České vysoké učení technické v Praze

Fakulta elektrotechnická

Bakalářská práce

Mapování logických obvodů do FPGA

Miroslav Karšulín

Vedoucí práce: Ing. Petr Fišer, Ph.D.

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

červenec 2008

Poděkování

Chtěl bych poděkovat Ing. Petru Fišerovi, Ph.D. za jeho odborné vedení a pomoc, kterou mi v průběhu celé práce poskytoval.

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů·(autorský zákon).

V Praze dne ………………………….. …………………………..

Abstract

The goal of this project is with usage known algorithms to implement an algorithm for mapping logical circuits into FPGA, concretely to decompose circuit into network of LUTs (look-up tables). The main obtain this decomposition is to minimize a delay of the circuit, i.e. to minimize number of LUTs on a critical path.

Abstrakt

Cílem této práce je s použitím známých algoritmů naprogramovat nástroj pro mapování logických obvodů do FPGA, konkrétně dekomponovat obvod do sítě LUTů (look-up tables). Hlavním záměrem této dekompozice je minimalizovat zpoždění obvodu, tj. minimalizovat počet LUTů na kritické cestě.

Obsah

Obsah ... xi

Seznam obrázků ... xiii

Seznam tabulek ... xv

1 Úvod ... 1

1.1 Obecný úvod ... 1

1.2 Cíl práce ... 1

2 Analýza a specifikace požadavků ... 3

2.1 Požadavky na systém ... 3

2.2 Analýza stávajících algoritmů ... 3

2.3 Důvody pro hledání vlastního algoritmu ... 4

3 Základní pojmy ... 5

3.1 Definice základní pojmů ... 5

3.2 Zpoždění ... 6

4 Převod obecné sítě na K-ohraničenou síť ... 7

4.1 Důvod převodu ... 7

4.2 Jednoduchá dekompozice pomocí vyváženého stromu ... 8

4.3 DMIG algoritmus ... 9

5 Mapovací algoritmus s využitím řezových množin ... 11

5.1 Obecný postup ... 11

5.2 Řezový výpočet ... 11

5.3 Zajištění minimálního zpoždění ... 14

5.4 Pseudokód řezového výpočtu ... 15

5.5 Paměťová náročnost výpočtu ... 15

5.6 Mapování řezů na LUTy ... 16

5.7 Plošná optimalizace ... 17

5.8 Optimalita minimální hloubky ... 18

5.9 Techniky pro zlepšení mapování ... 20

5.9.1 Použití značení ... 20

5.9.2 Redukce paměťové náročnosti ... 21

5.10 Zhodnocení této techniky ... 21

6 Vlastní algoritmus mapování na technologii LUTů ... 23

6.1 Primitivní algoritmus ... 23

6.2 Vylepšení primitivního algoritmu ... 25

7 Vytváření pravdivostních tabulek ... 27
7.1 Způsob vytváření ... 27
8 Popis implementace ... 29
8.1 Výběr programovacího jazyka ... 29
8.2 Datové struktury a metody ... 29
8.2.1 Třída Import ... 29
8.3 Třída Gate ... 30
8.4 Třída Lut ... 31
8.5 Třída Mapper ... 31
8.6 Třída Export_LUTs ... 31
9 Testování ... 32
10 Závěr ... 45
11 Použitá literatura ... 47
A. Seznam použitých zkratek ... 49
B. Uživatelská příručka ... 51
C. Obsah přiloženého CD ... 53

Seznam obrázků

Obr. 4.1.1 Transformace 4-vstupového hradla na hradlo s 2 vstupy. 8

Obr. 5.2.1 Ukázková síť pro výpočet řezů 12

Obr. 5.2.2 Dvě možná řešení namapování na LUTy 14

Obr. 5.5.1 Vyjádření závislost počtu řezů na uzel 16

Obr. 5.7.1 Ukázka duplikace hradel 17

Obr. 5.8.1 Monotónnost grafu 18

Obr. 5.8.2 Příklad sítě u které nedochází ke generování optimálního řešení 19

Obr. 6.1.1 Příklad obvodu na kterém postup nefunguje 24

Obr. 6.1.2 Ukázka namapování primitivním algoritmem 25

Obr. 6.2.1 Porovnání algoritmu primitiv a primitiv_better 26

Obr. 7.1.1 Ukázkový obvod pro výpočet pravdivostní tabulky 27

Seznam tabulek

Tabulka 8-1 Důležité metody třídy Import 29

Tabulka 8-2 Důležité proměnné třídy Gate 30

Tabulka 8-3 Důležité metody proměnné třídy Gate 30

Tabulka 8-4 Důležité proměnné třídy Lut 31

Tabulka 8-5 Důležité proměnné třídy Mapper 31

Tabulka 8-6 Důležité metody třídy Export_LUTs 31

Tabulka 9-1 Srovnání hloubky mapování malých obvodů pro K=3 33

Tabulka 9-2 Srovnání hloubky mapování malých obvodů pro K=4 34

Tabulka 9-3 Srovnání hloubky mapování malých obvodů pro K=5 35

Tabulka 9-4 Srovnání hloubky mapování malých obvodů pro K=6 36

Tabulka 9-5 Srovnání počtu LUTů mapování malých obvodů pro K=3 37

Tabulka 9-6 Srovnání počtu LUTů mapování malých obvodů pro K=4 38

Tabulka 9-7 Srovnání počtu LUTů mapování malých obvodů pro K=5 39

Tabulka 9-8 Srovnání počtu LUTů mapování malých obvodů pro K=6 40

Tabulka 9-9 Výsledky mapování malých obvodů pro K =10 41

Tabulka 9-10 Srovnání hloubky mapování velkých obvodů pro K=6 42

Tabulka 9-11 Srovnání počtu LUTů mapování velkých obvodů pro K=6 43

Tabulka 9-12 Srovnání časů mapování velkých obvodů pro K=6 44

1 Úvod

1.1 Obecný úvod

Programovatelná hradlová pole se díky jejich ceně, rychlému návrhu a jejich možnostem implementovat logické funkce stávají v poslední době velmi populárními. FPGA architektury založené na LUTech v současné době dominují programovatelným chipům. LUT (look-up table) je vpodstatě klasická pravdivostní tabulka, která definuje výstup hradla na základě jejich vstupních proměnných. Každý K-LUT, kde K udává počet vstupů, je tedy schopen implementovat libovolnou funkci z 2^{2^K} funkcí.

1.2 Cíl práce

Protože programovatelné obvody obvykle mívají delší zpoždění než klasická zařízení a celkové zpoždění obvodu má také velký vliv na rychlost a výkonnost zařízení, bude cílem této práce převést libovolný kombinační obvod do sítě K-LUTů s ohledem právě na minimální zpoždění obvodu. Tzn. budu se snažit minimalizovat zpoždění na kritické cestě obvodu.

V této práci se budu zabývat některými již existujícími algoritmy mapování na technologii FPGA (konkrétně LUTů). Kromě toho uvedu i vlastní postup. Níže uvedené algoritmy naimplementuji, otestuji a výsledky srovnám s již existujícími systémy jako je ABC [1] a SIS [2].

2 Analýza a specifikace požadavků

2.1 Požadavky na systém

Jak již bylo zmíněno, hlavním cílem je namapovat libovolný kombinační obvod sítě na technologii FPGA, konkrétně LUTů s minimální hloubkou sítě. Vstupem bude kombinační obvod zadaný netlistem ve formátu bench a výstupem bude soubor pravdivostních tabulek pro jednotlivé LUTy ve formátu blif.

Mým záměrem bude namapovat kombinační obvod na LUTy, s libovolný počtem vstupů K. Budu se snažit, abych byl schopen namapovat i ty největší obvody. Výsledky otestuji na testovacích obvodech a vzájemně porovnám výsledky jednotlivých algoritmů.

Jedním z důvodů, proč se snažím o minimální zpoždění obvodu je budoucí záměr využití při simulaci velkých obvodů. Pro velké obvody jsou simulace poměrně časově náročné. Namapováním na K-LUTy dojde k významnému zmenšení zpoždění a možnému urychlení simulace. Protože však mapování na LUTy také nějakou dobu trvá, bude potřeba zjistit, zda-li je tento postup výhodný nebo ne. Toto je však mimo rozsah této práce.

2.2 Analýza stávajících algoritmů

Mapovací algoritmy lze rozdělit na strukturální a funkcionální. Strukturální mapovací algoritmy považují obvodový graf jako daný a hledají pokrytí grafu K-vstupovým podgrafem korespondující s LUTy. Funkcionální algoritmy provádí booleovskou dekompozici logických funkcí do subfunkcí, které jsou limitovány podporovanou velikostí realizovatelných LUTů.

V praxi se pro mapování na technologii FPGA používají především strukturální algoritmy, protože funkcionální jsou mnohem více časově náročné a jsou tím omezeny na menší obvodové návrhy. Protože cílem této práce je schopnost namapovat libovolně velký obvod pro libovolné K, budu se dále zabývat strukturálními mapovacími algoritmy.

Existující strukturální mapovací algoritmy se dělí podle hlavního cíle algoritmu:

- **Plošná minimalizace –** zaměřují se zejména na minimální počet celkového počtu LUTů. Tyto algoritmy jsou pro řešení obecných obvodů jako jsou např.

orientované acyklické grafy (DAG) a nalezení optimálního řešení NP-těžké. Patří sem např. algoritmy: Chortle-crf[7], MIS-pga, XMap, VisMap, TechMap.

- **Minimalizace zpoždění –** tyto algoritmy se zaměřují zejména na minimalizaci počtu LUTů na kritické cestě. Patří sem algoritmy: Chortle-d, MIS-pga-delay, TechMap-L, Dag-Map[3], FlowMap[6], EdgeMap.

- **Energetická minimalizace –** snaží se minimalizovat výsledné energetické nároky. Tento problém je také NP-těžký. Patří sem algoritmy: PowerMap, PowerMinMap, Emap, DVmap.

Toto je pouze hrubé rozdělení mapovacích algoritmů. V praxi často dochází k jejich kombinaci. Asi nejvíce používané je mapování logického obvodu s minimálním zpožděním a poté pod tímto omezením vypočítané minimální hloubky je prováděna plošná minimalizace. Příkladem této kombinace je např. algoritmus DAOmap [3] a algoritmus implementovaný v systému ABC[1], který z něj vychází.

