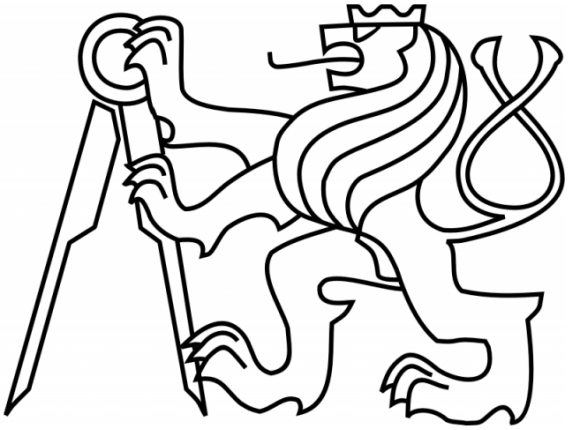
České vysoké učení technické v Praze
Fakulta elektrotechnická



Diplomová práce

Rychlý simulátor poruch

Jan Blažek

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika i informatika dobíhající magisterský

Obor: Informatika a výpočetní technika

listopad 2007

Poděkování

Rád bych na tomto místě upřímně poděkoval svému vedoucímu diplomové práce, Ing. Petru Fišerovi, za přínosné konzultace, poskytnutou literaturu a podporu při psaní a tvorbě této práce.

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně a použil jsme 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 ....
podpis

Anotace

V této práci bude představen rychlý simulátor poruch pro kombinační logické obvody. Simulátor využívá pro zrychlení techniku paralelní simulace poruch, která umožňuje otestovat až 64 poruch, uložených v bitovém slově, jedním průchodem obvodu. Dále bude uveden princip použití dominátorů v obvodu, který simulaci doplňuje a zrychluje. Rychlost simulátoru je v práci porovnávána s algoritmem HOPE, použitý v systému Atalanta.

Abstract

In this thesis will be presented fast fault simulator of combinational logical circuits. Simulator uses speed up technique of parallel fault simulation, which allows to test even 64 faults, stored in bit word, during one pass throught circuit. Further will be presented principe of dominators in circuit, which complements and speeds up simulation. Simulator speed is compared to algorithm HOPE, used in system Atalanta.

Obsah

1. Úvod

1.1. Úvod do simulace číslicových obvodů

Simulace číslicových obvodů (kombinačních i sekvenčních) je velice důležité odvětví technických věd. Využívá se k ověřování správné činnosti číslicových systémů. Testování těchto systémů vychází z principu determinismu funkcí, poruch a struktury celého systému jako celku. Jakoukoliv změnu v systému, tedy i poruchy, je tedy možné rozpoznat.

Zjištění správné funkce číslicových systémů je především důležité pro výrobce těchto obvodů. Testování se provádí již během samotného navrhování systému. Při každém kroku ve vývoji je třeba novou verzi vždy důkladně prověřit, než se započne v dalších fázích vývoje. Ale i po správném a bezporuchovém návrhu obvodu je nutné testování i při samotné výrobě obvodů. Nelze totiž zaručit bezporuchovou výrobu a ani výrobcům se nevyplatí neustále vylepšovat technologii výroby. Již předem se proto počítá s určitým procentem vadných součástek. A čím dříve se porucha v celém procesu od návrhu až po výrobu součástek objeví, tím je oprava méně nákladnější. Testování se však u některých aplikací (například ve zdravotnictví) musí provádět i za provozu systému. Proto jsou v takových systémech navíc zabudovány obvody pro testování – BIST (Built-in Self Test) – logika přidaná k systému, která umožňuje snažší, rychlejší a levnější testování obvodů.

Simulace číslicových systémů se provádí pomocí testů – sekvence dat, které jsou vybrány tak, aby, pokud možno, ověřily všechny funkce celého systému. Porucha v systému se projeví tak, že po aplikaci vstupních dat, které jsou systémem zpracovány a vykonány, poskytne systém odlišné výsledky ve srovnání s bezporuchovým stavem systému. Pokud se tedy porucha projeví na výstupu obvodu, stává se chybou, kterou je nutné odstranit.

Princip testování obvodů není tedy příliš složitý. Stačí "pouze" ověřit správnou funkci obvodu pro všechny možné kombinace vstupních vektorů (triviální test). Tento teoretický předpoklad však nelze v praktickém testování použít, vzhledem k obrovskému množství testovacích vektorů dat. Počet vstupních vektorů roste exponenciálně s počtem vstupů obvodu. Pokud by měl systém např. 50 vstupů a k otestování jednoho vstupního vektoru by bylo třeba 1µs, tak otestování všech vektorů (2^{50}) by trvalo zhruba 35 let! Pro redukci počtu vstupních vektorů se používají systémy ATPG (automatické generování testovacích vektorů). Tyto vektory jsou pak použity již při samotné simulaci.

Základním úkolem simulace obvodů je zkrátit čas potřebný pro otestování vstupních dat (vektorů) a zjistit, zda se v systému objevila nějaká porucha. Tímto problémem se budu v této práci dále zaobírat.

1.2. Cíle práce

Základním cílem této práce je vytvořit program, který umožní rychlou simulaci poruch v obvodu. K tomuto účelu je nutné nastudovat množství materiálů. Proto je možné vytyčit i další cíl. Tím je zmapování a popsání různých technik potřebných pro simulaci. Přeci jen není problém simulace tak často řešen, jako problém generování testovacích vektorů (ATPG). V českém jazyce nejsou materiály prakticky vůbec. Proto si myslím, že tato práce poskytne množství důležitých poznatků pro případné další řešitele simulace obvodů.

Rychlá simulace bude umožněna díky implementaci paralelní poruchové simulace založené na simulaci více poruch v jednom simulačním cyklu. Paralelismus je založen na uložení informací do bitového registrového slova. Dále je nutné implementovat pokročilejší techniky, které simulaci dále urychlí.

Úkolem simulátoru je odhalit (detekovat) poruchy, které jsou do obvodu vloženy (injektovány). Poruchy jsou detekovány na výstupu obvodu tak, že se liší správná hodnota na výstupním vodiči od hodnoty při vložené poruše do obvodu.

Simulované poruchy (fault list) jsou programu předávány ze vstupního souboru, taktéž i vstupní vektory. Výstupem programu je informace o tom, kolik poruch bylo detekováno, případně jaké poruchy jednotlivé vstupní vektory detekovaly.

1.3. Základní pojmy

1.3.1. Typy poruch

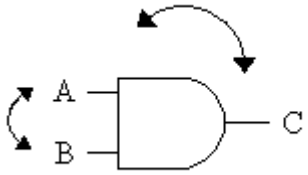
Příčiny poruch v číslicových systémech mají různé fyzikální či chemické příčiny. Nejčastěji se jedná přerušení vodivé cesty (rozpojený vodič), zkrat mezi vodiči či zkrat mezi různými metalickými vrstvami. Dále to mohou být poruchy zpoždění, pomalý náběh/sestup, výkonové změny...

Tyto poruchy se v obvodech projevují buď trvale (nezávisle na okamžiku testování), nebo jen dočasně (projevují se jen za specifických podmínek). Při simulaci se lépe odhalují trvalé poruchy. Pro naše potřeby této práce se spokojíme se simulací

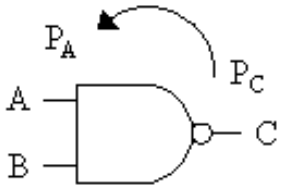
poruch trvalých – jsou to porucha typu trvalá 0 – Sa0 a trvalá 1 - Sa1 (Stuck-at 0, 1). Na daném signálním vodiči tedy bude trvale hodnota log. 0 nebo log. 1. Chyba systému je jev, který nastane v důsledku vzniku poruchy. Je to rozdíl mezi správnou a skutečnou hodnotou výstupní veličiny. Ovšem ne každá porucha musí v obvodu způsobit chybu [3].

1.3.2. Dominující a ekvivalentní poruchy

Porucha A dominuje poruchu B, pokud každá sada vektorů, které detekují poruchu A, detekují i poruchu B (nemusí ale platit opačné pravidlo, že vektory detekující poruchu B, detekují i poruchu A). Na obrázku *1.3.1*. Je znázorněna dominance poruchy P_C (Sa0) poruše P_A (Sa1). Porucha P_C je detekovatelná pro vektory (0, 0), (0, 1) a (1, 0). Poruchu P_A lze odhalit jen pro vstupní vektor (0, 1). Proto pokud detekujeme poruchu P_A, víme, že byla detekovaná i porucha P_C.



Obrázek 1.3.2: Ekvivalence poruch



Obrázek 1.3.1:Dominance poruch

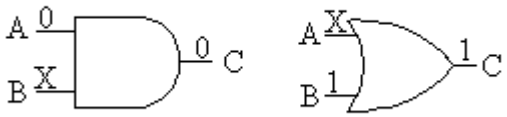
Ekvivalentní poruchy jsou poruchy, které lze detekovat shodnými sadami vektorů. Tyto poruchy se též v obvodu projevují se stejným efektem. Obrázek *1.3.2*. ilustruje ekvivalenci poruch Sa0 u obou vstupů i výstupu. Všechny tyto tři poruchy lze detekovat jen a pouze vektorem (1, 1). Jinými vstupními vektory nelze detekovat ani jednu z těchto poruch. Tyto poruchy tak tvoří stejnou třídu ekvivalence a při simulaci stačí testovat jen jednu z těchto poruch.

Znalostí těchto závislostí poruch můžeme snížit počet poruch, které musíme otestovat a tím snížit čas potřebný k simulaci. Jak lze tyto poznatky reálně využít a jak ovlivňují simulaci, ozřejmím dále v této práci.

1.3.3. Tříhodnotová logika

Vedle klasických binárních hodnot (0, 1) budeme používat i třetí stav, označený písmenem X. Tato hodnota se obvykle u vstupních vektorů používá pro označení hodnoty, na které 'nezáleží' (z angl. don't care) či je neznámá. V této práci proto

budeme při simulaci s touto hodnotou pracovat jako s neznámou hodnotou. Při reálné simulaci je na fyzickém vodiči samozřejmě nějaká hodnota (0, 1), ne žádná nová hodnota X. Použití hodnot X může být i výhodné, neboť v takovém případě nemusíme na daném vodiči zajistit určitou hodnotu (0, 1), přesto však budou poruchy detekovány, ať je na daném vodiči jakákoliv hodnota. Na obrázku *1.3.3.* je příklad použití hodnot X u hradel AND a OR.



Obrázek 1.3.3: Aplikace tříhodnotové logiky

Definované hodnoty (0, 1) na vstupech hradel jsou dominující pro daná hradla, na výstupu je tedy vždy definovaná hodnota, ať je na vstupních vodičích s označenou hodnotou X jakákoliv skutečná hodnota. Zároveň můžeme použít pro hradlo AND vstupní vektor (0X) pro detekci poruchy Sa1 na výstupu C, nemusíme tedy na vstupu B definovat žádnou hodnotu a přesto je porucha detekována.

1.3.4. Paralelní simulace

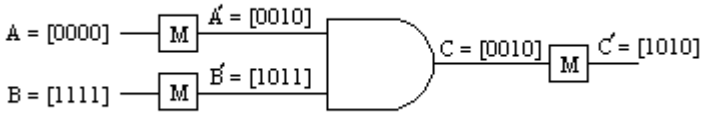
Pod tímto pojmem si lze představit dva druhy simulací, které se od sebe výrazným způsobem odlišují. Asi první nás napadne simulace na více procesorech současně, kde lze například rozdělit sadu testovacích vektorů na několik podsad, a ty poté simulovat odděleně na jednotlivých procesorech. V této práci však sousloví paralelní simulace vyjadřuje něco zcela odlišného. Jedná se o paralelismus na úrovni registrů procesoru.

Jednotlivé bity registrového slova (či spíše proměnné) lze využít pro uložení nějaké informace. Častěji používané metody ukládají do bitů hodnoty více vstupních vektorů a jeden bit je použit pro vloženou poruchu v obvodu. My se však budeme zabývat druhou variantou paralelismu – tedy vložení více poruch do obvodu současně a jejich odsimulováním jedním vektorem. Proměnná, které ohodnocuje hradlo, uchovává bezporuchový stav obvodu a zároveň stav, kdy je vloženo do obvodu více poruch současně (nejčastěji se používá 32 a 64 bitů). Anglická zkratka SPPFP (single pattern parallel fault propagation).

Hlavní výhodou této metody je využití rychlých instrukcí procesoru při použití bitových operací při simulaci obvodu. Touto simulací jedním průchodem obvodu otestujeme naráz velké množství poruch, při stejném simulačním čase, jako při simulaci jen jedné poruchy. Vedle rychlosti je další výhodou této metody i nízká paměťová náročnost, oproti deduktivní či souběžné simulační metodě..

Simulace obvodu s více vloženými poruchami je znázorněno obrázkem 1.3.4. Pro zjednodušení použijeme registrové slovo o velikosti 4 bity, kde první bit zprava

uchovává správnou hodnotu. Nastavení bitů registrového slova ilustruje tabulka 1.3.1. Jakým přesným způsobem se bity nastavují bude uvedeno v popisu mého řešení dále v této práci, kde popíšu i nastavení pro tříhodnotovou logiku.



Obrázek 1.3.4: Simulace hradla AND se třemi vloženými poruchami

Pozice ve slově	Porucha	Maska
1	bez poruchy	-
2	A Sa1	0010
3	B Sa0	0100
4	C Sa1	1000

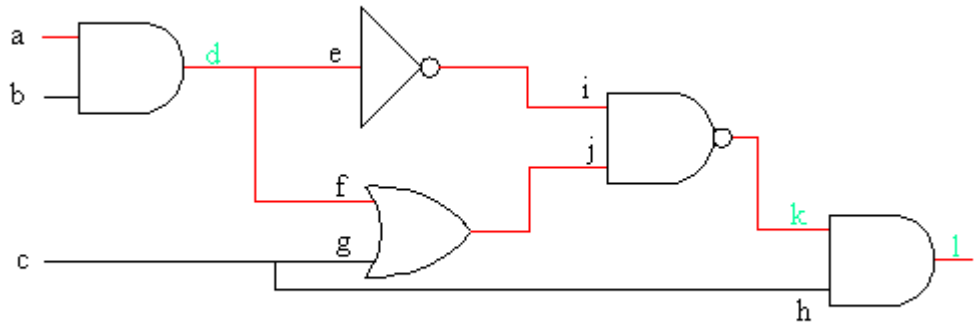
Tabulka 1.3.1: Nastavení bitů registrového slova

Hodnoty signálních vodičů v bezporuchovém stavu obvodu (bez vložené poruchy), jsou uchovávány ve slovech na první pozici (bráno zprava). V každém simulačním kroku je nejprve vypočtena hodnota na signálním vodiči a poté se na příslušné místo ve slově vloží porucha. Každá porucha musí mít samozřejmě svojí vlastní pozici, kterou si musí stále uchovávat při průchodu obvodem.

Výstupní slovo vyjadřuje hodnotu výstupu pro bezporuchový stav obvodu a stav po vložení tří poruch. Porucha je detekována, pokud hodnota na dané pozici odpovídající poruše, je různá od bitu na pozici 1. V našem příkladě jsou rozdílné bity 2 a 4 (zprava). To nám značí, že jsou detekovány poruchy A/Sa1 a C / Sa1. V jednom simulačním průchodu obvodem jsme tedy otestovali tři poruchy na místo jedné, což je hlavní přínos paralelní simulace [4].

1.3.5. Dominátory (dominators)

Dominátory libovolného signálního vodiče (z angl. signal line) v obvodu jsou prvky (hradlo, jiný signální vodič), přes které vedou všechny cesty ze signálního vodiče k primárnímu výstupu [2]. Těchto dominátorů může mít signální vodič i více, pro potřeby simulace však postačí použít jen jediný. Pro všechny signální vodiče, které jsou na cestě k dominátoru, se tento dominátor stává také dominátorem. Na obrázku *1.3.5.* je zvýrazněn průchod signálu ze vstupu *a* obvodem až na výstup.



Obrázek 1.3.5: Průchod vstupu 'a' obvodem s vyznačenými dominátory

Na cestě k výstupu prochází signál i několika dominátory. V tomto případě jsou to signální vodiče *d, k* a výstup *l*. Který z těchto dominátorů bude použit při simulaci, či jestli vůbec nějaký použit bude, popíši už při konkretní implementaci řešení dále. Jak již bylo uvedeno výše, tak i pro signální vodiče na cestě k dominátoru, se dominátor stává dominátorem. Tyto vodiče však mohou mít i další dominátory. Například vodič *f* má nejen dominátory *k* a *l*, ale navíc je jeho dominátorem i vodič *j*.

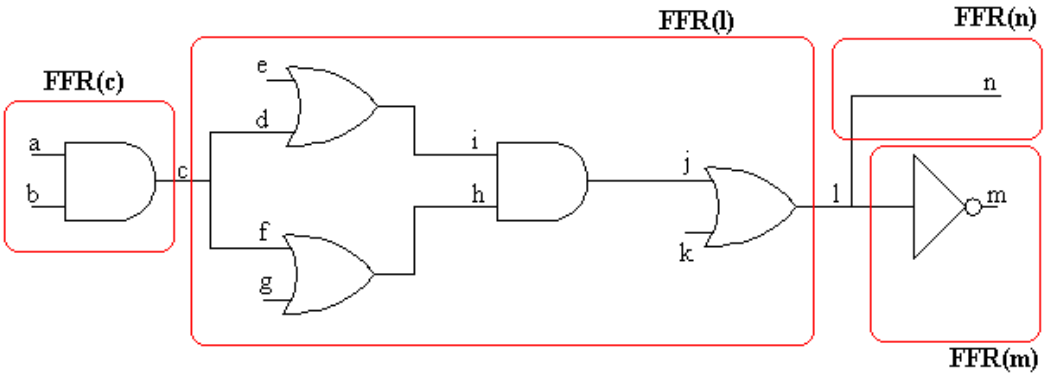
Proč se však dominátory vůbec zabývat? Odpověď se nabízí již ze samotné definice dominátoru. Pokud se má porucha propagovat na výstup, musí se vždy propagovat i přes dominátor. Proto při simulaci stačí zjistit, zda se porucha propaguje až k dominátoru, a pokud ano, tak dále už se pracuje jen s dominátorem. V ten okamžik jsou dvě možnosti, jak postupovat dále. Když byla odpovídající porucha na dominátoru pro stejný vstupní vektor již testována (ať již detekována na výstupu či ne), můžeme poruchu, kterou jsme k dominátoru propagovali, označit též za otestovanou pro daný vstupní vektor s výsledkem shodným s poruchou na dominátoru (detekována / nedetekována).

Tímto postupem lze výrazně zkrátit cesty potřebné na propagaci poruch a tím zrychlení simulace. Před samotným popisem, jak jsou v této práci dominátory využity, je třeba předeslat, že samotné dominátory se neuplatňují přímo v paralelní simulaci, ale jsou doplňujícím prvkem, vycházejícím ze simulace sériové.

1.3.6. Souběhy (stems)

Souběhem rozumíme místo v obvodu, do kterého vedou všechny signální cesty z jednoho či více primárních vstupů, nebo všechny signální cesty z jiného souběhu(ů) v obvodu. Souběhy tedy vlastně separují obvod do několika částí, které ovlivňují

ostatní části obvodu pouze změnou hodnoty v souběhu. Jakákoliv porucha uvnitř této 'uzavřené' oblasti se projeví nejprve v souběhu této oblasti, poté se teprve může propagovat dále v obvodu. Tyto oblasti jsou oblasti bez 'rozvětvení' do dalších částí obvodu (angl. fanout free region – FFR). Z definice souběhu je tedy patrné, že každý souběh je dominátorem všech signálních vodičů uvnitř oblasti souběhu.



