Sem vložte zadání Vaší práce.

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

Bakalářská práce

Dekompozice logických funkcí s použitím XOR hradel

Lukáš Rusin

Vedoucí práce: Ing. Petr Fišer, Ph.D.

4. ledna 2013

Poděkování

Na tomto místě bych rád poděkoval vedoucímu práce Ing. Petru Fišerovi, Ph.D. za cenné rady, poskytnutý framework a pomoc s testováním.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 4. ledna 2013 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2013 Lukáš Rusin. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Rusin, Lukáš. *Dekompozice logických funkcí s použitím XOR hradel.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2013.

Abstract

The content of this thesis is bi-decomposition of logic function and its implementation. Bi-Decomposition is realized by the XOR gate. The own method is used and explained. The result is logical circuit in the BLIF format using elementary gates. Results are evaluated and compared with outputs of other tools.

Keywords bi-decomposition, XOR, restrictional model, address variable, PLA, BLIF, BOOM framework

Abstrakt

Obsahem této práce je algoritmus bidekompozice logické funkce a jeho implementace. Bidekompozice je provedena pomocí hradla XOR. Je zmíněn a rozebrán vlastní přístup. Výsledkem je logický obvod ve formátu BLIF, používající elementární hradla. Výsledky jsou vyhodnoceny a porovnány s výstupy jiných nástrojů.

Klíčová slova bidekompozice, XOR, restriktivní model, adresní proměnná, PLA, BLIF, BOOM framework

Obsah

Úvod **1**

1 Teorie **3**
1.1 Formát PLA 3
1.2 Formát BLIF 3
1.3 Restriktivní model popisu funkce 3
1.4 Hradlo XOR 6
1.5 Definice adresní proměnné (AV) 7
1.6 Definice symetrické proměnné (SV) 8
1.7 Úplná dekompozice 8
1.8 Kandidát na bidekompozici 8

2 Algoritmus **9**
2.1 Popis 9
2.2 Bidekompozice pomocí XOR 9
2.3 Top-level 12
2.4 Asymptotická složitost výpočtu jednoho výstupu funkce . . . 14

3 Implementace **17**
3.1 Použité technologie 17
3.2 Povaha 17
3.3 Datové typy 17
3.4 Abstraktní datové struktury 18
3.5 Třídy 18
3.6 Příznaky a oblasti jejich použití 19
3.7 Algoritmus top-levelu 20
3.8 Optimalizace 21

4 Experimenty **23**
4.1 Srovnávací testování 23
4.2 Vliv prahu pro AV na strukturu obvodu 24

Závěr **29**

Literatura **31**

A Seznam použitých zkratek **33**

B Manuál k programu **35**
B.1 Spouštění 35
B.2 Argumenty 35
B.3 Příklad použití 37

C Obsah přiloženého CD **39**

Seznam obrázků

1.1 mapování inverzí - funkce f 6
1.2 mapování inverzí - funkce g, maska 6
1.3 mapování inverzí - výstupní funkce f XOR g 7
1.4 příklad AV 7
1.5 příklad SV 8

2.1 příklad dekompozice restrikce podle prioritní restrikce 11
2.2 bidekompozice pomocí XOR - zdrojová funkce 12
2.3 bidekompozice pomocí XOR - argument SV 13
2.4 bidekompozice pomocí XOR - argument AV 13

4.1 struktura obvodu při použití AV prahu 1 26
4.2 struktura obvodu při použití AV prahu 3 - default 27
4.3 struktura obvodu při použití AV prahu většího než počet vstupních proměnných 28

B.1 argumenty programu 35
B.2 příklad spuštění s použitím některých argumentů 37

Seznam tabulek

1.1 restriktivní součin 4

3.1 tabulka příznaků v programu 19

4.1 testování (1) 24
4.2 testování (2) 25

Úvod

Vytváření obvodů z logických hradel mě začalo zajímat záhy po zahájení vysokoškolského studia. Když jsem se v prvním ročníku na přednášce doc. Kubátové dozvěděl, že v implementaci logických funkcí XOR hradly je ještě nevyužitý potenciál a tato metodika se neustále vyvíjí, toto a záliba v nízkoúrovňovém hardware mě dovedly k výběru tématu mé bakalářské práce.

Logická funkce se dá popsat na abstraktní úrovni, ale až její implementace obvodem rozhodne o efektivitě plnění jejího úkolu. V této práci se zabývám hlavně tímto převodem, jehož výsledek se dá již přímo realizovat jako logický obvod, a ještě lze v něm provádět lokální optimalizace, což je např. implementace vícevstupového hradla AND. Techniky jsou pro to hlavně dvě: syntéza a dekompozice. Společně s vedoucím práce jsme se dohodli na výběru dekompozice, přesněji bidekompozice, kde se dá dobrým způsobem uplatnit rekurze.

Kapitola 1

Teorie

1.1 Formát PLA

Jedním z možných způsobů zápisu logické funkce je formát PLA. Data jsou v něm uložena na způsob tabulky. Řádek tabulky popisuje *součinový term*. Jsou v něm specifikovány hodnoty vstupních proměnných. Ty, které nejsou specifikovány, mají hodnotu DC. Pro term jsou (ale nemusí být) definovány hodnoty výstupních proměnných. Data v PLA mohou mít několik typů. Termy tak mohou tvořit SoP nebo PoS popis funkce, mohou také definovat DC oblasti funkce. [4]

1.2 Formát BLIF

BLIF je způsob zápisu konkrétní implementace logické funkce. Skládá se z uzlů, vstupů a výstupů. Vše je propojeno pomocí shodných názvů. Uzel reprezentuje opět logickou funkci (uloženou ve formátu PLA), ve speciálním případě jedno ze základních hradel. [5]

1.3 Restriktivní model popisu funkce

Pro popis další teorie zavedu pojmy *restriktivní součin*, *oblast logické funkce* a *restrikce*.

1.3.1 Určená hodnota

Určenou hodnotou je logická hodnota z množiny {`0`, `1`}.

1.3.2 Definice restriktivního součinu

Restriktivní součin je komutativní a je nadefinován na množině logických hodnot.

1. operand	2. operand	R*
0	0	0
1	1	1
DC	0	0
DC	1	1
DC	DC	DC
0	1	nedef.

Tabulka 1.1: restriktivní součin

1.3.3 Definice oblasti logické funkce

Mějme logickou funkci `n` proměnných. Potom oblastí logické funkce je uspořádaná n-tice hodnot jednotlivých proměnných. *Elementární oblast* je oblast neobsahující mezi hodnotami proměnných `DC`.

Restriktivní součin dvou oblastí stejné logické funkce je oblast s odpovídajícími hodnotami restriktivních součinů u každé proměnné. Nazvěme ho *průnik oblastí*. Pokud na některé proměnné není definován, není definován ani pro dané dvě oblasti. Oblast *obsahuje* elementární oblast, pokud je mezi nimi definován restriktivní součin.

Podmnožina oblasti je taková oblast, jejíž všechny elementární oblasti, které obsahuje, obsahuje i druhá oblast.

Oblast hodnoty proměnné je definována následovně: Mějme oblast se všemi hodnotami proměnných rovných `DC`. Poté za proměnnou `a` dosaďme určenou hodnotu `x`. Tato oblast se nazývá oblast hodnoty `x` proměnné `a`.

1.3.4 Definice restrikce

Restrikce je uspořádaná dvojice oblasti logické funkce a určené hodnoty. Restrikce je *konzistentní* s jinou restrikcí, pokud mají shodné hodnoty nebo není definován průnik jejich oblastí.

„Logická funkce (ne)obsahuje v oblasti hodnotu x“ je formalizováno následovně: Oblast (ne)obsahuje elementární oblast, pro jejíž hodnoty proměnných má hodnota funkce definovaný restriktivní součin s `x`. Funkce je *konzistentní* s restrikcí, pokud v oblasti dané restrikce neobsahuje hodnotu restrikce, tj. obsahuje pouze inverzi této hodnoty. Např. na obrázku 1.4

je funkce konzistentní s restrikcí s oblastí `(DC,1,1,DC)` a hodnotou `1` (při mapování proměnných v oblasti: `(d,c,b,a)` - což používám vždy takto).

1.3.5 Typy restrikcí

V dalším textu budeme všechny restrikce uvažovat ve vztahu ke stejné logické funkci. *Nulové restrikce* jsou množina restrikcí s logickou hodnotou `0`, analogicky pak v jednotném čísle a pro *jedničkové restrikce*.

1.3.6 Zavedení modelu

Restrikce nám umožňují popsat logickou funkci. Specifikujeme jimi, pro které hodnoty proměnných nesmí funkce nabývat určité hodnoty. Aby byl popis platný, musí být funkce konzistentní se všemi restrikcemi. Zde ovšem tuto definici nelze použít, protože funkce ještě není vůbec definována (definice kruhem). Již korektní podmínka platnosti se vztahuje k restrikcím. Množina všech restrikcí musí být *konzistentní*. To je v případě, že v ní neexistuje dvojice nekonzistentních restrikcí.

Odstranění proměnné z modelu znamená odstranění proměnné z oblastí všech jeho restrikcí, analogicky pak dosazením hodnoty `DC` za proměnnou.

Konstantní funkce (konstanta) je taková logická funkce, pro kterou existuje model s právě jedním typem restrikcí. *Hodnota konstantní funkce* je inverzí hodnoty těchto restrikcí.

1.3.7 Dekompozice restrikce

Dekompozicí restrikce podle jiné restrikce, kterou nazvěme *prioritní*, se rozumí vygenerování nejmenšího možného počtu restrikcí stejného typu jako má původní, jejichž oblasti jsou podmnožinami oblasti původní restrikce a sjednocení těchto oblastí tvoří doplněk průniku oblastí původní a prioritní restrikce.

1.3.8 Sjednocení restrikcí

Sjednocení restrikcí je definováno jako restrikce, pokud obě restrikce jsou stejného typu a zároveň sjednocení jejich oblastí je vyjádřitelné jako jedna oblast, tj. liší se pouze určenou hodnotou jedné proměnné. Výsledná restrikce je stejného typu a její oblastí je tato oblast. Hodnota zmíněné proměnné je nyní `DC`. V jiných případech není sjednocení restrikcí definováno.

1.4 Hradlo XOR

XOR je symetrické hradlo, vstupy jsou komutativní. Představuje součet logických hodnot svých vstupů modulo `2` - paritu. V jeho dvouvstupové variantě můžeme využít ještě jedné vlastnosti: **mapování inverzí**. Mějme libovolnou funkci, např. 4-vstupovou funkci `f`, u které chceme v konkrétních oblastech invertovat výstup. K tomu se dá jednoduchým způsobem použít XOR. Vytvoříme druhou funkci `g`, která na rozdíl od `f` musí být plně určená, tj. neobsahovat `DC`. Obě funkce spojíme XORem. V elementárních oblastech, kde má `g` hodnotu `1`, se výstup invertuje, v `0` se ponechá. Této klíčové vlastnosti využívá můj algoritmus. Pro úplnost je třeba zmínit, že v elementární oblasti obsahující `DC` je nadále opět `DC`, i po inverzi jsou možné obě logické hodnoty. Příklad mapování inverzí je na obrázcích 1.1, 1.2 a 1.3.

X	1	0	X
0	1	0	X
X	0	1	X
1	X	1	1

Obrázek 1.1: mapování inverzí - funkce *f*

1	0	0	0
1	0	0	0
1	0	1	1
1	0	1	1

Obrázek 1.2: mapování inverzí - funkce *g*, maska

X	1	0	X
1	1	0	X
X	0	0	X
0	X	0	0

Obrázek 1.3: mapování inverzí - výstupní funkce f XOR g

1.5 Definice adresní proměnné (AV)

Jako *adresní proměnná* se u logické funkce označuje vstupní proměnná, pro kterou platí: **Existuje právě jedna oblast její hodnoty**, ve které funkce neobsahuje hodnotu `1`. *Protihodnotou AV* je ta hodnota, pro kterou oblast existuje. *Hodnotou AV* je negace této hodnoty.

Příklad AV jako proměnné `b` s hodnotou `0` je na obrázku 1.4.

b
a

0	x	0	x
1	1	0	0
x	1	0	0
1	0	x	0

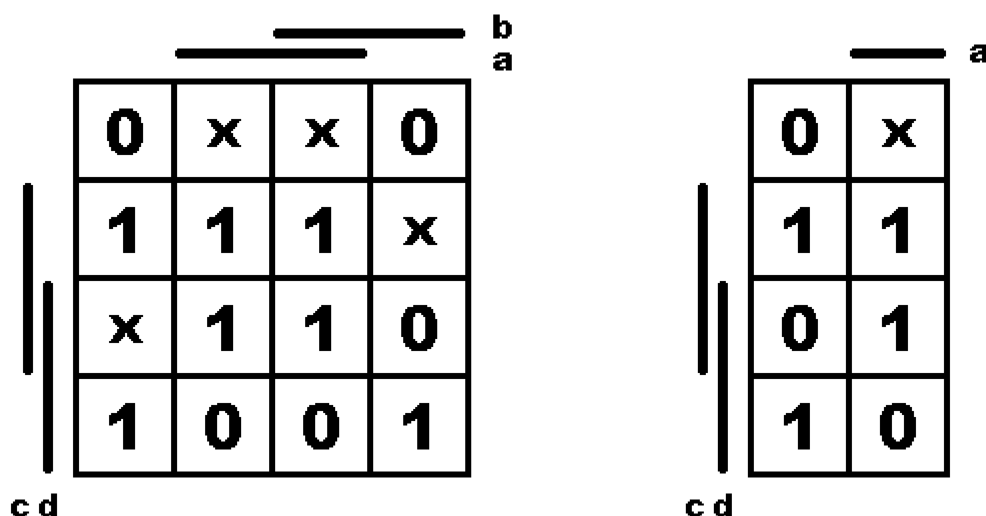
c d

Obrázek 1.4: příklad AV

1.6 Definice symetrické proměnné (SV)

Symetrická proměnná se váže k restriktivnímu modelu popisu logické funkce. Mějme model `m`. Vytvořme model `n` tak, že z modelu `m` odstraníme proměnnou `a`. Pokud je model `n` konzistentní, proměnná `a` je SV.

Příklad SV jako proměnné `b` je na obrázku 1.5.



Obrázek 1.5: příklad SV

1.7 Úplná dekompozice

Úplná dekompozice označuje vytvoření obvodu, který implementuje danou logickou funkci a který obsahuje pouze elementární hradla - XOR, AND, INV atd.

1.8 Kandidát na bidekompozici

Kandidátem rozumíme uspořádanou dvojici vstupní proměnné a její určené hodnoty. *Hodnotou kandidáta* je tato hodnota. *Protihodnotou kandidáta* je inverze této hodnoty. *Oblastí hodnoty x kandidáta* je oblast hodnoty `x` jeho proměnné.

Kapitola 2

Algoritmus

2.1 Popis

Ve své práci budu používat úplnou dekompozici, jednovýstupové uzly a stromovou strukturu obvodu. Výstupní proměnné dané logické funkce se zpracují každá zvlášť. K popisu funkce použiji restriktivní model definovaný v sekci 1.3.

Podstatou mého řešení je rekurze, což v důsledku vytváří i výše zmíněný strom. Ten se bude generovat od kořene. Způsoby větvení budou dva:

- bidekompozice na 2 větve vytvořením kořenového XORu
- filtrace AV vícevstupovým hradlem AND

2.2 Bidekompozice pomocí XOR

2.2.1 Základní myšlenka

Jádrem řešení problematiky je rozdělit logickou funkci na dva argumenty hradla XOR. Z modelů popisu obou takto nově vzniklých funkcí je možné odstranit jednu z proměnných. [1, strana 2, kap. 3.2] Proměnné, které se budou odstraňovat, budou SV a AV.

Každý argument se zpracovává od začátku zvlášť, zpočátku je to funkce identická s původní (stejný model). Pak se provádějí modifikace.

Pokud provedeme XOR obou argumentů, vznikne funkce, pro kterou existuje restriktivní model konzistentní s původní funkcí.

2.2.2 Společný krok

Označme nově vznikající logické funkce jako *SV argument* a *AV argument*, podle typu proměnné, které budeme odstraňovat z jejich restriktivních modelů. V kontextu dané zdrojové funkce náhodně vybereme kandidáta na bidekompozici.

2.2.3 Modifikace SV argumentu

V oblasti protihodnoty kandidáta se ve funkci provede inverze hodnoty `1` na `0`. Tedy všechny nulové restrikce, jejichž oblasti jsou podmnožinami této oblasti, se změní na jedničkové. Toto je důležitý krok k zachování všech restrikcí.

Poté se všechny hodnoty `1` funkce z oblasti hodnoty kandidáta **zpropagují i na druhou stranu**. Tedy v oblastech všech nulových restrikcí, které jsou podmnožinami oblasti hodnoty kandidáta, se hodnota proměnné kandidáta změní na `DC`. Tím ovšem může vzniknout nekonzistence modelu, která se musí odstranit.

2.2.4 Odstraňování zavlečené nekonzistence modelu

Z předchozího kroku nám zbývá odstranit zavlečenou nekonzistenci. Celý problém se řeší dekompozicí těch restrikcí, které jsou nekonzistentní s nově přidanými restrikcemi.

2.2.5 Modifikace AV argumentu

Ze sjednocení oblastí všech nulových restrikcí, které jsou podmnožinami oblasti hodnoty kandidáta, se vytvoří maska. To se provede dosazením hodnoty `DC` za proměnnou kandidáta do všech těchto oblastí.

Podle masky se ve funkci invertují hodnoty v průnicích masky s oblastmi všech restrikcí modelu. Vytvoří se nové prioritní restrikce opačného typu, s oblastmi průniků, nekonzistence se opět vyřeší dekompozicí původních restrikcí podle těchto prioritních. Nyní platí, že proměnná kandidáta je AV s hodnotou kandidátovy protihodnoty.

2.2.6 Popis dekompozice restrikce

Nazvěme tuto restrikci *generující*. Při výpočtu se použijí pouze oblasti obou restrikcí. Vytvoří se uspořádané dvojice hodnot pro všechny proměnné v oblastech. První hodnota je hodnotou vztahující se k prioritní restrikci,

druhá ke generující. Vygenerovaných restrikcí bude tolik, kolik existuje dvojic `(0,DC)` a `(1,DC)`. Oblast vygenerované restrikce se vztahuje vždy ke konkrétní dvojici a vypočte se následovně: V oblasti generující restrikce se do proměnné vztahující se ke dvojici dosadí inverze první hodnoty z dvojice.