V této práci se tedy budu zabývat zejména algoritmy minimalizující zpoždění. Po podrobnějším prostudování jednotlivých uvedených algoritmů jsem zjistil, že všechny vychází ze stejného základního algoritmu. Proto jsem se rozhodl jej důkladněji nastudovat a také vyzkoušet.

2.3 Důvody pro hledání vlastního algoritmu

Hlavní důvod pro vytvoření vlastního algoritmu byl ten, že požaduji schopnost namapovat obvod pro libovolné K. Protože však studované algoritmy zabývající se minimální hloubkou založené na prezentovaném základním algoritmu obvykle končí u K=6, rozhodl jsem se hledat jiné řešení, které by zvládlo větší K i za cenu mírného zhoršení kritické cesty obvodu.

3 Základní pojmy

3.1 Definice základní pojmů

K-LUT je základním elementem technologie FPGA. Můžeme si ji představit jako klasickou pravdivostní tabulku o K vstupech a 1 výstupu.

Booleovská síť může být reprezentována jako **orientovaný acyklický graf** (DAG), kde každý **uze**l reprezentuje logické hradlo a **orientovaná hrana (*i*, *j*)** existuje, jestliže výstup hradla *i* je vstupem hradla *j*.

Uzel má žádnou nebo více vstupních hran a žádnou nebo více výstupních hran. Každý uzel má své jedinečné ID.

Primární vstupní uzel (PI) nemá vcházející hrany.

Primární výstupní uzel (PO) nemá vycházející hrany.

Označení $input(v)$ použiji k označení množiny uzlů, které jsou vstupy hradla v. Je to množina všech uzlů sítě, které jsou dosažitelné skrz vstupní hrany uzlu v.

Je dána booleovská síť N, použiji O_v k označení podgrafu s kořenem v uzlu v. O_v je podsíť sítě N skládající se z uzlu v a některých jeho předchůdců tak, že pro každý uzel $w \in O_v$ existuje cesta z w do v, která celá leží v O_v. Maximální podgraf uzlu v je takový podgraf, který se skládá ze všech primární vstupů předchůdců uzlu v a značím jej F_v.

Označení $input(O_v)$ používám k označení uzlů mimo O_v, které se připojují na vstupy hradel v O_v.

Síť je **K-ohraničená**, jestliže počet vstupů každého uzlu nepřesáhne K.

Řez $cut(X)$ uzlu x je množina uzlů sítě tak, že každá cesta z primárních vstupů do uzlu x prochází skrz nejméně jeden uzel obsažený v množině $cut(X)$.

Triviální řez uzlu je řez skládající se z uzlu samotného.

Řez nazveme **K-vhodný**, jestliže počet uzlů v něm nepřesáhne číslo K.

Řez nazveme **dominující**, jestliže osahuje jiný řez stejného uzlu.

Hloubka uzlu je délka nejdelší cesty z kteréhokoliv primárního vstupu do uzlu. Samotný uzel se do délky cesty započítává, ale primární uzly se nepočítají. (tzn. primární uzly mají úroveň 0).

Hloubka sítě je největší úroveň uzlu x výsledného mapovaného obvodu.

Plocha výsledného mapovaného řešení je počítána jako počet výsledných LUTů.

3.2 Zpoždění

Je dobré specifikovat, kde může nastat zpoždění obvodu. U logického obvodů nastává zpoždění na jednotlivých hradlech. Zpoždění hradla je závislé na typu hradla a počtu jeho vstupů. Výsledkem této práce však nebude logický obvod složený z hradel, ale obvod skládající se z LUTů. Zde je situace trochu odlišná. Zpoždění nastává na spojeních mezi LUTy a na LUTech samotných. Protože však rozvrhové informace FPGA nejsou dostupné během mapovací fáze, modeluji každou spojovou hranu s pevným konstantním zpožděním. Protože přístupový čas každého K-LUTu je nezávislý na implementované funkci, aproximuji největší mapovací zpoždění obvodu jednotkovým zpožďovacím modelem. Tzn., že každý LUT přispívá jednou jednotkou zpoždění.

4 Převod obecné sítě na K-ohraničenou síť

4.1 Důvod převodu

Před vlastním mapováním na technologii FPGA je potřeba převést obecnou síť na síť, která je vhodná pro mapování. Popíši zde důvod tohoto předprocesního kroku a různé způsoby provedení.

Uvažujeme obecný logický obvod, tj. obvod ve kterém se vyskytují hradla (AND, OR, NAND, NOR, XOR, XNOR) s libovolným počtem vstupů. Jeho převod na K-ohraničenou síť je nutný, pokud chceme daný obvod namapovat na K-vstupové LUTy. Intuitivně přímé mapování logického 4-vstupového hradla na 3-vstupový LUT nelze provést bez dekompozice hradla a vyjádření funkce hradla minimálně dvěma hradly. Je dokázáno, že libovolnou booleovskou síť lze vyjádřit pouze pomocí 2-vstupových hradel (i s pomocí pouze základních hradel).

První krok který je potřeba provést je nahrazení komplexních hradel pomocí základních hradel. Zde uvažujeme komplexní hradlo XOR. Pokud vezmeme v úvahu, že XOR je vpodstatě parita, pak je lze nahradit stromem se stejnými hradly podobně jako hradlo AND. Obdobně dekomponuji i hradlo XNOR.

Převod obecné sítě se obvykle nepřevádí na K-ohraničenou síť s libovolným K, ale zpravidla na 2-ohraničenou síť a to zejména ze 2 důvodů:

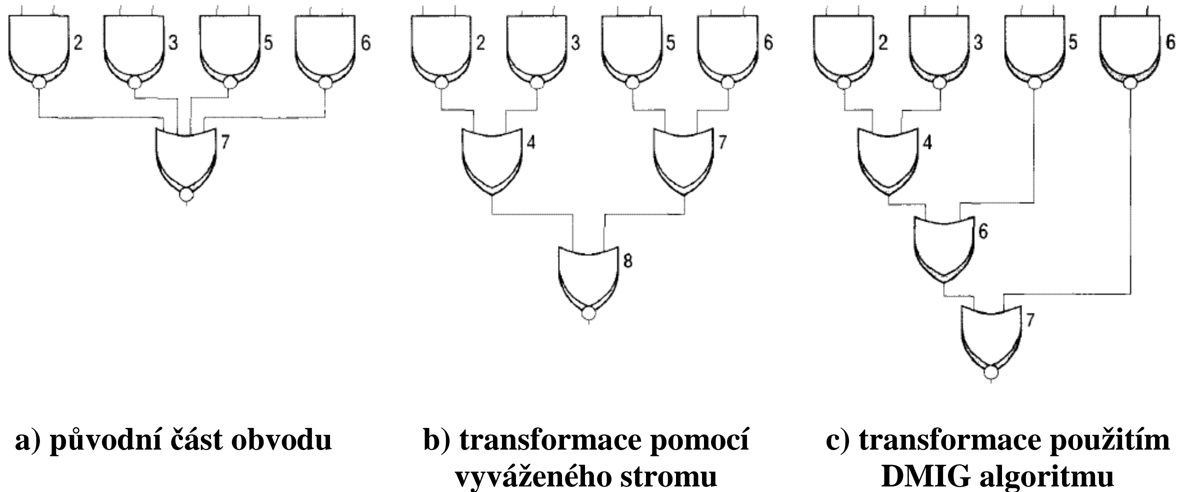
- Chceme omezit počet vstupů na jedno hradlo. Počet vstupů na jedno hradlo musí být menší nebo rovno K, abychom nemuseli dekomponovat toto hradlo během mapování.
- Pokud se zamyslíme nad FPGA technologií mapování jako proces zabalování logických hradel do K-LUTů, intuitivně je vidět, že čím menší hradla jsou tím snadněji padnou do K-LUTů a nedochází tak k přílišnému plýtvání prostoru v každém K-LUTu.

Vlastní převod vícevstupového hradla lze provést jeho náhradou odpovídajících hradel stromovou strukturou následovně:

Výstupní hradlo zůstává stejné, jako vícevstupové hradlo.

Pokud je dekomponované hradlo AND nebo NAND, nově vznikající hradla ve stromu jsou typu AND. Pokud je dekomponované hradlo OR nebo NOR nově vznikající hradla ve stromu jsou typu OR. Při dodržení těchto pravidel nedojde k narušení funkční ekvivalentnosti původního a nově vzniklého obvodu.

Je však zřejmé, že způsob vytváření ekvivalentního stromu vícevstupového hradla ovlivňuje zpoždění mapovaného obvodu a tedy celkové zpoždění výsledného řešení mapování logických obvodů do LUTů.



a) původní část obvodu **b) transformace pomocí vyváženého stromu** **c) transformace použitím DMIG algoritmu**

Obr. 4.1.1 Transformace 4-vstupového hradla na hradlo s 2 vstupy.

Způsobů jak převést obecnou síť na K-ohraničenou existuje opět několik. Na Obr. 4.1.1 a) je původní obvod. Čísla jednotlivých hradel označují hloubku jednotlivých hradel v obvodu. Na Obr. 4.1.1 b),c) jsou zobrazeny dvě možné varianty jak dekomponovat logické hradlo.

4.2 Jednoduchá dekompozice pomocí vyváženého stromu

Jednoduchý způsob jak převést n-vstupový uzel sítě na dvouvstupový uzel je nahradit každé n-vstupové hradlo ($n \geq 3$) vyváženým binárním stromem. Při pohledu na Obr. 4.1.1 b), kde jednotlivá čísla označují hloubku hradla v síti, vidíme, že celková hloubka obvodu se zvýšila z hloubky 7 na 8. Lze ovšem nalézt lepší řešení, jak nahradit tuto část obvodu, což popisuje následující algoritmus.

4.3 DMIG algoritmus

Z obrázku Obr. 4.1.1 c) je patrné, že převzatý algoritmus [3] umožňuje pro některé obvody generovat lepší řešení. Čísla na obrázku označují hloubku jednotlivých hradel v obvodu. Pokud porovnáme výsledek s původní částí obvodu na Obr. 4.1.1 a) zjistíme, že se hloubka obvodu nezměnila.

