TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

# MODIFIKACE AUTOMATICKÉHO GENERÁTORU TESTOVACÍCH VEKTORŮ PRO KOMPLEXNÍ HRADLA

## Bakalářská práce

*Studijní program:* B2646 – Informační technologie

*Studijní obor:* 1802R007 – Informační technologie

*Autor práce:* **Tomáš Kudrna**

*Vedoucí práce:* Ing. Jiří Jeníček, Ph.D.

Liberec 2014

TECHNICAL UNIVERSITY OF LIBEREC
Faculty of Mechatronics, Informatics
and Interdisciplinary Studies

# ATPG MODIFICATION FOR COMPLEX GATES

## Bachelor thesis

*Study programme:* B2646 – Information technology

*Study branch:* 1802R007 – Information technology

*Author:* **Tomáš Kudrna**

*Supervisor:* Ing. Jiří Jeníček, Ph.D.

Liberec 2014

TECHNICKÁ UNIVERZITA V LIBERCI
**Fakulta mechatroniky, informatiky a mezioborových studií**
Akademický rok: **2014/2015**

# ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

| | |
|---|---|
| Jméno a příjmení: | **Tomáš Kudrna** |
| Osobní číslo: | **M10000166** |
| Studijní program: | **B2646 Informační technologie** |
| Studijní obor: | **Informační technologie** |
| Název tématu: | **Modifikace automatického generátoru testovacích vektorů pro komplexní hradla** |
| Zadávající katedra: | **Ústav informačních technologií a elektroniky** |

Zásady pro vypracování:

1. Seznamte se s problematikou generování testů pro číslicové obvody pomocí automatického generátoru testovacích vektorů (ATPG).
2. Navrhněte a implementujte rozšíření existujícího ATPG pro zpracování komplexních hradel.
3. Navrhněte vhodné testovací příklady a ověřte funkčnost.

| | |
|---|---|
| Rozsah grafických prací: | **Dle potřeby dokumentace** |
| Rozsah pracovní zprávy: | **cca 30 stran** |
| Forma zpracování bakalářské práce: | **tištěná/elektronická** |

Seznam odborné literatury:


[1] **Hlavička, J.: Diagnostika a spolehlivost. Praha, Vydavatelství ČVUT, 1998, ISBN 8001018466**

[2] **Lavagno, L.; Martin, G.; Scheffer, L.: Electronic Design Automation for Integrated Circuits Handbook, 2006, ISBN 9780849330964**


| | |
|---|---|
| Vedoucí bakalářské práce: | **Ing. Jiří Jeníček, Ph.D.**<br>Ústav informačních technologií a elektroniky |
| Datum zadání bakalářské práce: | **12. září 2014** |
| Termín odevzdání bakalářské práce: | **5. ledna 2015** |

prof. Ing. Václav Kopecký, CSc.
děkan

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
vedoucí ústavu

V Liberci dne 12. září 2014

# Prohlášení

Byl jsem seznámen s tím, že na mojí bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiji-li bakalářskou práci, nebo poskytnu-li licenci k její využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval samostatně s použitím uvedené literatury.

Datum 4. ledna 2015

Podpis

# Poděkování

Především bych chtěl poděkovat všem, kteří mi poskytovali rady při zpracování mé bakalářské práce. Obrovské poděkování patří Ing. Jiřímu Jeníčkovi, vedoucímu bakalářské práce, za jeho ochotu a trpělivost, cenné rady a připomínky při realizaci této práce. Dále děkuji svým rodičům za podporu nejen při psaní této práce, ale i během celého studia

# Abstrakt

Bakalářská práce se zabývá navržením a implementací rozšíření existujícího ATPG pro zpracování komplexních hradel. Rozšíření se týká aplikace Atalanta, která umí testovat pouze základní jedno-vstupá a částečně dvou-vstupá hradla. Vzhledem k rozšiřujícímu trendu v oblasti číslicových obvodů je tato schopnost nedostatečná, proto bude aplikace rozšířena o možnost testování tří-vstupých až šesti-vstupých hradel. K tomuto účelu budou využita hradla typu LUT.

Koncepce programu zůstane zachována, poněvadž nové funkce pro vytvoření tabulek s hodnotami pro hradla LUT budou přizpůsobeny původnímu programu a jeho funkcím. Uživatel bude k programu přistupovat přes příkazový řádek a výsledek obdrží v nově vytvořeném souboru. Závěrem této práce bude schopnost aplikace Atalanta otestovat obvody, které obsahují hradla až se šesti vstupy, zda neobsahují vadu.



# Klíčová slova

Atalanta, ATPG, stuck-at, D-algoritmus, komplexní hradla, LookUpTable, Init string, hexa, C/C++



# Abstract

This bachelor thesis aims on the design and implementation of extension to existing ATPG, that will allow processing more complex gates. The extension applies to Atalanta application, which can only test the basic gates with one input and partially two inputs gates. Due to the expanding trends in digital circuits, this capability is insufficient, so the application will be extended to include the possibility of testing gates woth three or four inputs. For this purpose, gates of type LUT will be utilized.

The concept of the program will be maintained, since new features that creates tables with values for LUT gates will be adapted to the original program and its functions. The user will operate the application from the command line and will recieve the result in a form of newly created file. The result of this work will be the ability of Atalanta application to test circuits containing gates with up to four inputs whether they are not defective.

## Key words

Atalanta, ATPG, stuck-at, D-algorithm, complex gates, LookUpTable, Init string, hexa, C/C++

# Obsah

# Seznam obrázků



# Seznam tabulek

# Seznam zkratek

| | |
|---|---|
| Sa0 | porucha Stuck-at 0 |
| Sa1 | porucha Stuck-at 1 |
| ATPG | Automatic test pattern generation |
| AND | Logická konjunkce |
| OR | Logická disjunkce |
| NOT | Logická negace |
| NAND | Negovaná logická konjunkce |
| NOR | Negovaná logická disjunkce |
| XOR | Exkluzívní logický součet |
| XNOR | Negovaný exkluzívní logický součet |
| IDA | Identita A |
| NIDA | Negovaná identita A |
| AIB | Implikace - A implikuje B |
| NAIB | Negovaná implikace |
| GND | Logická nula |
| VCC | Logická jedna |
| LUT1 | Look Up table pro jeden vstup |
| LUT2 | Look Up table pro dva vstupy |
| LUT3 | Look Up table pro tři vstupy |
| LUT4 | Look Up table pro čtyři vstupy |
| INIT | Inicializace |
| GATE | Hradlo |
| INPUT | Vstup |
| OUTPUT | Výstup |
| $h_i$ | Vstupní hodnoty |
| $h_p$ | poruchová hodnota |
| $h_s$ | správná hodnota |
| D | hodnota udávající změnu z 0 na 1 |
| DBAR | hodnota udávající změnu z 1 na 0 |
| X | neznámá hodnota |
| MSB | Most Significant Bit - bit s největší hodnotou v binárním vyjádření čísla |
| LSB | Least Significant Bit - bit s nejmenší hodnotou v binárním vyjádření čísla |

# Úvod

Logická hradla (nebo logické členy) jsou (elektronické) obvody, které realizují základní logické funkce jako je logický součin, logický součet, negace atd. Logická hradla jsou základními stavebními prvky logických obvodů a rovněž představují nejjednodušší kombinační logické obvody. Vhodným zapojením logických hradel lze sestavit složitější logické obvody, a to jak kombinační, tak i sekvenční. Funkční bloky sestavené z logických hradel jsou i základem komplexních sekvenčních obvodů jako je mikroprocesor nebo celý mikrokontrolér.

Testování číslicových obvodů (kombinační i sekvenční) je velice důležité odvětví technických věd. Díky němu jsme schopni ověřit správnou funkčnost číslicových systémů. Zjištění správné funkce je důležité především pro výrobce těchto obvodů. Veškeré testování probíhá již při samotném navrhování systému. Vždy je potřeba důkladně prověřit danou fázi, než je přikročeno k dalšímu a rozsáhlému kroku výroby. Nelze totiž nikdy zaručit bezporuchovou výrobu.