U oblastí všech vygenerovaných restrikcí se otestuje, zda již není v modelu restrikce stejného typu, jejíž oblasti by byla podmnožinou. V tomto případě se restrikce do modelu nepřidá, byla by redundantní.

Na příkladu 2.1 je oblast prioritní restrikce z dvojice nahoře, pod ní oblast restrikce generující, oblasti tří vygenerovaných restrikcí pod nimi.

x	1	x	0	0	0	x	1
0	x	x	x	0	x	1	1

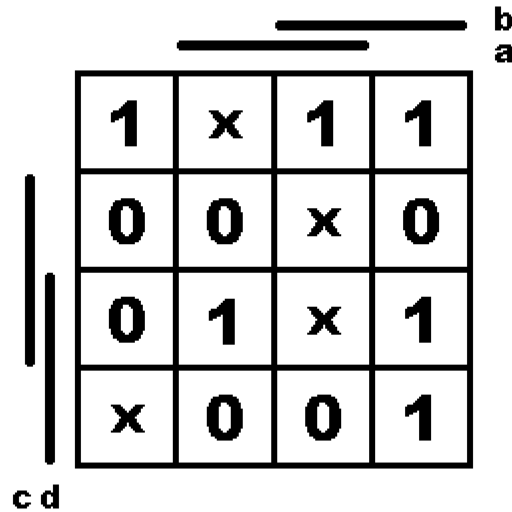
0	0	x	x	0	x	1	1
0	x	x	1	0	x	1	1
0	x	x	x	0	1	1	1

Obrázek 2.1: příklad dekompozice restrikce podle prioritní restrikce

2.2.7 Odstranění proměnných z výsledných argumentů

SV se odstraňuje triviálně (sekce 1.6).

Společně s AV musíme odstranit všechny restrikce, jejichž oblasti jsou podmnožinami oblasti její protihodnoty. AV se po odstranění zaznamená do AV zásobníku přidruženého k právě zpracovávané funkci.



Obrázek 2.2: bidekompozice pomocí XOR - zdrojová funkce

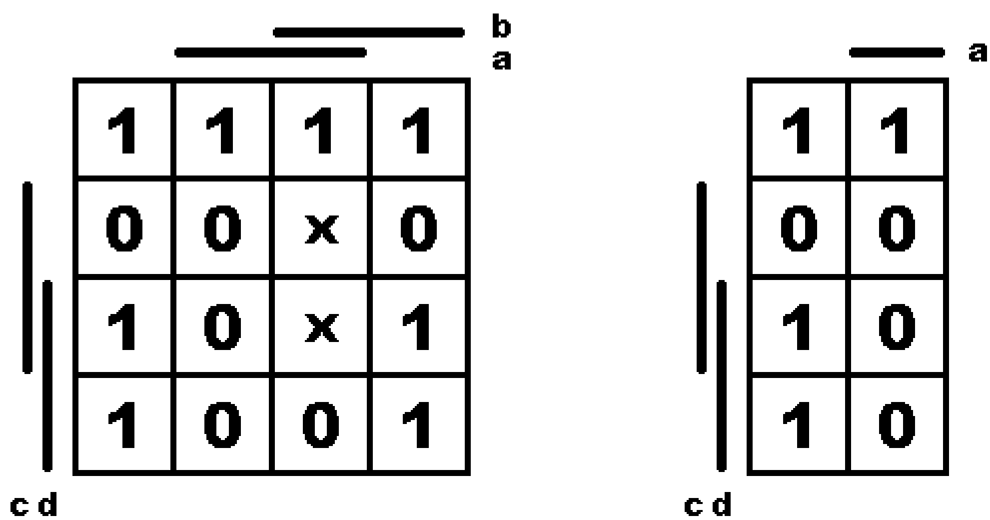
2.3 Top-level

Vstupem algoritmu jsou restriktivní modely funkce pro každou její výstupní proměnnou. Každý se zpracovává zvlášť. V rámci výpočtu se vytvoří uzly obvodu, který se pak optimalizuje.

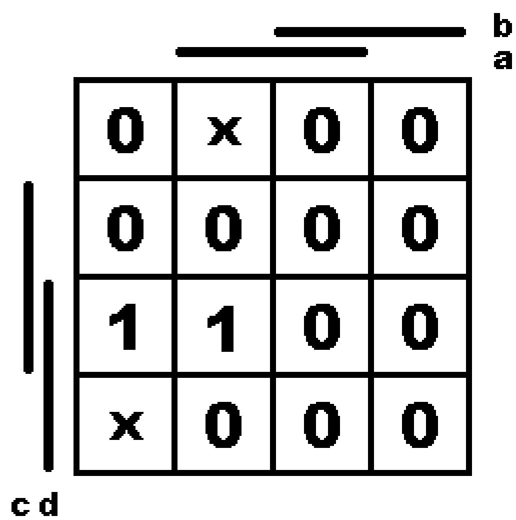
2.3.1 Algoritmus zpracování jednoho výstupu

Na začátku se provede kontrola konzistence modelu. Pokud model není konzistentní, algoritmus končí neúspěchem.

Model se začne rekurzivně zpracovávat. Napřed se provede *simplifikace*. Je složená z odstraňování vstupních proměnných a odstraňování a sjednocování restrikcí. Zkontroluje se, zda model již nepopisuje konstantu nebo není dosaženo určitého AV prahu (vnější parametr výpočtu). V tomto případě se vytvoří hradlo AND (nebo INV), na jehož vstup(y) se namapují



Obrázek 2.3: bidekompozice pomocí XOR - argument SV



Obrázek 2.4: bidekompozice pomocí XOR - argument AV

příslušné vstupní AV. Jejich hodnota se zpracuje prostřednictvím invertovaného vstupu, pokud je 0. Pokud funkce není konstantní, tvoří poslední vstup hradla AND a rekurze pokračuje, jinak pro tuto funkci algoritmus končí. Po vytvoření hradla AND se provede jádro výpočtu - bidekompozice (sekce 2.2) a přidá se hradlo XOR. Rekurze pokračuje na obou větvích - vstupech XORu.

