Sem vložte zadání Vaší práce.

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

Bakalářská práce

Dekompozice logických funkcí s použitím XOR hradel

Lukáš Rusin

Vedoucí práce: Ing. Petr Fišer

30. prosince 2012

Poděkování

hula

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 30. prosince 2012 .....

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, 2012.

Abstract

hula

Keywords bidecomposition, XOR, PLA, BLIF, BOOM framework, address variable

Abstrakt

hula

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

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Kapitola 1

Teorie

1.1 Formát PLA

1.1.1 Popis

V jádru poměrně jednoduchý způsob zápisu logické funkce umožňuje formát PLA. Soubor PLA obsahuje část deklarační a hlavní část datovou.

Z deklarační části se zaměřím na popis typů dat. Dále jsou důležité počty a jména vstupů a výstupů.

1.1.2 Datová část

Ta se skládá z množiny *termů* - anglicky se uvádí *cubes*. Je to vždy zápis na jedné řádce, hodnoty vstupů a výstupů oddělené mezerou (nebo jiným znakem dle specifikace). Výstupy jsou na sobě nezávislé. Hodnota výstupu určuje jeden ze tří druhů termu - onset, offset a DC-set. Vstupy tvoří vstupní proměnné, které také nabývají tří stejných hodnot, ovšem tady s trochu jiným významem. Hodnota určuje, jakým způsobem je proměnná v termu obsažena. Pokud je to `0` nebo `1`, hodnota výstupu je požadována jen když proměnná této hodnoty nabyde. Pomlčka značí DC a ten se dá myslím vykládat dvěma různými způsoby.

První interpretace značí, že term danou proměnnou neobsahuje, tudíž jeho hodnota je podmíněna jinými proměnnými. Další, že jde o tzv. *duální term* vzhledem k proměnné, a term je platný při jakékoliv její hodnotě, pokud mají ostatní proměnné odpovídající hodnoty těm v termu. V důsledku to znamená, že takovýto term představuje zhuštěný zápis dvou termů lišících se jen rozdílnou hodnotou této proměnné - první má nulu a druhý jedničku. Takto můžeme rozložit všechny termy a dostaneme funkci defino-

vanou tabulkou o velikosti

$$2^n$$

, kde `n` je počet vstupních proměnných. Velikost termu vzhledem k aktuálnímu počtu vstupních proměnných je určena právě počtem DC ve vstupech:

$$2^{n(DC)}$$

míst (variací vstupních proměnných), kam term zasahuje.

1.1.3 Kolize

Kolize termů, nebo také nekonzistence funkce, nastane, když průnik libovolné dvojice termů nestejné hodnoty (kromě DC) je neprázdná množina. V takovém případě nastane, že funkce musí v daném místě být zároveň `0` i `1`, což pochopitelně splnit nelze. Jedná se tedy o chybu ve vstupních datech a funkce není realizovatelná obvodem.

1.1.4 Typy dat

Klíčovým slovem `type` se uvádí, jakého data budou typu. Je teoreticky 7 možných typů. Vyplývá to z počtu kombinací tří písmen *f*, *r* a *d*, skupinu kterých za slovem `type` uvádíme. Jediná neakceptovatelná kombinace je prázdná kombinace. Defaultně je nastaveno *fd*. Co jednotlivá písmena znamenají? Říkají, které druhy termů se mají z dat do funkce zahrnout:

- *f* - termy typu `1` (onset)
- *r* - termy typu `0` (offset)
- *d* - termy typu DC

Platí zde určitá priorita. Pokud je *d* obsaženo v typu, je prostor DC termů zaručen a případné jiné termy jsou v tomto prostoru potlačeny.

Pokud chybí jedno z písmen *f* nebo *r*, nemůže dojít ke kolizi a chybějící prostor mimo případné DC termy se dopočítává.

1.1.5 Alternativní úhel pohledu

Pro účely následné implementace jsem výše zmíněné formální popisy pojal ještě z jiného úhlu pohledu. Dal by se označit jako *restriktivní model*. Vychází z myšlenky, že bez termů je výstup funkce DC, tudíž jeho hodnota může být jakákoliv. Postupným přidáváním termů přidáváme i restrikce,

kde již nemůže být druhá hodnota. Čili hodnota termu např. `1` z tohoto pohledu neříká, že „zde mají být jedničky“, ale naopak, že „zde již nesmí být nuly“. DC termy jsou v tomto modelu jen čistě technický prostředek formátu PLA, jak propagovat DC na určitá místa, a tady nemají žádný význam, protože žádnou restrikci nepřidávají.

