo, udávající v kolika bitech se 2 binární slova U a V stejné délky liší. Značíme se hd(U,V).[8]

Kódová vzdálenost je Hammingova vzdálenost mezi kódovými slovy.[8]

Minimální kódová vzdálenost pro kód K (značíme d_{min})je minimem z kódových vzdáleností, tedy min { hd(U,V) }, pro U,V $\in$ K, U $\neq$ V.[8]

Informační bit je v kódovém slově takový bit, který nese původní informaci kódovaného slova.[8]

Kontrolní bit je takový bit kódového slova, který byl k původní informaci přidán navíc, za účelem možnosti detekce/opravy chyb.[8]

Blokový (n, k) kód — nechť k udává počet informačních bitů v kódu, n celkový počet bitů kódového slova a n i k jsou konečná čísla. Pak kód s těmito vlastnostmi, tedy kód s n-bitovými kódovými slovy a k informačními bity v kódovém slově nazýváme blokový (n, k) kód.[8]

Redundance udává počet nadbytečných bytů v kódovém slově, sloužících k zajištění jeho detekčních/opravných vlastností. Značíme r, r = n - k.[9]

Systematický kód je takový kód, kde kódová slova jsou ve tvaru (a, f) — a je informační část a f jsou redundantní bity.[9]

Detekce/oprava chyb — Podle minimální kódové vzdálenosti můžeme určit, kolik chyb je daný kód schopný detekovat a kolik chyb je schopný opravovat. Pokud pro kód platí $d_{min} = 2t + 1$, pak kód může detekovat až 2t chyb, nebo t chyb opravovat. Pokud je $d_{min} = s + t + 1$, kde $s > t$, pak může opravit až t chyb a současně až s chyb detekovat.[8]

Generující matice kódu je matice o rozměrech k řádků $\times$ n sloupců. Obsahuje ve svých řádcích zapsané bazické vektory generující vektorový prostor kódových slov příslušného kódu. Značíme G.[8]

Kontrolní matice kódu je matice o rozměrech (n - k) řádků $\times$ n sloupců. Řádky této matice tvoří bázové vektory nulového vektorového prostoru příslušného kódu. Tedy všechny vektory z tohoto prostoru jsou ortogonální ke kódovým vektorům. Značíme H. Platí $H \times G^T = 0$.[8]

3.3.1 Přehled základních bezpečnostních kódů

Sudá parita je jedním z nejjednodušších možných kódů. Kódové slovo vznikne přidáním jednoho bitu ke vstupním datům. Tento redundantní bit má takovou hodnotu, aby počet všech jedničkových bitů ve výsledném kódovém slově byl sudý. Tedy například ke slovu 01101 by byl přidán paritní bit 1 a výsledné kódové slovo bude mít hodnotu 011011 Minimální kódová vzdálenost toho to kódu je 2. Pomocí sudé parity je možné detekovat libovolný **lichý** počet chyb.[9]

Příčná parita Vstupní slovo A vepíšeme do matice s rozměry $n_r \times n_c$, kde n_r je počet řádků a n_c počet sloupců této matice. Každý řádek této matice zabezpečíme sudou paritou (viz tabulka 3.1). Výsledných n_r paritních bitů spolu se vstupním slovem tvoří kód.[9]

Tabulka 3.1: Příklad zabezpečení 4-bitového slova A pomocí příčné parity.

A_0	A_1	P_0
A_2	A_3	P_1

Podélná parita Vstupní slovo A vepíšeme do matice s rozměry $n_r \times n_c$, kde n_r je počet řádků a n_c počet sloupců této matice. Každý sloupec této matice zabezpečíme sudou paritou (viz tabulka 3.2). Výsledných n_c paritních bitů spolu se vstupním slovem tvoří kód.[9]

Tabulka 3.2: Příklad zabezpečení 4-bitového slova A pomocí podélné parity.

A_0	A_1
A_2	A_3
P_0	P_1

Křížová parita je kombinací příčné a podélné parity. Vstupní slovo rovněž vepíšeme do matice s rozměry $n_r \times n_c$, kde n_r je počet řádků a n_c počet sloupců této matice. Každý řádek a sloupec matice zabezpečíme sudou paritou. Dalším paritním bitem můžeme pomocí sudé parity zabezpečit příčné nebo podélné paritní bity (viz tabulka 3.3, kde paritní bit p_4 je bit zabezpečující příčné nebo podélné paritní bity). Těchto $n_r + n_c$ paritních bitů (spolu s případným dalším bitem zabezpečujícím paritní příčné nebo podélné bity) se vstupním slovem tvoří kód. Minimální kódová vzdálenost tohoto kódu je 3, pokud příčné, či podélné paritní bity nezabezpečíme dodatečným paritním bitem. Pokud tak učiníme, minimální kódová vzdálenost bude 4.[9]

Bergerův kód je detekční kód pro detekci jednosměrných chyb. Jednosměrné chyby jsou takové, kde se mění pouze bity z 1 na 0, nebo z 0 na 1, ale nikdy oba případy současně. Při kódování se určí počet nulových bitů

Tabulka 3.3: Příklad zabezpečení 4-bitového slova A pomocí křížové parity.

A_0	A_1	P_0
A_2	A_3	P_1
P_2	P_3	P_4

v informačních bitech a kontrolní bity kódového slova jsou tvořeny tímto součtem zapsaným v binární soustavě. Pokud dojde k jednosměrným chybám, zvětší se nebo se zmenší počet nulových bitů v kódovém slově a přidaný kontrolní součet nesouhlasí.[10]

Dvoudrátová logika vytváří kódové slovo přidáním invertované hodnoty pro každý bit kódovaného slova. Např. pro slovo 0110 je tedy odpovídajícím kódovým slovem 0110|1001. Tento kód umožňuje detekovat všechny chyby v informační nebo kontrolní části kódového slova.

Hammingův kód umožňuje opravu jedné chyby nebo detekci dvou chyb. Jeho minimální kódová vzdálenost je 3. Generující matice systematického kódu vznikne z jednotkové matice $i \times i$, kde i je délka vstupního slova a z matice pro paritní bity. Tu získáme zapsáním i vzájemně různých řádků, kde každý řádek obsahuje alespoň dva jedničkové bity o minimální potřebné šířce. Např. pro vstupní slovo o 4 bitech potřebujeme pro paritní bity matici o 4 řádcích. Tyto řádky, zapsané dekadicky budou mít hodnoty 3, 5, 6 a 7, protože každé z těchto čísel v binárním zápisu obsahuje minimálně dva jedničkové bity. Protože 7 lze binárně zapsat jako 111, potřebná šířka bude 3, neboť pro menší počet sloupců paritní matici s těmito popsanými vlastnostmi nelze sestrojit. Matice pro paritní bity tedy bude mít rozměry 4 řádky a 3 sloupce. Generující matice bude ve tvaru:

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

kde levá část je jednotková matice 4×4 a pravá část je matice pro parity 4×3. Kód pro 4-bitová vstupní slova je tedy (n, k) = (7, 4). Kódování probíhá vynásobením vstupního slova V maticí G: $U = V \cdot G$. Dekódování probíhá pomocí kontrolní matice H. Ta pro Hammingův kód vzniká zapsáním n (n-k)-bitových nenulových a vzájemně různých sloupců. Uvedené matici G odpovídá následující matice H:

$$H = \left(\begin{array}{cccc|ccc} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{array}\right)$$

kde levá část odpovídá transponované paritní matici z matice G a pravá část je jednotková matice příslušné velikosti. Dekódování probíhá následujícím způsobem. Slovo určené k dekódování označme U. Vypočítáme tzv. syndrom S: $S = H \cdot U^T$. Pokud S = 0, pak nedošlo k žádné chybě, nebo chybu nelze detekovat. Pokud S $\neq$ 0, a S odpovídá hodnotě i-tého sloupce v kontrolní matici, pak je chyba v i-tém bitu kódového slova U a lze jí opravit znegováním tohoto bitu.[8]

Rozšířený hammingův kód vznikne rozšířením kódových slov Hammingova kódu o další paritní bit. Ten volíme tak, aby počet jedničkových bitů v každém kódovém slově byl sudý, tedy kódová slova zabezpečíme sudou paritou. Získáme tak kód s minimální kódovou vzdáleností 4. Pokud použijeme generující matici pro 4-bitová vstupní slova, uvedenou u Hammingova kódu, lze rozšíření provést doplněním dalšího sloupce tak, aby počet jedničkových bitů na každém řádku matice G byl sudý.[8]

$$G = \left(\begin{array}{cccc|ccc|c} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \end{array}\right)$$