Transformace sítě G do 2-vstupové sítě G' probíhá následovně. Procházíme uzly v síti G v topologickém pořadí. Začneme od primárních uzlů. Pro každý vícevstupový uzel v, zkonstruujeme binární strom $T(v)$ s kořenem v uzlu v s využitím algoritmu, který je podobný Huffmanovu kódování pro zkonstruování prefixového kódu o minimální průměrné délce. Intuitivně při konstrukci binárního stromu $T(v)$ chceme, aby se nejdříve kombinovaly uzly s nejmenší hloubkou. Podmínkou pro tento algoritmus je průchod v topologickém uspořádání. To znamená, že při nahrazování vícevstupového uzlu v binárním stromem je nutné aby všechny jeho vstupní vícevstupové uzly $input(v) = \{u_1, u_2, u_3, ..., u_m\}$ byly již nahrazeny binárním stromem. Při každém zpracování uzlu je vypočítána nová hloubka všech uzlů ve vzniklém binárním stromě $T(v)$ s využitím hloubky vstupních uzlů $depth(u_i)$ $(1 \le i \le m)$.

Vytvoření binárního stromu $T(v)$ vícevstupového hradla v popisuje následující pseudokód:

ALGORITMUS *DMIG*

$V = input(v) = \{u_1, u_2, ..., u_m\}$;

WHILE $|V| > 2$ ***DO***

BEGIN

VYBER u_i *A* u_j*, KTERÉ MAJÍ NEJMENŠÍ HLOUBKU;*

VYTVOŘ NOVÝ UZEL x*;*

$input(x) = \{u_i, u_j\} \cup \{x\}$

END-WHILE;

POUŽIJ POSLEDNÍ 2 UZLY, KTERÉ ZŮSTALY VE V*, JAKO VSTUPY UZLU* v*;*

VRAŤ BINÁRNÍ STROM $T(v)$ *S KOŘENEM V UZLU* v*.*

END-ALGORITMU

Tento algoritmus zaručuje, že hloubka nově vzniklé sítě $depth(G')$ nebude větší než $\log(2d).depth(G) + \log(I)$, kde d je maximální stupeň výstupů v G a I je počet primární vstupních uzlů. Nedůvěřivý čtenář je odkázán na literaturu[3], kde je uveden důkaz tohoto vztahu.

5 Mapovací algoritmus s využitím řezových množin

Hlavními zdroji tohoto výpočtu jsou literatury [3] [4] [8] uvedené v seznamu literatury.

5.1 Obecný postup

Protože naprostá většina studovaných strukturální algoritmů je založena na stejném základě, uvedu nejdříve obecný postup. Ten je možné shrnout do několika bodů:

- převod obecné sítě na K-ohraničenou síť (tento bod v literatuře je obvykle opomíjen a předpokládá již K-ohraničenou síť)
- řezový výpočet
- mapování řešení do LUTů s ohledem na optimální zpoždění
- plošná minimalizace s použitím heuristik
- záznam výsledné LUT sítě.

Jednotlivé body si následně rozebereme podrobněji.

5.2 Řezový výpočet

Touto fází začíná strukturální mapování na technologii K-LUTů. Probíhá zde výpočet množin K-vhodných řezů na jednotlivých vnitřních 2-vstupových uzlech předmětného grafu. Každý řez potom bude odpovídat jedné možné implementaci K-LUTu.

Uvedu zde standardní způsob výpočtu řezů, na kterém je většina strukturálních mapovacích algoritmů založena.

Nechť A a B jsou 2 množiny řezů. Potom definujeme operaci $A \Diamond B$ takto:

$$A \Diamond B = \{u \cup v \mid u \in A, v \in B, |u \cup v| \leq K\}. \quad 5.2.1$$

, kde u jsou uzly v řezu A

v jsou uzly v řezu B

K je maximální počet vstupů jednoho LUTu.

Jinými slovy se jedná o kartézský součin množin s odfiltrováním těch řezů, jejíž počet uzlů v něm přesahuje číslo K.

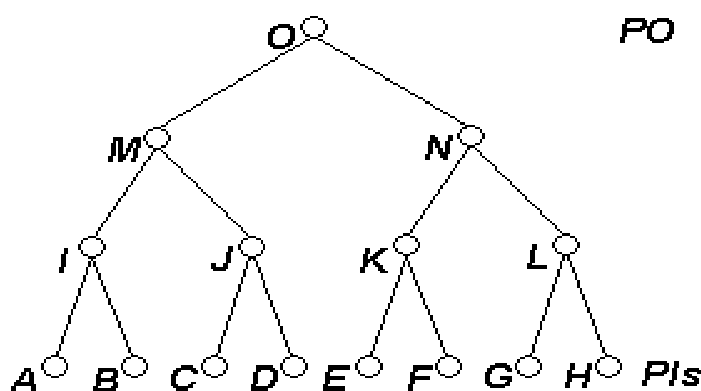
Nechť $\Phi(n)$ označuje množinu K-vhodných řezů uzlu n. Tuto množinu K-vhodných řezů uzlu n spočteme dle následujícího vztahu:

$$\begin{aligned} \Phi(n) &= \{\{\{\{n\}\}\} : n \in PI\} \\ \Phi(n) &= \{\{\{n\}\} \cup [\Phi(n_1) \Diamond \Phi(n_2)] : jinak\} \end{aligned} \qquad 5.2.2$$

Pokud je uzel primárním vstupem sítě, potom jeho množina řezů obsahuje pouze jeden jednoprvkový řez, který obsahuje uzel samotný.

Pokud je uzel vnitřním uzlem sítě, potom jeho množina řezů je spočítán jako kartézský součin řezů jeho vstupních uzlů, který je omezen tím, že počet uzlů v řezu nesmí přesáhnou číslo K a přidáním triviálního řezu (obsahující uzel samotný). Jinými slovy, dochází ke kombinaci každého řezu z jednoho vstupního uzlu s každým protějším řezem z druhého vstupního uzlu. Jestli nově vzniklý řez přesáhne počet uzlů K, řez je zahozen.

Takto nově vzniklé řezy označujeme jako K-vhodné, protože je lze přímo namapovat na K-LUTy. Řezy, které obsahají více než K uzlů jsou pro mapování sítě na K-LUTy již dále nepodstatné a proto je můžeme zahodit. Zahazovat tyto řežy je nutné hned během výpočtu, protože by nadále zpomalovaly běh algoritmu a zvyšovali paměťové nároky. Počet uzlů v řezu je závislý na počtu vstupu LUTu.



Obr. 5.2.1 Ukázková síť pro výpočet řezů

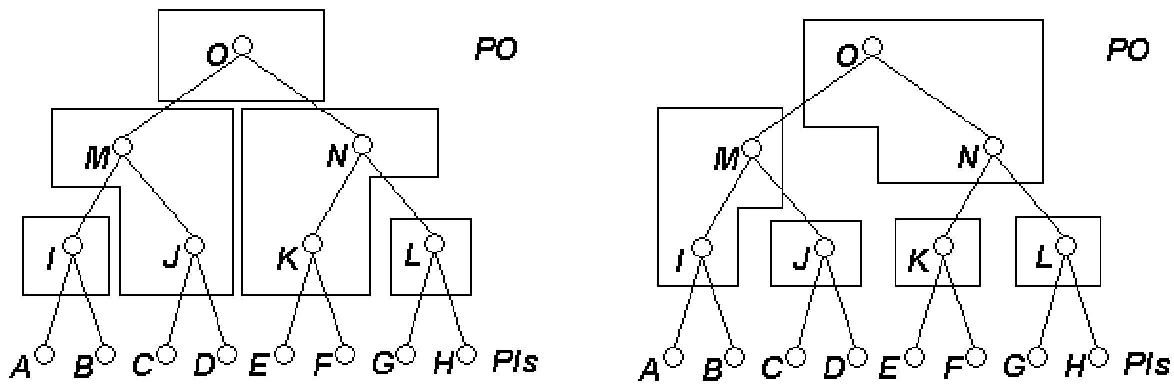
Výpočet řezů si ukážeme na příkladu. Uvažujme síť, která je zobrazena na Obr. 5.2.1 a $K = 3$. Primárním vstupy sítě jsou uzly A, B, C, D, E, F, G, H. Jejich množina řezů bude tedy obsahovat jedinou jednoprvkovou množinu s uzlem samotným. Označme množinu řezů jako $cuts(X)$, kde X je uzel.

$$cuts(A) = \{\{A\}\};$$
$$cust(B) = \{\{B\}\};$$
$$cuts(C) = \{\{C\}\};$$
$$cuts(D) = \{\{D\}\};$$
$$cuts(E) = \{\{E\}\};$$
$$cuts(F) = \{\{F\}\};$$
$$cuts(G) = \{\{G\}\};$$
$$cuts(H) = \{\{H\}\}$$

Následným průchodem sítě v topologickém uspořádání k primárnímu výstupu O a aplikací vztahu 5.2.2 a využitím vztahu 5.2.1 získáme následující řezy uzlů:

$$cuts(I) = \{\{I\},\{A,B\}\};$$
$$cust(J) = \{\{J\},\{C,D\}\};$$
$$cuts(K) = \{\{K\},\{E,F\}\};$$
$$cuts(L) = \{\{L\},\{G,H\}\};$$
$$cuts(M) = \{\{M\},\{I,J\},\{I,C,D\},\{A,B,J\}\};$$
$$cuts(N) = \{\{N\},\{K,L\},\{G,H,K\},\{E,F,L\}\};$$
$$cuts(O) = \{\{O\},\{M,N\},\{M,K,L\},\{I,J,N\}\};$$

Všimněte si, že např. u uzlu M jsme zahodili řez $\{A,B,C,D\}$. Výslednou funkci O jsme tedy schopni namapovat na LUTy několika způsoby. Na Obr. 5.2.2 jsou zobrazeny dvě možná mapovací řešení v závislosti na výběru řezu.



Obr. 5.2.2 Dvě možná řešení namapování na LUTy

Hlavním problémem není spočítání všech K-vhodných řezů, ač je to pro velké návrhy jak paměťově tak časově náročné, ale největším úskalím je vybírání správných řezů. Na této části totiž spočívá kvalita výsledného mapování.