1.1.5.1 Pohled na kolizi

Celý logický prostor výstupu funkce se může rozdělit maximálně na tolik částí, kolik vyplývá z počtu proměnných, jak je napsáno výše. Pokud funkci můžeme označit za bezkolizní, měla by mít v každé této části (pro každou variaci hodnot vstupů) **povolenu alespoň jednu hodnotu 0, 1**. Pokud se sejdou restrikce (termy) takové, že z nějaké části odstraní obě hodnoty, je to špatně a funkce je nekonzistentní, protože jakákoliv hodnota výstupu by byla ve své restrikci. Každá část tedy může obsahovat dva nezávislé typy restrikcí - pro každou logickou hodnotu.

Pokud bych to měl shrnout, pohledem tohoto model říkáme, které hodnoty na výstupu funkce nechceme.

1.2 Formát BLIF

1.2.1 Stručný popis

Na rozdíl od formátu PLA, u kterého jsem také zdaleka nezmínil všechny specifikace, zde také se budu soustředit pouze na to, co souvisí s implementací řešení mé problematiky.

V té nejjednodušší podobě se jedná o popis logické funkce, neomezené počtem vstupů a výstupů. Jsou povinně uvedeny jejich názvy. Další částí jsou uzly obvodu. Uzel je také funkcí s libovolným množstvím vstupů a výstupů. Jeho funkcionalita se zapisuje do PLA, např. typu f. Komplement je při interpretaci obvodu vždy dopočítáván. Uzel i všechny jeho výstupy jsou jednoznačně pojmenované.

1.2.2 Network

S BLIFem se úzce pojí pojem *Network*. Jde vlastně o model, jehož základní charakteristika je popsána výše. Celý je založen na dekompozici původní funkce do více uzlů, které mohou představovat hradla, ale také nemusí být triviální a tvoří podfunkce. Propojení jednotlivých uzlů i namapování vstupů a výstupů je zajištěno stejným pojmenováním.

Ve své práci a v programu budu používat úplnou dekompozici až na úroveň hradel, jednovýstupové uzly a stromovou strukturu jednotlivých výstupů původní funkce, které tak budou tvořit les na sobě nezávislých stromů.

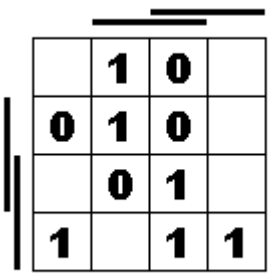
1.3 Myšlenka postupu

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

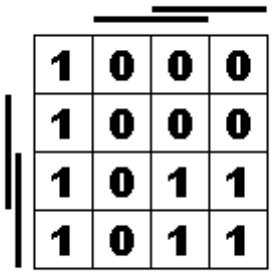
- bidekompozice na 2 větve vytvořením kořenového XORu
- filtrace proměnných vícevstupovým hradlem AND

1.4 Využití vlastností XORu

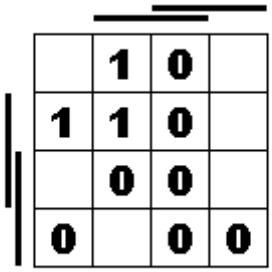
XOR je obecně symetrické hradlo. Představuje vlastně součet hodnot svých vstupů modulo 2 - na jednom bitu výstupu. V jeho dvouvstupové variantě můžeme využít ještě jedné vlastnosti: mapování inverzí. Mějme libovolnou funkci, např. 4-vstupovou funkci f, u které chceme na konkrétních místech invertovat 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. Na místech, kde má g jedničky, se výstup invertuje, v nulách se ponechá. Této klíčové vlastnosti algoritmus využívá. Pro úplnost je třeba zmínit, že v místě DC je nadále opět DC, i po inverzi jsou možné obě hodnoty.



Obrázek 1.1: využití vlastností XORu - funkce f



Obrázek 1.2: využití vlastností XORu - funkce g, maska



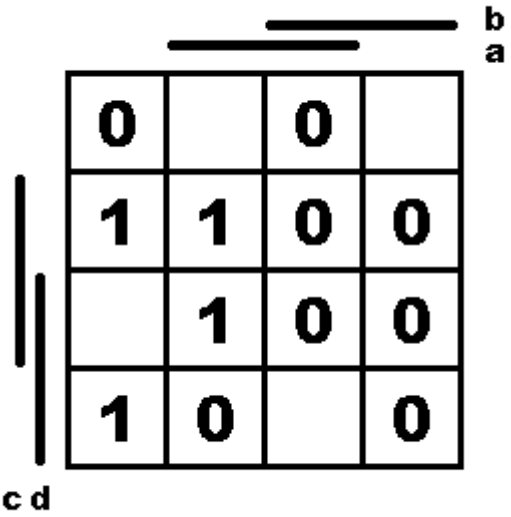
Obrázek 1.3: využití vlastností XORu - výstupní funkce

1.5 Adresní proměnné

1.5.1 Definice

V souvislosti s dalším postupem je třeba zmínit ještě jeden pojem. Adresní proměnnou (AV) definujeme jako uspořádanou dvojici vstupní proměnné a její hodnoty. Pro vstupní proměnnou musí platit, že **právě v jedné její hodnotě (0 nebo 1) nejsou ve funkci přítomné jedničky**. Hodnota AV je pak inverzí této hodnoty. Prakticky to znamená, že AV jedničky ve funkci adresuje, odtud název.