Výše zmíněný typ rekurze je *preorder*. [3]

2.3.2 Odstraňování vstupních proměnných

Napřed se detekují a odstraní SV (1.6).

Poté se zpracují AV (2.2.7). Detekce AV probíhá následovně: Oblasti všech nulových restrikcí musí mít u testované proměnné určenou hodnotu. Pokud ne, proměnná není AV. Provede se restriktivní součin všech těchto hodnot. Pokud je definován, je proměnná AV a její hodnotou je tento restriktivní součin.

2.3.3 Odstraňování a sjednocování restrikcí

Restrikce se odstraňují a sjednocují v iteracích. Každá iterace projde všechny dvojice restrikcí určitého typu. Iterace se opakují tak dlouho, dokud dochází ke slučování restrikcí. Celý tento postup se provádí odděleně pro nulové i jedničkové restrikce.

Restrikce se odstraňuje, pokud její oblast je podmnožinou oblasti jiné restrikce stejného typu.

Dvě restrikce se sjednocují, pokud je jejich sjednocení definováno.

2.3.4 Optimalizace obvodu

Provádí se procházení XOR hradel s právě jedním vstupem konstantní funkce hodnoty 1. Tato hradla se nahradí invertory. (3.8.3)

2.4 Asymptotická složitost výpočtu jednoho výstupu funkce

Pokusím se zde odhadnout průměrné asymptotické složitosti důležitých částí algoritmu. Některé části mají z principu jako nejhorší AS exponenciální, ale ve většině případů je riziko nastání takové složitosti malé a skutečná průměrná AS může být vyjádřitelná např. jako polynomiální.

V dalším textu: `v` jako počet vstupních proměnných, `n` jako celkový počet restrikcí v modelu na začátku, `r` pro jedničkové, `f` pro nulové restrikce.

2.4.1 AS zjištění konzistence

Tento algoritmus je kvadratický, porovnávají se všechny dvojice nulových a jedničkových restrikcí. Tedy:

$$\Theta(r \cdot f)$$

2.4.2 AS odstraňování a sjednocování restrikcí

Pokud jde o odstraňování, máme zde opět stejnou kvadratickou složitost. Ale tento algoritmus se navíc opakuje po alespoň jednom sloučení dvou restrikcí do jedné. Vyjdu z toho, jak rychle takto zjednoduší plně určenou funkci. Proces by vždy sjednotil polovinu restrikcí obou typů a zabralo by to `v` iterací. Vychází tedy:

$$\Theta(r \cdot f \cdot v)$$

Ale s tímto úbytkem nelze univerzálně počítat.

2.4.3 AS odstraňování proměnných

Dominantní složku zde opět tvoří algoritmus zjištění konzistence.

$$\Theta(r \cdot f)$$

2.4.4 AS bidekompozice - argument SV

Pokud by se počet restrikcí dynamicky neměnil vlivem jejich dekompozice, výpočet by vypadal následovně: Podle nulových restrikcí se procházejí jedničkové, což tvoří hlavní složku algoritmu. Dále se při každé dekompozici restrikce provádí optimalizační test na podmnožinu, což tvoří další stupeň polynomu.

$$\Theta(f \cdot r^2)$$

Vlivem možného nárůstu a zpětné vazby může výpočet nabrat až exponenciální ráz, vše záleží na struktuře a rozložení restrikcí. Nutno podotknout, že přesně kvůli tomuto problému byly obohaceny redukce restrikcí o jejich spojování.

2.4.5 AS bidekompozice - argument AV

Stejně jako u předchozího výpočtu, je zde velká nejistota, co se týče dekompozice a tedy přibývání restrikcí. Platí vše co pro výpočet u SV argumentu, jen s tím rozdílem, že tady se podle nulových restrikcí procházejí restrikce obou typů.

$$\Theta(f \cdot n^2)$$

2.4.6 AS celková

Pokud by byl strom rekurze vyvážený a nedocházelo by k průběžnému odstraňování SV, dostali bychom AS exponenciální, se základem 2, podle počtu vstupních proměnných. Náhodná volba kandidáta do algoritmu vnáší náhodnou strukturu výsledku i náhodnou AS. Při nevyváženém stromu nese průměrná AS polynomiální rysy a dá se vyjádřit podle nejvyšší dílčí AS takto:

$$\Theta(v \cdot f \cdot n^2)$$

Kapitola 3

Implementace

3.1 Použité technologie

3.1.1 Jazyk a prostředí

Program byl napsán v jazyce C++, testován v prostředí Windows. Přeloženo překladačem MinGW pro Windows 7 32-bit.

3.1.2 Framework

Při vývoji byl použit BOOM framework. Konkrétně v oblastech:

- načtení dat ze vstupního PLA souboru
- vytváření uzlů obvodu a jeho uložení do BLIFu

3.2 Povaha

Jedná se o nástroj pracující v příkazové řádce, načítající vstupní soubor a vytvářející soubor výstupní. Dle zvolených argumentů může vypisovat různé množství informací, sloužících pro jednoduchou vizualizaci zpracovávaných dat. Argumenty též zajišťují jeho veškerou konfiguraci.

3.3 Datové typy

Kromě datových struktur BOOMu, reprezentujících uzly obvodu, bylo použito jednoduchých datových typů. Jednotlivé restrikce jsou ukládány do typu *char*. Pro ně je v názvosloví programu použito termínu *cubes* (např.

1-cube označuje nulovou restrikci). Jako datová část pro definování jejich oblastí jsou vyhrazeny nejnižší 2 bity. Byty nesou ještě speciální příznaky vztahující se k restrikci, které se nastavují a ruší v průběhu částí výpočtu. Více o příznacích pojednává sekce 3.6.

3.4 Abstraktní datové struktury

S cílem dosáhnout co nejvyšší efektivity byl pro fyzické uložení restrikcí zvolen jednosměrně zřetězený spojový seznam datových bloků pevné délky (sekce 3.5.3). Tyto bloky mohou být opakovaně používány (sekce 3.5.2). Pro každý typ restrikce se používají oddělené seznamy.