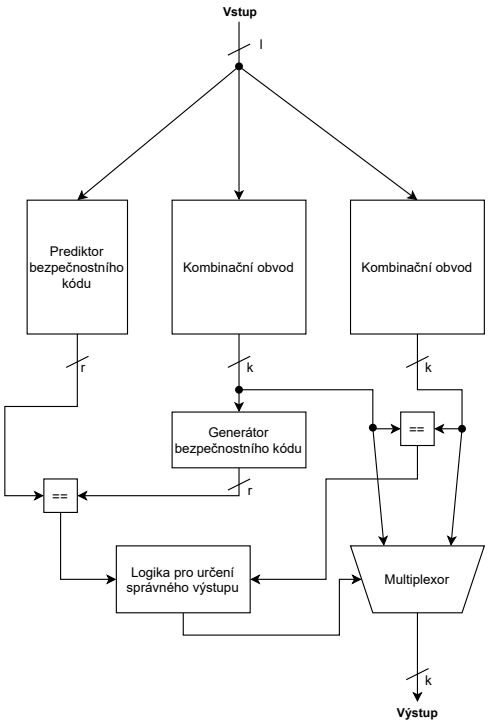
Součinové kódy vznikají složením dvou kódů. Mějme kódy $K_1 = (n_1, k_1)$ a $K_2 = (n_2, k_2)$. Součinem těchto kódů vznikne kód K = (n, k), kde n = $n_1 \times n_2$ a k = $k_1 \times k_2$. Pro minimální kódovou vzdálenost tohoto kódu platí, že se rovná součinu minimálních kódových vzdáleností kódů K_1 a K_2. Kódování vstupního k-bitového slova a = $(a_0, a_1, \ldots, a_{k-1})$ probíhá následovně:

1. Slovo a vepíšeme do matice o rozměrech k_2 řádků $\times k_1$ sloupců.
2. Každý řádek matice nahradíme kódovým slovem zakódovaným pomocí kódu K_1. Získáme matici o rozměrech k_2 řádků $\times n_1$ sloupců.
3. Každý sloupec této matice zakódujeme použitím kódu K_2. Získáváme výslednou matici o rozměrech n_2 řádků $\times n_1$ sloupců.
4. Kódové slovo vznikne popořadě vypsáním řádků výsledné matice.[9]

3.4 Návrh architektury

Jako základ pro zabezpečení kombinační logiky v FPGA jsem se rozhodl použít DMR. Zdvojením logiky je možné detekovat chyby na výstupech jedné z jednotek. Je zde tedy předpoklad, že porucha, která se projeví chybným výstupem na kombinační logice, postihne vždy jen jednu ze zdvojených jednotek. Aby architektura dokázala tyto chyby na výstupu i opravovat, využiji architekturu s prediktorem bezpečnostních kódů,

která byla popsána v [2]. Spojením DMR a prediktoru vznikne architektura schopná v případě nesouhlasu výstupů z DMR určit, který výstup je chybový, a zvolit správný. Navržená architektura je zobrazena na obrázku 3.4. Vstup do architektury je l-bitový. Každá z jednotek kombinační logiky produkuje k-bitový výstup. Prediktor bezpečnostního kódu a generátor bezpečnostního kódu generují pouze redundantní bity zvoleného (n, k) kódu. Tedy jejich výstup je r-bitový, kde $r = n - k$. Architektura dále obsahuje dva komparátory (==). Jeden porovnává na shodu výstupy kombinačních logik a druhý redundantní bity kódu z generátoru a prediktoru. Logika pro určení správného výstupu na základě výstupu těchto komparátorů ovládá multiplexor, který vybírá výstup z první nebo druhé jednotky kombinační logiky.



Obrázek 3.4: Navržená architektura pro zabezpečení kombinační logiky.

Vstup je předán dvěma identickým jednotkám kombinační logiky (DMR) a prediktoru bezpečnostního kódu. Pokud nenastane žádná porucha, výstupy obou jednotek jsou bez chyb a totožné. Komparátor jejich výstupu (==) detekuje shodu a logika pro určení správného výstupu zvolí výstup z pravé jednotky. (Samozřejmě lze zvolit i výstup z levé jednotky, bez chyb jsou v tomto případě výstupy obou.) Nyní uvažme situaci, kdy nastane porucha a výstup jedné z jednotek se vlivem chyby liší. Komparátor výstupů jednotek detekuje různé hodnoty. Aby logika pro určení

správného výstupu mohla určit bezchybný výstup, porovnají se hodnoty predikovaných a skutečných hodnot redundantních bitů kódu levé kombinační logiky. Pokud je zvolen kód schopný detekovat všechny chyby, které se mohou na výstupu kombinační logiky objevit, mohou nastat tyto dvě situace:

1. **Chyba se projevila na výstupu pravé jednotky kombinační logiky.** V takovém případě budou predikované a skutečné hodnoty redundantních bitů levé jednotky totožné a logika pro určení správného výstupu zvolí výstup levé jednotky jako bezchybný.
2. **Chyba se projevila na výstupu levé jednotky kombinační logiky.** Tedy redundantní bity z prediktoru a generátoru kódu se budou lišit a logika pro určení správného výstupu zvolí výstup z pravé jednotky.

Pokud zvolený kód není schopen detekovat všechny chyby (minimální kódová vzdálenost není dostatečná) a nastane chyba 2, určitá část chyb nebude opravena. Kvůli nedostatečné minimální kódové vzdálenosti se totiž může stát, že více různých chyb bude mít stejné redundantní paritní bity jako současná správná hodnota výstupu, a proto chyba může být považována za korektní výstup. Proto je nutné volit takový kód, který všechny možné chyby na výstupu dokáže pokrýt.

Chyba by se mohla projevit ještě na dalších místech:

- Porucha v prediktoru nebo generátoru bezpečnostního kódu způsobí chybný výstup tohoto obvodu. Protože ale předpokládáme výskyt poruchy vždy jen v jednom funkčním bloku, oba kombinační obvody budou mít správný a totožný výstup a chybou způsobené rozdílné hodnoty redundantních bitů se nebudou brát v úvahu. Naprosto stejně bude v tomto případě ignorována špatná funkce komparátoru redundantních bitů, pokud by poruchy nastala v tomto komparátoru.
- V logice pro určení správného výstupu, multiplexoru nebo komparátoru výstupů kombinačních obvodů se objeví porucha. Chybné výstupy těchto obvodů mohou způsobit špatný výstup architektury. Pro dostatečně velké zabezpečované obvody však bude velikost těchto obvodů vůči zabezpečovaným zanedbatelná a tudíž pravděpodobnost výskytu poruchy v těchto obvodech bude rovněž nižší. Pokud by bylo potřeba zcela vyeliminovat možnost selhání architektury, bude nutné zabezpečit proti selhání tyto obvody.

3.4.1 Volba bezpečnostních kódu pro zabezpečení

Pro detekci chyb jsem se rozhodl zvolit jednodušší kódy a jejich kombinace pomocí součinů kódů. Přesněji kódy:

- sudá parita,
- křížová parita,
- Hammingův kód,
- rozšířený Hammingův kód,
- součiny výše uvedených kódů.

Uvedené kódy mají minimální kódové vzdálenosti mezi 2 a 4, tedy dokáží detekovat 1 až 3 chyby. Protože pro minimální kódovou vzdálenost součinového kódu platí, že je součinem minimálních kódových vzdáleností kombinovaných kódů, je možné kombinací uvedených kódů dosáhnout až minimální kódové vzdálenosti 16, tedy detekovat až 15 chyb.

Chyby na výstupu zabezpečovaných logických kombinačních obvodů nemusí postihnout všechny výstupy. Proto je u každého obvodu nutné určit, jaký je maximální počet chyb na výstupu a zvolit bezpečnostní kód k detekci chyb tak, aby minimální kódová vzdálenost byla dostatečná k odhalení všech chyb a zároveň hardwarová redundance (plocha navíc) přidaná kódem byla co nejmenší. K určení maximálního počtu chyb na výstupech budu využívat simulačního softwaru dostupného na KČN FIT ČVUT[11]. Ten pomocí emulování obvodu v FPGA a injekcí poruch do tohoto emulovaného obvodu dokáže určit projevy těchto poruch na výstupech.

Kapitola 4

Realizace

V této části je popsaná hardwarová realizace jednotlivých součástí navržené architektury ve VHDL a skripty pro jejich automatické generování. Je zde diskutována hardwarová redundance jednotlivých komponent architektury. Pomocí simulačního softwaru jsou analyzovány obvody s ohledem na jejich vlastnosti se simulovanými poruchami, je vytvořena jejich klasifikace popisující možnosti opravy chybných výstupů a jsou navrženy úpravy simulačního SW pro výběr optimálního kódu pro pokrytí chyb.

4.1 Generátory kódů

Pro navrženou architekturu je zapotřebí vytvořit generátory kódů, které budou generovat kódová slova. Pro všechny zvolené kódy — sudá parita, křížová parita, Hammingův kód, rozšířený Hammingův kód a jejich kombinace pomocí součinu kódů — postačí, aby generátory generovaly redundantní bity těchto kódů, neboť navržená architektura pracuje právě s nimi. Všechny generátory kódů mají následující rozhraní VHDL entity, liší se jen název entity a šířky vstupního a výstupního portu:

```
ENTITY generator IS
  PORT (
    i  :  IN  std_logic_vector(7 DOWNTO 0);
    p  :  OUT std_logic_vector(3 DOWNTO 0)
  );
END ENTITY generator;
```

Sudá parita — Generování paritního bitu pro sudou paritu je velice jednoduché. Postačí provést XOR všech vstupních signálů a výsledkem této operace je požadovaný redundantní bit kódu sudá parita. Ve VHDL je typ výstupního signálu p pouze std_logic, neboť je pouze jednobitový.

Příklad výpočtu tohoto paritního bytu pro 4-bitový vstup ve VHDL je následující:

```
p <= i(3) XOR i(2) XOR i(1) XOR i(0);
```

Křížová parita — V podstatě jde pouze o rozšířenou aplikaci sudé parity. Vstupy se napřed vepíší do matice o rozměrech m × n. Každý řádek a sloupec je potom zabezpečen sudou paritou. Konečně jsou sudou paritou zabezpečeny také sloupcové nebo řádkové parity. Každý redundantní bit je tedy vytvořen XORem několika bitů vstupu, případně jiných paritních bitů, podle zvolené matice. Pokud existuje více řešení, jak tuto matici sestavit, volil jsem takové, kde rozdíl m a n bude co nejmenší možný. A tedy sloupcové a řádkové parity budou pokrývat co nejpodobnější počet bitů. Například 12 vstupů lze uspořádat do matice 12 × 1, 6 × 2, nebo 4 × 3. Nejmenší rozdíl je mezi 4 a 3, proto bude mít použitá matice tyto rozměry. Pokud má vstup prvočíselný počet bitů, nedá se zapsat jinak než do matice s jedním rozměrem rovným 1. Abych se tomu vyhnul, přidávám v tomto případě ke vstupu jeden nulový bit navíc, jak je ukázáno v tabulce 4.1 pro 5 vstupních bitů. Protože platí $0 \oplus 1 = 1$ a $0 \oplus 0 = 0$ je nulový bit k operaci XOR neutrálním prvkem a při sestavování řádkových a sloupcových parit lze tento přidaný nulový bit ignorovat. Sloupcová parita pro první sloupec a řádková parita pro první řádek bude mít tedy ve svém výpočtu o jeden XOR méně než ostatní sloupcové a řádkové paritní bity. Ve VHDL se musí vytvořit pomocný signál p_i o šířce výstupních paritních bitů, protože standard VHDL93 neumožňuje čtení z výstupních portů a pro výpočet parity parit (bit P_5 v tab 4.1) je nutné číst hodnoty některých již vypočtených paritních bitů. Následně se tento signál p_i musí přiřadit do výstupního portu p. Paritní bity pro příklad v tab. 4.1 jsou ve VHDL implementované následujícím způsobem:

```
p_i(0) <= i(4) XOR i(3);
p_i(1) <= i(2) XOR i(1) XOR i(0);
p_i(2) <= i(2);
p_i(3) <= i(4) XOR i(1);
p_i(4) <= i(3) XOR i(0);
p_i(5) <= p_i(2) XOR p_i(3) XOR p_i(4);
p <= p_i;
```

Tabulka 4.1: Příklad doplnění vstupu o prvočíselné šířce o nulový bit.

0	I_4	I_3	P_0
I_2	I_1	I_0	P_1
P_2	P_3	P_4	P_5

Hammingův kód, rozšířený Hammingův kód — Generování Hammingových kódů spočívá v sestavení XORů pro redundantní paritní bity podle tvaru generující matice. Vezměme si jako příklad kód (7, 4) s generující maticí ve tvaru:

$$G = \left(\begin{array}{cccc|ccc} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

Vidíme, že jde o systematický kód. Protože generátory mají generovat pouze redundantní paritní bity, nemusíme levou část generující matice obsahující jednotkovou matici uvažovat. Zajímat nás budou tedy jen poslední tři sloupce popisující paritní bity. Nechť první z těchto sloupců popisuje paritní bit nula, druhý paritní bit jedna a třetí paritní bit dva. Pro každý z těchto sloupců sestavíme rovnici pro příslušný paritní bit jako XOR určitých vstupních bitů. Očíslujeme-li si řádky shora dolů od nuly do tří, pak každý řádek s hodnotou jedna v příslušném paritním sloupci udává, že vstupní bit s číslem totožným jako je číslo řádku bude součástí XORu pro výpočet paritního bitu. První paritní sloupec obsahuje jedničky na prvním, druhém a třetím řádku. Tedy rovnice pro paritní bit nula bude ve tvaru: $P_0 = I_1 \oplus I_2 \oplus I_3$ Pro další paritní bity postupujeme analogicky. Ve VHDL budou paritní bity pro tento kód popsány takto:

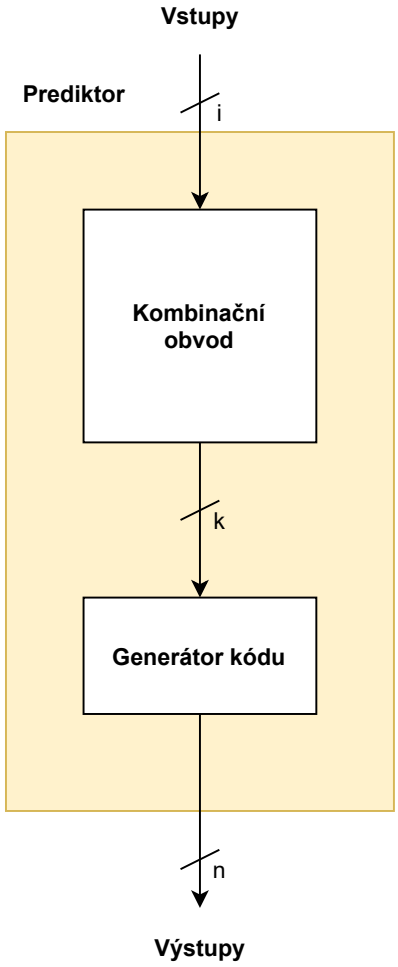
```
p(0) <= i(1) XOR i(2) XOR i(3);
p(1) <= i(0) XOR i(2) XOR i(3);
p(2) <= i(0) XOR i(1) XOR i(3);
```

Součiny kódů — Pro vytvoření součinu kódů je nutné uspořádat vstup do matice. To je provedeno podle stejných principů jako u vytváření matice pro křížovou paritu. Pro řádkový a sloupcový kód je nutné vygenerovat generátory požadovaných kódů o příslušné šířce podle velikosti sloupců a řádků této matice. Následně jsou řádky a sloupce matice zabezpečeny těmito kódy — instancemi jejich komponent, které jsou připojeny k příslušným signálům odpovídajícím sloupcům a řádkům matice. Nakonec jsou výstupní paritní bity uspořádány na výstup tak, aby odpovídaly pořadí v řádcích matice směrem shora dolů.

4.2 Prediktory kódů

Prediktory kódů slouží k predikování hodnoty redundantních bitů zvoleného kódu ze zadaných vstupů obvodu. Vznikne spojením zabezpečované jednotky kombinační logiky a generátoru kódu. Výstupy jednotky kombinační logiky jsou přímo připojeny na generátor kódu a nejsou vyvedeny ven z prediktoru.

Struktura prediktoru je zobrazena na obrázku 4.1 a je stejná pro všechny použité kódy. Jako generátor kódu se vždy použije generátor požadovaného kódu.



Obrázek 4.1: Struktura prediktoru kódu.

4.3 Kompletní architektura

Celková architektura vznikne spojením VHDL souborů reprezentujících zabezpečovaný kombinační obvod, generátor kódu a prediktor podle návrhu na obrázku 3.4. Pro syntézu, ověřování velikostí jednotlivých komponent a testování byl použit software Vivado 2019.1. Syntézu byla prováděna pro desky Basys 3, s nastavením strategie syntézy pro optimalizaci plochy — „Flow_AreaOptimized_high“. Je nutné se vypořádat s tím, že Vivado předpokládá bezporuchový stav navrhovaného obvodu a provádí optimalizace. Pokud se použije

např. DMR, tedy 2 identické jednotky a detekce chyby na výstupu — typicky XOR výstupů obou jednotek, Vivado detekuje na vstupu XORů logiku produkující vždy identické výstupy a celý XOR nahradí logickou nulou. Aby se tomuto zabránilo, je nutné použít ve VHDL kódu atribut, který tomuto chování zabrání:

```
ATTRIBUTE keep_hierarchy : string;
ATTRIBUTE keep_hierarchy OF rtl : ARCHITECTURE IS "YES";
```

Tento atribut je nutné definovat v architektuře zabezpečované logiky (protože zabezpečovaná logika bude použita v DMR) a také v architektuře prediktoru (kvůli dvojímu použití generátoru kódu — v prediktoru a pro výpočet redundantních bitů pro jednu z jednotek.)

4.4 Analýza vlastností různých typů kombinačních obvodů

Pro analýzu vlastností různých kombinačních obvodů byl použit simulační SW dostupný na KČN[11]. Byl aplikován na vybrané obvody benchmarkových sad LGSynth'91 (MCNC)[12], ISCAS'89[13] a sčítaček[14]. Byl sledován vliv poruch na výskyt chyb na výstupech obvodů a vlastnosti těchto chyb, konkrétně počet chybných bitů a počty chyb různých délek. Z těchto dat byly následně vypočítány podíly různých délek chyb a podíly různého počtu chyb na výstupech. V následujících tabulkách jsou zobrazena získaná data pro vybrané benchmarkové obvody. Tabulka 4.2 znázorňuje počty chybných bitů na výstupech benchmarků, objevených během simulace vlivu poruch uvnitř testovaných benchmarků. V tabulce 4.3 jsou zobrazeny počty chyb různých délek[1] a tabulky 4.4 a 4.5 udávají podíl jednobitových, dvoubitových a vícebitových chyb ku všem chybám na výstupu, respektive podíl chyb s délkou nula, jedna, dvě a více ku všem délkám objevených chyb. Sloupec poruchy zobrazuje počet všech detekovaných poruch, které jakkoliv ovlivnily výstup.

V tabulkách 4.4 a 4.5 můžeme pozorovat, že v chybách objevených simulací převažují chyby spíše jen několika bitů a současně menších délek. Větší chyby se také vyskytují, ale jejich počet je znatelně nižší. U některých obvodů chyby nepostihují všechny výstupy, ale jejich velikost je omezena konstantou, která je menší než počet výstupů daného obvodu. Toho lze využít při volbě kódu pro zabezpečení navrženou architekturou, který může být menší než kdyby zabezpečoval všechny bity výstupu.

Ukazuje se, že velkým problémem jsou přenosy v aritmetických obvodech, které dokáží při jistých hodnotách na vstupech propagovat vliv poruchy do vyšších řádů a zasáhnout větší počet výstupů. Například na obvodech sčítaček můžeme pozorovat, že chyby zasahují v podstatě všechny řády na výstupu,

[1] Sloupec udávající počty chyb s délkou nula je totožný se sloupcem popisujícím počty jednobitových chyb v tabulce 4.2, a proto byl vynechán.

Tabulka 4.2: Počty chybných bitů na výstupech vybraných benchmarkových obvodů.

Benchmarkový obvod				Počet chybných bitů na výstupu							
Název	**Poruchy**	**Vstupy**	**Výstupy**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
03_adder	251	7	4	2835	495	0	0	-	-	-	-
04_adder	355	9	5	8814	1495	937	0	0	-	-	-
05_adder	655	11	6	39170	15117	9911	6294	2130	493	-	-
06_adder	586	13	7	74422	15780	13743	5416	4274	763	0	-
07_adder	1062	15	8	270556	121890	100457	67744	49414	35493	21623	4736
root	1406	8	5	11671	2221	351	53	110	-	-	-
sao2	2081	10	4	12326	836	235	34	-	-	-	-
sqrt8	770	8	4	5941	605	225	0	-	-	-	-
sqrt8ml	364	8	5	3690	957	0	0	-	-	-	-
squar5	503	5	8	2244	298	240	46	56	0	11	0
alu1	281	12	8	71528	1419	318	17	0	0	0	0
alu2	2589	10	6	40977	2385	347	44	0	0	-	-
alu3	977	10	8	57943	2554	133	98	11	0	0	0
alu4	16236	14	8	1159894	199338	36785	9977	2331	2085	0	0
bbtas	206	5	5	771	6	2	2	0	-	-	-
beecount	603	6	7	3045	142	103	44	0	0	0	-
dist	1899	8	5	16840	2164	904	486	79	-	-	-
dk14	896	6	8	4859	218	307	136	67	11	11	0
dk15	503	5	7	2382	37	0	0	0	0	0	0
dk512	371	5	7	1485	58	21	0	0	0	0	0

Tabulka 4.3: Počty odhalených chyb různých délek na výstupu vybraných benchmarkových obvodů.

Benchmarkový obvod				Délka chyby na výstupu						
Název	**Poruchy**	**Vstupy**	**Výstupy**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
03_adder	251	7	4	495	0	0	-	-	-	-
04_adder	355	9	5	1495	937	0	0	-	-	-
05_adder	655	11	6	15117	9910	6295	2130	493	-	-
06_adder	586	13	7	15145	14378	5416	4274	763	0	-
07_adder	1062	15	8	121890	100457	67841	49414	35570	21623	4736
root	1406	8	5	2016	488	308	110	-	-	-
sao2	2081	10	4	531	362	212	-	-	-	-
sqrt8	770	8	4	509	454	198	-	-	-	-
sqrt8ml	364	8	4	616	341	0	-	-	-	-
squar5	503	5	8	80	132	104	81	43	121	90
alu1	281	12	8	1139	860	272	0	0	0	0
alu2	2589	10	6	944	55	1590	328	170	-	-
alu3	977	10	8	294	1118	948	205	71	160	0
alu4	16236	14	8	36143	11135	10637	36856	142266	14759	1716
bbtas	206	5	5	2	4	4	0	-	-	-
beecount	603	6	7	139	7	67	18	46	12	-
dist	1899	8	5	1608	782	634	659	-	-	-
dk14	896	6	8	30	187	83	79	182	167	22
dk15	503	5	7	32	5	0	0	0	0	-
dk512	371	5	7	31	35	13	0	0	0	-

i když počet chyb postihujících celý výstup sčítačky je nízký. Naopak například u obvodu pro odmocniny, který nejspíše přenos pro výpočet nepoužívá můžeme vidět, že žádná chyba nepostihuje celou šířku výstupů. Zabezpečení aritmetických obvodů s vnitřními přenosy tedy bude složitější, protože bude

Tabulka 4.4: Podíl jednobitových, dvoubitových a vícebitových chyb na výstupech vybraných benchmarkových obvodů.

Název	**Podíl n-bitových chyb v %**		
	n = 1	**n = 2**	**n >2**
03_adder	85,14	14,86	0,00
04_adder	78,37	13,29	8,33
05_adder	53,57	20,68	25,75
06_adder	65,06	13,79	21,15
07_adder	40,27	18,14	41,59
root	81,01	15,42	3,57
sao2	91,77	6,22	2,00
sqrt8	87,74	8,94	3,32
sqrt8ml	79,41	20,59	0,00
squar5	77,51	10,29	12,19
alu1	97,61	1,94	0,46
alu2	93,66	5,45	0,89
alu3	95,40	4,20	0,40
alu4	82,24	14,13	3,63
bbtas	98,72	0,77	0,51
beecount	91,33	4,26	4,41
dist	82,25	10,57	7,18
dk14	86,63	3,89	9,48
dk15	98,47	1,53	0,00
dk512	94,95	3,71	1,34

Tabulka 4.5: Podíl chyb o délce nula, jedna, dvě a více na výstupech vybraných benchmarkových obvodů.

Název	**Podíl chyb o délce n v %**			
	n = 0	**n = 1**	**n = 2**	**n >2**
03_adder	85,14	14,86	0,00	0,00
04_adder	78,37	13,29	8,33	0,00
05_adder	53,57	20,68	13,55	12,20
06_adder	65,06	13,24	12,57	9,14
07_adder	40,26	18,14	14,95	26,66
root	79,98	13,81	3,34	2,86
sao2	91,77	3,95	2,70	1,58
sqrt8	83,65	7,17	6,39	2,79
sqrt8ml	79,41	13,26	7,34	0,00
squar5	77,51	2,76	4,56	15,16
alu1	96,92	1,54	1,17	0,37
alu2	92,99	2,14	0,12	4,74
alu3	95,40	0,48	1,84	2,28
alu4	82,06	2,56	0,79	14,59
bbtas	98,72	0,26	0,51	0,51
beecount	91,33	4,17	0,21	4,29
dist	82,05	7,84	3,81	6,30
dk14	86,63	0,53	3,33	9,50
dk15	98,47	1,32	0,21	0,00
dk512	94,95	1,98	2,24	0,83

potřeba volit větší kódy, aby byly pokryté všechny možné chyby na výstupech.

Analýze byly podrobeny i některé vlastní obvody. Například obvody provádějící inkrementaci/dekrementaci vstupu o určitou konstantu. Výsledky pro výběr z těchto obvodů jsou v následujících tabulkách. Obvody jsou pojmeno-

vané ve formátu incX_Y, respektive decX_Y, kde inc značí, že jde o přičítající obvod, dec je obvod odečítající, X je šířka vstupu a Y konstanta, která je přičítána nebo odčítána. Tabulky jsou ve stejném formátu jako pro benchmarkové obvody. V tabulkách 4.6 a 4.7 jsou zobrazeny počty detekovaných chybných bitů na výstupech a počty chyb o určitých délkách. V tabulkách 4.8 a 4.9 jsou vypočítány poměry n-bitových chyb ku všem chybám a poměry chyb o určité délce ku všem délkám.

Tyto obvody jsou aritmetické s vnitřním přenosem. Ukazuje se ovšem, že i volená konstanta pro přičítání/odčítání má vliv na chyby na výstupu. Pokud převedeme konstanty do binární podoby, z tabulek je zřejmé, že menší vliv na chyby na výstupech mají konstanty s nulami v nižších řádech a jedničkami v co nejvyšších řádech. Patrné je to například na obvodu který od osmibitových čísel odčítá konstanty 15 a 16. Zatímco pro konstantu 15 jedničkové bity v nižších řádech způsobují velký počet chyb na výstupu, konstanta 16 způsobuje nejvýše chyby o délce 2. Tedy pro zabezpečení tohoto obvodu postačí kód o minimální kódové vzdálenosti 4. Stejně jako u testovaných benchmarkových obvodů se ukazuje, že většina chyb je menších a chyb, které postihují větší počet bitů výstupu je méně.