Příklad AV s hodnotou `(b,0)` je na obrázku 1.4.



Obrázek 1.4: příklad AV

1.5.2 Eliminace

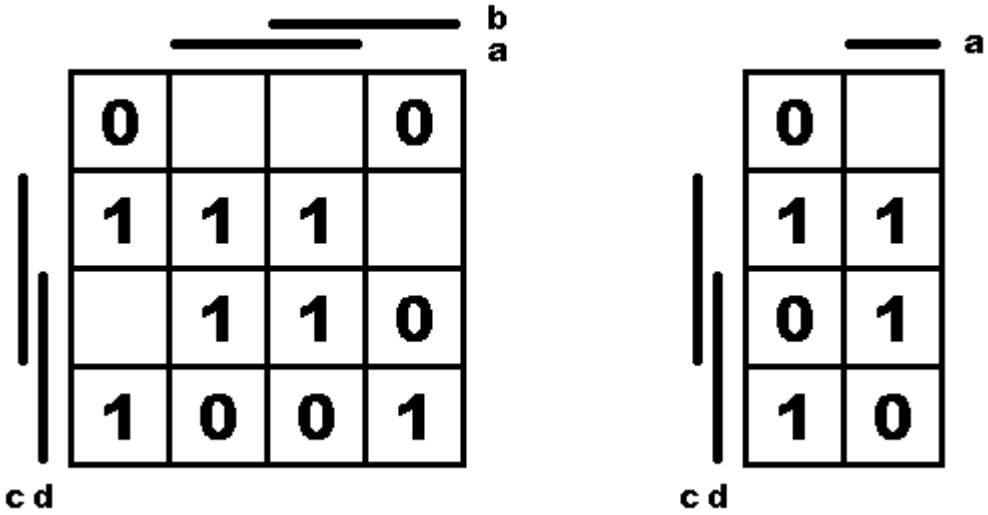
AV se z funkce odstraňují pomocí hradla AND. To má kromě vstupu pro funkci vstupy napojené na příslušné proměnné. Hodnota AV se zpracuje prostřednictvím invertovaného vstupu. Výsledkem v obvodu je tedy výstup z hradla AND, který je z jeho povahy nulový všude mimo adresu tvořenou hodnotami AV. Takovéto řešení má tedy restriktivní účinek, eliminuje všechny DC mimo oblast adresy.

1.6 Symetrické proměnné

1.6.1 Definice

Tato proměnná je ve funkci nadbytečnou. Neměla by se objevovat v obvodu. Je to proměnná, **jejímž odstraněním se zachová bezkoliznost (konzistence) funkce**. Dobrou představu získáme, představíme-li si vybranou proměnnou jako rozlišovací prvek, mapující svými hodnotami termy funkce do dvou zcela nezávislých světů. Pokud bychom proměnnou odebrali, nestane se nic jiného než že se tyto světy namapují na sebe (dojde ke sjednocení restrikcí - více sekce 1.1.5). Pokud nedojde ke kolizi restrikcí, byla proměnná symetrickou.

Příklad SV, v tomto případě `b`, je na obrázku 1.5.



Obrázek 1.5: příklad SV

1.6.2 Eliminace

SV nemusíme z funkce odstraňovat na úrovni obvodu, spíše je vhodné podívat se na důsledky jejího odstranění z PLA popisu funkce. Jak je vidět na obrázku 1.5, ve funkci bez proměnné opět mohou přibýt restrikce. DC se totiž zachová pouze v případě, že je přítomen v obou hodnotách dané proměnné. Pokud je v jedné z nich hodnota funkce definována, funkce bez proměnné musí převzít tuto hodnotu. V opačném případě (zachování DC) by funkce pak mohla být reprezentována obvodem, který by pro dané místo vracel hodnotu opačnou. Nutno dodat, že toto zúžení restrikcí se děje v PLA popisu zcela automaticky, je nutné dodržet pouze podmínku bezkoliznosti, pak lze proměnnou vždy odstranit.

1.7 Bidekompozice pomocí XOR

1.7.1 Základní myšlenka

Jádrem řešení problematiky je rozdělit funkci na dvě pomocí hradla XOR. Z každé takto nově vzniklé funkce by se měla dát již triviálním způsobem od-

stranit jedna z proměnných. Můj postup vytvoří dvě funkce jako argumenty XORu, ze kterých půjde odstranit SV, resp. AV výše uvedenými postupy.

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

Pokud provedeme XOR obou argumentů, nemusí vzniknout úplně identická funkce, část informace o DC se ztratí, restrikcí není nikdy méně. Nejdůležitější je, že máme opět vymezenou původní funkci, kde všechny určené hodnoty původní funkce se shodují s těmi jejími.