3.5 Třídy

Řešení bylo dekomponováno do těchto tříd:

- **XtPLA** - *Extended PLA*, jádro programu
- **BlockMng** - *Block manager*, třída pro správu paměti v datových blocích
- **Block** - jednoduchá třída reprezentující datový blok

3.5.1 Třída XtPLA

Zajišťuje veškeré výpočty. Pracuje vždy s jedním výstupem. Mezi její úkoly patří:

- zjištění konzistence
- simplifikace
- bidekompozice

Algoritmus *zjištění konzistence* je také používán v testech odstraňování SV.

Simplifikace i *bidekompozice* probíhají stejně jak byly popsány v části zabývající se algoritmem (2.2), jen navíc používají příznaky (3.6).

3.5.2 Třída BlockMng

Realizuje správu paměťových bloků. Do maximální míry využívá jejich recyklaci. Tvoří vrstvu odstiňující od alokování i uvolňování paměti. Alokuje blok, jen když je to nezbytně nutné (nemá zrovna žádný alokovaný) a tím se přispívá k minimalizaci času, pokud je potřeba blok alokovat. Toto řešení bylo zvoleno proto, že práce s bloky je v algoritmu velmi dynamická a nové bloky jsou potřeba neustále.

3.5.3 Třída Block

Triviální paměťová třída realizující pouze spojový seznam. Všechny alokované bloky se uvolní až se smazáním příslušného BlockMng, pod který spadají.

3.5.4 Správa a přístup k datům

Přístup k restrikcím vede přes vestavěné iterátory. Ty odkazují na blok a pozici v něm, kde se restrikce nachází. Jednotlivě jsou uloženy v bloku za sebou a jedna nikdy není rozdělena mezi dva bloky. Po ubrání jakékoliv proměnné se pořadí proměnných v restrikci změní tak, aby se díra zacelila. Vznikající mezery na konci restrikcí zruší restrukturalizace, ke které dochází několikrát při simplifikaci modelu funkce.

3.6 Příznaky a oblasti jejich použití

Bit	Příznak	Popis
7	NOCUBE	smazaná restrikce
6		volno
5		volno
4	NEWCUBE	nově přidaná restrikce
3	IGNCUBE	ignorovaná restrikce
2	IGNVAR	ignorovaná proměnná
1		hodnota proměnné - `1`
0		hodnota proměnné - `0`

Tabulka 3.1: tabulka příznaků v programu

Všechny příznaky kromě jednoho se ukládají do prvního bytu v restrikci. Příznak `IGNVAR` se vztahuje jako jediný k proměnné a vyskytuje se kdekoli

v restrikci. Všechny příznaky kromě jednoho jsou vratné. Příznak `NOCUBE` je jako jediný absolutní, restrikce se maže vždy nenávratně. Také všechny příznaky kromě `NOCUBE` jsou jen dočasné, pouze pro potřeby určitých částí výpočtu a poté se odstraňují.

3.6.1 Příznak NOCUBE

Používá se všude v jádře. Je důležitý pro iteraci přístupu k restrikcím, ty s hodnotou tohoto příznaku se přeskakují.

3.6.2 Příznak NEWCUBE

Používán pouze při výpočtu AV argumentu. Tímto příznakem se značí produkty dekompozice restrikce - vygenerované restrikce. Je to důležité proto, aby se poznalo, která z původních nulových restrikcí se má ještě použít jako maska pro inverzi (na začátku známe pouze jejich počet, ten samotný i restrikce se dynamicky mění). Je to daň jednoduché implementace, kde hlavní i vnořený cyklus pracují nad stejnými daty a je třeba zavádět synchronizační příznaky k jejich přístupu.

3.6.3 Příznak IGNCUBE

Také používán pouze při výpočtu AV argumentu. Takto označené restrikce prošly inverzí své hodnoty, tudíž se musí v dalších výpočtech ignorovat.

3.6.4 Příznak IGNVAR

Část, která odstraňuje proměnné, ho používá k testům konzistence po vratném odstranění příslušné proměnné.

3.7 Algoritmus top-levelu

3.7.1 Popis

Ze všeho nejdříve je třeba načíst data. To zajišťuje BOOM, který zpracovává i typ vstupního PLA a případně i vypočítá komplement. [2, strana 1012, kap. 6] Jádro programu XtPLA data převezme do svých struktur, od té doby se pracuje s restriktivním modelem funkce.

Pro každý výstup se vytvoří výstupní buffer, na který se postupně rekurzivně napojí celý budoucí obvod. Poté už se postupuje jen rekurzivně. Každé volání funkce má k dispozici rodičovský uzel, aby se jeho vstupy

mohly pojmenovat podle nově vytvářených uzlů. Hned jak se doplní názvy vstupů do rodičovského uzlu, přidá se rodič do obvodu.

Po úplném vytvoření uzlů všech výstupů se obvod optimalizuje (sekce 3.8.3) a BOOM ho uloží do BLIFu.

3.7.2 Diskuze výběru kandidáta na bidekompozici

Výběr se děje náhodně. Dosáhne se tak snadno nedeterministicky různě kvalitních výsledků, z nichž je možné si vybírat. Dalším argumentem pro náhodný výběr je konstantní složitost. V předchozí verzi programu jsem věnoval výběru kandidáta pozornost, až nakonec došel k závěru, že neexistuje dostatečně jednoduchý nejvýše kvadratický algoritmus, který by přes svou časovou dotaci zaručoval nejlepší volbu. U náhodného výběru je také tendence vyrovnávat charakteristiky obvodu k průměru.

3.8 Optimalizace

3.8.1 Optimalizace funkce před bidekompozicí

Je důležité mít co nejméně restrikcí. Algoritmy jsou polynomiální a takhle i nehrozí takové riziko exponenciálních nároků na paměť i čas. Proto je kladen velký důraz nejen na redukci podmnožin, ale i vyhrazen čas navíc k úplnému „zabalení“. Děje se tak prostým způsobem boolského příznaku, že v dané iteraci došlo ke sjednocení, a na konci se vyhodnotí, zda se iterace bude celá opakovat. Protože po nalezení prvního sjednocení se iterace předčasně neukončí, ale pokračuje i s pozměněnými daty, dává se tak prostor zpracování sjednocení i po větších skupinách.