Tabulka 4.6: Počty chybných bitů na výstupech vlastních testovacích obvodů.

Obvod				**Počet chybných bitů na výstupu**							
Název	**Poruchy**	**Vstupy**	**Výstupy**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
dec2_1	31	2	2	58	0	-	-	-	-	-	-
dec2_2	16	2	2	32	0	-	-	-	-	-	-
inc2_1	29	2	2	54	0	-	-	-	-	-	-
inc2_2	17	2	2	34	0	-	-	-	-	-	-
dec4_1	131	4	4	683	24	4	4	-	-	-	-
dec4_3	98	4	4	484	0	0	0	-	-	-	-
dec4_4	40	4	4	272	32	0	0	-	-	-	-
dec4_7	101	4	4	494	0	0	0	-	-	-	-
dec4_8	12	4	4	96	0	0	0	-	-	-	-
inc4_1	104	4	4	508	0	0	0	-	-	-	-
inc4_3	126	4	4	680	28	10	0	-	-	-	-
inc4_4	40	4	4	272	32	0	0	-	-	-	-
inc4_7	137	4	4	720	24	4	4	-	-	-	-
inc4_8	11	4	4	88	0	0	0	-	-	-	-
dec8_1	447	8	8	17397	927	13	6	0	0	0	0
dec8_15	463	8	8	19693	1411	242	84	40	20	12	0
dec8_16	105	8	8	7776	512	384	0	0	0	0	0
inc8_1	480	8	8	17204	718	146	78	28	18	10	0
inc8_15	479	8	8	17540	1660	400	111	55	28	14	14
inc8_16	120	8	8	9024	640	288	128	0	0	0	0

Tabulka 4.7: Počty odhalených chyb různých délek na výstupu vlastních testovacích obvodů.

Benchmarkový obvod				**Délka chyby na výstupu**						
Název	**Poruchy**	**Vstupy**	**Výstupy**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
dec2_1	31	2	2	0	-	-	-	-	-	-
dec2_2	16	2	2	0	-	-	-	-	-	-
inc2_1	29	2	2	0	-	-	-	-	-	-
inc2_2	17	2	2	0	-	-	-	-	-	-
dec4_1	131	4	4	24	4	4	-	-	-	-
dec4_3	98	4	4	0	0	0	-	-	-	-
dec4_4	40	4	4	32	0	0	-	-	-	-
dec4_7	101	4	4	0	0	0	-	-	-	-
dec4_8	12	4	4	0	0	0	-	-	-	-
inc4_1	104	4	4	0	0	0	-	-	-	-
inc4_3	126	4	4	28	10	0	-	-	-	-
inc4_4	40	4	4	32	0	0	-	-	-	-
inc4_7	137	4	4	24	4	4	-	-	-	-
inc4_8	11	4	4	0	0	0	-	-	-	-
dec8_1	447	8	8	635	7	292	0	6	6	0
dec8_15	463	8	8	933	614	166	52	32	12	0
dec8_16	105	8	8	512	384	0	0	0	0	0
inc8_1	480	8	8	443	168	309	50	18	10	0
inc8_15	479	8	8	1436	400	335	55	28	14	14
inc8_16	120	8	8	640	288	128	0	0	0	0

Tabulka 4.8: Podíl jednobitových, dvoubitových a vícebitových chyb na výstupech vlastních testovacích obvodů.

Název	**Podíl n-bitových chyb v %**		
	n = 1	**n = 2**	**n >2**
dec2_1	100,00	0,00	0,00
dec2_2	100,00	0,00	0,00
inc2_1	100,00	0,00	0,00
inc2_2	100,00	0,00	0,00
dec4_1	95,52	3,36	1,12
dec4_3	100,00	0,00	0,00
dec4_4	89,47	10,53	0,00
dec4_7	100,00	0,00	0,00
dec4_8	100,00	0,00	0,00
inc4_1	100,00	0,00	0,00
inc4_3	94,71	3,90	1,39
inc4_4	89,47	10,53	0,00
inc4_7	95,74	3,19	1,06
inc4_8	100,00	0,00	0,00
dec8_1	94,84	5,05	0,10
dec8_15	91,59	6,56	1,85
dec8_16	89,67	5,90	4,43
inc8_1	94,52	3,94	1,54
inc8_15	88,49	8,37	3,14
inc8_16	89,52	6,35	4,13

Tabulka 4.9: Podíl chyb o délce nula, jedna, dvě a více na výstupech vlastních testovacích obvodů.

Název	**Podíl chyb o délce n v %**			
	n = 0	**n = 1**	**n = 2**	**n >2**
dec2_1	100,00	0,00	0,00	0,00
dec2_2	100,00	0,00	0,00	0,00
inc2_1	100,00	0,00	0,00	0,00
inc2_2	100,00	0,00	0,00	0,00
dec4_1	95,52	3,36	0,56	0,56
dec4_3	100,00	0,00	0,00	0,00
dec4_4	89,47	10,53	0,00	0,00
dec4_7	100,00	0,00	0,00	0,00
dec4_8	100,00	0,00	0,00	0,00
inc4_1	100,00	0,00	0,00	0,00
inc4_3	94,71	3,90	1,39	0,00
inc4_4	89,47	10,53	0,00	0,00
inc4_7	95,74	3,19	0,53	0,53
inc4_8	100,00	0,00	0,00	0,00
dec8_1	94,84	3,46	0,04	1,66
dec8_15	91,59	4,34	2,86	1,22
dec8_16	89,67	5,90	4,43	0,00
inc8_1	94,52	2,43	0,92	2,13
inc8_15	88,49	7,24	2,02	2,25
inc8_16	89,52	6,35	2,86	1,27

4.5 Výsledky

Pro testování navržené architektury byly použity následující obvody:

ALU — 8-bitová ALU jednotka[2] provádějící následující operace:

- sčítání,
- odčítání,
- násobení,
- dělení,
- logický posun vlevo a vpravo o jednu pozici,
- rotaci vlevo a vpravo o jednu pozici,
- logický AND,
- logický OR,
- logický XOR,
- logický NOR,
- logický XNOR,
- logický NAND,
- porovnání operandů (A >B),
- rovnost operandů.

Barrel shifter — 8-bitový barrel shifter provádějící pomocí nastavitelných posuvů o jeden, dva a čtyři bity logický posuv vstupu vlevo a vpravo o nula až sedm bitů.

JTAG FSM — Kombinační logika pro další stav JTAG TAP kontroléru, jak je popsaný například zde: `https://www.xjtag.com/about-jtag/jtag-a-technical-overview/` a kombinační logika pro výstup indikující, že se kontrolér nachází v určitých stavech, konkrétně ve stavech:

- TL_RESET,
- CAPTURE_DR,
- SHIFT_DR,
- PAUSE_DR,
- UPDATE_DR,
- CAPTURE_DR,
- SHIFT_IR,

[2]Kód ALU byl převzat z `https://www.fpga4student.com/2017/06/vhdl-code-for-arithmetic-logic-unit-alu.html`

- PAUSE_IR,
- UPDATE_IR.

Sčítačka — 8-bitová sčítačka složená z jednobitových úplných sčítaček.

Obvod odečítající konstantu — Obvod odečítající od 8-bitového vstupu konstantu 16. Tento obvod (dec8_16) byl zvolen na základě simulace poruch z předchozí sekce, protože největší délka odhalené chyby byla dva. Proto na tomto obvodu bylo testováno zabezpečení nejmenším možným kódem, který pokryje tyto chyby.

Pro tyto obvody byla sestavena navržená architektura a byla měřena hardwarová redundance a schopnost opravit chyby na výstupu pro různé kódy. Nakonec byl vybrán optimální kód s co nejmenší přidanou hardwarovou redundancí pokrývající všechny chyby. Výsledky měření pro tyto obvody jsou prezentovány v následující části této práce.

4.5.1 Hardwarová redundance

V této části je popsána hardwarová redundance pro testovací obvody popsané v předchozí části. Jsou zde porovnány velikosti architektury pro každý z obvodů a pro srovnání je uvedeno i TMR. Velikost byla porovnána pomocí počtu LUTů po provedení syntézy pro FPGA Basys 3 s optimalizacemi pro snížení plochy — strategie syntézy „Flow_AreaOptimized_high“. V tabulkách 4.10, 4.11, 4.12 a 4.13 můžeme vidět srovnání plochy pro jednotlivé testovací obvody. Posledním dvěma sloupcům tabulek, které popisují pokrytí chyb se věnuje další část práce.

Můžeme pozorovat, že přidaná plocha je větší než pro TMR, zejména při zabezpečení malých obvodů o velikosti jen několika LUTů je nárůst plochy enormní (až 950% pro obvod dec8_16). Největším problémem jsou velké prediktory, proto jsem se pokusil provést jejich minimalizaci. Využil jsem minimalizační software boom[15], dostupný na KČN FIT ČVUT. Pokusil jsem se minimalizovat prediktory pro barrel shifter, jejichž počet vstupů (12) je tímto softwarem ještě možné minimalizovat. Nicméně se ukazuje, že minimalizované obvody se namapují na LUTy hůře než neminimalizované a počet LUTů po minimalizaci dále narůstá, jak je ukázáno v tabulce 4.14.