1.7.2 Společný krok

Vybere se proměnná a strana - její hodnota. Nazvěme tuto dvojici *kandidátem*. Výběr může být libovolný.

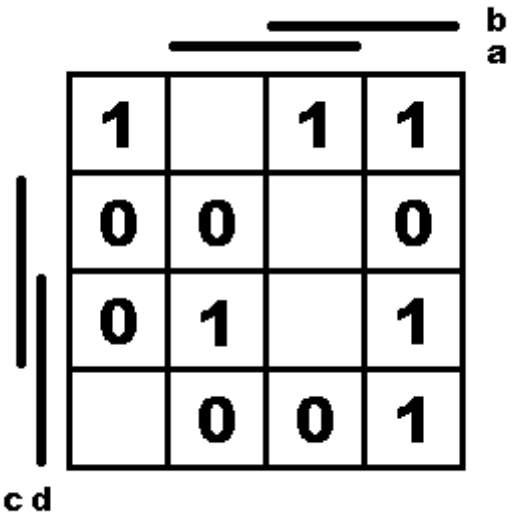
1.7.3 Modifikace SV argumentu

Na opačné straně funkce, než určuje kandidát, se napřed invertují jedničky na nuly. To, že se invertují místo odstraní, je velmi důležitý krok, zabraňující nepatřičnému uvolnění restrikcí. Všude, kde je určená hodnota, tam jedna určená hodnota musí zůstat, i hodnoty bezvýznamné pro tuto modifikaci se musí zachovat, v tomto případě nuly (i invertované jedničky) v celé oblasti funkce.

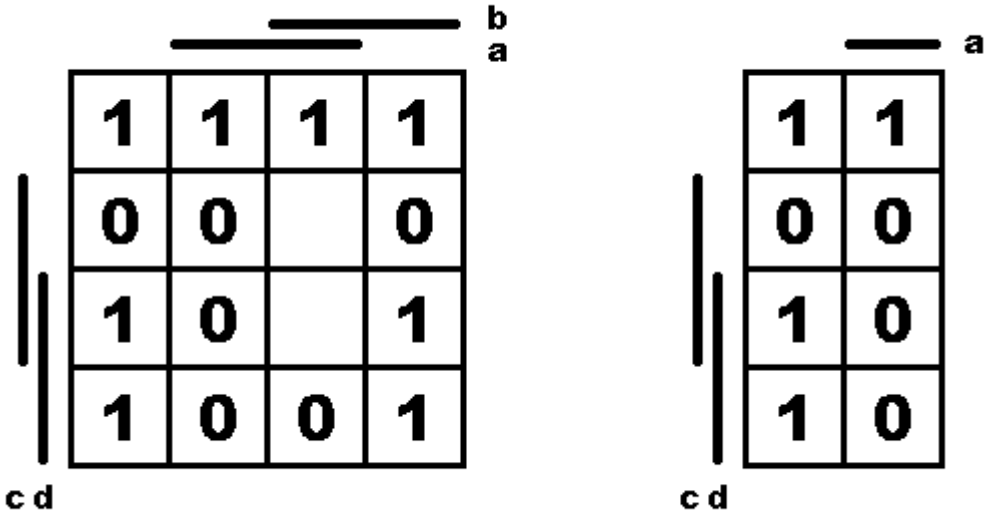
Poté se *všechny jedničky na straně kandidáta zpropagují i na druhou stranu*. Případné překážející nuly musí ustoupit. Podrobněji se touto problematikou zabývám na úrovni implementace v sekci 2.5.3.1. Technicky se na všech jedničkových termech objeví na místě proměnné DC, proměnná se tedy stane SV, a může se tedy odstranit.

1.7.4 Modifikace AV argumentu

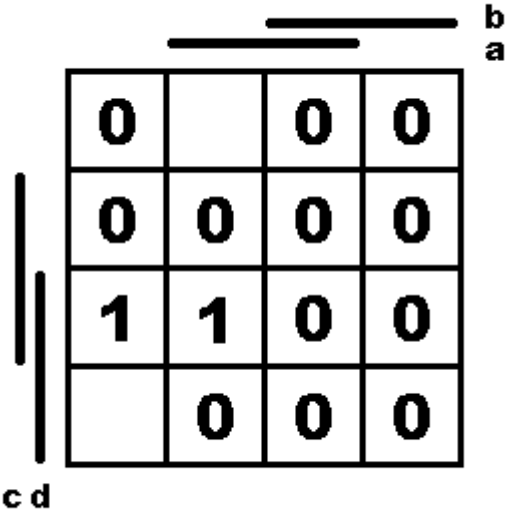
Liší se od předchozí v tom, že tentokrát se na druhé straně podle jedniček ze strany kandidáta invertuje, tj. provádí XOR. Celá strana kandidáta se vyplní nulami, což zajistí následné odstranění AV pomocí hradla AND. Opět z technického hlediska, tady se následně mohou všechny termy ze strany kandidáta odstranit, na rozdíl od SV modifikace, kde se nesmělo odstranit vůbec nic.