3.8.2 Optimalizace bidekompozice

Během bidekompozice se v obou výpočtech používá funkce generující dekomponované restrikce, ve které se provádí test podmnožiny (2.2.6). Tato optimalizace má největší podíl na vyrovnávání časů mezi jednoduchými a složitými výpočty, kde za cenu zvýšení stupně polynomu se dosáhne reálného času a hlavně paměťových nároků v některých případech, kde by bez této úpravy došlo k exponenciálnímu průběhu výpočtu.

3.8.3 Optimalizace výsledného obvodu

XORy, které mají na jednom vstupu konstantu `1` (na obou ji z povahy výpočtu mít nemohou), se změní na invertory. Konstanta `0` nemůže být

na vstupu XORu ani ANDu, pouze v případě, že celá funkce je nulovou konstantou. Opět to vyplývá z povahy výpočtu.

Pokud je to možné (na vstupu výstupního bufferu není vstupní proměnná), výstupní buffer se odstraní.

Kapitola 4

Experimenty

Pro potřeby testování existuje u programu speciální mód. Argument `-t` potlačí veškeré ostatní výpisy a vypíší se jen počty vteřin v desetinném zápisu, oddělené středníkem. Sledované údaje jsou dva: čas potřebný na načtení dat (i s výpočtem komplementu, což zajišťuje BOOM) a čas výpočtu obvodu.

4.1 Srovnávací testování

Testy byly provedeny na dvou sadách dat. První jsou obecné PLA soubory, druhá PLA soubory funkcí počítajících paritu. 1.4 U těch by mělo být dosahováno nejlepších výsledků, vzhledem k implementaci parity XORem.

Výsledky programu byly srovnány s těmi z nástrojů ABC a BDS. Navíc, protože výstup se dá dále zpracovat, byly všechny nástroje kombinovány ještě s ABC (kromě něj). Měřil se čas v sekundách, počet hradel, literálů a stupeň obvodu.

Uvedu výsledky pro několik obvodů, a tak demonstrovat rozdílné charakteristiky. Kompletní testový report je na přiloženém CD.

Někdy se může volbou špatných kandidátů na bidekompozici (3.7.2) dosáhnout po všech směrech opravdu špatného výsledku, jako je tomu v případě `09-adder`. Někdy může výsledek mírně vylepšit následná kombinace s ABC. U programu funguje nepřímá úměra mezi počtem literálů a stupněm. Nejlépe je to vidět na `alcom`, kde výsledek za cenu vysokého počtu literálů má i nízký počet hradel, v čemž syntézní nástroje překonává. 4.1

Ve druhém případě se již projevuje výhoda programu při počítání parity. Je to úzká specializace, ve které se program vyrovná, ba i předčí syntézní nástroje. Mohou se generovat extrémnější obvody z hlediska stupně. Je tomu tak proto, že při náhodné volbě kandidáta je jen jeden výrazně vhodnější,

ten vede k optimálnímu výsledku redukcí jednoho podstromu, jiný kandidát jen zvýší stupeň o 1 a žádné SV nevypadnou navíc. 4.2

4.2 Vliv prahu pro AV na strukturu obvodu

Následují tři příklady, na kterých je vidět rozdílná struktura obvodu při použití jiného AV prahu. V prvním (4.1) je nastaveno minimum, 1, AV se tedy promítají všechny do obvodu okamžitě, obvod má obecně méně vstupních uzlů, jeho stupeň je vysoký. Naproti tomu ve třetím (4.3) je AV práh nastaven na vyšší mez, než je vstupních proměnných, v důsledku čehož jsou všechny vstupy umístěny až na (koncových) ANDech. Tento obvod má opačnou charakteristiku. Menší stupeň, maximum vstupů. Jako kompromis jsem zkoušením zvolil hodnotu 3, což je vidět na druhém příkladě (4.2), kde je kombinovaná charakteristika.

Soubor	09-adder	alcom	04-adder
ABC - hradla	114	77	33
ABC - literály	242	156	74
ABC - stupeň	9	4	6
BDS - čas	65410	281	64598
BDS - hradla	54	109	24
BDS - literály	144	192	64
BDS - stupeň	27	5	12
BDS+ABC - hradla	54	74	24
BDS+ABC - literály	126	148	56
BDS+ABC - stupeň	19	4	9
program - load čas	4.07163	0	0.0312002
program - decomp čas	42.4947	0	0.0468003
program - hradla	4878	59	144
program - literály	16305	229	400
program - stupeň	16	4	7
program+ABC - hradla	552	77	47
program+ABC - literály	1298	156	114
program+ABC - stupeň	14	4	9

Tabulka 4.1: testování (1)

Soubor	apex3_p	Altera_ts_mike_fsm_p	alu2_p
ABC - hradla	263	8	284
ABC - literály	528	16	570
ABC - stupeň	14	4	14
BDS - čas	749	16	65160
BDS - hradla	313	9	330
BDS - literály	612	18	715
BDS - stupeň	26	6	38
BDS+ABC - hradla	237	8	268
BDS+ABC - literály	480	16	576
BDS+ABC - stupeň	13	4	25
program - load čas	0.0624004	0	0.0468003
program - decomp čas	1.68481	0	0.109201
program - hradla	953	11	289
program - literály	3828	34	993
program - stupeň	42	6	12
program+ABC - hradla	912	8	234
program+ABC - literály	2058	16	540
program+ABC - stupeň	17	4	12

Tabulka 4.2: testování (2)

```
AND
  x30
  XOR
    XOR
      AND
        NOT x15
        XOR
          NOT x19
          x7
      AND
        NOT x7
        x12
        XOR
          NOT x15
          x19
    AND
      x4
      XOR
        AND
          x12
          x19
          XOR
            x15
            AND
              x27
              XOR
                x15
                x16
        AND
          x27
          x7
          XOR
            x15
            x16
```

Obrázek 4.1: struktura obvodu při použití AV prahu 1

```
AND
  x19
  NOT x4
  NOT x16
  XOR
    XOR
      XOR
        AND
          x17
          x12
        XOR
          AND
            x17
            NOT x30
          AND
            NOT x12
            NOT x30
      XOR
        AND
          x12
          NOT x27
        XOR
          AND
            NOT x30
            NOT x27
          AND
            x17
            NOT x12
            NOT x30
            NOT x27
    AND
      x27
      x30
      NOT x15
      XOR
        x17
        NOT x12
```

Obrázek 4.2: struktura obvodu při použití AV prahu 3 - default

```
XOR
  XOR
    AND
      NOT x17
      NOT x27
      NOT x12
      NOT x30
      x5
      x7
    XOR
      AND
        NOT x17
        NOT x27
        NOT x30
        x15
        x5
        x7
      XOR
        AND
          NOT x17
          NOT x27
          x12
          x15
          x5
          x7
        XOR
          AND
            NOT x17
            x30
            x12
            x15
            x5
            x7
```

Obrázek 4.3: struktura obvodu při použití AV prahu většího než počet vstupních proměnných

Závěr

Úspěšně byl vytvořen softwarový nástroj implementující XOR-bidekompozici. Dalo by se určitě ještě několik jeho vlastností vylepšit, aby např. byl schopen v reálném čase a s reálnými paměťovými nároky zpracovat velké objemy dat. Použitý algoritmus na bidekompozici je velmi nízkoúrovňový, s důrazem na maximální efektivitu, přesto nejde o nijak sofistikovaný algoritmus a šel by vylepšovat podobně jako je např. výpočet komplementu logické funkce u BOOM frameworku.

Na druhou stranu i v této podobě se jedná o výkonný nástroj, zpracovávající většinu PLA souborů v reálném čase. Paměťové nároky jsou, i na úkor času, minimalizovány. Neznamená to, že nemohou být velké, ale spíše je kladen důraz na omezenou spotřebu paměti a neomezenou dobu výpočtu.

Při programování a následném psaní této práce jsem si osvojil využívání částí velkých frameworků a vytváření vlastního teoretického aparátu. V tomto byla pro mě práce přínosem.

Literatura

[1] Alan Mishchenko, M. P., Bernd Steinbach: An Algorithm for Bi-Decomposition of Logic Functions. 2001. Dostupné z: http://www.cs.york.ac.uk/rts/docs/SIGDA-Compendium-1994-2004/papers/2001/dac01/pdffiles/08_2.pdf

[2] Petr Fišer, J. H.: BOOM - A HEURISTIC BOOLEAN MINIMIZER. 2003. Dostupné z: http://users.fit.cvut.cz/~fiserp/papers/cai03.pdf

[3] Töpfer, P.: *Algoritmy a programovací techniky*. PROMETHEUS, s.r.o., první vydání, 1995, ISBN 80-85849-83-6.

[4] The University of Iowa, College of Engineering: *Espresso*. Manuál k PLA. Dostupné z: http://www.engineering.uiowa.edu/~switchin/OldSwitching/espresso.5.html

[5] VLSI/CAD RESEARCH GROUP - University of Colorado at Boulder: *Berkeley Logic Interchange Format (BLIF)*. Manuál k BLIF. Dostupné z: vlsi.colorado.edu/~vis/blif.ps

Příloha **A**

Seznam použitých zkratek

PLA programmable logic array

BLIF Berkeley logic interchange format

BOOM BOOM framework

AV address variable

SV symmetric variable

DC don't care

AS asymptotická složitost

Příloha B

Manuál k programu

B.1 Spouštění

Program se spouští z příkazové řádky. Pokud jej pustíme bez argumentů, vypíše jejich seznam.

B.2 Argumenty

```
arguments:
    INPUT_FILE              a file containing PLA
    [-o] OUTPUT_FILE        BLIF file
    [-a] NUMBER             address vars threshold
    [-g]                    display gates
    [-r]                    display raw data
    [-t]                    display timing only
    [-to] NUMBER            set timeout (in seconds, decimally)
    [-s] NUMBER             random seed
```

Obrázek B.1: argumenty programu

Nepovinné argumenty jsou v hranatých závorkách. Jediným povinným je jméno vstupního souboru. Argumenty jsou buď bez parametrů, nebo s parametrem typu řetezec nebo číslo. Všechny argumenty kromě jména vstupního souboru mají svůj prefix. Lze je proto všechny uvádět v libovolném pořadí.

B.2.1 Argument INPUT_FILE

Argument bez prefixu, kterým specifikujeme cestu ke vstupnímu souboru. Podporován je typ PLA.

B.2.2 Argument -o

Jméno a cesta výstupního souboru. Výstup je typu BLIF. Defaultně „out.blif“.

B.2.3 Argument -a

Tímto nastavujeme práh AV. Defaultně 3. Více o adresních proměnných v sekci 1.5.

B.2.4 Argument -g

Zobrazení výstupního obvodu v textové podobě. Toto slouží pouze pro kontrolu a zběžný náhled, jde o neoptimalizovanou variantu obvodu. Čte se zprava doleva, je stromové povahy. Odsazení odpovídá levelu zanoření.

B.2.5 Argument -r

Zobrazení vstupních dat, tak, jak byla načtena nebo vypočítána z PLA souboru. Jsou seskupena po hodnotách výstupu (`0`, `1`).

B.2.6 Argument -t

Tento argument slouží pro testování (sekce 4), ale jeho označení je odvozeno od *timing* - časomíra. Při jeho použití se vypíší jen časomíry vybraných úseků výpočtu - načítání dat a tvorby uzlů obvodu.

B.2.7 Argument -to

Nastavení timeoutu se uplatní u skriptu spouštějícím mnoho obvodů, z nichž se musí vyloučit ty příliš časově náročné.

B.2.8 Argument -s

Opět pro testování. Nastavuje pseudonáhodný generátor na pevnou hodnotu, takže je možné přesně zopakovat výpočet.

B.3 Příklad použití

```
rusinluk_XOR-bidex.exe ..\PLA\dk48.pla -r -g -a 5 -o dk48.blif
```

Obrázek B.2: příklad spuštění s použitím některých argumentů

Tímto nastavíme vstup i výstup, práh AV na 5, zapnuté zobrazení dat i hradel.

Příloha C

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
├── exe ............... adresář se spustitelnou formou implementace
├── src
│   ├── impl ........................ zdrojové kódy implementace
│   └── thesis ............ zdrojová forma práce ve formátu LaTeX
├── text ..................................... text práce
│   └── BP_Rusin_Lukas_2013.pdf ......... text práce ve formátu PDF
├── PLA ...................................... PLA benchmarky
└── test ................................... výsledky testování
    └── report.ods ............. kompletní report ve formátu ODS
```