4.5.2 Korekce chyb

Pro kontrolu korektně opravených chyb navrženou architekturou a zvoleným kódem byl navržený testbench, který simuluje všechny možné chyby na výstupu obvodu pro všechny kombinace vstupů a vyhodnocuje, zda zvolený kód je schopný tyto chyby detekovat, či nikoliv. Struktura testbenche je znázorněná na obr. 4.2. Způsob vkládání chyb je zobrazen na obrázku 4.3. Jak bylo diskutováno v návrhu architektury, viz 3.4, pokud se chyba objeví pouze na

Tabulka 4.10: Porovnání velikostí a vlastností architektury pro ALU s použitím různých kódů.

ALU		**# LUTů**	**HW redundance**	**# nepokrytých chyb**	**nepokryté chyby v %**
Samostatný obvod		129	0,00	535822336	100,00
Kódy	**Parita**	401	210,85	267321600	49,89
	Křížová parita	408	216,28	15724800	2,93
	Hammingův	406	214,73	32497920	6,07
	Rozšířený Hammingův	409	217,05	15724800	2,93
Součin kódů	**Hammingův + rozš. Hammingův**	456	253,49	0	0,00
TMR		394	205,43	0	0,00

Tabulka 4.11: Porovnání velikostí a vlastností architektury pro barrel shifter s použitím různých kódů.

Barrel shifter		**# LUTů**	**HW redundance**	**# nepokrytých chyb**	**nepokryté chyby v %**
Samostatný obvod		18	0,00	1044480	100,00
Kódy	**Parita**	62	244,44	520192	49,80
	Křížová parita	64	255,56	28672	2,75
	Hammingův	67	272,22	61440	5,88
	Rozšířený Hammingův	69	283,33	28672	2,75
Součin kódů	**Hammingův + Hammingův**	81	350,00	0	0,00
TMR		39462	244,44	0	0,00

Tabulka 4.12: Porovnání velikostí a vlastností architektury pro JTAG FSM s použitím různých kódů.

JTAG FSM		**# LUTů**	**HW redundance**	**# nepokrytých chyb**	**nepokryté chyby v %**
Samostatný obvod		7	0,00	262112	100,00
Kódy	**Parita**	45	542,86	131040	49,99
	Křížová parita	56	700,00	992	0,38
	Hammingův	48	585,71	8160	3,11
	Rozšířený Hammingův	54	671,43	4064	1,55
Součin kódů	**Kříž. parita + rozš. Hammingův**	88	1157,14	0	0,00
TMR		34	385,71	0	0,00

výstupu prediktoru nebo pouze na výstupu pravé jednotky, je opravena vždy. U levé jednotky záleží na voleném kódu, a proto jsou chyby vkládány pouze na výstup této jednotky. Chyby na výstupu obou jednotek, případně prediktoru a jednotky (nebo obou jednotek) nejsou uvažovány, protože k jejich opravě nebyla architektura navržena. Chyby jsou na výstup levé jednotky v navr-

Tabulka 4.13: Porovnání velikostí a vlastností architektury pro DEC8_16 s použitím různých kódů.

DEC8_16		**# LUTů**	**HW redundance**	**# nepokrytých chyb**	**nepokryté chyby v %**
Samostatný obvod		2	0,00	65280	100,00
Kódy	**Křížová parita**	21	950,00	0	0,00
	Rozšířený Hammingův	21	950,00	0	0,00
TMR		14	600	0	0,00

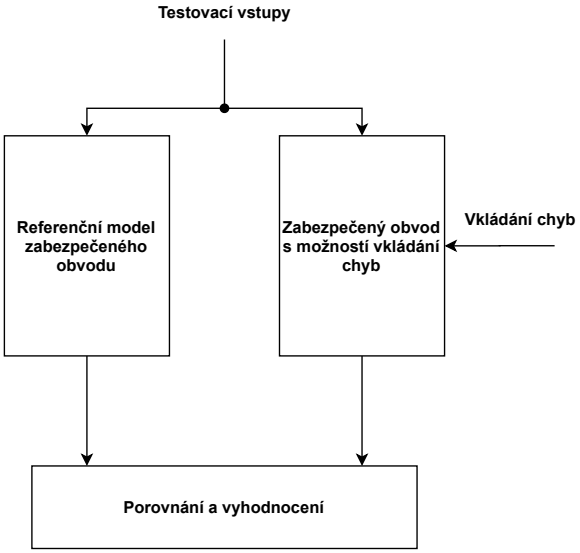
Tabulka 4.14: Porovnání velikostí neminimalizovaných a minimalizovaných prediktorů pro barrel shifter.

Komponenta	**Velikost (# LUTů)**
Barrel shifter	18
Prediktor parity	20
Prediktor parity minimalizovaný	24
Prediktor křížové parity	23
Prediktor křížové parity minimalizovaný	39
Prediktor Hammingova kódu	22
Prediktor Hammingova kódu min.	39
Prediktor rozš. Hammingova kódu	22
Prediktor rozš. Hammingova kódu min.	40
Prediktor součinu Hammingových kódů	29
Prediktor součinu Hammingových kódů min.	93

žené architektuře vloženy operací XOR výstupu jednotek a vkládaných chyb. Vlastní simulace probíhá testováním všech možných vstupů a ke každému vstupu všech možných chyb. Výstup zabezpečeného obvodu s vkládanými chybami se porovnává s výstupem referenčního zabezpečeného obvodu a počítají se případy, kdy se výstupy lišily, tedy vložená chyba nebyla opravena. Tato simulace na rozdíl od simulačního softwaru na KČN testuje všechny možné chyby, i když ne všechny se musí na reálném obvodu vlivem poruch projevit. Použil jsem jí především z důvodu rychlejšího testování a ověřování, protože k simulačnímu softwaru na KČN nemám přístup a vždy jsem musel zasílat obvody k testování a čekat na výsledky. Z tohoto důvodu jsou simulačním softwarem otestované pouze některé obvody. Rovněž se ukázalo, že některé zabezpečené architektury jsou tak velké, že je tímto softwarem otestovat nelze. Výsledky testování simulačním SW jsou popsány v následující části práce.

V tabulkách můžeme ve sloupci počet nepokrytých chyb sledovat, jak volba bezpečnostního kódu ovlivní teoretickou schopnost architektury opravovat chyby. Kódy, které mají dostatečnou minimální kódovou vzdálenost k tomu, aby pokryly všechny chyby na výstupu zabezpečovaného obvodu mají v tomto sloupci nulový počet nepokrytých chyb. S volbou kódů s menšími minimálními

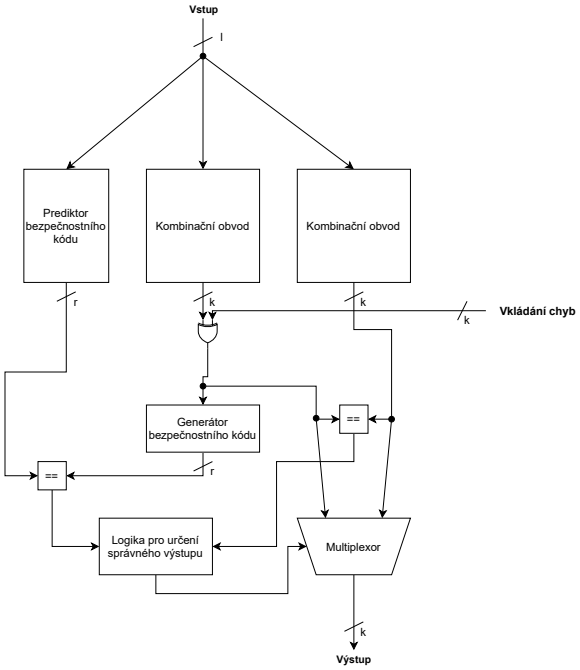
kódovými vzdálenostmi lze pozorovat nárůst počtu chyby, které architektura není schopná opravit. Tento počet je největší u sudé parity, které má ze všech použitých kódů nejmenší minimální kódovou vzdálenost.



Obrázek 4.2: Struktura testbenche pro ověřování korektního opravování chyb.

4.5.3 Ověření výsledků pomocí simulačního SW

Jak již bylo zmíněno v předchozí sekci, pomocí simulačního SW byly ověřeny pouze některé obvody. Konkrétně jde o ALU zabezpečenou pomocí parity, Hammingova kódu, rozšířeného Hammingova kódu. Dalším testovaným obvodem je barrel shifter, zabezpečený paritou, křížovou paritou, Hammingovým a rozšířeným Hammingovým kódem a součinem Hammingových kódů. Výsledky jsou zobrazeny v tabulce 4.15, která zobrazuje počty chybných bitů, které se projevily na výstupu a v tabulce 4.16 zobrazující vzdálenosti projevených chyb. Simulační software testuje výskyt poruch v celé architektuře, tedy například i v komparátorech a selektoru správného výstupu, proto se na rozdíl od navrženého testovacího testbenche projevují chyby na výstupu i při použití součinového kódu, který by měl detekovat všechny chyby a výstup by měl tedy být bezchybný. Také se ukazuje, že s rostoucím přidanou plochou použitého kódu roste i počet těchto chyb. U obvodu barrel shifteru tak lze sledovat, že zabezpečení pomocí rozšířeného Hammingova kódu eliminuje více chyb než zabezpečení pomocí součinu Hammingových kódů, které by teoreticky mělo pokrýt chyby všechny. Výsledky rovněž ukazují, že určité kódy mají lepší vlastnosti než použití TMR, tedy skutečnost, že všechny použité kódy způsobují větší nárůst plochy než u TMR může vyvážit lepší zabezpečení ob-



Obrázek 4.3: Způsob vkládání chyb do obvodu zabezpečeného navrženou architekturou.

vodu. U ALU můžeme pozorovat, že kódem s lepšími vlastnostmi než TMR je Hamingův kód, u barrel shifteru je to rozšířený Hammingův kód.

Tabulka 4.15: Počty detekovaných chybných bitů na výstupu zabezpečených obvodů.

Obvod	**Poruch**	**Vstupy**	**Výstupy**	**Počet chybných bitů na výstupu**								
				1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ALU, Hammingův kód	3471	20	9	548866	259008	799812	168609	106367	50665	65248	11759	0
ALU, roz. Hamming. kód	1688	20	9	450798	190085	136102	138670	57365	29372	17495	12355	11
ALU, sudá parita	6789	20	9	689773	2736464	307297	1263275	161897	339852	66081	57846	1406
ALU, TMR	1147	20	9	580184	309545	204166	126572	70197	48125	26364	6893	25
Barrel shifter, křížová parita	496	12	8	115565	20583	17085	14061	9364	3450	776	84	-
Barrel shifter, Hamming. kód	321	12	8	72369	12381	3825	706	9	0	0	0	-
Barrel shifter, roz. Hamming. kód	236	12	8	52921	9935	7888	5669	2838	751	119	8	-
Barrel shifter, součin Hammingových kódů	228	12	8	2150839	2108164	2104704	2101589	2099520	2098049	2097361	2097173	-
Barrel shifter, sudá parita	442	12	8	93262	25656	7563	5660	1199	444	47	8	-
Barrel shifter, TMR	247	12	8	61063	13970	9979	6399	2906	938	240	37	-

Tabulka 4.16: Počty detekovaných největších délek chyb na výstupu zabezpečených obvodů.

Obvod	**Poruch**	**Vstupy**	**Výstupy**	**Největší detekovaná délka chyb**							
				0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
ALU, Hammingův kód	3471	20	9	548866	195387	722227	127282	148323	122839	144772	76305	81314
ALU, roz. Hamming. kód	1688	20	9	450798	163451	101569	91246	112479	57348	46667	65963	77380
ALU, sudá parita	6789	20	9	689773	2521786	979635	1243681	562474	509380	240751	188948	153856
ALU, TMR	1147	20	9	580184	284887	191857	149704	101178	79969	103441	84445	79976
Barrel shifter, křížová parita	496	12	8	115565	12046	13087	11068	13138	12445	13550	9123	-
Barrel shifter, Hamming. kód	321	12	8	72369	287	6281	231	9666	260	2804	0	-
Barrel shifter, roz. Hamming. kód	236	12	8	52921	3862	7048	5012	6959	4281	4446	2313	-
Barrel shifter, součin Hammingových kódů	228	12	8	84857	12878	14270	12401	15972	14350	15403	10653	-
Barrel shifter, sudá parita	442	12	8	93262	4236	11297	5003	17332	4388	6144	2676	-
Barrel shifter, TMR	247	12	8	61063	4955	9630	3171	9335	5860	8866	2124	-

4.5.4 Zvolení kódu s nejmenší hardwarovou redundancí

Pro zabezpečení obvodu je vhodné volit takový kód, díky kterému výsledná architektura bude schopná opravit co nejvíce chyb a současně bude mít co nejmenší plochu. Zatímco z teoretické úvahy o volbě kódu, který bude schopný pokrýt všechny možné chyby na výstupu vychází volba větších a složitějších kódů, například součinových, ověřováním pomocí simulačního SW se ukazuje, že plocha přidaná komparátory a logikou pro volbu výstupu poskytuje prostor pro výskyt dalších chyb. Vzhledem k výsledkům simulací pomocí navrženého testbenche a simulačního SW se ukazuje, že je výhodnější použít kódy menší. Nicméně některé chyby se projeví na výstupu vždy, protože komparační logika a logika pro volbu správného výstupu není zabezpečena. Pro zabezpečení ALU se z hlediska opravy chyb jeví jako nejvýhodnější použít Hammingův kód s hardwarovou redundancí 214,73 % (HW redundance TMR je v tomto případě 205,43 %) a pro barrel shifter použít rozšířený Hammingův kód (HW redundance 283,33 % proti TMR s 244,44 %). Rozdíl v přidané ploše je proti TMR v řádech jednotek až desítek procent, ale použitím navržené architektury je možné více snížit počet chyb na výstupech.

4.6 Vytvoření klasifikace obvodů z hlediska opravy a detekce chyb

Navržená architektura slouží k opravování chyb, proto jsem vytvořil teoretickou klasifikaci obvodů z hlediska možnosti opravy chyb na výstupech. Nicméně zabezpečené obvody je nutné dále ověřit pomocí simulačního softwaru, protože se ukazuje, že kód, který je schopný pokrýt všechny chyby nemusí být nutně nejlepší v kontextu celé navržené architektury. Klasifikace využívá k rozdělení obvodů největší vzdálenosti chyb na výstupech zabezpečeného obvodu určenou pomocí simulačního softwaru. K teoretickému pokrytí chyb příslušných délek je pak volen odpovídající kód, viz tabulka 4.17. Nemá smysl volit kód pro pokrytí větších chyb, než udává tabulka. Současně je nutné pomocí simulačního softwaru otestovat architekturu pro zvolený kód a kódy z vyšších řádků tabulky a zvolit kód, propouštějící nejméně chyb. Tímto algoritmem bude zvolený kód s nejlepšími vlastnostmi a nejmenší možnou hardwarovou redundancí. Pro opravu více než 14 chyb nemá navržená architektura oporu v bezpečnostních kódech. Obvody, jejichž chyby splňují vlastnosti uvedené v tabulce lze klasifikovat jako obvody, jejichž výstup lze navrženou architekturou zabezpečit, ostatní obvody s většími délkami chyb jsou klasifikované jako nezabezpečitelné (nebo zabezpečitelné, ale s větším počtem neopravitelných chyb). Pro takové obvody by bylo nutné přidat další bezpečnostní kód požadovaných vlastností, nebo použít stávající kódy (které ale propustí více chyb).

Tabulka 4.17: Zvolení kódu pro zabezpečovaný obvod.

Největší detekovaná délka chyby v bitech	**Kód pro pokrytí**
0 (tedy jednobitová chyba)	sudá parita
1	Hammingův kód
2	rozšířený Hammingův kód, křížová parita
3-4	součin Hammingova kódu a sudé parity
5-7	součin Hammingových kódů
8-10	součin Hammingova kódu a rozšířeného Hammingova kódu nebo křížové parity
11-14	součin libovolné kombinace křížové parity a rozšířeného Hammingova kódu

4.7 Specifikace požadavků na úpravu simulačního SW

Pro lepší možnosti zkoumání zabezpečení obvodů navrženou architekturou by bylo potřeba rozšířit simulační software o možnosti sledování chyb. Bylo by výhodné zjistit, odkud přesně se chyby propagují, aby bylo možné dále zdokonalovat zabezpečení. Dále by bylo vhodné přidat možnost vkládání poruch jen do určité části obvodu, aby se dalo lépe sledovat, jak se projevují chyby v samotné zabezpečované logice, jak se v celkové architektuře projevují chyby v prediktoru a generátoru kódu, a jak chyby v komparační a korekční logice.

Úprava simulačního softwaru tak, aby byl schopný sám doporučit vhodný kód pro zabezpečení by byla složitá, protože se ukázalo, že zvolený kód schopný s minimální kódovou vzdáleností takovou, aby pokryl všechny možné poruchy na výstupu zabezpečované logiky nemusí být nejlepším možným kódem v kontextu celé zabezpečené architektury. Zde je potřeba vytvořit architektury s příslušnými kódy, znovu odsimulovat jejich vlastnosti a teprve poté zvolit optimální kód. Dalším problémem je, že vzniklá architektura je velká a především pro větší obvody je problém s jejich simulací, protože s větším obvodem roste také počet poruch, které se musejí ověřit a roste tedy i výpočetní náročnost simulace.

4.8 Skripty pro generování navržené architektury

Pro jednodušší generování a testování navržené architektury byly vytvořeny skripty, které jsou přiloženy na CD. Struktura skriptů je popsána zde:

Generator.py — hlavní skript sloužící k vygenerování architektury se zvoleným bezpečnostním kódem pro předaný VHDL soubor.