Obrázek 1.6: bidekompozice pomocí XOR - zdrojová funkce



Obrázek 1.7: bidekompozice pomocí XOR - argument SV



Obrázek 1.8: bidekompozice pomocí XOR - argument AV

Kapitola 2

Program

2.1 Použité technologie

2.1.1 Jazyk a prostředí

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

2.1.2 Framework

Při vývoji byl použit BOOM framework. Konkrétně v oblastech:

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

2.2 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.

2.3 Spouštění

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

2.4 Argumenty

```
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
```

Obrázek 2.1: argumenty programu

2.4.1 Popis

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.

2.4.2 Argument INPUT_FILE

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

2.4.3 Argument -o

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

2.4.4 Argument -a

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

2.4.5 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.

Jsou použita hradla:

- **XOR** - zde vždy 2-vstupý
- **AND** - minimálně 2-vstupý
- **NOT** - pouze u proměnné

2.4.6 Argument -r

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

2.4.7 Argument -t

Tento argument slouží pro testování (sekce 3), ale jeho označení je odvozeno od *timing* - časomíra. Po jeho použití se vypíší jen časomíry vybraných úseků výpočtu. Více v sekci 3.

2.4.8 Příklad použití

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

Obrázek 2.2: příklad spuštění s použitím všech argumentů

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

2.5 Proces zpracování dat

2.5.1 Datové typy

Kromě datových struktur BOOMu, reprezentujících jednotlivé termy nebo uzly networku, bylo použito jednoduchých datových typů. Jednotlivé termy jsou ukládány do typu *char*. Jako datová část pro uložení specifikace hodnoty jsou vyhrazeny nejnižší 2 bity. Byty nesou ještě speciální příznaky vztahující se k termu, které se nastavují a ruší v průběhu částí výpočtu. Více o příznacích pojednává sekce 2.5.5.

2.5.2 Abstraktní datové struktury

S cílem dosáhnout co nejvyšší efektivity byl pro fyzické uložení termů zvolen jednosměrně zřetězený spojový seznam datových bloků pevné délky (sekce 2.5.3.3). Tyto bloky mohou být opakovaně používány (sekce 2.5.3.2).

2.5.3 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

2.5.3.1 Třída XtPLA

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

- zjištění konzistence
- simplifikace
- bidekompozice

Algoritmus *zjištění konzistence* je kvadratický a je ho třeba provést jednou na začátku, než se začne funkce dekomponovat. Pokud zjistí kolizi nulových a jedničkových termů, nelze s funkcí pracovat. Dále je používán v testech odstraňování SV.

Simplifikace je také polynomiální a má dvě části. V první se detekují SV a AV. Oba typy se odstraňují, AV se zaznamenávají. Symetrická, a tedy přebytečná, proměnná má jediné kritérium: **Po jejím odstranění zůstane funkce nadále konzistentní**. Kritérium pro AV je mírně složitější a také její odstranění již není triviální. Adresní je proměnná v případě, že má konzistentní hodnotu ve všech jedničkových termech - tj. buď 0 nebo 1. Hodnota AV je pak inverzí této hodnoty. Při jejím odstranění se zároveň musejí odstranit všechny nulové termy z opačné (nulové) strany, je to důležité pro pozdější zjištění konstantnosti funkce.

Druhá část simplifikace redukuje počet termů tím, že hledá mezi jejich dvojicemi ty, kde jeden term je podmnožinou toho druhého. V tomto případě je nadbytečný a smaže se. Duplicita se detekuje stejným způsobem,

je to zvláštní případ podmnožiny. Dále se snaží termy spojovat a opakuje proces tak dlouho, dokud je co spojovat (sekce 2.7.1).

Bidekompozice se provádí s parametrem stejné struktury, jaký reprezentuje AV. Ovšem nejedná se o totéž. Zde se specifikuje proměnná a její hodnota, podle které se bude rozdělení provádět. Na této straně funkce chceme mít v jednom vstupu všechny termy, které se budou propagovat i na opačnou stranu, tato proměnná se tedy stane SV. U druhého vstupu budeme na opačné straně funkce invertovat termy v místech, kde jsou na protilehlé straně jedničkové termy. Ve výsledku, přesně podle teorie v sekci 1.7, dostaneme AV s hodnotou opačné strany funkce. XORem obou vstupů (při uvážení nulovosti druhého na opačné straně hodnoty AV) vznikne původní funkce. Může obsahovat méně DC, restriktivní popis je tedy jedině užší.

2.5.3.2 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 „na skladě“) 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.

