České vysoké učení technické v Praze
Fakulta elektrotechnická

Diplomová práce

Vysoce spolehlivý systém založený na bázi hradlových polí

Jiří Kvasnička

Vedoucí práce: Ing. Pavel Kubalík

Studijní program: Elektrotechnika a informatika

Obor: výpočetní technika (magisterský)

Květen 2006

Poděkování

Rád bych na tomto místě poděkoval vedoucímu mé diplomové práce, Pavlu Kubalíkovi, za jeho podněty ke zpracovávanému tématu, ochotu a věnovaný čas při psaní této diplomové práce.

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon)

V Praze dne 25. května. 2006 ………………………………….…

Abstract

This thesis describes the realization of complex system of self-testing circuits testing. The self-testing property and fault security is observed. Testing was done on set of benchmarks. This testing was realized in FPSLIC from ATMEL Company. Dynamic partial reconfiguration (driven from embedded AVR) was used to put faults into the FPGA.

This thesis also describes all logic needed to be added to circuits to ensure the measuring of fault security, self-testing and totally self-checking properties. Complete design flow from benchmark to bitstream with benchmark in testing environment is described in detail. Some software tools were made to safeguard the automatic VHDL source code generation and bitstream generation, as much automatic, as possible.

Measuring with couple of benchmarks was done. Results were compared with software simulation of fault injection.

.

Abstrakt

Tato práce se zabývá realizací systému pro testování vlastností samočinně kontrolovaných obvodů, reprezentovaných sadou testovaných benchmarků. Realizace proběhla na vývojové desce hradlového pole FPSLIC od firmy ATMEL. Pro vkládání poruch byla použita dynamická rekonfigurace.

Práce popisuje veškerou logiku, kterou je třeba k benchmarku přidat, aby bylo možné měřit jeho odolnost proti poruchám. Dále popisuje postup vytvoření bitstreamu se všemi nutnými kroky a softwarovými nástroji. V rámci této práce byly vytvořeny programy pro generování této logiky v jazyce VHDL, které byly navrženy pro zautomatizování pracovního postupu. Práce prezentuje i výsledky měření na sadě benchmarků.

Obsah

Seznam obrázků xi

Seznam tabulek xiii

1 Úvod 1

2 Popis problému 2

2.1 Poruchy v hradlových polích 2

2.2 Základní termíny 2

2.3 Klasifikace poruch 3

3 Analýza 5

3.1 Proč testovat v hardwaru 5

3.2 Volba hradlového pole 5

3.3 Základní charakteristika a možnosti FPSLICu 6

3.3.1 Bitstream FPSLICu 6

3.3.1.1 Struktura buňky FPSLICu 6

3.3.1.2 Formát bitstreamu MD4 9

3.3.1.3 Obsah bitstreamu 10

3.3.2 Odhad podílu testovaných bitů z celého bitstreamu 11

4 Návrh 14

4.1 Benchmark 15

4.2 Generování testovacích vektorů 15

4.3 Zpracování výsledků 16

4.4 Vkládání poruch 16

5 Popis implementace v FPSLICu 17

5.1 Logika zajišťující komunikaci FPGA-AVR 18

5.2 Mapování I/O prostoru v FPGA 19

5.3 Generátor signálu start 20

5.4 Vnitřní část nezávislá na implementaci vrstvy komunikující s AVR 21

5.5 Generátor testovacích vektorů 22

5.6 Saturační jednobitový čítač 23

5.7 Testované obvody 23

5.8 Hlídač kódu 25

5.8.1 Analýza prostorové a časové náročnosti hlídače parity 26

5.8.2 Implementace a dosažení optimální syntézy hlídače parity 27

5.9 Komparátor 29

5.9.1 Analýza prostorové a časové náročnosti komparátoru 30

5.9.2 Implementace a dosažení optimální syntézy hlídače parity 32

5.10 Návrh v AVR části 34

5.10.1 Základní konfigurace, celkový přehled návrhu 35

5.10.2 Organizace paměti 36

5.10.3 Komunikace s FPGA 37

5.10.4 Komunikace s PC 37

5.10.5 Injekce poruch a testování 40

6 SW nástroje 44

6.1 Generování VHDL benchmarku 45

6.2 Generování VHDL kódu pro implementaci v FPSLICu 46

6.3 Syntéza do EDIF 46

6.4 Vytvoření bitstreamu 48

6.4.1 Figaro IDS 48

6.4.1.1 Význam parametru mapování ... 49
6.4.1.2 Postup vytvoření makra ... 50
6.4.2 Ruční rozmístění a generování bitstreamu ... 51
6.4.3 Kompletace bitstreamu ... 52
6.5 Programování přípravku ... 53
6.6 PC – komunikační program ... 54
6.6.1 Podmínky pro běh programu ... 54
6.6.2 Hlavní okno ... 54
7 Testování a výsledky ... 59
7.1 Vliv mapování na parametry návrhu ... 59
7.2 Výsledky testování benchmarků pro sudou paritu ... 62
7.2.1 Routovací buňky ... 64
7.3 Čas měření ... 65
7.4 Porovnání výsledků ... 66
8 Zhodnocení ... 67
8.1 Doporučení pro další postup ... 67
9 Závěr ... 68
A Literatura a zdroje ... 69
B Seznam použitých zkratek ... 71
C Úplné výsledky ... 73
D Mapa přiřazení signálů AVR-FPGA ... 77
E Obsah přiloženého CD ... 79

Seznam obrázků

Obrázek 3.1: Struktura měření a klasifikace poruch ... 5
Obrázek 3.2: Propojení mezi buňkami ... 7
Obrázek 3.3: Propojení buňky s datovými vodiči ... 8
Obrázek 3.4: Struktura buňky FPSLICu ... 9
Obrázek 3.5: Ukázka bitstreamu ve formátu MD4 ... 9
Obrázek 3.6: Pořadí bytů v řádce bitstreamu MD4 ... 9
Obrázek 3.7: Přibližná distribuce bitstreamu ve struktuře FPSLICu ... 12
Obrázek 4.1: Struktura měření a klasifikace poruch ... 14
Obrázek 4.2: Struktura testovaného obvodu ... 14
Obrázek 5.1: Celkový pohled na implementaci ... 17
Obrázek 5.2: Top-level design FPGA ... 18
Obrázek 5.3: Mapování vstupů I/O prostoru v FPGA ... 20
Obrázek 5.4: Mapování výstupů I/O prostoru v FPGA ... 20
Obrázek 5.5: Generátor signálu start ... 20
Obrázek 5.6: Vnitřní část nezávislá na implementaci vrstvy komunikující s AVR ... 21
Obrázek 5.7: Generátor testovacích vektorů ... 22
Obrázek 5.8: Jednobitový saturační čítač ... 23
Obrázek 5.9: Modul TestedCircuit ... 24
Obrázek 5.10: Separace modulů komb1 a komb2 ... 24
Obrázek 5.11: Schéma hlídače parity ... 25
Obrázek 5.12: Parita rozdělená do 2 skupin ... 26
Obrázek 5.13: Optimální strom ... 26
Obrázek 5.14: Maximálně nevyvážený strom ... 26
Obrázek 5.15: Graf závislosti délky datové cesty na počtu bitů vstupního vektoru ... 27
Obrázek 5.16: Paritní strom – standardní syntéza (technology view) ... 28
Obrázek 5.17: Paritní strom s minimální hloubkou (technology view) ... 28
Obrázek 5.18: Číslování vodičů v paritním stromu ... 29
Obrázek 5.19: Struktura komparátoru ... 29
Obrázek 5.20: Sdružení XORů v komparátoru ... 30
Obrázek 5.21: Schematické rozdělení komparátoru na LUTy ... 31
Obrázek 5.22: Nevhodná syntéza komparátoru (technology view) ... 32
Obrázek 5.23: Detail nevhodné syntézy komparátoru z předešlého obrázku ... 32
Obrázek 5.24: Optimální syntéza komparátoru (výřez, technology view) ... 33
Obrázek 5.25: Graf plochy zabrané komparátorem v FPGA ... 34
Obrázek 5.26: Mapa paměti AVR ... 36
Obrázek 5.27: Kruhová fronta ... 38
Obrázek 5.28: Schéma komunikace ... 38
Obrázek 5.29: Struktura bytu parametrů ... 39
Obrázek 5.30: Pořadí bytů v bitstreamu buňky ... 39
Obrázek 5.31: Pořadí bytů ve výsledcích ... 40
Obrázek 5.32: Pořadí bitů ve výsledku ... 40
Obrázek 5.33: Obecné pořadí bytů u 16bitových slov ... 40
Obrázek 5.34: Generování poruch v LUTu ... 41
Obrázek 5.35: Příklady masek poruch ... 42
Obrázek 5.36: Vývojový diagram testování ... 43
Obrázek 6.1: Vývojový diagram implementace v FPSLICu ... 44
Obrázek 6.2: Soubory zpracovávané programem generatevhd ... 45
Obrázek 6.3: Soubory zpracovávané programem GenVHDL ... 46

Obrázek 6.4: Syntéza do EDIFu 47
Obrázek 6.5: Oprava FPGA.edf 47
Obrázek 6.6: Postup vytvoření bitstreamu 48
Obrázek 6.7: Nastavení mapování 49
Obrázek 6.8: TSC parity checker (výřez) 49
Obrázek 6.9: Checker – mapování povoleno 50
Obrázek 6.10: Checker – mapování zakázáno 50
Obrázek 6.11: FIGARO IDS - New Design 50
Obrázek 6.12: Library out of date 51
Obrázek 6.13: Rozmístění makra 51
Obrázek 6.14: Rozmístěný design 52
Obrázek 6.15: Nastavení paměti FPSLICu 53
Obrázek 6.16: Povolení rekonfigurace 53
Obrázek 6.17: Hlavní okno 54
Obrázek 6.18: Menu soubor 55
Obrázek 6.19: Menu komunikace 55
Obrázek 6.20: Menu Testy 56
Obrázek 6.21: Okno Výběr testovací oblasti 57
Obrázek 6.22: Výběr oblasti myší 58
Obrázek 6.23: Omezení výběru 58
Obrázek 7.1: Graf závislosti velikosti benchmarku na nastavení mapování 60
Obrázek 7.2: Graf mezní frekvence benchmarků v závislosti na nastavení mapování 61
Obrázek 7.3: B111 – vliv mapování 62
Obrázek 7.4: s1488 – vliv mapování 62

Seznam tabulek

Tabulka 3.1: Pravdivostní tabulka signálu XL s příkladem 11
Tabulka 3.2: Pravdivostní tabulka signálu YL s příkladem 11
Tabulka 3.3: Přibližná distribuce bitstreamu ve struktuře FPSLICu 13
Tabulka 5.1: Mapování vstupů I/O prostoru v FPGA 19
Tabulka 5.2: Mapování výstupů I/O prostoru v FPGA 20
Tabulka 5.3: Nejdelší datové cesty hlídače parity pro 2vstupový XOR 26
Tabulka 5.4: Nejdelší datové cesty hlídače parity pro 4vstupové LUTy 27
Tabulka 5.5: Nejdelší datové cesty komparátoru pro 2vstupové XNORy a ANDy 30
Tabulka 5.6: Nejdelší datové cesty komparátoru pro 2vstupový XOR 32
Tabulka 5.7: Konfigurace a odchylky pro různé přenosové rychlosti 4MHz krystalu 35
Tabulka 5.8: Mapování I/O adres v závislosti na konfiguraci registru FISCR 37
Tabulka 5.9: Velikosti přenášených dat 39
Tabulka 6.1: Přípony benchmarku 45
Tabulka 7.1: Závislost velikosti makra na nastavení mapování 59
Tabulka 7.2: Mezní frekvence benchmarků v závislosti na nastavení mapování 60
Tabulka 7.3: Výsledky testování vybraných částí LUTů (bez routovacích buněk) 62
Tabulka 7.4: Výsledky testování celých LUTů (bez routovacích buněk) 63
Tabulka 7.5: Podíl nevyužívaných částí LUTů 63
Tabulka 7.6: Výsledky testování vybraných částí LUTů v procentech 64
Tabulka 7.7: Výsledky testování kompletních LUTů v procentech 64
Tabulka 7.8: Výsledky testování LUTů v routovacích buňkách 65
Tabulka 7.9: Časy měření 66
Tabulka 7.10: Porovnání výsledků SW a HW testování 66
Tabulka 9.1: Výsledky testování pouze použitých bitů LUTu (bez routovacích buněk) 73
Tabulka 9.2: Výsledky testování všech bitů LUTu (bez routovacích buněk) 74
Tabulka 9.3: Výsledky testování LUTů routovacích buněk 75

1 Úvod

Obsahem této diplomové práce je návrh a implementace prostředí umožňující testování poruch přímo v hradlovém poli firmy ATMEL. Testovanými obvody byla sada zadaných benchmarků. Cílem bylo zjistit, jak se daný obvod vypořádá s jednobitovými poruchami, je-li zabezpečen bezpečnostním kódem a jak se budou výsledky lišit od softwarově provedených simulací.

Motivací pro zkoumání reakce na jednobitovou poruchu je jev označovaný jako SEU [1,2]. Vlivem SEU může dojít k poruchám v hradlovém poli a pravděpodobnost výskytu tohoto jevu se zvyšuje se vrůstající nadmořskou výškou, jelikož zemská atmosféra a hlavně zemské magnetické pole vytváří velmi účinný štít proti částicím způsobující SEU. Zvláště v letectví a kosmonautice nelze tento jev podceňovat.

Zvláštní pozornost v této diplomové práci byla kladena na sledování odolnosti proti poruchám (FS) a úplné samočinné kontrolovanosti (TSC) benchmarku při vkládání poruch do bitstreamu.

Hardwarová realizace proběhla na vývojové desce ATSTK94 s FPSLIC AT94K40AL a využívá se zde dynamické rekonfigurace pro rychlé vkládání poruch do hradlového pole.

Návrh byl rozdělen na 4 hlavní oblasti:

1. Návrh přídavné logiky k benchmarkům umožňující sledování spolehlivostních parametrů při vkládání poruch.
2. Vytvoření softwarových nástrojů pro generování přídavné logiky a maximální zautomatizování celého procesu končící vytvořením bitstreamu pro naprogramování FPSLICu.
3. Program běžící v zabudovaném AVR jádru obvodu FPSLIC řídící celé FPGA, testování a vkládání poruch
4. Obslužný program pro PC, kde se graficky vybírají oblasti poruch a pomocí sériového portu se přenášejí data do přípravku, spouští se testování a vyčítají výsledky

V následujících kapitolách rozebírám, kromě všech výše zmiňovaných oblastí, spolehlivostní ukazatele, které v obvodech měřím, možnosti obvodu FPSLIC a na závěr prezentuji výsledky z měření benchmarků.

2 Popis problému

2.1 Poruchy v hradlových polích

FPGA mají typicky SRAM strukturu, což znamená, že v sobě neobsahují bitstream. Ten se musí do hradlového pole nahrát, po vypnutí napájení se naprogramování ztrácí a po opětovném připojení napájení se musí bitstream opět nahrát.

Právě tím, že je konfigurace FPGA uložena v nestálé (volatile) paměti typu SRAM, jsou FPGA náchylná k poruchám v této paměti. Asi nejmarkantnější je tento vliv v letectví a kosmonautice, kdy se vzrůstající vzdáleností od povrchu klesá ochrana magnetického pole a vrstev atmosféry, která chrání před kosmickým zářením, slunečním větrem a jinými typy částic a záření, se kterými se ve vesmíru můžeme setkat.

Letící částice může způsobit různé fyzické defekty, trvalého, nebo dočasného charakteru. Nejrozšířenějším defektem je průlet nabité částice kanálem tranzistoru, kde může změnit vodivost a ovlivnit tak chování obvodu.

Těmto poruchám se říká SEU (Single Event Upset [1,2]). Jedná se o změnu stavu vyvolanou nabitou částicí, která se může projevit v digitálních, analogových i optických částech obvodu (CCD snímače). Jedná se o chyby, které nemají primárně destruktivní vliv na obvod a po resetu nebo po opětovném naprogramování zařízení funguje opět normálně. Tyto chyby se mohou projevit jako přechodný puls v logice, nebo jako změna stavu v klopných obvodech. Tyto projevy rychle odezní a záleží pouze na struktuře obvodu, jak závažné škody porucha napáchá.

Dalším možným projevem je změna bitu v paměťové buňce. Mezi paměťové buňky se v FPGA řadí i konfigurační paměť, která je většinou typu SRAM. Pak tyto poruchy zůstávají a navrácení k původnímu stavu se děje až po novém zápisu do paměťové buňky nebo v případě konfigurační paměti až po rekonfiguraci nebo opětovném natažení bitstreamu.

Současný trend trhu je vyrábět FPGA s technologicky co možná nejmenšími rozměry tranzistoru. Dalším trendem je snižování napájecího napětí, potažmo prahového napětí tranzistorů., což vede k nižší šumové imunitě a stále větší náchylnosti k SEU, proto má smysl sledovat, jak se FPGA zachová v případě výskytu této poruchy

2.2 Základní termíny

Jelikož se v této práci zabývám spolehlivostí návrhu v FPGA, bylo by dobré nejprve definovat základní termíny, se kterými v této práci operuji. Během měření spolehlivosti se setkáváme s pojmy porucha a chyba. Jejich smysl by měl být intuitivně zřejmý, přesto bych zde rád uvedl jejich definice[5] podle ČSN 010102:

- **Porucha** (fault) je jev spočívající v ukončení schopnosti objektu plnit požadovanou funkci podle technických podmínek.
- **Chyba** (error) je rozdíl mezi správnou a skutečnou hodnotou nějaké veličiny, zjištěný měřením nebo pozorováním.

Z uvedených dvou definic vyplývá, že chyba je obvykle důsledkem nějaké poruchy, avšak každá porucha se nemusí nutně projevit jako chyba (např. nepoužívá-li se při provádění funkce žádná z poruchových částí).

Při zkoumání obvodu sledujeme, zda je obvod:

- **Odolný proti poruchám** (FS, Fault Security): Obvod je odolný proti poruchám, pokud pro každou poruchu výstupní vektor nepatří do kódových slov, nebo se porucha neprojeví (jinými slovy chybný výstup nesmí být kódové slovo)

- **Samočinně testovaný** (ST, Self-Testing): obvod je samočinně testovaný, jestliže pro všechny poruchy existuje vstupní vektor, který generuje výstupní vektor nepatřící do kódového slova
- **Úplně samočinně kontrolovaný** (TSC, Totally Self-Checking): obvod je úplně samočinně kontrolovaný, pokud je odolný proti poruchám a zároveň samočinně testovaný (tedy pokud splňuje výše uvedené 2 vlastnosti)

Podle způsobu, jakým jsou jednotlivé funkční bloky testovány, hovoříme o 2 základních přístupech k testování:

- **Off-line testování**: Obvod se většinou přepíná do speciálního testovacího režimu, ve kterém je funkční blok otestován pomocí speciální logiky, která při normálním provozu není využívána. Během tohoto testovacího režimu není obvod dostupný.
- **On-line testování**: Tento přístup nám zachovává dostupnost obvodu. Testování probíhá během normálního provozu, kdy se vyhodnocuje správnost výsledků pomocí speciální logiky, která tuto kontrolu neustále provádí. Cílem tohoto přístupu je, aby byla jakákoli chyba ihned detekována a pokud to je možné i opravena

Existuje více způsobů, jak současně za běhu obvodu (on-line) detekovat chyby (CED techniky, Concurrent Error Detection). V této práci jsem použil dvou technik:

- **Bezpečnostní kód** (parita). Ke sledování vlastností (FS, ST, TSC) obvodů bohužel bezpečnostní kód nepostačuje. Bezpečnostní kód nám poskytne pouze informaci o správnosti kódu, nikoli o správnosti výsledku.
- **Zdvojení**. Obvod je přítomny ve 2 kopiích a výsledky obou kopií jsou porovnávány.

Tato práce zkoumá právě detekční schopnosti bezpečnostního kódu. Zdvojení obvodu se zde nepoužívá z důvodu zvýšení spolehlivosti, ale pouze pro potřeby měření vlastností (FS, ST, TSC) obvodu zabezpečeného bezpečnostním kódem. Druhá kopie benchmarku je využívána pouze jako referenční zdroj správných výsledků.

Jako sledovaný bezpečnostní kód byla použita sudá parita, výsledné navýšení zabrané plochy v FPGA bude uvedeno ve výsledcích v tabulkách.

2.3 Klasifikace poruch

Pro klasifikaci poruch potřebujeme zjišťovat nejenom, jestli výstupní vektor tvoří kódové slovo, ale i jestli je výstupní vektor správný, takový jaký by byl, kdyby nebyla vložena porucha. Může se totiž stát, že výsledný výstupní vektor bude patřit mezi kódová slova, ale nebude správný. Definujeme 3 možné reakce na vstupní vektor v přítomnosti poruchy:

- **Žádná chyba**: výstupem testovaného obvodu s generátorem bezpečnostního kódu je kódové slovo a zároveň je výsledek správný. Chyba se tedy neprojevila
- **Detekovatelná chyba**: Výstupem je nekódové slovo. To nás informuje o tom, že v obvodu je někde porucha.
- **Nedetekovatelná chyba**: Výstupem je sice kódové slovo, ale špatné, a tak vůbec nepoznáme, že je v daném obvodu porucha.

Abychom mohli exaktně vyhodnotit atributy FS, ST a TSC, je potřeba otestovat všechny vstupní vektory, jak vyplývá z definic těchto vlastností. Musíme ovšem být schopni i rozpoznat případy, kdy je výsledek kódové slovo, ale výsledek je špatný (nedetekovatelné chyby). Toho docílíme jedině tím, že porovnáme výsledky se správnými odezvami. Pak jsme schopni vyhodnotit, kolik chyb bylo detekováno bezpečnostním kódem a kolik jich spadá do kategorie nedetekovatelných. Z těchto čísel jsme schopni poruchy rozdělit do následujících kategorií pojmenovaných písmeny **A** až **D**:

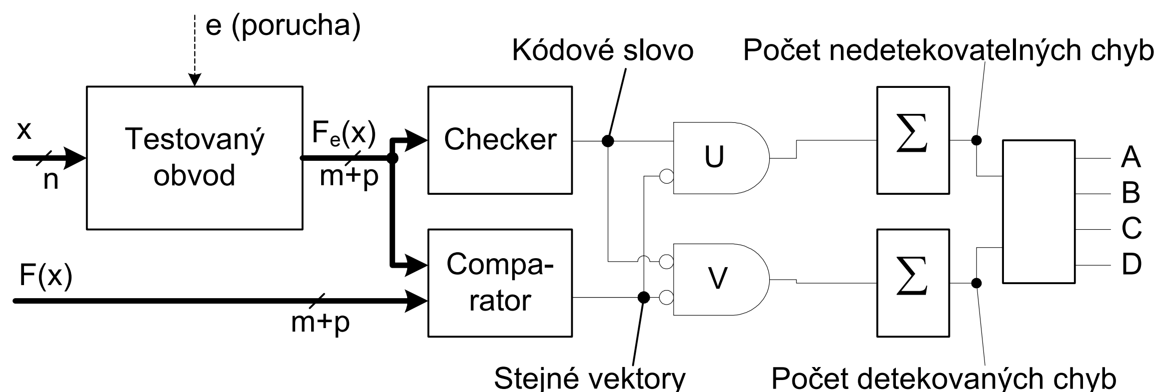
A) **Skrytá porucha** (hidden fault): Porucha nikterak neovlivnila funkci obvodu, jelikož pro všechny testovací vektory se nevyskytla ani jedna detekovatelná, ani nedetekovatelná chyba
B) **Detekovatelná porucha** (detectable fault): Porucha je detekovatelná alespoň pro 1 testovací vektor a nevyskytla se žádná nedetekovatelná chyba. V této kategorii bychom si přáli mít všechny poruchy, poněvadž pak můžebe o obvodu prohlásit, že je úplně samočinně kontrolovaný.
C) **Nedetekovatelná porucha** (undetectable fault): Porucha způsobila, že obvod sice nadále produkuje kódová slova, ale alespoň jedno kódové slovo je špatné. Znamená to, že jsme nenapočítali žádnou detekovatelnou chybu, zato nedetekovatelnou chybu alespoň jednu. Tuto poruchu nejsme schopni detekovat daným zabezpečením kódem
D) **Částečně detekovatelná porucha** (temporary detectable fault): Porucha způsobila, že projevila chybami, přičemž některé chyby byly detekovatelné a některé detekovatelné nebyly.

Těmito kategoriemi jsme schopni klasifikovat každou poruchu. Otestujeme-li všechny poruchy, tak můžeme z celkové statistiky kategorií všech poruch určit platnost vlastností (ST, FS, TSC) obvodu:

- Obvod je odolný proti poruchám (FS) pouze pokud všechny poruchy spadají do kategorie A nebo B
- Obvod je samočinně testovaný (ST) pouze pokud všechny poruchy spadají do kategire B nebo D
- Jelikož obvod je úplně samočinně kontrolovaný (TSC) pouze když jsou splněny obě vlastnosti zároveň (FS i ST), pak platí, že obvod je úplně samočinně kontrolovaný pouze, pokud všechny poruchy spadají do kategorie B

3 Analýza

Na následujícím obrázku 3.1 je znázorněna struktura systému pro klasifikaci poruch. Tato struktura byla v podstatě zadána a byla využita pro měření spolehlivosti pomocí softwarové simulace[6].



Obrázek 3.1: Struktura měření a klasifikace poruch

3.1 Proč testovat v hardwaru

Pokud používáme softwarovou simulaci, tak musíme pro každý testovací vektor počítat odezvu, což u složitějších obvodů s tisíci ekvivalentními hradly je časově náročné, navíc většinou máme k dispozici pouze jeden procesor, který musí postupně (sekvenčně) počítat všechny produkty.

Pokud však nahrajeme benchmark do hradlového pole, získáme mnohem rychlejší výpočet, jelikož za jeden takt hodin získáme výsledný výstupní vektor z benchmarku. Veškeré další zpracování (porovnání se správnými výsledky, klasifikace) se dá rovněž provádět v hardwaru a velmi tak urychlit výpočet.

Při vkládání poruch navíc získáme přesné výsledky a máme zde možnost vkládat poruchy i do míst, která kopírují strukturu hradlového pole a možnost konfigurace bitstreamem, jako jsou především propojení. Tato místa jsou v SW obtížněji testovatelná, neboť k jejich alokaci dochází až po rozmístění a propojení (place&route) v hradlovém poli. Proto simulací v HW máme větší možnosti v rozsahu poruch, které můžeme vkládat.

3.2 Volba hradlového pole

Jak vyplývá ze zadání, cílem bylo realizovat návrh v hradlovém poli firmy ATMEL. Ta vyrábí z hlediska struktury jedinou variantu FPGA, a to AT40K. Tato hradlová pole se od ostatních hradlových polí liší jednou výjimečností, a tou je možnost částečné dynamické rekonfigurace, a to po velmi malých částech (po bytech) [7,8,9], narozdíl od hradlových polí XILINX, které jsou rekonfigurovatelné po sloupcích [11].

Touto možností se tento typ hradlového pole stal jednoznačnou volbou pro realizaci této diplomové práce. ATMEL vyrábí variantu tohoto hradlového pole, nazvanou FPSLIC, která má navíc v sobě zabudovaný mikrořadič AVR s funkcemi přímo podporující rekonfiguraci. Na katedře je k dispozici vývojový kit ATSTK94 s tímto obvodem. Návrh si nežádal žádné další přídavné obvody a všechny potřebné obvody (budiče RS-232, krystal, konfigurační paměť, tlačítka), proto nebylo třeba navrhovat vlastní desku.

3.3 *Základní charakteristika a možnosti FPSLICu*

V této kapitole bych rád přiblížil obvod, v němž byl návrh implementován. FPSLIC [7] v sobě kombinuje dva jinak samostatně dostupné obvody:

- Hradlové pole AT40K
    - 5000 až 40000 ekvivalentních hradel, záleží na variantě obvodu
    - 2 až 18 Kb distribuované 2portové paměti typu SRAM
    - možnost dynamické rekonfigurace po velmi malých částech (nejmenší adresovatelná jednotka je 1 byte)
- Mikrokontrolér AVR
    - frekvence až 25 MHz
    - RISC architektura s výkonem blížícím se téměř 1 MIPS na 1 MHz
    - typické periferie AVR (UART, čítače)

Mezi specifika tohoto obvodu patří:

- 36KB SRAM rozdělená na instrukční paměť a paměť dat s možností posouvání hranice mezi programem a daty v rozmezí 4 až 16 KB pro data a 20 až 32 KB pro program. Tato paměť je sdílená s FPGA
- možnost rekonfigurace z AVR
- Přímá datová sběrnice mezi AVR a FPGA s až 16 adresami
- Až 16 přerušení do AVR

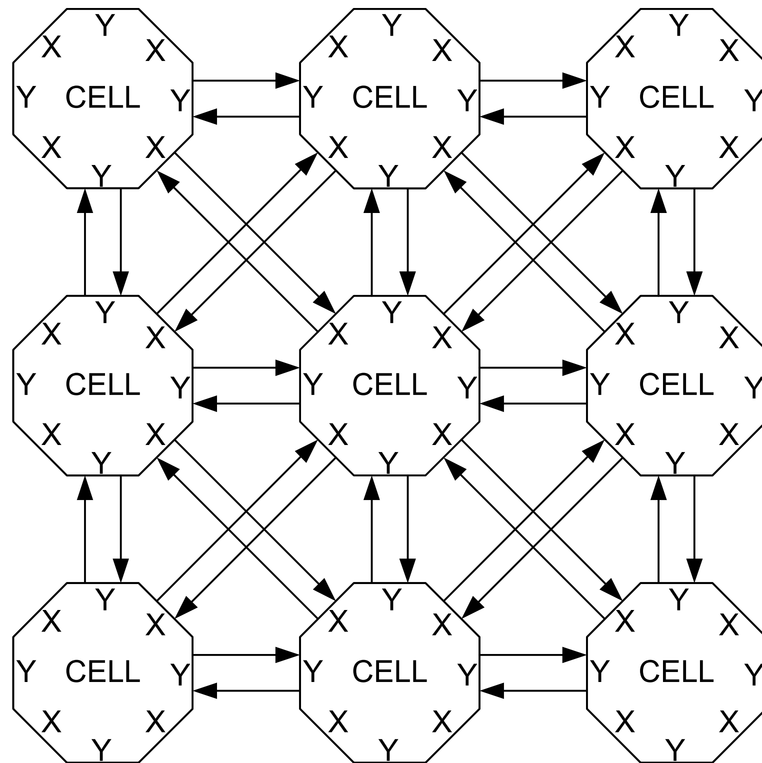
3.3.1 Bitstream FPSLICu

Abychom mohli provádět rekonfiguraci v hradlovém poli FPSLIC, je zapotřebí znát jeho bitstream. V případě tohoto hradlového pole existuje totiž jedinečná příležitost měnit bitstream po nejmenších částech v podobě jednoho bytu. Při rekonfiguraci tak není potřeba nahrávat bitstream celý, stačí před každý zapsaný byte zapsat adresu (podrobnější popis bude uveden v další části), kam daný byte patří.

Place & Route nástroj FIGARO IDS (součást programu System Designer [16]) produkuje bitstreamy v několika formátech: *.hex, *.bst a *.md4. Právě posledně uvedený MD4 formát je nejsnadněji analyzovatelný a je možné ho využít při rekonfiguraci ze zabudovaného AVR, který má speciálně vyhrazená místa v paměti, kde se po zapsání 3 bytů adresy a 1 bytu dat změní naprogramování hradlového pole.

3.3.1.1 Struktura buňky FPSLICu

Vynecháme-li anomálie krajních buněk, tak mezi každými dvěma sousedními buňkami je přímé propojení pomocí dvou vodičů. Jeden slouží jako vstup, druhý jako výstup. Schematické znázornění je na obrázku 3.2. Jak je patrné na obrázku 3.4, je výstup signálu zapojen vždy a je pouze na příjemci (přijímající buňce), jestli tento signál použije. Pokud ano, záleží na tom, z které sousední buňky ho použije. Přijímat signál ze sousedních buněk je možné dokonce ze dvou najednou, platí ovšem pravidlo, že jeden musí být namapován na vstup X a jeden na vstup Y. Právě v případě signálu ze sousedních buněk ovšem s mapováním nejde hýbat. Zda signál přijde na vstup X nebo Y je natvrdo určenou polohou sousední buňky. Přímá propojení buněk ze severu, jihu, východu a západu jsou mapována na vstup Y a přímá propojení ze severovýchodu, jihovýchodu, jihozápadu, severozápadu jsou mapována na vstup X. Jedná se o nejrychlejší propojení mezi jednotlivými buňkami

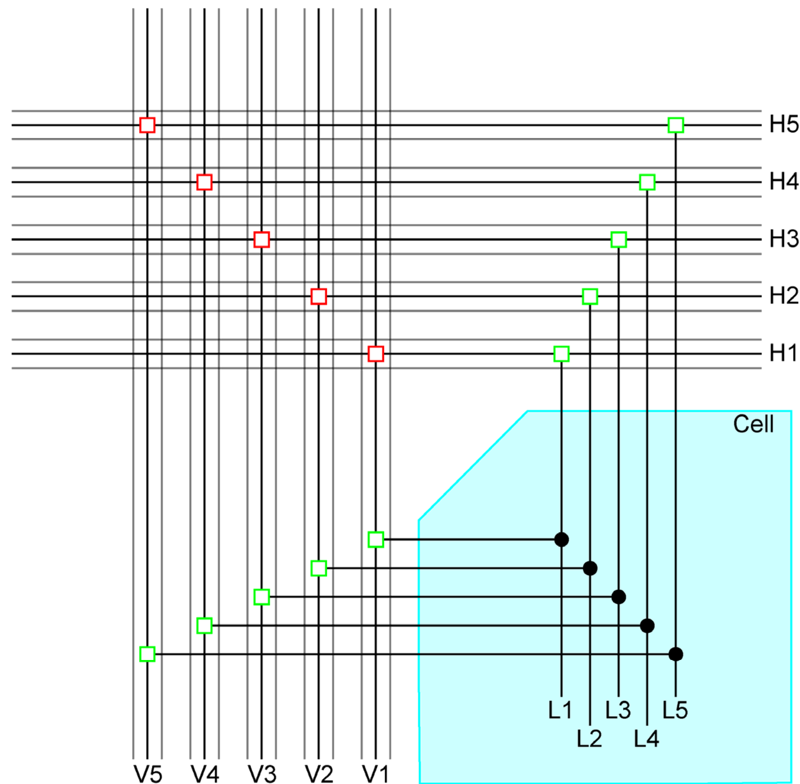


Obrázek 3.2: Propojení mezi buňkami

Kromě propojení se sousedními buňkami je každá buňka umístěna na křížení 5 horizontálních datových vodičích (v datasheetu k AT94K [7] jsou tyto vodiče nazývány local buses) a 5 svislých datových vodičích, jak je znázorněno na obrázku 3.3.

Každá buňka má 5 svých vodičů (označených L), které umožňují připojení na horizontální či vertikální datové vodiče, jak je ukázáno na obrázku 3.3

V dokumentaci od firmy ATMEL [7] je schéma znázorněno jinak. Z toho schématu se může zdát, že je možné napojit signály z buňky X, Y, W, Z a L na libovolnou kombinaci datových vodičů. Při práci s nástrojem FIGARO IDS to tak téměř vypadá, ovšem objevují se zde zdánlivě nepotopitelná omezení nemožnosti kombinovat horizontální a vertikální datové vodiče stejného indexu a podobně. Je to dáno právě tím, že struktura je ve skutečnosti jiná, než je zakresleno ve schématu originální dokumentace. Navíc na originálním schématu se prakticky nedá vyznačit význam jednotlivých bitů bitstreamu a přesně ukázat, za kterým spínačem (přenosovým hradlem) nebo multiplexorem se skrývá který bit bitstreamu, což v následujících přepracovaných schématech lze.

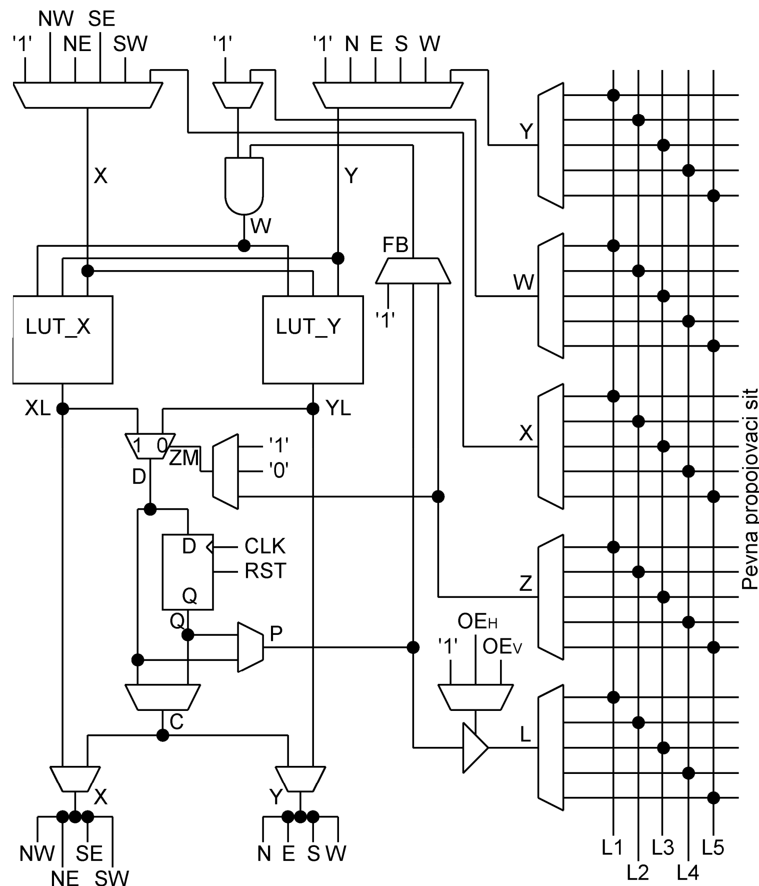


Obrázek 3.3: Propojení buňky s datovými vodiči

Pro potřeby této diplomové práce nás zajímají pouze propojení Hx s Lx, resp. Vx s Lx a propojení Hx s Vx (x je 1,2,3,4 nebo 5). Křížení horizontálních a vertikálních datových vodičů Hx s Vx (viz obrázek 3.3) jsou pouze fiktivní, tato propojení jsou vždy realizována pomocí buňky. Na obrázku 3.3 si můžeme všimnout, že vedle každého horizontálního, resp. vertikálního vodiče jsou umístěny 2 další. Ty na svém křížení (ke každému horizontálnímu vždy jeden vertikální) programovatelné propojení mají, ovšem do těchto míst se (zatím) nepředpokládá injekce poruch. Mohlo zde totiž dojít k propojení 2 odlišně buzených vodičů (tudíž ke zkratu). Narozdíl od buněk nelze place&route nástroji zakázat, aby do testované oblasti nezasahovaly datové cesty, je zde další důvod, proč se nepředpokládá injekce poruch – mohlo by dojít k narušení řídící logiky, která se nachází na stejném čipu, ale nemůžeme zaručit, že se lokálně neobjeví na propojovacích signálech v zakázané oblasti (uzamčené pro benchmark) i signál z řídící logiky.

Na následujícím obrázku 3.4 je ukázána vnitřní struktura buňky FPSLICu. Struktura v tomto schématu se opět trochu liší od schématu v originální dokumentaci [7]. Odlišnost je pouze v propojovací síti: V originální dokumentaci je přímo znázorněno propojení na horizontální/datové vodiče. Další odchylky jsou v pojmenování vodičů. V tomto případě jsem se snažil názvy zjednoznačnit, aby se 2 vodiče nejmenovaly stejně, jak tomu v dokumentaci je.

Nejdůležitějším místem celé buňky jsou dva 3vstupové LUTy , označené LUTX a LUTY (příslušné přidané písmenko X nebo Y se řídí podle toho, který signál daný LUT budí). Oba LUTy mají vstupní vodiče X, Y a W a generují libovolnou funkci těchto 3 proměnných na signálech XL, resp. YL. Zapojení do nejčastěji používané funkce 4vstupového LUTu se děje pomocí multiplexoru ZM, který lze ovládat čtvrtým signálem Z, místo statického přiřazení některým z konfiguračních bitů, jak je to u všech ostatních multiplexorů v tomto schématu.



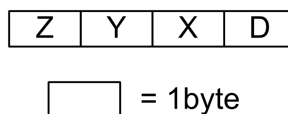
Obrázek 3.4: Struktura buňky FPSLICu

3.3.1.2 Formát bitstreamu MD4

```
001D1F01
011D1F00
021D1F00
031D1F00
041D1F01
051D1F02
061D1F00
071D1F22
081D1F6C
091D1F00
```

Obrázek 3.5: Ukázka bitstreamu ve formátu MD4

Bitstream MD4 obsahuje řádky po 8 hexadecimálních číslicích. Každá dvojice číslic reprezentuje jeden byte. V každém řádku jsou tato data uložena v pevně daném pořadí, jak je ukázáno na obrázku 3.6



Obrázek 3.6: Pořadí bytů v řádce bitstreamu MD4

Jak již bylo v předchozím textu zmíněno, tyto 4 byty se dělí na „adresní“ část (byty X, Y a Z) a datovou část (D). Adresa ovšem nemá typickou formu takovou, jak ji známe například z adresace pamětí. To bychom potřebovali konfigurační paměť 4MB. Adresní prostor bitstreamu není lineárně adresovaný. Byte Z rozlišuje typ dat, byty X a Y jsou úzce spjaty se souřadnicemi umístění na čipu.

Ve většině případů (jako jsou například LUTy, propojení uvnitř buněk, propojení buněk s okolními vodiči) tyto byty přímo odpovídají souřadnici [X,Y]. Oproti souřadnicím udávaným ve FIGARO IDS jsou ale odlišné. Zatímco ve FIGARO IDS jsou souřadnice číslovány od čísla 1 (jsou tedy v rozsahu 1 až 48), v bitstreamu jsou souřadnice číslovány od 0 (X nebo Y tedy mohou nabývat rozsahu 0 až 47)

Toto číslování ovšem pro všechny případy dat neplatí. Pro některé případy jsou totiž adresy, jako je například hodinový rozvod nebo reset, číslovány jinak. Vyplývá to ze skutečnosti, že rozvod hodin je společný pro bloky 4 buněk.

Pro potřeby této práce se všechna potřebná data nacházejí v prvně zmiňovaném systému číslování (podle souřadnic umístění buňky v hradlovém poli).

Hodnota Z nám říká, k čemu se váží souřadnice a data (např. zda se jedná o LUTy, propojení). Jak tato hodnota souvisí se strukturou FPSLICu bude vysvětleno dále.

Pokud budeme analyzovat tento bitstream, zjistíme, že pokud je některá část (místo) hradlového pole obsazená, tak se v bitstreamu na daném místě nacházejí data dvakrát. Je to dáno tím, že bitstream ve formátu MD4 se skládá ze 2 částí. V první části je nahrán bitstream, který vyčistí hradlové pole od předchozího návrhu (bezpečně nakonfiguruje do definovaného stavu). Teprve poté se nahraje bitstream s novým návrhem. To je nutné mít na paměti při analýze bitstreamu

Příklad:
V bitstreamu MD4 máme následující řádek (z ukázky bitstreamu na obrázku 3.5):

```
071D1F22
```

Jeho význam je následující: vrstva 7 (což odpovídá přesně obsahu levého LUTu generujícímu XL), souřadnice [X,Y] = [31,29] (pozor na prohození X a Y!) a konfigurační data zmiňovaného LUTu 0x22.

3.3.1.3 Obsah bitstreamu

Význam bitstreamu bohužel není veřejně přístupný a nebylo by dobré se o něm rozšiřovat. V rámci práce jsem neměl tyto informace k dispozici, musel jsem experimentálně přijít na význam sám. Z důvodu vázanosti těchto informací podpisem smlouvy NDA se omezím pouze na popis LUTů.

Konfigurační data pro obsah LUTX jsou ukryta pod hodnotou Z = 7, konfigurační data pro obsah LUTY pod hodnotou Z = 6. Hodnota D pak odpovídá hexadecimálnímu zápisu obsahu pravdivostní tabulky.

Při sestavování pravdivostní tabulky si musíme dát zvláštní pozor na to, že pro oba LUTy platí jiná permutace sloupců, než pro které se sestavuje pravdivostní tabulka. Pro levý LUTX (vrstva 7, funkce XL) platí permutace vstupů pravdivostní tabulky WYX. Pro pravý LUTY (vrstva 6, funkce YL) platí permutace vstupů pravdivostní tabulky WXY.

Příklad pro názornost:

W	Y	X	Bit	XL	Hexadecimálně
0	0	0	7.0	0	C
0	0	1	7.1	0	
0	1	0	7.2	1	
0	1	1	7.3	1	
1	0	0	7.4	1	7
1	0	1	7.5	1	
1	1	0	7.6	1	
1	1	1	7.7	0	

Tabulka 3.1: Pravdivostní tabulka signálu XL s příkladem

W	X	Y	bit	YL	Hexadecimálně
0	0	0	6.0	1	3
0	0	1	6.1	1	
0	1	0	6.2	0	
0	1	1	6.3	0	
1	0	0	6.4	0	A
1	0	1	6.5	1	
1	1	0	6.6	0	
1	1	1	6.7	1	

Tabulka 3.2: Pravdivostní tabulka signálu YL s příkladem

Ve výsledném bitstreamu budou data 0x7C pro Z = 7 a 0xA3 pro Z = 6.

Ve struktuře buňky si můžeme všimnout, že u každého multiplexoru je jeden vstup nepropojený s žádným konfiguračním bitem. Jedná se o jakousi defaultní hodnotu, často se jedná o log. 1. Všimněme si konkrétně multiplexorů vybírající signál X. Ačkoli na schématu to vypadá jako 2 multiplexory, ve skutečnosti je jenom jeden, umožňující vybírat 9 různých vodičů (vodič X vedoucí přímo ze sousední buňky – celkem 4 možné směry (symbolicky označeno podle světových stran) a pak jedna z 5 možností L1 až L5). Pokud ani jeden z těchto vodičů nebude použit, bude na vstupu X vždy 1.

Tím dostáváme odpověď na otázku, jak je používán LUT. Pokud jeden ze vstupních signálů X, Y či Z není využit, případně pokud 2 z těchto vstupních signálů nejsou využity, pak můžeme konstatovat, že na daném nezapojeném signálu je konstantní log. 1.

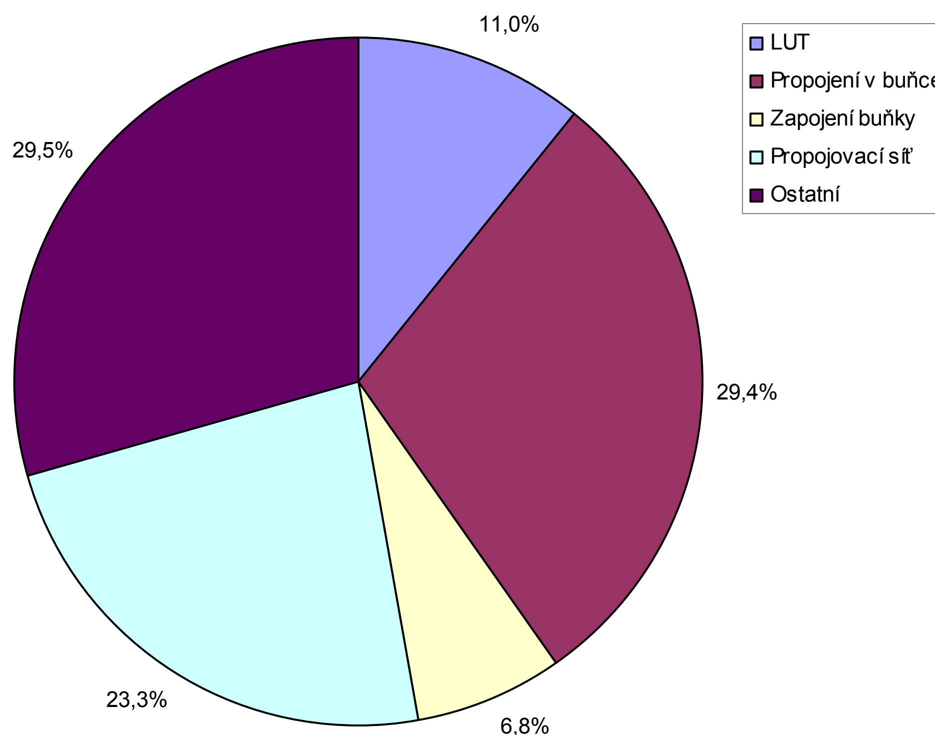
Dále se nabízí otázka: Jak vypadá bitstream buňky, pokud buňka má funkci pouze 3 nebo méně proměnných? Pak je vždy jeden LUT nevyužitý (zaplněný nulami). Nikdy se logická funkce 3 vstupních proměnných nepočítá oběma LUTy za pomoci multiplexoru ZM. Jednalo by se o pomalejší zapojení, než kdyby tuto funkci počítal pouze jeden z LUTů.

3.3.2 Odhad podílu testovaných bitů z celého bitstreamu

Tento údaj by šel spočítat exaktně, pokud bychom znali přesný počet SRAM buněk uchovávající bitstream v hradlovém poli. Ze zdrojů [AT94 a AT40] jsem získal 2 odlišné hodnoty. Pro hradlové pole AT40K40AL má být velikost bitstreamu 336504 bitů, kdežto pro AT94K40AL je to 520920 bitů (týká se pouze bitstreamu hradlového pole, s programem pro

AVR je to pochopitelně více). Tato nesrovnalost je zarážející, jelikož se jedná o hradlová pole se stejnou strukturou a se stejným počtem buněk.

Obě velikosti jsou ovšem zadané jako velikosti bitstreamu, o skutečném počtu používaných SRAM buněk se volně přístupné dokumenty firmy ATMEL nezmiňují.



Obrázek 3.7: Přibližná distribuce bitstreamu ve struktuře FPSLICu

Na obrázku 3.7 je znázorněno přibližné rozdělení bitstreamu do jednotlivých funkčních částí FPSLICu, jaké byly popsány v kapitole 3.3.1. Navíc zde byla do obrázku přidána zastoupení propojovací sítě (sestávající se s propojení u každé buňky a horizontálních a vertikálních konfigurovatelných opakovačů).

Pod ostatní části spadají:

- IO PADy
- Distribuovaná paměť a její zapojení
- Rozvody hodin a resetu a jejich konfigurace

Pouze přibližný je z důvodu, že neznáme přesný počet SRAM buněk uchovávající informaci o naprogramování hradlového pole. Uvedený obrázek je odhadován pro strukturu AT40K40AL, tj. 336504 bitů. Pro AT94K40AL s 520920 bity by podíl ostatních částí bitstreamu vzrostl na 55 procent.

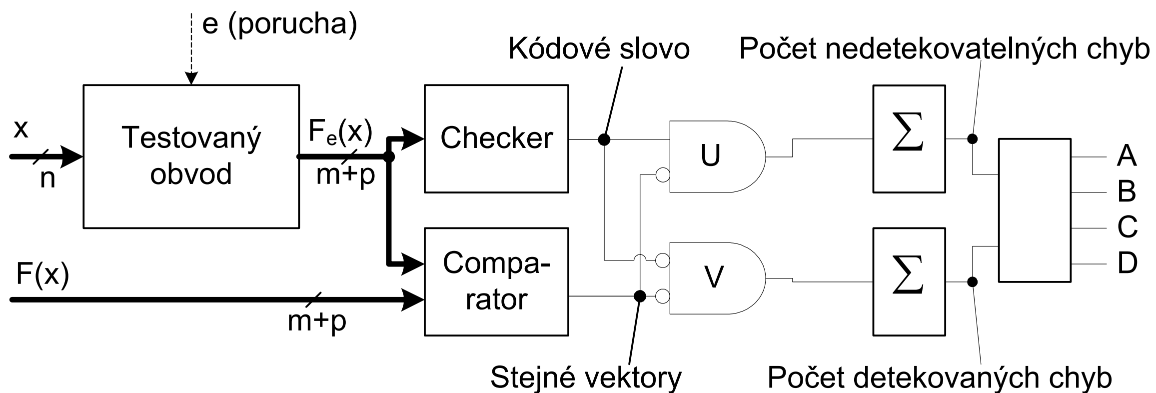
Část	bitů	podíl AT40K40 [%]	podíl AT94K40 [%]
LUT	36864	11.0	7.1
Propojení v buňce	99072	29.4	19.0
Zapojení buňky	23040	6.8	4.4
propojovací síť	78336	23.3	15.0
ostatní	99192/ 283608*	29.5	54.4*

* pro AT94K40

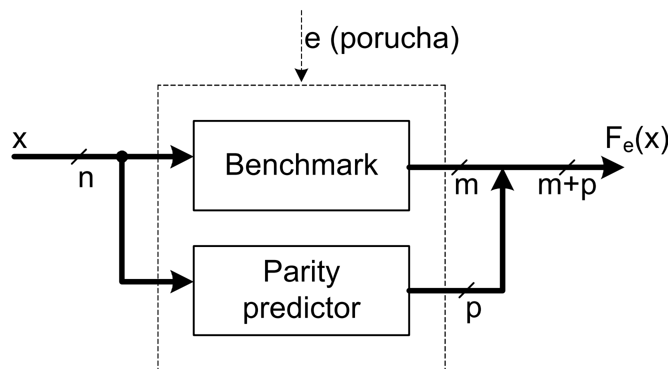
Tabulka 3.3: Přibližná distribuce bitstreamu ve struktuře FPSLICu

4 Návrh

V této kapitole bych rád zmínil, jakým způsobem jsem navrhoval celý systém testování, jakým způsobem jsem rozvrhl jednotlivé vykonávané činnosti. Návrh vychází ze struktury na obrázcích 4.1 a 4.2, která byla využita i pro softwarovou simulaci[6].



Obrázek 4.1: Struktura měření a klasifikace poruch



Obrázek 4.2: Struktura testovaného obvodu

Symboly n, m, p na obrázcích 4.1 a 4.2 mají následující význam:

- n …… počet bitů testovacího vektoru
- m …… počet bitů výstupního slova
- p …… počet paritních vodičů (či obecně počet signálů zabezpečující výstupní slovo nějakým bezpečnostním kódem)

V prvé řadě je potřeba uvědomit si, jaké prostředky máme k dispozici a co potřebuje do těchto prostředků rozvrhnout.

K dispozici máme:

- Hradlové pole
- 16 KB sdílené paměti SRAM
- I/O sběrnici a přerušovací
- AVR
- Sériovou linku
- PC

Potřebujeme rozvrhnout následující činnosti:

- generování testovacích vektorů
- kontrola kódu výsledných vektorů a porovnání výsledků s druhou kopií benchmarku
- počítání výsledné kategorie poruch
- počítání statistiky kategorií
- výběr oblastí s očekávanou injekcí poruch
- injekce poruch

4.1 Benchmark

U benchmarku je jednoznačné, že bude uložen v FPGA, jelikož sledujeme právě chování jeho umístění v hradlovém poli. Musíme ovšem vyřešit způsob porovnávání výsledků. Jako jediná rozumná volba se mi zdá vložit druhou kopii benchmarku také do hradlového pole a výsledky porovnávat přímo s ní. Jakákoli jiná varianta by vyžadovala poměrně objemnou datovou komunikaci do hradlového pole ($n \cdot m$ bitů, kde n je počet testovacích vektorů (u úplného testu 2^n) a m počet bitů výstupního vektoru), která by nutně zpomalovala testování. Navíc větších benchmarků by se výsledky nevešly do paměti FPSLICu a bylo by nutné je při každé poruše (např. 1000krát) nahrávat přes sériovou linku, což by nebylo časově únosné

LFSR pro kontrolu správnosti všech výsledných vektorů zde bohužel místo druhého benchmarku použít nejde. Sice bychom pomocí něj byli schopni detekovat, jestli byly všechny výsledné vektory správně, tím ale veškeré možnosti končí. To znamená, že bychom byli schopni rozlišit pouze kategorii A od těch ostatních (B,C,D)

Spokojíme se tedy s tím, že pro benchmark s prediktorem máme k dispozici pouze necelou polovinu hradlového pole. Druhou polovinu zabere druhá kopie a další logika (generátor testovacích vektorů, checker, komparátor a logika v komunikační vrstvě).

4.2 Generování testovacích vektorů

Zde se dostáváme k velmi závažnému problému. pro správné měření bychom totiž měli znát výsledek testování s naprostou jistotou, kterou nám zaručí pouze úplný test. Na výběr máme několik možností, jak generovat testovací vektory:

- Přímo v hradlovém poli vytvořit generátor úplného testu (není příliš náročné na plochu)
- V hradlovém poli implementovat sofistikovanější generátor testovacích vektorů, např. pomocí LSFR s tím, že se spokojíme s tím, že neotestujeme všechny kombinace vstupních vektorů
- Testovací vektory generovat v AVR

Vzhledem k tomu, že testované benchmarky měly všechny velikost vstupního vektoru menší jak 20 bitů, rozhodl jsem se implementovat generátor triviálního testu. Současně jsem ponechal rezervu pro případné nahrávání testovacích vektorů z AVR.

Nahrávání testovacích vektorů lze dělat dvěma způsoby: pomocí 8bitové I/O datové sběrnice s adresními vodiči IOSEL, nebo v FPGA navrhnout řadič paměti, který by testovací vektory přímo vyčítal ze sdílené SRAM datové paměti. Zvolil jsem komunikaci přes I/O datovou sběrnici, jelikož výsledná implementace zabírá v FPGA mnohem méně místa, než řadič paměti a hlavně tento přístup umožňuje snazší řízení z AVR.

4.3 Zpracování výsledků

Tato kapitola se týká především způsobu, jakým zpracujeme 2 výstupní vektory z obou benchmarků do výsledné kategorie poruchy.

Jak je vidět ze struktury systému testování na obrázku 3.1, checker a komparátor jsou bloky, které vyhodnocují a porovnávají výsledky na úrovni jednoduché a málo místa zabírající kombinační logiky (viz kap. 5.8 a 5.9), proto považuji za naprosto jednoznačnou volbu implementovat je v hardwaru.

Rychlejšího zpracování docílíme, pokud budeme v hradlovém poli i počítat výskyty detekovaných chyb a nedetekovatelných chyb. Z definice klasifikačních kategorií poruch na straně 4 vyplývá, že nám stačí rozlišit 2 stavy, že jsme nenapočítali žádnou chybu od stavu, že tato chyba byla alespoň jedna a nezajímá nás, jestli byla 1 nebo jich bylo 1000.

Tím se situace zjednoduší a čítač můžeme s minimální logikou umístit do FPGA, jelikož nám stačí jediný bit. Tento čítač musí být zabezpečen proti přetečení.

Dalším krokem ve zpracování výsledků je počítání statistiky kategorií poruch. Jelikož z každé poruchy dostaneme 1 výsledek (kategorie), pokládám za zbytečné tuto statistiku provádět v FPGA. museli bychom implementovat více jak 10 bitové čítače, které by byly využívány jenom jednou za dlouhou dobu testování. Proto považuji za rozumné statistiky nepočítat v hradlovém poli.

Zároveň považují za zbytečné posílat výslednou kategorii každé poruchy po sériové lince do počítače, zabralo by to mnohem více času, než počítání v AVR a pak pouze konečné poslání četnosti výskytu kategorií.

4.4 Vkládání poruch

Pro nahrání poruchy do FPGA potřebujeme znát 5 bytů dat:

- adresu poruchy (byty X, Y a Z)
- původní data D, abychom byli schopni rekonstruovat původní bitstream
- změněná data s poruchou nebo poruchový vektor

V následující úvaze předpokládám vkládání poruch pouze do LUTů. V případě, že bychom každou poruchu řídili z PC, potřebovali bychom na otestování 1 buňky $16 \cdot 5 = 80$ přenesených bytů dat, chceme-li mít i kontrolu nad tím, který byt LUTu budeme a nebudeme testovat. Tento počet by šel zredukovat zavedením vhodného formátu posílaných dat, aby se neposílala 8krát opakovaně stejná data adresy. Pokud bychom použili původní data a masku poruch a z masky bychom v AVR vyrobili příslušný počet poruch, dostali bychom se na 10 bytů.

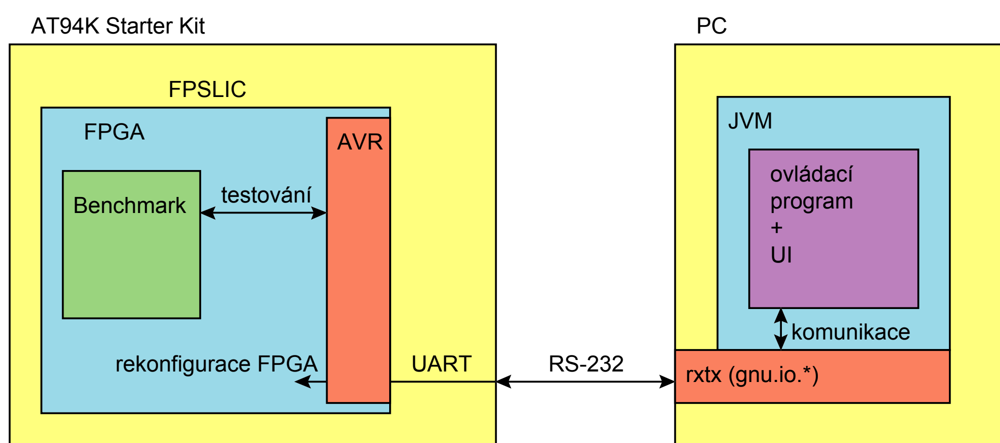
Druhou možností je do AVR nakopírovat část bitstreamu do paměti AVR a z tohoto bitstreamu generovat jednotlivé poruchy. Pro každou buňku je to postačujících 11 bytů (2 byty souřadnice a 9 bytů dat). Tento počet by se dal snížit dokonce na 4 byty, pokud bychom uvažovali pouze LUTy, přišli bychom ale o možnost analýzy zapojení LUTu a větší možnosti při vkládání poruch.

Zvolil jsem druhou možnost, jelikož dává větší možnosti testování (odhalování buněk použitých jako propojovací buňky, testování pouze využitých částí LUTů) a do budoucna umožňuje i vkládání poruch do dalších konfiguračních bitů buňky.

5 Popis implementace v FPSLICu

Jelikož jde o komplexní systém testování, výsledná implementace se dotýká všech oblastí, které bylo potřeba implementovat. První je implementace v FPGA, ve kterém se provádí samotné testování. Dalšími oblastmi jsou: implementace komunikačního rozhraní do vestavěného AVR v FPSLICu, provádění injekci poruch, vyčítání výsledků z FPGA a komunikaci s FPGA.

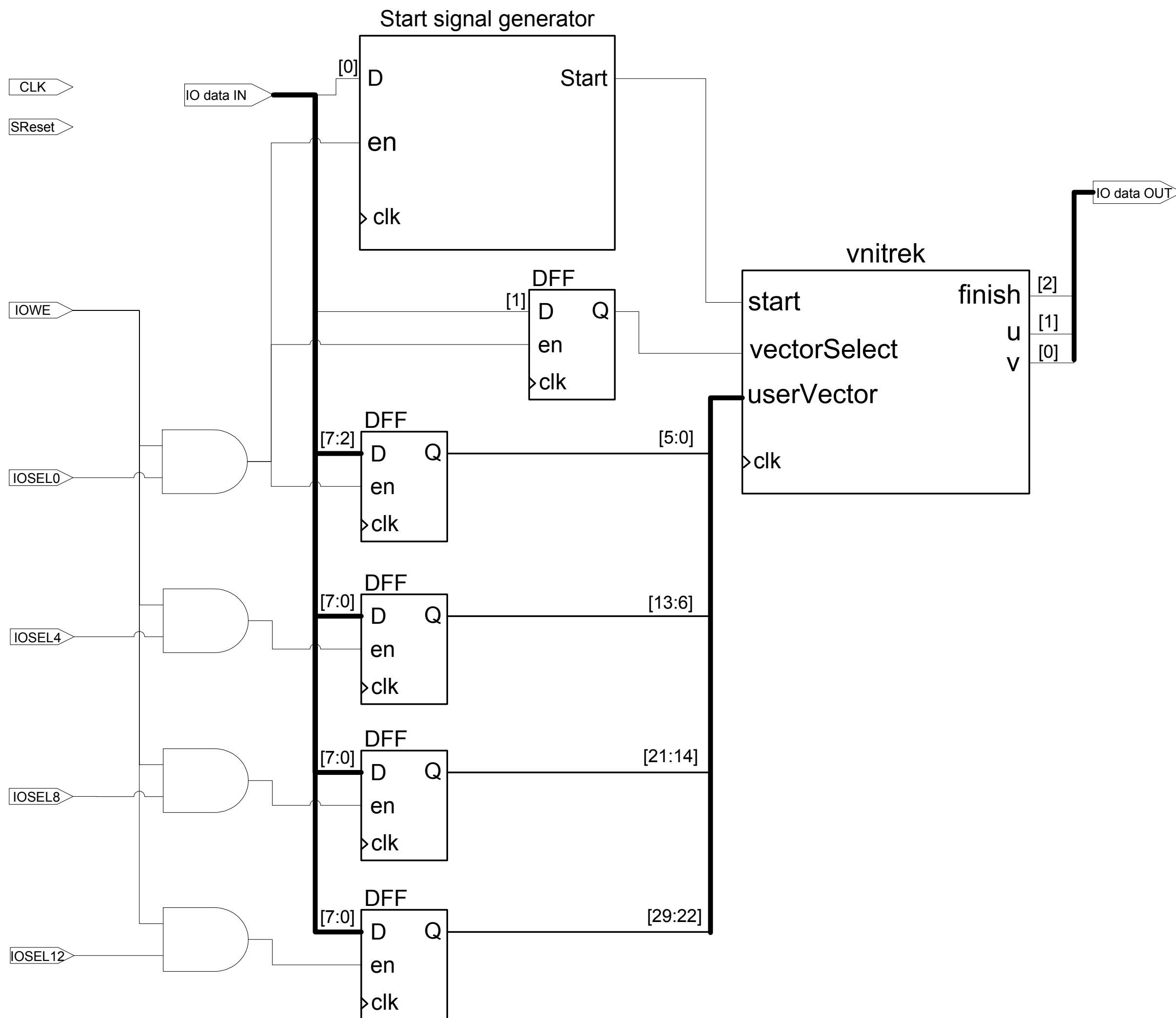
Samostatnou kapitolu jsem vyčlenil pro softwarové nástroje, které generují VHDL kód pro implementaci v FPSLIC a další nástroje, které vytvářejí bitstream. Nechybí zde ani popis ovládacího program v PC, který ovládá celou vývojovou desku s FPSLICem, nahrává poruchy a zobrazuje výsledky testování. Pomocí grafického rozhraní rovněž umožňuje vybrat lokality injekce poruch.



Obrázek 5.1: Celkový pohled na implementaci

Na vývojové desce se nachází několik možností použití hodinového signálu. Jak již bylo řečeno, byl použit 4MHz krystal. Hodinový rozvod je společný jak pro AVR, tak pro FPGA, což odpovídá konfiguraci přepínačů JP17 v pozici 2–3 (čímž dojde k propojení výstupu z 4MHz oscilátoru a hodinového vstupu XTAL1_138) a JP18 v rozpojeném stavu.

5.1 Logika zajišťující komunikaci FPGA-AVR



Obrázek 5.2: Top-level design FPGA

Na obrázku 5.2 je schéma návrhu řídící logiky, která má za úkol zprostředkovat komunikaci s AVR částí FPSLICu. Způsobů komunikace s AVR je více (kromě komunikace přes I/O sběrnici je možné komunikovat přes sdílenou paměť SRAM a dále je možné využívat signálů přerušení).

Smyslem modulu FPGA je vytvořit nezávislou vrstvu pro samotné jádro návrhu testování benchmark. S touto vrstvou pak vnitřek komunikuje následujícími signály:

Vstupy:

- `Start` (zahájení testování)
- `VectorSelect` (signál, který určuje, odkud se budou brát testovací vektory, zda z vestavěného generátoru testů, či z externě přivedeného vektoru `UserVector`)
- `UserVector` (testovací vektor)

Výstupy:

- `Finish` (signalizuje ukončení testování, tzn. že již byly generovány všechny testovací vektory)

- u, v (signály klasifikující daný benchmark)

Vrstva komunikující s AVR využívá ke komunikaci AVR-FPGA následující signály:

- IOdataIN (8bitová datová sběrnice pro data směrem z AVR do FPGA. Platná data jsou na této sběrnici definována aktivním signálem IOWE a aktivním příslušným adresním vodičem IOSEL)
- IOdataOUT (8bitová datová sběrnice pro data z FPGA do AVR. Informaci o čtení dat nám dává signál IORE, který navíc zapojuje budiče dat na společnou datovou sběrnici, která (je společná pro čtení i zápis a proto nelze číst i zapisovat zároveň. Při čtení je opět možné vybírat zdroje pomocí signálů IOSEL, v tomto případě to ale není nutné, poněvadž zpět do AVR se přenáší pouze 3 bity.
- IOWE (signál indikující zápis na datové sběrnici z AVR do FPGA)
- IOSEL0..15 (adresní vodiče adresující data na I/O datové sběrnici)

5.2 Mapování I/O prostoru v FPGA

Z celkových 16 adresních vodičů jsou použity jen 4 (což ovšem nezabraňuje budoucímu využití všech 16, pokud by to bylo potřeba). Použité signály jsou IOSEL0, IOSEL4, IOSEL8 a IOSEL12. Tento výběr adres má specifický důvod, je jím rychlejší komunikace v AVR bez nutnosti překonfigurovávat registr FISCR (pokud chceme přepínat mezi těmito 4 adresami, tak stačí pouze jednou nakonfigurovat konfigurační registr FISCR, který do paměti namapuje patřičnou čtveřici adresních vodičů, podrobněji viz [7].

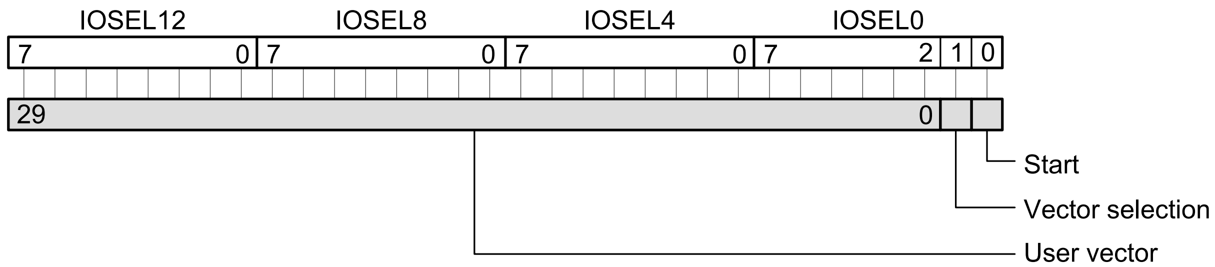
Pro každý adresní vodič (IOSELx) je logickým součinem vytvořen signál enable, který povoluje zápis do příslušných registrů, jak je vidět na obrázku 5.2

V níže uvedené tabulce 5.1 jsou vypsána přiřazení všech signálů:

Signál	Bity sběrnice IOdataIN	Adresa
Start	0. bit	IOSEL0
VectorSelect	1. bit	IOSEL0
UserVector[5..0]	[7..2]	IOSEL0
UserVector[13..6]	[7..0]	IOSEL4
UserVector[21..14]	[7..0]	IOSEL8
UserVector[29..22]	[7..0]	IOSEL12

Tabulka 5.1: Mapování vstupů I/O prostoru v FPGA

Na následujícím obrázku 5.3 je schematicky znázorněno mapování signálů start, vectorSelect a testovacího vektoru UserVector.



Obrázek 5.3: Mapování vstupů I/O prostoru v FPGA

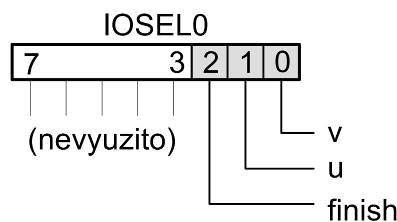
Pro výstupní signály je situace jednodušší. Sledujeme totiž pouze 3 signály (`finish`, `u` a `v`), takže nepotřebujeme signály přepínat více adresními vodiči. Všechny signály se totiž vejdou najednou do 8bitové sběrnice `IOdataOUT`. Jelikož FPSLIC má vlastní budiče (řízené signálem IORE), kterými připojuje sběrnici `IOdataOUT` k I/O datové sběrnici, nemusíme dokonce řídit buzení této sběrnice; stačí napevno připojit signály.

V následující tabulce 5.2 lze najít, jak jsou jednotlivé výstupní signály (`finish`, `u` a `v`) připojeny na sběrnici `IOdataOUT`.

Signál	Bity sběrnice IOdataOUT	Adresa
v	0. bit	N/A (IOSEL0)
u	1. bit	N/A (IOSEL0)
finish	2. bit	N/A (IOSEL0)

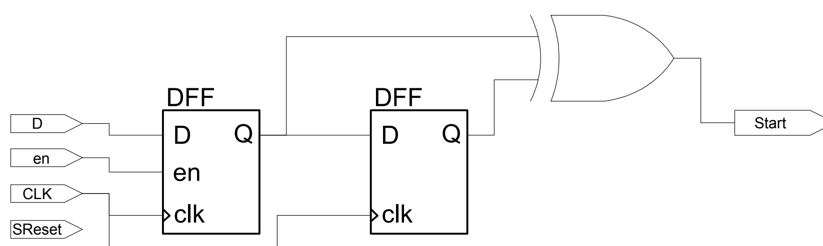
Tabulka 5.2: Mapování výstupů I/O prostoru v FPGA

Přehledněji a graficky je mapování znázorněno na následujícím obrázku 5.4



Obrázek 5.4: Mapování výstupů I/O prostoru v FPGA

5.3 Generátor signálu start

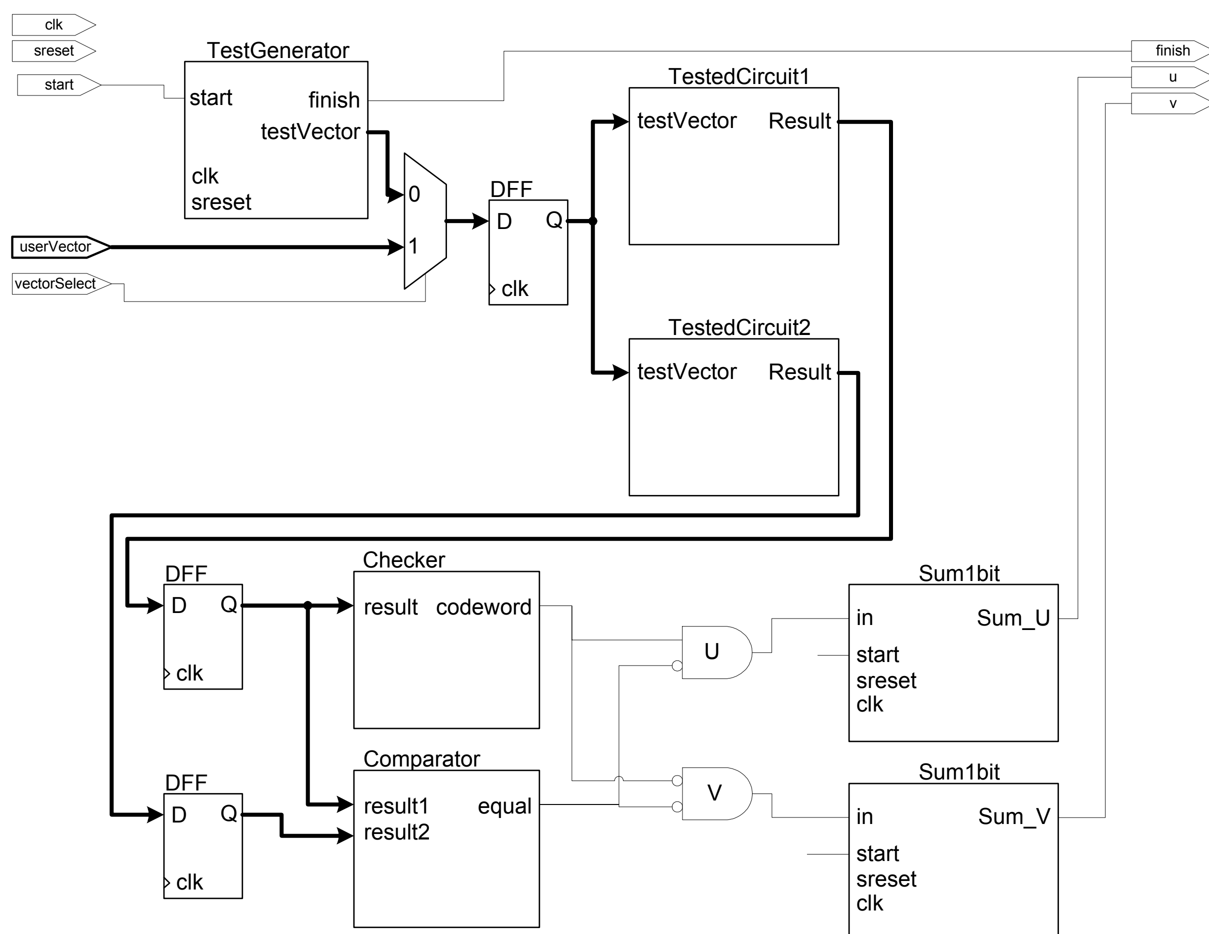


Obrázek 5.5: Generátor signálu start

Z obrázku 5.5 je patrné, že se jedná o velmi jednoduchý obvod. Jeho účelem je při každé změně vstupního signálu (D) vytvořit puls aktivní úrovně po dobu jednoho taktu hodin. Přidaný je signál en, který povoluje zápis do vstupního registru (vstup D je totiž sběrnice, která je využívána i jinými registry).

Funguje tak, že při povoleném zápisu do registru signálem en se do 1. registru s náběžnou hranou hodin zapíše nová hodnota z IOdataIN(0). V 2. registru je stále ještě uchována hodnota z předešlého taktu. Pokud se tyto 2 signály liší, vyhodnotí logická funkce XOR log. 1. V dalším taktu se do 2. registru zapíše hodnota z 1. registru a na výstupu bude opět log. 0.

5.4 Vnitřní část nezávislá na implementaci vrstvy komunikující s AVR



Obrázek 5.6: Vnitřní část nezávislá na implementaci vrstvy komunikující s AVR

Vnitřní část se skládá z několika modulů, které fungují poměrně samostatně a budou detailně popsány dále.

Vstupní signál start má hned 2 funkce:

- Zahájit generování testovacích vektorů
- Vynulovat 1bitové saturační čítače sum1bit (pro přehlednost nejsou v obrázku 5.6 zakreslena propojení signálu start do těchto modulů).

Signálem `vektorSelect` je řízen výběr zdroje testovacích vektorů pro benchmarky (0 pro testovací vektory generované vestavěným generátorem testů, 1 pro použití externího testovacího vektoru).

Nachází se zde 2 kopie benchmarku. První, `TestedCircuit1`, slouží jako testovací, do které se injektují poruchy. Druhá, `TesteCircuit2`, slouží jako referenční, která nám poskytne vždy správný výsledek, tedy žádné poruchy se do této kopie neinjektují.

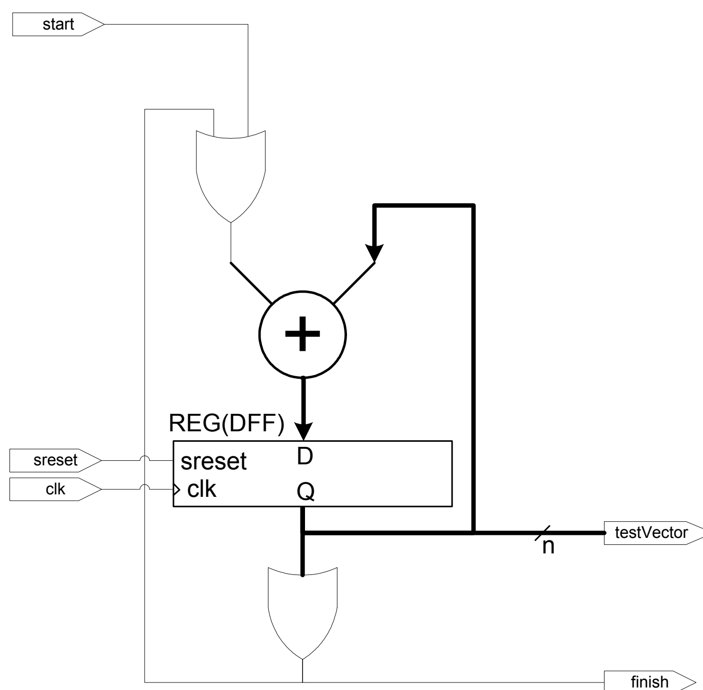
Moduly `Checker` a `comparator` generují informace, zda prvním testovaným obvodem (`TestedCircuit1`) bylo generováno kódové slovo a zda jsou výsledky z obou kopií benchmarku stejné.

Výsledky jsou zakódovány do signálů `u` a `v`, které jsou dále sčítány v modulech `Sum1bit`. Podrobnější funkce a struktura `Sum1bit` bude probrána dále.

5.5 Generátor testovacích vektorů

Současně implementovaná verze generátoru testovacích vektorů je triviální (generování úplného testu), což nám dává reálné omezení na počet vstupních bitů (přes 20 bitů už začíná být testování časově náročné). Vhledem k charakteru generování testovacích vektorů je doba generování všech kombinací exponenciálně závislá na počtu bitů testovacích vektorů.

Pro větší benchmarky by bylo zapotřebí implementovat jiný způsob generování testů, například pomocí LFSR. Vzhledem k tomu, že generátor testovacích vektorů je implementován jako samostatná komponenta, nebylo by složité jiný generátor testů zakomponovat do současné implementace, který by tak nahradil implementovaný generátor generátoru úplného testu.



Obrázek 5.7: Generátor testovacích vektorů

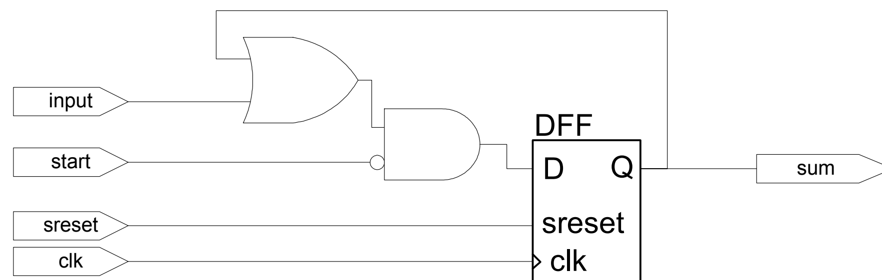
Na obrázku 5.7 je schematicky znázorněna struktura generátoru testovacích vektorů. Skládá se ze sčítačky, která inkrementuje testovací vektor o 1, dokud výsledný vektor nebudou samé nuly. Opětovné spuštění generování testovacích vektorů je zajištěno signálem start, který je po dobu jednoho hodinového taktu v log. 1, přičemž tato jednička projde až do sčítačky.

Z testovacího vektoru je navíc generován signál finish, který je aktivní v log. 0, tj. během výpočtu (generování testovacích vektorů) je jeho hodnota 1 a po ukončení generování testovacích vektorů je jeho hodnota 0, což je signál pro AVR, že může vyčíst výsledek a po vracení bitstreamu do původní podoby nahrát další poruchu a pokračovat v testování.

Signál finish je aktivní v log. 0 z důvodů případného využití přerušovacího signálu do AVR. Zde je přerušení generováno při sestupné hraně signálu (a to tak, aby sestupná hrana signálu finish byla právě při ukončení generování testovacího vektoru, což je žádoucí)

5.6 Saturační jednobitový čítač

Pro zjištění, do které kategorie patří testovaný obvod, potřebujeme vědět, kolikrát byly aktivní signály `u` a `v`. Naštěstí nám stačí pouze informace, zda počet případů, kdy byl aktivní daný signál, je větší jak 0 ($u > 0$ či $u = 0$, resp. $v > 0$ či $v = 0$). Proto si můžeme dovolit celý čítač redukovat na minimum, jak je vidět na obrázku 5.8.



Obrázek 5.8: Jednobitový saturační čítač

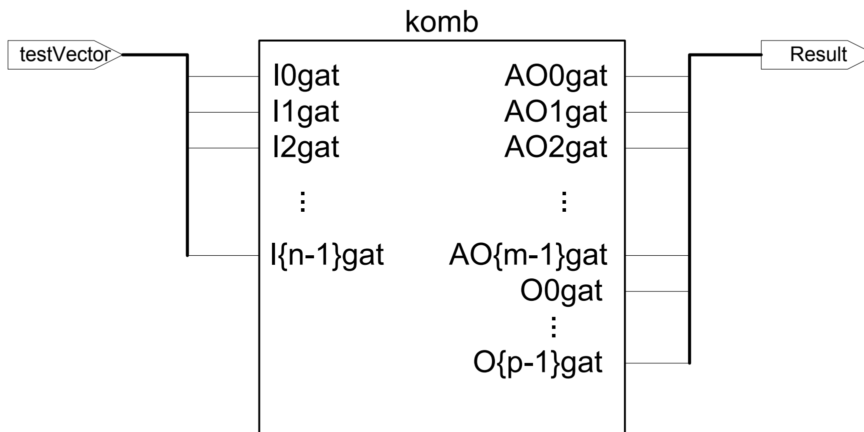
Čítač je nulován signálem `start` a na log. 1 je nastaven při výskytu log. 1 na vstupu `input`. Díky kladné zpětné vazbě se tato log. 1 bude držet, dokud nebude čítač nulován signálem `start`.

5.7 Testované obvody

Moduly označené TestedCircuit1 a TestedCircuit2 (jejich umístění vystihuje obrázek 5.6) v sobě obsahují souhrnný modul označený komb. Důvod, proč se mezi modul komb a modul vnitrek vkládají moduly TestedCircuid je především kvůli snadnějšímu generování VHDL kódu, jelikož modul komb neobsahuje signály typu std_logic_vector, ale každý signál je vyveden na samostatný port jako std_logic, což znesnadňuje parametrizaci pomocí generických parametrů. Moduly TestedCircuit tak vlastně fungují jako převodník typu signálů, jak je vidět na obrázku 5.9, kde jsou vnější porty `TestVector` a `Result` typu `std_logic_vector`. Je zde zakresleno jejich rozdělení do modulu komb.

Písmena n, m, p mají v obrázku 5.9 následující význam:

- n …… počet bitů testovacího vektoru
- m …… počet bitů výstupního slova
- p …… počet paritních vodičů (či obecně počet signálů zabezpečující výstupní slovo nějakým bezpečnostním kódem)



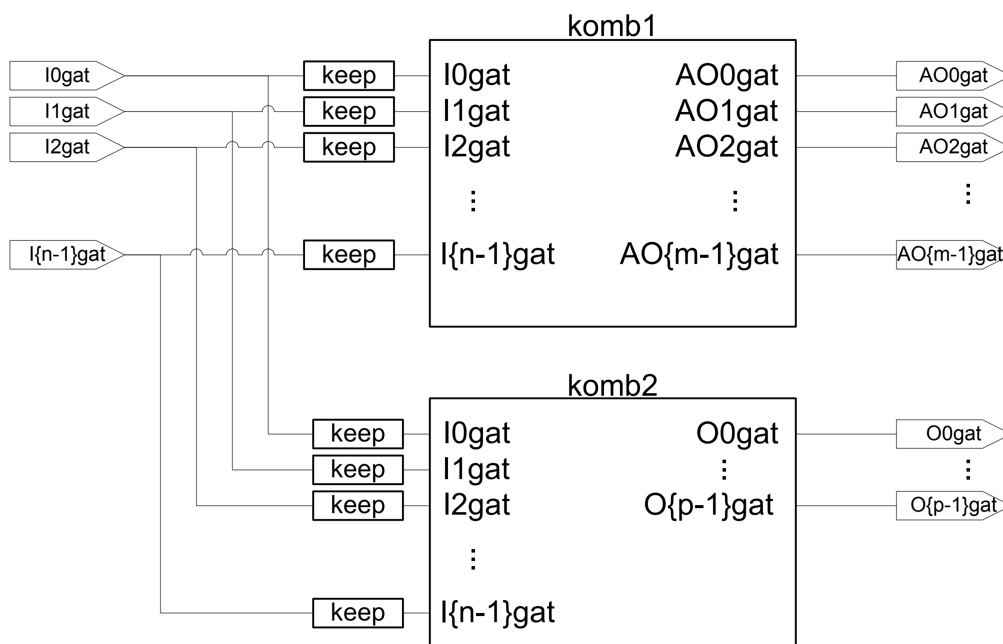
Obrázek 5.9: Modul TestedCircuit

Modul komb se skládá ze 2 částí:

- komb1 …… benchmark
- komb2 …… prediktor bezpečnostního kódu

Vstupy jsou pro oba moduly stejné (`I0gat` až `I{n-1}gat`, kde n je počet vstupů). Výstupy z komb1 (benchmarku) jsou označeny `AO0gat` až `AO{m-1}gat`, kde m je počet bitů výstupního výstupů. Výstupy z komb2 (prediktoru kódu) jsou označeny `O0gat` až `O{p-1}gat`, kde p je počet bitů bezpečnostního kódu.

Požadujeme-li, aby syntéza modulu komb1 a modulu komb2 probíhala odděleně a aby syntézní nástroj Synplify Pro [21] neslučoval oba moduly dohromady a tak nevytvářel signály, které by byly společné pro oba moduly, je zapotřebí na vstupní signály aplikovat atribut `syn_keep`, stejným způsobem, jakým je to ukázáno v kapitole 5.8.2 u hlídače parity. Na obrázku 5.10 jsou takto upravené signály označené názvem keep.



Obrázek 5.10: Separace modulů komb1 a komb2

5.8 Hlídač kódu

Modul Checker je obecně koncipován jako samostatný modul, který má na vstupech kódové slovo (obecně v libovolném kódu) a na výstupu indikuje, jestli se skutečně jedná o kódové slovo. Měření byla prováděna se sudou paritou.

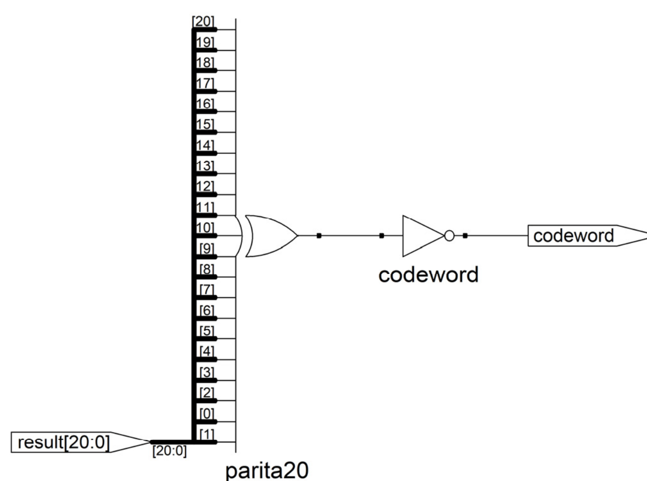
Zabezpečení paritou je nejjednodušší způsob zabezpečení kódu, mající pouze detekční charakter, který nás informuje o jednoduchých chybách, ale nedokáže je opravit, stejně jako nedokáže odhalit vícenásobné chyby. V tomto případě navíc opravovat chyby ani nelze, jelikož samoopravné kódy předpokládají chyby ve výsledném slově (například při příjmu signálu, chybě na sběrnici, atd.). Zde neplatí předpoklad, že chyby budou jednobitové (příp. tolikabitové, kolik je schopný samoopravný kód opravit). Pakliže je porucha na začátku datové cesty, může mít vliv na více bitů a opravou bychom se dostali do špatného kódového slova. Aby bylo možné opravovat, museli bychom zaručit samostatnou datovou cestu pro každý výstupní bit, což by samo o sobě vedlo k zvýšení zabrané plochy, natož pak logika zajišťující opravu.

Hlídač parity je blok, který kontroluje, jestli je na vstupních vodičích správná parita, resp. zda je na vstupních signálech kódové slovo. Výstupem je jeden signál, který říká, zda se jedná o kódové slovo (log. 1), nebo nejedná (log. 0)

Výstupní bit (zda se jedná o kódové slovo) pro *n* vstupů lze vyjádřit logickou funkcí:

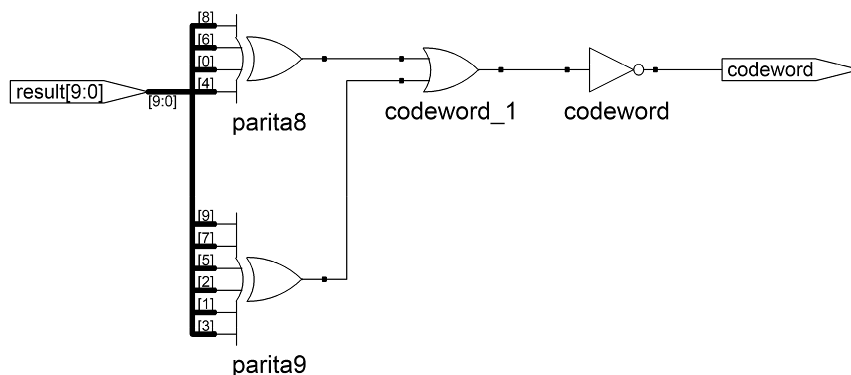
$$codeword = \overline{in_0 \oplus in_1 \oplus in_2 \oplus \ldots \oplus in_{n-1}} \text{ pro sudou paritu a}$$

$$codeword = n_0 \oplus in_1 \oplus in_2 \oplus \ldots \oplus in_{n-1} \text{ pro lichou paritu.}$$



Obrázek 5.11: Schéma hlídače parity

Kromě jednoduchého hlídače sudé parity byla ještě použita náhodná sudá parita rozdělená do dvou skupin (tedy i do dvou paritních vodičů). Pak pro každou skupinu se musí zkontrolovat parita zvlášť. Na závěr platí, že pokud obě parity sedí, tak se jedná o kódové slovo. Ukázka parity rozdělené do dvou skupin je na obrázku 5.12.



Obrázek 5.12: Parita rozdělená do 2 skupin

5.8.1 Analýza prostorové a časové náročnosti hlídače parity

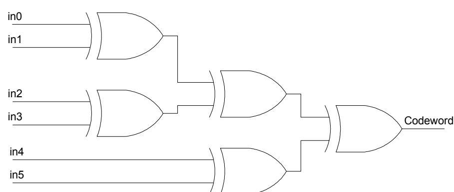
Použijeme-li základní stavební prvky (2vstupový XOR), použijeme $M = n - 1$ 2vstupových XORů na výstavbu hlídače parity o n vstupech.

Délka nejdelší cesty L ovšem závisí na způsobu (vyváženosti) XOR stromů.

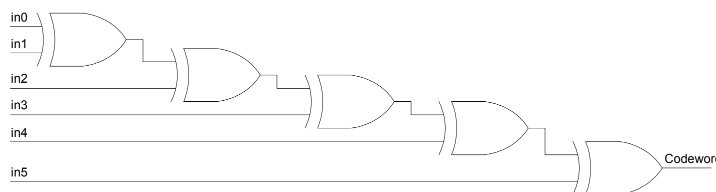
Typ stromu	Nejdelší cesta
Vyvážený strom	$L = \lceil \log_2 n \rceil$
Nevyvážený strom	$L = n - 1$

Tabulka 5.3: Nejdelší datové cesty hlídače parity pro 2vstupový XOR

Z následujících obrázků 5.13 a 5.14 je zřetelně patrné, proč se délky cest liší.



Obrázek 5.13: Optimální strom



Obrázek 5.14: Maximálně nevyvážený strom

Při návrhu do FPGA ovšem nepracujeme s 2vstupovými XORy. Základním blokem, ve kterém si můžeme realizovat logickou funkci je LUT s typicky 4 vstupy a jedním nebo 2 výstupy. V FPGA AT94K firmy ATMEL máme k dispozici 4vstupové LUTy, a proto má smysl se zajímat, kolik místa v FPGA zabere checker (počítáno právě na počet LUTů).

I zde se paritní checker implementuje paritním stromem ze 4vstupových LUTů (případně i ze 3vstupových a 2vstupových)

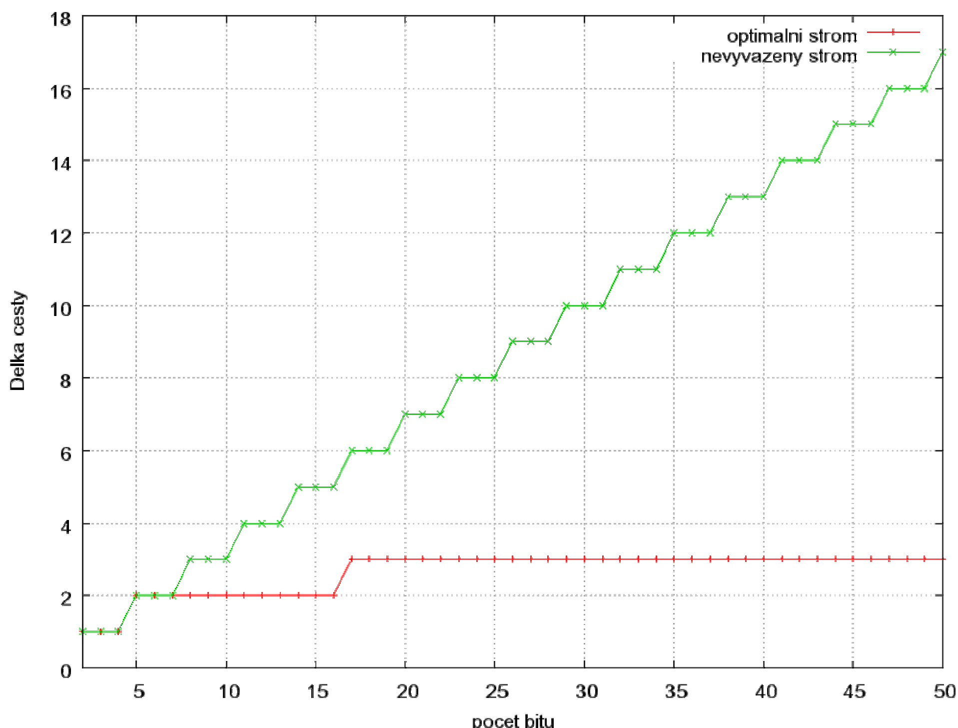
Počet použitých LUTů M hlídače parity v závislosti na počtu vstupních vodičů lze vyjádřit vzorcem:

$$M = \left\lceil \frac{n-1}{3} \right\rceil, \text{ kde n je počet vstupů.}$$

V následující tabulce 5.4 je uvedena délka nejdelší cesty L, udávající počet LUTů, kterými musí signál projít. Znázorněno grafem na obrázku 5.15.

Strom	Nejdelší cesta
Optimální strom	$L = \lceil \log_4 n \rceil$
Nevyvážený strom	$L = \left\lceil \frac{n-1}{3} \right\rceil$

Tabulka 5.4: Nejdelší datové cesty hlídače parity pro 4vstupové LUTy

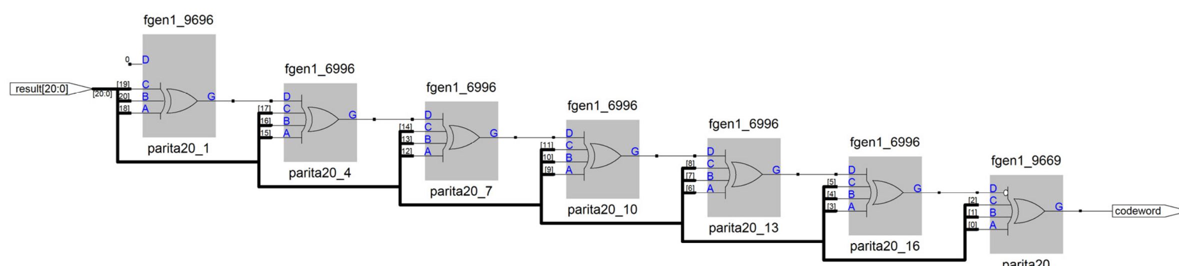


Obrázek 5.15: Graf závislosti délky datové cesty na počtu bitů vstupního vektoru

5.8.2 Implementace a dosažení optimální syntézy hlídače parity

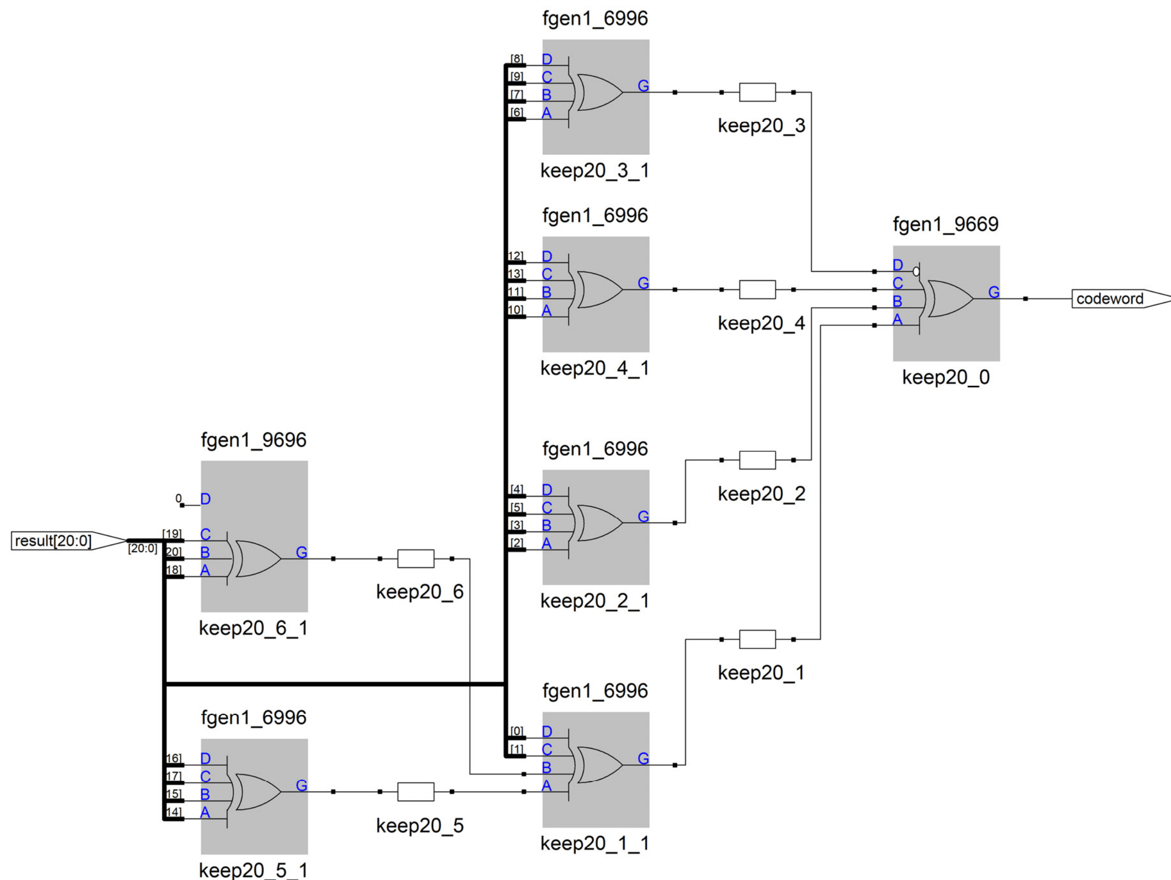
Při syntéze je potřeba si ohlídat, jak vypadá výsledek mapování na hradlové pole při použití syntézního nástroje Synplify Pro [21].

Na obrázku 5.16 je ukázka, jakým způsobem syntetizuje Synplify – strom je maximálně nevyvážený (hloubky jednotlivých listů (= vstupů) se liší o $n-1$). Takto syntéza dopadne, i když ve VHDL jsou signály napsány tak, že vytvářejí vyvážený paritní strom. RTL pohled syntetizovaného modulu je na obrázku 5.11.



Obrázek 5.16: Paritní strom – standardní syntéza (technology view)

Na následujícím obrázku 5.17 je již strom o poznání lepší, jelikož hloubka jednotlivých listů XOR-stromu se liší maximálně o 1.



Obrázek 5.17: Paritní strom s minimální hloubkou (technology view)

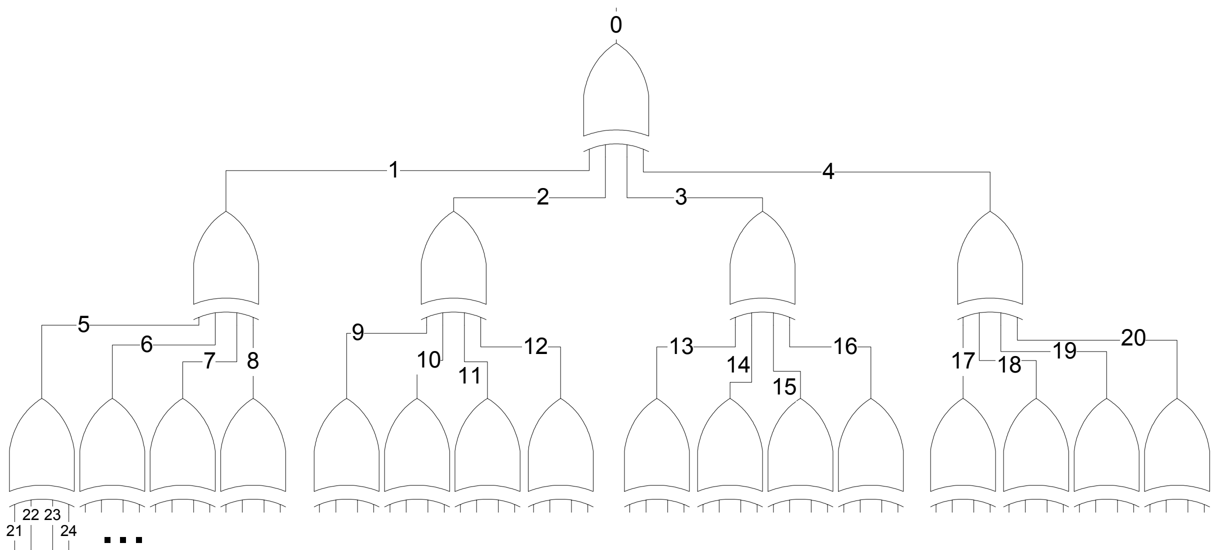
K syntéze do stromu na obrázku 5.17 bylo použito atributu `syn_keep` ve VHDL kódu. Tento atribut je specifický pro syntézní nástroj Synplify a umožňuje nastavit, které signály mají být při syntéze zachovány. Jednoduchý příklad signálu `keep`:

```
Architecture arch of checker is
…
signal keep:std_logic; -- signál, který chceme zachovat
                       -- (syntéza ho nezruší)
Attribute syn_keep: boolean;
Attribute syn_keep of keep:signal is true;
…
begin
end arch;
```

VHDL kód modulu checker je generován funkcí `writeOptimalParityChecker()` v programu GenVHDL. Postup vytvoření optimálního paritního stromu (jako je na obrázku 5.17) vypadá následovně:

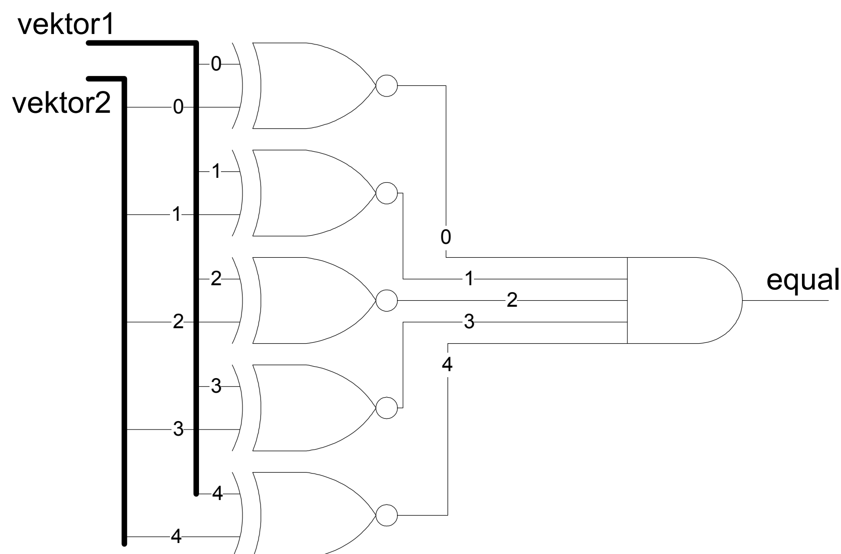
1. Vytvoření signálů `keep` (je jich celkem $\left\lceil \frac{n-1}{3} \right\rceil$), nastavení atributů `syn_keep`.

   Jednotlivé vodiče `keep` jsou číslovány podle pořadí navštívení uzlů (4vstupových XORů) při procházení paritního stromu do šířky (z kořene, tj. z výstupního signálu, který říká, jestli se jedná nebo nejedná o kódové slovo). Číslování je znázorněno na obrázku 5.18. Tím, že máme dané jednoznačné očíslování, můžeme říci, jaká budou čísla (indexy) větví (vstupních signálů do XORu), tj. $4i+1$, $4i+2$, $4i+3$ a $4i+4$.
2. všem signálům `keep` s indexem i se přiřadí exkluzivní součet (XOR) 4 signálů keep s indexy $4i+1$, $4i+2$, $4i+3$ a $4i+4$. Pokud je některý z indexů mimo rozsah signálů `keep`, použije se první nepoužitý vstupní signál. Pokud nezbyl žádný vstupní signál, nepoužije se žádný (a XOR bude mít méně jak 4 vstupy)



Obrázek 5.18: Číslování vodičů v paritním stromu

5.9 Komparátor



Obrázek 5.19: Struktura komparátoru

Komparátor je blok, který porovnává 2 vektory vstupních dat. Výstupem je signál, který nás informuje o tom, jestli jsou vektory shodné (v tom případě je na výstupu log. 1), nebo jestli se liší byť jen v jediném bitu (v tom případě je na výstupu log. 0).

Výstup komparátoru pro 2 vstupní vektory (*r, s*) o n bitech můžeme vyjádřit logickou funkcí:

$$equal = \overline{r_0 \oplus s_0} \cdot \overline{r_1 \oplus s_1} \cdot \overline{r_2 \oplus s_2} \cdot \ldots \cdot \overline{r_{n-1} \oplus s_{n-1}} .$$

5.9.1 Analýza prostorové a časové náročnosti komparátoru

Použijeme-li základních stavebních prvků (2vstupový XOR či XNOR, 2vstupový AND), budeme pro porovnání 2 vstupních vektorů s počtem bitů *n* potřebovat $M_1 = n$ 2vstupových XNORů (pro porovnání dvojic bitů) a $M_2 = n - 1$ 2vstupových ANDů (pro součin výstupu XNORů; místo toho by šel použít 1 n-vstupý AND).

Celková prostorová náročnost pro 2vstupové stavební prvky je

$$M = n + (n - 1) .$$

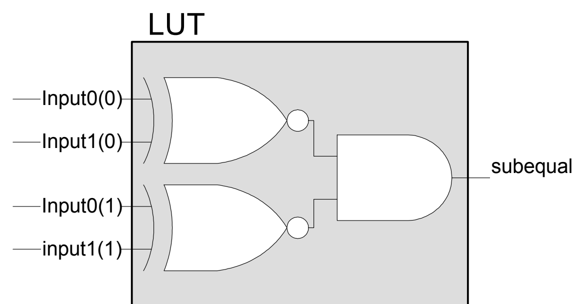
Délka nejdelší cesty L je podobná jako u hlídače parity, závisí na způsobu (vyváženosti) stromů AND hradel. Rozdíly mezi stromy hradel AND jsou stejné jako u stromů hradel XOR na obrázcích 5.13 a 5.14.

Typ stromu	Nejdelší cesta
Optimální strom	$L = 1 + \lceil \log_2 n \rceil$
Nevyvážený strom	$L = n$

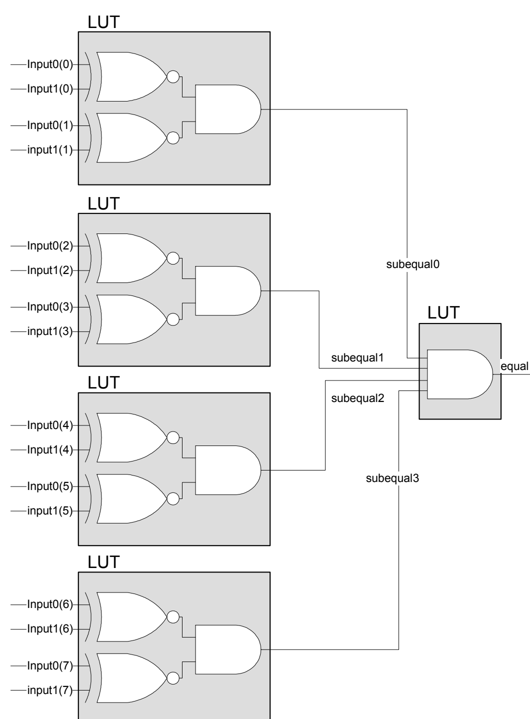
Tabulka 5.5: Nejdelší datové cesty komparátoru pro 2vstupové XNORy a ANDy

Při návrhu do FPGA ovšem nepracujeme s 2vstupovými XORy. Základním blokem, ve kterém si můžeme realizovat logickou funkci je LUT s typicky 4 vstupy a jedním, nebo 2 výstupy. V FPGA AT94K firmy ATMEL máme k dispozici 4vstupové LUTy, proto má smysl zajímat se, kolik místa v FPGA zabere komparátor, počítáno právě na počet LUTů.

Optimálně jsou využity LUTy, které počítají kombinační funkci všech 4 vstupů. Proto můžeme v LUTech porovnávat 2 bity vstupních vektorů současně, jak je znázorněno na obrázku 5.20.



Obrázek 5.20: Sdružení XORů v komparátoru



Obrázek 5.21: Schematické rozdělení komparátoru na LUTy

Celý komparátor pak bude schematicky vypadat podle obrázku 5.21. Pro výpočet počtu obsazených LUTů je možné ho rozdělit na 2 části. První část, která vyhodnocuje signály subEqual (viz obrázek 5.20), zabere $M_1 = \left\lceil \frac{n}{2} \right\rceil$ LUTů. Druhá část, která počítá logický součin (AND) všech signálů subEqual, zabere stejný počet LUTů jako checker (místo XOR je použit AND), tj. $M_2 = \left\lceil \frac{M_1 - 1}{3} \right\rceil$, tj. $M_2 = \left\lceil \frac{\left\lceil \frac{n}{2} \right\rceil - 1}{3} \right\rceil$, kde n je počet bitů porovnávaných vektorů.

Počet použitých LUTů M komparátoru v závislosti na velikosti porovnávaných vektorů (počet bitů n) lze vyjádřit vzorcem:

$$M = \left\lceil \frac{n}{2} \right\rceil + \left\lceil \frac{\left\lceil \frac{n}{2} \right\rceil - 1}{3} \right\rceil$$

kde n je počet bitů vstupních vektorů.

V následující tabulce 5.6 je uvedena délka nejdelší cesty L, udávající počet LUTů, kterými musí signál projít.

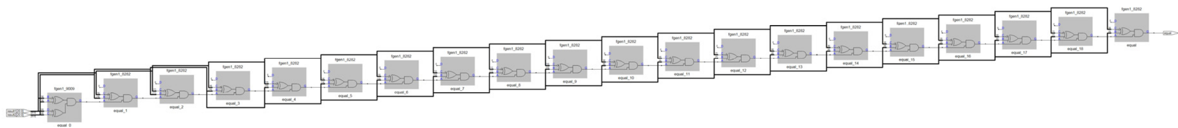
Strom	Nejdelší cesta
Optimální strom	$L = 1 + \left\lceil \log_4 \left\lceil \frac{n}{2} \right\rceil \right\rceil$
Nevyvážený strom	$L = 1 + \left\lceil \frac{\left\lceil \frac{n}{2} \right\rceil - 1}{3} \right\rceil$

Tabulka 5.6: Nejdelší datové cesty komparátoru pro 2vstupový XOR

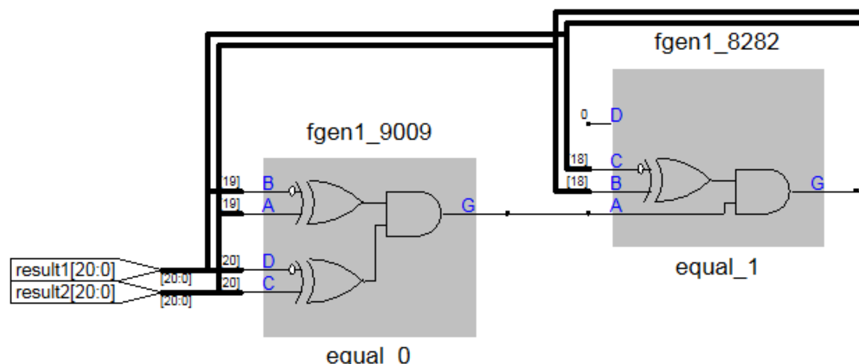
5.9.2 Implementace a dosažení optimální syntézy hlídače parity

Na rozdíl od hlídače parity má syntéza dopad nejenom na délku kritické cesty, ale i na rozlohu v FPGA (počet LUTů), proto má daleko větší smysl se touto syntézou zabývat.

Syntézní nástroj Synplify totiž syntetizuje komparátor velmi neekonomicky a není zachována struktura stromu, jak je patrné z obrázku 5.22 a v detailu na obrázku 5.23.



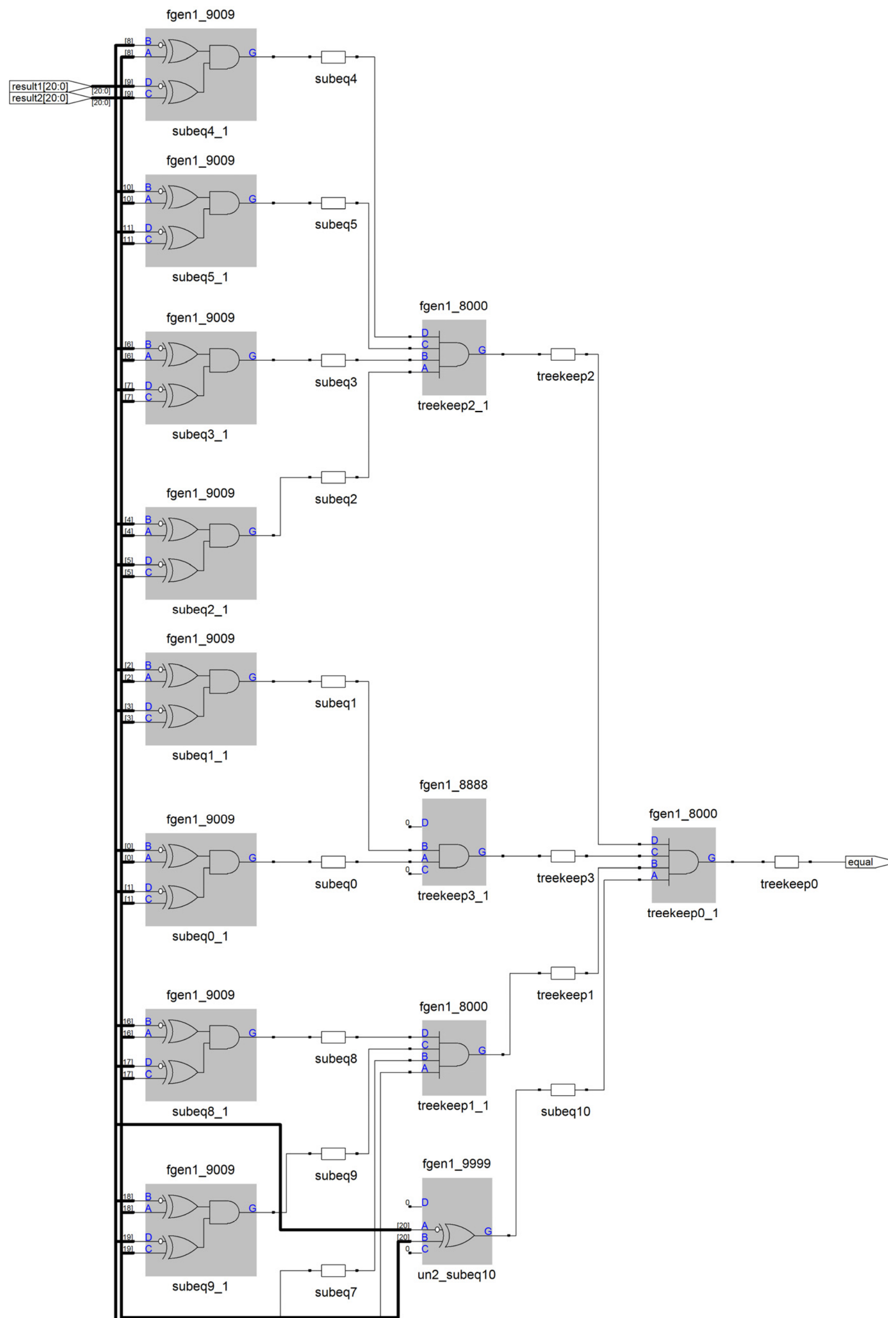
Obrázek 5.22: Nevhodná syntéza komparátoru (technology view)



Obrázek 5.23: Detail nevhodné syntézy komparátoru z předešlého obrázku

Z obrázků se dá jednoduchou úvahou odvodit závislost počtu použitých LUTů na počtu bitů vstupních vektorů. První LUT (na obrázku 5.23 vlevo) porovnává 2 bity ze vstupních vektorů. Ostatní LUTy jsou zapojeny sériově za sebe, přičemž každý LUT porovná 1 bit ze vstupních vektorů. Proto bude komparátor dvou n-bitových vektorů zabírat $M = n - 1$ LUTů.

K této neekonomické syntéze dojde i navzdory tomu, že signály přesně kopírují optimalizovanou strukturu, jaká je vidět na obrázcích 5.21. Proto i zde lze s úspěchem využít atributu `syn_keep`, jak již bylo popsáno v kapitole 5.8.2 (Implementace a dosažení optimální syntézy hlídače parity). Výsledek je pak vidět na následujícím obrázku 5.24.

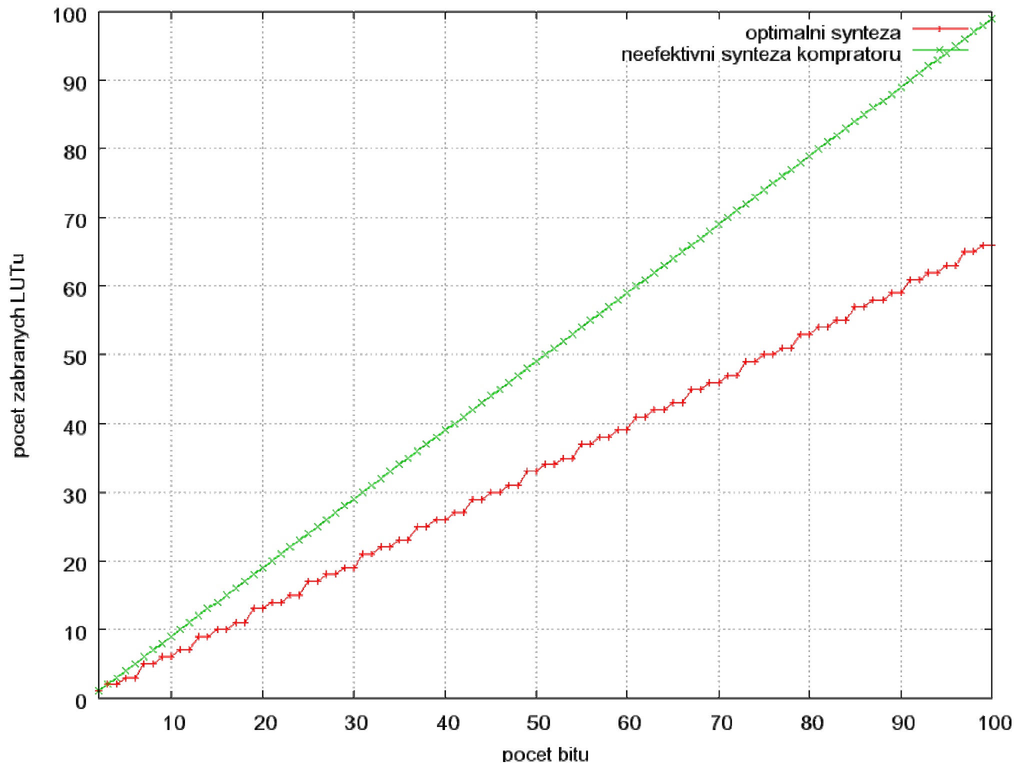


Obrázek 5.24: Optimální syntéza komparátoru (výřez, technology view)

VHDL kód modulu comparator je generován funkcí `writeOptimalComparator()` v programu GenVHDL. Postup vytvoření optimálního komparátoru je následující:

1. Vytvoření signálů `subeq`, které jsou mezivýsledkem porovnání 2 bitů ze vstupních vektorů (log. 1 odpovídá shodě, log. 0 odpovídá rozdílu). Jejich počet je $M = \left\lceil \frac{n}{2} \right\rceil$.
2. Vytvoření optimálního stromu ANDů, jak bylo popsáno v kapitole 5.8.2 (Implementace a dosažení optimální syntézy hlídače parity) s tím rozdílem, že vstupy do AND stromu jsou signály `subeq` .

Zde je na místě uvést i významnou úsporu počtu zabraných LUTů v FPGA, která se asymptoticky blíží k 33 % ($\frac{2}{3}$ velikosti výsledku neefektivní syntézy), jak je částečně vidět i z následujícího obrázku 5.25.



Obrázek 5.25: Graf plochy zabrané komparátorem v FPGA

5.10 Návrh v AVR části

Zatímco v FPGA části probíhal návrh na úrovni jazyka VHDL, pro programování AVR se používá standardní jazyk C. Pro AVR je sice také možné psát v assembleru, ovšem v tomto případě se nejedná o časově kritický návrh a výsledná nepřehlednost assemblerového kódu by práci spíše ztěžovala, navíc překladač z jazyka C, gcc [22], se k překladu staví velmi odpovědně a výsledný kód je velmi efektivní. Jádro procesoru AVR je již svojí architekturou předurčeno k psaní programů v jazyku C.

Pro překlad zdrojových kódů C byl použit populární překladač gcc [22,23]. Kromě zde používaného překladače jazyka C existuje ještě více překladačů (IAR C compiler, Codevision AVR C compiler, ImageCraft C compiler (ICC)). Gcc je poměrně robustním multiplatformním překladačem C a C++. Verze umožňující překlad zdrojových kódů pro mikrokontroléry AVR je pod windows dostupná pod názvem winavr [23]. Spolu s programem AVR studio [24] tvoří velmi kvalitní vývojové prostředí, které je navíc k dispozici zcela zdarma. Umožňuje simulaci AVR a dokonce ko-simulaci s VHDL modelem návrhu v FPGA simulovaným programem ModelSim (což vzhledem k nemožnosti simulovat poruchy nemá význam pro tuto práci).

Jako prostředí pro psaní C kódu byl využit program Elipse s modulem pro vývoj C kódu, CDT. Výhoda tohoto vývojového prostředí se skrývá ve výborných možnostech editoru, možnostech kompletace kódu a umožnění práce s moduly. Rovněž nesporná motivace pro

jeho využití byla možnost využívat jedno vývojové prostředí pro vývoj kódu pro AVR a vytváření obsluhující aplikace běžící na PC.

5.10.1 Základní konfigurace, celkový přehled návrhu

Jak již bylo uvedeno, by využit oscilátor 4 MHz, který jek dispozici na vývojové desce. Ten bohužel není příliš vhodný jako zdroj hodin pro komunikaci po sériové lince. Důvodem je, že tato frekvence není dělitelná žádnou ze standardních rychlostí přenosu po sériové lince. Proto se musíme spokojit s přibližnou frekvencí, kterou dostaneme vydělením základního kmitočtu dělitelem UBR. V tabulce 5.7 je ukázáno, jaká je reálná skutečná frekvence, kterou dostaneme vydělením základní frekvence 4MHz.

Skutečnou přenosovou rychlost, na které UART pracuje, zjistíme podle vzorce:

$$BAUD = \frac{f_{CLK}}{16 \cdot (UBR + 1)}.$$

FPSLIC ovšem umožňuje využít dvojnásobné přenosové rychlosti, jak je zaznamenáno v pravé tabulce 5.7. Výslednou přenosovou rychlost v tomto režimu vypočítáme podle následujícího vzorce:

$$BAUD_{Double} = \frac{f_{CLK}}{8 \cdot (UBR + 1)}$$

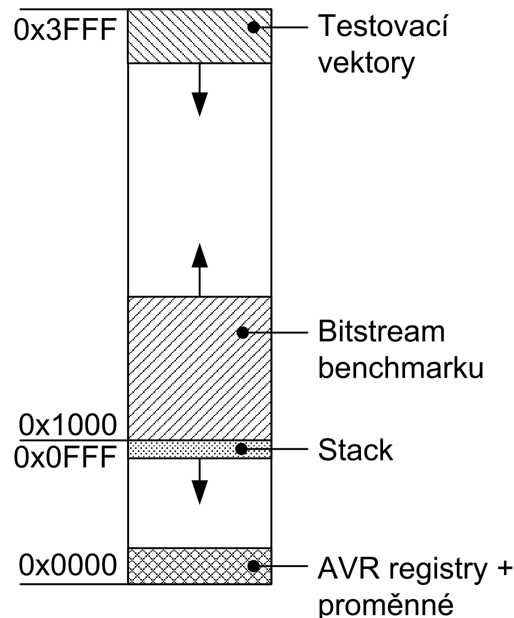
	Standart operation			Double data rate		
BaudRate	real	UBR	chyba	real	UBR	chyba
2400	2404	**103**	**0.2**	2404	**207**	**0.2**
4800	4808	**51**	**0.2**	4808	**103**	**0.2**
9600	9615	**25**	**0.2**	9615	**51**	**0.2**
14400	14706	**16**	**2.1**	14286	**34**	**0.8**
19200	19231	**12**	**0.2**	19231	**25**	**0.2**
28800	27778	**8**	**3.7**	29412	**16**	**2.1**
38400	35714	**6**	**7.5**	38462	**12**	**0.2**
57600	62500	**3**	**7.8**	55556	**8**	**3.7**
76800	83333	**2**	**7.8**	71429	**6**	**7.5**
115200	125000	**1**	**7.8**	125000	**3**	**7.8**
128000	125000	**1**	**2.4**	125000	**3**	**2.4**

Tabulka 5.7: Konfigurace a odchylky pro různé přenosové rychlosti 4MHz krystalu

Pro spolehlivou komunikaci je doporučeno, aby odchylka přenosové rychlosti byla menší jak 1%, proto byla použita nejvyšší možná rychlost s uspokojující chybou, 38400 b/s.

Zvolenou rychlost 38400 b/s je nutné nastavit v konfiguračních registrech AVR, konkrétně registr UBRR0 na hodnotu 12 a spodní 4 bity registru UBRRHI na nulovou hodnotu.

5.10.2 Organizace paměti



Obrázek 5.26: Mapa paměti AVR

FPSLIC má k dispozici 36KB paměti typu SRAM, přičemž tato paměť je rozdělena na programovou paměť (v ní je uložen kód programu) a datovou paměť. Hranice rozdělení těchto částí není pevně stanovena a dá se konfigurovat. Pro datovou paměť můžeme zvolit velikost 4 až 16KB, po 4KB krocích. Jelikož paměť bude velice zapotřebí na testovací vektory a bitstream obsahující benchmark, tak je účelné využít maximum, tj. 16KB.

Na obrázku 5.26 je schématicky znázorněno, jak je využívána a zaplňována paměť. Od adresy 0x1000 nahoru je ukládán bitstream, kam jsou injektovány poruchy. Proti bitstreamu jsou od adresy 0x3FFF směrem dolů nahrávány testovací vektory.

Paměti pro bitstream je akorát tolik, aby vystačila zhruba na injekci poruch do poloviny buňek celého FPSLICu (přesně 1117 z 2304 buněk). To je plně dostačující pro jakýkoli benchmark, který se do FPSLICu vejde, tedy zhruba do velikosti 1000 LUTů. Stačí si uvědomit, že v FPSLICu jsou 2 kopie benchmarku, tudíž více jak polovinu obvodu benchmark zabírat nemůže, poněvadž by se tam nevešel. Navíc polovina obvodu je jenom teoretická mez, u které je prakticky už problém provést routování a při maximální velikosti právě kolem 1000 LUTů se již routování nemusí podařit.

Do paměti není nahráván celý bitstream, ale pouze důležité části, které nesou informaci o konfiguraci buňky. Ostatní části (rozvody hodin, opakovače datových signálů, rozvody resetu a další) jsou vynechány pro úsporu místa a také proto, že nejsou objektem testování a ani nejsou potřeba pro analýzu obsahu buňky.

Pro ukládání bitstreamu je vyhrazená struktura `CELL` v programu main.c, která zabírá celkem 11 bytů, kterou se dynamicky zaplňuje paměť od adresy 0x1000. Pro ukládání není vyhrazena žádná struktura, jelikož počet bitů testovacích vektorů je proměnný v poměrně velkém rozsahu. Proto je pro ukládání vyhrazena pouze počáteční adresa (0x3FFF) a paměťový prostor je dynamicky alokován.

5.10.3 Komunikace s FPGA

Jak již bylo ukázáno v kapitole 5.2, s FPGA částí se komunikuje přes sdílenou IO datovou sběrnici. Na obrázku 5.3 je ukázáno, na které adresy jsou mapovány které signály.

V AVR se s adresními signály pracuje trochu jinak, než v FPGA. Z AVR vycházejí 4 signály, které obsahují číslo adresy. V HW je natvrdo zadrátovaný adresní dekodér, kretý binární kód (4 vodiče) dekóduje na 16 adresních vodičů, kódovaných signálem 1 z N (v tomto případě 1 z 16)

V AVR se data zapisují na sdílené místo v paměti (adresy 0x14 až 0x17), označené FISUA, FISUB, FISUC a FISUD. Sdílená místa v paměti jsou jen 4 a pomocí nastavení konfiguračního registru FISCR (jeho 2 nejnižších bitů, XFIS1 a XFIS0) se řídí, která čtveřice adres je mapována na zmiňovaná 4 místa v paměti (FISUx).

Zmiňované 4 adresy (0x14 až 0x17) mají ještě jednu možnost využití, a to jako řídící registry přerušení. Z tohoto důvodu se musí řádně nastavit registr (FISCR) (pro podrobnější informace odkazuji na datasheet k FPSLICu [7])

XFIS1	XFIS0	FISUA	FISUB	FISUC	FISUD
0	0	IOSEL0	IOSEL4	IOSEL8	IOSEL12
0	1	IOSEL1	IOSEL5	IOSEL9	IOSEL13
1	0	IOSEL2	IOSEL6	IOSEL10	IOSEL14
1	1	IOSEL3	IOSEL7	IOSEL11	IOSEL15

Tabulka 5.8: Mapování I/O adres v závislosti na konfiguraci registru FISCR

Z tabulky 5.8 je patrné, proč bylo použito primárně 4 adresních vodičů IOSEL0, IOSEL4, IOSEL8 a IOSEL12 místo čtveřice IOSEL0, IOSEL1, IOSEL2 a IOSEL3. Je to právě z toho důvodu, že adresní vodiče s indexy 0,4,8 a 12 jsou mapovány na adresy FISUA…D, bez nutnosti zasahovat do řídícího registru (což zabírá kód a čas při běhu programu).

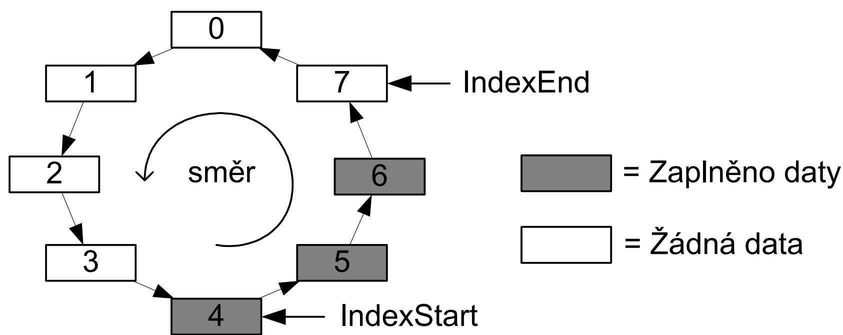
Význam jednotlivých bitů na daných adresách již byl popsán v kapitole 5.2, proto není potřeba jej opakovat

5.10.4 Komunikace s PC

FPSLIC je v roli podřízeného zařízení, veškeré příkazy a iniciace ke komunikaci vydává PC, resp. obslužný program.

Komunikace s PC probíhá přes UART0, standardní komunikace přes sériovou linku RS-232. O přizpůsobení napěťových úrovní se stará obvod MAX212, který současně umožňuje propustnost maximálně 120 kb/s, což je dostatečná rezerva k zvolené přenosové rychlosti 38400 b/s.

V AVR je implementována fronta na příchozí i odchozí data z UART0. Defaultní velikost je 64B a dá se změnit definováním `READ_BUFFER_SIZE` a `WRITE_BUFFER_SIZE` (platí zde omezení, že velikost fronty musí být mocnina 2). Obě fronty jsou implementovány pomocí kruhové fronty. Ukázka kruhové fronty pro 8bytů je na následujícím obrázku 5.27:

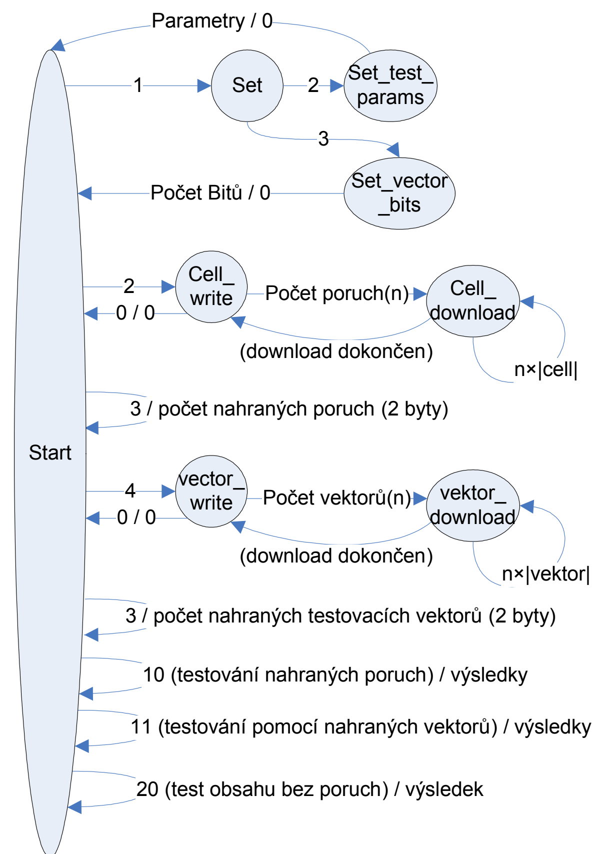


Obrázek 5.27: Kruhová fronta

Index IndexStart ukazuje na první data, která jsou připravena k vyzvednutí z fronty a IndexEnd ukazuje na první volné místo.

V případě fronty k zápisu je aktivováno přerušení při vyprázdnění vysílacího registru (`SIGNAL (SIG_UART0_DATA){}`). Během přerušení se zkontroluje, jestli jsou nějaká data k vysílání a pokud ano, tak se zapíší do vysílacího registru. Pokud nejsou, přerušení se zakáže až do dalšího nahrání dat do fronty k zápisu. V případě fronty přijímaných dat je aktivováno přerušení od příchozích dat (`SIGNAL (SIG_UART0_RECV){}`), které pouze ukládá data do fronty přijatých dat.

Čísla jednotlivých příkazů udává orientační stavový automat na obrázku 5.28, ve kterém jsou znázorněny všechny důležité operace.



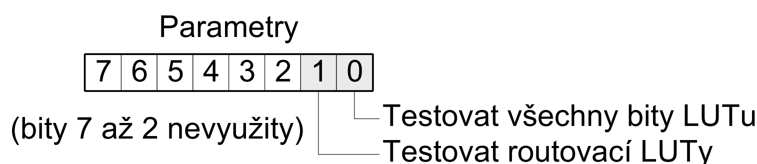
Obrázek 5.28: Schéma komunikace

Kolik je v jednotlivých krocích přenášeno dat je uvedeno v následující tabulce 5.9. Počet je přesně dán až na testovací vektor, jehož počet obsazených bytů se odvíjí od počtu jeho bitů. Řídícími byty se rozumí povely, kterými se přechází mez stavy, jejichž čísla jsou zakreslena v obrázku 5.28

Položka v komunikaci	**Počet bytů**
Řídící byty	1
Parametry	1
Počet bitů	1
Počet poruch	1
Cell (buňka)	11
Počet vektorů	1
Testovací vektor	(proměnný)
Počet nahraných testovacích vektorů	2
Počet nahraných buněk	2
výsledky	8
výsledek	1

Tabulka 5.9: Velikosti přenášených dat

Veškeré parametry testování se nahrávají jedním konfiguračním bytem, jehož struktura je na následujícím obrázku 5.29.



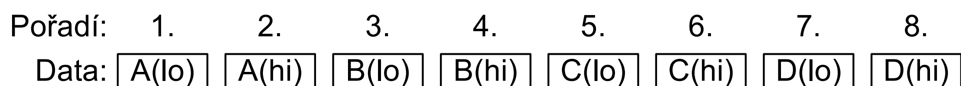
Obrázek 5.29: Struktura bytu parametrů

Bitstream buněk, do kterých se předpokládá injekce poruch, se nenahrává celý, ale pouze část, která stačí na dekódování funkce buňky a zapojení buňky. Nejprve se přenesou 2 byty udávající souřadnici buňky a potom 9 bytů bitstreamu odpovídající hodnotám Z v intervalu 0 až 8, jak je znázorněno na následujícím obrázku 5.30.

Data	Pořadí
X	1.
Y	2.
Z0	3.
Z1	4.
Z2	5.
Z3	6.
Z4	7.
Z5	8.
Z6	9.
Z7	10.
Z8	11.

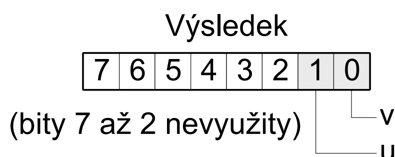
Obrázek 5.30: Pořadí bytů v bitstreamu buňky

Po testování se z AVR odesílá 8 bytů udávající, kolik poruch spadlo do které kategorie. Kategorie jsou 4 (A, B, C, D) a každá kategorie je 16bitová. Data se přenáší v pořadí uvedeném v obrázku 5.31.



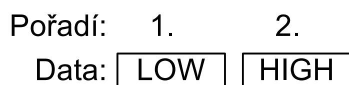
Obrázek 5.31: Pořadí bytů ve výsledcích

V případě, že se jedná o test aktuálního nahraného obsahu bez injekce poruch, výsledek má jenom 1 byte ve tvaru podle následujícího obrázku 5.32. Mimo jiné je tento byte přímo roven třídě (hodnota 0 = třída A, hodnota 1 = třída B, hodnota 2 = třída C, hodnota 3 = třída D), do které nahraný benchmark spadá. Nedošlo-li k chybě, či pokud nebyla injektována porucha, pak by výsledkem testování měla být vždy třída A.



Obrázek 5.32: Pořadí bitů ve výsledku

Tato funkce (testování bez poruch) je velice užitečná, pokud chceme ověřit, že po testování se obvod dostal do původního stavu. Pokud by se například stalo, že nahrané poruchy budou patřit jinému bitstreamu a místa buněk určených k testování se budou shodovat, tak je pravděpodobné, že se funkce benchmarku poruší a výsledek nespadne do kategorie A.

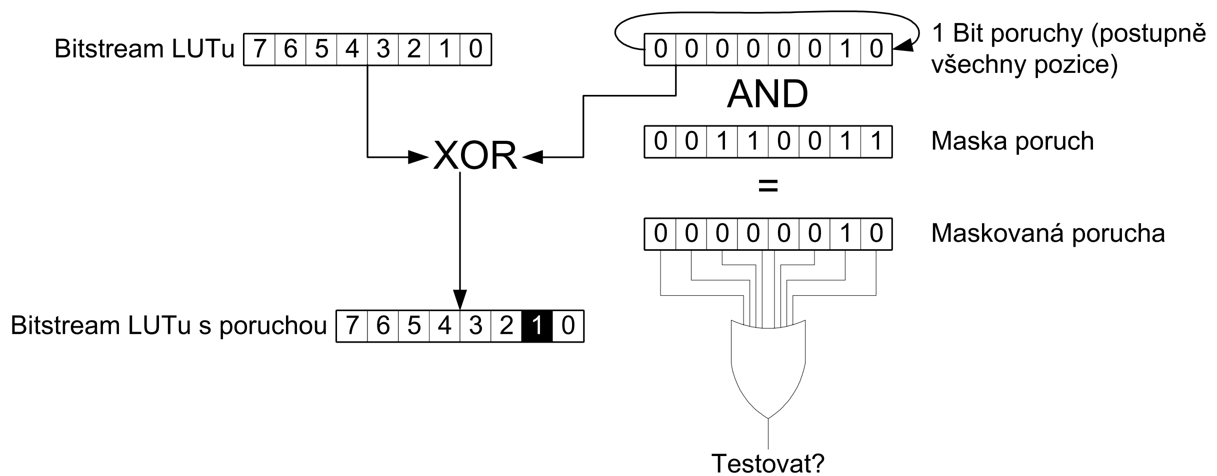


Obrázek 5.33: Obecné pořadí bytů u 16bitových slov

Kdekoli se přenáší více jak 8bitová slova, je pořadí jednotlivých částí (jako je spodní a horní byte u 16bitového slova na obrázku 5.33), je pořadí stanoveno tak, že se nejprve přenáší nejmenší řád a nakonec největší.

5.10.5 Injekce poruch a testování

Nejnosnějším přínosem celého návrhu a implementace v FPSLICu je právě možnost injekce poruch do FPGA. Nejmenší část bitstreamu, kterou je možné změnit, je 1 byte (8 bitů). Cílem měření je ale sledovat reakce na provedení změny 1 bitu v bitstreamu. Když chceme vyzkoušet všechny bity, musíme postupně 8krát nahrát byte bitstreamu, pokaždé s pozměněným bitem bitstreamu na jiné pozici.



Obrázek 5.34: Generování poruch v LUTu

Poruchou se v tomto případě rozumí negace pouze 1 bitu z bitstreamu, nikoli celého bytu. V tom případě by se jednalo o vícenásobnou poruchu.

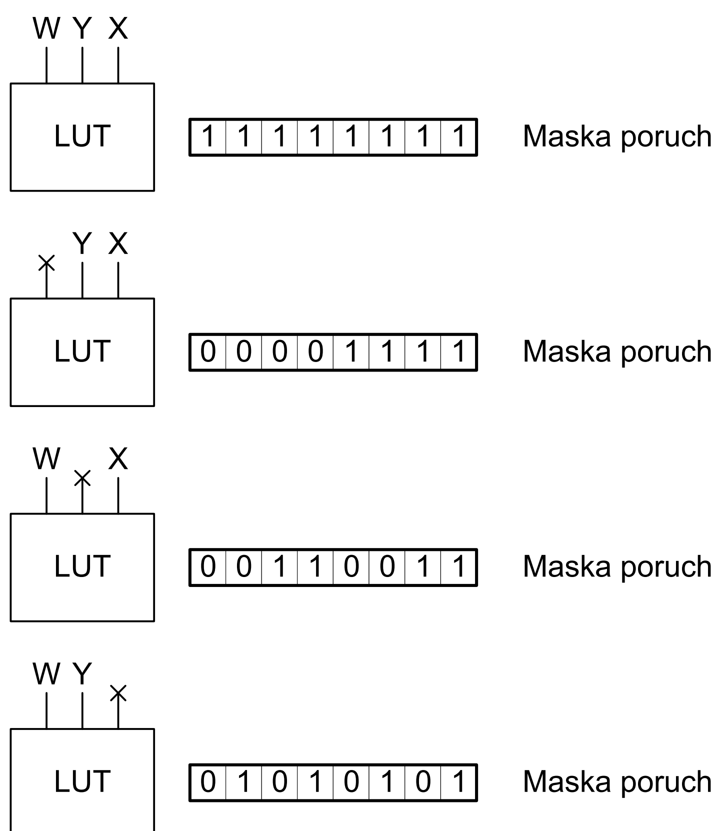
Při testování LUTu se využívá toho, že bitstream pro každý z obou LUTů zabírá právě jeden celý byte bitstreamu. Proto se vytvoří byte 1bitové poruchy (00000001b), který se 8krát rotuje a pokaždé použije ke generování bitstreamu s poruchou, jak je ukázáno na obrázku 5.34.

Nemá smysl zkoušet injekovat poruchy do LUTů, které nebyly použity pro design. Výsledek bude vždy stejný, tj. porucha se neprojeví. Pak existují zvláštní případy LUTů, o nichž víme, že nejsou plně využívány. Mezi takové případy patří:

- Použití 3 či méně vstupních signálů
- Buňka ve funkci propojování signálů (z důvodu omezených možností propojovací sítě horizontálních a vertikálních datových vodičů)

V takových případech jsme schopni zjistit, které bity bitstreamu jsou a které nejsou využívány. Ty, které nejsou využívány, nemá teoreticky vůbec smysl testovat, jelikož výsledek bude vždy stejný: porucha v nevyužívaných částech bitstreamu se neprojeví (výsledný test spadne do skupiny A).

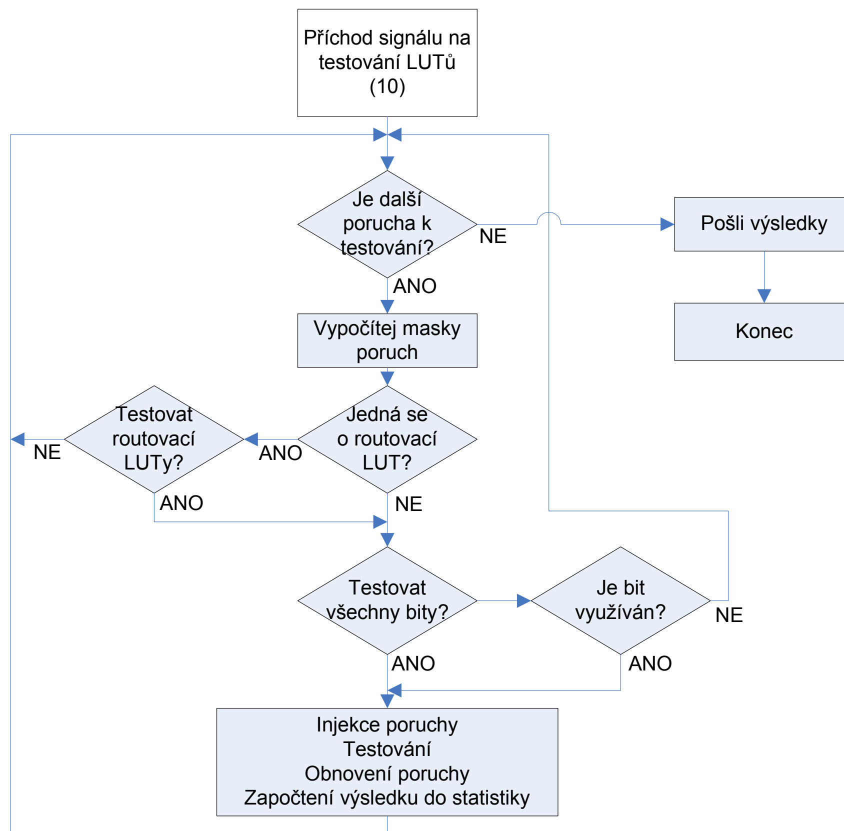
Právě pro tyto případy je použita maska poruch udávající, které bity jsou využívané (tam je hodnota 1) a které nejsou využity (hodnota 0). Příklad několika masek poruch je na následujícím obrázku 5.35. V prvním řádku je maska pro plně využitý LUT, v dalších jsou masky pro částečně nevyužívané LUTy



Obrázek 5.35: Příklady masek poruch

Pokud je využíván jen jeden z LUTů, pak jsme tuto situaci schopni detekovat podle konfigurace multiplexoru řízeného signálem ZM a obsahu daného LUTu. Viz obrázek 3.4. V těchto detekovaných případech se nastaví nulová maska poruch daného LUTu (00000000b).

Pokud je v nastavení testování uvedeno, že se nevyužívané bity testovat nebudou, tak tam, kde vektor poruchy v součinu s maskou poruch dá nulový vektor, se přeskočí injekce poruchy a její testování. Obdobně pokud je nastaveno, že se nebudou testovat buňky primárně využívané k routování, tak v buňkách, které jsou odhaleny jako routovací buňky, se přeskočí testování všech poruch v celé buňce. Graficky je to znázorněno na obrázku 5.36.



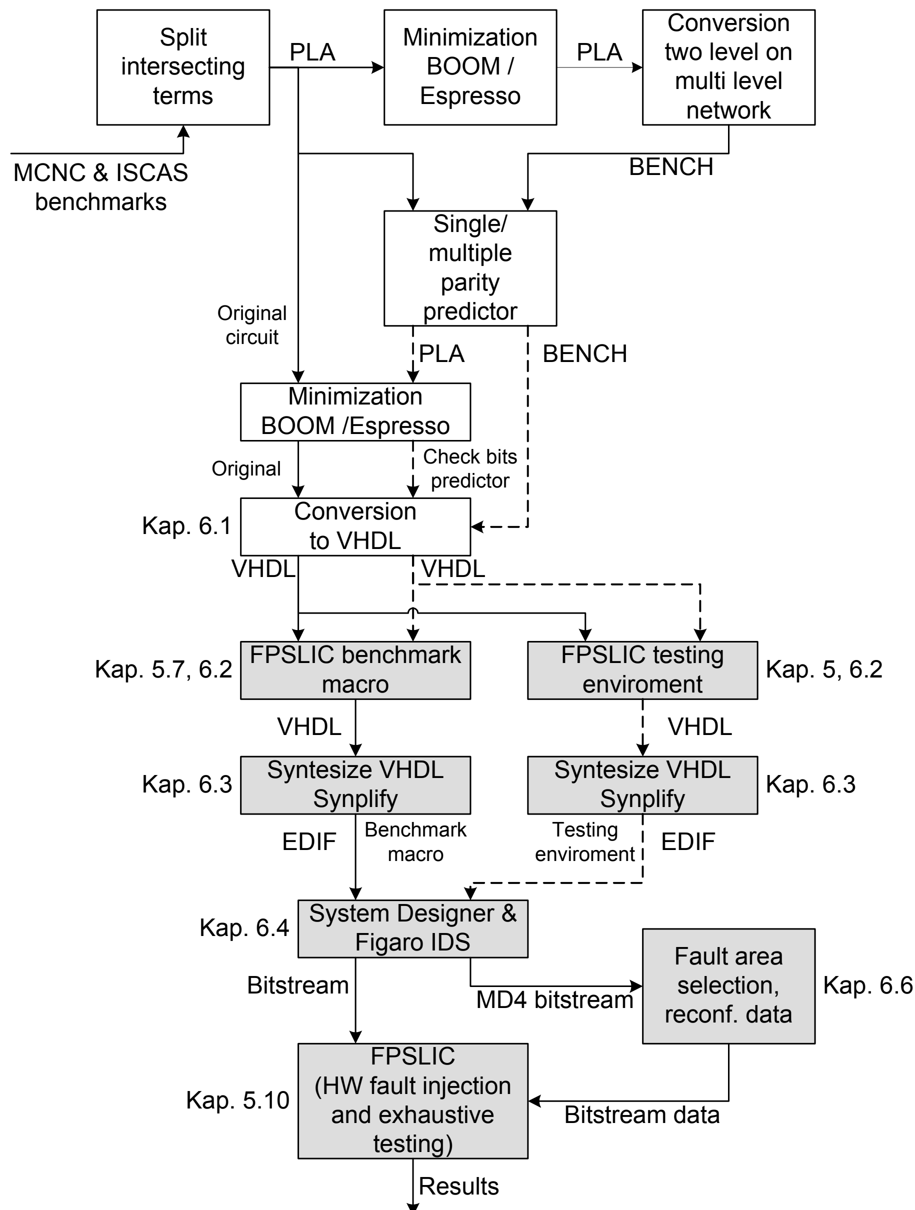
Obrázek 5.36: Vývojový diagram testování

Analýza buňky a vytvoření masky poruch jsou opřeny o detailní znalost struktury buňky a znalosti významu jednotlivých bitů. Informace o významu těchto bitů nejsou firmou ATMEL veřejně dostupné a je možné je oficiálně získat (jedná se o „FPGA MODE 4 configuration") pouze pod smlouvou NDA (Non-disclosure Agreement). Proto se zde na tomto místě nebudu podrobněji rozepisovat, jakým způsobem se pozná zapojení a funkce LUTu.

6 SW nástroje

V této kapitole jsou popsány všechny softwarové nástroje, které byly použity pro vytvoření bitstreamu a následně pro testování poruch. Použitých nástrojů je více, proto budou popsány postupně, podle toho, jak jsou použity.

V této práci se budu věnovat popisu SW nástrojů pro testování v FPSLICu. Pro nástroje přípravy benchmarků do VHDL formy odkazuji na práce [4,10,13,14], jelikož využívám výsledky z těchto prací v podobě již vytvořeného VHDL kódu benchmarků.



Obrázek 6.1: Vývojový diagram implementace v FPSLICu

Pro potřeby této diplomové práce byly vytvořeny 2 programové nástroje pro generování EDIF kódu celého návrhu do FPSLICu, GenVHDL a CorrectEdif. Dále byl vytvořen program pro komunikaci s PC a grafickým výběrem oblastí poruch. Všechny byly napsány v Javě [17] za pomoci vývojového prostředí Eclipse [18]. Jejich význam a funkce je popsána v následujících kapitolách.

6.1 Generování VHDL benchmarku

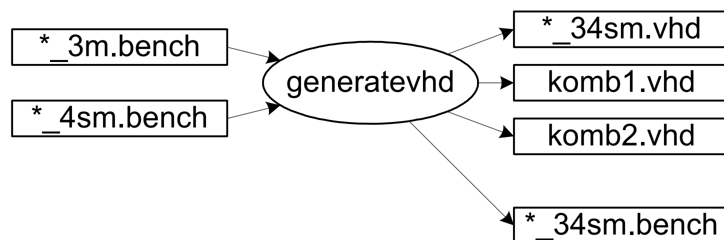
Vstupním formátem benchmarku, se kterým začínají pracovat vytvořené nástroje je formát VHDL. V tomto formátu byly také benchmarky zadávány. Nejedná se ale o původní formát, vstupní formát je formát *.pla. Ten je za pomoci již vytvořených nástrojů [6,25,26] zminimalizován a převeden do formátů *.bench a posléze VHDL.

Názvy benchmarků mají přídavky k jménu oddělené „_“, aby bylo možné jednoznačně určit, o jaký soubor se jedná. Pro generování těchto benchmarků jsem použil sw nástroje používané např. v [6,13]:

přípona	**význam**
*_4s	prediktor parity
*_4sm	minimalizovaný prediktor parity
*_3	benchmark
*_3m	minimalizovaný benchmark
*_34sm	minimalizovaný benchmark i s prediktorem

Tabulka 6.1: Přípony benchmarku

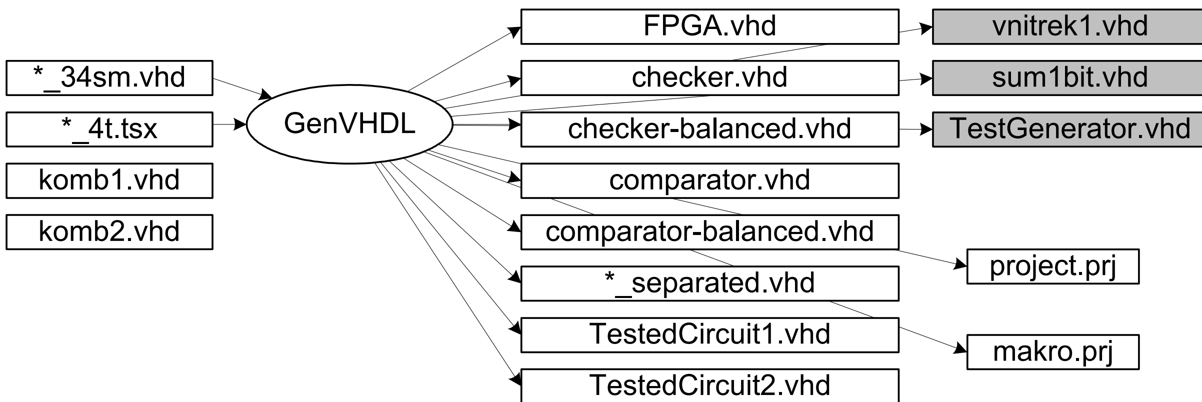
Minimalizované formy benchmarku ve formátu *.pla se získají pomocí programu espresso [26] a BOOM[25] a do formátu *.bench se převedou programem pla2bench.



Obrázek 6.2: Soubory zpracovávané programem generatevhd

Ze souborů *.bench program generatevhd vytvoří VHDL entity komb1.vhd (benchmark) a komb2.vhd (prediktor parity), dále souboru pojmenovaný *_34SM.vhd, který spojuje moduly komb1 a komb2 jako komponenty do samostatné entity komb.

6.2 Generování VHDL kódu pro implementaci v FPSLICu



Obrázek 6.3: Soubory zpracovávané programem GenVHDL

Na obrázku 6.3 je znázorněno, jaké vstupní soubory program GenVHDL potřebuje a jaké generuje. Jako hlavní vstup používá benchmark s jménem *_34sm.vhd. Z úvodních 2 řádek se načtou velikosti vstupních a výstupních vektorů. Tyto 2 řádky začínají následovně:

```
--          lines from primary input  gates .......
--          lines from primary output gates .......
```

Následně se dekóduje soubor *_4t.tsx, ve kterém je zapsáno, kolik a případně které bity jsou paritní vodiče a ze kterých vodičů je parita počítána.

Ze získaných informací o počtu vodičů a názvu benchmarku se generují všechny soubory. Jsou vyjmenovány na obrázku 6.3. Ke generování se využívá prototypů těchto souborů uložených v adresáři source. Prototypy jsou v podstatě kompletní vytvořené soubory, kde jsou důležité části, které se musí nahradit, označeny klíčovým slovem uvozených v procentech, např. `%InputVectorSize%`. Pro každý soubor je pak v programu GenVHDL vytvořena jedna funkce, která všechny tyto parametry najde a nahradí požadovanými daty. V některých případech se jedná pouze o doplnění počtu bitů, v jiných může jít o složitější konstrukci, jako je tomu například u checkeru. V souborech projektů pro Synplify se doplní kompletní jména souborů.

U souborů, které jsou na obrázku 6.3 v šedém rámečku, nedochází k žádnému generování, soubory jsou pouze zkopírovány. Všechny potřebné parametry jsou nastaveny pomocí prostředků jazyka VHDL, tj. pomocí generických parametrů.

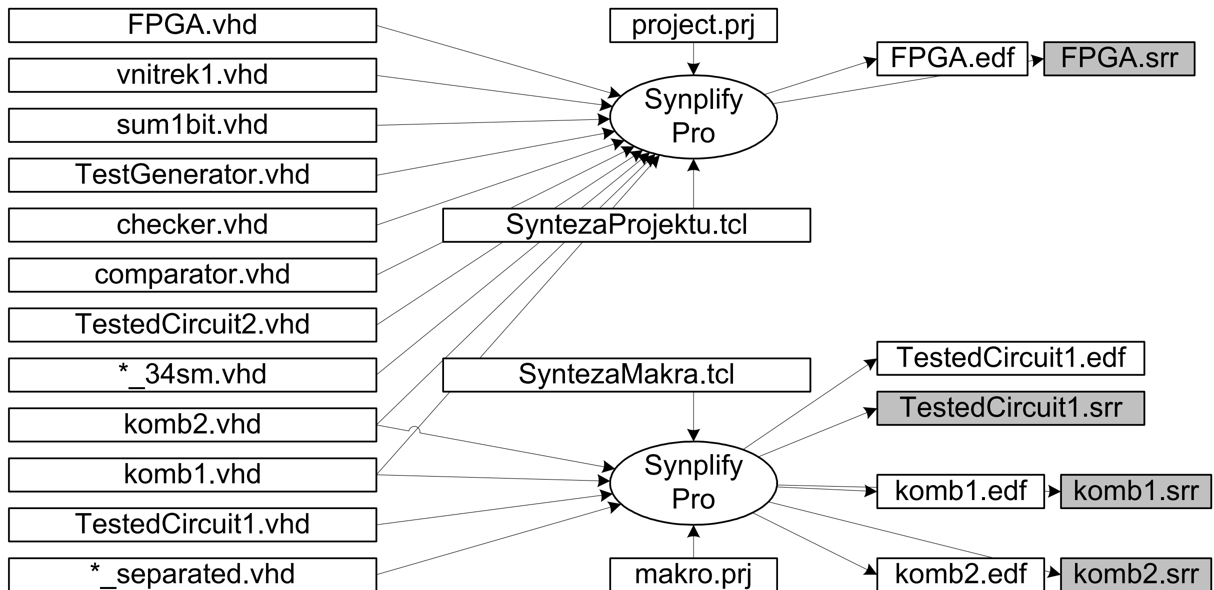
Program GenVHDL je napsán v programovacím jazyku JAVA. Po kompilaci není vytvořen spustitelný soubor, ale java třída, která se spouští následujícím příkazem, kde %1 je nahrazeno názvem benchmarku (např. `ALU11`):

```
java -classpath .\bin\ GenVHDL %1
```

6.3 Syntéza do EDIF

Pro syntézu do EDIF byl použit nástroj Synplify Pro, jelikož právě pro tento syntézní nástroj jsou zaměřeny atributy syn_keep, o kterých zde již bylo několikrát psáno. Je třeba si uvědomit, že jiný syntézní nástroj by tyto parametry nejspíše ignoroval.

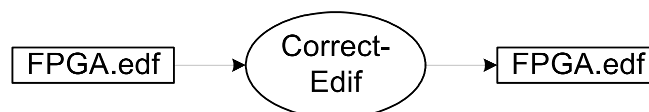
Program je obsluhován z příkazového řádku bez nutnosti interakce uživatele, což velmi usnadňuje dávkové zpracování. Syntézní nástroj je řízen 2 TCL skripty. První, SyntezaMakra.tcl, je určen pro samostatnou syntézu testovaného obvodu TestedCircuit1. Druhý TCL skript, SyntezaProjektu.tcl, je určen pro syntézu celého návrhu do FPGA bez přítomnosti makra. Jaké soubory jsou nahrány do projektu a přítomny syntéze je určeno v projektových souborech Synplify (*.prj), jak je znázorněno na obrázku 6.4.



Obrázek 6.4: Syntéza do EDIFu

Při syntéze celého návrhu v FLGA je důležité, aby benchmark s prediktorem byly překládány jako samostatný modul, především z důvodu snazší lokalizace a rozmístění v hradlovém poli. Je to i z toho důvodu, aby nedocházelo k promíchávání s jinými částmi návrhu, což v extrému může mít za následek vynechání celé kopie benchmarku, nebo v ještě horším případě zjednodušení obvodu až tak daleko, že výsledky checkerů a komparátorů budou nahrazeny trvalým generátorem log. 1, jelikož syntézní nástroj neví, že může docházet k poruchám.

Nejjednodušší způsob, jak zajistit oddělenou syntézu benchmarku a prediktoru, do nichž budou vkládány poruchy, je nepřipojit soubor s VHDL kódem k projektu. Syntézní nástroj pak nepozná, že jde o stejnou kopii benchmarku, jaká už v návrhu je a benchmark s prediktorem (TestedCircuit1) pak syntetizuje jako černou skříňku (black box). Nevýhoda tohoto postupu je špatné označení této „černé skříňky“ v netlistu EDIF. Místo názvu makra `testedCircuit1` napíše `testedCircuit1_work_fpga1_komunikaceio_0`, s čímž si poté neporadí FIGARO IDS. Tento problém odstraňuje program CorrectEdif, jehož schéma je na obrázku 6.5.



Obrázek 6.5: Oprava FPGA.edf

Tento program provádí pouze zmiňovanou záměnu názvů, což je poměrně snadná úprava, která by se dala zvládnout v unixu pomocí programů typu sed či awk. Jelikož je ale používána

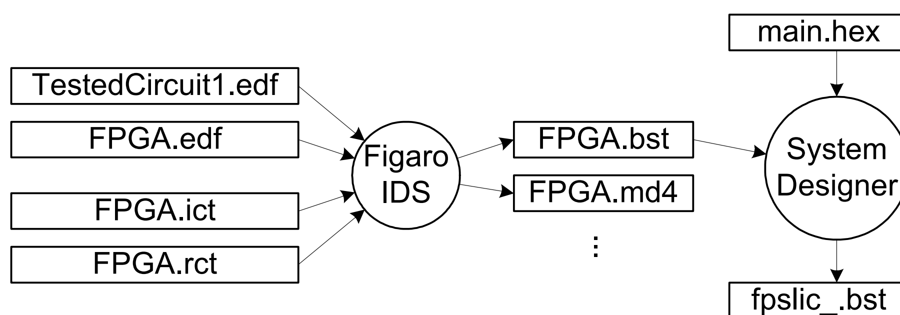
platforma Windows, bylo nutné tento program naprogramovat, aby se tato úprava nemusela dělat ručně a mohla se využít dávka pro generování EDIFu.

6.4 Vytvoření bitstreamu

Až do vytvoření souborů EDIF bylo možné celý postup zautomatizovat pomocí příkazové řádky. To bohužel není možné u kroku vytvoření bitstreamu FPGA části FPSLICu, jelikož nástroj FIGARO IDS má velmi omezený repertoár TCL příkazů a neumožňuje pomocí nich provést rozmístění makra TesteCircuit1 podle potřeb (samostatně a pokud možno vzdáleně od okolní logiky).

Již zcela nemožné je zautomatizování kompletace bitstreamu z FPGA a AVR části FPSLICu, jelikož program SystemDesigner vůbec nepodporuje obsluhu z příkazové řádky.

Z pohledu vytvářených souborů si můžeme udělat představu o fungování programů z následujícího obrázku 6.6. Soubory FPGA.ict a FPGA.rct slouží k automatickému přiřazení portů a vybrání správného obvodu, jejich vznik a použití je popsán v kapitole 6.4.2.



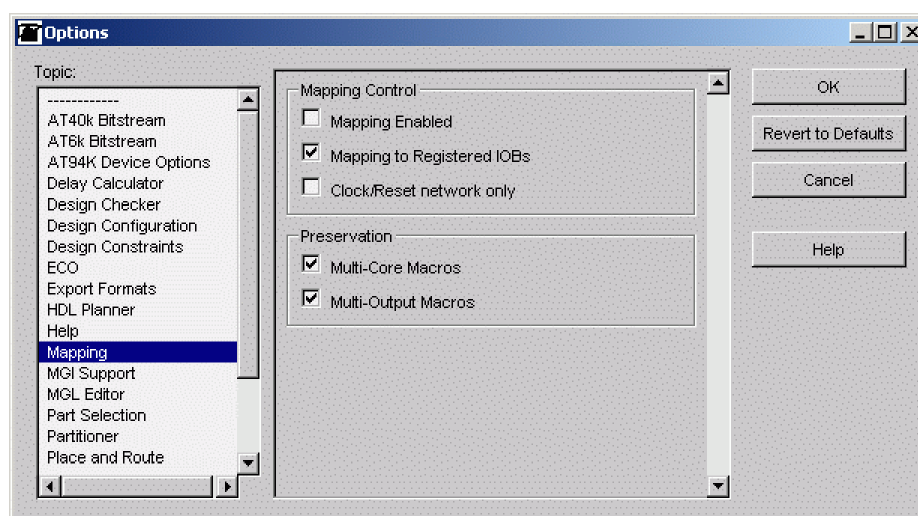
Obrázek 6.6: Postup vytvoření bitstreamu

6.4.1 Figaro IDS

Proces vytvoření bitstreamu pomocí nástroje FIGARO IDS se skládá ze 2 částí: vytvoření makra z TestedCircuit1 a jeho následného ručního rozmístění a vložení do návrhu celého FPGA.edf.

FIGARO IDS je nástroj s poněkud zvláštním chováním, kde během procesu vytváření bitstreamu číhá mnoho pastí (a někdy je tak komplikované vrátit krok zpět (pokud to vůbec lze), že je lepší začít úplně od počátku). Z tohoto důvodu si dovolím postup vytváření bitstreamu a práci s programem FIGARO IDS popsat velmi podrobně.

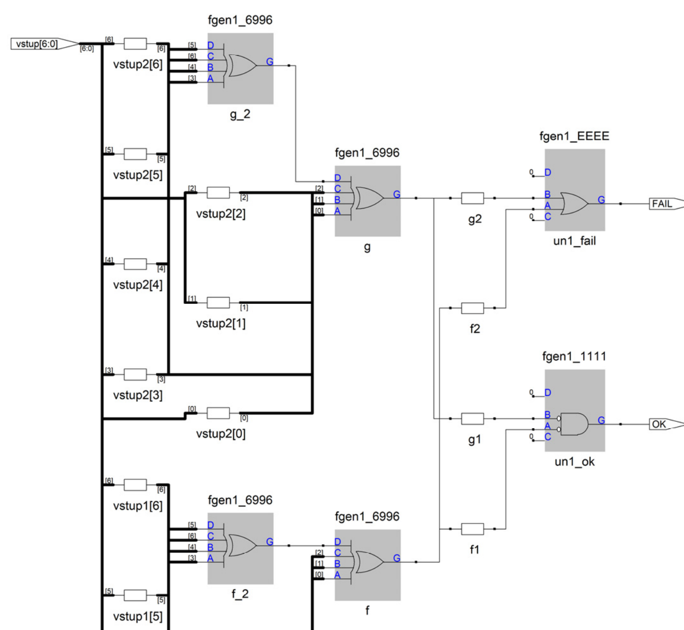
6.4.1.1 Význam parametru mapování



Obrázek 6.7: Nastavení mapování

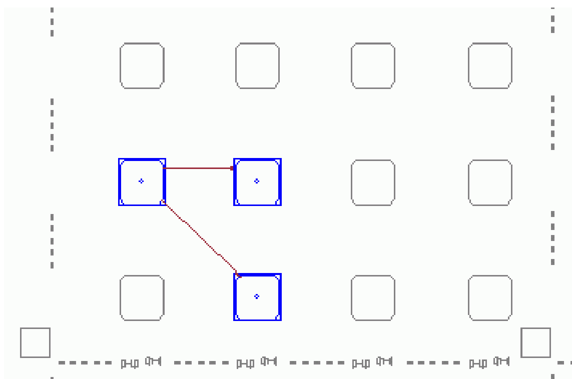
Kromě první nástrahy a nebezpečí nechtěné optimalizace návrhu v podobě syntézy do EDIFu (o níž byla již řeč v kapitole 5.8.2, kde je podrobně popsán i způsob, jak se s tímto problémem vypořádat) na nás číhá druhé nebezpečí: optimalizace programem FIGARO IDS. Pokud je povoleno mapování (v menu Options – Options, téma mapping, položka mapping enabled), tak může dojít k nechtěnému vypuštění zdvojené logiky a podobně. To zvláště dopadá na obvody odolné proti poruchám, kde je zdvojení běžné a je nutné redundantní logiku zachovat, aby byla zachována vlastnost odolnosti proti poruchám, či úplná samočinná kontrolovanost.

Uvedu zde příklad úplně samočinně kontrolovatelného (TSC) hlídače parity, jehož schéma je na obrázku 6.8. Je na něm dobře patrné zdvojení paritních XOR-stromů a nezávisle generované signály OK a FAIL.

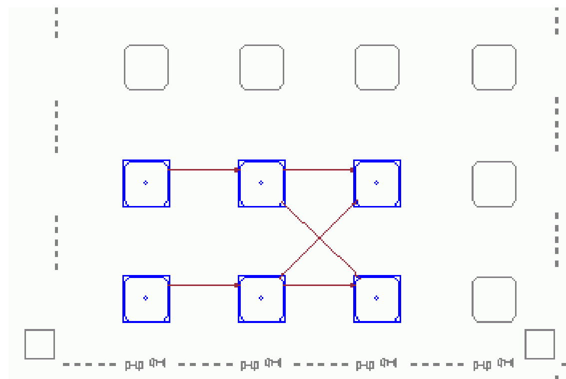


Obrázek 6.8: TSC parity checker (výřez)

Pro tento paritní hlídač následují dva obrázky. První (obrázek 6.9) je po výsledku place&route, kdy je mapování povoleno. Na druhém obrázku 6.10 je mapování zakázáno.



Obrázek 6.9: Checker – mapování povoleno

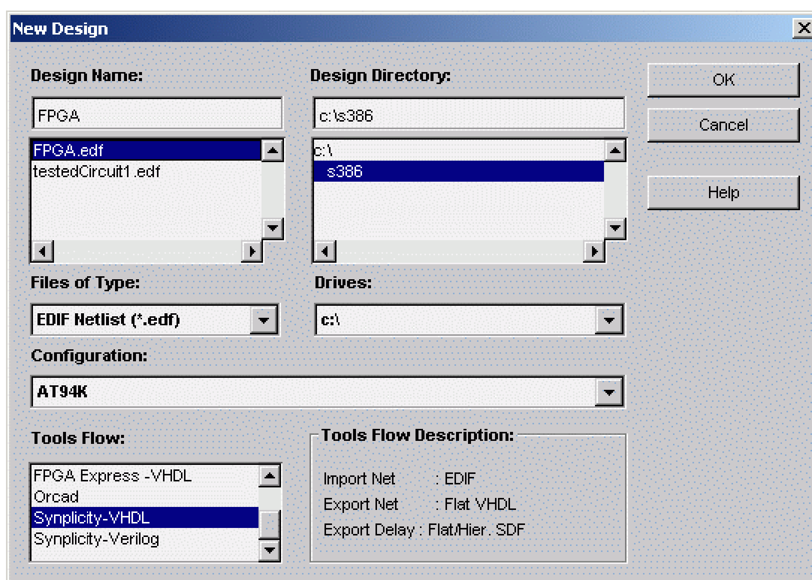


Obrázek 6.10: Checker – mapování zakázáno

Výsledek je zcela zřejmý: mapování nezachovává redundantní logiku, což v tomto příkladu vede ke zcela fatálnímu porušení vlastnosti TSC a dokonce i odolnosti proti poruchám. Tomu se dá právě zabránit tím, že se v nastavení options zakáže mapování.

6.4.1.2 Postup vytvoření makra

Po spuštění FIGARO IDS je nejprve nutné nastavit adresář, kde je umístěný návrh a soubory, se kterými se bude pracovat (položka menu File → Design Setup). Nejprve je nutné tlačítkem remove smazat ze seznamu všechny předchozí soubory, včetně použitého adresáře. Poté je možné tlačítkem New Design přidat soubory z nového návrhu.



Obrázek 6.11: FIGARO IDS - New Design

Při vybírání souborů je nesmírně důležité zvolit Tools Flow na Synplicity-VHDL, jak je ukázáno na obrázku 6.11. V opačném případě, pokud se na toto nastavení zapomene, tak FIGARO IDS se chová zcela normálně a problém nastane až při otevírání FPGA.edf, kdy FIGARO IDS neporozumí syntetizované sčítačce (`SYNLPM_ADDI13`) a musíme začít znova.

Obrázek 6.12: Library out of date

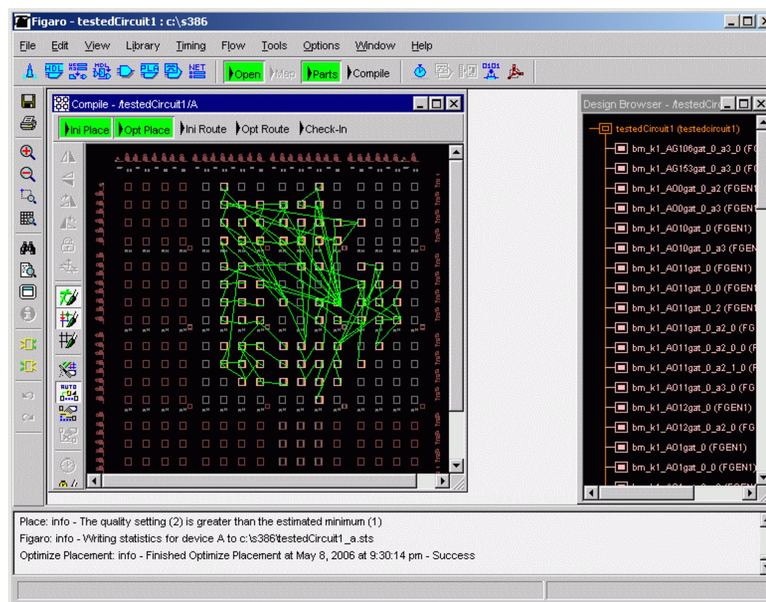
Po odsouhlasení všech vložených souborů EDIF se program se zeptá, jestli se má aktualizovat knihovna user94k (obrázek 6.12). Správná odpověď je ano.

Nyní je důležité upravit možnosti programu, neboť program je při otevření nového designu změní. Nejdůležitějším parametrem je mapování, jak bylo popsáno v předchozí kapitole. Zakázané mapování je možné zkontrolovat pohledem na tlačítka programu FIGARO IDS. Tlačítko MAP by mělo být neaktivní (zakázané).

Po provedení těchto kroků je již snadné makro testovaného obvodu zkompilovat. Stačí otevřít návrh TestedCircuit1 (typ EDIF) a FIGARO IDS již zvolí nejmenší obvod, do kterého se makro vejde a automaticky vybere i oblast povolených buněk, na které je možné návrh rozmístit. Ostatní buňky uzamkne a znemožní tak automatickému umístění na tyto buňky.

Buňky a tvary zakázaných a povolených oblastí lze upravovat i ručně vybráním myší. Uzamčení se provádí příkazem close location (a naopak uvolnění příkazem open location).

Makro se uloží do knihovny příkazem Check-in.



Obrázek 6.13: Rozmístění makra

6.4.2 Ruční rozmístění a generování bitstreamu

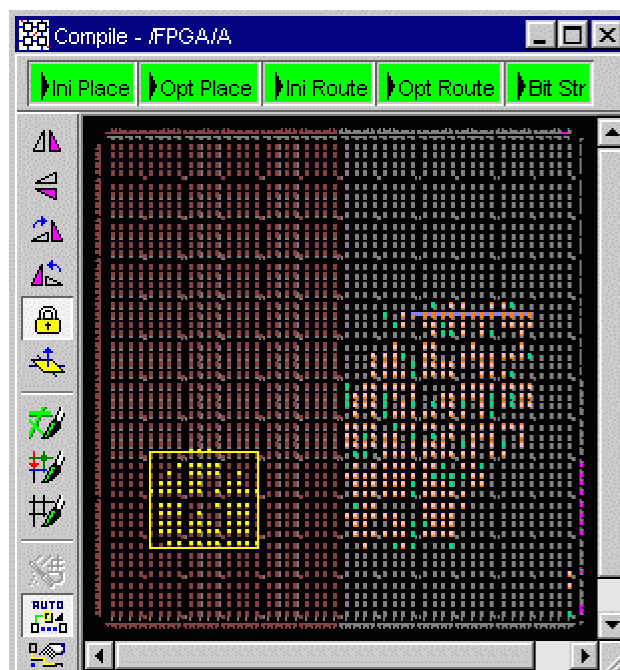
Po předchozím kroku již máme v knihovně makro testovaného obvodu TestedCircuit1. Nyní můžeme pokračovat v rozmístění celého návrhu. V adresáři kromě návrhu FPGA.edf je nutné mít i další 2 soubory, ve kterých je přiřazení pinů a portů. V celém návrhu je zapojen pouze jediný pin, a to externí reset připojený na tlačítko manuálních hodin na vývojové desce. Informace o přiřazení tohoto pinu se spolu s použitou verzí obvodu FPSLIC nachází v souboru FPGA.rct. Tento soubor je generován programem FIGARO IDS po úspěšném nakonfigurování těchto parametrů a je v adresáři automaticky uložen pro případné opakované rozmístění.

Druhý soubor, FPGA.ict, slouží k přiřazení příslušných portů návrhu ke speciálním signálům vedoucím mezi AVR a FPGA. Pokud by se tento soubor v adresáři nenacházel, není již žádná jiná možnost, jak tyto signály správně připojit. Ke generování tohoto souboru je možné použít program System Designer v záložce advanced flow pomocí tlačítka AVR-FPGA interface. Jelikož se ale toto propojení nemění, tak stačí toto propojení naklikat pouze jednou a pak už stačí zmiňovaný soubor FPGA.ict zkopírovat do návrhu dalšího benchmarku.

Při otevírání se musí dodržovat postup uvedený dále, aby bylo rozmístění a propojení úspěšně dokončeno. Nejprve je nutné spustit počáteční rozmístění (Ini Place). Během tohoto kroku proběhne důležité přiřazení jednotlivých portů. To je podstatné udělat dříve, než se ručně umístí makro TestedCircuit1 a než se vyznačí oblasti povoleného rozmístění.

Po počátečním rozmístění se zamkne jedna půlka FPSLICu, kam přijde benchmark s prediktorem v podobě makra TestedCircuit1 a kam se budou injekovat poruchy. Zamknutí se provádí vybráním buněk myší a v menu příkazem edit → close location. Zamknutá oblast má tu vlastnost, že automaticky rozmísťované buňky jsou umístěny mimo tuto oblast. (pokud v dané oblasti nejsou zamknuté – viz ikonka zámku, nebo příkaz edit → lock. Do této zamknuté části se přetáhne blok makra TestedCircuit1, pokud možno ne přímo na hranici zamknuté oblasti, aby se tak do této oblasti nemapovaly buňky ve funkci routování pro řídící logiku.

Příklad, jak by mělo výsledné rozmístění vypadat je na obrázku 6.14.

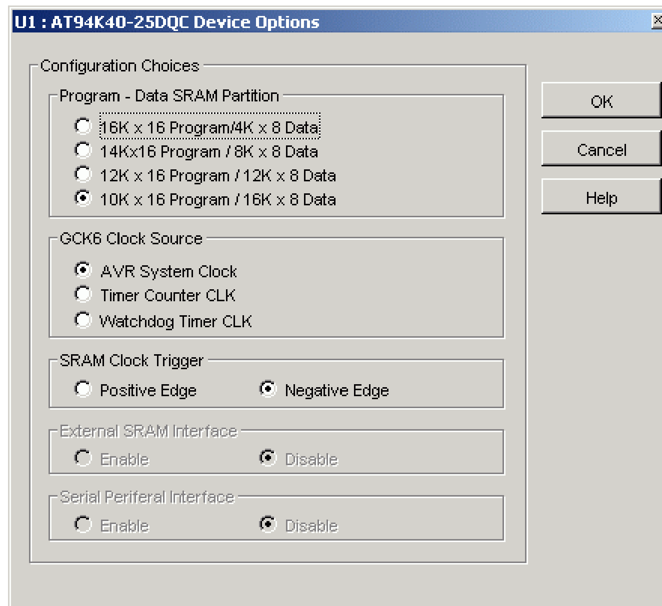


Obrázek 6.14: Rozmístěný design

6.4.3 Kompletace bitstreamu

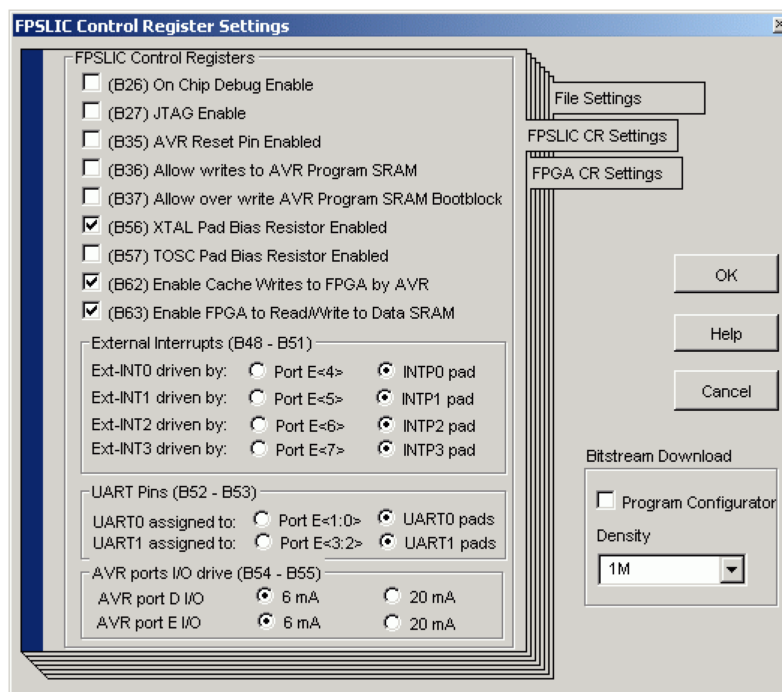
Bitstream, kterým se programuje konfigurační paměť FPSLICu, se kompletuje v programu System Designer. V tomto programu se rovněž nastavují i různé konfigurační bity, které jsou velmi důležité pro rekonfiguraci a funkčnost celého návrhu.

První nastavovací okno se nachází záložce Advanced Flow pod tlačítkem AT94K Device Option. V tomto okně je především důležité nastavení rozvržení programové paměti a datové paměti. Nejvíce paměti pro data je možné nastavit 16KB, jak je ukázáno na obrázku 6.15.



Obrázek 6.15: Nastavení paměti FPSLICu

Další místo, kde se nastavují parametry se nachází v dialogovém okně pod tlačítkem Device Programming. Nesmírně důležité je nezapomenout v záložce FPSLIC CR Settings zatrhnout položku (B62) Enable Cache Writes ti FPGA by AVR, jak je vidět na obrázku 6.16.



Obrázek 6.16: Povolení rekonfigurace

6.5 Programování přípravku

Když už máme hotový bitstream fpslic_.bst, je snadné ho nahrát do přípravku pomocí programu Atmel CPS (Atmel AT17 Configurator programming system). Bitstream se posílá

přes paralelní port pomocí programovacího kabelu ATDH2225 do konfigurační paměti AT17LV010.

Během programování je nutné přípravek přepnout přepínačem do programovacího režimu. Po ukončení nahrání bitstreamu je zapotřebí přípravek opět přepnout do režimu normálního běhu a stisknout reset, čímž se nový bitstream nahraje do FPSLICu.

6.6 PC – komunikační program

Pro PC jsem vytvořil obslužnou aplikaci s grafickým rozhraním v programovacím jazyce JAVA. Tento program umí načítat bitstream ve formátu MD4 a pomocí myši vybírat oblasti, kam budou injekovány poruchy. Následně pak umožňuje bitstream vybraných lokalit poslat do FPSLICu, poslat příkaz k jeho otestování a stejně tak vyčítat výsledky testování.

6.6.1 Podmínky pro běh programu

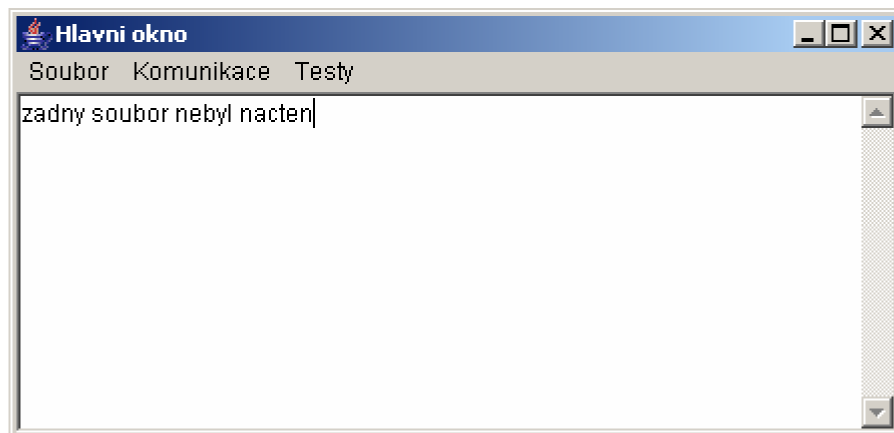
Jak již bylo řečeno, program je napsán v Javě, proto je nutné mít nainstalovaný Java Runtime Enviroment ve verzi 1.5, případně vyšší.

Program se dále opírá o knihovnu RxTx umožňující komunikaci po sériovém a paralelním portu, kompletní řízení těchto portů a nastavení. K tomu je zapotřebí zkopírovat soubor RXTXcomm.jar do podadresáře Javy jre\lib\ext a soubor rxtxSerial.dll do adresáře \jre\bin. Tím jsou splněny všechny požadavky pro běh programu.

6.6.2 Hlavní okno

Na obrázku 6.17 je snímek obrazovky základního okna aplikace. Hlavní okno je rozděleno na menu a textovou oblast, do které jsou psány všechny informační hlášky a výsledky testů.

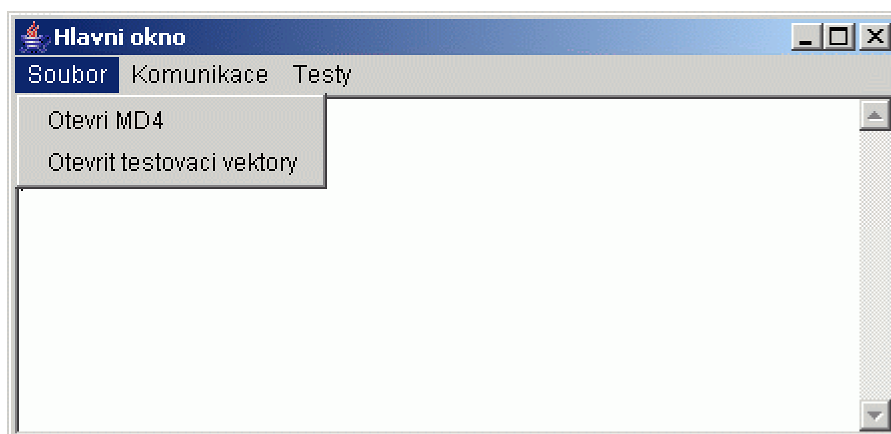
Další oblast, kam jsou vypisovány informace je konzole. Jedná se o podpůrné informace typu verze nahraných knihoven, probíhané akce, souřadnice myši při pohybu nad strukturou hradlového pole, včetně hodnot obou LUTů. Dále je do tohoto okna vypisována komunikační aktivita na sériové lince – znak „|“ pro odeslaná byte a „.“ pro přijatý byte.



Obrázek 6.17: Hlavní okno

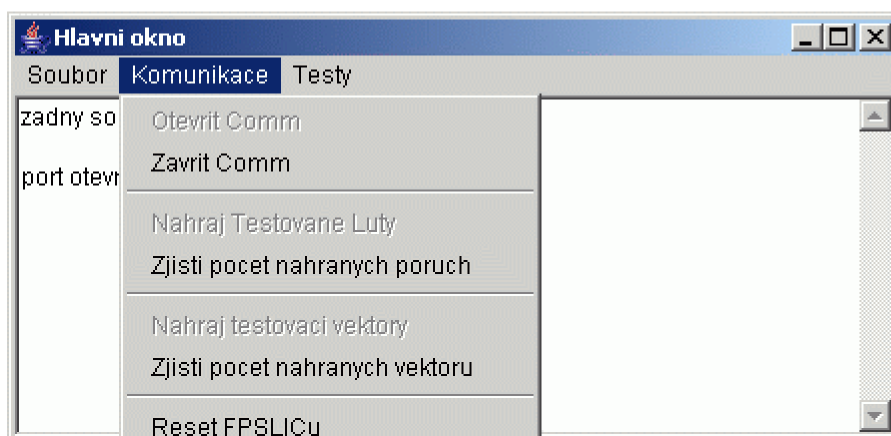
V menu Soubor se nacházejí příkazy na otevření bitstreamu ve formátu MD4 a otevření souboru testovacích vektorů. Při otevírání bitstreamu se načítá soubor FPGA.md4. Je potřeba vybrat soubor z adresáře správného benchmarku, který je aktuálně nahrán v FPSLICu. V opačném případě by mohlo dojít k nahrání špatného bitstreamu. V tomto případě nemůže dojít ke zničení obvodu, pouze k chybné funkci vlivem přepisování bitstreamem jiného benchmarku. Pokud by byla práce rozšířena o testování routování, mohlo by dojít k mnoha přím o vítězství logické úrovně na sběrnici, které by v konečném důsledku mohly dojít

k přehřátí a zničení obvodu. Oficiální zdroje neuvádějí, jestli se obvod vlivem špatného bitstreamu může zničit.



Obrázek 6.18: Menu soubor

V menu Komunikace se nachází více příkazů. Příkaz **Otevřít Comm** slouží k natažení knihovny RxTx a nakonfigurování režimu sériového přenosu (8 bitů bez parity, 1 start bit, 1 stop bit, rychlost 38400 b/s). Tímto příkazem program zamkne sériový port jen pro sebe, tudíž by se jinému programu nemělo podařit tento port otevřít. Pokud je tento port momentálně využíván jiným programem, tak se otevření nezdaří a musí se počkat na uvolnění portu.



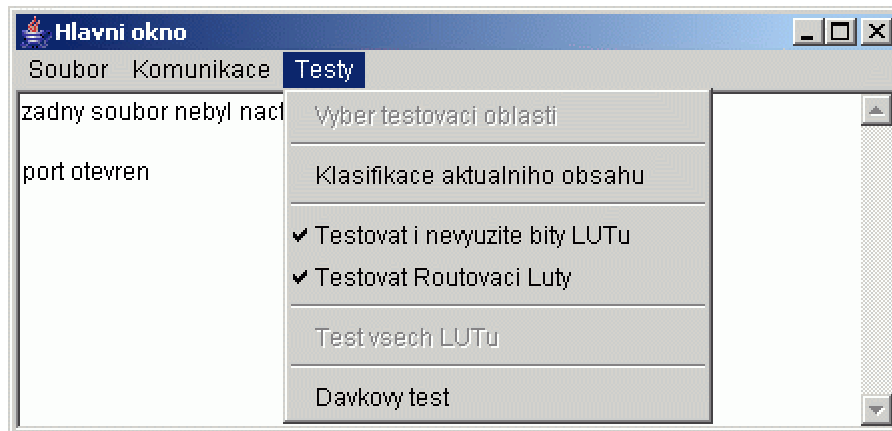
Obrázek 6.19: Menu komunikace

Příkazem **Zavřít Comm** se sériový port uvolní pro ostatní programy. Příkazem **Reset FPSLICu** se provede vynulování všech důležitých ukazatelů, proměnných a nastavení v FPSLICu. Nejedná se tedy o reset v pravém slova smyslu, pouze vynulování. K úplnému resetu je potřeba použít příslušné tlačítko na vývojové desce. V každém případě se po tomto resetu smažou veškerá nahraná data (testované LUTy a nahrané vektory). Před vlastním měřením a nahráním LUTů je dobré vždy poslat tento příkaz resetu. Zároveň se tímto způsobem ověří, zda přípravek reaguje na příkazy po sériové lince.

Příkazy **Zjisti počet nahraných poruch** a **Zjisti počet nahraných vektorů** snad není potřeba podrobněji rozebírat. Program pouze pošle dotaz do FPSLICu a vrácený výsledek se vždy vypíše do textové oblasti.

Příkazem **Nahraj testované LUTy** se do FPSLICu nahrají všechny vybrané buňky (o tom, jak se vyberou, bude zmínka později). Je nutné si uvědomit, že nahrávání LUTů probíhá

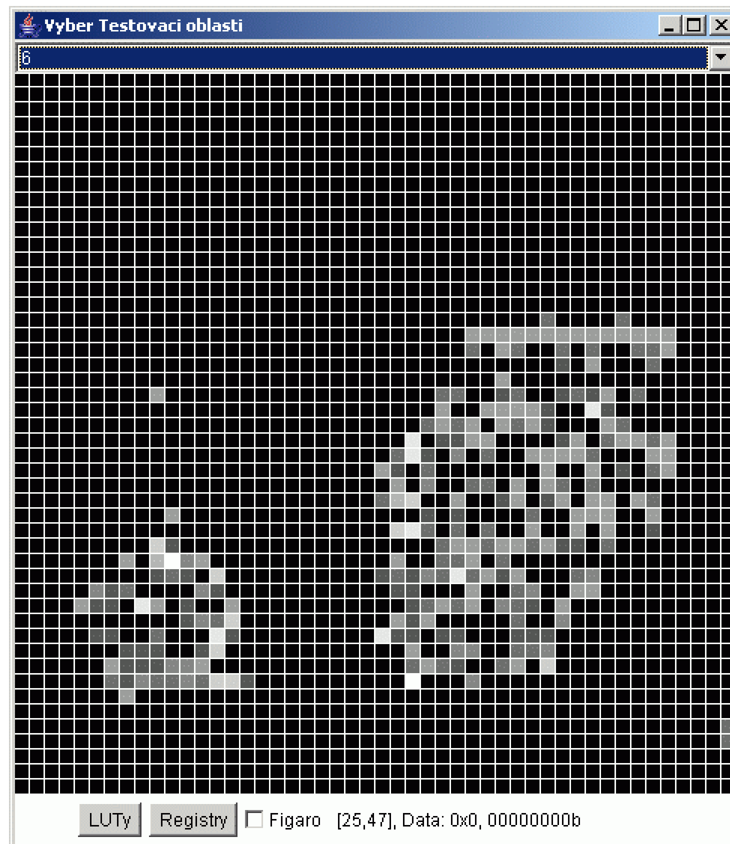
přírůstkově, tj. že se nemažou dříve nahrané testované buňky. Mazání obsahu paměti se dělá prostřednictvím resetu.



Obrázek 6.20: Menu Testy

Položka z menu Testů **Výběr testovací oblasti** se aktivuje až po otevření bitstreamu MD4. Po otevření se automaticky aktivuje okno s grafickým výběrem testovací oblasti, jak je vidět na obrázku 6.21. Příkazem **Klasifikace aktuálního obsahu** se pouze otestuje, do které kategorie spadá aktuální obsah. Pokud je vše v pořádku, měla by vždy vyjít kategorie A (0). Dále se v menu testy nacházejí zaškrtávací parametry, které se nahrávají před každým testováním. Test všech nahraných poruch se spouští příkazem **Test všech LUTů**, výsledkem je statistika, kolik poruch spadlo do jaké kategorie.

Dávkový test je spíše pracovní příkaz, do něhož se dá naprogramovat libovolná posloupnost příkazů. V současné implementaci jsou to 4 testy po sobě s různou kombinací parametrů.

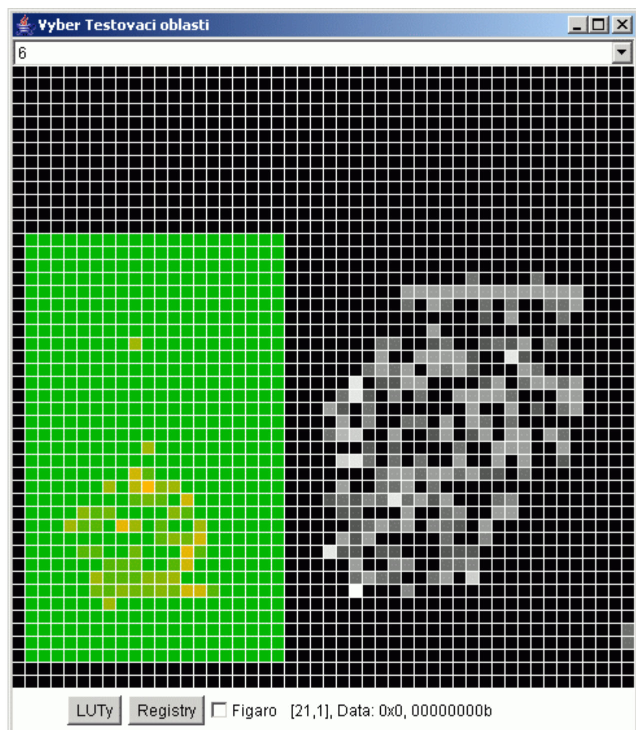


Obrázek 6.21: Okno Výběr testovací oblasti

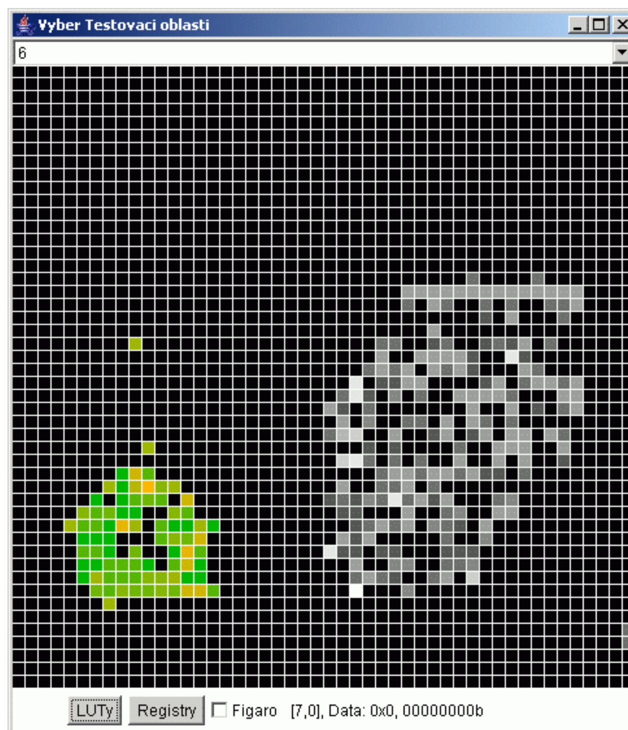
V okně **Výběr testovací oblasti** (obrázek 6.21) je graficky znázorněna hustota bitů v bitstreamu pro **vrstvu Z** zadanou v první řádce. Slouží k přibližné orientaci rozmístění v hradlovém poli. Orientace zobrazení je stejná s programem FIGARO IDS. Zde je možné udělat srovnání s obrázkem 6.14. Jedná se o stejný design. Barva odpovídá počtu jedničkových bitů bitstreamu (černá = žádný jedničkový bit, bílá = všechny bity jedničkové).

Myší je možné dělat výběr oblastí. Tyto oblasti jsou barevně odlišeny od ostatních buněk, jak je vidět na obrázku 6.22. Levým tlačítkem myši se přidává oblast, pravým tlačítkem myši se naopak odznačuje. Prostřední tlačítko má speciální funkci. Vypíše bitstream vztahující se k aktuální buňce do textového pole hlavního okna.

V okně výběru testovací oblastí se nacházejí dvě tlačítka. Tlačítko **LUTy** omezí výběr buněk pouze na ty, jejichž LUTy mají nenulový obsah (čili pokud jsou využité). Obdobně funguje tlačítko **Registry**, které omezí výběr pouze na ty buňky, jejichž výsledek je hradlován hodinami. Pokud před stisknutím některého z těchto tlačítek není vybrána žádná buňka, tak je výsledek stejný, jako by před stisknutím tlačítka byly vybrány všechny buňky.



Obrázek 6.22: Výběr oblasti myší



Obrázek 6.23: Omezení výběru

Zaškrtávací políčko **Figaro** změní způsob číslování buněk. V nezaškrtnutém stavu jsou buňky číslovány od čísla 0, jak tomu je v bitstreamu. Pokud bychom se chtěli současně dívat do rozmístění v nástroji FIGARO IDS, tak toto zaškrtávací políčko sjednocuje číslování právě s programem FIGARO IDS, který čísluje buňky od čísla 1. Usnadňuje, ale hlavně zrychluje to orientaci v obou programech.

Dále se ve spodní liště vypisují souřadnice a byte bitstreamu (vypsaný hexadecimálně a binárně, který odpovídá souřadnici a hodnotě zvolené vrstvy Z.

7 Testování a výsledky

Zde popsaným postupem byl vytvořen bitstream mnoha benchmarků. Všechny benchmarky, až na jedinou výjimku, se podařilo vložit do návrhu a vytvořit k nim příslušné EDIF soubory připravené na umístění do FPSLICu. Jedinou, již zmíněnou výjimkou, byl benchmark s298, u něhož byl velmi velký časový problém syntetizovat prediktor parity. Výsledný prediktor byl nakonec tak obrovský, že neměl sebemenší šanci se vejít do hradlového pole.

Pro 2 benchmarky se mi nepodařilo vytvořit bitstream. První byl benchmarl AL21, s jehož syntetizovanou formou EDIF si nedokázal FIGARO IDS poradit. Další neúspěšný benchmark je ALU4, který se nepodařilo rozmístit, neboť se do FPSLICu nevešel.

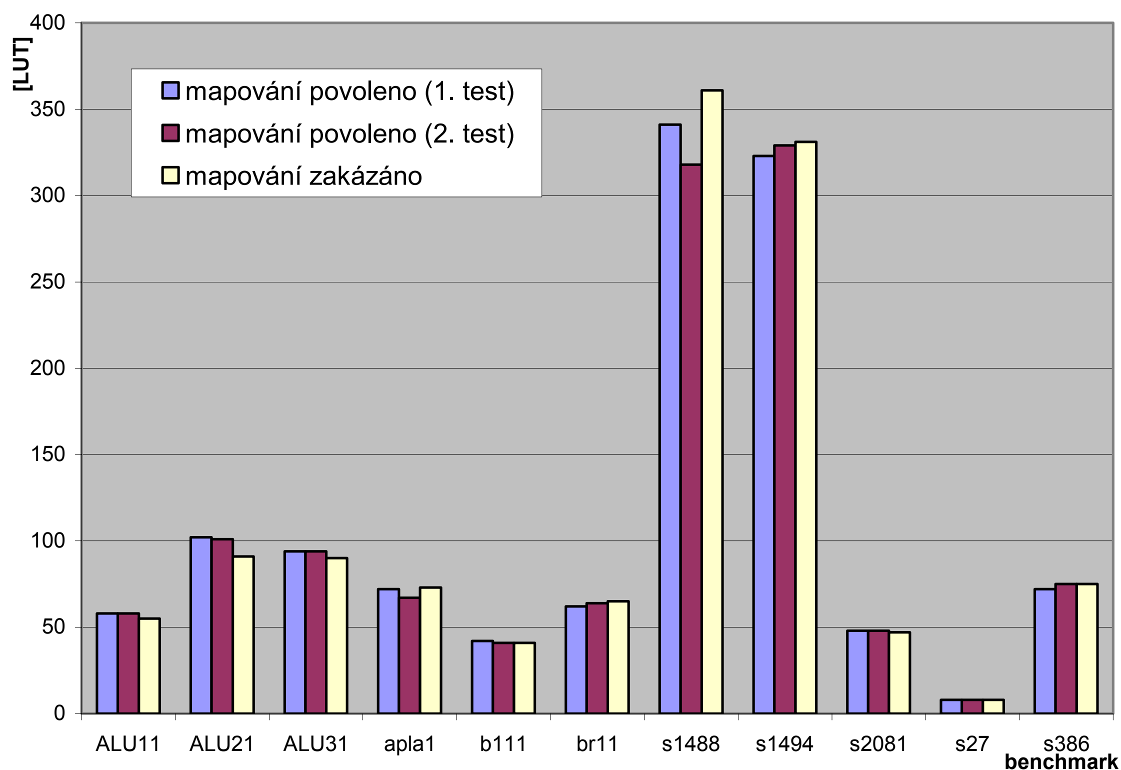
7.1 Vliv mapování na parametry návrhu

Během testování byl z vybrané skupiny benchmarků vytvořen bitstream vícekrát. Došel jsem k několika zajímavým zjištěním:

1. Povolené mapování ve FIGARO IDS neznamená nutně zmenšení a optimalizaci návrhu, v některých případech je tomu právě naopak (benchmarky ALU). Zdá se však, že pro některé benchmarky (s1488, s1494) k úspoře místa dojde. Data jsou v tabulce 7.1 a též vynesená do grafu jsou na obrázku 7.1. Testy s povoleným mapováním byly provedeny 2x stejným způsobem, přesto výsledek skončil pokaždé trochu jinak.

Benchmark	mapování povoleno (1. test)	mapování povoleno (2. test)	mapování zakázáno
ALU11	58	58	55
ALU21	102	101	91
ALU31	94	94	90
apla1	72	67	73
b111	42	41	41
br11	62	64	65
s1488	341	318	361
s1494	323	329	331
s2081	48	48	47
s27	8	8	8
s386	72	75	75

Tabulka 7.1: Závislost velikosti makra na nastavení mapování

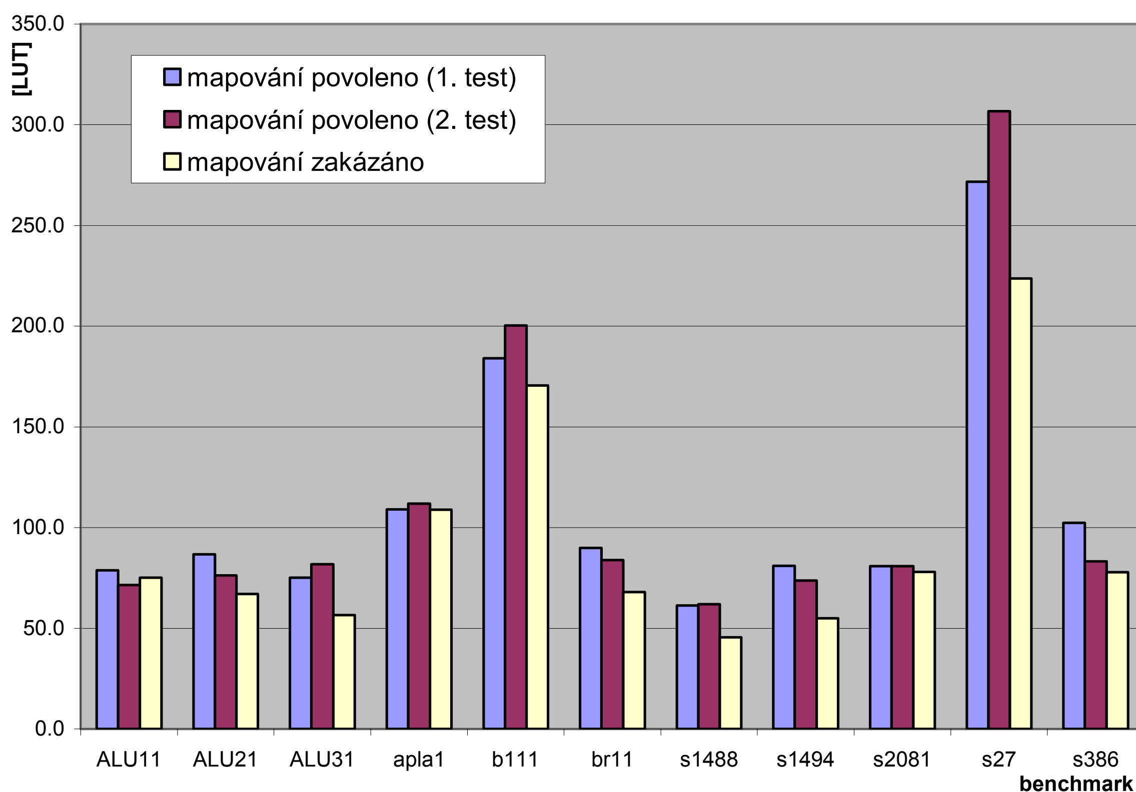


Obrázek 7.1: Graf závislosti velikosti benchmarku na nastavení mapování

2. Pokud je ve FIGARO IDS povoleno mapování, tak výsledný návrh makra benchmarku měl rychlejší odezvu, tj. šel ve většině případů provozovat na vyšší frekvenci. Data jsou v tabulce 7,2 a též vynesená do grafu jsou na obrázku 7.2.

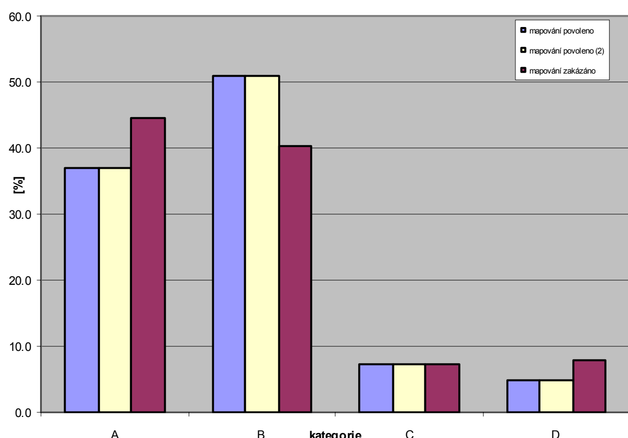
Benchmark	mapování povoleno (1. test)	mapování poveoleno (2. test)	mapování zakázáno
ALU11	78.8	71.5	75.1
ALU21	86.7	76.2	67.1
ALU31	75.1	81.8	56.6
apla1	109.0	111.9	108.9
b111	184.0	200.4	170.6
br11	89.8	83.8	68.0
s1488	61.4	62.0	45.5
s1494	81.0	73.7	55.0
s2081	80.8	80.8	78.0
s27	271.7	306.7	223.7
s386	102.4	83.2	77.9

Tabulka 7.2: Mezní frekvence benchmarků v závislosti na nastavení mapování

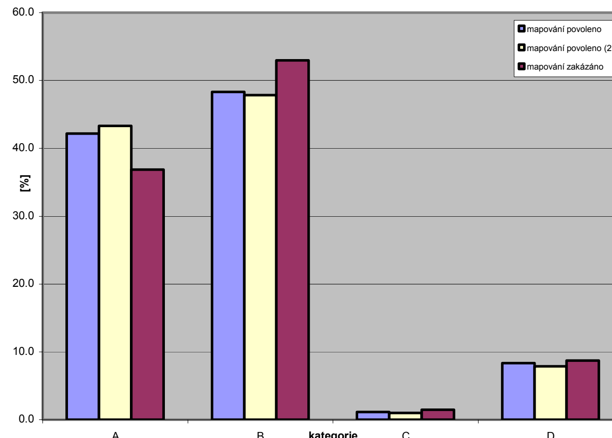


Obrázek 7.2: Graf mezní frekvence benchmarků v závislosti na nastavení mapování

3. Povolení či zakázání mapování může ovlivňovat statistické rozložení kategorií poruch, v celkovém měřítku i více jak o 10%. Ze sady benchmarků jsem vybral dva, viz obrázek 7.3 a 7.4. Na příkladu těchto 2 grafů jsem chtěl ukázat, že se mohou vlivem mapování kategorie změnit libovolně. Nedá se paušálně říci, že povolené mapování například zvyšuje počet nedetekovatelných chyb. U každého benchmarku je vlivem mapování změna rozložení kategorií jiná, na obrázcích jsou vybrané extrémní případy.



Obrázek 7.3: B111 – vliv mapování



Obrázek 7.4: s1488 – vliv mapování

7.2 Výsledky testování benchmarků pro sudou paritu

benchmark	input	output	parity	LUT	parita	overhead [%]	Celkový pořet poruch	A (skryté poruchy)	B (detekované poruchy)	C (Nedetekovatelné poruchy)	D (částečně detekovatelné poruchy)	ST pokrytí [%]	FS pokrytí [%]	čas [s]
ALU11	12	8	1	8	47	587.5	656	**0**	**656**	**0**	**0**	100.00	100.00	**0.68**
ALU21	10	8	1	44	47	106.8	1072	**109**	**935**	**0**	**28**	89.83	97.39	**0.29**
ALU31	10	8	1	45	45	100.0	1044	**130**	**877**	**8**	**29**	86.78	96.46	**0.29**
apla1	10	12	1	48	25	52.1	900	**141**	**625**	**5**	**129**	83.78	85.11	**0.25**
br11	12	8	1	50	15	30.0	810	**141**	**456**	**69**	**144**	74.07	73.70	**0.84**
s1488	14	25	1	310	50	16.1	4286	**638**	**3060**	**85**	**503**	83.13	86.28	**17.64**
s1494	14	25	1	276	53	19.2	3938	**645**	**2785**	**67**	**441**	81.92	87.10	**16.21**
s2081	18	9	1	22	25	113.6	536	**22**	**494**	**0**	**20**	95.90	96.27	**35.14**
s386	13	13	1	57	18	31.6	976	**170**	**646**	**25**	**135**	80.02	83.61	**2.02**

Tabulka 7.3: Výsledky testování vybraných částí LUTů (bez routovacích buněk)

V tabulce 7.3 jsou uvedeny výsledky měření, ve kterém je injekce poruch omezena pouze na lokality, které jsou fyzicky využity. Týká se to především buněk, jejichž LUTy generují funkci méně jak 4 proměnných. V nevyužitých částech LUTU je pak zcela zřejmé, že injekované poruchy se nemohou projevit.

benchmark	input	output	parity	LUT	parita	overhead [%]	Celkový pořet poruch	A (skryté poruchy)	B (detekované poruchy)	C (Nedetekovatelné poruchy)	D (částečně detekovatelné poruchy)	ST pokrytí [%]	FS pokrytí [%]
ALU11	12	8	1	8	47	587.5	880	**224**	**656**	**0**	**0**	74.55	100.00
ALU21	10	8	1	44	47	106.8	1456	**493**	**935**	**0**	**28**	66.14	98.08
ALU31	10	8	1	45	45	100.0	1440	**526**	**877**	**8**	**29**	62.92	97.43
apla1	10	12	1	48	25	52.1	1168	**409**	**625**	**5**	**129**	64.55	88.53
br11	12	8	1	50	15	30.0	1040	**371**	**456**	**69**	**144**	57.69	79.52
s1488	14	25	1	310	50	16.1	5776	**2128**	**3060**	**85**	**503**	61.69	89.82
s1494	14	25	1	276	53	19.2	5296	**2003**	**2785**	**67**	**441**	60.91	90.41
s2081	18	9	1	22	25	113.6	752	**238**	**494**	**0**	**20**	68.35	97.34
s386	13	13	1	57	18	31.6	1200	**394**	**646**	**25**	**135**	65.08	86.67

Tabulka 7.4: Výsledky testování celých LUTů (bez routovacích buněk)

V další tabulce 7.4 jsou výsledky měření LUTů pro všechny bity LUTu. Toto měření je v jistém smyslu reprezentativnější, jelikož jakmile nějaká funkce zabrala LUT (byť jenom část), tak se zabrala celá buňka a bitstream celého LUTu se rezervoval pro danou funkci. Neboli nevyužívané části, které jsou v tomto testu také podrobeny testování, již nemohou být využity pro jinou funkci.

Z porovnání tabulek 7.4 a 7.3 je dobře patrné, že výsledky se liší pouze v absolutních číslech kategorie A, což bylo přesně podle očekávání. Jaký je procentuální podíl zmiňovaných nevyužívaných částí LUTu je vypsáno pro vybrané benchmarky v tabulce 7.5. Pohybuje se v rozmezí zhruba 15 až 40%, typicky kolem 25%

Benchmark	nevyužití LUTů [%]
ALU11	25.5
ALU21	26.4
ALU31	27.5
apla1	22.9
br11	22.1
s1488	25.8
s1494	25.6
s2081	28.7
s386	18.7

Tabulka 7.5: Podíl nevyužívaných částí LUTů

V tabulkách 7.6 a 7.7 je zaznamenáno procentuální rozdělení jednotlivých kategorií (A,B,C a D), jednou s testováním úplně celých LUTů a jednou s přihlédnutím k obsazení LUTů pouze používané bity LUTu. Oboje tabulky neberou v úvahu LUTy v routovacích buňkách.

benchmark	A (skryté poruchy) [%]	B (detekované poruchy) [%]	C (Nedetekovatelné poruchy) [%]	D (částečně detekovatelné poruchy) [%]
ALU11	0.0	100.0	0.0	0.0
ALU21	10.2	87.2	0.0	2.6
ALU31	12.5	84.0	0.8	2.8
apla1	15.7	69.4	0.6	14.3
br11	17.4	56.3	8.5	17.8
s1488	14.9	71.4	2.0	11.7
s1494	16.4	70.7	1.7	11.2
s2081	4.1	92.2	0.0	3.7
s386	17.4	66.2	2.6	13.8

Tabulka 7.6: Výsledky testování vybraných částí LUTů v procentech

benchmark	A (skryté poruchy) [%]	B (detekované poruchy) [%]	C (Nedetekovatelné poruchy) [%]	D (částečně detekovatelné poruchy) [%]
ALU11	25.5	74.5	0.0	0.0
ALU21	33.9	64.2	0.0	1.9
ALU31	36.5	60.9	0.6	2.0
apla1	35.0	53.5	0.4	11.0
br11	35.7	43.8	6.6	13.8
s1488	36.8	53.0	1.5	8.7
s1494	37.8	52.6	1.3	8.3
s2081	31.6	65.7	0.0	2.7
s386	32.8	53.8	2.1	11.3

Tabulka 7.7: Výsledky testování kompletních LUTů v procentech

7.2.1 Routovací buňky

Některé buňky jsou využity pouze pro propojení signálů. Takovéto buňky lze snadno poznat podle zapojení signálů a obsahu LUT tabulek. Využívají pouze 2 bity z LUT tabulky, ostatní jsou nevyužity a zákonitě pak poruchy injekované do nevyužitých míst se neprojeví. Proto je mezi routovacími buňkami tak velké zastoupení kategorie A.

Poněkud výjimečná situace nastala u benchmarku B12, kdy se chyba projevila pouze v 1 bitu. To znamená, že daný signál je konstantní pro všechny možné vstupní vektory. Soudím, že se jedná o nedokonalost syntézního nástroje Synplify pro, který nedokázal detekovat generátor trvalé hodnoty. Došel jsem k závěru, že se jednalo o dosti velkou náhodu, že se takto přímo povedlo signál odhalit.

Výsledky poruch vkládaných do routovacích buněk jsou zaznamenány v tabulce 7.8

benchmark	Celkový pořet poruch	A (skryté poruchy)	B (detekované poruchy)	C (Nedetekovatelné poruchy)	D (částečně detekovatelné poruchy)	čas [s]
ALU11	32	**28**	**4**	**0**	**0**	0.033
ALU21	80	**70**	**10**	**0**	**0**	0.020
ALU31	112	**98**	**14**	**0**	**0**	0.029
apla1	144	**126**	**10**	**0**	**8**	0.037
br11	32	**28**	**2**	**0**	**2**	0.033
s1488	400	**350**	**31**	**1**	**18**	1.638
s1494	672	**588**	**54**	**2**	**28**	2.753
s2081	32	**28**	**4**	**0**	**0**	2.097
s386	80	**70**	**5**	**0**	**5**	0.164

Tabulka 7.8: Výsledky testování LUTů v routovacích buňkách

7.3 Čas měření

Na času celého měření se podílejí 3 složky:

1. generování všech vektorů triviálního testu v FPGA (doba $t_1 = \frac{2^n}{f}$, kde n je počet bitů testovacího vektoru a f je frekvence hradlového pole, v tomto případě 4 MHz),
2. běh programu v AVR (včetně injekce poruch),
3. komunikace po sériové lince (doba potřebná k nastavení parametrů a přijetí výsledků $t_2 = (n+11) * \frac{10}{38400}$, kde n je počet buněk nahraných v paměti AVR).

Naměřené časy zcela odpovídaly teoretickým předpokladům. Bod 2 (čas, který spotřebuje AVR na běh programu) byl přitom zanedbán, poněvadž oproti bodu 1 a 3 je tento čas velmi malý, jak ukazuje vlastní měření. Naměřené časy byly jen nepatrně vyšší a většinou se pohybovaly na hranici měřitelnosti v PC.

Tyto časy se týkají pouze generování testovacích vektorů pomocí navrženého generátoru úplného testu a jsou nezávislé na tom, jaký je použit bezpečnostní kód. Jsou též nezávislé na počtu bitů výstupního vektoru.

V tabulce 7.9 jsou uvedeny vedle sebe časy softwarové simulace [6], odhadovaný čas podle předchozí úvahy a skutečný naměření čas testování v FPSLICu. Rozdíl odhadovaného času a naměřeného času v HW je opravdu téměř zanedbatelný. Naproti tomu je zde vidět výrazné zrychlení testování v HW oproti softwarové simulaci (viz poslední sloupec).

	vstupních bitů	SW simulace [s]	HW - odhad výpočtem [s]	HW měření [s]	HW rozdíl [s]	SW/HW poměr časů
5xp1	7	-	0.03	0.04	0.01	-
ALU11	12	34.0	0.92	0.92	0.01	37.0
ALU21	10	9.0	0.40	0.41	0.02	22.0
ALU31	10	6.9	0.39	0.41	0.03	16.8
apla1	10	6.0	0.32	0.34	0.02	17.6
b111	8	0.8	0.05	0.06	0.00	13.3
b12	15	-	6.31	6.32	0.02	-
br11	12	18.0	1.08	1.10	0.02	16.4
bw	5	-	0.03	0.06	0.04	-
f51m	8	-	0.04	0.05	0.02	-
misex1	8	-	0.03	0.04	0.01	-
s1488	14	2406.3	23.76	23.82	0.08	101.0
s1494	14	2518.9	21.79	21.84	0.07	115.3
s2081	18	1217.9	49.30	49.30	0.01	24.7
s27	7	0.1	0.01	0.03	0.02	3.3
s386	13	677.7	2.48	2.49	0.02	272.2
sao2	10	50	0.35	0.36	0.02	138.9

Tabulka 7.9: Časy měření

7.4 Porovnání výsledků

	HW (všechny bity)				HW (použité bity)				SW			
	A[%]	B[%]	C[%]	D[%]	A[%]	B[%]	C[%]	D[%]	A[%]	B[%]	C[%]	D[%]
ALU11	25.5	74.5	0.0	0.0	0.0	100.0	0.0	0.0	0.0	100.0	0.0	0.0
ALU21	33.9	64.2	0.0	1.9	10.2	87.2	0.0	2.6	0.0	92.5	0.0	7.5
ALU31	36.5	60.9	0.6	2.0	12.5	84.0	0.8	2.8	0.0	90.3	0.0	9.7
apla1	35.0	53.5	0.4	11.0	15.7	69.4	0.6	14.3	0.0	82.8	0.2	17.1
b111	44.5	40.4	7.3	7.8	4.7	69.4	12.6	13.4	0.5	75.5	10.0	14.0
br11	35.7	43.8	6.6	13.8	17.4	56.3	8.5	17.8	0.0	62.9	8.7	28.4
s1488	36.8	53.0	1.5	8.7	14.9	71.4	2.0	11.7	1.6	86.3	2.7	9.3
s1494	37.8	52.6	1.3	8.3	16.4	70.7	1.7	11.2	2.1	86.3	1.4	10.2
s27	45.3	32.8	9.4	12.5	5.4	56.8	16.2	21.6	0.0	72.2	13.9	13.9
s386	32.8	53.8	2.1	11.3	17.4	66.2	2.6	13.8	0.0	71.1	2.3	26.7

Tabulka 7.10: Porovnání výsledků SW a HW testování

V tabulce 7.10 jsem vedle sebe postavil výsledky měření provedeném v hradlovém poli (HW) a výsledky ze simulace softwarem. Z tabulek lze na první pohled vyvodit několik poznatků:

1. Při testování v hradlovém poli rapidně vzrostl počet skrytých poruch, které se nikdy neprojeví chybou (kategorie A). Vysvětlením je struktura hradlového pole a LUTů, které vzhledem k odlišnosti struktury mohou být všechny plně využity jen ve vzácných případech. Oproti tomu SW simulace vychází pouze ze struktury EDIF
2. Ve většině případů klesl počet poruch, které se někdy projeví a někdy neprojeví (kategorie D).

8 Zhodnocení

Povedlo se vytvořit implementační prototyp systému na měření odolnosti proti poruchám v bitstreamu. Tento prototyp má podobu VHDL kódu a VHDL šablon, ke kterým byl vytvořen podpůrný software, který ze šablon vygeneruje VHDL kód pro libovolný obvod, což bylo ověřeno na benchmarcích uvedených v předchozí kapitole.

Povedlo se maximálně zautomatizovat generování VHDL kódu pro sudou paritu a skupiny sudých parit. Tato automatizace má podobu obslužných programů ovládaných z příkazové řádky a končí až generováním souborů EDIF, což je vstupní formát place&route nástroje pro hradlová pole ATMEL. Zde automatice končí a nastupuje podrobně popsaný postup, jak vytvořit bitstream. Automatizaci ručního rozmístění v hradlovém poli bohužel FIGARO IDS nepodporuje, což je na tento program nadměrný požadavek. Program sice podporuje určitý soubor TCL příkazů, pro potřeby tohoto návrhu ale nejsou postačující. Program pro sjednocení bitstreamu hradlového pole a AVR kódu, System Designer je z příkazové řádky zcela neovladatelný, proto se musí tento krok nevyhnutelně dělat manuálně.

Velmi náročnou součástí této práce byla analýza bitstreamu pro zjištění základních informací, jako je např. umístění LUTů v bitstreamu a dále propojení signálů v buňce v souvztažnosti s konfigurací bitů v bitstreamu. Informace o významu těchto bitů nejsou firmou ATMEL veřejně dostupné a je možné je oficiálně získat pouze pod smlouvou NDA (Non-disclosure Agreement). Výsledné poznatky o bitstreamu byly použity v kódu zabudovaného AVR k rozhodování o obsazení buněk a k injekcím poruch, ovšem právě z důvodu oficiálního utajení obsahu bitstreamu jsem nemohl podrobně popsat význam bitstreamu, na který jsem přišel.

8.1 Doporučení pro další postup

Na závěr bych rád naznačil několik dalších směrů, kam by mohl směřovat další vývoj:

1. V rámci této diplomové práce byly testovány pouze „bezpečné poruchy“. Bezpečné proto, že nemohou vyvolat zkrat. Poruchy se vždy týkaly pouze změny ve funkci generované LUTy. LUTy jsou však pouze malá část bitstreamu (odhadem 11%), daleko více prostoru v bitstreamu zabírá konfigurace propojení v rámci buňky (multiplexory na obrázku 3.4 a propojení na obrázku 3.3. Bylo by dobré rozšířit obor testovaných částí bitstreamu i na propojení, abychom získali přesnější představu o vlivu SEU na testovaný obvod. V případě rozšíření oboru testovaných poruch je ale potřeba dávat si daleko větší pozor na to, aby se obvod nezničil.
2. Sudá parita je nejjednodušší možný bezpečnostní kód. Jistě by bylo zajímavé rozšířit testování i o jiné bezpečností kódy a srovnání výsledků. S možností jiného bezpečnostního kódu jsem počítal již v současné implementaci. Jediná věc, která je potřeba upravit, je modul Checker, tzn. napsat jeho variantu pro jiný bezpečnostní kód.
3. V současné implementaci je pouze generátor úplného testu. Pro počet bitů vstupních vodičů větší jak 20 je tento test časově velmi náročný. V návrhu bylo sice počítáno s vkládáním vlastních testovacích vektorů, ovšem tato funkce zůstala zatím neimplementovaná, jelikož jsem předpokládal poměrně velký datový tok přes sériovou linku (testovací vektory). Bylo by zajímavé prozkoumat jiný způsob generování testovacích vektorů, např. pomocí LFSR, byť by se nejednalo o úplný test. S tímto rozšířením je již od začátku počítáno v podobě modulárnosti generátoru testovacích vektorů.

9 Závěr

Výsledkem této diplomové práce bylo kompletní funkční prostředí pro testování a klasifikaci poruch v hradlovém poli FPSLIC. Toto prostředí v sobě zahrnuje implementaci řídící logiky (a vytvoření programů pro její generování), metodiky vytvoření bitstreamu a vkládání poruch pomocí dynamické rekonfigurace v hradlovém poli.

S využitím částečné rekonfigurace se podařilo urychlit testování klasifikace poruch oproti softwarové simulaci[6] i 100x (závisí především na velikosti benchmarku).

Během návrhu jsem se detailně seznámil s hradlovým polem firmy ATMEL, včetně objevení souvislosti mezi bitstreamem a strukturou buňky FPSLICu, což je nutné znát k provádění rekonfigurace. FPSLIC se ukázal jako velmi kvalitní obvod právě díky možnosti své rekonfigurace a jejího pohodlného řízení z vestavěného jádra AVR. Rekonfiguraci v FPSLICu je navíc možné provádět po velmi malých částech (po bytech), na rozdíl od architektury VIRTEX, rekonfigurovatelné pouze po sloupcích [11].

Úspěšně se povedlo otestovat 17 různých benchmarků zabezpečených sudou paritou. Zjistil jsem výrazně vyšší počet skrytých poruch (to jsou takové poruchy, které nemají vliv na funkci obvodu) oproti softwarově provedené simulaci[6]. V ostatních kategoriích poruch se výsledky příliš neliší od softwarové simulace.

A Literatura a zdroje

[1] Single Event Upsets in FPGAs, QuickLogic Corporation
http://www.quicklogic.com/images/Single_Event_Upsets_in_FPGAs.pdf

[2] NASA Office of Logic Design: Programmable Logic in the Radiation Environment
http://klabs.org/richcontent/Tutorial/MiniCourses/radiation_mapld_2002/programmable_logic_in_the-radiation_environment.htm

[3] Mitra S, McCluskey E. J.: Which concurrent error detection scheme to choose? Proceedings of the International Test Conference, 2000, p. 985-994

[4] Kafka L.: Návrh TSC obvodů v FPGA, diplomová práce 2004

[5] Hlavička J., Racek S., Golan P., Blažek T.: Číslicové systémy odolné proti poruchám, 1992

[6] KAFKA L., KUBALÍK P., KUBÁTOVÁ H., NOVÁK O.: Fault Classification for Self-checking Circuits Implemented in FPGA, Proceedings of IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop. Sopron: University of Western Hungary, 2005, p. 228-231. ISBN 963 9364 48 7.

[7] AT94K05/10/40AL Datasheet
http://www.atmel.com/dyn/resources/prod_documents/doc1138.pdf

[8] AT40K Series Configuration – application note
http://www.atmel.com/dyn/resources/prod_documents/DOC1009.PDF

[9] AT94K Series Configuration – application note
http://www.atmel.com/dyn/resources/prod_documents/DOC2313.PDF

[10] Kubalík P., Kubátová H.: Design of Self Checking Circuits Based on FPGA, Proceedings of the 15th International Conference on Microelectronics. Cairo: Cairo University, 2003, p. 378-381. ISBN 977-05-2010-1.

[11] Virtex Series Configuration Architecture User Guide, Apllication note XAPP151, http://www.xilinx.com

[12] Graham P., Caffrey M., Zimmerman J., Sundararajan P., Johnson E., and Patterson C.: Consequences and Categories of SRAM FPGA Configuration SEUs, 2003

[13] Kubalík P., Kubátová H.: Highly Reliable Design Based on TSC Circuits, Počítačové architektury & diagnostika. Prague: CTU, Faculty of Electrical Engineering, Department of Computer Science and Engineering, 2005, vol. 1, p. 101-106. ISBN 80-01-03298-1.

[14] Kubalík P., Kubátová H.: Reconfigurable Duplex System Increasing Fault Tolerance for Circuits Based on FPGAs, Proceedings of the Work in Progress Session. Linz: Johannes Kepler University, 2005, p. 13-14. ISBN 3-902457-09-0.

[15] Kubalík P., Kubátová H.: On-line Testing for FPGA (Paper in Conference Proceedings), Proceedings of the Sixth International Scientific Conference Electronic Computers and Informatics ECI 2004. Košice: Department of Computers and Informatics of FEI, Technical University Košice, 2004, s. 194-199. ISBN 80-8073-150-0.

[16] System Designer 3.0 + FIGARO IDS 7.6, http://ww.atmel.com/

[17] Java development kit (JDK) 5.0, http://java.sun.com/

[18] Eclipse, http://www.eclipse.org/

[19] CDT - C/C++ Development Tools (Eclipse plugin), http://www.eclipse.org/cdt/

[20] RXTX : serial and parallel I/O libraries supporting Sun's CommAPI, http://www.rxtx.org/

[21] Synplify Pro, http://www.synplicity.com/

[22] gcc, http://gcc.gnu.org/

[23] WinAVR, avr-gcc for windows, http://winavr.sourceforge.net/

[24] AVR studio, http://www.atmel.com/

[25] BOOM, http://service.felk.cvut.cz/vlsi/prj/BOOM/

[26] ESPRESSO, http://service.felk.cvut.cz/vlsi/Download/

[27] Kubalík P., Kubátová H.: Design Methodology for High Reliable System, 2006

B Seznam použitých zkratek

MB – megabyte
Mb – megabit
b – bit
B – byte
b/s – bitů za sekundu
FPGA – Field Programmable Gate Array
VHDL – Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
LUT – Look-up table
FPSLIC – Field Programmable System Level Integrated Circuits
LFSR – Linear Feedback Shift Register
NDA – Non-disclosure Agreement
EDIF – Electronic Design Interchange Format
SEU – Single Event Upset

C Úplné výsledky

benchmark	input	output	parity	LUT	parita	overhead [%]	Celkový pořet poruch	A (skryté poruchy)	B (detekované poruchy)	C (Nedetekovatelné poruchy)	D (částečně detekovatelné poruchy)	ST pokrytí [%]	FS pokrytí [%]	čas [s]
5xp1	7	10	1	22	12	54.5	374	18	312	16	28	90.91	88.24	0.02
ALU11	12	8	1	8	47	587.5	656	0	656	0	0	100.00	100.00	0.68
ALU21	10	8	1	44	47	106.8	1072	109	935	0	28	89.83	97.39	0.29
ALU31	10	8	1	45	45	100.0	1044	130	877	8	29	86.78	96.46	0.29
apla1	10	12	1	48	25	52.1	900	141	625	5	129	83.78	85.11	0.25
b111	8	31	1	37	4	10.8	382	18	265	48	51	82.72	74.08	0.03
b12	15	9	1	28	20	71.4	624	58	550	0	16	90.71	97.44	5.12
br11	12	8	1	50	15	30.0	810	141	456	69	144	74.07	73.70	0.84
bw	5	28	1	63	2	3.2	802	145	546	35	76	77.56	86.16	0.02
f51m	8	8	1	19	13	68.4	398	23	319	0	56	94.22	85.93	0.03
misex1	8	7	1	19	5	26.3	320	49	195	51	25	68.75	76.25	0.03
s1488	14	25	1	310	50	16.1	4286	638	3060	85	503	83.13	86.28	17.64
s1494	14	25	1	276	53	19.2	3938	645	2785	67	441	81.92	87.10	16.21
s2081	18	9	1	22	25	113.6	536	22	494	0	20	95.90	96.27	35.14
s27	7	4	1	5	3	60.0	74	4	42	12	16	78.38	62.16	0.00
s386	13	13	1	57	18	31.6	976	170	646	25	135	80.02	83.61	2.02
sao2	10	4	1	57	22	38.6	992	144	770	2	76	85.28	92.14	0.27

Tabulka 9.1: Výsledky testování pouze použitých bitů LUTu (bez routovacích buněk)

benchmark	input	output	parity	LUT	parita	overhead [%]	Celkový pořet poruch	A (skryté poruchy)	B (detekované poruchy)	C (Nedetekovatelné poruchy)	D (částečně detekovatelné poruchy)	ST pokrytí [%]	FS pokrytí [%]	čas [s]
5xp1	7	10	1	22	12	54.5	544	**188**	**312**	**16**	**28**	62.50	91.91	**0.02**
ALU11	12	8	1	8	47	587.5	880	**224**	**656**	**0**	**0**	74.55	100.00	**0.91**
ALU21	10	8	1	44	47	106.8	1456	**493**	**935**	**0**	**28**	66.14	98.08	**0.39**
ALU31	10	8	1	45	45	100.0	1440	**526**	**877**	**8**	**29**	62.92	97.43	**0.39**
apla1	10	12	1	48	25	52.1	1168	**409**	**625**	**5**	**129**	64.55	88.53	**0.32**
b111	8	31	1	37	4	10.8	656	**292**	**265**	**48**	**51**	48.17	84.91	**0.05**
b12	15	9	1	28	20	71.4	768	**202**	**550**	**0**	**16**	73.70	97.92	**6.30**
br11	12	8	1	50	15	30.0	1040	**371**	**456**	**69**	**144**	57.69	79.52	**1.08**
bw	5	28	1	63	2	3.2	1040	**383**	**546**	**35**	**76**	59.81	89.33	**0.02**
f51m	8	8	1	19	13	68.4	512	**137**	**319**	**0**	**56**	73.24	89.06	**0.04**
misex1	8	7	1	19	5	26.3	384	**113**	**195**	**51**	**25**	57.29	80.21	**0.03**
s1488	14	25	1	310	50	16.1	5776	**2128**	**3060**	**85**	**503**	61.69	89.82	**23.74**
s1494	14	25	1	276	53	19.2	5296	**2003**	**2785**	**67**	**441**	60.91	90.41	**21.77**
s2081	18	9	1	22	25	113.6	752	**238**	**494**	**0**	**20**	68.35	97.34	**49.29**
s27	7	4	1	5	3	60.0	128	**58**	**42**	**12**	**16**	45.31	78.13	**0.01**
s386	13	13	1	57	18	31.6	1200	**394**	**646**	**25**	**135**	65.08	86.67	**2.47**
sao2	10	4	1	57	22	38.6	1264	**416**	**770**	**2**	**76**	66.93	93.83	**0.34**

Tabulka 9.2: Výsledky testování všech bitů LUTu (bez routovacích buněk)

benchmark	input	output	parity	LUT	parita	overhead [%]	Celkový pořet poruch	A (skryté poruchy)	B (detekované poruchy)	C (Nedetekovatelné poruchy)	D (částečně detekovatelné poruchy)	čas [s]
5xp1	7	10	1	22	12	54.5	16	**14**	**2**	**0**	**0**	0.001
ALU11	12	8	1	8	47	587.5	32	**28**	**4**	**0**	**0**	0.033
ALU21	10	8	1	44	47	106.8	80	**70**	**10**	**0**	**0**	0.020
ALU31	10	8	1	45	45	100.0	112	**98**	**14**	**0**	**0**	0.029
apla1	10	12	1	48	25	52.1	144	**126**	**10**	**0**	**8**	0.037
b111	8	31	1	37	4	10.8	0	**0**	**0**	**0**	**0**	0.000
b12	15	9	1	28	20	71.4	112	**99**	**11**	**0**	**2**	0.918
br11	12	8	1	50	15	30.0	32	**28**	**2**	**0**	**2**	0.033
bw	5	28	1	65	0	0.0	64	**56**	**6**	**0**	**2**	0.001
f51m	8	8	1	19	13	68.4	16	**14**	**2**	**0**	**0**	0.001
misex1	8	7	1	19	5	26.3	48	**42**	**6**	**0**	**0**	0.003
s1488	14	25	1	310	50	16.1	400	**350**	**31**	**1**	**18**	1.638
s1494	14	25	1	276	53	19.2	672	**588**	**54**	**2**	**28**	2.753
s2081	18	9	1	22	25	113.6	32	**28**	**4**	**0**	**0**	2.097
s27	7	4	1	5	3	60.0	0	**0**	**0**	**0**	**0**	0.000
s386	13	13	1	57	18	31.6	80	**70**	**5**	**0**	**5**	0.164
sao2	10	4	1	57	22	38.6	112	**98**	**12**	**0**	**2**	0.029

Tabulka 9.3: Výsledky testování LUTů routovacích buněk

D Mapa přiřazení signálů AVR-FPGA

```
AVR_IO_Selects (
   'IOSEL0'='IOSELA0'
   'IOSEL4'='IOSELA4'
   'IOSEL8'='IOSELA8'
   'IOSEL12'='IOSELA12'
)
AVR_Controls (
   'iowe'='FIOWEA'
)
Data_from_AVR (
   'iodatain(0)'='ADINA0'
   'iodatain(1)'='ADINA1'
   'iodatain(2)'='ADINA2'
   'iodatain(3)'='ADINA3'
   'iodatain(4)'='ADINA4'
   'iodatain(5)'='ADINA5'
   'iodatain(6)'='ADINA6'
   'iodatain(7)'='ADINA7'
)
Data_to_AVR (
   'iodataout(0)'='ADOUTA0'
   'iodataout(1)'='ADOUTA1'
   'iodataout(2)'='ADOUTA2'
   'iodataout(3)'='ADOUTA3'
   'iodataout(4)'='ADOUTA4'
   'iodataout(5)'='ADOUTA5'
   'iodataout(6)'='ADOUTA6'
   'iodataout(7)'='ADOUTA7'
)
AVR_Side_Clocks (
   'clk'='GCLK5'
)
```

E Obsah přiloženého CD

Adresář	**Popis**
/benchmarks/source	Zadané benchmarky s predikátorem parity ve formátu *.bench
/benchmarks/EDIF	Vytvořené benchmarky s testovacím prostředím pro FPSLIC
/benchmarks/placed	Rozmístěné benchmarky v FPSLICu s bitstreamy ve všech dostupných formátech
/dokumentace	Články a texty, ze kterých bylo čerpáno
/programy	Vytvořené programy v rámci této diplomové práce (zdrojové kódy i zkompilované, případně i spouštěcí dávky)
/programy/AVR	Ovládací program pro AVR
/programy/CorrectEdif	Program pro úpravu EDIF souboru
/programy/genFPGAVHDL	Program pro generování VHDL kódu pro implementaci v FPSLICu
/programy/TSC	Ovládací program pro PC
/software	Kopie instalačních souborů použitého software
/text	Tato diplomová práce