Obrázek 1.3.6: Souběhy (stems) v obvodu

Dominátory, které nejsou souběhy, se při simulaci využívají především právě u souběhů. U ostatních signálních vodičů pouze tehdy, pokud tyto vodiče nejsou v oblasti bez rozvětvení. Tedy pokud jejich dominátorem není souběh.

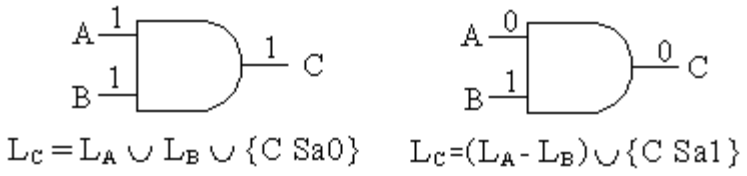
1.4. Další metody simulace

1.4.1. Sériová simulace

Sériová simulace je základní simulací obvodu. Sériová se nazývá proto, že v jednom průchodu obvodem otestuje jen jednu poruchu. Je tedy zřejmé, že je pomalejší než paralelní simulace. Přesto se tato simulace využívá jako doplňková simulace. I v této práci bude použita. Výhodou je, že lze brzy odhalit, že porucha nemůže být v obvodu propagována dále (bude dále vysvětleno). Dále se tato používá pro simulaci obvodů pro zjištění poruch zpoždění [3].

1.4.2. Deduktivní simulace (deductive fault simulation)

Velkou předností této metody je, že celý obvod je odsimulován jen jednou. Při simulaci se pro každý signální vodič vytváří a upravuje seznam poruch. Na základně



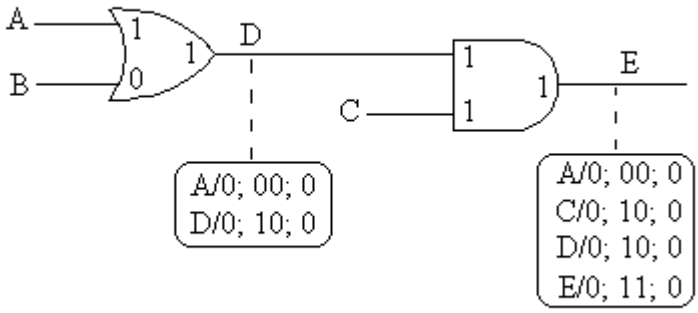
Obrázek 1.4.1: Deduktivní simulace

bezporuchových hodnot signálů se 'deduktivně' u každého logického členu zjistí, které poruchy se mohou propagovat na výstup hradla. Popis této metody je naznačen obrázkem *1.4.1*. Pro první hradlo AND se propagují na výstup seznamy poruch z obou vstupů, jelikož oba dva vstupy jsou v log. 1 – oba vstupy jsou citlivé na změnu hodnoty signálu. Zároveň se na výstupu přidá k seznamu i porucha C Sa0. Na obrázku u druhého hradla se mohou propagovat jen poruchy, které jsou v seznamu vstupu A (jen tento vstup je zcitlivěný). Ovšem ze seznamu poruch u vstupu A se musí vyjmout poruchy, které jsou stejné i v seznamu vstupu B. Tato stejná porucha by změnila hodnotu vstupu B a tím by vstup A již nebyl citlivý na změnu, a proto by se tato porucha nemohla stejně dále propagovat.

Tento způsob simulace potřebuje jen jeden průchod obvodem, tento průchod však trvá déle, než třeba u paralelní simulace (ale i sériové). Celkově je však tato metoda rychlejší než paralelní simulace, má však mnohem větší nároky na paměť[3].

1.4.3.Souběžná simulace (concurrent fault simulation)

Souběžná simulace je podobná simulaci deduktivní. Pro každý člen obvodu je vytvořen seznam poruch. Tento seznam zároveň uchovává informace o vstupních a výstupních hodnotách daného členu. Poté se při událostmi řízené simulaci vybírají ty seznamy, které na sebe vhodně navazují[3]. V příkladě na obrázku *1.4.2* je ukázáno vybírání dílčích seznamů poruch u jednotlivých hradel.



Obrázek 1.4.2: Souběžná simulace

Vždy jsou vybrány jen ty vstupní poruchy, které se liší alespoň jednou hodnotou od správné vstupní hodnoty (u každé poruchy je uloženo, jaké je nastavení vstupů pro danou poruchu), a zároveň se liší i hodnotou výstupu (taktéž je u poruchy uložen). U výstupních poruch samozřejmě stačí jen změna od správné hodnoty výstupu. Ve druhém hradle se opět naleznou poruchy, které je možné vložit pro známé správné hodnoty. Tato metoda je také rychlejší než metoda paralelní, ale také je (jako deduktivní) mnohem náročnější na paměť.

1.5. Referenční software

Z existujících simulačních programů byl jako kontrolní a srovnávací program vybrán systém Atalanta[1], který v sobě zahrnuje části pro ATPG a simulaci obvodů. Tento systém obsahuje dva simulační programy – HOPE[2] a FSIM[1]. Program FSIM je umožňuje simulaci jen kombinačních obvodů (HOPE dovoluje simulovat i sekvenční obvody), také se oba liší tím, že pouze HOPE dovoluje používat tříhodnotovou logiku.

Oba dva algoritmy jsou dobře zdokumentovány, a i když jsou vytvořeny na počátku 90.let , tak poskytují velmi dobré výsledky. Jejich nevýhodou však je, že nepracují dobře pro větší obvody.

Systém Atalanta jsem zprvu používal jen pro kontrolu správnosti mého simulátoru, pro generování testovacích vektorů a také k získání seznamů poruch. Teprve poté, co jsem vyčerpal všechny své nápady pro vylepšování mého simulačního programu, jsem použil některé techniky z tohoto systému.

Po nastudování principů obou výše zmíněných algoritmů jsem zvolil postupy z programu HOPE. Ten pracuje, stejně jako tento program, na principu propagace více poruch a jednoho vstupního vektoru v jednom simulačním kroku.

1.5.1. HOPE

Simulátor, který je popsán v této práci, je porovnáván s algoritmem HOPE. Ten je součástí ATPG systému Atalanta. HOPE využívá principu paralelní propagace 32 poruch v bitovém slově naráz. Pro urychlení simulace používá princip oblastí bez rozvětvení (obdobný principu dominátorů uvedený zde). Tento simulátor umožňuje také simulaci vektorů s hodnotami don't care. Algoritmus HOPE navíc dokáže simulovat i sekvenční obvody. Více informací se lze dočíst v dokumentaci[2].

2. Analýza a zpracování problému

2.1. Úvod

Na programové části této diplomové práce jsem pracoval v průběhu 14 měsíců. Nevycházel jsem z žádného již existujícího řešení, které bych vylepšoval. Musel jsem tedy při vytváření programu postupovat od úplného počátku – načítání vstupních dat, vytváření struktur v paměti... V úvodních měsících bylo hlavním cílem vytvořit funkční simulátor poruch, nejprve sériový, který jsem později rozšířil na simulátor paralelní. Postupem času jsem se snažil implementovat vlastní techniky a nápady na zrychlení simulace.

Při samotném utváření simulátoru jsem zpočátku vystačil s informacemi z předmětu *Diagnostika a spolehlivost* a dostupnými materiály, které jsou pro něj dostupné. Po vyčerpání vlastních nápadů, jsem se teprve seznámil s materiály, které popisují simulační algoritmy HOPE a FSIM, ty jsou součástí ATPG systému Atalanta.

Správnost mého řešení a rychlost programu jsou v celé práci vždy porovnávány se systémem Atalanta, konkrétně se simulátorem HOPE.

2.2. Načítání vstupních souborů a vytváření struktur

2.2.1. Struktura obvodu

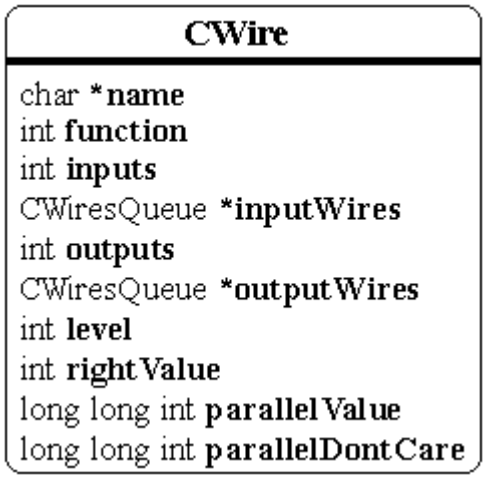
Načítání vstupních souborů s popisem obvodu je nejobtížnější část z fáze načítání struktur. Hlavním úkolem bylo důkladně nastudovat možné formáty, které tento soubor popisují, důsledně se jich držet. Možné vstupní formáty souboru jsou ISCAS '85 a ISCAS '89. Tyto formáty jsou podporovány i systémem Atalanta, což byl další důvod jejich použití v této práci.

Pro načítání souboru s popisem obvodu bylo nutné vytvořit lexikální analyzátor, který umožní identifikovat jednotlivé objekty v obvodu. Velikost načítaných souborů (počet hradel) je téměř neomezené, na rozdíl od systému Atalanta, který má bohužel některá svá omezení.

Veškerá načítaná data jsou ukládána do již předem vytvořených datových struktur. Pro datové struktury jsem používal objekty, a to především kvůli přehlednosti a zjednodušení. Mají však tu nevýhodu, že vytváření takových struktur je mírně časově náročnější.

Veškeré struktury jsem vytvářel v době, kdy jsem měl sice problematiku simulace teoreticky nastudovanou, ale neměl jsem znalosti praktické, ani informace, ze kterých bych mohl vycházet. Přesto jsem mnou vytvořené struktury v průběhu práce neměnil, tedy až na některé dílčí atributy.

Základní stavební jednotkou celého obvodu je třída *CWire*, jejíž základní popis je naznačen obrázkem *2.2.1*. Ta reprezentuje libovolný signální vodič v obvodu.



Obrázek 2.2.1: Základní stavební prvek obvodu

Každý člen je identifikovatelný svým jménem (*name*). Atribut *function* popisuje chování hodnoty na daném vodiči v závislosti na jeho vstupech. Jeho pomocí identifikujeme, zda je daný vodič výstupem hradla (AND, OR, NOT...), vstupním vodičem či rozvětvením jiného vodiče. Dále si u prvků ukládáme informace o počtu vstupních a výstupních signálních vodičích, tyto vodiče zároveň ukládáme do datové struktury *CWiresQueue*. Tou je běžný spojový seznam s přidanými metodami pro přidávání, odstraňování, výpis prvků. Atributy pro uchování logických hodnot jsou *rightValue*, ta se využívá pro ukládání správné (bezporuchové) hodnoty. Pro paralelní simulaci jsou využity atributy *parallelValue* a *parallelDontCare*. Oba dva jsou použity pro uložení tříhodnotové logiky. Toto ukládání hodnot bude vysvětleno dále. Poslední zde uvedený atribut *level* slouží pro zaznamenání úrovně daného vodiče v obvodu. Výpočet úrovně vodiče popíši v další podkapitole.

Popisovaná struktura obsahuje i další atributy, ty však již nejsou podstatné a slouží pouze například jako příznaky pro některé operace. Takto vytvořená struktura vytvoří důležitá spojení mezi závislými prvky, a proto poté, při samotné simulaci, se potřebná data získají jednoduchým 'sáhnutím' do paměti, namísto zdlouhavého procházení jiných struktur. Na druhou stranu samozřejmě spojení mezi závislými prvky mírně prodlouží proces načtení obvodu.

Při načítání struktury obvodu jsem narazil na první větší problém v práci, který vyústil ve změnu a mírné prodloužení samotného načítání obvodu. Problém byl způsoben v rozdílném ukládání jednotlivých prvků obvodu v souboru u některých kombinačních obvodů (cxxxx.bench) a kombinačních verzí obvodů sekvenčních (sxxxx.bench).

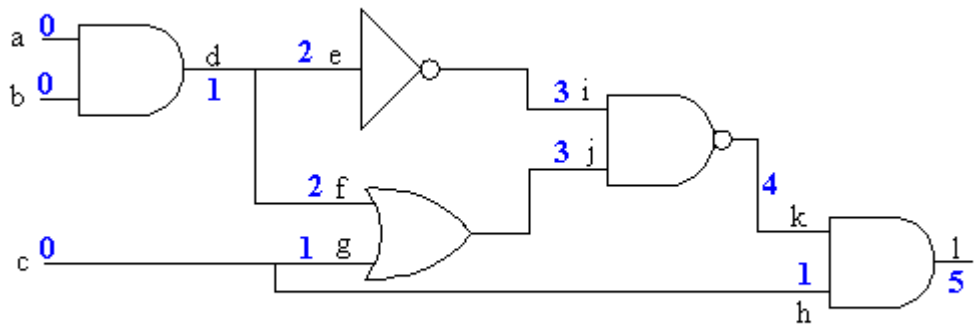
Obvody kombinační jsou většinou v souboru ukládány v logické posloupnosti. Pokud je z takového souboru načítán vodič (hradlo), jsou veškeré jeho vstupy uvedeny

již dříve a tedy již i vytvořeny v samotné struktuře obvodu. Tato logická souvislost však ale u sekvenčních obvodů není. Proto je nutné před každým vytvořením prvku nejprve ověřit, zda daný prvek již nebyl vytvořen dříve (jako vstupní vodič jiného vodiče). Toto ověřování způsobí ono mírné prodloužení načítání obvodu.

2.2.2. Výpočet úrovně vodičů

Úroveň vodičů je velmi důležitý atribut prvků v obvodu. Díky tomu můžeme uložit jednotlivé prvky obvodu do struktury, která nám usnadní a zrychlí simulaci. Jedním z předpokladů pro simulaci, či lépe pro nastavení hodnoty vodiče, je, že v okamžiku výpočtu hodnoty prvku, jsou známy hodnoty všech prvků na jeho vstupu. To zajistíme právě díky úrovním vodičů, kdy všechny vodiče uložíme do struktury rozdělené podle úrovní (například dvourozměrný zřetězený seznam). Poté při výpočtu hodnot na vodičích postupujeme po úrovních od nulté výše, a vždy tedy bezpečně víme, že je na vstupu zpracováváného prvku již nastavená hodnota.

Algoritmus výpočtu úrovně jednotlivých vodičů je poměrně jednoduchý. Výchozím bodem jsou vstupní vodiče. Ty jsou na nulté úrovni obvodu. Jako datovou



Obrázek 2.2.2: Úrovně vodičů v obvodu

strukturu pro práci s vodiči použijeme frontu. Samotný algoritmus se pak skládá z těchto částí:

1) nastavení vstupních vodičů na úroveň 0 a jejich uložení do zásobníku ostatní vodiče považujeme za nenastavené (úroveň -1)
2) vyjmutí prvku z počátku fronty
   pro každý jeho výstupní vodič :
   - pokud je úroveň výstupního vodiče menší nebo rovna, úrovni vstupního vodiče:

a) úroveň výstupního vodiče = úroveň vstupního + 1
b) ulož výstupní vodič do fronty

Nastavení úrovní vodičů pomocí tohoto algoritmu si lze ověřit za pomoci obrázku *2.2.2.,* kde jsou tyto hodnoty uvedeny. Při tomto výpočtu zároveň zjistíme i maximální úroveň vodiče v obvodu, kterou později použijeme pro vytváření dalších struktur.

Po výpočtu úrovní uložíme vodiče do již zmíněného dvourozměrného zřetězeného seznamu. V této práci se skrývá pod strukturou *CLeveledWires.* Zde opět doplněná o základní operace pro operace s touto strukturou.

2.2.3. Načtení seznamu poruch

Formát souboru se seznamem poruch je jednotný a tak je i vytvoření lexikálního analyzátoru snažší (i formát vstupních dat je poměrně jednoduchý) v porovnání s načítáním souboru s popisem obvodu.

Poruchy jsou ukládány do nové struktury *SFaultList.* V ní si zaznamenáváme nejen typ poruchy, ale i odkaz na vodič, ke kterému porucha patří. U tohoto vodiče zase naopak uložíme odkaz na poruchu. Proto ke struktuře *CWire* byly přidány dvě datové položky, které uchovávají odkazy na případné poruchy (Sa0, Sa1). Tímto propojíme seznam poruch se strukturou obvodu a naopak. To nám dále pomůže k rychlému přístupu k datům.

2.2.4. Načtení vstupních vektorů

V tomto případě je načtení struktury vůbec nejjednodušší. Vstupní vektory se skládají z posloupností nul a jedniček, popřípadě znaků (x, X, -, 2,), které zastupují hodnotu don't care (neznámá hodnota).

Pro snížení nároků na operační paměť jsem se rozhodl jednotlivé hodnoty ukládat jako dvoubitová čísla (00 – log. 0, 01 – log. 1, 11 – X). Takto je do jedné 64-bitové proměnné možné uložit 32 vstupních hodnot, namísto uložení jedné hodnoty do třeba 32-bitového integeru.

2.3. Implementace paralelní simulace – první verze

2.3.1. Kódování hodnot tříhodnotové logiky

Při používání tříhodnotové logiky je nutné vyřešit problém, jak ukládat tři možné stavy do bitového slova a zároveň si uchovat výhody paralelní simulace. Rozhodl jsem se použít dvě 64-bitová slova (*parallelValue* a *parallelDontCare*). První z nich uchovává hodnotu a druhé informaci o tom, zda je daná hodnota platná, nebo zda není (X - don't care). Samotné uložení hodnoty je jednoduché – pokud je bit na požadované pozici ve slově *parallelDontCare* nastaven na 1, je daná hodnota neznámá, v opačném případě je hodnota na stejné pozici ve slově *parallelValue* platná.

Správné nastavení hodnot do obou slov se liší u každé bitové operace (u každého typu hradla). Základním nastavením je uložení hodnot u vstupních vodičů obvodu (*tab. 2.3.1.*). Jednoduché je i nastavení u výstupních vodičů hradel typu buffer a u rozvětvení vodičů. U obou se jen zkopírují obě hodnoty ze vstupu. Obdobné je nastavení i u hradla NOT. Liší se jen tím, že všechny bity ve slově *parallelValue* se musí znegovat. Zjištění hodnoty u ostatních hradel (AND, NAND, OR, NOR, XOR) je již složitější, a proto ho vysvětlím podrobněji.

hodnota	*parallelValue*	*parallelDontCare*
0	0	0
1	1	0
X	0	1

Tabulka 2.3.1: Uložení paralelní hodnoty u vstupu

Pro nastavení výstupu u hradla AND jsou použity dvě pomocné proměnné. Zde pro ilustraci například *value* a *dontcare*. V prvním kroku se použije první vstup hradla AND. Proměnná *value* se nastaví na (*parallelValueI* | *parallelDontCareI*). Do proměnné *dontcare* se uloží hodnota *parallelDontCareI*. Poté, pro každý další vstup, se provádějí tyto operace (krok 2):

value = value & (parallelValueI | parallelDontCareI)
dontcare = dontcare | parallelDontCareI

kde hodnoty *parallelValueI* a *parallelDontCareI* jsou hodnotami vstupů (I – input).

Nakonec se uloží hodnoty do výstupu takto (krok 3):

parallelDontCareO = dontcare & value
parallelValueO = value & (~ dontcare),

zde jsou však již hodnoty *parallelValueO* a *parallelDontCareO* jsou hodnotami

výstupu (O – output). V tomto posledním kroku je nutné zachovat pořadí těchto dvou operací. (pozn.: znak '&' značí operaci bitový součin, znaménko '|' znamená bitovou operaci součet a '~' je bitová negace)

Cílem těchto operací je zajistit nastavení hodnoty výstupu hradla tak, že paralelní hodnoty obou slov odpovídají hodnotám logickým, jak je uvedeno v tabulce *2.3.1*. Nastavení ostatních hradel je obdobné jako u hradla AND, liší se jen typem bitových operací, podle typu hradla. Tímto příkladem jsem chtěl ukázat, že se nastavování výstupů neprovádí jen jednoduchou aplikací bitové operace na vstupní hodnoty, že je třeba tento mechanismus rozšířit.

2.3.2. Paralelní simulace

Při simulaci jsou v této práci použity 64-bitové proměnné. Proto můžeme při průchodu obvodem odsimulovat až 63 poruch – 1 bit je určen pro správnou, bezporuchovou hodnotu. Struktura obvodu je uložena již ve výše zmíněném dvourozměrném zřetězeném seznamu prvků. Ty jsou rozděleny podle své úrovně v obvodu.

Průchod obvodem začíná na úrovni 0 – na vstupních vodičích. Těm jsou přiřazeny hodnoty vstupního vektoru. Dále se pokračuje na úrovni 1, kde nastavíme všem vodičům hodnotu podle jeho vstupních vodičů a pokračujeme na další úrovni...

V první verzi mého simulátoru jsem nejprve vybral prvních 63 poruch. Informaci o tom, že bude porucha odsimulována jsem přidal k odpovídajícímu vodiči, kde se uloží nejen typ poruchy, ale i pozice, na kterou bude porucha ve slově umístěna. Zároveň se zaznamená, která pozice v bitovém slově odpovídá jaké vložené poruše. Poté jsme spustil průchod obvodem. Pokud se při procházení obvodem narazí na vodič, u kterého má být nastavena porucha, tak se tato porucha nastaví na příslušnou hodnotu a pozici. Po odsimulování obvodu se otestují všechny výstupní vodiče obvodu a zjistí se, zda byla nějaká porucha nalezena (rozdílný bit na dané pozici od první pozice a zároveň nesmí být hodnota na dané pozici neznámá).

Takto vytvořený simulátor je plně funkční, avšak při analýze principu jeho funkce je patrné, že je do obvodu zbytečně vloženo mnoho poruch, které určitě nemohou být detekovány na výstupu. Tím je zbytečně vysoký i počet průchodů obvodem a tím i délka simulace.

Proto jsem se rozhodl změnit způsob vkládání poruch do obvodu. Poruchy nejsou vkládány předem, ale vkládají se až při průchodu obvodem. Pokud se narazí na vodič, který má dosud nedetekovanou poruchu a počet vložených poruch v obvodu je

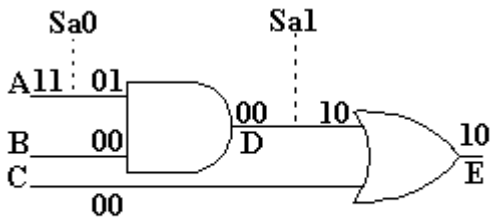
menší než 63, zjistí se, zda daná hodnota poruchy se liší od správné hodnoty na vodiči. Pokud se tyto hodnoty liší, porucha je vložena. Touto drobnou změnou ve vkládání poruch se rychlost simulace urychlí.

Dalším studiem chování poruch v obvodu jsem zjistil, že mnoho poruch ukončí svou propagaci velice brzy – projdou jen o několik úrovní dále, než v jaké úrovni byly vloženy.

2.3.3. Test propagace poruch

Tím, že většina poruch ukončí svou propagaci obvodem velmi brzy po vložení, dochází ke zbytečnému obsazení bitů v registrovém slově poruchami, které nemohou být detekovány. Tyto obsazené pozice by mohly být použity pro vložení jiných poruch.

Z těchto důvodů jsem implementoval do simulátoru systém, který umožní detekovat poruchy, které již dále nemohou být propagovány, a umožní tak použití volných pozic pro jiné poruchy. Při simulaci jsem zjistil, že z hlediska rychlosti se vyplatí testovat propagaci poruchy jen pro vodiče, které jsou vstupem hradel, které mají jen jeden výstup, tedy nejsou rozvětvené.



Obrázek 2.3.1: Testování propagace poruch

U hradel s více výstupy je již tento test časově náročnější, než samotný zisk z uvolnění pozice ve slově. Příklad je uveden na obrázku *2.3.1,* kde je velikost slova 2 bity – bit na první pozici zprava uchovává bezporuchový stav obvodu. Při procházení obvodem nejprve navštívíme vodič A, kde nastavíme poruchu Sa0. Zároveň u výstupního vodiče (D) nastavíme, že má otestovat propagaci poruchy na pozici 2. V okamžiku, kdy jsme v simulaci pokročili až k vodiči D, otestujeme, zda může být detekovaná porucha na pozici 2. Tato porucha být detekovaná nemůže, proto se nám uvolnila pozice v bitovém slově. Tuto pozice tedy využijeme pro poruchu D Sa1, kterou detekujeme na konci.

Jak je patrné i z příkladu, můžeme včasným odhalením nedetekovatelných poruch otestovat více poruch v jednom průchodu obvodem, než kolik nám dovoluje velikost bitového slova.

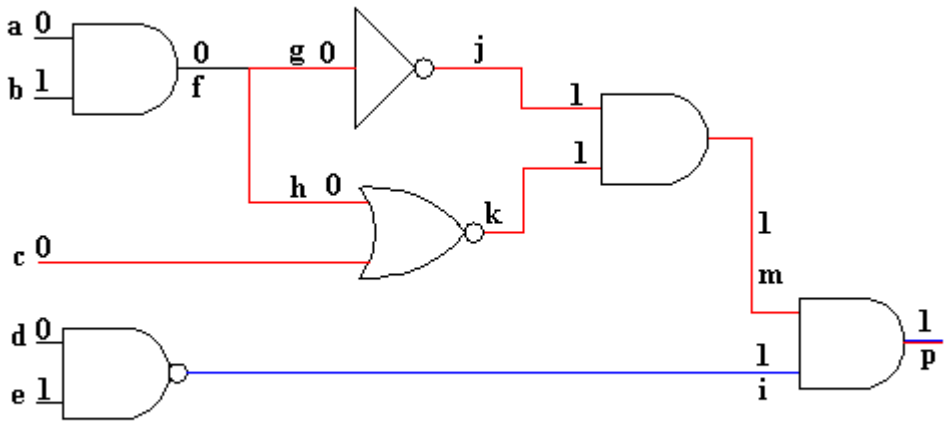
2.3.4. Zpětná analýza obvodu

Zpětnou analýzou zde myslím průchod obvodu od výstupních vodičů směrem ke vstupním, při kterém jsou detekovány poruchy za použití správných hodnot a znalosti topologie obvodu. Tyto poruchy mohou být detekovány i v simulaci dopředné, ovšem zde je výhodou, že se obvod projde pouze jednou. Takovéto odhalování poruch se dá přirovnat k hledání citlivých cest v obvodu, ovšem bez nastavování hodnot vodičů.

Analýzu začneme od výstupních vodičů, které vložíme do fronty. Poté, dokud není fronta prázdná, vyjímáme vodiče z počátku fronty. U tohoto vyjmutého vodiče nejprve otestujeme, zda můžeme na tomto vodiči detekovat dosud nenalezenou poruchu. Tento test je triviální – pokud je správná hodnota na vodiči log. 0, tak může být detekována log. 1, a opačně. Druhou fází je uložení některých jeho vstupních vodičů do fronty.

Pro další zpracování lze vložit do fronty jen ty vstupní vodiče hradla, které při změně své hodnoty, změní i hodnotu výstupu hradla. Například, pokud vyjmeme z fronty výstupní vodič hradla AND, který má správnou hodnotu log. 1, můžeme dále do fronty vložit všechny vstupní vodiče daného hradla – vstupní vodiče mají správnou hodnotu také log. 1, změna hodnoty libovolného z nich na log. 0 způsobí i změnu na výstupu hradla AND. Proto tato porucha může být detekována. Obdobně se algoritmus chová pro hradlo NAND s hodnotou log. 0 na výstupu, OR a NOR s výstupy log. 0, respektive log. 1. U hradla NOT můžeme vložit vstupní vodič kdykoliv. U ostatních hradel a stejně tak i u rozvětvení vodičů nelze tyto závislosti použít, protože nelze předem jednoduše rozpoznat (bez vícenásobných zpětných návratů), zda se citlivá cesta vytvoří a zda zpětně neovlivní danou citlivou cestu.

Na obrázku *2.3.2.* jsou v obvodu vyznačeny citlivé cesty, které byly nalezeny zpětnou analýzou. Poruchy, které jsou na těchto vodičích, je možné, vzhledem ke správným hodnotám, detekovat.

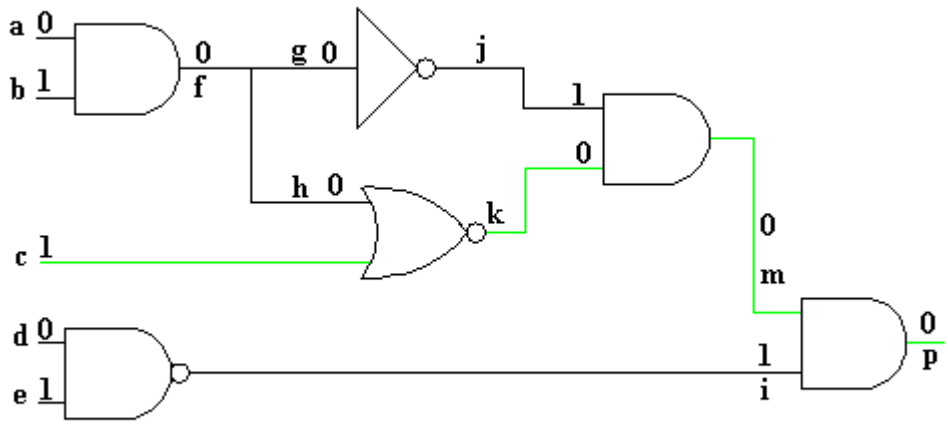


Obrázek 2.3.2: Zpětná analýza

Algoritmus začíná na výstupu *p*. Ten je výstupem hradla AND a jeho správná výstupní hodnota je log. 1. Pokud tedy existuje porucha *p* Sa0, můžeme jí označit za detekovanou. Z údajů o typu a hodnotě na výstupu víme, že oba dva vstupy při změně způsobí i změnu na výstupu hradla AND. Proto je oba uložíme do fronty a pokračujeme dále. Opět vyjmeme první prvek z fronty – nyní to bude vodič *i*. U něj lze detekovat poruchu *i* Sa0. Tento vodič je výstupem hradla NAND. Jelikož je ale hodnota na vodiči *i* log 1, nelze jednoduše určit, který ze vstupních vodičů způsobí změnu na výstupu (pokud lze vůbec změnou jednoho vstupu ovlivnit i výstup).

Proto z vodiče *i* nelze dále v zpětné analýze pokračovat. Vyjmeme tedy z fonty vodič *m* a postupujeme dále podle algoritmu. Takto se postupuje, dokud není fronta prázdná.

Výše popsaný algoritmus lze ještě vylepšit. Z obrázku *2.3.2.* je patrné, že by šlo při zpětné analýze detekovat i poruchu Sa1 na vodiči *d*. Stačí projít vstupy vodiče *i* a pokud hradlo NAND, jehož je *i* výstupem, má pouze jeden vstup ohodnocený log. 0, můžeme tento vstup označit za citlivou cestu.



Obrázek 2.3.3: Zpětná analýza - vylepšení

Tedy můžeme na tomto vstupu detekovat poruchu a zároveň můžeme přes tento vstup pokračovat ve zpětné analýze.

Vyhodnocení, zda má hradlo jen jeden "citlivý" vstup (vstup, jehož změna ovlivní výstup), jsem implementoval přímo simulace. Díky tomu u každého hradla předem víme, zda má smysl procházet jeho vstupy a hledat onen "citlivý" vstup. Vylepšením tohoto algoritmu můžeme aplikovat zpětnou analýzu i na obvod na obrázku *2.3.3*. V původní verzi algoritmu, bychom mohli odhalit jen poruchu na vodiči *p*. Tím bychom analýzu ukončily. Po vylepšení však lze projít i na vodič *m* a úspěšně pokračovat v analýze směrem ke vstupům.

Při teoretické analýze tohoto algoritmu se zdálo, že se povede simulaci hodně urychlit. Vždyť jedním průchodem obvodu by se dalo detekovalo velké množství poruch. Po implementaci a prvních testech se ale ukázala realita. Algoritmus opravdu dokáže detekovat velké množství poruch, avšak pouze v prvních fázích simulace, kdy je v obvodu více nedetekovaných poruch. Poté jeho účinnost rapidně klesá a jeho použitím simulaci prodloužíme. Z jeho definice je patrné, že lze použít jen jednou, pro každé nové nastavení vstupního vektoru. Použití více než jednou pro stejný vstupní vektor nemá smysl, protože jsou stále stejné hodnoty na vodičích. Testováním jsem dále zjistil, že jeho použití se vyplatí jen průměrně třikrát. Při více opakováních už odhalí minimálně poruch a je zbytečné jej použít.

Celkově je však jeho přínos marginální, neboť při simulaci obvodu jsou použity stovky vstupních vektorů a pro každý vektor je obvod procházen v desítkách případů. Množství těchto průchodů tedy zcela zastíní výhody algoritmu. Navíc je jeho účinnost snížena též strukturou vstupních poruch. Seznamy poruch, které jsou v simulaci

používány, jsou předem generovány z ATPG systémů – ty již vytvářejí redukované seznamy poruch, ve kterých se nevyskytují dominance a ekvivalence poruch, tedy typy poruch, který tento algoritmus především využívá. Nakonec jsem tedy tento algoritmus přestal používat, neboť jsem odhalil rychlejší způsoby detekce poruch.

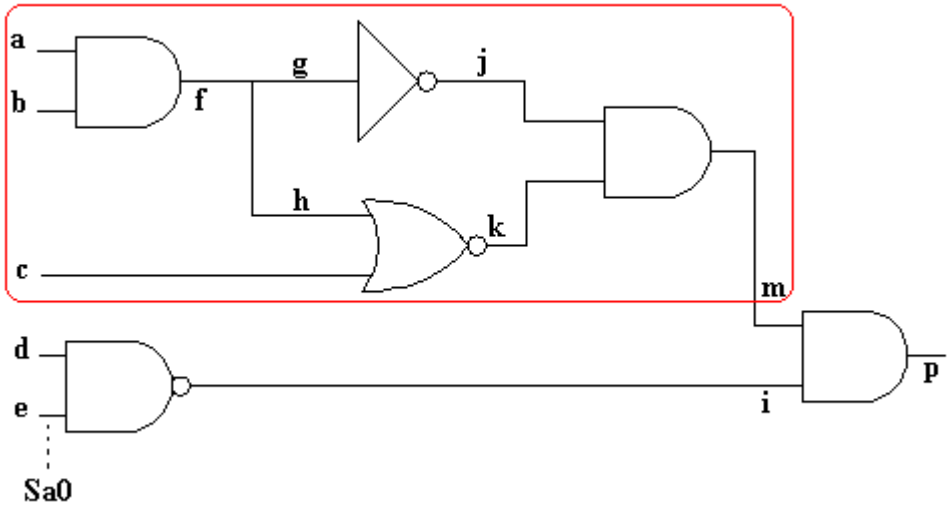
2.3.5. Dominance a ekvivalence poruch

Teoretický základ k tomuto tématu byl již uveden v úvodu této práce. Cílem využití těchto závislostí je snížit počet testovaných poruch. I v tomto simulátoru jsem zprvu implementoval jejich detekci. Po prvních testech se však i zde odhalil vliv struktury seznamu poruch generovaný ATPG systémem. Tyto systémy však již dominantní a ekvivalentní poruchy odstraňují. Nemá tedy smysl i při simulaci testovat, zda tyto závislosti existují, zvláště pokud toto testování nelze provést v konstantním čase.

2.3.6. Omezení průchodu obvodem

Prozatím jsem se v teoretických úvahách snažil snížit počet poruch, které je třeba otestovat při paralelní simulaci. Dalším problémem při simulaci je však i zbytečné procházení některých hradel (vodičů). Dosud se při simulaci stále používala struktura obvodu, která byla vytvořena při načtení obvodu ze souboru. Tento princip má výhodu v tom, že se obvod nemusí vytvářet za chodu simulace, tedy šetří výpočetní čas, který je na vytvoření obvodu třeba. Nevýhodou této struktury však je, že se procházejí vodiče, které není třeba navštívit.

Příkladem může být nám již známý obvod na obr. *2.3.4*. Na počátku máme otestovat 10 poruch. Velikost bitového slova je 4. V jednom průchodu obvodem tedy otestujeme 3 poruchy. První průchod obvodem je nutný, aby jsme získali správné hodnoty na všech vodičích. V dalších třech průchodech otestujeme 9 poruch. Nakonec nám zůstane porucha *e* Sa0. Pro její propagaci na výstup stačí průchod jen přes vodiče *e, i, p*. Na místo toho však navštívíme i ostatní vodiče v obvodu. Pokud bychom nemuseli tyto, zde redundantní, vodiče procházet, simulaci bychom urychlili.



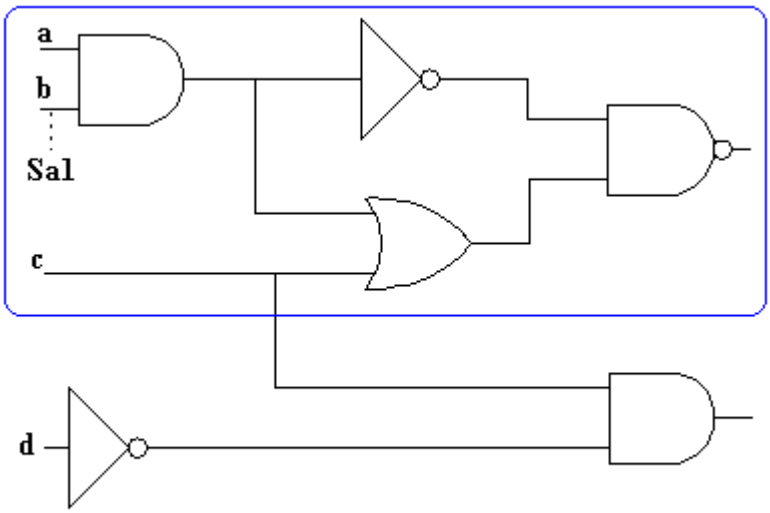
Obrázek 2.3.4: Zbytečně navštívené hradla

Výše zmíněný příklad je trochu extrémní, ovšem vystihuje podstatu problému se zbytečným navštěvováním vodičů. To jsem se pokusil vyřešit redukcí obvodu. Tato redukce však vedla k prodloužení simulace, přesto jí zde uvedu.

2.3.7. Redukce obvodu

Redukcí obvodu dosáhneme požadavku na snížení počtu navštívených stavů, při zachování toho, že obvod vytvoříme jen jednou. Nejprve vybereme libovolnou poruchu v obvodu, nejlépe aby byla co nejblíže vstupních vodičů obvodu (nejlépe porucha na vstupních vodičích obvodu). Vodič, který odpovídá zvolené poruše, přidáme do nově vytvářeného obvodu. Dále pokračujeme přes jeho výstupní vodiče a ukládáme všechny vodiče, které vedou k výstupu. Tímto všechny cesty z dané poruchy vedou k výstupu. Nyní ještě pro všechny vodiče potřebujeme uložit jejich vstupy. Postupně tedy ukládáme vstupní vodiče všech již uložených vodičů, dokud nepostoupíme až ke vstupům celého obvodu. Získaný obvod je podmnožinou původního obvodu, který jsme načetli ze vstupního souboru. V této struktuře můžeme detekovat nejen onu prvotní poruchu, ale i všechny poruchy, které jsou na vodičích ukládaných v první fázi vytváření obvodu (při ukládání směrem k výstupu).

.



Obrázek 2.3.5: Redukce obvodu

Pro ukázku je připraven obr. *2.3.5.*, což je mírně upravený obvod *2.3.4.* pro potřeby tohoto algoritmu. Zobrazená porucha *b Sa1* je porucha, od které se začne redukovaný obvod vytvářet. Výsledek redukce je na obrázku vyznačen modře. V tomto příkladě je patrné, že by se redukce obvodu mohla vyplatit. Při simulaci poruch uvnitř označené oblasti se nemusejí procházet dvě hradla ve spodní části původního obvodu. Pokud však chceme otestovat poruchu na vodiči *c*, získáme redukcí původní obvod, navíc ještě ztratíme čas vytvářením redukovaného obvodu.

Jak bylo již uvedeno v předešlé kapitole, redukce obvodu zrychlení simulace neprospěl, ba naopak. Přesto se však dají některé poznatky použít a mohou pomoci při řešení ukládání struktury obvodu.

2.3.8. Přeskočení úrovní obvodu

Do této chvíle provádíme simulaci tak, že je obvod procházen od vstupních vodičů (úroveň 0) po výstupní. Takto postupujeme, i když je porucha třeba až na výstupním vodiči. Vyřešením tohoto nedostatku lze simulaci určitě hodně urychlit.

Řešení je docela snadné. K parametrům obvodu jsem přidal pole o velikosti počtu úrovní. Každý jeho prvek obsahoval aktuální počet poruch, které jsou třeba v obvodu na dané úrovni odsimulovat. Při simulaci se poté postupuje od úrovně, která jako první obsahuje alespoň jednu poruchu. Díky tomu je simulace opravdu urychlena, hlavně v pozdějších fázích, kdy je v obvodu menší množství poruch.

2.3.9. Závěr - první verze simulátoru

Na první verzi simulátoru jsem pracoval zcela sám, tedy jen za pomoci svých znalostí a zkušeností, které jsem nabyl při jeho implementaci. Dalším nápadem na vylepšení bylo zrychlení založené na využití dominátorů v obvodu (tento nápad se však mírně lišil od principu, který je v práci použit a bude vysvětlen dále). Zde jsme se ale dohodli s mým vedoucím práce, že pro další zrychlování simulace bude třeba nastudovat sofistikovanější techniky pro simulaci. Zde je třeba uvést, že první verze simulátoru byla zhruba 3krát pomalejší, než verze nová.

Jako zdroj inspirací byly použity materiály popisující princip simulace u systému Atalanta, konkrétně simulační programy HOPE a FSIM. Nepokoušel jsem se však kopírovat přesně postupy v těchto programech, ale postupně jsem vylepšoval stávající simulátor zlepšujícími technikami.

Simulátor se postupem času výrazně měnil. Při jeho popisu však nebudu uvádět dílčí změny, jak časem přibývaly, ale popíši konečnou verzi tak, jak je nyní naprogramovaná.

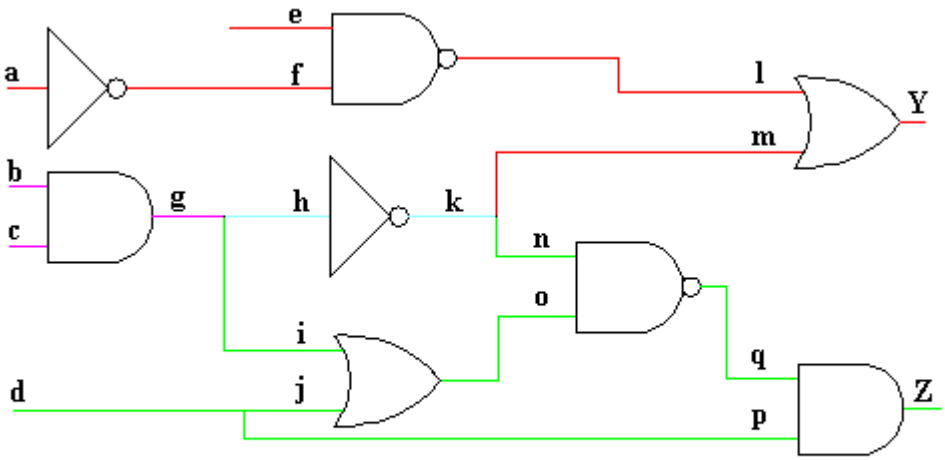
2.4. Implementace paralelní simulace – druhá verze

Nová verze simulátoru řeší hlavní problémy, se kterými jsme se potýkali ve verzi původní. Jde především o zbytečné navštěvování některých stavů a redukce počtu poruch, které bude třeba odsimulovat. Veškeré změny by se daly velikostí přirovnat k programování první verze simulátoru, zde jsem však již mohl využít nejen nově nabyté znalosti, ale i části programové logiky (např. výpočet hodnoty na vodiči ze vstupních vodičů). Proto nová verze byla i méně časově náročnější na implementaci, než verze první.

Nejzásadnější změnou, která vedla k urychlení simulace, bylo použití dominátorů v obvodu. Jejich důležitost je popsána již v úvodu, vše ale bude více specifikováno v dalších kapitolách. Změnou prošlo i uložení struktury obvodu, kdy se již nepoužívá stále stejná struktura, která obsahuje všechna hradla, ale obvod (jeho části) je vytvářen v průběhu simulace.

2.4.1. Identifikace dominátorů

Jen pro připomenutí – vodič A je dominátorem vodiče B, pokud všechny signální cesty z vodiče B, vedou na výstup přes vodič A. Dominátory hrají klíčovou roli v urychlení simulace. Jejich použitím snížíme počet poruch, které je nutné od



Obrázek 2.4.1: Dominátory v obvodu

odsimulovat paralelně. Identifikace dominátorů se provádí po načtení souboru s obvodem, dominátory tedy zůstávají stále stejné po celou dobu simulace.

Detekce se provádí od výstupních vodičů celého obvodu a postupně se postupuje až k vodičům vstupním. Algoritmus zjištění dominátorů lze popsat takto:

1. ulož výstupní vodiče obvodu do fronty
2. vyjmi vodič z fronty a nastav jeho dominátor
3. ulož vstupní vodiče zpracovaného vodiče do fronty

Nastavení dominátoru vodiče:

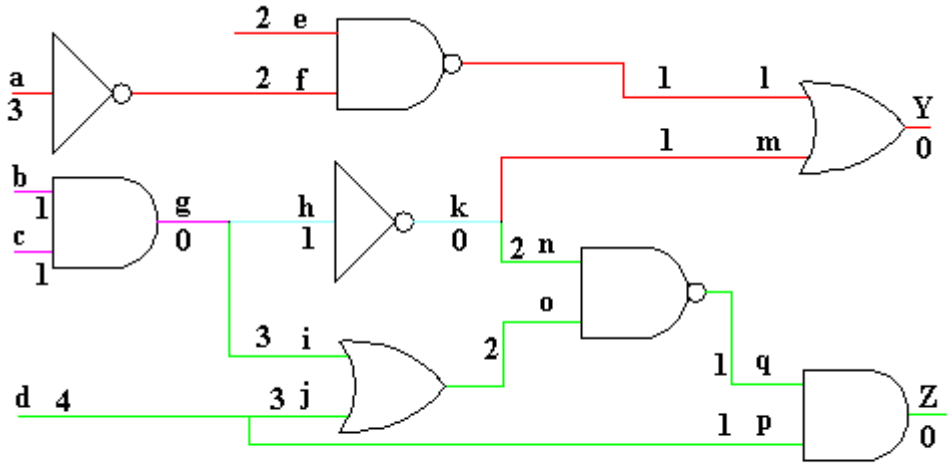
a) dominátorem výstupního vodiče obvodu je sám tento vodič
b) pokud není vodič rozvětven
   1. pokud výstupní vodič hradla, do kterého vodič vede, není rozvětven – dominátorem je dominátor onoho výstupního vodiče hradla
   2. výstupní vodič je rozvětven – dominátorem je výstupní vodič hradla
c) vodič je rozvětven

1. pokud všechny jeho výstupní větve mají stejný dominátor, pak dominátorem daného vodiče je dominátor jeho rozvětvení
2. výstupní větve mají různé dominátory – vodič je dominátorem sám sobě

Pro ilustraci tohoto algoritmu je uveden obrázek *2.4.1*. Výstupní vodiče obvodu (Y a Z) jsou dominátory sami sobě (a). Dále se postupuje po vstupních vodičích hradel těchto výstupů. Začneme například u výstupu Y: vstupy příslušného hradla (l, m) mají dominátor taktéž vodič Y (b 1). U vodiče *l* je a jeho vstupů postupujeme stále stejným způsobem. Zajímavější je vodič *m* a jeho vstup *k*. Vodič *k* je rozvětven a jeho rozvětvení mají různé dominátory. Proto se dominátorem k stává vodič *k* (c 2). Detekce tohoto se provádí jednoduše: při první návštěvě vodiče *k* se u něj uloží, že jeho dominátorem je *m*, při další návštěvě (přes vodič *n*) zjistíme, že stávající dominátor se neshoduje s novým. Situaci, kdy se rozvětvený vodič nestává sám sobě dominátorem (c 1), lze vidět u vodiče *d*. Zbývající možnost (b 2) si lze představit tak, že do obvodu přidáme hradlo před vodič *d*.

Na výše uvedeném obrázku je vidět, že některé vodiče (např. *e*) mají na cestě k dominátoru vodič (pro *e* je jím *l*), kterým procházejí veškeré výstupy vodiče (*e*), avšak tento vodič(*l*) není dominátorem. Dále v práci v ukázce simulace však uvidíme, že i takové vodiče(*l*) jsou využity.

Při detekci dominátorů souběžně detekujeme úroveň vodiče v dominující oblasti. Tu lze vysvětlit tak, že udává počet vodičů, přes které je třeba projít cestou k nejvyššímu dominátoru daného vodiče. Pokud je více cest k nejvyššímu dominátoru, vezme se cesta delší. Nejvyšším dominátorem je vodič, který nemá žádný dominátor.



Obrázek 2.4.2: Úroveň vodičů v dominující oblasti

Na obrázku *2.4.2.* je obvod i s vyznačenými úrovněmi v dominují oblasti.

Tyto úrovně použijeme později při ukládání poruch. Seznam poruch již nebudeme ukládat ve spojovém seznamu, ale pro jejich uložení použijeme dvourozměrnou strukturu. Při detekci poruch se budou postupně testovat poruchy od úrovně 0. Tím zajistíme, že pokud má vodič dominátor s poruchou, bude tato porucha vždy detekována dříve. Dále bude i každý vodič s poruchou, který je blíže dominátoru, otestován dříve. Příklad si uvedeme na obrázku *2.4.2*. a dominující oblast s dominátorem *Z*. Pokud při simulaci zjistíme, že na vodiči *Z* nemůže být detekována porucha, nemá smysl testovat libovolnou poruchu v dominující oblasti. Jiným případem může být, když při simulaci detekujeme poruchu na vodiči *o* na výstupu. Pokud poté testujeme poruchu na vodiči *i* a tato porucha je detekovatelná na vodiči *o*, můžeme tuto poruchu též označit za detekovanou. Pokud porucha na vodiči *o* není detekovatelná na výstupu, není i *i* detekovatelná.

Pokud bychom neměli poruchy uloženy ve podle úrovní v dominující oblasti, mohlo by se na příkladu s vodičem *o* stát, že budeme nejprve detekovat poruchu na vodiči *o*, čímž se připravíme o výhodu uvedenou výše. Přesný popis využívání dominátorů a výhod z vylepšené struktury pro ukládání poruch, bude popsán v dalších kapitolách.

2.4.2. Bezporuchová simulace

V nové verzi simulátoru došlo i ke změně principu získávání správných hodnot na vodičích. Původní verze ukládala bezporuchovou hodnotu na vodiči v prvním bitu registrového slova, kde byla tato hodnota uložena po celou dobu simulace. Správné hodnoty se vypočítávaly přímo při simulaci, konkrétně v prvním průchodu při nastavení nového vstupního vektoru (v dalších průchodech pro stejný vstupný vektor se samozřejmě správná hodnota neměnila).

Tento princip ukládání má mírné nevýhody. Je patrné, že při paralelní simulaci nevyužijeme pro testování poruch onen jeden bit. V ideálním případě by se tedy simulace měla zrychlit o 1/64. Další nevýhodou bylo navštívení všech vodičů v obvodu, tedy i těch, které nebylo třeba nastavovat. Jejich správná hodnota se nemohla změnit, protože se nezměnily ani hodnoty jejich vstupů. Původní struktura však byla výhodnější v tom, že se vytvořila jen jednou a dále se nemusela sestavovat, což se u nové verze musí.

Zásadní změnou ve vylepšené bezporuchové simulaci je použití proměnné speciálně jen pro uchování správné hodnoty. Ta se již nebude ukládat do bitového slova spolu s poruchovými hodnotami. Vzhledem k tomu, že je nyní bezporuchová simulace plně oddělena od simulace s poruchami, je možné provést urychlení. To spočívá v ukončování průchodu obvodem u větví, kde nedošlo ke změně správné hodnoty od posledního průchodu obvodem. Pokud se nezmění hodnoty na vstupu hradla, nemůže se přeci změnit ani hodnota na jeho výstupu, proto nemá smysl u něj nastavovat správnou hodnotu. Obvod se projde celý jen jednou, a to při úplně prvním nastavení správných hodnot, tedy po načtení prvního vstupního vektoru.

Kontrola změny hodnoty na vodiči se provádí již při načítání vstupního vektoru. Pokud nedojde ke změně hodnoty na vodiči, výstupy tohoto vodiče nejsou uloženy pro další zpracování (nastavení hodnoty). V případě, že by po sobě následovaly dva shodné vstupní vektory, nenavštívil by se ani jeden vodič v obvodu (vyjma oněch vstupních). Toto se v praxi nestane a vypadá to i jako nesmyslný příklad, slouží však pro pochopení problému.

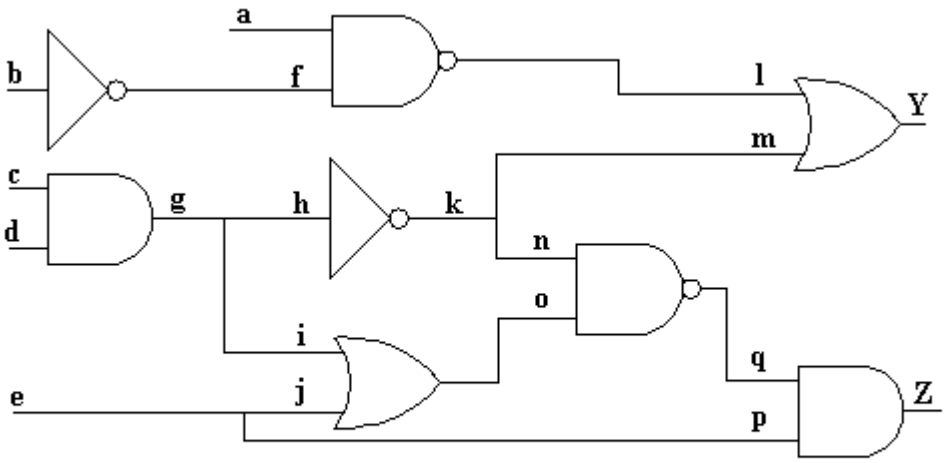
Výše uvedený příklad nás přivede na myšlenku, že čím méně dojde ke změnám hodnot již na vstupních vodičích, tím méně poté navštívíme i vodičů dále v obvodu.

Cílem tedy je, nalézt takovou posloupnost vstupních vektorů, která způsobí na vstupních vodičích obvodu co nejméně změn.

Pro vyzkoušení této hypotézy jsem sestavil jednoduchý algoritmus, který vstupní vektory seřadí podle počtu změn. První vstupní vektor slouží jako řídící vektor, od nějž se budou změny počítat. Po jeho aplikaci na obvod, se ve zbylých vektorech nalezne ten, který má nejméně změn (co nejmenší hammingova vzdálenost). Tento vektor se aplikuje a k němu se hledá opět jiný vektor s co nejmenší hammingovou vzdáleností. Algoritmus je triviální a jeho složitost je velká, slouží však pouze pro ilustraci a také pro vygenerování nového souboru se vstupními vektory – již seřazenými. Tento nový seřazený soubor bude sloužit pro otestování, zda má seřazení vektorů vliv na délku bezporuchové simulace.

Při testovacích měřeních však seřazení vektorů nemělo téměř žádný vliv, ba naopak u některých obvodů došlo i k mírnému prodloužení simulace. Čím je to způsobeno si ukážeme na obrázku *2.4.3*. Mějme první vstupní vektor (00100) pro vstupy (a, b ,c ,d ,e). Pro tento vektor projdeme celý obvod a nastavíme správné počáteční hodnoty na vodičích. V dalším kroku budeme mít na výběr ze dvou vstupních vektorů (00110) a (11000). Ve výše uvedeném algoritmu bychom vybrali první vektor (má jen jednu změnu), ovšem při analýze obvodu zjistíme, že je lepší vybrat druhý vektor, který způsobí změny jen na vodičích *a, b, f*. První vektor však způsobí změny více vodičích – *d, g, h, i, k, o, n, m, Y.*

Tento neduh jsem mírně odstranil tím, že jsem zavedl váhy jednotlivých vstupních vodičů podle toho, kolik má hradlo, do kterého vodič vede, výstupních vodičů. Vycházel jsem z předpokladu, že čím více výstupních vodičů, tím více bude vodičů ovlivněno dále v obvodu. Pro toto vylepšení jsem opět provedl testy. Výsledkem je, že se při nastavování správných hodnot navštíví asi o 5% vodičů méně, což vede i na obdobnou hodnotu u zrychlení.



Obrázek 2.4.3: Srovnání vstupních vektorů

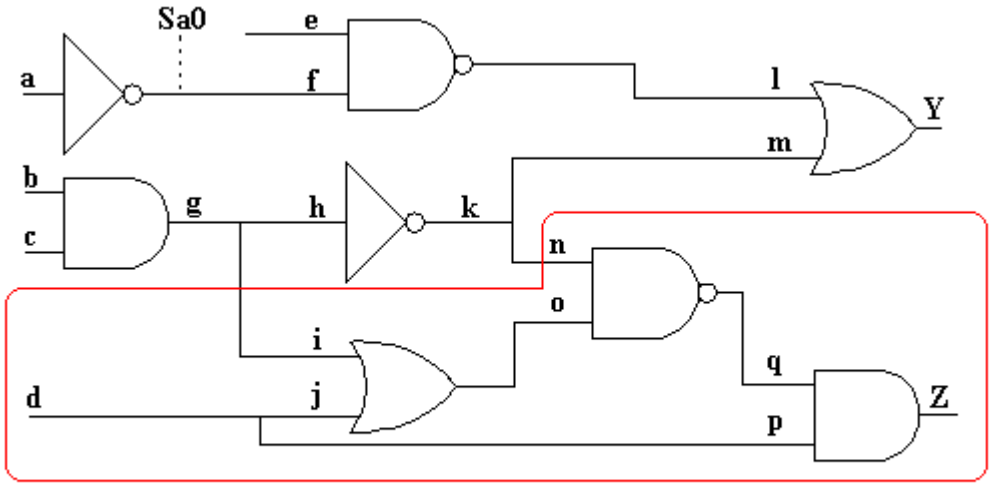
Zjišťování správných hodnot je však jen částí simulačního cyklu, proto se v celkovém čase simulace srovnání vektorů projeví zhruba 1%, dle testovacích měření. Toto vše platí jen v případě, že vstupní vektory načítáme již seřazené, pokud bychom je měli řadit během simulace, tak řazení způsobí prodloužení simulace a je proto nepoužitelné.

Výše uvedená zjištění mě přivedly na myšlenku, že by se řazení vektorů mohlo implementovat přímo do ATPG systémů. Generování vstupních vektorů se provádí pro obvod jen jednou, kdežto simulace se provádí pro velké množství testovaných obvodů, proto by mírné prodloužení při generování vektorů nemělo vadit. Dále by se v ATPG systému mohl implementovat mnohem lepší algoritmus pro nastavování vah jednotlivých vodičů a tím i ovlivňování výběru vektoru – mohlo by se pracovat přímo s počtem vodičů, které jsou na cestě z daného vstupu na výstup, dále by se mohlo využít i zjišťování průchodu změny hodnoty na vstupu obvodem. Myslím, že vylepšením srovnávání vstupních vektorů přímo v ATPG, by se mohlo dosáhnout zrychlení simulace v jednotkách procent.

2.4.3. Paralelní bezporuchová simulace

Při dopisování teoretické části této práce a prvních zkušebních měřeních jsem zjistil, že hlavní část simulace zabere zjištění správných hodnot na vodičích. Tento problém není jen u tohoto simulátoru, ale také u systému Atalanta. Bezporuchová část simulace trvá u obou programů průměrně 75% celkového simulačního času.

Pokoušel jsem se vymyslet algoritmus, kterým by se podařilo nenastavovat hodnoty u vodičů, u kterých to není potřeba. Tedy u vodičů, kterými neprochází pro daný vstupní vektor žádná cesta některé chyby k výstupu obvodu a zároveň tato hodnota nebude potřeba pro některý jiný vodič. Takový případ je ukázán na obrázku *2.4.4*. Pro simulaci označené poruchy není potřeba znát hodnoty na vodičích ve vyznačené oblasti. Tento příklad je však pouze ilustrační a v reálných obvodech se takovéto oblasti nevyskytují, neboť vodiče jsou vzájemně velmi propojené a zároveň je i v obvodech mnoho poruch. Dále je i těžké vůbec předem odhadnout, které vodiče nebude třeba nastavovat.



Obrázek 2.4.5: Zbytečné nastavení správné hodnoty

Nakonec jsem problém s bezporuchovou simulací velice úspěšně vyřešil pomocí paralelní simulace. Tento způsob je velmi jednoduchý, přesto se však v systému Atalanta nevyskytuje, což mě dost překvapuje, neboť i v něm je bezporuchová simulace dominující složkou v celkovém simulačním čase.

Správné hodnoty na vodičích stále počítáme před samotnou simulací poruch. Nyní se ale nepočítá jen jedna správná hodnota jako dosud, ale využije se celé bitové slovo (64 bitů) pro vypočítání správné hodnoty pro 64 vektorů naráz. Při bezporuchové simulace bude znovu použita původní struktura obvodu, která je vytvořená při načtení obvodu a dále se nemění. Tedy vždy procházíme celý obvod, jak je uvedeno v první verzi simulátoru. Takovýto průchod je pro jeden vektor zhruba třikrát pomalejší, než průchod uvedený v kapitole 2.4.2. My však počítáme paralelně 64 vektorů, tedy dojde k teoretickému zrychlení 64/3, oproti bezporuchové simulaci dosud používané. Při měření se tento předpoklad téměř potvrdil – zrychlení je asi 20 krát.

Bezporuchová simulace se nyní tedy provádí jednou na každých 64 vstupních vektorů. Správná hodnota se získává v libovolném okamžiku z bitového slova, podle aktuálního čítače vstupních vektorů. Paměťová náročnost se také příliš nezměnila – jsou použity navíc dvě 64-bitové proměnné, dále však již nemusíme používat původní proměnnou, která uchovávala správnou hodnotu dříve. Za tuto cenu však získáme opravdu znatelné zrychlení, jak bude i dále uvedeno v kapitole naměřených hodnot.

2.4.4. Simulace poruch s dominátorem

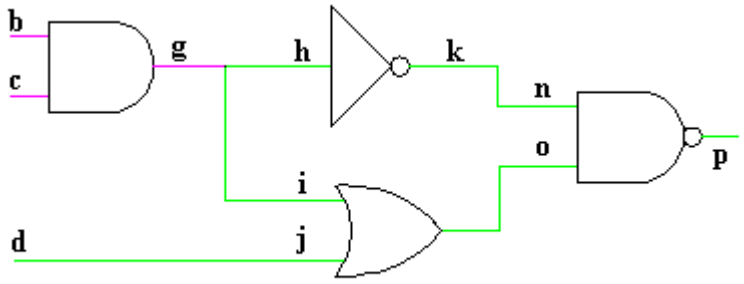
V nové verzi simulátoru došlo i na výraznou změnu v simulaci poruch. Původní verze testovala veškeré poruchy paralelně, nyní se paralelně simulují jen některé poruchy. Při simulaci budou využity informace o dominátorech, které jsme získali již dříve při načtení obvodu.

Pokud má vodič, na kterém je porucha, dominátor, tak již z našich znalostí víme, že všechny cesty jdoucí z vodiče, vedou na výstup obvodu přes dominátor. Tedy pokud má být porucha na takovém vodiči detekovaná na výstupu obvodu, musí být také detekována i na dominátoru.

Při detekci poruch bez dominátoru nejprve provedeme propagaci poruchy k dominátoru. Pokud se porucha nepropaguje až k dominátoru, nemůže být detekována na výstupu obvodu a proto se s ní již dále nepracuje. Když poruchu na dominátoru detekujeme, tak je odpovídající porucha na dominátoru vložena do bitového slova a později simulována paralelně. Při její detekci poté označíme i za detekované všechny poruchy, které byly na dominátor namapovány[2].

Vodiče mohou mít i více dominátorů, jak je vidět na obrázku *2.4.4* u vodičů *b* a *c*. Vodiče *b,c* mají dominátor vodič *g,* ten má však také dominátor –výstup *p*. Pokud tato situace nastane, tak je porucha postupně propagována přes všechny dominátory až

posledním – tedy k vodiči bez dominátoru. Na ten je poté původní porucha namapována. Když je dominátorem výstupní vodič obvodu, je detekce poruchy triviální.



Obrázek 2.4.4: Vodiče s více dominátory

Použití propagace poruch k dominátoru vylepšuje původní simulace tím, že nám umožní brzy detekovat a odstraňovat poruchy s krátkou propagační cestou. Většina poruch, které jsou obtížně detekovatelné na výstupu obvodu, se podaří propagovat jen přes několik vodičů dále v obvodu. Pokud tyto poruchy simulujeme paralelně, tak se nám v pozdějších fázích simulace (kdy jsou v obvodu jen ony obtížně detekovatelné poruchy) stává, že na výstupu detekujeme jen zlomek z původního počtu vložených poruch do bitového slova. Tím tedy ztrácíme výhodu paralelní simulace. Propagací poruch k dominátoru tento nedostatek odstraníme a tím zvýšíme efektivitu paralelní simulace.

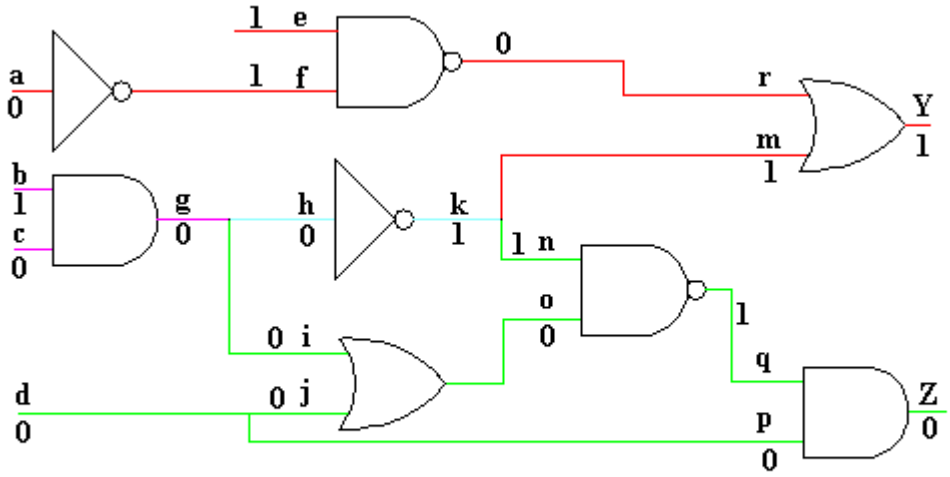
V tabulce *2.4.1.* je pro příklad uvedena redundance testovaných poruch u některých obvodů. Redundancí se zde myslí poruchy, které jsou vloženy do bitového slova a paralelně simulovány, ale nejsou detekovány na výstupu obvodu. Čím nižší redundance, tím vyšší efektivita při vkládání poruch. Tabulka je rozdělena na simulaci při použití a bez použití dominátorů.

	bez dominátorů			*s dominátory*		
	otestováno	nalezeno	redundance	otestováno	nalezeno	redundance
c7552	807 710	7 169	**99,2%**	74 731	2445	**97,8 %**
s9234	199 674	5 977	**97%**	17 833	2091	**88.2%**
s38417	574 186	25 584	**95.5%**	58 430	9608	**83.5%**

Tabulka 2.4.1: Redundance testovaných poruch

Z výsledků je patrné, že při použití dominátorů se efektivita detekce poruch zvýší. Toto zvýšení vypadá celkem malé, je však nutné si uvědomit, že se testuje i mnohem méně poruch, ostatní jsou odfiltrovány již při propagaci. Takže snížíme počet poruch při paralelní simulaci a navíc zvýšíme efektivitu při jejich simulaci. Dále můžeme při bližším rozboru zjistit, že při použití dominátorů jsou poruchy, které paralelně testujeme mnohem blíže k výstupům obvodu – jsou tedy i kratší simulační cesty při průchodu obvodem.

Další výhodou, která není hned patrná je, že většina oblastí s dominátorem, se simuluje pro stejný vstupní vektor pouze jednou. Pokud se dominátory nevyužijí a tedy všechny poruchy se testují paralelně, mohou být tyto oblasti testovány vícekrát - pro různé poruchy. Tento případ, jakož i další výhody dominátorů a jejich princip budou ukázány v následujícím příkladě.



Obrázek 2.4.5: Princip použití dominátorů

Mějme ohodnocený obvod (obr. *2.4.5*), ve kterém je třeba otestovat veškeré poruchy, které má smysl vložit – tedy *a/1, b/0, c/1, d/1, e/0....q/0 a p/1*. Detekce na výstupu je triviální, proto ji nebudeme provádět. Zároveň při simulaci zanedbáme ekvivalenci a dominanci poruch, jde nám jen o příklad.

Nejprve si předvedeme simulaci paralelní bez využití dominátorů. Velikost bitového slova zvolíme 4 (3poruchy + správná hodnota). Obvod procházíme od nulté úrovně – vstupní vodiče. V prvním kroku vložíme do bitového slova poruchy *a/1*, *b/0* a *c/1*. Pro tyto poruchy je třeba projít k výstupu celkem přes 15 vodičů (*a, b, c, f, r, Y, g, h, k, m, i, o, n, l, Z*). V druhém kroku vložíme poruchy *d/1, e/0, f/0* a projdeme

dalších 10 vodičů. Takto budeme postupovat, dokud neotestujeme veškeré poruchy. Celkem pro simulaci těchto 17-ti poruch projdeme 52 vodičů. (25 +ghi10+jkm7+nop5+qr+4). Nalezené poruchy nás v této chvíli nezajímají. Průměrně tedy navštívíme tři vodiče pro nalezení jedné poruchy (ideálně by nám stačilo navštívit průměrně jen jeden vodič). Důvodem pro vysoký počet navštívených vodičů je opakovaná simulace některých částí obvodu. Například při testování poruch na vodičích *a,b,c*, projdeme kvůli poruše na vodiči *a* sekvenci vodičů *a-f-t-Y*. Při detekci poruchy *f/0* však procházíme obdobnou část obvodu *f-t-Y*. Toto se nám podaří při využití dominátorů odstranit.

Nyní si ukážeme simulaci za využití dominátorů. Při vybírání poruch, které se mají testovat, využijeme již zmíněnou dvourozměrnou strukturu pro ukládání poruch. V první kroku otestujeme ty poruchy, které nemají dominátor. V našem obvodu jsou to poruchy *g/1* a *k/0*. Výstupy opět považujeme za triviální. Obě poruchy vložíme do bitového slova o provedeme simulaci. Při ní navštívíme celkem 10 vodičů.
Nyní nás zajímají nalezené poruchy, na rozdíl od "čisté" paralelní simulaci. Vložené poruchy jsou detekovány, s nimi považujeme za detekovatelné i výstupy obvodu (*Y* a *Z*).

Ve druhém kroku jsou simulovány poruchy v dominantní oblasti v úrovni 1. V tomto obvodu jsou jimi poruchy *r/1, m/0, q/0, p/1, h/1, b/0, c/1*. Poruchy budeme simulovat propagací k dominátoru. Před detekcí poruchy nejprve zjistíme, zda byl dominátor detekován – v tomto případě je tomu tak vždy. Pro každou poruchu dále provedeme propagaci a zjistíme, zda je porucha detekována na dominátoru – pro každou tuto poruchu je třeba navštívit 2 hradla (celkem 14 vodičů). U každé poruchy si poznamenáme informace o (ne)nalezení, (ne)propagaci. To se nám bude hodit u testování dalších poruch.

Poruchy ve třetím kroku simulace jsou e/0, f/0, n/0, o/1. Podle předpokladu bychom je měli vždy propagovat až k dominátoru. Zde však využijeme zlepšující techniku. Pokud při propagaci poruchy procházíme jedinou cestou (nedojde k rozvětvení) - narazíme tedy na vodič, kterým procházejí všechny výstupy propagovaného vodiče (ale nejedná se o dominátor) – můžeme propagovanou poruchu označit za (ne)nalezenou a (ne)propagovanou podle toho, jak jsou tyto informace nastaveny u vodiče, kterým právě procházíme. Tato situace nastane u například poruchy *e/0*. Poruchu lze propagovat na vodič *r*, zde zjistíme, že odpovídající porucha *r/1* již byla simulována a nebyla nalezena. Tím simulaci poruchy *e/0* ukončíme a uložíme informace o propagaci a nenalezení. Podobně provedeme simulaci poruch f/0, n/0, o/1. Celkem pro všechny 4 poruchy ve třetím kole navštívíme 8 vodičů.

V dalším kroku odsimulujeme poruchy *i/1,a/1* a *j/1*. Opět postupujeme jako ve třetím kroku – návštěva 6-ti vodičů. Nakonec otestujeme poruchu *d/1*. Pro ní nelze

použít žádnou dřívější detekci a je nutné jí propagovat až k výstupu *Z* (projdeme 6 vodičů).

Celkově při použití propagace navštívíme 44 vodičů. Vidíme, že se podařilo snížit počet navštívených hradel a tím i délku simulace, i když jsme ne vždy použili pro detekci poruch paralelní simulaci. Je to dáno tím, že většina oblastí s dominátorem se simuluje jen jednou pro jeden vstupní vektor. Navíc v našem příkladě byl dominátor vždy detekován. Pokud by však vodič *Z* nebyl detekován (nebyl by výstupem obvodu), nemuseli bychom testovat žádnou poruchu v dominující oblasti (6 poruch). Takovéto případy se zvláště v pozdějších fázích simulace vyskytují často, proto je použití dominátorů velmi výhodné.

V příkladě jsme si ukázali postup simulace při prvním vstupním vektoru. Po nastavení dalších vstupních vektorů nastává situace, kdy dominátor byl již detekován pro dřívější vstupní vektor, a proto není pro aktuální vstupní vektor otestován. V takovém případě se provede propagace až k poslednímu dominátoru daného vodiče. Pokud se propagace provede až k dominátoru, je odpovídající porucha na dominátoru vložena a odsimulována paralelně, po jejím otestování se výsledek zaznamená u původní testované poruchy.

Dalším vylepšení propagace získáme ukládáním propagační cesty. Již několikrát byla v práci zmíněna nevýhoda opakovaného vytváření struktur obvodu. Při použití propagace k dominátoru si lze všimnout, že daná porucha používá stále stejnou cestu k dominátoru. Toho využijeme a při první propagaci poruchy uložíme i cestu k dominátoru. Pokud je později porucha propagována znovu, využije již vytvořenou cestu a díky tomu simulaci urychlíme.

2.4.5. Simulace poruch bez dominátoru

Poruchy bez dominátoru jsou vždy simulovány paralelně v bitovém slově. Pokud je tato porucha na vodiči, který je výstupem obvodu, je simulace triviální. Možným vylepšením by mohlo být testování, zda je porucha detekována alespoň na jednom z výstupů daného vodiče (příslušného hradla). Pokud je na některém z výstupů detekována, je odsimulována paralelně, v opačném případě není[2].

Pro tento předpoklad jsem provedl několik testů. Při tomto testování se podaří průměrně odfiltrovat zhruba 15% poruch, které nemusejí být do paralelního slova vloženy pro daný vstupní vektor. Tím u větších obvodů (s38417.bench) ušetříme řádově stovky průchodů paralelní simulace. Ovšem samotná detekce je časově náročnější než zisk, který nám detekce umožní. Proto se tato detekce nevyplatí a není v simulátoru zapracována. Navíc při simulaci poruch je po vyhodnocení výstupu hradla

otestováno, zda je v daném slově možné detekovat nějakou poruchu (všechny bity nejsou stejné). Tímto se problém taktéž řeší.

3. Popis implementace

3.1. Vývojový diagram algoritmu

Na obrázku *3.1.1.* je znázorněn zjednodušený popis funkce algoritmu. I tento popis však plně vystihuje princip simulace obvodů. Simulační krok začíná nastavením hodnot vstupních vodičů obvodu – tedy aplikace vstupního vektoru na obvod. Dále je nutné zjistit správné hodnoty na všech vodičích. Poslední fází je již samotná simulace a s ní spojená detekce poruch.

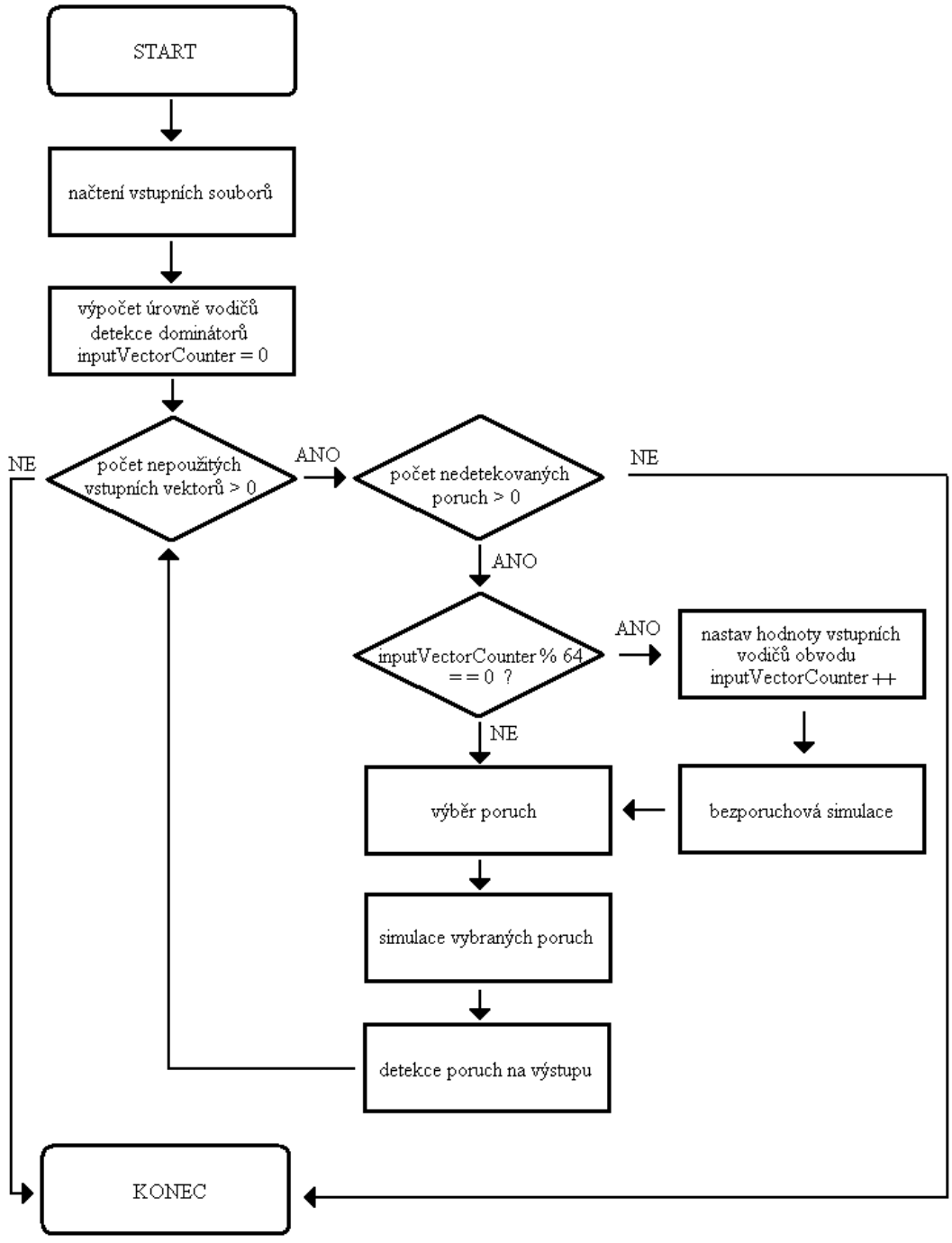
3.2. Použité struktury

V následujících podkapitolách budou uvedeny a blíže popsány datové struktury, které jsou nejčastěji v programu použity. Veškeré struktury se týkají druhé verze simulátoru. Některé jsou popsány již v kap.*2.2.*, proto zde budou zmíněné jen okrajově.

3.2.1. Třída *CWire*

Třída *Cwire* definuje základní stavební prvek obvodu – signální vodič. Struktura obvodu se skládá jen z těchto vodičů, nejsou tedy použity zvláštní prvky pro hradla. Hradlo je definováno již samotným typem vodiče. Třída obsahuje proměnné

pro uložení hodnot na vodičích, odkazy na poruchy, seznamy vstupních a výstupních vodičů. Toto vše nám postačí na vytvoření komplexní struktury obvodu. Všechny důležité proměnné a metody této třídy jsou popsány v tabulce *3.2.1*., respektive tab. *3.2.2*.



Obrázek 3.1.1: Vývojový diagram algoritmu

datový typ	***název***	***popis***
long long int	*rightValueParallel, dontCareValueParallel*	obsahují správné hodnoty na vodiči pro více vstupních vektorů
long long int	*ParallelValue, parallelDontCare*	obsahují aktuální hodnoty na vodiči pro více vložených poruch
int	*function*	určuje typ hradla, jehož je vodič výstupem
int	*Inputs, outputs*	počet vstupů a výstupů příslušného hradla
*char**	*name*	název vodiče
int	*visited*	uchovává hodnotu, kdy byl vodič nastaven – důležité pro zjištění, kdy máme na vodiči nastavit správnou hodnotu
int	*level*	úroveň vodiče v obvodu
int	*dominantLevel*	úroveň vodiče v oblasti s dominátorem
*SFaultList**	*faultListIndex0, faultListIndex1*	odkazy na poruchu na vodiči
*CWiresQueue**	*inputWires, outputWires*	odkazy na vstupní a výstupní vodiče daného hradla

Tabulka 3.2.1: Proměnné třídy CWire

návratová hodnota	***název***	***popis***
*long long int**	*SetParallelValueTrue*	nastavení správných hodnot při bezporuchové simulaci
*long long int**	*SetParallelValue*	nastavení hodnot na vodičích při simulaci poruch
*CWire**	*isFaultPropagateThruDominators*	zjištění, zda lze poruchu propagovat přes dominátor(y)
*CWire**	*getDominantWire*	vrátí dominantní vodič pro tento vodič

Tabulka 3.2.2: Metody třídy Cwire

Metody *SetParallelValueTrue* a *SetParallelValue* pracují na stejném principu, každá však používá své vlastní proměnné. Oddělení nastavování správných hodnot a hodnot při simulaci bylo již od počátku práce, až v poslední fázi byl změněn princip bezporuchové simulace na paralelní, tedy stejný princip jako u simulace poruch. Ikdyž by tedy šlo obě metody spojit v jednu, není to podle mě výhodné, vzhledem k možnému budoucímu pokračování na programu.

Nastavování správných hodnot se provádí aplikací bitových operací na hodnoty

vstupních vodičů hradla. V těchto metodách se uplatňuje proměnná *visited*, která je uvedená výše. Před aplikací bitové operace se nejprve u vstupního vodiče hradla zjistí, zda byl vodič již v daném simulačním cyklu nastavován. Pokud nebyl, nastaví se všechny bity příslušné proměnné podle správné hodnoty na daném vodiči. Vodič mohl být navštíven a nastaven již dříve – mohl být výstupem jiného hradla a tedy v daném simulačním cyklu uchovává již nějakou hodnotu simulace. V takovém případě nesmíme tuto hodnotu přepsat. Je třeba ještě uvést, že v každém simulačním cyklu je vodič vždy správně nastaven alespoň jednou.

Při simulaci je velmi důležitá metoda *isFaultPropagateThruDominators*. Ta se používá u poruch na vodičích, která jsou v oblasti s dominátorem. Jejím cílem je zjisit, zda se porucha propaguje až k nejvyššímu dominátoru (*kap. 2.4.1.*). Pokud ano, je návratovou hodnotou tento dominátor, v opačném případě hodnota *NULL*. Metoda pracuje na stejném principu jako paralelní simulace, zde je však vložena jen jedna porucha. Jde tedy o simulátor na malé části obvodu. Propagace k nejvyššímu dominátoru se provádí rekurzivně přes dílčí dominátory (pokud jsou).

Při propagaci poruchy jsou použity techniky pro její urychlení. Jelikož většinou propagujeme v oblastech bez rozvětvení, je v každém kroku propagace (pokud nejsme v oblasti s rozvětením vodiče) test, zda porucha přes hradlo přešla dále v obvodu. Pokud jsme v této oblasti bez rozvětvení, zaznamenáváme si do fronty navštívené vodiče. Na konci propagace pak i u nich můžeme nastavit výsledek propagace, který odpovídá propagaci původní poruchy. Posledním vylepšením je zde uchovávání propagační cesty pro případné použití v dalších cyklech simulace. Z tohoto důvodu je metoda *isFaultPropagateThruDominators* rozdělena na dvě části. První z nich se použije jen poprvé (v ní uchováváme onu propagační cestu), v dalších cyklech používáme druhou část, kde již postupujeme po dříve vytvořené cestě.

Poslední zde uvedenou metodou je *getDominantWire*, která se využívá při načítání souboru se strukturou obvodu. Jejím výsledek je dominatní vodič pro vodič, který metodu zavolá. Algoritmus získání dominantního vodiče je již uveden v kapitole *2.4.1.*, proto zde nebude uveden.

3.2.2. Třída *CCircuit*

Tato třída "zastřešuje" celý simulátor. Vedle důležitých a dále uvedených datových struktur a metod obsahuje několik datových struktur, které jsou průběžně využívány i dílčími prvky obvodu. Jedná se tedy o globální proměnné, které se vytvoří jen jednou při vytvoření objektu pomocí této třídy. Jsou to hlavně datové kontejnery z

knihovny STL. Časová náročnost jejich vytvoření se může zdát zanedbatelná, ovšem zde by se při simulaci opakovaně vytvářely a ničily řádově i v milionech , což je již znát na délce simulace. Jsou použity například při propagaci poruch, pro ukládání dvourozměrné struktury obvodu atd.

datový typ	***název***	***popis***
long long int	step	určuje pořadí simulačního kroku
long long int	*inputCounter*	pořadí vstupního vektoru
int	*maxLevel*	nejvyšší úroveň hradla v obvodu
int	*inputs, outputs*	počet vstupních a výstupních vodičů obvodu
*CWiresQueue**	*inputWires, outputWires*	odkazy na vstupní/výstupní vodiče obvodu
int	*faults*	počet poruch k otestování
vector<SVectorsList>*	*inputVectorsVector*	struktura pro uložení vstupních vektorů
vector<vector<vector< SFaultList> > >*	*dominantFaultListVectors*	struktura pro uložení poruch podle úrovní vodiče v dominantní oblasti

Tabulka 3.2.3: Proměnné třídy CCircuit

návratová hodnota	***název***	***popis***
void	*ParallelFaultSimulate*	základní metoda, která obsluhuje celou simulaci
void	*SetInputVector*	nastavení vstupního vektoru
void	*setTrueValues*	bezporuchová simulace obvodu
*SFaultList**	*getFault*	vybere poruchu pro simulaci
int	*faultPropagateThruOutputs*	zjistí, zda lze poruchu detekovat na výstupu hradla, na jehož vstup je porucha vložena

Tabulka 3.2.4: Metody třídy CCircuit

Ve třídě *ParallelFaultSimulate* je řízena celá simulace – od nastavování vstupních vektorů, přes bezporuchovou simulaci (zjištění správných hodnot na vodičích), výběr poruch pro simulaci až po simulaci poruch a jejich detekci. Zde se

provádějí různé testy, které zjistí, zda má smysl vybranou poruchu simulovat. Základním předpokladem pro simulaci poruchy je, že správná hodnota na vodiči se liší od hodnoty poruchy a zároveň, že hodnota na vodiči není X. Dále se zjistí, zda je možné poruchu propagovat k dominátoru (pokud je) a pokud ano, tak se porucha na dominátoru odsimuluje paralelně. Pokud porucha na dominátoru není (nebyla ve vstupním seznamu poruch), vytvoří se, ale po jejím odsimulování a případném detekování na výstupu, se neuvede ve výsledku jako nalezená. Po vložení poruch se v této metodě provede i poruchová simulace a následně i detekce poruch. Nedetekované poruchy se vloží zpátky do seznamu poruch.

Hodnoty na vstupních vodičích nastavujeme v metodě *SetInputVector*. Toto nastavení se provádí jen jednou, za každých 64 simulačních cyklů, neboť v bezporuchové simulaci odsimulujeme naráz 64 vstupních vektorů. Hodnoty vstupních vektorů jsou uloženy v poli proměnných, kde každá 64-bitová proměnná uchovává 32 až vstupních hodnot. Hodnoty jsou uloženy ve dvou bitech – 00 pro log. 0, 01 pro log. 1 a 11 pro hodnotu X.

Správné hodnoty na vodičích získáme díky metodě *setTrueValues*. Ta provede bezporuchovou simulaci obvodu. Využívá strukturu vytvořenou při načtení obvodu ze souboru – projde tedy všechny vodiče. Stejně jako nastavení vstupního vektoru se provádí jen jednou za 64 simulačních cyklů.

Metoda *getFault* jen vrátí poruchu, která bude odsimulována. Výběr provádí z dvourozměrné struktury *dominantFaultListVectors*. Poslední uvedenou metodou je *faultPropagateThruOutputs*. Ta zjistí, zda se porucha propaguje alespoň přes jeden výstup hradla, na jehož vstup je porucha vložena. Tato metoda sice není v simulátoru použita, ale je možné její použití v budoucnu (či její mírná modifikace), proto jí zde uvádím.

3.2.3. Ostatní datové struktury

Struktura *SfaultList* se používá pro uložení poruchy. Obsahuje odkaz na příslušný vodič, typ poruchy a informace o tom, v jakém simulačním cyklu byla (ne)detekována.

Nakonec ještě uvedu často používanou třídu *CWiresQueue*. Jedná se o spojový seznam vodičů. Obsahuje metody pro vložení / odstanění prvku, výpis...

4. Testovací měření

V této kapitole budou hlavně uvedeny srovnávací měření rychlosti paralelní simulace se systémem Atalanta (algoritmus HOPE). Tyto výsledky však budou doplněny o další různá měření a statistiky, které více objasní některé problémy simulace. Těmito výsledky, a jejich patřičným zdůvodněním, bych chtěl i nastínit další možná budoucí pokračování na tomto simulátoru. Měření byla prováděna v operačním systému Windows XP s procesorem AMD Athlon XP 2600+ (2,1GHz), 512MB RAM.

4.1. Rychlost simulace

Zde bude uvedeno srovnání tohoto simulátoru s algoritmem HOPE a dále porovnání s první verzí simulátoru. Nejprve bude provedena simulace na třech různých obvodech, na kterých bude ukázáno "hrubé" porovnání jednotlivých algoritmů. Pro podrobnější srovnávání bude použit největší obvod *s38417.bench*. Pokud nebude uvedeno jinak, bude vždy při simulaci použit neupravovaný fault list vygenerovaný systémem Atalanta. Při měřeních je zanedbáno načítání vstupních dat (pro naše měření se jedná řádově o desetiny sekundy).

4.1.1. Základní srovnání obou verzí simulátoru s programem HOPE

Na počátku měření jsem provedl porovnání obou verzí simulátoru uvedených v této práci a systému Atalanta (algoritmus HOPE). Testovací vektory jsem vygeneroval náhodně, to mělo za následek mírné znevýhodnění mých algoritmů. Důvod bude vysvětlen dále, kdy bude uvedeno i podrobnější měření. Měření bylo provedeno pro sady 1000, 5000 a 10000 vstupních vektorů (bez hodnot X) s uvedeným počtem poruch k otestování.

Prvotní srovnání rychlostí slouží především pro ilustraci vylepšení druhé verze simulátoru od první. Z naměřených dat lze odvodit, že vylepšená verze mého simulátoru je zhruba 5-krát rychlejší než verze první. V tomto měření jsou zároveň program HOPE a můj algoritmus v rychlosti srovnatelné, záleží na použitém benchmarku. Dále však bude provedeno podrobné porovnávání obou simulátorů, kde získáme přesnější a výstižnější výsledky.

		1 000	5 000	10 000	počet poruch
první verze	**c7552**	7,2s	32,5s	63,7s	7 500
	s9234	13,2s	51,4s	94,4s	6927
	s35417	59,5s	268,1s	524,3s	31180
vylepšený simulátor	**c7552**	1,3s	4,0s	6,7s	7 500
	s9234	5,1s	18,0s	30,1s	6927
	s35417	11,5s	47,3s	90,0s	31180
HOPE	**c7552**	1,2s	5,6s	11,3s	7 500
	s9234	2,5s	10,9s	20,8s	6927
	s35417	9,3s	44,8s	90,2s	31180

Tabulka 4.1.1: Porovnání rychlosti dvou verzí simulátoru s programem HOPE

4.1.2. Porovnání vylepšeného simulátoru s programem HOPE - I

Od teď se budeme zabývat porovnáváním jen mého vylepšeného simulátoru a algoritmu HOPE. Pro testování byl zvolen největší obvod *s38417*. Jak již jeho název napovídá, obsahuje tento obvod 38417 vodičů – celkem 8709 hradel a 13 470 invertorů. Pro simulaci budou použity vstupní vektory vygenerované systémem Atalanta bez hodnot don't care (X).

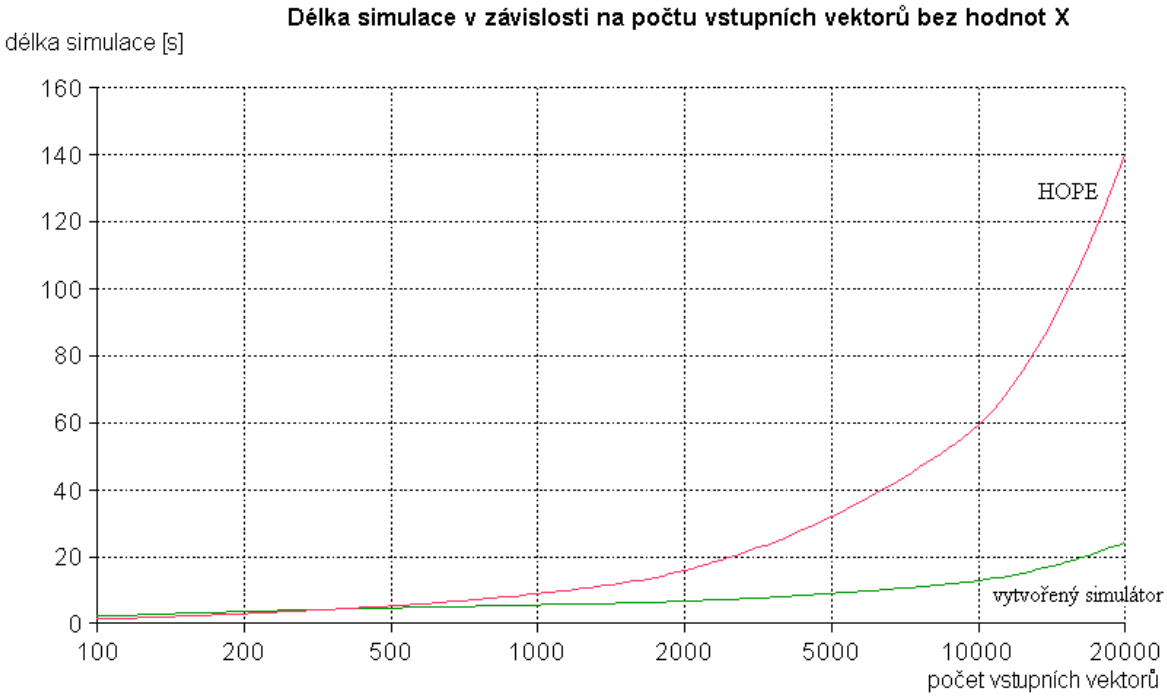
Před samotnými výsledky ještě trochu rozvedu výše zmíněný nedostatek při náhodném generování vstupních vektorů. Každý vstupní vektor vygenerovaný systémem Atalanta pokrývá (detekuje) alespoň jednu poruchu v obvodu (za předpokladu že použijeme i seznam poruch vygenerovaný Atalantou). Tedy žádný tento vektor není zbytečný. Hlavně však jsou v této sadě vstupní vektory, které umožňují nalézt obtížně detekovatelné poruchy.

Pokud ale používáme náhodné vstupní vektory, budou takto vygenerované sady velmi pravděpodobně obsahovat vektory, které nedetekují žádnou poruchu a jsou tedy zbytečné. Program HOPE zřejmě obsahuje techniky (v dokumentaci o tom ale není zmínka), které umožní brzkou detekci toho, že pro zvolený vstupní vektor a aktuálně vložené poruchy do obvodu, nelze detekovat žádnou poruchu na výstupu obvodu. Zřejmě proto dosahuje algoritmus HOPE v porovnání s tímto algoritmem lepší výsledky pro náhodně generované vstupní vektory (viz tab. *4.1.1.*). Jak je tomu pro vstupní vektory získané systémem Atalanta bude uvedeno dále.

V tabulce *4.1.2* je uvedeno porovnání rychlostí tohoto simulátoru s programem HOPE pro různý počet vstupních vektorů (100, 200, 500 ...). Vektory jsou bez hodnot X. Testovacím obvodem je *s38417*.

	100	**200**	**500**	**1 000**	**2 000**	**5 000**	**10 000**	**20 000**
vytvořený simulátor	2,2s	3,1s	4,5s	5,5s	6,6s	9,5s	14,5s	24,3s
HOPE	1,3s	2,1s	4,5s	8,1s	15,1s	35,9s	70,7s	140,2s
poměr zrychlení	0,59	0,67	1	1.47	2.78	3,77	4,84	5,76

Tabulka 4.1.2: Porovnání rychlosti simulátoru s programem HOPE – vektory generované systémem Atalanta bez hodnot X

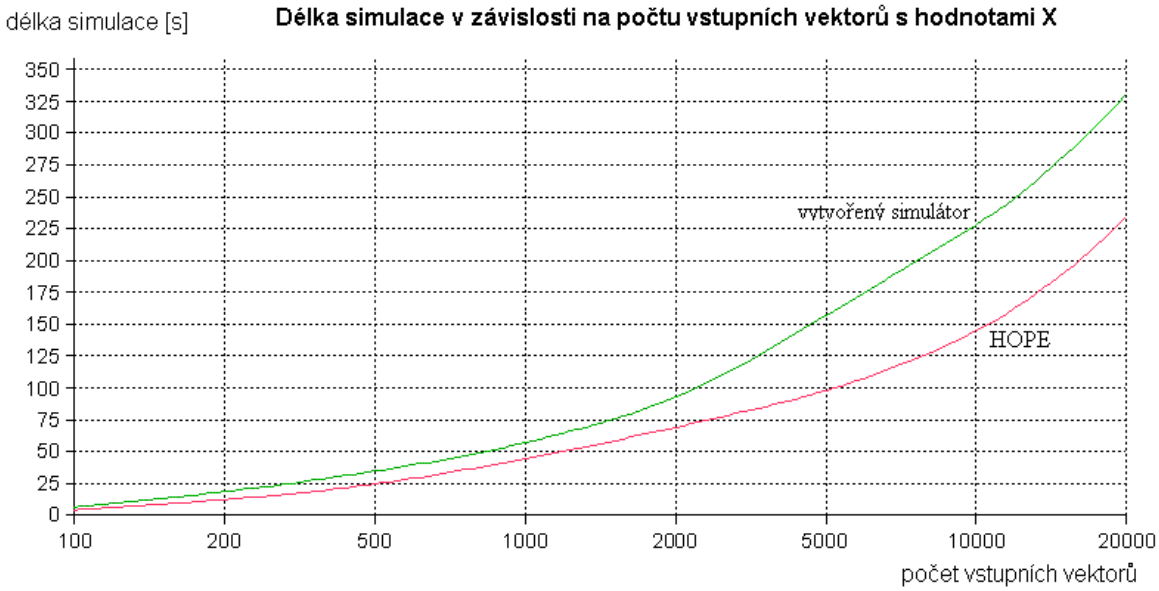


Graf 4.1.1: Porovnání rychlosti simulátoru s programem HOPE

Z výsledků měření je patrné postupné zvyšující se zrychlení mého algoritmu se zvyšujícím se počtem vstupních vektorů. Hlavním jeho výhodou oproti HOPE je využití paralelismu i pro zjišťování správných hodnot na vodičích (bezporuchová simulace). Základem je však velmi dobrý algoritmus samotné simulace poruch. Dále jsou uvedena data pro vektory obsahující hodnoty don't care.

	100	200	500	1 000	2 000	5 000	10 000	20 000
vytvořený simulátor	6,3s	12,4s	29,3s	52,8s	91,8s	174,8s	256,5s	330,3s
HOPE	4,1s	8,3s	19,4s	41,5s	70,8s	103,5s	166,4s	234,0s
zrychlení	0,65	0,66	0,66	0.78	0.77	0,59	0,64	0,70

Tabulka 4.1.3: Porovnání rychlosti simulátoru s programem HOPE – vektory generované systémem Atalanta s hodnotami X



Graf 4.1.2: Porovnání rychlosti simulátoru s programem HOPE

V tabulce *4.1.3.* jsou uvedeny hodnoty měření při použití vstupních vektorů s hodnotami X. Zde již ztrácíme výhodu paralelismu při bezporuchové simulaci. Výsledky jsou tedy mírně příznivé ve prospěch algoritmu HOPE. I přesto jsou však výsledky povzbudivé, zvláště pokud víme, že je zde možnost urychlení (viz dále)

Jak již bylo uvedeno v kapitole *2.4.3.*, bezporuchová simulace zabírá větší část z celkového simulačního cyklu. Zvláště při použití vstupních vektorů bez hodnot X. Zde jsme provedli vylepšení díky paralelismu. Bezporuchovou simulaci provádíme jen jednou za 64 původních bezporuchových simulačních cyklů. V tomto vylepšení však vždy procházíme celý obvod. V původní verzi bezporuchové simulace byly navštíveny jen ty vodiče, u nichž došlo ke změně hodnoty od posledního průchodu bezporuchové simulace. Ve většině případů jsme tedy navštívili jen část obvodu. Při použití hodnot X se ale navštíví ještě mnohem méně vodičů, hlavně u větších obvodů. U těchto obvodů dojde ke změně na vstupních vodičích jen u zlomku vodičů (jednotky procent) a tyto změny se tedy projeví i na velmi malé části obvodu. Proto u vstupních vektorů s hodnotami X nemáme takové zrychlení, jako u vektorů jen s hodnotami log. 0, 1.

Řešením bezporuchové simulace u vektorů s hodnotami X by mohl být návrat k předchozí verzi bezporuchové simulace. Do této verze by ale mohl být také implementován paralelismus. Nepoužilo by se však všech 64 bitů (jako nyní), ale jen například 4 bity. Tím bychom získali výhodu paralelní simulace a zároveň bychom nemuseli navštěvovat vždy celý obvod. Navíc by počet bitů nemusel být dán pevně, ale mohl by se dynamicky měnit.

V této kapitole jsme se zabývali porovnáváním rychlosti pro vstupní vektory generované systémem Atalanta. Takové měření má vypovídací hodnotu jen pro velký obvod, pro malé obvody máme k dispozici jen malý počet vstupních vektorů, běh simulace je tedy velmi krátký, a proto nemůžeme takovéto výsledky považovat za validní.

Jak již bylo dříve uvedeno, pro náhodně generované vektory obdržíme mírně odlišné výsledky. Takovéto vektory se také velmi často používají. Proto v následující kapitole budeme provádět srovnání simulátorů pro náhodně generované vektory. Těchto vektorů je již velmi mnoho, můžeme tak provádět srovnání pro více obvodů.

4.1.3. Porovnání vylepšeného simulátoru s programem HOPE - II

V předchozí kapitole jsme si ukázali srovnání programů pro jeden obvod a vstupní vektory vygenerované Atalantou. Nyní si ukážeme poměry rychlostí pro náhodně generované vektory, bude také použito více obvodů[6], [7]. Vektory budou vždy bez hodnot X. Výběr použitých obvodů není náhodný, ale jsou zvoleny ty obvody, které ze statistik[5] potřebují velké množství vstupních vektorů pro detekci všech poruch.

Výsledky měření jsou spolu s poměry rychlostí obou algoritmů uvedeny v tabulkách *4.1.4.* a *4.1.5*. Poměr zrychlení je poměr rychlosti algoritmu HOPE a tohoto algoritmu. Tedy pro čísla menší než 1 je rychlejší HOPE, pro čísla větší než 1 je rychlejší tento algoritmus.

Obecně lze z hodnot vyčíst, že HOPE je rychlejší pro menší obvody. U větších obvodů se oba algoritmy vyrovnávají, leckde je vytvořený simulátor v této práci rychlejší. Proč tomu tak je, nelze přesně zdůvodnit, jelikož neznáme detaily implementace algoritmu HOPE.

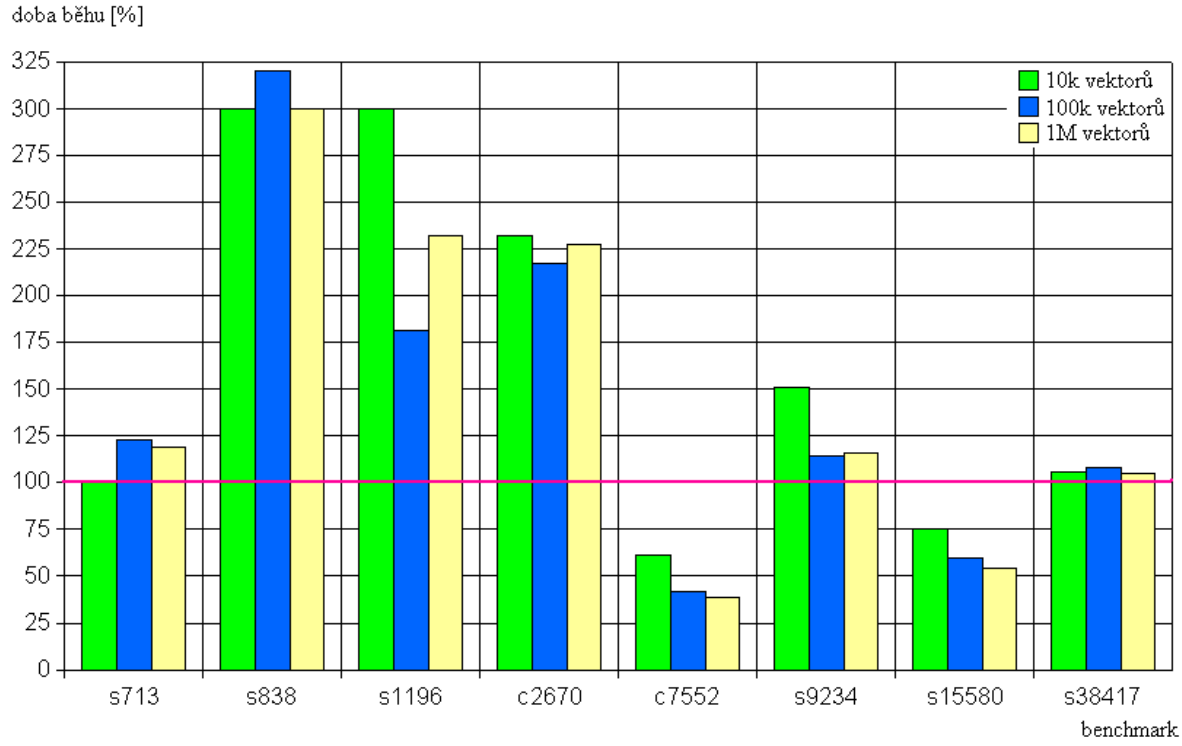
	10 000		100 000		1 000 000	
	vytvořený simulátor	HOPE	vytvořený simulátor	HOPE	vytvořený simulátor	HOPE
s713	0,2s	0,2s	2,1s	1,7s	22,6s	19,1s
s838	0,6s	0,2s	5,8s	1,8s	57,8s	19,1s
s1196	0,6s	0,2s	4,3s	2,4s	61,6s	26,7s
c2670	4,1s	1,8s	37,6s	17,3s	387,7s	172,3s
c7552	6,5s	10,7s	43,7s	102,7s	397,3s	1 020,6s
s9234	29,8s	19,8s	194,5s	170,0s	1 943s	1 668s
s15580	24,8s	32,9s	181,8s	300,6s	1 721s	3 136s
s38417	95,1s	90,0s	919,4s	852,6s	9 075s	8 636s

Tabulka 4.1.4: Porovnání rychlosti simulátoru s programem HOPE – náhodně generované vektory

	10 000	100 000	1 000 000
s713	1	0,81	0,84
s838	0,33	0,31	0,33
s1196	0,33	0,55	0,43
c2670	0,43	0,46	0,44
c7552	1,64	2,35	2,56
s9234	0,66	0,87	0,86
s15580	1,32	1,65	1,82
s38417	0,94	0,92	0,95

Tabulka 4.1.5: Poměr zrychlení algoritmů – data z tab. *4.1.4.*

Grafické vyjádření naměřených výsledků je v grafu *4.1.3*. Na tomto grafu je znázorněna doba běhu tohoto algoritmu vzhledem k programu HOPE pro různé obvody a různý počet vstupních vektorů.



Graf 4.1.3: Porovnání rychlosti simulátoru s programem HOPE

4.1.4. Znázornění vylepšování algoritmu

V této podkapitole uvedu, jak postupně docházelo k zrychlování algoritmu. Výchozím bodem je paralelní simulace bez jakýchkoliv zlepšujících technik. Testovacím obvodem bude s38417 a 10000 vstupních vektorů bez hodnot X.

vylepšení	**délka simulace [s]**	**poměr zrychlení**
bez vylepšení	164,7	1
použití dominátorů	101,9	1,61
vylepšení použití dominátorů	99,6	1,65
vylepšení v propagaci	97,1	1,69
testování při simulaci	91,5	1,8
paralelní bezporuchová simulace	25,1	6,5

Tabulka 4.1.6: Postupné vylepšování algoritmu

Na vylepšení algoritmu má největší vliv paralelní bezporuchová simulace. Ta se na celkovém zrychlení podílí zhruba polovinou. Tento výsledek však není překvapivý. Jak již bylo několikrát zmíněno, bezporuchová simulace se na celkovém simulačním čase podílí kolem 75 procenty. Dále je důležité i použití dominátorů. Jejich vliv na simulační čas uvedu dále.

V tabulce *4.1.4.* uvedené vylepšené použití dominátorů znamená to, že využíváme informace získané během simulace. Tedy například pokud propagujeme poruchu A k dominátoru D a ten byl již pro daný vstupní vektor nalezen, rovnou můžeme poruchu A označit za nalezenou a nemusíme ji paralelně simulovat. Vylepšení v propagaci je popsáno v kap. *2.4.4*. na konci. Poslední v tabulce uvedené testování při simulaci vyjadřuje zjišťování, zda při simulaci poruch může být v bitovém slově detekována porucha. Pokud během průchodu obvodem dojde k situaci, že na vodiči (vstupní vodič hradla) nemůže být detekována porucha, nemusíme navštívit výstupní vodiče hradla, protože na nich také nemůžeme poruchu detekovat. Tím snižujeme počet navštívených vodičů při paralelní simulaci.

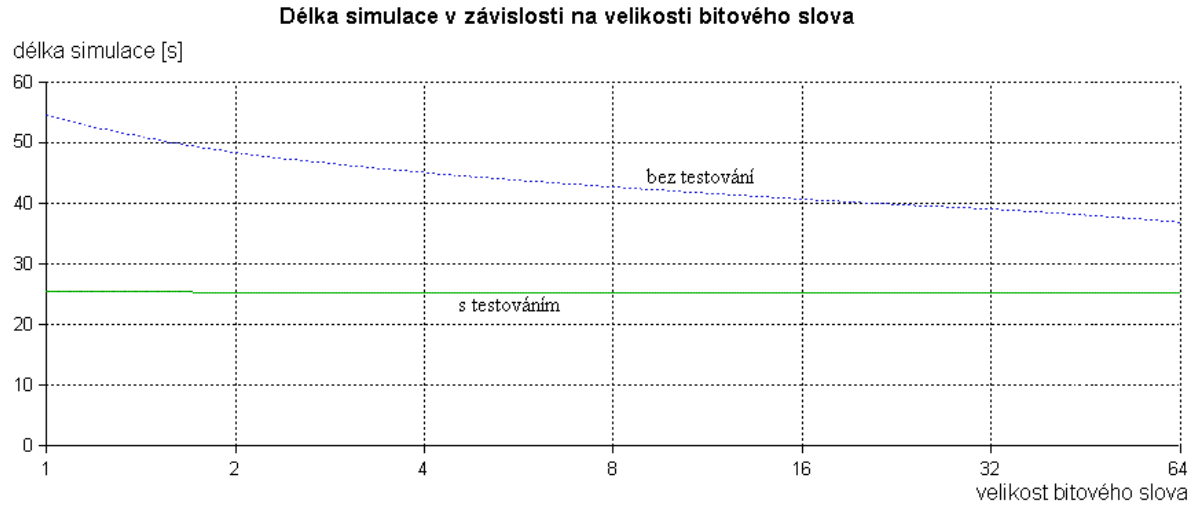
4.1.5. Proměnlivá velikost bitového slova

Při simulaci poruch využíváme vždy bitové slovo o velikosti 64 bitů. Můžeme tedy najednou otestovat až 64 poruch. Jak je rychlost simulace závislá na tomto počtu si uvedeme v této podkapitole.

Testovacím obvodem byl zvolen *s38417*, počet vstupních vektorů je 10000. V simulátoru se měnil jen počet bitů při paralelní simulaci a také testování při simulaci. Tímto testováním je opět míněno zjišťování detekce libovolné poruchy při navštívení hradla (viz *4.1.3. dole*). Počtem bitů ovlivňujeme počet možných vložených poruch, které jsou odsimulovány v jednom průchodu obvodem.

počet bitů	**1**	**2**	**4**	**8**	**16**	**32**	**64**
délka simulace	25,4	25,3	25,2	25,2	25,1	25,1	25,1
délka simulace bez testování	54,4	49,7	45,4	42,6	40,2	38,4	36,8

Tabulka 4.1.7: Závislost délky simulace na velikosti bitového slova



Graf 4.1.3: Závislost rychlosti simulace na velikosti bitového slova

Naměřené výsledky mohou být překvapující. Teoreticky by se dalo předpokládat, že čím větší je velikost bitového slova, tím více se simulace urychlí. Proč tomu tak není, lze však, za pomoci znalostí získaných během práce, vysvětlit. Na pomoc nám poslouží i třetí řádek tabulky *4.1.5*.

Hlavním důvodem zanedbatelné závislosti mezi počtem bitů délky simulace je použití testování toho, zda může být na vodiči detekována alespoň jedna chyba v paralelním slově. Pokud použijeme velké bitové slovo, je malá pravděpodobnost, že nelze na vodiči detekovat chybu. Tedy ve většině případů projdeme všechny cesty od vodiče s poruchou k výstupům obvodu. Těchto průchodů ale bude celkově méně, protože pro jeden průchod testujeme více poruch. Při použití bitového slova o malé velikosti (v krajním případě 1), je mnohem větší šance, že budou všechny bity ve slově shodné a rozdílné od správné hodnoty, tedy že nelze detekovat jedinou poruchu na vodiči. Často tak nastane situace, kdy je paralelní simulace ukončena pro danou větev dříve, než dojdeme k výstupu. Průměrně je tedy počet vodičů navštívených při odsimulování vložených poruch menší, ale zase opačně je počet průchodů vetší z důvodu menšího počtu vložených poruch na jeden průchod obvodem.

4.2. Statistická měření

Statistiky nevyjadřují přímo rychlost algoritmu, ale mohou být nápomocny při dalších pracích na jeho vylepšení. Pokusím se zde odhalit slabá místa algoritmu, či spíše oblasti, na kterých by se dalo dále pracovat.

4.2.1. Opakované návštívení vodičů

Zde se budu zabývat opakovaném navštívením vodičů při propagaci poruch k dominátoru a při paralelní simulaci poruch pro stejný vstupní vektor. V oblasti s dominátory jsou poruchy propagovány k dominátoru – sériová simulace. Všechny vodiče bez dominátorů jsou simulovány paralelně. Navštívení vodičů přímo určuje délku simulace.

obvod	vodiče s dominátory	vodiče bez dominátorů	poměr s/bez
c7552	6 876	676	10,1
s9234	8 332	902	9,2
s38417	33 992	4 425	7,6

Tabulka 4.2.1: Poměr vodičů v oblastech s a bez dominátorů

Počty vodičů v obou oblastech a jejich poměr je uveden v tabulce *4.2.1*. Tato čísla jsou více vypovídající v kontextu s tabulkou *4.2.2,* kde jsou statistiky průměrného opakovaného navštívení vodičů v oblastech s dominátorem a bez nich pro obvod *s38417*. Z této tabulky je patrné, že je u obou způsobů simulace možné provést vylepšení – snížení opakovaného navštívení vodičů.

	100	200	500	1 000	2 000	5 000	10 000	20 000
propagace	59%	59%	59%	57%	54%	49%	43%	37%
paralelní simulace	44%	37%	31%	25%	19%	13%	9%	7%

Tabulka 4.2.2: Opakované navštívení vodičů při propagaci a paralelní simulaci

Opakované navštěvování vodičů je tedy problém, který by se mohl dále řešit. V kapitole 5 uvedu příčiny opakovaného navštěvování a možné řešení tohoto problému.

4.2.2. Průměrné nalezení vložené poruchy

Dalším problémem je stále ještě velké množství poruch, které jsou paralelně odsimulovány, ale nejsou na výstupu detekovány (viz *tab. 4.2.3*). Tento počet stoupá s přibývajícími vstupními vektory. První vektory detekují většinu poruch, každý další vektor nalezne již jen několik poruch. Přesto jsou i pro něj odsimulovány veškeré zbylé poruchy.

počet vstupních vektorů	**100**	**200**	**500**	**1 000**	**2 000**	**5 000**	**10 000**	**20 000**
nalezené / vložené	0,22	0,18	0,15	0,14	0,14	0,12	0,10	0,07

Tabulka 4.2.3: Poměr vložených a nalezených poruch v obvodu s38417

Zatímco pro prvních 100 vstupních vektorů je každá pátá porucha, kterou paralelně odsimulujeme, nalezena, tak pro 10 000 vstupních vektorů je to průměrně již jen každá desátá. Tento průměr se ale vztahuje ke všem 10 000 vektorů, pro například posledních 2 000 vektorů je tento průměr samozřejmě ještě nižší.

Z výše uvedených skutečností vyplývá další možné vylepšení algoritmu, tedy lepší detekce toho, že poruchu nepůjde odhalit na výstupu a tím jí nebudeme pro daný vstupní vektor muset simulovat, či její simulaci brzy ukončíme.

5. Budoucí práce

Paralelní simulátor popsaný v této práci dosahuje velmi dobrých výsledků při porovnávání s referenčním programem HOPE. V některých oblastech jej i překonává. Přesto jsou podle mě další možnosti pro jeho vylepšení. Ty nejpodstatnější uvedu v této kapitole.

5.1. Snížení počtu navštívených vodičů

Snížením počtu navštívených vodičů bychom dosáhli i snížení výpočetního času. Tímto tématem jsem se zabýval již v kapitole *4.2.1.*, zde se pokusím nastínit řešení, které mě při psaní této práce napadlo.

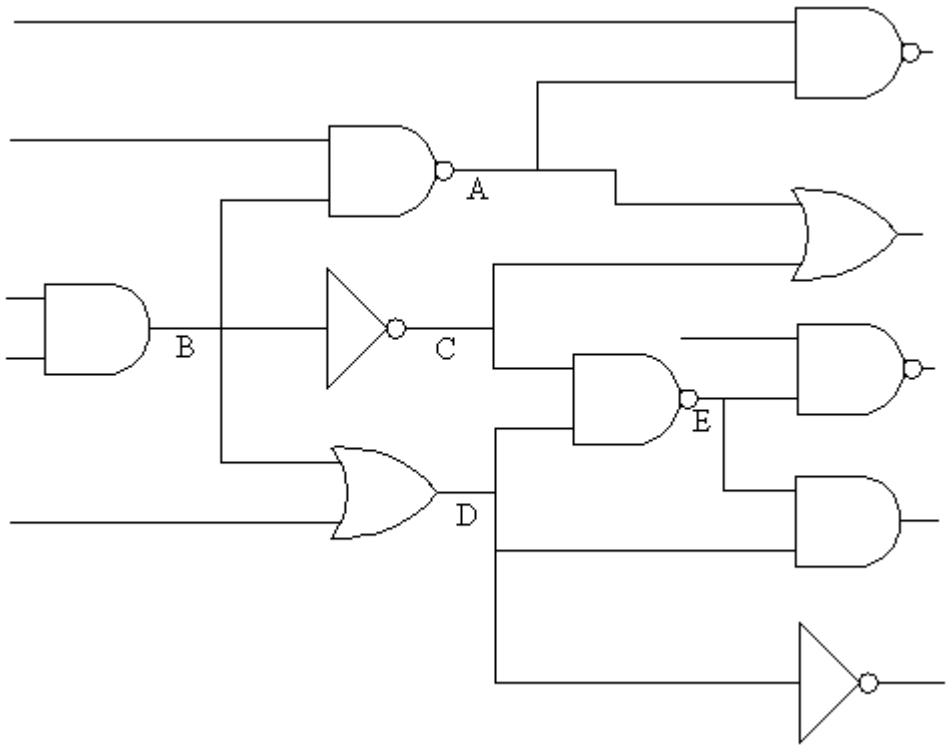
Problém s opakovaným navštěvováním vodičů při simulaci dominátorů se vyskytuje hlavně v prvních fázích simulace, kdy je potřeba odsimulovat velké množství poruch. Při tom je procházena velká část obvodu, a proto se některé simulační cesty překrývají.

Dosud jsme poruchy, které paralelně simulujeme, vybírali "chaoticky". Výběr poruch probíhá postupně podle toho, zda je nutné simulovat přímo poruchu na vodiči bez dominátoru, nebo zda je simulace poruchy na dominátoru způsobená namapováním poruchy z oblasti, které dominuje tento dominátor. Možná by bylo výhodnější nejprve zjistit, které všechny poruchy budeme pro daný vstupní vektor paralelně simulovat (poruchy na dominátorech). Poté by se pro tuto sadu poruch daly vytvořit skupiny poruch, které vložíme do bitového slova, kde by tyto skupiny celkově pro daný vstupní vektor navštívily méně vodičů, než když stejné poruchy simulujeme jako dosud. Pro lepší vysvětlení si pomůžeme obrázkem *5.1.1*.

V obvodu na tomto obrázku máme celkem 5 dominátorů. Uvedeme si dva možné postupy při jejich simulaci. Velikost bitového slova je 3 – můžeme odsimulovat 3 poruchy v jednom průchodu. Nejprve provedeme simulaci "náhodně" – v prvním kroku odsimulujeme poruchy na dominátorech *A*, *C*, a *D*, v druhém průchodu na dominátorech *B* a *E*. V tomto případě navštívíme při simulaci 40 vodičů.

Nyní bychom se pokusili o sofistikovanější přístup. Nejprve bychom z možných poruch vybrali tu, jejíž vodič má nejnižší úroveň v obvodu. K této poruše by byly nalezeny další poruchy, které jsou na cestě od vodiče s poruchou k výstupu obvodu (tedy při simulaci původní poruchy navštívíme ony další hledané poruchy). V

našem příkladu se vybere porucha na dominátoru *B*, poté se nalezne další porucha, která je na cestě k výstupu obvodu – zde třeba na dominátoru *A,* třetí porucha bude na vodiči *C*. V druhém kroku odsimulujeme dominátory *D* a *E*. Tímto postupem se navštíví jen 32 vodičů (zvídavý čtenář nechť se sám přesvědčí).

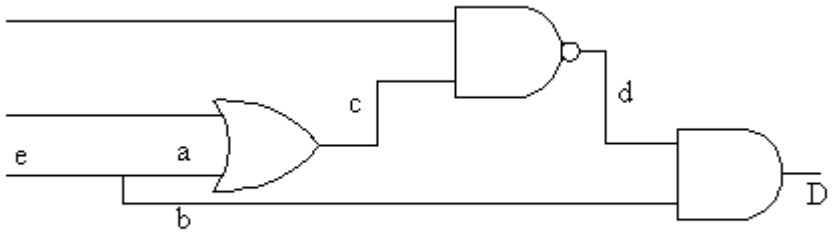


Obrázek 5.1.1: Vylepšení simulace dominátorů

Principem vylepšení simulace dominátorů je využít toho, že při simulaci jednoho dominátoru, navštívíme i jiný dominátor. Tím se vyvarujeme opakovaného simulování některých částí obvodu.

Dalším možným vylepšením by mohlo být to, že by se dalo v jednom kroku simulovat i více poruch, než nám dovoluje velikost bitového slova. Na obrázku *5.1.1*. si to lze představit u dominátorů *A, B, D, E*. Kdybychom vložili do bitového slova o velikosti 3 poruchy na dominátorech *B, D, E*, je ze struktury obvodu patrné, že by se mohla odsimulovat i porucha na dominátoru *A*, jelikož v dané části obvodu nedojde k situaci, že by se do bitového slova nedaly uložit všechny informace o poruchách – tedy v dané větvi nebude třeba na žádném vodiči uchovat informaci o více než čtyřech poruchách. Je však otázkou, zda by se dal obvod uložit do takové struktury, která by rychle umožnila zjišťování těchto skutečností.

Čím je způsobené opakované navštěvování vodičů u propagace poruch k dominátoru dané poruchy, si ukážeme na obrázku *5.1.2*. Princip propagace k dominátoru je uveden v *kap. 2.4.4*. Při použití tohoto algoritmu postupně propagujeme poruchy v oblasti s dominátorem od poruch nejblíže k dominátoru – zde postupně *d, b, c, a, e*. Při propagaci poruchy na vodiči *e* tedy opět navštívíme všechny vodiče v oblasti s dominátorem *D*.



Obrázek 5.1.2: Vylepšení propagace k dominátoru

Princip vylepšené propagace by tedy mohl být použití paralelismu i při propagaci poruch. V jednom průchodu by se tedy otestovaly všechny poruchy v oblasti s dominátorem a tím by se mohlo snížit opakované navštěvování vodičů v této oblasti.

5.2. Snížení redundance vkládaných poruch

Tento problém jsem již objasnil v *kap. 4.2.2.* Jde o to, že velké množství poruch simulovaných paralelně není posléze na výstupu detekováno. V bitovém slově je uloženo 64 poruch, na výstup se ale například propaguje jen 1 – simulaci tedy nemůžeme ukončit dříve, protože je stále možné poruchu ve slově detekovat. Tímto přicházíme o výhody paralelismu. Možným řešením by mohlo být vkládání poruch za běhu simulace. Pokud bychom během simulace zjistili, že na žádném vodiči nelze detekovat poruchu na určité pozici, mohli bychom danou pozici obsadit poruchou jinou, na kterou bychom během průchodu obvodem narazili.

Dalším možností by mohlo být dynamické úpravy velikosti bitového slova. Z naměřených dat (*tab. 4.2.3.*) lze usoudit, že v prvních fázích simulace se velké bitové slovo vyplatí – je nalezeno poměrně velké množství odsimulovaných poruch. U dalších vektorů však úspěšnost nalezení klesá, proto by mohlo být výhodnější bitové slovo zmenšit, díky čemuž by bylo možné simulaci jednotlivých větví obvodu dříve

ukončit. Zvětšila by se tedy šance toho, že projdeme menší část obvodu. Velikost bitového slova by se tedy mohla průběžně měnit v závislosti na úspěšnosti nalezení poruch. Tohoto by se dalo ještě více využít při použití vstupních vektorů s hodnotami X.

5.3. Redukce obvodu

Při bližším prozkoumání obvodů zjistíme, že zvláště větší obvody obsahují velké množství hradel typu invertor, buffer. Některé mohou obsahovat až třetinu těchto prvků ze všech hradel v obvodu. Tato skutečnost je však podstatná až v souvislosti s tím, že tyto prvky tvoří v obvodu zřetězení. Tedy, že jsou za sebou dva a více těchto prvků. Zřetězení prvků by se dalo nahradit buď jedním invertorem nebo jen vodičem, podle toho, jakou logickou funkci celý řetěz tvoří. Tím bychom snížili počet vodičů v obvodu a tedy i rychlost simulace. Pokud by se v takovém zřetězení objevila porucha, mohla by se namapovat na onen prvek, který nahradí celý řetěz.

6. Závěr

Při vytváření této práce jsem nastudoval množství materiálů, které se týkají problému simulace kombinačních logických obvodů. Těchto materiálů a svých znalostí jsem využil pro vytvoření programu, který umožní rychlou simulaci poruch u kombinačních logických obvodů. Simulátor umožňuje i použití vstupních vektorů s hodnotami X (don't care). Zároveň jsem v této práci vytvořil teoretický základ pro případné další práce na simulaci obvodů.

Pro implementaci rychlé simulace jsem použil princip paralelní poruchové simulace. Paralelismus je založen na uložení více informací (zde poruch) do bitového slova (registr) o velikosti 64, kde jsou všechny tyto poruchy odsimulovány jedním průchodem obvodu. Hlavní výhodou paralelní poruchové simulace tedy je snížení počtu průchodů obvodem a tím i snížení celkového simulačního času. Tento způsob simulace má další výhodu v nízké paměťové náročnosti oproti jiným metodám.

Samotný paralelismus však nestačí pro získání výsledků srovnatelných s konkurenčním softwarem (HOPE). Proto je třeba použít i jiné techniky, kterými se simulace urychlí. Z tohoto důvodu jsem při simulaci využil princip dominátorů. Za pomoci dominátorů redukujeme množství poruch, které paralelně simulujeme. Při použití dominátorů odhalíme poruchy s krátkou propagační cestou. Jedná se o poruchy, které jsou detekovatelné jen několik úrovní dále, než kde byly vloženy. Tyto poruchy jsou obtížně detekovatelné a způsobují největší problémy při simulaci.

Během práce na simulátoru jsem odhalil i problém s bezporuchovou simulací (zjišťování správných hodnot na vodičích). Bezporuchová simulace tvořila překvapivě kolem 75% celkového běhu simulátoru (stejné výsledky byly i u HOPE). Tento nedostatek jsem nakonec velice úspěšně vyřešil zavedením paralelismu i do této části simulace.

Výsledkem mého snažení je program, který může v rychlosti směle konkurovat referenčnímu algoritmu HOPE. A nejen to, v některých oblastech je vytvořený simulátor dokonce rychlejší, což potvrzují i testovací měření.

Přestože program dosáhl velmi dobrých výsledků v rychlosti simulace, pokusil jsem odhalit a popsat některá slabá místa tohoto algoritmu. U těchto možných nedostatků jsem se pokusil i navrhnout teoretická vylepšení, na kterých by se mohlo v budoucnu dále pracovat a které by mohly zlepšit, již tak dobré, výsledky programu.

7. Použité zdroje

[1] H.K. Lee and D.S. Ha, *"An efficient forward fault simulation algorithm based on the parallel pattern single fault propagation,"* Proc. Int. Test Conf., pp. 946-955, October 1991

[2] H. K. Lee and D. S. Ha, *"HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits,"* IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 15, pp. 1048-1058, September 1996.

[3] Materiály k předmětu *Diagnostika a spolehlivost*, Prof.Ing. Ondřej Novák, CSc, Katedra počítačů, FEL ČVUT Praha

[4] www.cedcc.psu.edu/ee497i/rassp_43

[5] Fišer, P. - Kubátová, H.: *Pseudorandom Testability - Study of the Effect of the Generator Type*, Acta Polytechnica, Vol. 45, No. 2, August 2005, CVUT, ISSN 1210-2709

[6] F. Brglez and H. Fujiwara. A Neutral Netlist of 10 Combinational Benchmark Circuits and a Target Translator in Fortan, Proc. of International Symposium on Circuits and Systems, pp. 663-698, 1985

[7] F. Brglez, D. Bryan and K. Kozminski. Combinational Profiles of Sequential Benchmark Circuits, Proc. of International Symposium of Circuits and Systems, pp. 1929-1934, 1989

8. Přílohy

8.1. Uživatelská příručka

Program se ovládá pomocí předávání parametrů, proto je nejlepší jej spouštět z příkazové řádky. Jednotlivé možné použité parametry jsou uvedeny v tabulce 8.1. Ještě je nutné dodat, že prvním parametrem musí být vždy simulovaný obvod.

parametr	popis
-n soubor	načtení souboru s popisem obvodu
-f soubor	načtení souboru se seznamem poruch
-t soubor	načtení souboru se vstupnímy vektory
-m soubor	uložení výstupní poruchové masky do souboru – informace o tom, zda byla porucha detekována pro daný vektor
-P soubor	uložení informací o simulaci do souboru

8.1.1. Formát souboru s popisem obvodu

Soubory s popisem obvodu lze použít podle specifikace ISCAS 85 a ISCAS 89. Názvy hradel mohou obsahovat znaky a-z, A-Z, 0-9, _. Řádek s komentářem je označen znakem #. Takovýto řádek je při načítání ignorován. Hlavička souborů s podrobnějším popisem být uvedena nemusí. Na pořadí uložení hradel v souboru nezáleží. Obsahem mohou být hradla typu: INPUT, OUTPUT, AND, NAND, OR, NOR, XOR, NOT, BUFF, BUF.

Příklad souboru s popisem obvodu (c17.bench):

```
# c17 iscas example (to test conversion program only)
# -----------------------------------------------
#
#
#  total number of lines in the netlist .....    17
#  simplistically reduced equivalent fault set size =     22
#       lines from primary input  gates .....     5
#       lines from primary output gates .....     2
#       lines from interior gate outputs .....    4
#       lines from **     3 ** fanout stems ...     6
#
#       avg_fanin  =  2.00,     max_fanin  =  2
#       avg_fanout =  2.00,     max_fanout =  2
#
#
#
#
#
INPUT(G1gat)
INPUT(G2gat)
INPUT(G3gat)
INPUT(G6gat)
INPUT(G7gat)
OUTPUT(G22gat)
OUTPUT(G23gat)

G10gat = nand(G1gat, G3gat)
G11gat = nand(G3gat, G6gat)
G16gat = nand(G2gat, G11gat)
G19gat = nand(G11gat, G7gat)
G22gat = nand(G10gat, G16gat)
G23gat = nand(G16gat, G19gat)
```

8.1.2. Formát souboru se seznamem poruch

Soubor se seznamem poruch používá stejnou specifikaci jako systém Atalanta. Poruchy jsou uloženy po řádcích. Řádek vždy obsahuje název vodiče, na kterém porucha je a dále oddělovač (/) a typ poruchy.

Formát souboru se seznamem poruch:

```
vodičA / 0
vodičA  / 1
vodičA->vodičB /0
vodičA->vodičB /1
```

8.1.3. Formát souboru se vstupními vektory

Tento formát souboru je nejjednodušší. Jedná se o posloupnosti vstupních hodnot, které přísluší k jednotlivým vstupům obvodu. První hodnota tedy přísluší k prvnímu vstupu obvodu, tak jak je uvedeno ve specifikaci obvodu. Vstupními hodnotami mohou být 0, 1, které reprezentují příslušnou logickou hodnotu, dále znaky x, X, 2, -, které značí hodnotu don't care.

8.2. Obsah přiloženého CD

Na přiloženém CD jsou následující adreáře s obsahem:

- \documentation – tato diplomová práce ve formátu pdf
- \bin – spustitelný soubor programu
- \sources – zdrojové kódy programu
- \data – měřená data