2.5.3.3 Třída Block

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

2.5.3.4 Správa a přístup k datům

Přístup k termům vede přes pevně vestavěné iterátory. Ty odkazují na blok a pozici v něm, kde se term nachází. Jednotlivé termy jsou uloženy v bloku za sebou a jeden nikdy není rozdělen mezi dva bloky. Po ubrání jakékoliv proměnné se pořadí proměnných v termu změní tak, aby se díra zacelila. Vznikající mezery na konci termů zruší restrukturalizace, ke které dochází několikrát při simplifikaci funkce.

2.5.4 Algoritmus dekompozice - použité techniky

Zbývá popsat algoritmus dekompozice z nízkoúrovňového pohledu. Jak ve výpočtu SV, tak AV argumentu narážíme na problém propagace jedničko-

vých termů, resp. inverze termů, které mohou přinést do funkce nekonzistenci. Prosté odstranění kolizních termů nestačí, protože oblast, ve které nekolidují, přestane být termem pokrývána. Řešením je term dekomponovat podle propagovaného, který nově na místě musí být, a odstranit jen tu část jejich průniku, ostatní okolo ponechat.

2.5.4.1 Dekompozice termu

Provádí se vždy podle jiného termu. Úkolem je nalézt/vygenerovat minimální počet termů, které pokrývají oblast mimo vzorový term a zároveň samozřejmě leží v oblasti termu, z něhož se generují. Řešení problému, na první pohled náročného na představivost, je v překvapivě triviální. Stačí nalézt v generujícím termu všechny DC, které mají na odpovídajícím místě u vzorového termu specifikovanou hodnotu - tedy ne DC. Term se vygeneruje tak, že v kopii generujícího termu jeho DC hodnotu změním na inverzi hodnoty v termu vzorovém, a takto to udělám pro kažkou takovou dvojici. Že žádný z vygenerovaných termů se vzorovým nesousedí, vyplývá už z povahy algoritmu. Na jednom místě se vždy rozcházejí hodnotou. Tento postup má smysl samozřejmě pouze tehdy, mají-li oba výchozí termy nějaký průnik.

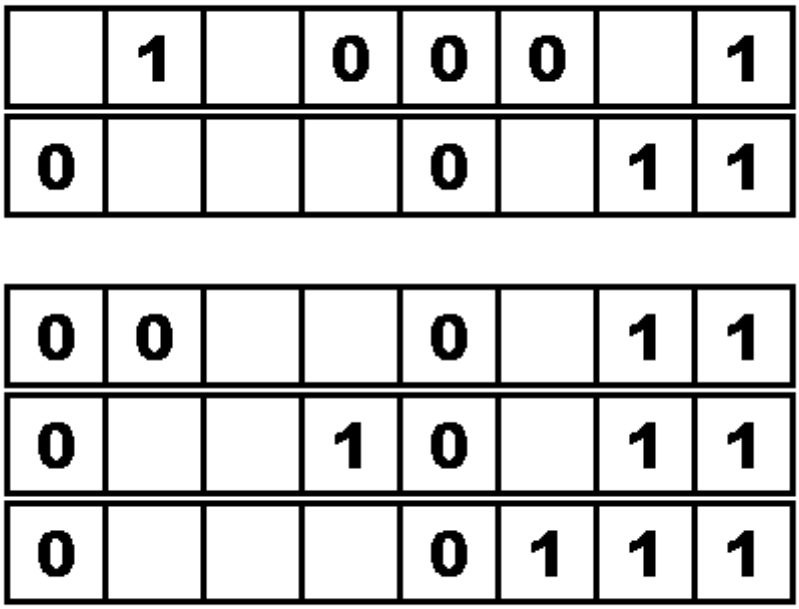
Na příkladu 2.3 je vzorový term z dvojice nahoře, pod ním term generující, tři vygenerované pod nimi.

2.5.5 Příznaky a oblasti jejich použití

Bit	Příznak	Popis
7	NOCUBE	smazaný term
6		volno
5		volno
4	NEWCUBE	nově přidaný term
3	IGNCUBE	ignorovaný term
2	IGNVAR	ignorovaná proměnná
1		restrikce pro 1
0		restrikce pro 0

Tabulka 2.1: tabulka příznaků v programu

Všechny příznaky kromě jednoho se ukládají do prvního bytu v termu. Příznak `IGNVAR` se vztahuje jako jediný k proměnné a vyskytuje se kdekoli v termu. Všechny příznaky kromě jednoho jsou vratné. Příznak `NOCUBE` je jako jediný absolutní, term se maže vždy nenávratně. Také všechny příznaky



Obrázek 2.3: příklad dekompozice termu podle vzoru

kromě `NOCUBE` jsou jen dočasné, pouze pro potřeby určitých částí výpočtu a poté se odstraňují.

2.5.5.1 Příznak NOCUBE

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

2.5.5.2 Příznak NEWCUBE

Používán pouze při výpočtu AV argumentu. Tímto příznakem se značí produkty dekompozice termu. Je to důležité proto, aby se poznalo, který z původních jedničkových termů se má ještě použít jako maska pro inverzi (na začátku známe pouze jejich počet, ten samotný i termy se dynamicky

mění). Je to daň jednoduché implementace, kde hlavní i vnořený cyklus pracují nad stejnými daty a vzájemně si je mění „pod rukama“.

2.5.5.3 Příznak IGNCUBE

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

2.5.5.4 Příznak IGNVAR

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

2.5.6 Algoritmus top-levelu

2.5.6.1 Popis

Ze všeho nejdříve je třeba načíst data. To zajišťuje BOOM, který zpracovává i typ vstupního PLA a případně i vypočítá komplement. Jádro programu XtPLA data převezme do svých struktur, vždy se pracuje s typem nulových a jedničkových termů. Každý výstup se zpracuje zvlášť. Zkontroluje se konzistence dat. Pak se vytvoří výstupní buffer, na který se postupně rekurzivně napojí celý budoucí obvod. Poté už se postupuje jen rekurzivně. Provede se simplifikace funkce, což zahrnuje i detekci konstanty. V případě překročení AV prahu nebo konstantnosti funkce se vytvoří příslušná forma vícevstupového ANDu, který AV odfiltruje. Zároveň se tak doplní názvy do rodičovského uzlu a ten se tak následně přidá do netlistu. Pokud funkce ještě není konstantou, provede se bidekompozice a každá větev se zpracuje zvlášť. Výše zmíněný typ rekurze je *preorder*.

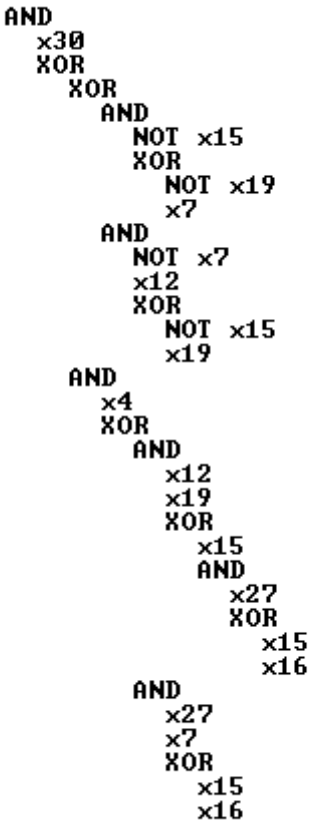
Po úplném vytvoření uzlů všech výstupů se network optimalizuje (sekce 2.7.3) a BOOM ho uloží do BLIFu.

2.5.6.2 Výběr kandidáta na bidekompozici

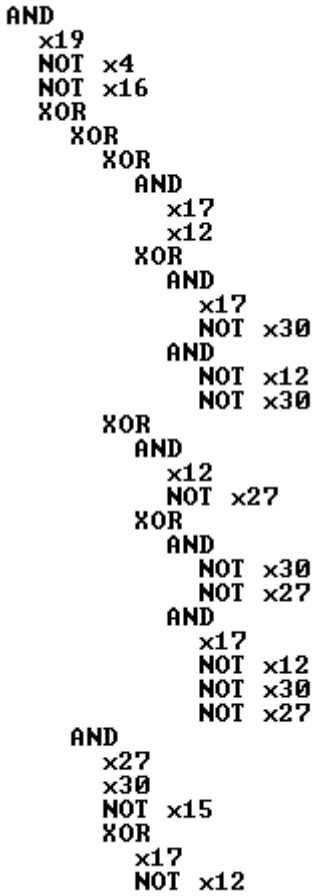
Výběr se děje náhodně. Dosáhne se tak snadno nedeterministicky různě kvalitních výsledků, z nichž je možné si vybírat. Dalším argumentem pro náhodný výběr je konstantní složitost. V předchozí verzi programu jsem věnoval výběru kandidáta jistou pozornost, až nakonec došel k závěru, že neexistuje dostatečně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

2.5.7 Vliv prahu pro AV na strukturu obvodu

Tři příklady, na kterých je vidět rozdílná struktura obvodu při použití jiného AV prahu. V prvním 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 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, kde je kombinovaná charakteristika.



Obrázek 2.4: struktura obvodu při použití AV prahu 1

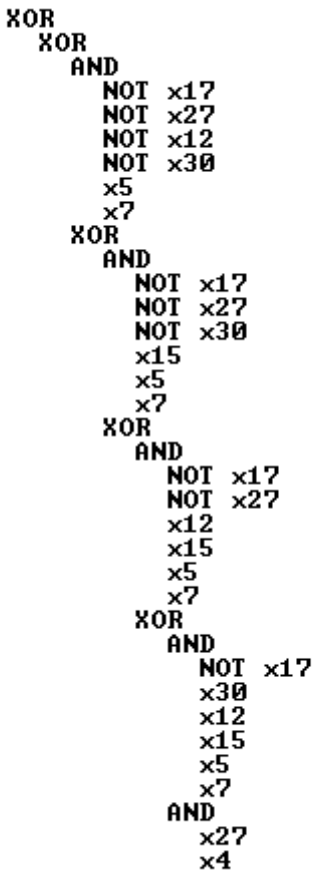


Obrázek 2.5: struktura obvodu při použití AV prahu 3 - default

2.6 Asymptotická složitost

Pokusím se zde odhadnout průměrné asymptotické složitosti důležitých částí programu. Některé části mají z principu jako nejhorší AS exponenciální, přesto ve většině případů je riziko sklouznutí k takovéto složitosti malé a skutečná průměrná AS může být vyjádřitelná např. jako polynomiální.