5.3 Zajištění minimálního zpoždění

Prvním krokem pro zajištění minimálního zpoždění bylo vhodně převést síť na K-ohraničenou síť tak, aby hloubka obvodu vzrostla co nejméně. Dalším krokem je vhodně namapovat tuto K-ohraničenou síť na K-LUTy, tak aby zpoždění bylo minimální.

Toho může být docíleno již během řezového výpočtu, kdy se průběžně počítá dosahovaný čas (zpoždění). Během výpočtu nemusí být získáván pouze dosahovaný čas, ale i mapovací plocha, jako je tomu např. u algoritmu DAOmap[4].

Dosahovaná hloubka primárních vstupních uzlů je 0. Dosahovaná hloubka se nadále rozšiřuje skrze vypočítávané řezy z primárních vstupů do primárních výstupů, kde každý řez(LUT) na cestě reprezentuje 1 jednotkové zpoždění. K získání minimálního příchozího času pro uzel sítě definujeme následující vztah:

$$depth(v) = \min_{\forall C \in v} \left[\max_{i \in input(C)} \left(depth(i) \right) + 1 \right] \qquad 5.3.1$$

,kde C reprezentuje každý řez vygenerovaný pro uzel v pomocí řezového výpočtu a $depth(i)$ je minimální příchozí čas na vstupní signál v řezu C.

Dle tohoto vztahu je vypočítána hloubka všech uzlů. Po provedení řezového výpočtu už je známa výsledná hloubka mapovaného řešení. Ta je rovna maximální hloubce libovolného uzlu v síti, tedy maximální hloubce z primárních výstupů.

Za povšimnutí stojí, že uzel může obsahovat více řezů s minimální hloubkou. Výběr libovolného z nich už však neovlivní výslednou hloubku mapovaného obvodu, ale může výrazně ovlivnit počet LUTů ve výsledném řešení.

5.4 Pseudokód řezového výpočtu

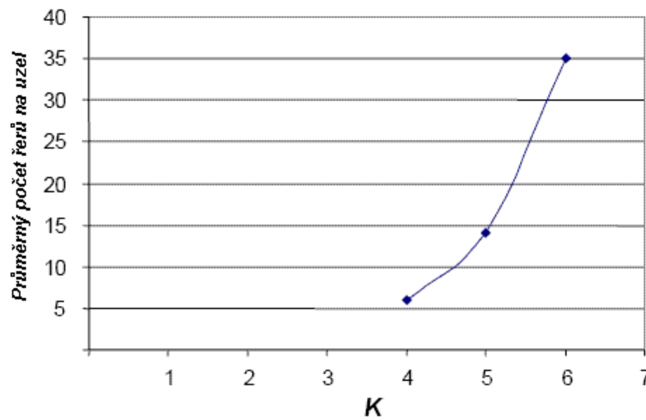
Celý algoritmus řezového výpočtu lze vyjádřit pomocí jednoduché procedury, která v jednom topologickém průchodu počítá dle vztahu 5.2.2 všechny K-vhodné řezy všech uzlů sítě a dle vztahu 5.3.1 hloubku řezů a tedy i hloubku výsledného řešení. Tuto proceduru můžu vyjádřit tímto pseudokódem.

```
ALGORITMUS SPOČTI_ŘEZY
 FOR KAŽDÝ PRIMÁNÍ UZEL v DO
  h(v) = 0;
 T =LIST PRIMÁRNÍCH UZLŮ V TOPOLOGICKEM POŘADÍ;
 WHILE T NENÍ PRÁZDNÉ DO
....BEGIN
  ODSTRAŇ PRVNÍ UZEL v Z T;
  SPOČTI ŘEZY UZLU v POMOCI ŘEZŮ PŘEDCHŮDCŮ
  h(v) = MIN ZE VŠECH ŘEZŮ (MAX(HLOUBKA UZLŮ OBSAŽENÝCH V ŘEZU)+1);
  END;
END-ALGORITMUS;
```

5.5 Paměťová náročnost výpočtu

Jak již bylo zmíněno v předchozí části, paměťová náročnost se snižuje díky zahazování řezů s větším počtem uzlů než K. I přes toto použití je paměťová náročnost poměrně vysoká. V nejhorším případě počet řezů na uzel je $O(n^K)$, kde n je celkový počet uzlů v síti a K je počet vstupů K-LUTu. Pokud K je malé, tak je generován malý počet řezů na uzel. Ale pokud K je velké ($K \geq 7$), je již generováno velké množství těchto řezů. Přibližnou závislost ukazuje Obr. 5.5.1, který byl převzat z literatury[4]. Tento graf byl zkonstruován jako průměr počtu řezů na uzel na 20 obvodech největších MCNC benchmarků. Graf naznačuje, že

složitost řezu na uzel roste lineárně pro malá K. Nicméně pro velká K může být generováno velké množství řezů.



Obr. 5.5.1 Vyjádření závislost počtu řezů na uzel

5.6 Mapování řezů na LUTy

V předchozí části jsme vypočítali všechny K-vhodné řezy uzlů v síti. Nyní každý uzel sítě můžeme převést na K-LUT s využitím jednoho libovolného řezu daného uzlu. Jak jsme již zmínili dříve, závislost kvality mapovaného řešení závisí právě na výběru správného řezu.

Použijeme topologický uspořádaný průchod začínající z opačného konce oproti řezovému výpočtu. Začneme tedy od primárních výstupů. Následně iterativním průchodem sítě mapujeme hradla na LUTy s využitím řezů pod zpožďovacím omezením. To znamená, že pro mapování vybíráme ty řezy, které obsahují uzly s nejmenším zpožděním. Tato procedura pokračuje, dokud nedosáhneme všech primárních vstupů. Tento postup lze vyjádřit následujícím pseudokódem:

ALGORITMUS *GENERUJ_LUTY*

L = LIST PRIMÁRNÍCH VÝSTUPŮ

WHILE *L OBSAHUJE VNITŘNÍ UZLY* ***DO***

ODSTRAŇ VNITŘNÍ UZEL v *Z* L, $(L = L - \{v\})$;

VYBER VHODNÝ ŘEZ UZLU v *POD ČASOVÝM OMEZENÍM;*

VYTVOŘ K-LUT IMPLEMENTACI FUNKCE POMOCI UZLŮ V ŘEZU;

$L = L \cup input(K - LUT)$;

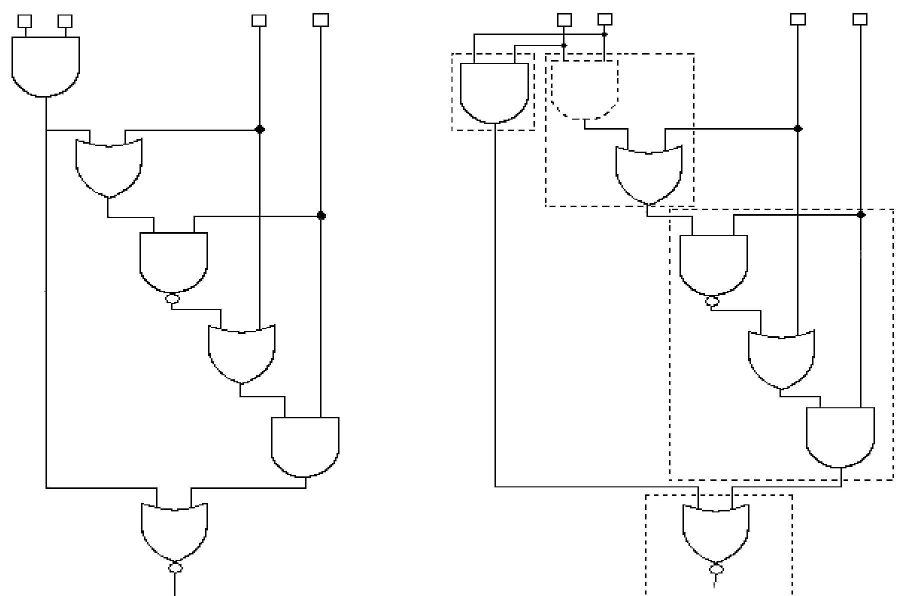
END-WHILE

END-ALGORITHM

Řezů uzlů, které mají nejmenší zpoždění může být více. V takovém případě výběr kteréhokoliv z nich neovlivní výslednou mapovací hloubku obvodu, ale může ovlivnit počet K-LUTů, tedy plochu.

5.7 Plošná optimalizace

Při mapování obvodu se záměrem minimálního počtu K-LUTů na kritické cestě výše zmíněným algoritmem dochází k duplikaci jednotlivých hradel a tudíž nárůstu celkového počtu K-LUTů. Obr. 5.7.1 znázorňuje, jak dochází k duplikaci čárkovaného hradla.



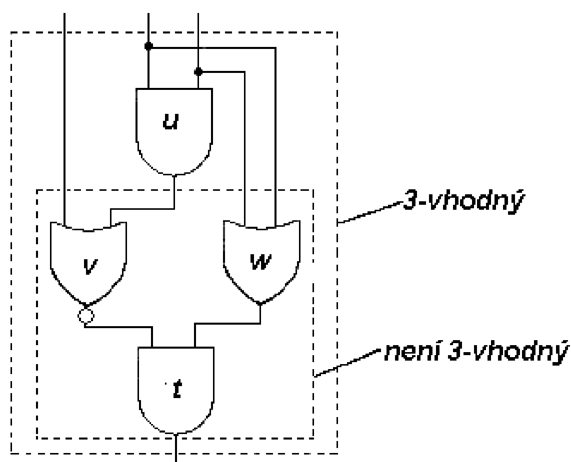
a) Původní obvod **b) Obvod namapovaný na 3-LUTy**

Obr. 5.7.1 Ukázka duplikace hradel

Tuto duplikaci nemůžeme zakázat z důvodu minimálního počtu hradel na kritické cestě. Plošná minimalizace se často provádí až po namapování obvodu na K-LUTy a je závislá na výběru řezů jednotlivých uzlů. Pokud má obvod více výstupů, tak kritickou cestu zpravidla určuje pouze jeden. Tato kritická hloubka se stanoví za základní a nepřekročitelnou hranici zpoždění a přiřadí se všem ostatním výstupům i přesto, že mají kratší zpoždění. Tyto výstupy, jimž vznikl prostor pro větší zpoždění jsou následně s využitím různých heuristik přemapovány za účelem omezení počtu celkového počtu LUTů. Zpoždění těchto cest se zpravidla zvyšuje, ale nesmí překročit zpoždění stanovené kritické cesty. Protože plošná optimalizace není předmětem této práce, nebudu se dále zabývat jednotlivými heuristikami, které jsou využívány k této optimalizaci.

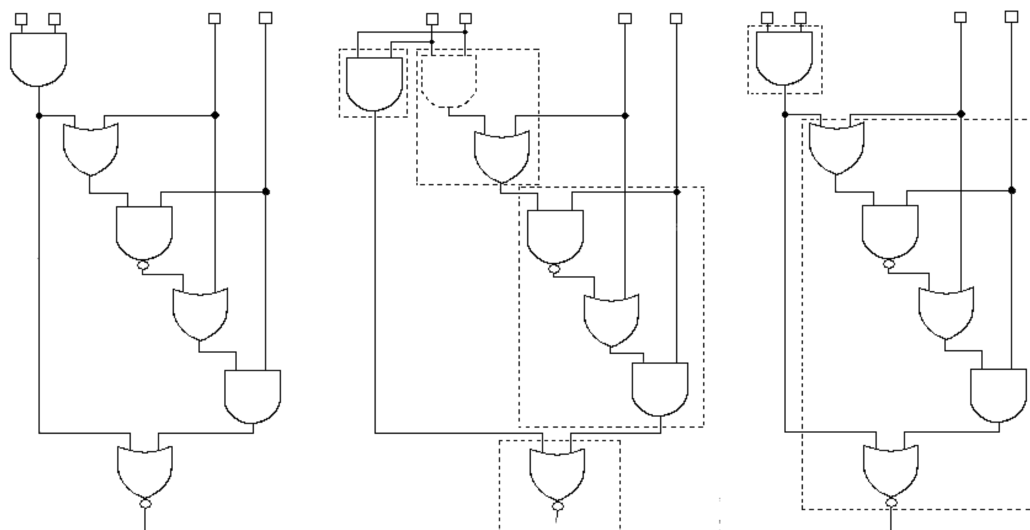
5.8 Optimalita minimální hloubky

Algoritmus, který byl prezentován, generuje optimální řešení, pokud je mapovaná síť stromem. Avšak pro obecné sítě toto tvrzení neplatí a mohou být generována suboptimální řešení. Optimální řešení je schopen generovat pouze tehdy, pokud by byly mapovací omezení pro každý programovatelný blok monotónní. Dle Lawrerovy definice[5] omezení X je monotónní, pokud je splněno následující tvrzení: „Jestliže síť H splňuje X, pak každý podgraf sítě H také splňuje X.“. Omezení na počet vstupů každého programovatelného bloku (zde LUTů) bohužel není monotónním omezením, což je vidět na Obr. 5.8.1.Celá síť má 3 odlišné vstupy, ale podsíť skládající se z uzlů t, u, w má 4 odlišné vstupy a při generování jednotlivých LUTů pomocí řezového výpočtu není nalezeno optimální řešení.



Obr. 5.8.1 Monotónnost grafu

Příkladem monotónního omezení je např. omezení na počet hradel v programovatelném logickém bloku. Na Obr. 5.8.2 je zobrazena síť, u které dochází ke generování suboptimálního řešení.



a) původní graf b) řešení poskytnuté řezovým výpočtem c) optimální řešení

Obr. 5.8.2 Příklad sítě u které nedochází ke generování optimálního řešení

5.9 Techniky pro zlepšení mapování

Tato technika hledání řešení s minimální hloubkou na kritické cestě nachází poměrně kvalitní řešení. Její velkou nevýhodou je však to, že pro větší K je časově i paměťově náročná. Proto jsem hledal zlepšení, která by urychlila výpočet. Následující dvě zlepšení jsou prezentovány v literatuře [8].

5.9.1 Použití značení

Značení jednotlivých řezů napomáhá k jednoduššímu a rychlejšímu nalezení K-vhodných řezů, dominujících a duplikovaných řezů. Princip této techniky spočívá v tom, že každý řez si označíme dle následujícího vztahu:

$$sign(C) = \sum_{n \in C} 2^{ID(n) \bmod M} \qquad 5.9.1$$

, kde ID je číselné označení uzlu a M je maximální počet bitů pro ID (32 na 32bitovém počítači). Pro určení K-vhodnosti, duplikaci a dominance řezů je využito následujících tvrzení:

- Jestliže řezy C1 a C2 jsou stejné, tak jsou stejné i jejich označení.
- Jestliže řez C1 dominuje řezu C2, jedničky označení $sign(C1)$ jsou obsaženy v jedničkách označení $sign(C2)$.
- Jestliže $C1 \cup C2$ je K-vhodný řez, pak $|sign(C1) + sign(C2)| \leq K$. Zde $|n|$ určuje počet jedniček v binární reprezentaci n a součet je bitový OR.

Testování těchto vlastností s využitím značení a výše zmíněných tvrzení by mělo být rychlejší, nežli přímé porovnávání řezů, protože jsou to jednoduché operace, které procesor zvládne rychle provést. Pokud některá podmínka selže, víme, že nemá tuto vlastnost. Pokud však vydrží, je nutné provést detailní srovnání, kvůli tzv. aliasingu. Tzn., že 2 odlišné řezy mohou mít stejné označení.

5.9.2 Redukce paměťové náročnosti

Protože počet řezů pro velká K je značný, dochází tak k velké paměťové náročnosti výpočtu. Proto je vhodné průběžně uvolňovat ty řezy, které již nebudou potřeba. Tzn., že můžeme uvolnit ty řezy uzlu, které se nepoužijí k výslednému mapování (např. kvůli velkému zpoždění). Zároveň však všechny řezy jednotlivých výstupů uzlu již musí být vypočteny. Řez nemůžeme uvolnit hned, jakmile zjistíme, že má větší zpoždění, než některý jiný řez. Je to proto, že řez nemusí být optimální pro daný uzel, ale může se stát optimálním řezem uzlu následujícího.

5.10 Zhodnocení této techniky

Tento postup mapování poskytuje velmi dobré výsledky ohledně minimální hloubky obvodu na kritické cestě. Jeho nevýhodou však je, že s rostoucím K rychle roste počet počítaných a uchovávaných řezů, což zabírá čas mapování a paměti počítače.

Naimplementoval jsem nejdříve základní algoritmus této techniky mapování a zjistil jsem, že do K=3 nenastávají žádné problémy. Pro K=4 a K=5 už pro velké obvody docházelo k obrovské časové a paměťové náročnosti. Použití znační a využití pro testování K-vhodnosti řezů zmíněné výše významně napomohlo k urychlení, ale stále to nebylo dostačující. Proto jsem se rozhodl odstraňovat duplikované a dominantní řezy opět s využitím značení. Během výpočtu se opravdu promazávalo velké množství řezů, čímž se ušetřilo na dalších výpočtech, ale složitost testování dominance a duplikace byla natolik velká, že se program ve výsledku zpomalil.

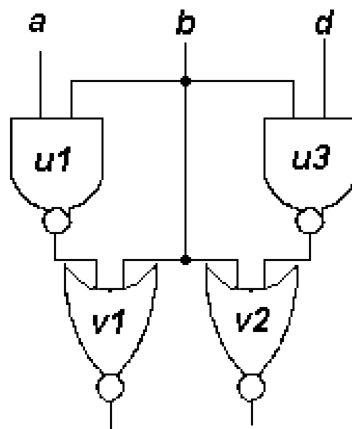
6 Vlastní algoritmus mapování na technologii LUTů

Technikou prezentovanou v předchozí kapitole je implementován i systém ABC. Jeho maximální K, pro které je schopen spočítat řešení je 6. Pro větší K končí okamžitě s chybou programu. Mnou navrhovaná implementace na tom byla ještě hůře. Proto jsem začal hledat jiný způsob, kterým lze úlohu mapování vyřešit pro větší K i za cenu mírného zhoršení hloubky obvodu.

Při hledání vlastního řešení jsem vycházel ze sítě, která již byla převedena na 2-ohraničenou síť pomocí již výše zmíněného DMIG algoritmu.

6.1 Primitivní algoritmus

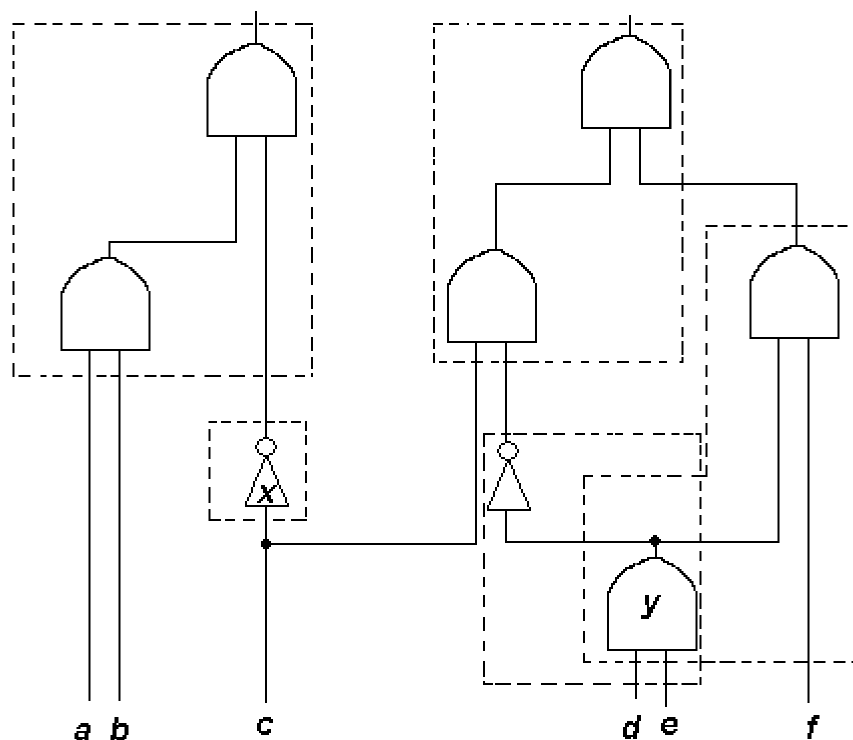
Prvním nápadem, jak provést mapování byl tento postup. Vytvořím si seznam do kterého na začátku vložím všechny primární vstupy. Ze seznamu vyberu tolik uzlů, kolik je K a budu postupovat v síti dle topologického pořadí. Až se mi sloučí v některém hradlu, tak daný podstrom sítě namapuji na LUT a výstup vložím do seznamu uzlů k mapování. Ze seznamu vyberu dalších K uzlů a pokračuji tímto způsobem dále, dokud nedosáhnu všech primárních výstupů. Tato myšlenka však byla mylná, protože pokud např. uvažuji obvod se dvěma výstupy, tak každý vstup může ovlivňovat pouze jiný výstup a nemusí se mi nikdy sloučit v některém uzlu. Příkladem takového obvodu je obvod na Obr. 6.1.1. Zde vstup a ovlivňuje pouze výstup $v1$ a vstup d ovlivňuje pouze výstup $v2$. Pokud bych uvažoval K=3 a postupoval bych zmíněným postupem, tak navrhovaný algoritmus nikdy neskončí.