Cílem mé bakalářské práce je seznámení se s problematikou generování testů pro číslicové obvody pomocí automatického generátoru testovacích vektorů (ATPG). Následné navržení a implementace rozšíření existujícího ATPG pro zpracování komplexních hradel. A následné vytvoření vhodných testovacích příkladů a ověření jejich funkčnosti. Celkovým výsledkem bude aplikace pro testování komplexních hradel.

# Kapitola 1

# Testování obvodů

Na číslicové obvody je kladen velký nárok v oblasti funkčnosti, tím vzrůstá jejich složitost a potencionální chybovost při výrobě oproti primitivním obvodům. Chybovost může být způsobena jak špatným návrhem, nedokonalou technologií při výrobě, stárnutím či provozem mimo tolerance (teplota, napětí). Možností by bylo, vylepšení technologie výroby, ale tato cesta by byla velmi nákladná a komplikovaná, proto je dána přednost využít testování obvodů a vadné kusy vyřadit. Tímto dokážeme odstranit poškozené kusy, aby se nedostaly k zákazníkovi, ale nedokážeme odstranit příčiny vzniku těchto vad.

Vzhledem k nárokům dnešních výrobců není možné otestovat všechny funkce obvodu, ale výrobcům jde převážně o zisk a ten závisí na počtu vyrobených/prodaných kusů správně funkčních obvodů. Odhalit poruchu obvodu co nejdříve je levnější, než později vracet zákazníkovi finance. Důležité je otestovat funkce obvodu, nebo jeho části, co nejrychleji a s co nejmenšími nároky na další přidané prvky do obvodu.

Testování obvodů se dělí do dvou částí - generování strukturních testů a generování funkčních testů. První fáze se děje ihned po jejich vyrobení, kdy jsou vygenerovány testy pro odzkoušení obvodu a jejich následná simulace na zjištění úplnosti, rychlosti a funkčnosti obvodu. Po testování následuje simulace, která poskytuje výsledky a umožňuje vytvořit slovník poruch.

Tento celkový postup je aplikován v nástroji ATPG, který generuje výstupy, které jsou následně užity při výrobě logického obvodu. Který se na základě výsledků z ATPG ihned po výrobě otestuje. Přesněji se vezme seznam vygenerovaných testovacích vektorů, který otestujeme na daném výrobku a porovnáme správné výstupy s výstupy tohoto výrobku.

# Kapitola 2

# Základní pojmy

## 2.1 Úvod do názvosloví

Každý obvod obsahuje vstupy a výstupy, pomocí kterých komunikuje se svým okolím, tyto vstupy a výstupy mají označení **primární vstupy a výstupy**. Při testování budeme zjišťovat dva základní stavy **poruchový** a **bezporuchový**, to znamená jestli je nebo není daná jednotka schopna plnit předepsanou funkci. U poruchového stavu budeme rozlišovat pojmy **porucha** a **chyba**.

## 2.2 LookUp Table - LUT

Český název pro LookUp table je vyhledávací tabulka. Jedná se o pole, které nahradí složité počítání pouhým indexováním v poli. Úspora času na zpracování může být značná, neboť postup získávání hodnot z paměti, je často rychlejší než složitý výpočet. Tabulky mohou být uloženy staticky v programu, nebo být součástí inicializační vrstvy programu, nebo dokonce uloženy v hardwaru.

## 2.3 Poruchy

V praxi se nám může stát, že máme logický obvod s poruchou, známe jeho zapojení, ale jeho složitost je velmi náročná pro nalezení místa poruchy. Pokud má daný obvod mnoho vstupů, tak je to ještě složitější a často i časově náročnější.

Co je to vlastně porucha? *Porucha* je jev spočívající v ukončení schopnosti obvodu plnit požadovanou funkci. Jak se v obvodu vyskytuje porucha,

v tu chvíli je jeho technický stav **poruchový**, v opačném případě **bezporuchový**. Určení místa poruchy se nazývá **lokalizace poruchy**. Projevem poruchy může být chyba, což je rozdíl mezi správnou a skutečnou hodnotou testovaného obvodu. Chyba je vždy důsledkem nějaké poruchy, ale každá porucha se nemusí vždy projevit chybou.

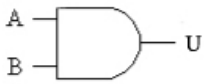
V číslicových obvodech mohou mít poruchy mnoho příčin, ale není důležitá příčina, jako její projev - chyba. Zajímá nás jak se na výstupních hodnotách projeví porucha při určitém vstupu. Bez znalosti topologie obvodu by kompletní test funkčnosti obvodu s **n** vstupy zahrnoval otestování $2^n$ hodnot vstupů, což je pro velký počet vstupů nerealizovatelné.

### 2.3.1 Model poruch

Nejpoužívanějším a nejstarším modelem poruch je Single stuck-at. Výhodou tohoto modelu je, že každá porucha má tři vlastnosti:

- Porucha může být na vstupu nebo výstupu hradla
- Porušený vodič má trvalou hodnotu 0 (t0) nebo 1 (t1)
- Porucha je pouze na jediném vodiči v celém obvodě

| | | | Porucha | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | b | U | a/0 | b/0 | U/0 | a/1 | b/1 | U/1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | (1) |
| 0 | 1 | 0 | 0 | 0 | 0 | (1) | 0 | (1) |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | (1) | (1) |
| 1 | 1 | 1 | (0) | (0) | (0) | 1 | 1 | 1 |




Obrázek 2.1: Model poruch

Vektor vstupních signálů a,b detekuje poruchy typu t, v jejichž sloupci se pro kombinaci vstupních hodnot danou tímto vektorem liší výstupní hodnota od hodnoty U bez poruchy.

Každý vodič v tomto modelu může mít tedy dva druhy poruch Stuck-at 0 (Sa0) a Stuck-at 1 (Sa1): Trvalá jednička se projevuje jako konstantní hodnota "1"na určitém vodiči. A analogicky trvalá nula se projevuje jako

konstantní hodnota "0"na určitém vodiči. Vodičem rozumíme spojení mezi 2 hradly, nebo hradlem a větvením.

V obvodech se vyskytují i další poruchy jako například pomalý náběh/sestup, zkraty, přerušení nebo parametrické poruchy, všechny tyto poruchy jsou při určitém použití shodné s poruchami Sa0 a Sa1. Bohužel to vždy ale neplatí.

### 2.3.2 Seznam poruch

Seznam poruch obsahuje všechny poruchy, které v daném modelu mohou nastat. Pro námi používaný model poruch Stuck-at lze celkový počet poruch vyčíslit jako [1]:

$$\mathbf{N}_p = \mathbf{2} * (\mathbf{N}_i + \mathbf{N}_h + \mathbf{N}_v)$$

Kde:

$\mathbf{N}_p$ - celkový počet možných Sa poruch
$\mathbf{N}_i$ - počet vstupů
$\mathbf{N}_h$ - počet hradel
$\mathbf{N}_v$ - počet větvení

Prakticky je seznam poruch identický s počtem vstupů, hradel a větvení. Vše je vynásobeno právě dvěma, protože nás zajímá na každém vodiči již zmíněné dvě trvalé poruchy Sa0 a Sa1. Seznam poruch slouží jako základní seznam toho, co máme zkontrolovat. Vybíráme z něj jednotlivé poruchy, ke kterým generujeme testovací vektory.

### 2.3.3 Druhy poruch

Pokud narazíme na poruchu, pro kterou ATPG nástroj není schopen vytvořit testovací vektor, není důvod se obávat. Poněvadž některé poruchy jsou netestovatelné a tak je můžeme rozdělit do tří skupin:

- **Testovatelná porucha**: ATPG nástroj je schopen vytvořit testovací vektor pro tuto poruchu
- **Netestovatelná porucha**: ATPG nástroj není schopen vytvořit testovací vektor pro tuto poruchu
- **Redundantní porucha**: jedná se o poruchu, pro kterou neexistuje test

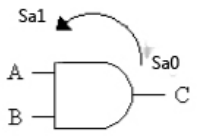
Redundantní porucha vzniká přidáním redundantního prvku do obvodu a je podmnožinou netestovatelných poruch. Důležité je, aby ATPG nástroj nalezl testovací vektory pro všechny poruchy, mimo redundantních. A vypsal i netestovatelné poruchy.

## 2.4 Dominance a ekvivalence poruch

Poruchy můžou být také ekvivalentní nebo dominantní. Tyto druhy poruch se snaží snížit počet kroků testu, když nám test vygeneruje velké množství testů, které je z hlediska detekce nepříjemné. Znalostí těchto závislostí dosáhneme snížením počtu poruch, tím snížíme potřebný čas k simulaci.

### 2.4.1 Dominance

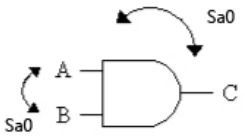
Dominance poruch je, když porucha dominuje druhou poruchu, pokud sada vektorů, které detekují poruchu první, detekují i poruchu druhou. Na následujícím obrázku je znázorněna dominance poruchy Sa0 poruše Sa1. Porucha Sa0 je detekována pro vektory (0, 0), (1, 0) a (0, 1). Poruchu Sa1 lze odhalit jen pro vstupní vektor (0, 1). Proto pokud detekujeme poruchu Sa1, víme že byla detekována porucha Sa0.




Obrázek 2.2: Dominance poruch

### 2.4.2 Ekvivalence

Jedná se o poruchy, které lze detekovat shodnou sadou vektorů a také se projevují stejným efektem. Obrázek ekvivalence poruch ilustruje poruchu Sa0 u obou vstupů i výstupu. Všechny tyto poruchy lze detekovat pouze vektorem (1, 1). Tyto poruchy tedy tvoří stejnou tří ekvivalenci a při simulaci stačí otestovat jen jednu z nich.




Obrázek 2.3: Ekvivalence poruch

## 2.5 Testy

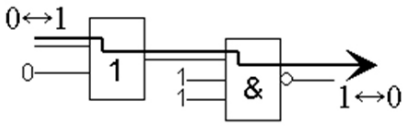
Na testování, přesněji pro generování testů, se používá velmi mnoho různých algoritmů, které se víceméně liší pouze dobou výpočtu a dosaženými výsledky. Tyto algoritmy a jejich testy můžeme rozdělit do dvou skupin **funkční** testy a **strukturní** testy.

### 2.5.1 Funkční testy

Funkční testy vycházejí na základě funkce testované jednotky, takže není nutná podrobná znalost její struktury. Je třeba vytvořit model, který má stejnou funkci testovaného obvodu. Tento model je potřeba vytvořit na stejné úrovni, jako je popsaná funkce.

### 2.5.2 Strukturní testy

Strukturní testy vycházejí ze znalostí struktury obvodu a mají co nejblíže specifikovat místo v obvodu, kde se porucha vyskytla. Tyto testy jsou velmi důležité, protože určují přesné místo v jaké části obvodu došlo k poruše. Pokud se podíváme na tento příklad v praxi, tak v případě, že technik najde jednu poruchu, tak vymění celý obvod. To znamená, že již dále nezáleží na celkovém počtu poruch. Při hledání poruch, se u generování testů používá metoda zcitlivění cesty a metoda generování jednoho kroku testu.




Obrázek 2.4: Citlivá cesta

- **Metoda zcitlivění cesty** - Základem této metody je sestavení přenosové cesty z místa výskytu poruchy na některý z výstupů testované jednotky. Cesta je citlivá, je-li schopna přenášet změny signálu ze svého začátku na jeho konec.

- **Metoda generování jednoho kroku testu** - Tato metoda má za základ vygenerování testovacího vektoru, který pokryje co nejvíc poruch, které lze pokrýt. Metodu opakujeme dokud nejsou pokryty všechny

možné poruchy, krom nedetekovatelných. Úplným pokrytím je zde tedy myšleno vygenerování testovacích vektorů pro všechny detekovatelné poruchy.

## 2.6 D-algoritmus

**D-algoritmus** je jedna z metod, která umí vytvořit citlivou cestu (resp. více citlivých cest naráz). Tato metoda byla nazvána podle použité signálu D, který znamená, že při poruše je na daném vodiči jiná hodnota, jak v bezporuchovém stavu. Tento signál se šíří z místa poruchy na výstup. Použití tohoto signálu vede k 5-hodnotovému modelu šíření signálu v obvodě. Každá z hodnot je definována jako uspořádaná dvojice správné $h_s$ a poruchové $h_p$ hodnoty. Hodnoty modelu jsou v tabulce 2.1.

| **hodnota ($h_i$)** | **význam v bezporuchovém obvodu ($h_s$)** | **význam v obvodu s poruchou ($h_p$)** |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| D | **1** | **0** |
| DBAR | **0** | **1** |
| X | neznámo | neznámo |

Tabulka 2.1: Definice 5-hodnotové modelu šíření signálu v obvodě

## 2.7 Simulace

- **Sériová simulace** - Základní simulací obvodu je simulace sériová. Tato simulace vždy otestuje pouze jednu poruchu při jednom průchodu obvodem. Simulace je jedna z nejpomalejších, ale je jí využito jako doplňkové simulace [5].

- **Paralelní simulace** - Touto simulací otestujeme naráz velké množství poruch, při stejném čase, jako při simulaci jen jedné poruchy, to vše je při simulaci jednoho průchodu obvodem. Vedle rychlosti je další výhodou této metody i nízká paměťová náročnost, oproti deduktivní či souběžné simulační metodě [5].

- **Souběžná simulace** - Souběžná simulace má pro každý člen obvodu vytvořený seznam poruch. Tento seznam uchovává informace o vstupních

a výstupních hodnotách daného členu. Následně při simulaci jsou vybírány seznamy, které na sebe příhodně navazují [5].

- **Deduktivní simulace** - Deduktivní simulace je podobná souběžné simulaci. Avšak tato simulace má jednu velkou přednost, a to že simulace celého obvodu je provedena pouze jednou. Seznam poruch se vytváří a upravuje pro každý signální vodič. Na základě bezporuchových hodnot se u každého logického členu zjistí, které poruchy se mohou propagovat na výstup hradla. Tento způsob simulace potřebuje jen jeden průchod obvodem, ale je časově náročnější než u sériové. Celkově je však tato metoda rychlejší, ale má mnohem větší nárok na paměť [5].

# Kapitola 3

# Rozbor a zpracování problému

## 3.1 Úvodní seznámení

Program Atalanta pochází z Technické univerzity ve Virginii [3] a jeho tvůrci jsou Hyung K. Lee a Prof Dong S. Ha. Atalanta dokáže otestovat funkčnost jednoduchých hradel, nikoliv složitých komplexních hradel. Úkolem bylo tento fakt napravit a původní program rozšířit o možnost testování komplexních hradel.

Prvním krokem bylo potřeba nastudovat problematiku týkající generování testů pro číslicové obvody za pomoci automatického generátoru testovacích vektorů (ATPG). Největším zdrojem informací byly materiály od Hyung K. Lee a Prof Dong S. Ha [3]. Dále jsem využil skripta Diagnostika a spolehlivost od prof. Ing. Jana Hlavičky, DrSc [2]. Po nastudování daných problematik jsem mohl přikročit k navržení vlastního rozšíření.

## 3.2 ATPG Software

V dnešní době je pouze pár kvalitních aplikací, které jsou schopny generovat vhodné testovací vektory ze zadaného obvodu. Uvést můžeme například TetraMAX od firmy Synopsys a FlexTest od Mentor Graphics. Bohužel u těchto nástrojů se nelze seznámit s vnitřní strukturou programu ani jeho výstupy, poněvadž jsou chráněny autorskými právy.

Proto byl zvolen systém Atalanta, který je vyvíjen od roku 1990. Bohužel poslední úpravy byly provedeny v roce 1997. Konkurencí Atalanty může být považován systém Mixed Level Automatic Test Pattern Generation (Milef)[7], který byl vyvinut v roce 1992 v Německu U. Glöserem a H. T. Vierhausem.

## 3.3 Metody generování testů pomocí ATPG

APTG nástroj pracuje na bázi splnitelnosti booleovské formule. Obecně se používají dvě metody generování testu pomocí ATPG - strukturální, které vycházejí ze struktury hradel a reprezentace číslicového obvodu, a algebraické, které převádějí strukturu obvodu do algebraické formy.

### 3.3.1 Strukturální metoda

Metoda, která je založena na algoritmu na prohledávání grafu. Celkový obvod je rozdělen na jednotlivé části, které tvoří vrcholy = hradla, hrany = vodiče. Algoritmus se skládá ze tří základních kroků [1] :

1. Přenastavení hodnoty výstupu hradla na inverzní hodnotu ( 1 -> nastavím 0, 0 -> nastavím 1)
2. Hledáme citlivou cestu z místa poruchy na primární výstup
3. Projdeme graf od výstupů ke vstupům a nalezneme konzistentní ohodnocení vstupů s vnitřními hradly.

### 3.3.2 Algebraická metoda

Tato metoda využívá převodu poruchy na problém řešení algebraického výrazu. Z výsledku, který nám z tohoto algebraického vznikne, odvodíme řešení poruchy. Tuto metodu můžeme využít v případě, že potřebujeme vyřešit poruchu bez znalosti hradel či kombinačních obvodů, poněvadž máme algebraický výraz, který se dá snáze vypočítat.

## 3.4 Atalanta, Atalanta-M

Atalanta je automatický generátor testovacích vzorů pro kombinační obvody. Využívá FAN algoritmu pro generování testovacích vzorů a paralelních jednoduchých vzorů pro simulaci poruch.

Atalanta je vyvinuta v Bradley Department of Electrical Engineering, Virginia Polytechnic Institute & State University (VPI&SU). Zdrojový kód je přístupný pouze pro výzkumné účely, co se týče komerčních účelů, tak ty jsou chráněny autorským zákonem.

Původní Atalanta byla upravena a přejmenována na **Atalanta-M**.

### 3.4.1 Atalanta-M

Aplikace Atalanta-M[8] je spojující ATPG nástroj a integrovaný simulátor poruch HOPE. Aplikace dokáže generovat testovací vektory pro všechny Sa poruchy, nebo pro jednotlivě specifikované poruchy. Umožňuje třístavovou simulaci. A vstupní soubory jsou typu *.bench*.

### 3.4.2 HOPE

Algoritmus HOPE využívá principu paralelní propagace 32 poruch v bitovém slově naráz. Pro urychlení simulace používá princip oblasti bez rozvětvení. Simulátor také umožňuje simulaci vektorů s hodnotami *"don't care"*. Algoritmus HOPE navíc dokáže simulovat i sekvenční obvody. Více informací se lze dočíst v dokumentaci [6].

## 3.5 Volba programovacích prostředků

Původní program Atalanta je naprogramován v programovacím jazyce C a toho jsem se rozhodl držet. Dalším úkolem bylo zvolit si prostředí, ve kterém bude možné program vytvořit. Nejlepším byl zvolen Microsoft Visual Studio 2012, který sice obsahuje pouze překladače pro C++, ale po předefinování knihoven a souborů je schopen přeložit i klasické soubory typu *.c*.

# Kapitola 4

# Popis problému, specifikace cíle

## 4.1 Specifikace cíle

Cílem práce je rozšířit stávající program Atalanta pro testování pomocí vektorových testů hradla s počtem vstupů jedna až čtyři. Stávající program měl pro testovaná hradla vytvořené statické tabulky, ve kterých vyhledával. Tabulky byly vytvořeny pouze pro všechny jedno-vstupé a část dvou-vstupých hradel.

Možností tedy bylo pokračovat ve statickém vytváření tabulek, což z hlediska chybovosti vytvoření tabulek, bylo zamítnuto. Proto bylo přistoupeno k tvorbě dynamických tabulek, které budou vytvářeny na základě vstupních hodnot. Tohoto by mělo být dosaženo pomocí vytvoření nového funkčního bloku LUT.

## 4.2 Popis problému

Aplikací je možné testovat hradla s omezením na počet vstupů. Ale v současné době jsou produkovány větší obvody, které mohou překročit velikost omezení aplikace a tím se aplikace stává nepoužitelnou. Proto je nutné tyto omezení najít a odstranit. Veškeré statické tabulky jsou vytvořeny v souboru *truthtable.h*, kde byly doplněny o nová základní hradla (např. AIB, IDA, NBIA a další).

Jako pokus byly staticky vytvořena některá tří-vstupá hradla jako AND, NAND. To vše aby bylo možné demonstrovat problém v exponenciálním rozšíření a možné chybovosti při deklaraci. Tvorba tabulek pro LUT hradla byla umístěna do souboru *lut.c*, kde jsou veškeré LUT tabulky pro vstupy 1,2,3,4,5 a 6.

## 4.3 Řešení problému

### 4.3.1 Statické

Rozšíření můžeme aplikovat buďto staticky, pomocí tvorby nových obvodů, které staticky dodefinujeme do hlavičkového souboru, nebo můžeme využít vyhledávacích seznamů (LookUp table), která nám umožní definovat vstupní hradlo, podle výstupních hodnot. Následně bude nutné nalézt části kódu, která načítají informace ze vstupního souboru a rozlišují jejich následující funkce.

Tento návrh byl velmi nevýhodný, poněvadž by se jednalo o obrovské množství tabulek, které by byly velmi nepřehledné a pokud by v deklaraci došlo k chybě, tak by bylo jen velmi těžké tuto chybu nalézt. Pro ukázku jsem zvolil hradlo AND.

| Vstupy | | | | | | |
|---|---|---|---|---|---|---|
| **A ↓** | **B →** | **0** | **1** | **X** | **D** | **DBAR** |
| **0** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **1** | | ZERO | ONE | X | D | DBAR |
| **X** | | ZERO | X | X | X | X |
| **D** | | ZERO | D | X | D | ZERO |
| **DBAR** | | ZERO | DBAR | X | ZERO | DBAR |

Tabulka 4.1: Ukázka pravdivostní tabulky pro dvou-vstupé hradlo AND

| Vstupy | | | | | | | |
|---|---|---|---|---|---|---|---|
| **A ↓** | **B ↓** | **C →** | **0** | **1** | **X** | **D** | **DBAR** |
| **0** | **0** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **0** | **1** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **0** | **X** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **0** | **D** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **0** | **DBAR** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **1** | **0** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **1** | **1** | | ZERO | ONE | X | D | DBAR |
| **1** | **X** | | ZERO | X | X | X | X |
| **1** | **D** | | ZERO | D | X | D | ZERO |
| **1** | **DBAR** | | ZERO | DBAR | X | ZERO | DBAR |
| **X** | **0** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **X** | **1** | | ZERO | X | X | X | X |
| **X** | **X** | | ZERO | X | X | X | X |
| **X** | **D** | | ZERO | X | X | X | X |
| **X** | **DBAR** | | ZERO | X | X | X | X |
| **D** | **0** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **D** | **1** | | ZERO | D | X | D | ZERO |
| **D** | **X** | | ZERO | X | X | X | X |
| **D** | **D** | | ZERO | D | X | D | ZERO |
| **D** | **DBAR** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **DBAR** | **0** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **DBAR** | **1** | | ZERO | DBAR | X | ZERO | DBAR |
| **DBAR** | **X** | | ZERO | X | X | X | X |
| **DBAR** | **D** | | ZERO | ZERO | ZERO | ZERO | ZERO |
| **DBAR** | **DBAR** | | ZERO | DBAR | X | ZERO | DBAR |

Tabulka 4.2: Ukázka pravdivostní tabulky pro tří-vstupé hradlo AND

Na těchto dvou tabulkách je jasně vidět, jak se exponenciálně zvyšuje počet hodnot, či-li je větší pravděpodobnost k chybnému zadání dané hodnoty a tím by celý program ztratil svou funkcionalitu.

Proto bylo rozhodnuto, že vlastní logika programu zůstane zachována, poněvadž detekování poruch bude probíhat stejným způsobem jako tomu bylo u původního programu. Pouze dojde ke změně, že tabulky již nebudou staticky naprogramovány v souboru, ale budou dynamicky vytvářeny na základě vstupních parametrů.

### 4.3.2 Dynamické

Dynamické vytváření bylo již krokem vpřed. Hlavním aspektem bylo využití algoritmu, který by vytvořil pravdivostní tabulku na základě init stringu LUT hradla, který bude zadán ve vstupním souboru. Tato problematika byla využita pro tvorbu tabulek LUT.

## 4.4 Popis základního algoritmu

### 4.4.1 Init string LUT hradla

Init string LUT hradla je inicializační konstanta v šestnáctkové soustavě (hexadecimální číslo), která určuje, výstupy dané tabulky pro základní vstupy 1 a 0. Tento způsob můžeme využít u všech LUT hradel, která potřebujeme vytvořit.

### 4.4.2 Dekódování init stringu LUT hradla

Nyní si ukážeme, jak jsme schopni dekódovat init string LUT hradla, pomocí našeho algoritmu. Init string LUT hradla je v našem případě zakódován ve stylu "zleva doprava". MSB bit je první bit z levé strany a LSB je první bit z pravé strany, v případě, že by došlo k prohození, tak by došlo k vygenerování chybných výsledků. Pokud tedy se ve vstupním souboru objeví řetězec např.

**4 = LUT3( 1, 2, 3, "07")**

Řetězec obsahuje:

- 1x výstup "4"
- typ hradla
- 3x vstup "1", "2", "3"

- init string LUT hradla

Z řetězce použijeme prozatím informaci o kolika-vstupé hradlo LUT se jedná a jeho init string. Velikost init stringu LUT hradla je pro každé hradlo jiná, rozlišuje se podle počtu bitů, které jsou potřeba zakódovat.
init string LUT hradel:

- LUT1 $\rightarrow 2^1 = 2$ bity $\rightarrow$ jedna hexadecimální číslice (akceptováno jen číslo 0 až 3)
- LUT2 $\rightarrow 2^2 = 4$ bity $\rightarrow$ jedna hexadecimální číslice
- LUT3 $\rightarrow 2^3 = 8$ bitů $\rightarrow$ dvě hexadecimální číslice
- LUT4 $\rightarrow 2^4 = 16$ bitů $\rightarrow$ čtyři hexadecimální číslice
- LUT5 $\rightarrow 2^5 = 32$ bitů $\rightarrow$ pět hexadecimálních číslic
- LUT6 $\rightarrow 2^6 = 64$ bitů $\rightarrow$ šest hexadecimálních číslic

Init string LUT hradla udává pozici výstupu, který je ONE. Pokud bude init string nulový, tak to znamená, že pro dvouhodnotovou logiku jsou všechny výstupy ZERO.

| **Řádek** | **Vstup** | **Init string** |
|---|---|---|
| 0 | 000 | 01 |
| 1 | 001 | 10 |
| 2 | 010 | 02 |
| 3 | 011 | 20 |
| 4 | 100 | 04 |
| 5 | 101 | 40 |
| 6 | 110 | 08 |
| 7 | 111 | 80 |

Tabulka 4.3: Tabulka pro init string

V této tabulce je možné vidět, jak jednotlivé init stringy odpovídají řádku a vstupu. Na základě této logiky jsme schopni vytvořit init string pro všechny naše LUT hradla, dokonce i pro hradla s více vstupy než čtyřmi. Vstupní init string vždy rozdělíme na základní bitové hodnoty - 1,2,4,8.

Vždy je dopočítáván nejbližší počet bitů, pokud se jedná například o číslo "0A", tak je to init string "02"a "08". Jelikož se jedná o hexadecimální čísla, tak používáme celou soustavu od 0 až po F.
Pro příklad přidávám i ukázku pro LUT3:

$$\mathbf{F2 = 80 + 40 + 20 + 10 + 02}$$

a výsledkem je, že hodnota true bude na řádku 1, 2, 3, 5, 7.

### 4.4.3 Obecná tvorba tabulky LUT

V této části popíši obecný postup pro vytváření LUT tabulek. U každé LUT tabulky na vstupu obdržíme init string LUT hradla. S ním provedeme potřebné operace, které nám ho rozloží na základní čtyři hodnoty 1, 2, 4, 8. Díky těmto hodnotám jsem schopni sestavit tabulku LUT pro 2-hodnotovou logiku a na její výstup umístit hodnoty ONE a ZERO dle vstupního init stringu.

Logika je u všech tabulek LUT stejná, pouze se exponenciálně zvětšují tabulky a init string. Díky init stringu jsme schopni vytvořit hodnoty pro 2-hodnotovou logiku a následně pomocí hodnoty X a D-algoritmu jsme schopni vytvořit zbylé hodnoty pro 5-hodnotovou logiku.

### 4.4.4 Tvorba tabulky LUT1

Tabulka pro LUT1 nabývá čtyř hodnot:

- 0 - Trvalá 0
- 1 - Trvalá 1
- 2 - Předání vstupu na výstup
- 3 - Negace vstupu na výstup

Tyto hodnoty lze nastavit u LUT1.

### 4.4.5 Tvorba tabulky LUT2

Tabulka pro LUT2 nabývá $2^2$ hodnot, zde pracujeme s jednou hexadecimální číslicí.

| **Vstup A** | **Vstup B** | **Init string** |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 2 |
| 1 | 0 | 4 |
| 1 | 1 | 8 |

Tabulka 4.4: Tabulka init string LUT2 pro 2-hodnotovou logiku

### 4.4.6 Tvorba tabulky LUT3

Tabulka pro LUT3 nabývá $2^3$ hodnot, zde pracujeme s dvěma hexadecimálními číslicemi.

| Vstup A | Vstup B | Vstup C | Init string |
|---|---|---|---|
| 0 | 0 | 0 | 01 |
| 0 | 0 | 1 | 10 |
| 0 | 1 | 0 | 02 |
| 0 | 1 | 1 | 20 |
| 1 | 0 | 0 | 04 |
| 1 | 0 | 1 | 40 |
| 1 | 1 | 0 | 08 |
| 1 | 1 | 1 | 80 |

Tabulka 4.5: Tabulka init string LUT3 pro 2-hodnotovou logiku

Následující tabulka je už praktická ukázka pro init string "07".

| Vstup A | Vstup B | Vstup C | Výstup |
|---|---|---|---|
| 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 |

Tabulka 4.6: Tabulka pro init string číslo „07“ pro 2-hodnotovou logiku

Naše tabulka ovšem vypadá jinak, poněvadž nepotřebuje znát vstupy, ty nám dává program sám, tak potřebujeme pouze ukládat výstupy, proto tabulka vypadá v tuto chvíli takto.

Tabulka je neúplná, poněvadž zbytek tabulky by byl prázdný. V tuto chvíli nám algoritmus doplnil pouze hodnoty z init stringu a v další fázi bude doplněn zbytek tabulky.

### 4.4.7 Rozšíření tabulky LUT

Tabulka byla rozšířena i o vstupy D (změna z 1 na 0), DBAR (změna z 0 na 1), X (neznámá hodnota na vstupu, může obsahovat ONE nebo ZERO), které jsou v programu využívány.

| Vstupy | | | | | | | |
|---|---|---|---|---|---|---|---|
| **A ↓** | **B ↓** | **C →** | **0** | **1** | **X** | **D** | **DBAR** |
| **0** | **0** | | ONE | ONE | | | |
| **0** | **1** | | ZERO | ZERO | | | |
| **0** | **X** | | | | | | |
| **0** | **D** | | | | | | |
| **0** | **DBAR** | | | | | | |
| **1** | **0** | | ONE | ZERO | | | |
| **1** | **1** | | ZERO | ZERO | | | |

Tabulka 4.7: Tabulka pro init string číslo „07“ pro 5-hodnotovou logiku

| Vstupy | | | | | | | |
|---|---|---|---|---|---|---|---|
| **A ↓** | **B ↓** | **C →** | **0** | **1** | **X** | **D** | **DBAR** |
| **0** | **0** | | **ONE** | **ZERO** | X | DBAR | D |
| **0** | **1** | | **ONE** | **ZERO** | X | DBAR | D |
| **0** | **X** | | X | X | X | X | X |
| **0** | **D** | | DBAR | DBAR | X | DBAR | DBAR |
| **0** | **DBAR** | | D | D | X | DBAR | D |
| **1** | **0** | | **ONE** | **ZERO** | X | ZERO | ZERO |
| **1** | **1** | | **ZERO** | **ZERO** | X | ZERO | ZERO |
| **1** | **X** | | X | X | X | X | X |
| **1** | **D** | | ZERO | ZERO | X | ZERO | ZERO |
| **1** | **DBAR** | | ZERO | ZERO | X | ZERO | ZERO |
| **X** | **0** | | X | X | X | X | X |
| **X** | **1** | | X | X | X | X | X |
| **X** | **X** | | X | X | X | X | X |
| **X** | **D** | | X | X | X | X | X |
| **X** | **DBAR** | | X | X | X | X | X |
| **D** | **0** | | DBAR | ZERO | X | DBAR | ZERO |
| **D** | **1** | | DBAR | ZERO | X | DBAR | ZERO |
| **D** | **X** | | X | X | X | X | X |
| **D** | **D** | | DBAR | ZERO | X | DBAR | ZERO |
| **D** | **DBAR** | | DBAR | ZERO | X | DBAR | ZERO |
| **DBAR** | **0** | | D | ZERO | X | ZERO | D |
| **DBAR** | **1** | | D | ZERO | X | ZERO | D |
| **DBAR** | **X** | | X | X | X | X | X |
| **DBAR** | **D** | | D | ZERO | X | ZERO | D |
| **DBAR** | **DBAR** | | D | ZERO | X | ZERO | D |

Tabulka 4.8: Kompletní tabulka pro LUT3

V této tabulce je možné vidět, jak je tabulka vytvořena v programu, kde se zachovávají v tabulce pouze výstupy. V tabulce jsem zvýraznil hodnoty, které byly získány z init stringu.

### 4.4.8 Aplikace algoritmu

V první řadě bylo nutné najít, kde dochází k deklaraci původních tabulek a k nim doplnit nové tabulky LUT. Deklarace probíhá v hlavičkovém souboru *fan.h*, *learn.h*, *lsim.h* a *fsim.h*. Na základě těchto poznatků byl vytvořen soubor *lut.h*, do kterého byly naprogramovány vstupy a hlavičky funkcí, které na základě vstupního init stringu vytváření tabulky s danými hodnotami, pro LUT hradla:

```
level LuTbl1[ATALEVEL];
void GenLut1(char* Hexa);
level LuTbl2[ATALEVEL][ATALEVEL];
void GenLut2(char* Hexa);
level LuTbl3[ATALEVEL][ATALEVEL][ATALEVEL];
void GenLut3(char* Hexa);
level LuTbl4[ATALEVEL][ATALEVEL][ATALEVEL][ATALEVEL];
void GenLut4(char* Hexa);
level LuTbl5[ATALEVEL][ATALEVEL][ATALEVEL][ATALEVEL][ATALEVEL];
void GenLut5(char* Hexa);
level LuTbl6[ATALEVEL][ATALEVEL][ATALEVEL][ATALEVEL][ATALEVEL][ATALEVEL];
void GenLut6(char* Hexa);
```

Nyní máme vytvořeny deklarace pro práci s LUT tabulky, ale musíme si vytvořit ještě definice pro názvy LUT tabulek. Tyto definice probíhají v hlavičkovém souboru *define.h*. Zde byly definice doplněny o proměnné pro init string LUT hradel:

```
#define NUM_LUT1 1    /* init string for init LUT1 */
#define NUM_LUT2 2    /* init string for init LUT2 */
#define NUM_LUT3 3    /* init string for init LUT3 */
#define NUM_LUT4 4    /* init string for init LUT4 */
#define NUM_LUT5 5    /* init string for init LUT5 */
#define NUM_LUT6 6    /* init string for init LUT6 */
```

A následně proměnné pro LUT1 až LUT6:

```
#define LUT1 23
#define LUT2 24
#define LUT3 25
#define LUT4 26
#define LUT5 27
#define LUT6 28
```

Nyní už máme deklaraci jednotlivých proměnných a můžeme upravit část kódu, kde dochází k detekci vstupního hradla a následnému přesunu k tabulkám. Tyto funkce se nacházejí v souborech *fan.c*[řádek 125 - 153] a *learn.c*[řádek 108 - 122]. Funkce se jmenuje *gate_eval1*. Tuto funkci jsem upravil pro práci se vstupní proměnnou LUT1 až LUT6 a následného odkazu na jejich vytvořené tabulky.

Pro ukázku, zde přepis kódu ze souboru *fan.c*:

```
#define gate_eval1(g,v,f,i) {\
   switch (g->input):\
      case 1:\
         if(g->fn == LUT1) {\
            GenLut1(g->num_init);\
            v = LuTbl1[g->inlis[0]->output];\
         } else {\
            v = a_truthtbl1[g->fn][g->inlis[0]->output];
         }\
      break;\
      case 2:\
         if (g->fn == LUT2) {\
            GenLut2(g->num_init);\
            v = LuTbl2[g->inlis[0]->output][g->inlis[1]->output];\
         } else {\
            v = a_truthtbl2[g->fn][g->inlis[0]->output][g->inlis[1]->output];
         }\
      break;\
      case 3:\
         if (g->fn == LUT3) {\
            GenLut3(g->num_init);\
            v = LuTbl3[g->inlis[0]->output][g->inlis[1]->output]
            [g->inlis[2]->output];\
         } else {\
            v = a_truthtbl3[g->fn][g->inlis[0]->output][g->inlis[1]->output]
            [g->inlis[2]->output];
```

```
            }\
         break;\
         case 4:\
            if (g->fn == LUT4) {\
               GenLut4(g->num_init);\
               v = LuTbl4[g->inlis[0]->output][g->inlis[1]->output]
               [g->inlis[2]->output][g->inlis[3]->output];\
            } else {\
               v = a_truthtbl3[g->fn][g->inlis[0]->output][g->inlis[1]->output]
               [g->inlis[2]->output[g->inlis[2]->output]];
            }\
         break;\
         case 5:\
            if (g->fn == LUT5) {\
               GenLut5(g->num_init);\
               v = LuTbl5[g->inlis[0]->output][g->inlis[1]->output]
               [g->inlis[2]->output][g->inlis[3]->output][g->inlis[4]->output];
            }\
         break;\
         case 6:\
            if (g->fn == LUT6) {\
               GenLut6(g->num_init);\
               v = LuTbl6[g->inlis[0]->output][g->inlis[1]->output]
               [g->inlis[2]->output][g->inlis[3]->output][g->inlis[4]->output]
               [g->inlis[5]->output];
            }\
         break;\
         default:\
            f = (g->fn == NAND) ? AND : (g->fn == NOR) ? OR : g->fn;\
            v = a_truthtbl2[f][g->inlis[0]->output][g->inlis[1]->output];\
            for( i = 2; i < g->ninput; i++)\
               v = a_truthtbl2[f][v][g->inlis[i]->output];\
            v = (g->fn == NAND || g->fn == NOR)? A_NOT(v) : v;\
         break;\
   }\
```

Na přiloženém kódu vidíte, jak program detekuje počet vstupů a následně odkazuje na příslušnou tabulku v souboru *lut.h*. Účelem našeho kódu bylo zachovat původní funkce, a proto tento kód byl doplněn o funkci "if", zdali se jedná o hradlo LUT a následného přesunu do tabulek, které byly pro příslušný LUT vytvořeny.

### 4.4.9 Vlastní algoritmus v souboru lut.c

Vlastní algoritmus byl naprogramován do nově vytvořeného souboru *lut.c*, který byl následně přidán do původního programu. Soubor obsahuje veškeré deklarace a funkce k příslušným tabulkám LUT.

Algoritmus je naprogramován pro LUT1, kde nabývá pouze čtyř hodnot, buďto hodnotu předává na výstup, nebo jí neguje, nebo je trvale nastaven na 0 nebo 1. Pro LUT2 byly dvě možnosti, buďto odkazovat, po dekódování init stringu, na staticky vytvořené tabulky pro dvou-vstupá hradla, nebo tabulky vytvářet dynamicky. V souboru *lut.c* jsou vytvořeny obě verze, ale verze pro odkazování do *truthtable.h* je zakomentována, poněvadž byla nevýhodná.

V tuto chvíli jsou vytvořeny tabulky LUT1 a LUT2 a je možnost začít vytvářet tabulky pro LUT3 až LUT6. Princip algoritmu je u všech tabulek víceméně stejný. Pro demonstraci jsem si vybral tabulku LUT3. Na vstupu obdržíme init string, které si rozložíme na jednotlivé prvky pole charů. Připravenou tabulku pro LUT3 si naplníme hodnotami ZERO, nikoli celou tabulku. Ale jen pozice, které získáváme z init stringu. Přes funkci *for* postupně projdeme celé pole charů Hexa a pomocí funkce switch-case zjistíme, které pozice máme přepsat na hodnoty ONE. Ukázka na následujícím kódu:

```
void GenLut2(char* Hexa){
   for(a=0;a<5;a++) for(b=0;b<5;b++) { LuTbl2[a][b] = ZERO; }
   switch (Hexa[0]) {
      case '1': LuTbl2[0][0]=ONE; break;
      case '2': LuTbl2[0][1]=ONE; break;
      case '3': LuTbl2[0][0]=ONE; LuTbl2[0][1]=ONE; break;
      case '4': LuTbl2[1][0]=ONE; break;
      case '5': LuTbl2[0][0]=ONE; LuTbl2[1][0]=ONE; break;
      case '6': LuTbl2[0][1]=ONE; LuTbl2[1][0]=ONE; break;
      case '7': LuTbl2[0][0]=ONE; LuTbl2[0][1]=ONE; LuTbl2[1][0]=ONE;
         break;
      case '8': LuTbl2[1][1]=ONE; break;
      case '9': LuTbl2[0][0]=ONE; LuTbl2[1][1]=ONE; break;
      case 'A': LuTbl2[0][1]=ONE; LuTbl2[1][1]=ONE; break;
      case 'B': LuTbl2[0][0]=ONE; LuTbl2[0][1]=ONE; LuTbl2[1][1]=ONE;
         break;
      case 'C': LuTbl2[1][0]=ONE; LuTbl2[1][1]=ONE; break;
      case 'D': LuTbl2[0][0]=ONE; LuTbl2[1][0]=ONE; LuTbl2[1][1]=ONE;
         break;
      case 'E': LuTbl2[0][1]=ONE; LuTbl2[1][0]=ONE; LuTbl2[1][1]=ONE;
         break;
```

```
    case 'F': LuTbl2[0][0]=ONE; LuTbl2[0][1]=ONE; LuTbl2[1][0]=ONE;
       LuTbl2[1][1]=ONE; break;
}
```

Hodnoty ONE jsou zaneseny na pozice, které jsme získali v prvním kroku, dále musíme doplnit hodnoty X, kdy nám není známo, zda-li vstupní hodnota nabývá hodnoty ONE nebo ZERO. Ukázka na následujícím kódu:

```
LuTbl2[0][2] = X;
LuTbl2[1][2] = X;
LuTbl2[2][0] = X;
LuTbl2[2][1] = X;
LuTbl2[2][2] = X;
LuTbl2[2][3] = X;
LuTbl2[2][4] = X;
LuTbl2[3][2] = X;
LuTbl2[4][2] = X;
```

A posledním krokem bylo doplnění hodnot D a DBAR hodnot podle D-algoritmu. D-algoritmus je algoritmické vylepšení intuitivní metody zcitlivění cesty. Test je vytvářen topologickým průchodem obvodu. D-algoritmus je založený na 5-hodnotové logice a D-kalkulu. Také ukázka na kódu:

```
LuTbl2[0][4] = DBAR;
LuTbl2[1][4] = DBAR;
LuTbl2[4][0] = DBAR;
LuTbl2[4][1] = DBAR;
LuTbl2[4][3] = DBAR;
LuTbl2[4][4] = DBAR;

LuTbl2[0][3] = D;
LuTbl2[1][3] = D;
LuTbl2[3][0] = D;
LuTbl2[3][1] = D;
LuTbl2[3][3] = D;
LuTbl2[3][4] = D;
```

Nyní už máme doplněnou celou tabulku pro LUT2 a jsme schopni s ní pracovat při testování obvodu, který toto hradlo obsahuje. Snaha byla zachovat funkčnost programu a přizpůsobit nové algoritmy, aby princip zůstal

stejný. Takto vytvořené funkce a tabulky jsem zakomponoval do programu *Atalanta*, kde bylo potřeba zjistit, kde všude program vyhledává ve statických tabulkách a přidat sem funkce, pro hledání v nových tabulkách. Poté následovalo testování, které je popsáno v následující kapitole.

# Kapitola 5

# Testování obvodu

## 5.1 Testy

Testování probíhá formou testovacích vektorů. Pro test každé stuck-at poruchy je potřeba jen jeden vektor pro primární vstup a odpovídající vektor primárních výstupů. U stuck-at poruch nezáleží na pořadí volby testovacích vektorů jako u delay faults.

U většiny poruch lze vytvořit více různých testovacích vektorů, samozřejmě toto není pravidlem. Pro samotný test stačí pak použít jen jeden vektor. Ten určí jestli algoritmus půjde správným nebo špatným směrem.

V následující tabulce jsou uvedeny testy, na kterých byla ověřena funkčnost programu.

| Řetězce | |
|---|---|
| 10 = AND( 1); | 10 = LUT2( 1, 2, "7"); |
| 10 = OR( 1); | 10 = LUT2( 1, 2, "8"); |
| 10 = AND( 1, 2); | 10 = LUT2( 1, 2, "9"); |
| 10 = NAND( 1, 2); | 10 = LUT2( 1, 2, "A"); |
| 10 = OR( 1, 2); | 10 = LUT2( 1, 2, "B"); |
| 10 = NOR( 1, 2); | 10 = LUT2( 1, 2, "C"); |
| 10 = XOR( 1, 2); | 10 = LUT2( 1, 2, "D"); |
| 10 = XNOR( 1, 2); | 10 = LUT2( 1, 2, "E"); |
| 10 = AIB( 1, 2); | 10 = LUT2( 1, 2, "F"); |
| 10 = NAIB( 1, 2); | 10 = LUT3( 1, 2, 3, "24"); |
| 10 = IDA( 1, 2); | 10 = LUT3( 1, 2, 3, "FF"); |
| 10 = IDB( 1, 2); | 10 = LUT3( 1, 2, 3, "00"); |
| 10 = NIDA( 1, 2); | 10 = LUT3( 1, 2, 3, "AB"); |
| 10 = NIDB( 1, 2); | 10 = LUT3( 1, 2, 3, "50"); |
| 10 = BIA( 1, 2); | 10 = LUT4( 1, 2, 3, 4, "FFFF"); |
| 10 = NBIA( 1, 2); | 10 = LUT4( 1, 2, 3, 4, "0000"); |
| 10 = VCC( 1, 2); | 10 = LUT4( 1, 2, 3, 4, "F001"); |
| 10 = GND( 1, 2); | 10 = LUT4( 1, 2, 3, 4, "ABCD"); |
| 10 = LUT1( 1, "0"); | 10 = LUT4( 1, 2, 3, 4, "1234"); |
| 10 = LUT1( 1, "1"); | 10 = LUT5( 1, 2, 3, 4, 5, "00000"); |
| 10 = LUT1( 1, "2"); | 10 = LUT5( 1, 2, 3, 4, 5, "FFFFF"); |
| 10 = LUT1( 1, "3"); | 10 = LUT5( 1, 2, 3, 4, 5, "12345"); |
| 10 = LUT2( 1, 2, "0"); | 10 = LUT5( 1, 2, 3, 4, 5, "ABCDE"); |
| 10 = LUT2( 1, 2, "1"); | 10 = LUT5( 1, 2, 3, 4, 5, "62489"); |
| 10 = LUT2( 1, 2, "2"); | 10 = LUT6( 1, 2, 3, 4, 5, 6, "000000"); |
| 10 = LUT2( 1, 2, "3"); | 10 = LUT6( 1, 2, 3, 4, 5, 6, "FFFFFF"); |
| 10 = LUT2( 1, 2, "4"); | 10 = LUT6( 1, 2, 3, 4, 5, 6, "123456"); |
| 10 = LUT2( 1, 2, "5"); | 10 = LUT6( 1, 2, 3, 4, 5, 6, "ABCDEF"); |
| 10 = LUT2( 1, 2, "6"); | 10 = LUT6( 1, 2, 3, 4, 5, 6, "251264"); |

Tabulka 5.1: Testy pro ověření funkčnosti

## 5.2 Vlastní testování programu

Testování probíhalo metodou porovnání výsledků (výstupních souborů). Porovnávaly se výstupy předchozího a nového programu. Testovací skript byl sestaven z mnou vybraných vzorků, které byly následně testovány v obou verzích programu. Pro každý vzorek byla provedena série testů, aby byly splněny podmínky.

Hlavním cílem testovaných nově vytvořených hradel bylo, aby vygenerovali testovací vektor. Vzhledem k tomu, že aplikace nemá žádnou dokumentaci, tak jsem se snažil aplikovat definici seznamu chyb na nově vytvořená hradla, ale logika programu dává převážně prioritu hradlům AND/NAND a OR/NOR. Pro ostatní hradla je logika zanedbána. Tuto situaci jsem se pokusil napravit, ale výsledkem bylo zacyklení programu při testování.

Aplikace pro všechny obvody, které jsem pro testování použil, vygenerovala testovací vektor. Tím že definice seznamu chyb se mi nepodařila aplikovat na všechna nově vytvořená hradla, tak některé výsledky neodpovídají předpokladům.

## 5.3 Struktura vstupních a výstupních souborů

### 5.3.1 Soubor .bench

Vstupní soubor má předem definovanou jednoduchou strukturu, která musí být dodržena.

Soubor s definicí obvodu má koncovku *.bench*. Má předem definovanou stavbu, kde vstupy jsou označeny INPUT(číslo stavu), výstupy OUTPUT(číslo stavu). A následuje definice hradel, která je tvořena číslo_stavu = hradlo(vstupA, vstupB) v případě hradla s dvěma vstupy. Vše je zobrazeno v následujícím přepisu kódu.

```
# název obvodu
# počet vstupů
# počet výstupů
# počet invertorů
# počet hradel

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
```

```
INPUT(7)

OUTPUT(22)

10 = AIB(2, 3)
11 = LUT3(1, 6, 7, "24")
17 = NAND(10, 7)
20 = XOR(11, 17)
22 = LUT2(20, 10, "6")
```

### 5.3.2 Soubor .flt

Jedná se o vstupní soubor, kde jsou definované testy, podle kterých se posléze provádějí testy na definovaném hradle.

```
gate_A → gate_B / 0
gate_A → gate_B / 1
gate_A / 0
gate_B / 0
```

Ve výše uvedeném příkladu, gate_A a gate_B jsou názvy bran. První řádek, "gate_A → gate_B / 0"popisuje stuck-at 0 na vstupním řádku gate_B, když je připojen k gate_A. Podobně, druhý řádek, "gate_A → gate_B / 1"popisuje stuck-at 1 na vstupu gate_B, když je připojen k gate_A. Třetí a čtvrtý řádek popisuje stuck-at 0 na výstupu gate_A a výstupem gate_B.

### 5.3.3 Soubor .pat

Výstupní soubor typu .pat nám vytváří sám program, který do něj ukládá výsledky testovacích vektorů, ze kterých my jsme schopni definovat, jestli daný obvod je vyroben s vadou nebo ne. Ukázka výstupního souboru:

```
1 : 00xxx 0x
2 : 101xx 1x
3 : 1x1xx 1x
4 : 010xx 11
5 : 100xx 0x
6 : 001xx 0x
7 : 101xx 1x
```

```
8 : 100xx 0x
9 : 010xx 1x
10: 00xxx 0x
11: x111x x0
```

Před dvojtečkou (:) je pořadové číslo zkušebního vzorku pro konkrétní poruchy. Následuje pětice číslic (bitů), která je hodnota vstupů, v našem případě máme 5 vstupů. První bit je vstup1, druhý pro vstup2, atd. Následuje dvojice bitů, což jsou testovací vektory, které jsou číslovány v opačném pořadí, pro jednodušší analýzu.

# Závěr

Cílem této bakalářské práce bylo seznámení, navržení a implementace rozšíření existujícího ATPG pro zpracování komplexních hradel. Získané informace poté využít při tvorbě rozšíření aplikace. Následně aplikaci otestovat pomocí vektorových testů. Tyto cíle byly naplněny, jak dokazuje aplikace Atalanta uložená na přiloženém CD.

Prvním úkolem bylo doplnění základních jednoduchých hradel. Ty byly doprogramovány do statických tabulek, které byly sestaveny v původním programu Atalanta. Jelikož byly všechna jednoduchá hradla doprogramována do hlavičkových souborů, bylo možné přejít ke kroku rozšíření pro více-vstupá hradla.

Dalším bodem bylo rozšíření stávajícího programu Atalanta pro komplexní hradla. Tímto směrem byly vytvořeny funkce pro tvorbu dynamickým tabulek, které obsahují veškeré výstupy pro testované hradlo. Hradla, která byla vytvořena se nazývají LUT1, LUT2, LUT3, LUT4, LUT5 a LUT6. Pomocí zadaného init stringu LUT hradla, který je předáván do funkcí jako vstupní parametr, se generují pomocí algoritmu tabulky pro 5-hodnotovou logiku, která je využita v aplikaci Atalanta.

Následně byly vytvořeny potřebné testy, které ověřili funkčnost zpracování nově vytvořených hradel. Výsledky tohoto testování jsou popsány v kapitole Testování.

# Literatura


[1] J. Červák: *Automatický generátor testovacích vektorů (ATPG) založený na testu splnitelnosti booleovské formule*. Diplomová práce,FEL ČVUT v Praze, 2008.

[2] prof. Ing. J. Hlavička, DrSc: *Diagnostika a spolehlivost*. Vydavatelství ČVUT, Zikova 4, Praha 6, CZ, 1998.

[3] H.K. Lee, D.S. Ha: *Atalanta: an Efficient ATPG for Combinational Circuits*. Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993.

[4] L. Lavagno, G. Martin, L. Scheffer: *Electronic Design Automation for Integrated Circuits*. Handbook, 2006, ISBN 9780849330964.

[5] prof. Ing. Ondřej Novák, Csc: *Materiály k předmětu Diagnostika a spolehlivost*, Katedra počítačů, FEL ČVUT Praha

[6] H.K. Lee, D.S. Ha: *HOPE: An Efficient Parallel Fault Simulator for Synchronous Sequential Circuits*, IEEE Transaction on Computer-Aided Design of integrated Circuits and Systems, Vol. 115, pp. 1048-1058, September 1996,.

[7] U. Glöser, H. T. Vierhaus: *MILEF: an efficient approach to mixed level automatic test pattern generation*. In Proc. of the Conference on European Design Automation, Los Alamitos, CA, 1992, pp. 318-321

[8] P.Fišer: *Atalanta-M, FEL ČVUT v Praze, 2008.*

# Příloha A

# Obsah přiloženého CD

- Složka "Soubory"– veškeré vstupní soubory, které byly při testování použity
- Složka "Zdrojové kódy"– zdrojové kódy aplikace
- Soubor "Bakalarska_prace_Kudrna_2014.pdf"– text bakalářské práce