V dalším textu: v jako počet vstupních proměnných, n jako počet termů (celkem, bez rozlišení hodnoty), r pro nulové, f pro jedničkové. Vše se vztahuje k výpočtu jednoho výstupu, protože výstupy se zpracovávají nezávisle sekvenčně.



Obrázek 2.6: struktura obvodu při použití AV prahu většího než počet vstupních proměnných

2.6.1 AS zjištění konzistence

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

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

2.6.2 AS odstraňování a slučování termů

Pokud jde o odstraňování, máme zde opět stejnou kvadratickou složitost. Ale tento algoritmus se navíc opakuje po alespoň jednom sloučení dvou

termů do jednoho. Vyjdu z toho, jak rychle takto zjednoduší plně určenou funkci. Proces by vždy sloučil polovinu termů, ale s tímto úbytkem nelze univerzálně počítat, a zabralo by to v iterací. Vychází tedy:

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

2.6.3 AS odstraňování proměnných

Dominantní složku zde opět tvoří algoritmus zjištění konzistence.

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

2.6.4 AS bidekompozice - argument SV

Pokud by se počet termů dynamicky neměnil vlivem jejich dekompozice, výpočet by vypadal následovně: Podle jedničkových termů se procházejí nulové, což tvoří hlavní složku algoritmu. Dále se při každé dekompozici termu provádí optimalizační test na podmnožinu, což obnáší 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 termů. Nutno podotknout, že přesně kvůli tomuto nepříjemnému problému byla obohacena simplifikace redukcí termů o jejich spojování.

2.6.5 AS bidekompozice - argument AV

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

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

2.6.6 AS celková

Pokud by byl strom rekurze vyvážený a nedocházelo by k průběžnému odstraňování SV, dostali bychom hned AS exponenciální, se základem 2, podle počtu vstupních proměnných. Tak tomu naštěstí nebylo u žádné z testovaných funkcí. Vždy došlo k nepředpovídatelnému lokálnímu rozvětvení (na což neměla ani tak vliv náhodná volba kandidáta - sekce 2.5.4), které z globálního hlediska neslo velkou nevyváženost. Ta podle mého znesnadňuje bližší určení celkové AS.

2.7 Optimalizace

2.7.1 Optimalizace funkce před bidekompozicí

Je důležité mít co nejméně termů. 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“ obou množin termů. Děje se tak prostým způsobem boolského příznaku, že v dané iteraci došlo ke spojení, a na konci se vyhodnotí, zda se iterace bude celá opakovat. Protože po nalezení prvního spojení se iterace předčasně neukončí, ale pokračuje i s pozměněnými daty, dává se tak prostor zpracování spojení i po větších skupinách.

2.7.2 Optimalizace bidekompozice

Během bidekompozice se v obou výpočtech používá funkce generující dekomponované termy. Před přidáním se přesvědčí, zda není vygenerovaný term již **podmnožinou** nějakého jiného. V tomto případě term nepřidá. Tato optimalizace má největší podíl na vyrovnávání časů mezi jednoduchými a složitými výpočty, kde za cenu zvýšení stupně polynomu se dosáhne reálného času i paměťových nároků v některých případech, kde by bez této úpravy došlo k exponenciálnímu průběhu výpočtu.

2.7.3 Optimalizace výsledného networku

XORy, které mají na jednom vstupu jedničku (na obou ji z povahy výpočtu mít nemohou), se změní na invertory. Nulová konstanta nemůže být na vstupu XORu ani ANDu, pouze v případě, že celá funkce je nulovou konstantou. Opět to vyplývá z povahy výpočtu - algoritmus top-levelu, sekce 2.5.6.

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

Kapitola 3

Testování

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 networku.

3.1 hula

hula

Závěr

Příloha **A**

Seznam použitých zkratek

XOR-bidex XOR bidecomposition - název programu

PLA Programmable logic array - vstupní formát souboru

BLIF Berkeley logic interchange format - výstupní formát souboru

BOOM Boolean minimizer, BOOM framework

AV Address variable - adresní proměnná

SV Symmetric variable - symetrická proměnná

DC Don't care

AS Asymptotická složitost

Příloha **B**

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
  - `thesis.pdf` ..... text práce ve formátu PDF
  - `thesis.ps` ..... text práce ve formátu PS