Obr. 6.1.1 Příklad obvodu na kterém postup nefunguje

Tento postup tedy nefunguje. Jeho nevýhodu však odstraní jednoduchá změna pořadí procházení obvodu. Do seznamu k spočítání vložím všechny primární výstupy. Vyberu jeden uzel a postupně přidávám jeho předchůdce. Přidáním jednoho předchůdce uzlu mi v případě dvouvstupového hradla vzroste počet vstupů o 1, v případě investoru a hradla BUF mi počet vstupů LUTu nevzroste. Takto přidávám předchůdce uzlu od výstupu ke vstupům dokud nedosáhnu požadovaného K, nebo dokud všechny předchůdce uzlu nevyčerpám. Vytvořený podstrom namapuji na LUT a jeho vstupy vložím do seznamu k spočítání (kromě primárních vstupů). Vyberu další uzel a stejným algoritmem pokračuji dále, dokud není seznam k spočítání prázdný. Poté vytvořím pravdivostní tabulky LUTů, pomocí zjištěných podstromů a zapíši výsledky.

První verze, označená jako primitiv, byla naimplementována tak, že přidávání předchůdců probíhalo rekurzivně. Tzn. že se zavolal předchůdce, který má přidat své vstupy. Jako parametr se mu předal odkaz seznamu vstupů počítaného LUTu a počet požadovaných vstupů K. Předchůdce nejdříve odebere ze seznamu vstupů sám sebe a poté přidá své vstupy. Pokud je počet vs tupů stále menší než požadovaný počet vstupů požádá své předchůdce o přidání svých vstupů stejnou metodou. Jinými slovy dochází k přidávání hradel do LUTu stejným způsobem, jako když procházíme strom v pořadí preorder. Obr. 6.1.2 ukazuje způsob mapování tímto jednoduchým algoritmem.



Obr. 6.1.2 Ukázka namapování primitivním algoritmem

U tohoto postupu nebyl brán ohled ani na minimalizaci hloubky ani na počet LUTů. Za povšimnutí stojí, že hradlo x nemuselo být implementováno samostatným LUTem, ale mohlo být zahrnuto do LUTu s hradly AND, protože nezvyšuje počet vstupů. Kdyby hradlo x leželo na kritické cestě obvodu, tak by zbytečně došlo ke zvýšení jeho zpoždění.Dalším postřehem je, že podobně jako při řezovém výpočtu, zde dochází k duplikaci hradel. Zde konkrétně hradla y.

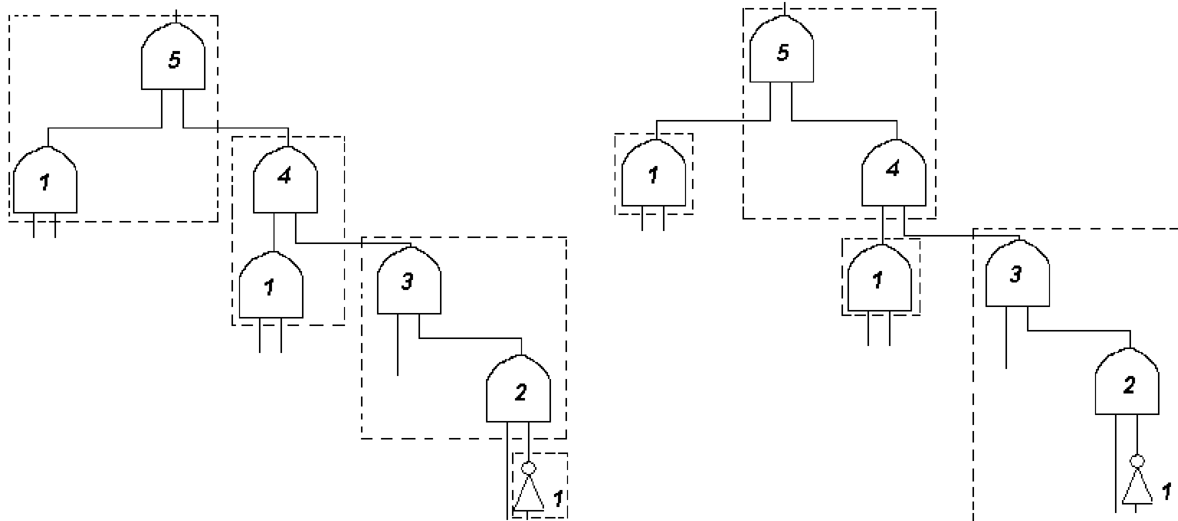
6.2 Vylepšení primitivního algoritmu

Protože výsledky mapování primitivního algoritmu, které jsou uvedeny v kapitole 9, nebyly příliš přívětivé, bylo nutné hledat, další zlepšení, které by minimalizovalo hloubku.

Dříve zmíněné zlepšení, které navrhuje zahrnout do LUTu všechny jednovstupové předchůdce, nevykazuje výrazné zlepšení. Je zřejmé, že u tohoto případu dojde maximálně ke zvýšení hloubky a 1, protože takovýto případ může nastat pouze u jednovstupových hradel, jejíchž předchůdci jsou primární uzly.

Hloubka výsledného řešení závisí na výběru předchůdců. Protože již známe hloubku jednotlivých hradel získanou použitím algoritmu DMIG, tak by bylo nejlepší vybírat nejdříve

ty předchůdce, které mají maximální hloubku. Srovnání primitivního a tohoto vylepšení ukazuje Obr. 6.2.1. Tímto způsobem se zajistí, že budu mapovat nejdříve kritickou cestu obvodu a budu ji tak minimalizovat. Teprve potom namapuji ostatní hradla, která na kritické cestě neleží.



a) primitiv b) primitiv_better

Obr. 6.2.1 Porovnání algoritmu primitiv a primitiv_better

Tyto 2 vylepšení jsem naimplementoval do programu pod názvem primitiv_better. Toto vylepšení vede k mnohem menší hloubce a také mírnému zmenšení počtu LUTů. Ve srovnání se systémem ABC tento algoritmus sice generuje o nepatrně větší zpoždění, ale na druhou stranu není omezen počtem vstupů pro K=6.

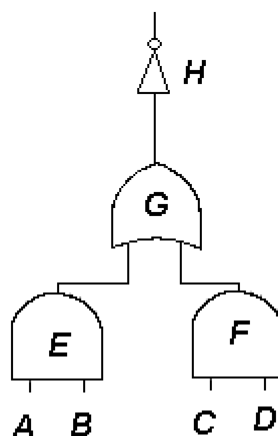
7 Vytváření pravdivostních tabulek

Po mapování na technologii LUTů již znám, která hradla budou součástí kterého LUTu. Posledním krokem tedy zbývá vytvořit pravdivostní tabulku LUTu, která bude následně implementována do FPGA.

7.1 Způsob vytváření

Zde se pokusím vysvětlit vlastní způsob vytváření těchto pravdivostních tabulek. Opět začínám výstupním hradlem. Na něm zavolám funkci, která spočítá jeho pravdivostní tabulku (tu vrací jako návratovou hodnotu formou množiny řádků pravdivostní tabulky). Jako parametry mu předám vstupy LUTu (tedy hraniční mez, po která hradla se má počítat pravdivostní tabulka) a požadovanou hodnotu (výchozí je vždy 1). Hradlo potom rekurzivně zavolá spočítání pravdivostních tabulek stejným způsobem na jeho vstupech a poté v závislosti na typu hradla dojde ke sloučení řádku pravdivostní tabulky nebo jejich přidání. Způsob jakým se mění požadovaná hodnota závisí na typu hradla. Např. pokud budu počítat pravdivostní tabulku na hradlu typu NOT a budu chtít vědět, kdy je jeho výstup roven 1, tak mě zajímá kdy je jeho vstup roven 0.

Tento postup ukáži na krátkém příkladě. Uvažujme LUT skládající se z hradel vyobrazených na Obr. 7.1.1.



Obr. 7.1.1 Ukázkový obvod pro výpočet pravdivostní tabulky

Počátečním krokem tedy je požádat hradlo H o výpočet pravdivostní tabulky. Parametry volání funkce jsou tyto:

$$vstupy = \{A, B, C, D\}$$
$$požadovaná_hodnota = 1$$

Vstupy nadále jsou pro všechna volání stejná. Hradlo H se nejdříve podívá, jestli není v seznamu vstupu, zjistí-li, že ne, pak zavolá spočítání pravdivostní tabulky na svých vstupech, zde na hradlu G a předá mu parametr: $požadovaná_hodnota = 0$.

Hradlo G postupuje obdobně. Zavolá spočítání pravdivostních tabulek na hradlech E a F s předáním parametru $požadovaná_hodnota = 0$, protože hradlo OR je nulové jenom tehdy, pokud jsou nulové oba jeho vstupy.

Hradla E a F postupují úplně stejně jako hradlo G.

A konečně hradla A, B, C, D zjistí, že se nachází v seznamu vstupů a vrátí pravdivostní tabulku, která bude obsahovat jeden řádek. V tomto řádku bude proměnná o hodnotě, která byla požadována:

$$t_A = \{\{A = 0\}\}$$
$$t_B = \{\{B = 0\}\}$$
$$t_C = \{\{C = 0\}\}$$
$$t_D = \{\{D = 0\}\}$$

Tyto pravdivostní tabulky se vrátí hradlům E a F. Protože jsou to hradla typu AND a požaduje se na výstupu 0, stačí aby 0 byla na jednom vstupu. Proto dojde pouze ke sjednocení řádků:

$$t_E = \{\{A = 0\}\{B = 0\}\}$$
$$t_F = \{\{C = 0\}\{D = 0\}\}$$

Hradlo G požaduje na obou vstupech zároveň 0, proto dojde ke kombinaci řádků jedné tabulky s řádky druhé tabulky.

$$t_G = \{\{A = 0, C = 0\}, \{A = 0, D = 0\}, \{B = 0, C = 0\}, \{B = 0, D = 0\}\}$$

A nakonec hradlo H vrátí pravdivostní tabulku, kterou obdrželo od hradla G.

8 Popis implementace

8.1 Výběr programovacího jazyka

Pro implementaci uvedených algoritmů jsem vybral programovací jazyk C++, zejména z důvodu rychlosti provádění programu. Programovací jazyk Java není příliš vhodný, protože je to interpretovaný jazyk a tudíž i pomalejší. Dalším důvodem pro jazyk C++ byla možná budoucí implementace do stávajícího systému EDA (EDuArd) vyvíjeném na katedře počítačů ČVUT.

8.2 Datové struktury a metody

Celý program se skládá z několika tříd. O načtení logického obvodu do vnitřní reprezentace se stará třída Import. Využívá třídy lexikálního analyzátoru LexAn a LexSymbol. Třída Import provádí rekurzivní sestup a načítá obvod do vnitřní reprezentace, která je vyjádřena třídou Gate. Nejdůležitější částí programu je třída Mapper, která s obvodem manipuluje, provádí převod na K-ohraničenou síť a mapuje obvod na LUTy. Právě implementace této části programu je odlišná ve všech uvedených verzích. LUTy jsou reprezentovány třídou Lut. O prezentaci výsledků do soboru ve formátu BLIF se stará třída Export_LUTs.

Pro vyjádření pravdivostních tabulek jsou definovány 2 typy:

```
Typedef std::map<int, int> Row;     (id vstupního hradla, hodnota=0/1)
Typedef stdd::list<Row*> True_table;
```

Následně uvedu nejdůležitější proměnné a metody tříd.

8.2.1 Třída Import

Název metody	Parametry	Návratová hodnota	Význam
Import	char* name_of_file		konstruktor, přebírá název souboru, ze kterého s bude číst
read_circuit	std::list<Gate*>& _inputs std::list<Gate*>& _outputs std::list<Gate*>& inner_nodes	void	Načte obvod do vnitřní reprezentace

Tabulka 8-1 Důležité metody třídy Import

8.3 Třída Gate

Název_proměnné	Typ	Význam
id	unsigned int	ientifikátor hradla
name	String	Název hradla
depth	unsigned int	Hloubka hradla v obvodu
depth_of_lut	unsigned int	Hloubka případného lutu
K_bounded	Bool	Příznak, jestli již byl převeden na K-ohraničenou síť
type	Type_of_gate	Typ hradla
cuts	std::list<Cut*>	Seznam řezů
inputs	std::list<Gate*>	Seznam vstupních hradel (předchůdců)
outputs	std::list<Gate*>	Seznam výstupních hradel (následníků)

Tabulka 8-2 Důležité proměnné třídy Gate

Název metody	Parametry	Návratová hodnota	Význam
compute_cuts	Unsigned int K	Void	Spočítá řezy s využitím řezů předchůdců [1])
compute_true_table	Cut& c, int hodn	True_table*	Vypočítá pravdivostní tabulku LUTu
merge_cuts	Cut& cut Cut& cut1 Cut& cut2 unsigned int K	Bool	Vypočte řezy daného hradla pomocí řezů cut1 a cut2 s omezením vstupů K [1])
merge_tables	True_table* t True_table* t1 True_table* t2	Void	Sloučí 2 pravdivostní tabulky do 1
add_inputs	std::set<Gate*>& vstupy unsigned int K	Void	Přidává své předchůdce do množiny vstupy, dokud jich není K [2])
Add_inputs	std::multiset<Gate*, cmp>& vstupy std::set<Gate*>& vstupy_mn	Void	Přidá své předchůdce [3])

Tabulka 8-3 Důležité metody proměnné třídy Gate

[1]) pouze verze vyžívající množiny řezů

[2]) pouze verze primitiv

[3]) pouze verze primitiv_better

8.4 Třída Lut

Název_proměnné	Typ	Význam
id	unsigned int	ientifikátor hradla
name	String	Název hradla
true_table	True_table*	Pravdivostní tabulka
names_inputs	std::list<Gate*>	Názvy vstupů

Tabulka 8-4 Důležité proměnné třídy Lut

8.5 Třída Mapper

Název metody	Parametry	Návratová hodnota	Význam
inputs_k_bounded	Gate* g	Bool	Zjistí, zda-li hradlo g, již bylo k-ohraničeno
dmig	Gate *g unsigned int K	Void	Provede dekompozici hradla na K-vstupove
transform_to_K_bounded	unsigned int K	Void	Převede celý obvod na K-ohraničený
compute_cuts	unsigned int K	Void	Spočítá řezy všech uzlů 1)
make_luts	std::list<Lut*>& luts	Void	Pomocí řezů vytvoří LUTy 1)
Map	std::list<Lut*>& luts unsigned int K	Void	Namapuje obvod na K-Luty 2)

Tabulka 8-5 Důležité proměnné třídy Mapper

1) pouze verze využívající množiny řezů

2) pouze verze primitiv a primitiv_better

8.6 Třída Export_LUTs

Název metody	Parametry	Návratová hodnota	Význam
export_LUTs	char* name_of_file	Void	Zápíše výsledky mapování do souboru ve formátu BLIF

Tabulka 8-6 Důležité metody třídy Export_LUTs

9 Testování

Implementované algoritmy jsem otestoval na některých MCNC [9] testovacích obvodech. Výsledky testování jsem porovnal s již existujícími systémy ABC [1] a SIS [2]. Protože moje implementace pomocí řezových množin (označena cuts) byla pro větší obvody časově náročná, rozdělil jsem testování na skupinu mapování malých obvodů a velkých obvodů.

Pro malé obvody porovnávám počty LUTů a hloubku obvodu všech mých algoritmů.

Testování velkých obvodů jsem provedl pro K=6, což je současná maximální hranice pro systém ABC [1]. Pro moje implementace primitiv a primitiv_better a systém SIS jsem nenalezl žádnou pevně stanovenou hranici, pro kterou se stává mapování časově náročné. Vždy velmi záviselo na obvodu. S algoritmy primitiv a primitiv_better jsem však byl schopen namapovat obvody s výrazně vyšším K.

Pro velké obvody jsem také srovnal doby mapování. Toto testování bylo prováděno na notebooku s procesorem Pentium M 740 (1,73GHz) s operační pamětí 1GB.

Funkční ekvivalentnost původních a namapovaných obvodů jsem kontroloval pomocí systému SIS [2] příkazem verify.

Následující tabulky ukazují získané výsledky:

Obvod	Původní hloubka obvodu	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	14	5	7	6	9	6
s208.1	11	5	8	6	6	6
s27	6	3	5	3	4	3
s298	9	4	9	5	7	5
s344	20	6	11	6	9	7
s349	20	6	11	6	9	7
s382	9	5	7	5	6	6
s386	11	5	7	6	6	6
s400	9	5	7	5	6	6
s420	28	9	13	12	15	12
s420.1	13	7	13	7	8	8
s444	11	5	7	6	7	6
s510	12	6	11	6	8	6
s526	9	4	9	5	7	5
s526n	9	4	9	5	7	5
s641	74	10	16	12	16	13
c432	17	16	20	16	20	16

Tabulka 9-1 Srovnání hloubky mapování malých obvodů pro K=3

Obvod	Původní hloubka obvodu	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	14	4	6	5	8	5
s208.1	11	4	7	4	6	4
s27	6	2	4	2	3	2
s298	9	4	6	4	5	4
s344	20	4	4	4	6	5
s349	20	4	4	4	6	5
s382	9	3	6	4	5	4
s386	11	4	6	4	5	4
s400	9	3	6	4	5	4
s420	28	7	14	9	12	9
s420.1	13	4	11	5	7	5
s444	11	3	5	4	6	4
s510	12	4	10	4	8	5
s526	9	4	6	4	6	4
s526n	9	4	7	4	5	4
s641	74	6	10	9	12	9
c432	17	10	22	10	16	10

Tabulka 9-2 Srovnání hloubky mapování malých obvodů pro K=4

Obvod	Původní hloubka obvodu	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	14	3	7	4	7	4
s208.1	11	3	4	3	5	4
s27	6	2	3	2	2	2
s298	9	3	4	3	5	3
s344	20	3	5	3	6	5
s349	20	3	5	3	6	5
s382	9	3	4	3	5	3
s386	11	3	4	3	4	3
s400	9	3	4	3	5	3
s420	28	5	7	6	11	6
s420.1	13	4	5	4	6	5
s444	11	3	4	3	5	3
s510	12	4	6	4	7	4
s526	9	3	4	4	5	3
s526n	9	3	4	3	5	3
s641	74	5	10	-	10	7
c432	17	8	19	8	16	9

Tabulka 9-3 Srovnání hloubky mapování malých obvodů pro K=5

Obvod	Původní hloubka obvodu	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	14	3	4	3	3	3
s208.1	11	3	5	3	5	4
s27	6	1	2	1	2	1
s298	9	2	3	2	5	3
s344	20	3	4	3	6	3
s349	20	3	4	3	6	3
s382	9	2	4	3	4	3
s386	11	2	4	3	4	3
s400	9	2	3	3	4	3
s420	28	5	7	5	10	5
s420.1	13	3	8	4	5	4
s444	11	2	4	3	5	3
s510	12	3	5	3	6	3
s526	9	3	4	3	5	3
s526n	9	3	4	3	5	3
s641	74	4	9	-	9	6
c432	17	7	13	7	15	8

Tabulka 9-4 Srovnání hloubky mapování malých obvodů pro K=6

Obvod	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	42	50	63	66	51
s208.1	38	39	63	76	47
s27	8	6	10	9	9
s298	61	63	92	110	85
s344	49	104	75	113	82
s349	49	104	75	115	85
s382	70	80	111	128	94
s386	83	101	129	130	116
s400	70	86	112	136	95
s420	84	107	125	138	100
s420.1	85	107	133	161	101
s444	70	99	102	146	119
s510	131	168	168	169	151
s526	118	116	187	204	177
s526n	118	118	187	205	176
s641	101	106	156	182	132
c432	105	120	143	151	129

Tabulka 9-5 Srovnání počtu LUTů mapování malých obvodů pro K=3

Obvod	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	31	29	42	56	42
s208.1	26	27	41	61	31
s27	6	6	9	8	8
s298	42	41	72	82	70
s344	44	43	69	97	69
s349	49	43	69	100	70
s382	58	62	84	112	81
s386	57	69	96	98	95
s400	58	62	83	120	87
s420	62	62	95	119	87
s420.1	62	67	103	133	79
s444	59	57	85	122	86
s510	102	111	139	141	126
s526	81	82	146	165	142
s526n	81	83	146	167	141
s641	82	81	143	153	117
c432	100	79	123	130	125

Tabulka 9-6 Srovnání počtu LUTů mapování malých obvodů pro K=4

Obvod	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	23	25	37	45	29
s208.1	22	22	33	52	29
s27	5	5	6	7	6
s298	30	32	57	75	60
s344	35	50	54	83	60
s349	35	50	54	84	60
s382	46	46	72	92	64
s386	43	48	77	77	80
s400	46	44	76	98	69
s420	47	52	81	94	61
s420.1	51	51	81	118	71
s444	46	44	78	117	72
s510	75	80	115	119	113
s526	60	61	146	141	126
s526n	60	61	116	139	125
s641	72	69	-	146	114
c432	79	62	116	124	130

Tabulka 9-7 Srovnání počtu LUTů mapování malých obvodů pro K=5

Obvod	ABC	SIS	Cuts	Primitiv	Primitiv_better
s208	18	19	31	37	27
s208.1	18	21	34	43	27
s27	4	4	4	8	4
s298	24	27	47	63	51
s344	33	44	45	72	57
s349	33	33	45	74	57
s382	37	35	55	81	55
s386	38	39	61	71	64
s400	37	34	55	91	55
s420	40	38	72	85	58
s420.1	45	46	77	106	63
s444	37	35	57	104	68
s510	44	52	68	101	93
s526	49	48	97	118	93
s526n	49	47	97	117	93
s641	72	66	-	141	106
c432	77	56	118	119	117

Tabulka 9-8 Srovnání počtu LUTů mapování malých obvodů pro K=6

	Primitiv		Primitiv_better	
Obvod	**Počet LUTů**	**Hloubka**	**Počet LUTů**	**Hloubka**
s208	24	6	17	2
s208.1	24	4	17	2
s27	4	1	4	1
s298	33	3	39	2
s344	51	4	39	2
s349	51	4	39	2
s382	66	4	43	2
s386	45	3	53	2
s400	70	4	45	2
s420	58	7	43	3
s420.1	80	5	45	3
s444	68	4	48	2
s510	52	4	65	3
s526	69	3	85	2
s526n	69	3	83	2
s641	131	7	91	4
c432	119	15	117	5

Tabulka 9-9 Výsledky mapování malých obvodů pro K =10

Obvod	Původní hloubka obvodu	ABC	SIS	Primitiv	Primitiv better
c1355	24	4	-	20	5
c1908	40	6	13	18	8
c2670	32	5	11	17	7
c3540	47	8	19	28	13
c499	11	4	-	10	5
c5315	49	6	12	21	10
c6288	124	16	56	75	22
c7552	43	6	11	19	8
c880	24	5	11	18	7
s15850.1	82	9	22	34	12
sS35932	29	3	5	7	4
s38417	47	6	11	19	8
s38584.1	56	6	15	19	9
s5378	25	4	9	12	5
s9234.1	58	6	14	19	8

Tabulka 9-10 Srovnání hloubky mapování velkých obvodů pro K=6

Obvod	Původní počet hradel	ABC	SIS	Primitiv	Primitiv_better
c1355	546	64	-	370	116
c1908	880	96	123	538	267
c2670	1193	137	238	701	536
c3540	1669	257	343	964	740
c499	202	64	-	152	104
c5315	2307	299	377	1350	1039
c6288	2416	518	651	2271	842
c7552	3512	467	585	2347	1406
c880	383	92	102	201	183
s15850.1	9775	1039	1132	2691	1793
s35932	16385	2624	3072	8505	4576
s38417	22257	2591	2859	7581	4936
s38584.1	19405	2897	3593	9399	5620
s5378	2836	403	445	1084	812
s9234.1	5597	525	595	1732	1124

Tabulka 9-11 Srovnání počtu LUTů mapování velkých obvodů pro K=6

Obvod	ABC	SIS	Primitiv_better
c1355	1,39s	-	0,08s
c1908	0,44s	21,8s	0,08s
c2670	0,33s	11,9s	0,09s
c3540	0,34s	19,4s	0,14s
c499	0,47s	-	0,03s
c5315	0,69s	28,9s	0,16s
c6288	5,55s	48,1s	0,23s
c7552	1,66s	50,2s	0,25s
c880	0,16s	84,4s	0,05s
s15850.1	0,83s	85,3s	0,63s
s35932	4,77s	49s	3,63s
s38417	2,30s	259,2s	1,70s
s38584.1	0,91s	89,6s	2,19s
s5378	0,38s	10,1s	0,17s

Tabulka 9-12 Srovnání časů mapování velkých obvodů pro K=6

10 Závěr

V této práci jsem prezentoval možnosti mapování logických obvodů na technologii FPGA, konkrétně LUTů. Prezentoval jsem algoritmus mapování, na kterém je založeno spousta strukturálních algoritmů. Tento algoritmus se také podařilo naimplementovat. Z testů je vidět, že podával vcelku velmi dobré výsledky. Nicméně jsem však musel řešit problém výpočetní náročnosti již pro relativně malá K a neschopnosti mapovaní velkých logických obvodů.

Z toho důvodu jsem hledal řešení, kterým namapuji i velmi velké logické obvody. Jakmile jsem takové řešení nalezl, snažil jsem se zlepšit hloubku. Prezentované výsledky v této práci ukazují , ve většině případech dosahuji navrhnutým algoritmem lepší hloubky než u systému SIS [2] a také téměř ve všech případech je mapování rychlejší než systém SIS [2] i ABC[1]. Nevýhodou navrhnutého algoritmu je výsledný počet LUTů, který je podstatně větší než u uvedených systémů. Touto problematikou by bylo dobré se ještě dále zabývat.

11 Použitá literatura

[1] ABC, A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/~alanmi/abc/

[2] E.M. Sentovich et al.: SIS: A System for Sequential Circuit Synthesis, Electronics Research Laboratory Memorandum No. UCB/ERL M92/41, University of California, Berkeley, CA 94720, 1992

[3] K. C. Chen, J. Cong, Y. Ding, A. B. Kahng, P. Trajmar, „DAG-Map: Graph-Based FPGA Technology Mapping for Delay Optimization“, IEEE Design and Test of Computers, 1992

[4] D. Chen, J. Cong, „DAOmap: A Depth-optimal Area Optimization Mapping Algorithm for FPGA Designs

[5] E. L. Lawler, K. N. Levitt, and J. Turner, „Module Clustering to Minimize Delay in Digital Networks“, IEEE Trans. Computers, 1969

[6] J. Cong, Y. Ding, „FlowMap: An Optima Technology Mapping Algorithm for Delay Optimization in Lookup-Table Based FPGA Designs“, IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systems, vol. 13, no.1, January 1994

[7] R. Francis, J. Rose, Z. Vranesic, „Chortle-crf: Fast Technology Mapping for Lookup Table-Based FPGAs“, Department of Electrical Engineering, University of Toronto, Canada

[8] A. Mischenko, S. Chatterjee, R.K. Baryton, „Improvements to Technology Mapping for LUT-Based FPGAs“, IEEE Transaction On Computer-Aided Design of Integrated Circuits and Systéme. vol 26, no. 2, February 2007

[9] S. Yang. Logic Synthesis and Optimization Benchmarks User Guide, Technical Report 1991-IWLS-UG-Saeyang, MCNC, Research Triangle Park, NC, January, 1991

A. Seznam použitých zkratek

BENCH	
BLIF	Berkeley logic interchange format
DAG	directed acyclic graph
DMIG	decompose multi input gate
FPGA	field programmable gate array
LUT	look-up table
EDA	EDuArd, výukový systém vyvíjený na katedře počítačů ČVUT v Praze
ČVUT	České vysoké učení technické v Praze

B. Uživatelská příručka

Všechny spustitelné binární soubory se spouští stejným způsobem. Program se spouští z příkazového řádku v prostředí Windows následovně:

mapper.exe vstupni_soubor.bench vystupni_soubr.blif K

, kde K je počet vstupů LUTu.

C. Obsah přiloženého CD

adresář **bakalarka**

bakalarka.doc	- text bakalářské práce ve formátu doc
bakalarka.pdf	- text bakalářské práce ve formátu pdf

adresář **benchmarky**

adresář **bench**	- obsahuje soubory testovaných MCNC obvodů ve formátu bench
adresář **blif**	- obsahuje soubory testovaných MCNC obvodů ve formátu blif

adresář **SIS**	- obsahuje balík systému SIS verze 1.2
adresář **ABC**	- obsahuje balík systému ABC verze 70930
adresář **literatura**	- obsahuje literaturu, ze které jsem čerpal
adresář **execute**	-obsahuje spustitelné soubory implementovaných algoritmů (cuts, primitiv a primitiv_better)
adresář **source**	-obsahuje zdrojové kódy ve formě projektu pro Microsoft Visual Studio 2005