Parser.py — skript sloužící k parsování VHDL entit. Umožňuje získat informace o názvu entity, portech a generic hodnotách.

EntityGenerator.py – slouží ke generování deklarací entit, komponent a instanciace komponent.

CodeGenerator.py generuje VHDL soubory s generátory bezpečnostních kódů.

ProcessGenerator.py generuje VHDL procesy.

template.vhd šablona pro generování VHDL souborů.

vhdl_template.py soubor s textovými konstantami používanými při nahrazování šablon v souboru template.vhd.

Pro generování architektury slouží skript Generator.py, který pro svou funkci využívá ostatní popsané skripty. Pro použití musí všechny popsané soubory a skripty zůstat ve stejném adresáři. Pro spuštění je nutné mít nainstalovaný Python 3, skripty nevyžadují instalaci žádných dalších balíků. Pro zahájení generování je nutné skriptu jako parametry předat cestu ke vstupnímu VHDL souboru (s logikou k zabezpečení), výstupnímu VHDL souboru (pro uložení vygenerovaného VHDL). Pomocí přepínače -g je možné specifikovat, co bude skript generovat. Možnosti jsou TMR, DMR, nebo sec pro vygenerování navržené architektury. Pokud je zvolena možnost sec, je nutné specifikovat kód pro zabezpečení. Ke specifikaci kódu slouží přepínače -c pro výběr kódu, nebo -p pro výběr součinu kódů. Přepínačem -h je možné vyvolat nápovědu. Ukázka použití skriptu je zobrazena na obr. 4.4.

```
usage: Generator.py [-h] [-g {TMR,DMR,sec}]
                    [-c {parity,cross_parity,hamming,extended_hamming} | -p {parity,cross_parity,hamming,extended_hamming} {parity,cross_parity,hamming,extended_hamming}]
                    input_file output_file
```

Obrázek 4.4: Volání skriptu pro generování navržené architektury.

Máme-li tedy například ALU (s top level entitu v souboru alu_top.vhd), kterou chceme zabezpečit navrženou architekturou a kódem pro křížovou paritu, použijeme následující volání: `python3 Generator.py alu_top.vhd alu_top_secured.vhd -g sec -c cross_parity`.

Pokud je zvoleno generování TMR nebo DMR, je vygenerován jeden VHDL soubor. V případě generování navržené architektury jsou kromě zvoleného výstupního souboru vygenerovány ještě soubory pro prediktor a generátor(y) kódů, které budou uloženy ve stejném adresáři, jako je **adresář hlavního výstupního souboru**, předaného jako skriptu jako parametr. Tyto soubory

obsahují komponenty pro hlavní vygenerovaný VHDL soubor s architekturou. Tyto soubory používají jako prefix jméno vstupního souboru a jejich pojmenování je popisné. Např. pro ALU z ukázky výše se tak ještě vygenerují kromě souboru alu_top_secured.vhd ještě soubory: alu_top_cross_parity.vhd a alu_top_predictor_cross_parity.vhd.

V současné chvíli skript podporuje generování pro VHDL entity bez použití generických hodnot a pouze pro vstupní a výstupní porty typu std_logic a std_logic_vector. Pokud toto entita nesplňuje, je nutné pro ni před použitím skriptu vytvořit wrapper s těmito vlastnostmi.

Skript podporu i pro generování samotných prediktorů, generátorů kódu a Parser.py dokáže parsovat i entity s porty typu std_logic_vector se šířkou udanou pomocí generické hodnoty. Tyto funkcionality jsou však zatím pouze experimentální a pro účely poskytnutí skriptů k této práci nejsou přepínači hlavního skriptu dostupné.

Skripty obsahují dokumentaci v **Docstrings**, kterou je v případě potřeby možné vygenerovat nástroji pro generování dokumentace Pythonu, například pomocí **pydoc**.

Závěr

Cílem této práce bylo nalézt vhodný způsob zabezpečení kombinačních obvodů pomocí bezpečnostních kódů. Za tímto účelem byla navržena a otestována architektura založená na DMR a volbě správného výstupu za pomoci bezpečnostních kódů. Tato architektura byla otestována z hlediska přidané plochy a schopností opravy chyb.

Ukázalo se, že vybrat optimální kód tak, aby nárůst plochy byl co nejmenší, je komplexní úkol. Bylo zjištěno, že kódy schopné pokrýt veškeré chyby na výstupech zabezpečených obvodů mají příliš velkou přidanou plochu a vnášejí do architektury další chyby. Optimální kód proto musí být hledán nejenom podle svých schopností detekovat a opravovat chyby, ale i pomocí simulace poruch ve výsledné architektuře s různými použitými kódy. U testovaných obvodů se podařilo najít takové kódy, že nárůst plochy je jen o jednotky až desítky procent větší než při použití TMR a navržená architektura přitom dokáže opravit více chyb. Nicméně se ukazuje, že některé chyby touto architekturou opravit nelze, zejména pokud se stanou vlivem poruchy mimo zabezpečovaný obvod v komparační a opravné části navržené architektury.

Na základě navržené architektury byla vytvořená klasifikace kombinačních obvodů z hlediska možnosti opravy chyb a vypracována doporučení na rozšíření softwaru pro simulaci poruch v obvodech.

Navržená architektura umožňuje další výzkum s využitím jiných bezpečnostních kódů pro zabezpečení. Možným směrem výzkumu může být také zabezpečení komparační a opravné části navržené architektury za účelem dalšího snížení poruch. K této práci byly přiloženy skripty pro generování navržené architektury, DMR a TMR umožňující pro zadanou VHDL entitu vygenerovat zabezpečený design. Tyto skripty je možné dále rozšiřovat a jejich součásti, například parser VHDL entit, využít i pro jiné manipulace s VHDL soubory a pro jejich generování.

Literatura

[1] Valinataj, M.: A novel self-checking carry lookahead adder with multiple error detection/correction. Oct 2014. Dostupné z: `https://www.sciencedirect.com/science/article/pii/S0141933114001422`

[2] Dutta, A.; Jas, A.: Combinational Logic Circuit Protection Using Customized Error Detecting and Correcting Codes. In *9th International Symposium on Quality Electronic Design (isqed 2008)*, 2008, s. 68–73, doi: 10.1109/ISQED.2008.4479700.

[3] Michael, G.; Graf, S.: *Error detection circuits*. McGraw-Hill, 1993.

[4] Mathew, J.; Banerjee, S.; Mahesh, P.; aj.: Multiple Bit Error Detection and Correction in GF Arithmetic Circuits. In *2010 International Symposium on Electronic System Design*, 2010, s. 101–106, doi: 10.1109/ISED.2010.28.

[5] Ondřej, N.; Elena, G.; Ubar, R.: *Handbook of testing electronic systems*. Czech Technical University Publishing House, 2005, ISBN 80-01-03318-X.

[6] Bushnell, M. L.; Agrawal, V. D.: *Essentials of Electronic Testing for Digital, Memory & Mixed-signal VLSI Circuits*. Kluwer Academic Publisher, 2000.

[7] McCluskey, E. J.; Tseng, C.-W.: In *Stuck-fault tests vs. actual defects*, Proceedings International Test Conference 2000 (IEEE Cat. No.00CH37159, 2000, s. 336–342.

[8] Jan, H.; Vladislav, J.: *Číslicové systémy odolné proti poruchám*. České vysoké učení technické, 1992.

[9] Adámek, J.; České vysoké učení technické v Praze. Elektrotechnická fakulta: *Kódování a teorie informace*. Praha: Ediční středisko Českého vysokého učení technického, první vydání, 1991.

[10] Lala, P. K.: *Self-checking and fault-tolerant digital design*. Morgan Kaufmann, 2001.

[11] Borecký, J.; Hülle, R.; Fišer, P.: Evaluation of the SEU Faults Coverage of a Simple Fault Model for Application-Oriented FPGA Testing. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, 2020, s. 684–691, doi:10.1109/DSD51259.2020.00111.

[12] Yang, S.: *Logic Synthesis and Optimization Benchmarks User Guide: Version 3.0*. Microelectronics Center of North Carolina (MCNC), 1991. Dostupné z: `https://books.google.cz/books?id=7ruGuAAACAAJ`

[13] Brglez, F.; Bryan, D.; Kozminski, K.: Combinational profiles of sequential benchmark circuits. In *IEEE International Symposium on Circuits and Systems,*, 1989, s. 1929–1934 vol.3, doi:10.1109/ISCAS.1989.100747.

[14] Fišer, P.: Adders [Online]. Dostupné z: `https://ddd.fit.cvut.cz/prj/Benchmarks/Adders.7z`

[15] Fišer, P.: BOOM-II: The PLA minimizer [Online]. Dostupné z: `https://ddd.fit.cvut.cz/prj/BOOM/`

Příloha A

Seznam použitých zkratek

DMR Double modular redundancy

DPS Deska plošných spojů

FPGA Field programmable gate array

FSM Final state machine

HW Hardware

NMR N-modular redundancy

RS Reed-Solomon

SW Software

TMR Triple modular redundancy

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `scripts` ..... zdrojové kódy generujících skriptů
  - `examples` ..... VHDL soubory měřených obvodů
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF