Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra číslicového návrhu

Diplomová práce

Generátor prostředků vestavěné diagnostiky

Bc. Robert Hülle

Vedoucí práce: Ing. Martin Daňhel

7. května 2015

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 7. května 2015 .....

Odkaz na tuto práci

Hülle, Robert. *Generátor prostředků vestavěné diagnostiky.* Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstrakt

V této práci implementuji algoritmické generování obvodů vestavěné diagnostiky pro obvod tvořený náhodnou kombinační logikou.

Klíčová slova vestavěná diagnostika, testování, generování testů, kombinační logika, generování logických obvodů

Abstract

In this work, I implement algorithmic generator of built-in self-test circuitry for random combinatoric logic.

Keywords built-in self-test, testing, test generating, combinatoric logic, logic circuit generation

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Testování logických obvodů je nutnost. Ať se již jedná o výrobní vady, či opotřebení v provozu a stárnutí. Je důležité odhalit chybný výrobek co nejdříve, protože cena opravy či výměny integrovaného obvodu velmi rychle stoupá.

Ze zvyšující se hustotou integrace se také zvětšuje velikost a složitost číslicového návrhu. To vede k neúnosnému nárůstu ceny a délky testu. Proto se využívá takzvaných vnitřních testů (built-in self-test), což jsou obvody, které dokáží otestovat samy sebe, nebo alespoň ulehčí externímu testování.

Takovýto vnitřní test se vyplatí i za cenu nárůstu plochy obvodu a jeho zpomalení. Vnitřní test může urychlit a zlevnit testování ve všech fázích života obvodu, od výroby až po použití v produkčním prostředí.

V této práci se zabývám automatickým generováním vnítřních testů a jejich obvodů.

Kapitola 1

Cíl práce

Cílem této práce je navrhnout a implementovat automatizovaný softwarový nástroj (SW), který bude pro vstupní logický obvod generovat vnitřní test (built-in self-test; BIST).

Generovaný test bude strukturní, s poruchovým modelem trvalá nula/trvalá jedna.

Vstupem programu bude kombinační logický obvod zadaný v podobě netlistu, ve formátu BLIF.

Výstupem programu bude logický obvod obsahující původní obvod, doplněný o logiku BIST, ve formátu BLIF a VHDL.

V této práci si tedy vytyčuji následující cíle:

- generování strukturního testu pro kombinační obvod,
- syntéza tohoto testu v podobě logického obvodu, nezasahujícího do testovaného obvodu,
- export této testovací logiky ve formátu BLIF,
- export této logiky ve formátu VHDL pro snadnější simulaci,
- zhodnocení efektivity vygenerovaného testu,
- navržení a implementace tohoto nástroje umožňující snadné rozšíření o dlaší funkce.

Kapitola 2

Analýza

2.1 Testování číslicových obvodů

2.1.1 Externí test

Externí test je systém a technika testování číslicových obvodů, kdy obvod samotný neobsahuje žádnou testovací logiku, ale test je uložen a vykonán v externím zařízení.

Stroje, které provádí externí testování na industriální úrovni jsou drahé a s rostoucí složitostí testovaných obvodů jejich pořizovací a provozní cena roste neúměrně.

2.1.2 Vnitřní test

Jedním řešením tohoto problému je použití vnitřního testu, tedy obvodu, který dokáže sám sebe, nebo svou část otestovat.

Vnitřní testy mají své vlastní problémy, zabírají plochu, která je v běžném provozu neužitečná, zpomalují vlastní běh funkční části obvodu.

Jejich výhodou je pak zlevnění testu při výrobě a zkrácení opravného cyklu při jejich produkční fázi života.

2.2 Generování testu

2.2.1 Funkční test

Funkční test se na testovaný obvod dívá jako na černou skříňku, o jejímž vnitřním uspořádání nic neví. Je známo pouze chování obvodu jako celku. Pokud se jedná o malý obvod, nebo neznáme jeho vnitřní strukturu, hodí se právě tento test.

Naopak pro velké obvody, zejména z pohledu počtu vstupů, ale i z pohledu počtu vnitřních stavů, je tento test nevhodný, neb jeho velikost a hlavně délka roste velmi rychle.

2.2.2 Strukturní test

Strukturní test se dívá na strukturu testovaného obvodu, bez ohledu na jeho funkci. Jedná se tedy o jakýsi opak funkčního testu.

Tento test využívá znalosti vnitřní struktury obvodu ke kompakci testovacích vektorů. To je umožněno výčtem možných poruch v obvodu a jejich pokrytí co nejmenším počtem vstupních vektorů, neboli nalezení právě těch vstupů, u kterých pokrytá porucha způsobí rozdílný výstup. Praktická kvalita takového testu pak závisí také na zvoleném poruchovém modelu.

2.2.3 Úrovně poruchových modelů

2.2.3.1 Defekt

Defekt je fyzický kaz obvodu. Je vzniklý z nedokonalé výroby, poškozením obvodu při provozu, stárnutím, či jinými náhodnými jevy jako jsou vysokoenergetické částice.

Příkladem defektu může být propálený tranzistor, smetí na substrátu, přerušená kovová vrstva, zkrat na křemíku, ...

2.2.3.2 Porucha

Porucha je model defektu na logické úrovni. Je způsobena defektem, ale defekt nemusí nutně vést k poruše. Existuje několik poruchových modelů, které pokrývají různé typy defektů.

Porucha může, ale nemusí, vést k chybě v obvodu.

2.2.3.3 Chyba

Chyba je projev poruchy, kdy obvod při své činnosti vrací nesprávnou hodnotu.

2.2.3.4 Selhání

Selhání je projev chyby v obodu, kdy obvod není schopen vykonávat svou činnost.

2.2.4 Poruchový model

Porucha je projevem fyzického defektu v obvodu, na logické úrovni popisu. Každý defekt se na logické úrovni projeví jiným způsobem. Používá se tedy několik různých poruchových modelů, každý z nich modeluje jinou skupinu defektů.

2.2.4.1 Trvalá hodnota

Tento model popisuje poruchu jako trvalou logickou hodnotu na signálu. Takto lze modelovat například defekt zkratu signálu se zemí nebo s napájením, defekt na vstupu či výstupu hradla.

Pro správné použití toho modelu je třeba uvažovat poruchu na každém vstupu a výstupu hradla, dále pak na vstupech a výstupech obvodu. V některých případech však lze poruchu vypustit.

2.2.4.2 Zkrat

Tato porucha modeluje vodivé spojení dvou signálů. V závislosti na technologii a povaze defektu se modeluje pomocí rezoluční funkce AND či OR.

Počet možných poruch tohoto typu v obvodu roste kvadraticky s počtem signálů. Pro netriviální obvody je tedy třeba počet uvažovaných poruch snížit. Typicky se uvažují pouze dvojice signálů, které se nalézají ve fyzické blízkosti na čipu. To je nevýhoda proti ostatním modelům, protože musíme znát fyzické rozmístění vodivých cest.

2.2.4.3 Tranzistor

Tato porucha modeluje defekty jednotlivých tranzistorů, jedná se tedy o model na nižší úrovní, než trvalá hodnota. Tranzistor s defektem modelujeme jako trvale sepnutý, či trvale rozepnutý. Pokud jde o trvale sepnutý tranzistor, může takový defekt vést ke zkratu země a napájení.

2.2.4.4 Přerušený obvod

Přerušený signál je modelován touto poruchou. Chování takového obvodu je závislé na použité technologii. Většinou lze tento defekt pokrýt poruchou typu trvalá 0/1.

2.2.4.5 Dynamické modely

Skupina dynamických poruchových modelů se zabývá poruchami časování, zpoždění, dynamickými hazardy a dalšími dynamickými vlastnostmi obvodu.

Tyto modely mají uplatnění hlavně v testování asynchronních obvodů. U synchronních obvodů tyto poruchy zpravidla nemají vliv na funkci obvodu.

2.2.5 Hledání testovacích vektorů

Intuitivní zcitlivění cesty Nejjednodušší způsob hledání testovacích vektorů je metoda intuitivního zcitlivění cesty. Jedná se o manuální generování testu, kde postupně pro všechny poruchy hledáme takové nastavení vstupních signálů, které zcitliví cestu od místa poruchy na některý výstup a zároveň nastaví hodnotu signálu v místě poruchy tak, aby se tato porucha projevila.

Například pro poruchu trvalá 1 na vstupu hradla AND chceme tento vstup nastavit na hodnotu 0 a druhý vstup hradla na hodnotu 1.

D-algoritmus D-algoritmus je založen na myšlence intuitivního zcitlivění cesty. Využívá pětihodnotové logiky, d-kalkulu, která mimo hodnoty logická 0/1 vyjadřuje i stav, kdy došlo k poruše a změně hodnoty z 0 na 1 či naopak.

Tento algortmus nejprve generuje pro hradla v obvodu přenosové krychle, které právě popisují fragment citlivé cesty. Po výběru poruchy se vytvoří přenosová krychle této poruchy. Pomocí průniku s přenosovými krychlemi hradel se porucha nejprve propaguje na výstup obvodu, poté se odvodí ohodnocení vstupů obvodu. Pokud lze v libovolném kroku použít více přenosových krychlí, zvolí se jedna náhodně. To může vést ke konfliktu, který je třeba řešit návratem a volbou jiné přenosové krychle. Pokud jinou krychli nelze zvolit, je porucha nedetekovatelná. [1]

Problém tohoto algoritmu je počet možností, kterými lze vybrat přenosové krychle při propagaci poruchy či odvození vstupních signálů. Těchto možností je $O(2^s)$, kde s je počet signálů v obvodu. Jedná se o NP-úplný problém.

PODEM Path Oriented Decision Making (PODEM) je algoritmus vytvořený jako odpověď na neefektivitu D-algoritmu. Na rozdíl od D-algoritmu, PODEM zkouší postupně nastavovat vstupy obvodu tak, aby došlo k excitaci a propagaci poruchy na výstup.

Stejně jako u D-algoritmu se jedná o prohledávání celého stavového prostoru. Jeho velikost je ale na rozdíl od předchozího algoritmu $O(2^n)$, kde n je počet vstupů obvodu. [2]

Další vylepšení PODEM Fanout-Oriented Test Generation (FAN) je modifikace PODEM algorimu, ve které se provádí kromě dopředné propagace i zpětné nastavení signálů směrem ke vstupům obvodu, pokud jsou tyto signály již jednoznačně určené. Toto vylepšení vede ke dřívějšímu ukončení návratu případě konfliktu. [3]

SOCRATES je vylepšení FAN algoritmu. Hlavní rozdíl je v předpočítání závislostí v obovodu a pamatování závislostí objevených při běhu. Díky tomuto učení může objevit konflikty dříve a zrychlit tak návrat. [4]

Booleovská diference U tohoto algoritmu řešíme derivaci funkce obvodu.

$$\frac{\partial F(x)}{\partial x_i} = F(x_1, x_2, \dots x_i = 0, \dots x_n) \oplus F(x_1, x_2, \dots x_i = 1, \dots x_n)$$

Tuto derivaci položíme rovnu 1, čímž vyřešíme propagaci. Můžeme zároveň řešit i excitaci poruchy.

$$x_i \cdot \frac{\partial F(x)}{\partial x_i} = 1$$

Nevýhodou tohoto algoritmu je exponenciální paměťová náročnost pro obvody obsahující funkci XOR. Dále se jedná o funkční test.

[5]

SAT Další algebraický algoritmus generování testu je pomocí splnitelnosti booleovských formulí (SAT).

Princip tohoto testu spočívá v porovnání výstupu bezporuchového obvodu a poruchového obvodu hradlem XOR. Obvod je pak popsán strukturně ve formě CNF. Tyto CNF formule jsou spojeny do jedné CNF funkcí AND. Tedy testujeme, že všechny tyto formule jsou splnitelné. Ohodnocení splněné formule je testem dané poruchy. [6]

Řešení SAT je NP-úplný problém, existují však kvalitní řešiče SAT.

2.3 BIST

BIST se skládá ze tří hlavních částí. První část je generátor testovacích vektorů (test pattern generator; TPG). Jak jeho název napovídá, tento obvod generuje testovací vektory, které jsou přivedeny na vstupy CUT. TPG může být realizován jako generátor pseudonáhodné sekvence vektorů, jako paměť s uloženými vektory, nebo jako kombinace těchto možností.

Druhá část je kompaktor odezvy (outpur response analyzer; ORA). Test může být a bývá značně dlouhý, je třeba odezvu nějakým způsobem zkompaktovat. Toho je dosaženo z pravidla ztrátovou kompresí odezvy. Zrátová komprese znamená, že dochází ke ztrátě informace. Nechceme však, aby ztracená informace byla o detekované poruše. Pokud taková situace nastace, jedná so o tzv. zakrytí poruchy (aliasing).

Třetí část BIST je řadič, jednoduchý automat, který řídí proces testování. Na konci testu vyhodnotí odezvu CUT, podle obsahu registru ORA.

2.4 Generátor testovacích vektorů

2.4.1 Triviální test

Triviální (exhaustive) tesd generuje všechny možné vstupní vektory. Jedná se tedy o funkční test. Jeho výhodou je, že pokryje všechny poruchy, které jsou detekovatelné, i všechny kombinace poruch. Délka takového testu je $O(2^n)$ kde n je počet vstupů obvodu. V praxi je tedy takový test nepoužitelný.

2.4.1.1 Čítač

Čítač stejné délky jako vstup CUT. Nevýhodou tohoto obvodu je, že logika zajišťující čítání může být složitá a pomalá, hlavně s ohledem na přenosy z nižších řádů.

2.4.1.2 LFSR

LFSR stejné délky jako vstup CUT. Při použití ireducibilního polynomu generuje všechny vstupní vektory, mimo nulového, generuje tedy $2^n - 1$ vektorů. Pokud chceme nulový vektor zahrnout do testu, potřebujeme další logiku, která to zajistí. Výhoda proti čítači je pak rychlejší a jednodušší logika, než přičtení jedničky.

2.4.1.3 Pseudotriviální test

Pseudotriviální (Pseudo-exhaustive) test je triviální test aplikovaný na obvod rozdělený na menší části. Výhodou tohoto přístupu je, že dlouhý test je rozdělen na několik kratších testů. [7] Další výhodou je malý nárůst plochy, proti triviálnímu testu. Pro velké obvody je však tento přístup stále nepraktický.

2.4.2 Pseudonáhodný test

Tato skupina TPG generuje pseudonáhodnou sekvenci vstupních vektorů. Výhoda proti úplnému testu je kratší doba testování, nevýhodou je pak typicky menší míra pokrytí poruch, zvláště u větších obvodů.

2.4.2.1 Čítač

Jedná se o stejný obvod jako u úplného testu, ale může být inicializován nenulovým vektorem. Jako pseoudonáhodný generátor testovacích vektorů má výraznou slabinu v tom, že vyšší bity se nemění příliš často, nebo vůbec. Nižší bity se naopak mění často.

2.4.2.2 Aritmetický generátor

Jedná se o zobecnění čítače. Místo přičítání jedné však inkrementujeme střadač o kladnou konstantu větší než jedna. Výpočet probýhá v modulární aritmetice, pro delší periodu je vhodné zvolit inkrementační kontantu nesoudělnou s modulem sčítačky:

$$R(t) = R(t-1) + C(mod 2^n)$$

Zde $R(t)$ je obsah střadače v čase t, C je aditivní konstanta generátoru, n je šířka sčítačky.

Tento generátor se používá zejména ve vysokoúrovňovém návrhu, často s využitím již existujících bloků v datové cestě návrhu. Typická je také kombinace tohoto generátoru s aritmetickým kompresorem. [8]

2.4.2.3 LFSR

Výhoda oproti čítači je, že mění všechny bity vstupu přibližně stejně často. Nevýhoda je opět v absenci nulového vektoru.

Existují dvě varianty LFSR, Fibonacciho LFSR a Galoisův LFSR. Jejich funkcionalita je identická, liší se však konstrukcí. Fibonacciho varianta je konstruována tak, že hodnota nejnižšího registru v následujícím stavu je dána XORem hodnot registrů daných polynomem tohoto LFSR. Galoisova konstrukce naopak XORuje hodnotu nejnižšího registru k registrům daným polynomem.

V použití jako TPG je vhodnější Galoisova konstrukce, protože konstrukce s vnitřními hradly XOR vede na kratší kombinační cesty mezi registry, co se počtu hradel týče.

2.4.2.4 Ring generator

Tento obvod je funkčně okvivalentní LFSR, ale jeho struktura umožňuje snadnější a kratší propojení.

2.4.2.5 Buněčný automat

Buněčný automat (cellular automaton; CA) je obvod, kde budoucí stav jedné buňky je dán pouze současným stavem okoních buňek. Oproti LFSR zabírá méně plochy, je snadněji propojitelný a celkově je rychlejší. [9]

Nevýhoda proti LFSR je jeho kratší perioda. Pokud je však náš test kratší, než tato perioda, není tato vlastnost problém.

Další vlastností CA je nevyváženost vstupů. U LFSR je pravděpodobnost výskytu 1 na konkrétní pozici $\frac{1}{2}$. U CA tomu tak není. Tato vlastnost může být výhodná i nevýhodná, v závislosti na tom, co chceme.

Podle toho, jakou funkcí je počítán následující stav buňky, rozlišujeme různé typy CA. Pro účely konstrukce TPG uvažujeme zpravidla jednorozměrné CA. Buňky jsou tedy uspořádány v jedné řadě. Každá buňka má dva sousedy, kromě dvou krajních buněk, které mají jednoho souseda každá.

Nejčastější funkce jsou „90“ a „150“. Pravidlo „90“ počítá následující stav jako xor okolních buněk. Pravidlo „150“ k tomuto výsledku navíc xoruje i stav počítané buňky. [10]

2.4.2.6 Vážené vstupy

Někdy se nám hodí, aby některé vstupy byly častěji v jednom logickém stavu, například pokud to vede ke zcitlivění cesty. Některé generátory mají nevyvážené vstupy jako svou vlastnost. U generátorů s vyváženými vstupy můžeme použít další logiku.

Pokud například dva výstupy LFSR přivedeme do hradla AND, výsledná pravděpodobnost výskytu logické 1 na výstupu hradla je $P(1) = \frac{1}{2} \times \frac{1}{2} = \frac{1}{4}$. Podobně pro hradlo OR $P(1) = 1 - (\frac{1}{2} \times \frac{1}{2}) = \frac{3}{4}$.

Nevýhodou tohoto generátoru je náročnější generování testu a v případě BIST také zvětšení obsazené plochy na čipu, zvláště při implementaci více nevyvážených distribucí. Tato nevýhoda však může být vyvážena zkrácením doby testu. [11]

2.4.3 Deterministický test

Deterministický test je takový, který obsahuje deterministické testovací vektory, neboli vektory které jsou spočítány ze struktury CUT tak, aby pokryly konkrétní poruchy. Jinými slovy, každý vektor v takovém testu byl vygenerován aby pokryl konkrétní poruchu. V tom se liší od PRNG testů, které generují náhodné vektory.

Výhodou tohoto testu je úplnější pokrytí a kratší test, neb je možné test minimalizovat tak, aby každý testovací vektor pokryl nějakou unikátní poruchu. Test tedy neobsahuje redundantní vektory.

Nevýhodou je pak nutnost uložit tyto vektory v paměti. Vektorů může být v deterministickém testu hodně a paměťové nároky mohou být nepřijatelné.

2.4.3.1 Kombinace s PRNG

Kompromisem mezi PRNG a deterministickým testem je jejich kombinace. Tedy nejprve PRNG test, který pokryje většinu poruch a poté deterministický test, který pokryje pouze poruchy nepokryté PRNG testem.

2.4.4 Reseeding

Reseeding je další varianta kombinace deterministického testu a pseudonáhodného testování. Hlavní myšlenka této metody je, že jedena posloupnost generovaná PRNG nepokryje všechny detekovatelné poruchy, ale více kratších posloupností téhož generátoru poskytne lepší pokrytí.

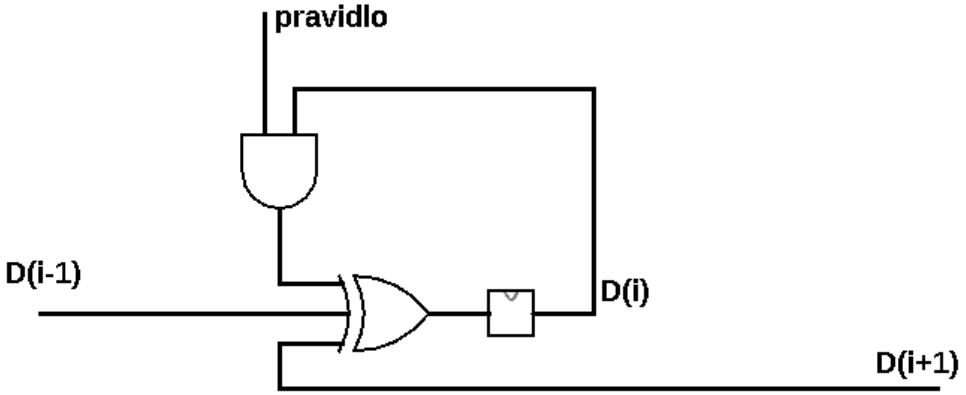
Stejně jako u deterministického testu je součástí TPG paměť, ale tentokrát neobsahuje deterministické vektory, nýbrž seedy a případně další nastavení TPG pro generování různých pseudonáhodných posloupností. Tato metoda je použitelná pro libovolný TPG, který má volitelný počáteční stav.

LFSR Reseeding pro LFSR je tvořen uloženými inicializačními hodnotamy, případně může paměť obsahovat i různé primitivní polynomy. V takovém případě LFSR obsahuje logiku, která umožňuje přenastavit jeho polynom. To vede ke zvýšení složitosti obvodu a plochy kterou zabírá. Ziskem pak je výrazné zmenšení počtu nutných testovacích vektorů pro dosažení požadovaného pokrytí.

Myšlenka za použitím LFSR s konfigurovatelným polynomem je, že jeden polynom hůře pokrývá některé skupiny poruch a lépe pokrývá jiné. Požadované polynomy jsou generovány tak, aby jejich náhodné sekvence obsahovaly co nejvíce deterministických testovacích vektorů. Jedna metoda jak je lze spočítat je řešením lineárních rovnic nad binárním tělesem. Tyto polynomy navíc nemusí být ireducibilní, neboť v tomto režimu nepotřebujeme celou periodu $2^n - 1$, ale stačí nám perioda stejné délky, jako je délka posloupnosti jednoho seedu.

Buněčný automat Stejně jako u LFSR lze reseeding použít i pro TPG tvořený buněčným automatem. Tímto se stírá nevýhoda, kterou CA mají proti LFSR, tedy kratší perioda náhodné sekvence. Ostatní výhody použití reseedingu jsou stejné jako u LFSR.

Buněčný automat nemá primitivní polynom, který by bylo možné měnit, má však pravidla, která výrazně ovlivňují generovanou posloupnost a jejich topologický prostor. Vedle počátečního nastavení tedy mohou řádky v paměti obsahovat i nastavení pravidel buněk.



Obrázek 2.1: Buňka CA, signál pravidlo přepíná mezi pravidlem 90 a 150.

2.5 Kompaktor odezvy

Kompaktor odezvy je část BIST, která střádá odezvu CUT na testovací vektory a počítá z nich otisk, signaturu obvodu. Na konci testu se pak tento otisk porovná s předpočítanou hodnotou funkčního obvodu. V případě, že se tento otisk liší, víme, že obvod je vadný.

Protože odezva obvodu je ztrátově komprimována, může dojít k zakrytí chybné odezvy. Tento jev je třeba při návrhu BIST také sledovat.

2.5.1 Aritmetický kompaktor

Aritmetický kompaktor je sčítačka se střadačem, kde se výstup CUT interpretuje jako číslo a to se přičítá ke střadači. Lze použít různé druhy sčítaček, s přenosem, s cyklickým přenosem, bez přenosu, sčítačku v doplňkovém kódu, ... Sčítání s cyklickým přenosem je pak nejvhodnější pro omezení aliasingu. [8]

Více sčítaček lze zapojit do kaskády, kdy vstupem další sčítačky je obsah střadače předchozí sčítačky. Otiskem CUT je pak obsah střadače poslední sčítačky na konci testu. Toto kaskádní zapojení výrazně snižuje pravděpodobnost maskování poruchy při příchodu dvou vzájemně opačných odezev. [10]

2.5.2 LFSR

Výstup CUT je přiveden na vstup LFSR, který funguje v režimu děličky mnohočlenů. Obsah LFSR na konci testu je otiskem obvodu.

2.5.3 MISR

MISR je LFSR s více paralelními vstupy, které jsou xorované s obsahem jednotlivých registrů.

2.5.4 Buněčné automaty

Buněčné automaty mohou být použity v podobném režimu jako MISR, tedy v každém kroku se k jejich obsahu xoruje odezva obvodu.

Některé práce naznačují, že buněčné automaty jsou lepší pro kompakci odezy, než LFSR, protože jsou odolnější proti závislým chybám. [12]

2.6 Scan

Dalším používaným typem návrhu na testování je Scan. V tomto návrhu se před vstupy a za výstupy obvodu vloží registry zapojené jako posuvný registr s paralelním čtením a zápisem. Tyto registry jsou při běžném provozu obvodu transparentní, ale při testování lze vstupy a výstupy obvodu přepnout na čtení z a zápis do registru.

Tento návrh se často používá v kombinaci s externím testováním. Využívá se v existujících standardech, např. JTAG, IEEE 1500.

2.6.1 Test per Scan

V tomto režimu se nejprve sériově nasune celý testovací vektor do scan-chainu, přičemž testovaný obvod je v nečinnosti. Po nasunutí se vstupy obvodu přepnou na čtení z registrů a výstupy na zápis do registrů. Po jednom hodinovém cyklu se vstupy a výstupy přepnou zpět do normálního stavu. V tomto okamžiku je odezva obvodu zachycena ve výstupních registrech. Může začít nasouvání dalšího testovacího vektoru, přičemž se zároveň vysouvá odezva předchozího testu ven ze scan-chainu.

2.6.2 Test per Clock

V tomto režimu je testovací vektor nasouván a odezva se zachytává při každém hodinovém taktu. Ze scan-chainu se odezva vysouvá zároveň s tím, jak se do něj nasouvá testovací vektor.

Výhodou návrhu scan je jeho jednoduchost a z toho plynoucí menší nároky na plochu a propojení. Počet externích sond je také snížen, ale za cenu prodloužení testu. Velkou nevýhodou je potřeba externího testu.

2.6.3 Respin

Respin je metoda generování bitstreamu pro použití v paralelních scan řetězcích. Tyto řetězce jsou součástí standardu IEEE 1500 a v tomto případě fungují jako dekopresor vnějšího testu. [13]

Generátor testu generuje takový bitstream, aby na výstupech použitého posuvného registru postupně vznikaly dané deterministické vektory. Registr postupně rotuje svůj obsah a zároveň jsou některé jeho bity nahrazovány bitstreamem.

Nevýhoda tohoto algoritmu je, že pořadí ve které jsou testovací vektory generovány silně ovlivňuje kvalitu, tedy velikost, bitsreamu.

2.6.4 SAT kompress

Tento algoritmus je vylepšením metody Respin. Rozdíl je v tom, že nepoužívá již vygenerované testovací vektory, ale generuje vlastní, tak aby efektivně pokryl poruchy v obvodu, se znalostí obsahu posuvného registru. Díky tomu může vygenerovat vektor pokrývající danou poruchu dříve, než Respin. Využívá metody řešení splnitelnosti booleovských formulí. [14]

Kapitola 3

Návrh a implementace

3.1 Návrh

3.1.1 BIST

Implementoval jsem vnitřní test, bez kombinace s vnějším testováním. Vstupem obvodu je příkaz spustit testování. Výstupem obvodu je signál indikující konec testu a signál indikující výsledek testu.

Uvažoval jsem pseudonáhodné i deterministické testy. Pseudonáhodné testy jsou pro netriviální obvody příliš dlouhé, nebo nemají dostatečné pokrytí poruch. Deterministické testy na druhou stranu zabírají příliš mnoho paměti a tedy plochy na čipu. Mezi těmito extrémy existuje však existuje kompromis.

3.1.1.1 Generátor testovacích vektorů

První generátor pseudonáhodných testovacích vektorů, který jsem zvažoval, je lineární zpětnovazební posuvný registr. Tento generátor je často používaný a jeho vlastnosti jsou dobře známy.

Druhý generátor, na který jsem se zaměřil, je jednorozměrný buněčný automat. Tyto generátory jsou také známé a používané. Porovnával jsem tedy tyto dva generátory vůči sobě navzájem.

Výhodou LFSR je jeho délka periody, jeho nevýhodou je silná závislost jednotlivých výstupů. Buněčný automat má proti tomu vzájemnou závislost výstupů více skrytou, má však kratší periodu. Buněčný automat je také jednodušší konstrukčně.

Reseeding Mé první testy s LFSR odhalily nedostatečné pokrytí poruch tímto TPG. Proto jsem se rozhodl využít techniky reseeding. Později jsem změnil typ TPG na buněčný automat, využil jsem však již hotového návrhu pro reseeding, která se velice hodí pro použití s CA.

3.1.1.2 Kompaktor odezvy

Jako kompaktor odezvy jsem uvažoval LFSR s více vstupy, tedy MISR. Po teoretické přípravě jsem ale zjistil, že tento kompresor má problémy se závislými chybami, protože se jeho vnitřní stav posouvá a xoruje s následujícím výstupem testovaného obvodu.

Jako alternativu jsem uvažoval buněčný automat, který tímto neduhem netrpí, protože stav buňky závisí na předchozím stavu tří okolních buněk. Buněčný automat má navíc výhodu proti LFSR v tom, že je jednodušší a zabírá méně místa.

K zamyšlení však je, zda kratší perioda CA nevede k pravděpodobnějšímu aliasingu. Možných konstrukcí CA je celá řada a pro ty nejznámější varianty již bylo experimentálně vyhodnoceno, jaké mají vlastnosti, co se aliasingu týče. [12]

Hortensius popsal 4 metody, jak lze CA využít jako kompaktor. Pro každou z těchto metod uvažoval různé typy CA. Já jsem zvolil první variantu, tedy xorování výstupu testovaného obvodu k budoucímu obsahu buňky, podle vztahu:

$$R(t+1) = R(t)' \oplus O(t+1)$$

Tuto metodu lze použít pro buňku s libovolným pravidlem.

Ze zkoumaných variant nejlépe obstály varianty CA s pravidlem 45 a hybridní CA s pravidly 90/150. [12] Dále jsem tedy uvažoval tyto varianty.

Stav buňky s pravidlem 90 je dán xorem stavu jeho okolních buněk v předchozím kroku:

$$a_i(t+1) = a_{i+1}(t) \oplus a_{i-1}(t)$$

Podobné je pravidlo 150, kde následující stav je xorem okolních buněk a sebe sama:

$$a_i(t+1) = a_{i+1}(t) \oplus a_i(t) \oplus a_{i-1}(t)$$

Pravidlo 45 je složitější, má následující předpis:

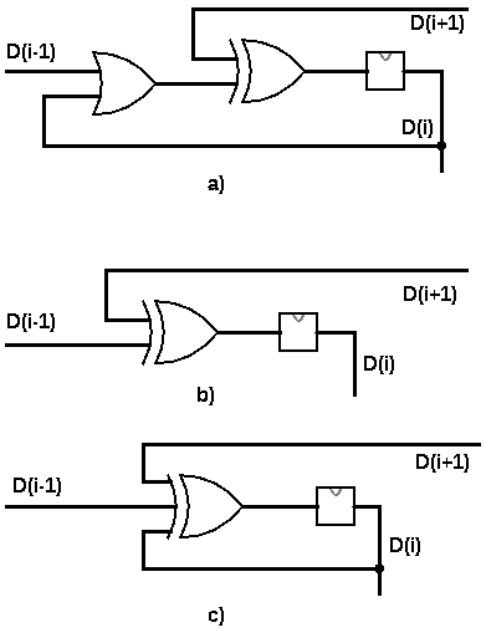
$$a_i(t+1) = [\overline{ai-1(t)} + ai] \oplus ai + 1(t)$$

Zajímavostí je, že CA s pravidlem 45 nezůstává v nulovém stavu.

Výsledky Hortensiových experimentů naznačují, že hybridní CA s pravidly 90/150 je odolnější proti aliasingu více po sobě jdoucích chyb, než CA s pravidlem 45, které je přibližně stejně odolné jako LFSR.

S tímto na vědomí a také protože CA s pravidlem 45 bylo použito v cyklické konfiguraci, zvolil jsem pro implementaci hybridní CA 90/150. V tomto CA se buňky s pravidly 90 a 150 pravidelně střídají, to znamená, že každá lichá buňka je například typu 90 a každá sudá pak typu 150. Koncové buňky mají jen jednoho souseda, jejich chybějící soused je brán, jako by měl hodnotu 0.

Pro zjednodušení návrhu jsem zafixoval pravidla koncových buněk jako 150, i v případě, že by měly mít pravidlo 90. To se však týká jen jedné koncové buňky, a to v případě, že CA má sudou délku.



Obrázek 3.1: Zapojení buněk CA pro a) pravidlo 45; b) pravidlo 90; c) pravidlo 150

3.2 Implementační platforma

Pří výběru implementační platformy, tedy programovacího jazyka, běhového prostředí a podobně, jsem hledal následující cíle.

Snadnost návrhu. Hledal jsem platformu, která mi umožní věnovat se návrhu na vyšší úrovni, která přede mnou skryje detaily nesouvisející s cílem samotným, tedy s generováním BIST logiky. Jinými slovy hledal jsem programovací jazyk, který je na vyšší abstrakční úrovni, umožňuje snadno rozdělit program na samostatné části a podobně.

Rychlost cílového programu. Tento požadavek na běhové prostředí je přímočarý. Můj požadavek není na nejrychlejší existující běhové prostředí, ale spíše aby rychlost byla v rozmezí do jednoho řádu proti jakémusi ideálu, či běžnému vzoru. V tomto případě se nabízí srovnání rychlosti s jazykem C přeloženým překladačem gcc či llvm. Tento požadavek je v rozporu s předchozím, lze pozorovat obecný trend, kdy jazyky s vyšší mírou abstrakce jsou pomalejší, než jazyky na nižší úrovni.

Podpůrné vývojové prostředky, tedy nástroje, které nejsou přímo součástí jazyka či běhového prostředí, ale pomáhají s vývojem SW na této platformě.

Podpora HW a OS. Pokud půjde výsledný SW sestavit a spustit na všech hlavních OS na nejpoužívanějším HW, bude to samozřejmě výhoda.

V neposlední řadě snadnost osvojení dané platformy. V tomto případě jsem uvažoval pouze platformy, které již do jisté míry znám. Pokud bych daný jazyk

a běhové prostředí neznal, bojoval bych na dvou frontách, vyvíjel bych nový SW na platformě, jejíž chování neznám.

C, C++ Jako první kandidát se nabízí jazyky C a C++ a celá řada jejich překladačů. C nabízí pravděpodobně nejvyšší běhovou rychlost, i když to je hodně ovlivněno zvoleným překladačem a běhovým prostředím. Podpora vývoje pro tyto jazyky je také na dobré úrovni, což je dáno jejich rozšířením a věkem. Mezi jejich neduhy pak patří roztříštěnost, každý překladač si C upravuje trochu podle sebe, na každém OS jsou trochu jiné knihovny, zcela jiné rozhraní jádra OS. Varovným signálem může být už samotný fakt, že uvažuji rozhlaní jádra, C je opravdu na nízké úrovni, blízko HW. Blízkost k HW mi není cizí, od tohoto projektu ale očekávám něco jiného. C++ některé věci vylepšuje, ale ne o mnoho, obsahuje stále mnoho detailů, o které se uživatel musí starat sám. Tento mikromanagement pak odvádí pozornost od hlavního vývoje. C podporuje asi nejvíc různých HW a OS, ale pokud bych chtěl psát SW, který poběží na více z nich, musel bych na to myslet, což by zkomplikovalo některé detaily.

Interpretované jazyky Další kandidát je jazyk Perl, který je na mnohem vyšší úrovni, ale jeho běhová rychlost je hluboko pod rychlostí C. Jeho výhodou však je rychlost, se kterou v něm lze prototypovat SW. Z mé vlastní zkušenosti s tímto jazykem však vím, že absence typové kontroly je v něm velký problém. Podpora vývoje v něm také pokulhává, ať se již jedná o možnosti statické analýzy, či ladění. Na druhou stranu podporuje zažitý systém pro testování kódu, což je značné plus. Celkově perl vidím jako výborný nástroj pro zpracování textu, ale pro obecné použití jsou lepší alternativy.

Jazyk Python je zajímavá alternativa k perlu. Oproti perlu má velkou výhodu v rychlosti běhového prostředí, v úrovni abstrakce je na tom podobně. Stejně jako perl však postrádá typovou kontrolu, což je velká nevýhoda. Tento jazyk znám z uvažovaných možností nejméně, to mu také nepřidává.

Java Java je hojně používaný jazyk a platforma, na desktopových i mobilních platformách. Jedná se o interpretovaný jazyk, který běží ve formě byte kódu ve virtuálním stroji. Přesto je jeho běhový výkon značný, porovnatelný s výkonem C. Nejrozšířenější implementace běhového prostředí dále obsahují just-in-time překlad (JIT), což dále zrychluje jeho běh. Jako bytekód pro virtuální stroj je přeložený SW nezávislý na HW a OS, může běžet beze změny všude, kde je přítomný jeho virtuální stroj. Existuje mnoho podpůrných prostředků pro vývoj v Javě, od build systémů, přes nástroje pro ladění, po rozšířené knihovny pro testování kódu.

Go Go je moderní, mladý jazyk. Jeho vznik je možné vysledovat do roku 2007, kdy byl počat ve firmě Google, Robem Pikem, Kenem Thompsonem a

Robertem Griesemerem. Jazyk byl ohlášen veřejnosti v roce 2009 a ihned se dočkal velkého ohlasu. Od roku 2012, kdy byla vydána verze 1, je tento jazyk stabilní a všechny jeho změny jsou zpětně kompatibilní, což umožňuje jeho ještě širší nasazení. Stejně jako C a C++ je Go překládán do strojového kódu, jeho běhové prostředí je však mnohem bohatší. Přímo v jazyce je zabudována podpora souběžnosti (concurrency), implementována jako velmi lehká aplikační vlákna. Běhové prostředí se pak stará o plánování vláken v rámci vláken OS. Jazyk také obsahuje prostředky pro bezpečnou komunikaci mezi těmito vlákny (gorutinami). I bez uvažování souběžnosti Go nabízí velmi příjemné a silné prostředky pro abstrakci. Jeho podpora pro různé HW a OS zaostává za Javou, ale je rozšiřována každým okamžikem. Velmi velká výhoda proti C je velmi snadný překlad pro jinou platformu, na téměř stejné úrovni jako Java.

Pro implementaci SW jsem zvolil jazyk Go.

3.3 Reprezentace obvodu

Vnitřně obvod reprezentuji jako síť hradel, každé hradlo je reprezentováno pomocí PLA. Jedná se o ekvivalentní reprezentaci k formátu BLIF, který jsem zvolil jako vstupní formát, pro jeho jednoduchost a univerzálnost.

Z tohoto vzešlo několik omezení reprezentovatelného obvodu. Každé hradlo má právě jeden výstup. Pokud chceme reprezentovat více výstupové hradlo, musíme toto hradlo rozdělit na několik jednovýstupových hradel, nebo ho reprezentovat jako vnořený obvod.

Další omezení je zákaz cyklů v obvodu. Jediný případ, kdy je cyklus povolen, je v případě, že vede přes registr ovládaný hodinovým signálem. Toto omezení jsem nevolil samovolně, jedná se o omezení ve specifikaci formátu BLIF, podle kterého jsem tuto reprezentaci modeloval. To vede k situaci, kdy obvody, které mají cykly, ale přesto se jedná o kombinační obvod, například odčítačka inverzního kódu, není reprezentovatelná tímto modelem. Za zmínku však stojí, že takový obvod nelze popsat ani (validním) BLIF souborem.

Samotný model obvodu však toto omezení nevynucuje, pokud však obsahuje sekvenční obvod, simulace nebude fungovat.

Tento model je implementován v balíčku `bistgen/netlist`, typem `model`, který je exportován přes rozhraní `Network`.

Základní operace, které toto rozhraní umožňuje jsou:

- extrakce poruch podle modelu stuck-at,
- simulace obvodu bez poruch,
- simulace obvodu s injektovanými poruchami,
- export obvodu ve formátu BLIF
- export obvodu ve formátu VHDL

Export ve formátu BLIF je přímočarý, protože samotná reprezentace obvodu je modelována podle formátu BLIF. Jedná se tedy o jakýsi přirozený export.

Export ve formátu VHDL je komplikovanější a je implementován v jiném balíčku.

V tomto balíčku je dále rozhraní `NetworkBuilder`, implementované stejným typem `model`. Toto rozhraní odráží strukturu formátu BLIF, který je plochý, obsahuje seznam příkazů, které jsou předávány modelu v pořadí, ve kterém jsou uvedeny v BLIF.

Po vytvoření modelu obvodu a jeho následném exportu je zachováno pořadí relativní vstupů a výstupů, ale pořadí vnitřních struktur se může měnit.

3.3.1 Simulace obvodu

Simulace obvodu je implementována ve stejném balíčku a exportována přes rozhraní `Network`.

Při implementaci jsem uvažoval pouze simulaci kombinačních obvodů, což mi umožnilo využít některých optimalizací.

Paralelní simulace Nejjednodušší optimalizace je využití paralelní simulace, kde místo jedné booleovské hodnoty signálu, je tato hodnota reprezentována polem bitů, v mém případě 64b číslem bez znaménka. Operaci logický součet a logický součin odpovídají operace bitový or a bitový and. Společně s negací, operace bitová negace, tyto operace zcela postačují k simulaci obvodu popsaného v BLIF.

Simulace hradla Základní jednotkou simulace je simulace jednoho hradla. Hradlo je reprezentováno rozhraním `Gate`, se dvěma implementacemi, jedna pro hradlo v konjuktivní normální formě (CNF), druhá pro hradlo v disjunktivní normální formě, tedy ve formách, ve kterých jsou popsány ve formátu BLIF.

Tato simulace sestává z jednoduché iterace přes klauzule a aplikování operací bitový and/or. Nevýhodou tohoto řešení je nemožnost časného ukončení simulace, z důvodu použití paralelní simulace. Pro jedno konkrétní hradlo tedy simulace trvá stejný počet kroků.

Celková doba jedné simulace hradla je $O(2^n)$, kde n je počet vstupů hradla. Protože je však cílem tohoto SW generovat strukturní test, vstupní obvod by měl odpovídat výsledné struktuře v cílové technologii, velikost hradel je tedy omezena a pro další výpočty budu uvažovat simulační dobu jednoho hradla jako $O(1)$.

Původní simulace obvodu Typický postup pro simulaci obecných obvodů je iterování simulačních kroků dokud se mění ohodnocení signálů v obvodu, typickým příkladem je VHDL, kde je k tomuto využit tzv. delta čas.

Protože předpokládám kombinační obvod, mohu vypočítat horní mez počtu simulačních kroků obvodu, po kterém musí dojít k ustálení. Tato mez je rovna počtu hradel v obvodu, celková horní mez simulace je tedy k^2 simulací hradla, kde k je počet hradel v obvodu.

Tato implementace se záhy ukázala jako pomalá již pro malé obvody. Uvažoval jsem o dvou variantách, jak tuto simulaci zrychlit. První variantu jsem již zmínil výše, tedy zastavit simulaci po ustálení obvodu. Druhá varianta je využít znalosti datových závislostí signálů a simulovat je jedním průchodem obvodu.

Topologické uspořádání hradel Zvolil jsem druhou variantu optimalizace, neboť tato mi umožní simulovat obvod v k krocích, což je spodní mez první varianty. Druhá varianta je zároveň necitlivá na ohodnocení vstupů simulovaného obvodu.

Zřejmou nevýhodou tohoto řešení je nutnost existence topologického uspořádání, které ale ve validním BLIF souboru musí existovat. Proto jsem také výše zmínil nutnost tuto vlastnost dodržet. Pokud dodržena není a topologické uspořádání neexistuje, je možné obvod simulovat původním, pomalejším algoritmem. Pro tyto případy jsem optimalizaci již neprováděl, neboť takový vstup stejně není validním BLIF.

Optimalizaci je nutné provést explicitně. Použitý původní, či optimalizovaný algoritmus je automaticky zvolen na začátku simulace, podle toho, zda je v modelu obvodu uložena informace o topologickém uspořádání.

V jednom a jediném simulačním kroku jsou pak hradla simulována v pořadí jejich topologického uspořádání. Délka simulace je tedy $O(k)$ simulací hradel. Nalezení topologického uspořádání zabere $O(k)$ kroků.

Simulace poruch Hlavním cílem simulace obvodu však není bezporuchová simulace, ale simulace s injektovanými poruchami.

Poruchy stuck-at jsem implementoval předáním seznamu poruch do simulace, před a po každé simulaci jednoho hradla provedu aplikování poruchy na svtupní a výstupní signály tohoto hradla.

Protože provádím paralelní simulaci v 64b vektoru, má každá předaná porucha kromě svého umístění v obvodu i chybový vektor s nastavenými bity pro ty simulace, ve kterých se vyskytuje.

Toto řešení umožňuje simulovat více poruch paralelně a jako vedlejší efekt také umožňuje simulovat více současných poruch v jedné simulaci.

Použití, které jsem zamýšlel je paralelní simulace jednoho bezporuchového obvodu a 63 obvodů s různými poruchami.

Lze samozřejmě simulovat i jeden obvod s 64 různými vstupními vektory.

3.4 Parsování BLIF

V balíčku `bistgen/blif` je implementován parser vstupního souboru. Je rozdělen do dvou částí, lexikální analyzátor a parser.

Lexikální analyzátor je klasicky implementován jako stavový automat, kde stav je reprezentován stavovou funkcí. Tato konstrukce je inspirována parserem použitém ve standardní knihovně Go. [15]

Duhou část, tedy parser jsem implementoval stejně, tedy jako stavový automat. Takto jednoduchý parser lze použít, protože formát BLIF je plochý. V jednom modelu jsou BLIF příkazy umístěny lineárně a gramatika neobsahuje žádné konflikty.

Omezení parseru je, že umí načíst pouze jeden model obvodu, i když BLIF podporuje definování více modelů v jednom souboru. Příkaz pro prohledání dalšího souboru, který je v BLIF také povolen, není podporován. Z tohoto důvodu také není podporován příkaz `.subckt`, který instancuje vnořený obvod.

Parser tedy podporuje pouze podmnožinu BLIF specifikace, konkrétně pouze ploché kombinační obvody.

3.5 Export VHDL

V balíčku `bistgen/hdl` je implementován export obvodu ve formátu VHDL.

Problém exportu do VHDL je jeho odlišnost od BLIF, a tedy i odlišnost od vnitřní reprezentace obvodu.

Reprezentace ve formátu vhodném pro generování VHDL je ve struktuře `HDL`. Model obvodu sám sebe převede do tohoto formátu a rekurzivně pak také všechny vnořené obvody.

Struktura `HDL` je poté předána jako parametr pro hierarchický template, ve kterém je VHDL soubor napsán. Tato struktura poté rekurzivně vygeneruje VHDL kód pro všechny své vnořené obvody.

Výstupem je jeden VHDL soubor, který obsahuje všechny entity a jejich architektury.

Tento balíček jsem implementoval s ohledem o snadnou rozšiřitelnost pro jiné HDL, například Verilog, nebo i BLIF samotný.

3.6 Kompaktor odezvy

Kompaktor odezvy jsem implementoval v balíčku `bistgen/ora/cmisr` jako typ `CMISR`. Jeho rozhraní je exportováno jako `MISR` z balíčku `bistgen/ora`. I přes toto pojmenování obsáhne rozhraní `MISR` jakýkoli typ kompaktoru s paralelními vstupy.

Implementace samotná, `CMISR`, je založena na buněčném automatu s hybridními pravidly 90 a 150, jak je popsáno v článku [12].

Obvod samotný pro účely simulace reprezentuji pouze funkčně, nikoli jako síť hradel. Tím dosahuji rychlejší simulace. Stejně jako v případě netlistu jsem implementoval paralelní simulaci, což umožňuje snadné a efektivní použití zároveň se simulovaným obvodem.

Pro účely exportu jsem pak v tomto balíčku implementoval i export buněčného automatu do vnitřní reprezentace v podobě netlistu. Tuto reprezentaci pak lze dále exportovat do formátu BLIF či VHDL.

3.7 Generátor testovacích vektorů

Generátor testovacích vektorů jsem implementoval v balíčku `bistgen/tpg/ca` jako typ `ca90150`. Jeho rozhraní pak jako `TPG` z balíčku `bistgen/tpg`.

Jak již název napovídá, jedná se o buněčný automat stejného typu, který jsem použil pro kompaktor odezvy. Pro účely simulace jsem využil již existujícího kódu z kompaktoru odezvy, jen jsem upravil rozhraní tak, aby odpovídalo typu `TPG`.

Tento obvod také podporuje paralelní simulaci, které ale nevyužívám, neboť paralelismu jsem využil k simulaci poruch. Tato optimalizace tedy ztracena není, jen je využita v jiné části programu, v rámci stejné simulace.

Vnitřní zapojení obvodu se liší od kompaktoru odezvy, místo xorování výstupu se vstupem je vnitřní stav tímto vstupem zcela nahrazen, když dochází k načtení nového inicializačního vektoru. Z tohoto důvodu nevyužívám netlistu generovaného v kompaktoru, ale generuji ho zvlášť.

Zajímavostí je, že kód tohoto automatu v kompaktoru je starší, protože původně jsem TPG implementoval jako LFSR. V porovnání s LFSR je vnitřní struktura CA mnohem jednodušší, což vede ke snadnějšímu generování struktury tohoto obvodu.

Implementaci LFSR jsem v balíčku `bistgen/tpg/lfsr` nechal, ale nevyužívám jí.

3.8 Paměť

V balíčku `bistgen/tpg/mem` implementuji paměť pro čtení (read-only memory; ROM). Tento obvod využívám k uložení inicializačních vektorů pro reseeding.

Paměť je organizovaná na řádky, které obsahují celý inicializační vektor. Během jednoho běhu TPG je tento řádek trvale vystaven na výstup. To je z důvodu, pokud bych chtěl v TPG nastavit nějaké volitelné parametry, například polynom LFSR nebo pravidla CA. Této možnosti však v současnosti nevyužívám.

3.9 Generátor testu

Poslední částí skupiny balíčků souvisejících s TPG je `bistgen/pfind`. Tento balíček implementuje hledání inicializačních vektorů pro daný TPG.

Pracuje s jakýmkoli TPG, který splňuje jednotné rozhraní. Nevýhoda tohoto řešení je, že nelze využít případné znalosti vlastností TPG. Je na něj pohlíženo jako na černou skříňku.

Tento balíček generuje pseudonáhodný test. Neprve vygeneruje sadu inicializačních vektorů, které na zadaném obvodu odsimuluje. Po odsimulování těchto vektorů vyhodnotí, které poruchy jsou jimi pokryty.

Poté vybere zadaný počet vektorů, omezený velikostí paměti, tak aby tyto vektory pokryly co nejvíce poruch. K tomu využívá hladového algoritmu, tedy vždy uvažuje inicializační vektor, který pokrývá co nejvíce poruch, které nebyly pokryty dříve vybraným vektorem.

Další funkce v tomto balíčku je pokrytí preferovaných poruch. Uživatel může programu předložit seznam poruch, které chce pokrýt přednostně. Algoritmus pak vybere ty inicializační vektory, které pokryjí co nejvíce preferovaných poruch, i za cenu nižšího celkového pokrytí.

Nastavitelné parametry tohoto algoritmu jsou kromě preferovaných poruch, maximální velikost paměti pro BIST, délka testu.

Maximální velikost paměti je horní limit, algoritmus může využít méně paměti, pokud žádný další nalezený vektor nepokrývá novou poruchu.

Délka testu se vztahuje k délce testování jedním inicializačním vektorem. To znamená, že všechny inicializační vektory jsou využity ke generování testu stejné délky. Výhodou je jednodušší logika řadiče a menší paměťové nároky, protože není nutné ukládat délku testu. Nevýhodou je naopak menší efektivita testování, čili prodloužení testu nebo nižší pokrytí.

3.10 Řadič

Řadič jsem implementoval v balíčku `bistgen/ctrl`. Pro jeho složitost jsem ho rozdělil do několika menších obvodů, všechny jsou umístěny v tomto balíčku.

Controller Hlavním obvodem tohoto balíčku je `Controller`, který kupodivu nemusí znát nic o ostatních částech BIST, kromě výsledné signatury a velikosti adresy paměti.

Tento obvod obstarává komunikaci s okolním světem. Signál `start` spouští testování, signály `done` a `ok` pak podává zprávu o výsledku testu.

Stavový automat Samostatnou částí Kontroleru je stavový automat, který má kupodivu jen 4 stavy a je zcela nezávislý na zbytku obvodu. Jako jediný obvod v mé implementaci není reprezentován vlastním typem a instancí, ale

existuje pouze v podobě netlistu, ve formátu BLIF, který je na přání parsován do vnitřní formy, když je do této reprezentace převáděn Kontroler.

Čítač Čítač je klasický obvod tvořený registrem a půlsčítačkou, který ke svému obsahu přičítá jedničku. Využívám ho na dvou místech, pro časování přepnutí na další inicializační vektor a pro adresování paměti. Jsou zapojeny tak, že fukčně tvoří jeden čítač, ale výstup z nižší části využívám i v jiných částech řadiče, konkrétně pro ovládání TPG, proto jsem je implemetoval odděleně.

Top Obvod `TOP` je nejvyšší entitou v mém návrhu, obsahuje všechy ostatní komponenty BIST a testovaný obvod. Zajišťuje jeho propojení s logikou BIST.

3.11 BISTGEN

Jednotlivé komponenty tohoto SW lze samostatně použít v podobě knihovny. Implementoval jsem však rozhraní v podobě programu ovládaného z příkazové řádky operačního systému.

Tento program je umístěn v balíčku `bistgen/cmd/bistgen`. Spustitelný soubor pak má název `bistgen` (případně `bistgen.exe` na alternativním OS).

`bistgen` je rozdělen na několik příkazů, který je každý implementován ve vlastní funkci.

Funkce, které jsem implementoval jsou:

- echo
- faults
- bist

Echo pouze vypíše načtený obvod zpět do souboru. Využití může být testování parseru formátu BLIF, či překlad obvodu z BLIF do VHDL.

Faults z načteného obvodu extrahuje všechny poruchy tupu stuck-at. Preferované poruchy pak program očekává ve stejném formátu.

Bist je nejzajímavější příkaz programu. Tento příkaz pro načtený soubor spočítá inicializační vektory a uloží logiku BIST ve formátu BLIF nebo VHDL. Dále vypíše základní informace o proběhlém procesu, například seznam nepokrytých poruch.

Kapitola 4

Zhodnocení

4.1 Splnění cílů

V této práci jsem splnil následující cíle.

Implementoval jsem SW nástroj pro generování strukturního testu pro obecný kombinační obvod. Tento obvod je zadán jako netlist ve formátu BLIF. Omezení mé implementace je, že vstupní obvod musí být plochý, nesmí obsahovat vnořené obvody.

Implementoval jsem syntézu tohoto testu v podobě logického obvodu, který v sobě obsahuje testovaný obvod a jeho rozhraní doplňuje o možnost spustit testování.

Tento testovací obvod exportuji jako netlist ve formátu BLIF, nebo jako netlist ve formátu VHDL.

Program zvládá základní simulaci poruch a vytváří zprávu o vygenerovaném testu a jeho pokrytí poruch v obvodu. Chybí však analýza aliasingu chyb v kompaktoru odezvy.

Rozšiřitelnost programu se různí podle konkrétních částí. Doplnění dalších poruchových modelů je ztížené, protože model, ketrý jsem použil jsem těsně zakomponoval do vnitřní reprezentace logického obvodu. Nebylo by příliš obtížné tuto simulaci od struktury obvodu oddělit, ale ostatní části programu, zejména generátor testu, používají tuto simulaci poruch, bylo by třeba je upravit pro simulaci více druhů poruch. Očekávám však možnost využití většiny kódu, který nesouvisí přímo s poruchovým modelem.

Rozšiřitelnost v podobě použití jiných obvodů tvořících části BIST je dobrá, návrh a implementace počítají a jsou připraveny pro nahrazení jednoho bloku jiným, například nahrazení kompaktoru odezvy, či generátoru testovacích vektorů.

velikost paměti	2	4	8	16	32	64	128	256	512
c499 (998)	8	8	8	8	8	8	8	8	8
c880 (1760)	24	42	25	46	34	22	40	45	28
c1355 (2710)	60	22	8	8	8	26	12	12	14
c2670 (5340)	892	880	882	890	882	882	880	880	882

Tabulka 4.1: Počet nepokrytých poruch v závislosti na velikosti testu. Velikost paměti je v počtu uložených vektorů.

4.2 Efektivita pokrytí poruch

Pokrytí poruch, délka testu a velikost testovací logiky jsou tři různé ukazatele, které spolu ale úzce souvisí, v této části tedy popíši jejich vzájemné působení jedoho na druhého.

4.2.1 Velikost testu

Zkoumal jsem vliv velikosti testu, tedy počet uložených inicializačních vektorů na pokrytí poruch. Pro tento test jsem zafixoval celkovou délku testu, tedy celkový počet testovacích vektorů.

To znamená, že jsem měnil délku pseudonáhodné sekvence a počet těchto sekvencí. Se zvyšováním paměti a zkracováním sekvence se test blíží víc k deterministickému testu. Se snižováním využité paměti a prodlužováním náhodných sekvencí se pak test více přibližuje klasickému pseudonáhodnému testu bez reseedingu.

Výsledky simulace vybraných obvodů jsou v tabulce 4.1.

Z naměřených dat je vidět, že každý obvod se z pohledu pokrytí chová jinak, lze však vypozorovat, že nejméně nepokrytých poruch se vyskytuje v oblasti kde je množství paměti a délka testu vyrovnaná. Snižování velikosti paměti pak má výraznější vliv na pokrytí, než snižování délky náhodné sekvence.

Obvod c880 je pak zvláště zajímavý, je vidět, že jeho pokrytí je hodně citlivé na výběr inicializačních vektorů. Tento výsledek může ukazovat na nevhodnost zvolené heuristiky pro tento konkrétní obvod.

4.2.2 Délka testu

Pro zkoumání závislosti pokrytí na délce testu jsem ponechal velikost paměti testu konstantní. Měnil jsem tedy délku generované pseudonáhodné sekvence, přičemž počet inicializačních vektorů jsem ponechal konstantní 4.2.

Pokud zvětšujeme pouze délku testu, tedy počet testovacích vektorů, pokrytí obvodu roste, což je výsledek, který nepřekvapí. Po zvětšení testu na určitou délku se však růst zastaví a více poruch již další prodlužování testu nepokryje.

délka testu	4	16	64	256	1024
c499	281	149	45	8	8
c880	438	161	58	42	6
c1355	728	386	209	22	

Tabulka 4.2: Počet nepokrytých poruch v závislosti na délce testu.

Závěr

V této práci jsem navrh a implementoval softwarový prostředek pro generování strukturního testu pro kombinační obvody. Tento program načte zdrojový soubor cílového obvodu, spočítá pro něj pseudonáhodný test s využitím techniky reseeding a vygeneruje logický obvod, který tento test implementuje.

Vstupní formát testovaného obvodu je síť hradel, popsaná formátem BLIF. Výstupní formát je volitelný mezi BLIF a VHDL. Další výstup programu je stručná zpráva o pokrytí poruch v obvodu.

Implementoval jsem generátor testovacích vektorů i kompaktor odezvy jako buněčné automaty, ale nechal jsem možnost tyto obvody transparentně nahradit jinými, dokud mají stejné rozhraní.

V tomto ohledu je tedy program snadno rozšiřitelný. Obtížnější bude rozšíření o jiné poruchové modely, neboť model trvalé nuly a jedničky, který jsem použil, je pevně zabudovaný do simulace obvodu.

Jako další pokračování práce vidím tři směry. Jeden směr je rozšíření o další typy generátorů testovacích vektorů a kompaktorů odezvy. Druhý směr je rozšíření o další poruchové modely, což by vyžadovalo oddělení simulace od reprezentace obvodu. Třetí směr je implementace některého z algoritmů pro generování deterministických vektorů, které pokryjí vybrané poruchy.

Literatura

[1] Roth, J. P.: Diagnosis of Automata Failures: A Calculus and a Method. *IBM Journal*, July 1966: s. 278–287.

[2] Goel, P.: An Implicit Enumeration Algorithm to Generate Tests for Combinatinal Logic Circuits. *IEEE Transactions on Computers*, ročník 30, č. 3, March 1981: s. 215–222.

[3] Fujiwara, H.; Shimono, T.: On the Acceleration of Test Generation Algorithms. *IEEE Transactions on Computers*, ročník 32, č. 13, December 1983: s. 1137–1144.

[4] Schulz, M. H.; Trischler, E.; Sarfert, T. M.: SOCRATES: A Highly Efficient Automatic Test Pattern Generation System. *IEEE Transactions on Computer-Aided Design*, ročník 7, č. 1, January 1988: s. 126–137.

[5] Sellers, F. F.; Hsiao, M. Y.; Bearson, L. W.: Analyzing Errors with the Boolean Difference. *IEEE Transactions on Computers*, ročník 17, č. 7, July 1968: s. 676–683.

[6] Larrabee, T.: Test Pattern Generation Using Boolean Satisfiability. *IEEE Transactions on Computer-Aided Design*, ročník 11, č. 1, January 1992: s. 4–15.

[7] Chen, C.-I. H.: BISTSYN - A Built-In Self-Test Synthesizer. IEEE, 1991.

[8] Mukherjee, N.; Kassab, M.; Rajski, J.; aj.: Arithmetic Built-In Self-Test for High-Level Synthesis. IEEE, 1995.

[9] Boubezari, S.; Kaminska, B.: Mixed deterministic and pseudorandom test vector generator based on cellular automata structures. In *Circuits and Systems, 1995. ISCAS '95., 1995 IEEE International Symposium on*, ročník 3, Apr 1995, s. 1928–1931 vol.3, doi:10.1109/ISCAS.1995.523796.

[10] Stroud, C.: *A Designer's Guide to Built-in Self-Test*. Frontiers in Electronic Testing, Springer, 2002, ISBN 9781402070501.

[11] Muradali, F.; Agarwal, V. K.; Nadeau-Dostie, B.: A New Procedure for Weighted Random Bilt-In Self-Test. In *International Test Conference*, IEEE, 1990.

[12] Hortensius, P. D.; McLeod, R. D.; Card, H. C.: Cellular Automata-Based Signature Analysis for Built-In Self-Test. *IEEE Transactions on Computers*, ročník 39, č. 10, October 1990: s. 1273–1283.

[13] Schäfer, L.; Dorsch, R.; joachim Wunderlich, H.: RESPIN++ - Deterministic Embedded Test. In *Proc. European Test Workshop*, 2002, s. 37–44.

[14] Balcárek, J.; Fišer, P.; Schmidt, J.: Test Patterns Compression Technique Based on a Dedicated SAT-based ATPG. In *PROC. OF 13TH EUROMICRO CONFERENCE ON DIGITAL SYSTEMS DESIGN (DSD10)*, 2010, s. 805–808.

[15] Pike, R.: Lexical Scanning in Go. Sydney Google Technology User Group, 2011. Dostupné z: `http://blog.golang.org/two-go-talks-lexical-scanning-in-go-and`

Příloha **A**

Seznam použitých zkratek

BIST Built-In Self-Test; vnitřní test

BLIF Berkeley Logic Interchange Format

CA Cellular automaton; buněčný automat

CMISR CA-based multiple-input signature register

CNF Conjunctive normal form; konjunktivní normální forma

CUT Circuit Under Test; testovaný obvod

DNF Disjunctive normal form; disjunktivní normální forma

HDL Hardware Description Language

HW Hardware

JIT Just-in time compilation

OS Operating System; operační systém

ROM Read-only memory; paměť pouze pro čtení

LFSR Linear feedback shift register; lineární zpětnovazební posuvný registr

MISR multiple-input signature register

NP nondeterministic polynomial time; nedeterministicky polynomiální

ORA Output Response Analyzer; kompaktor odezvy

PRNG Pseudo-random number generator; pseudonáhodný generátor čísel

PRTG Pseudo-random test generator; pseudonáhodný generátor testů

SAT Boolean Satisfability Problem; problém splnitelnosti booleovských formulí

SW Software

TPG Test Pattern Generator; generátor testovacích vektorů

VHDL VHSIC Hardware Description Language

Příloha B

Uživatelský manuál

B.1 Sestavení a instalace

Program je psaný v jazyce Go. Pro jeho překlad je tedy třeba mít nainstalovaný některý z jeho překladačů. S současné době existují dvě hlavní implementace, gc a gccgo.

Doporučuji použít překladač gc, který lze stáhnout z adresy `http://golang.org`. Jeho instalace je jednoduchá a přímočará.

Po instalaci Go je třeba správně nastavit své vývojové prostředí, konkrétně je třeba nastavit proměnnou GOPATH, aby ukazovala do Vašeho vývojového adresáře.

V Adresáři `GOPATH/src` poté umístíte zdrojové kódy programu takto: `GOPATH/src/hullerob/bistg`

V adresáři `GOPATH/src/hullerob/bistgen/cmd/bistgen` poté stačí spustit příkaz `go build`, který sestaví tento program.

Výsledný soubor je staticky sestavený a nevyžaduje žádné další knihovny pro svůj běh.

B.2 Užití

Tento program je obládán z příkazové řádky. Jeho syntaxe je následující:

`bistgen <volby> <příkaz> <soubor>`

Volby jsou:

- mem - omezuje množství paměti, které BIST v obvodu využije
- length - omezuje dobu po kterou test běží
- f - formát výstupního souboru, BLIF nebo VHDL
- fault - název souboru s preferovanými poruchami, které chceme pokrýt
- effort - nastavuje jak moc se má program snažit najít co největší pokrytí

- report - jméno souboru, do kterého zapíše zprávu o průběhu generování testu
- seed - počáteční nastavení pseudonáhodného generátoru

Příkazy jsou:

- echo - přeloží soubor z BLIF do zvoleného výstupního formátu
- faults - vypíše seznam poruch v načteném obvodu
- bist - tento příkaz vygeneruje strukturní test a zapíše jeho logický obvod ve zvoleném formátu

Příklad použití:
`bistgen -f vhdl -o bist.vhdl -mem 4 -length 10 bist c880.blif`

Upozornění Program nepodporuje DOS konce řádků. Soubor s takovými řádky nenačte.

Příloha C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `bin` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF