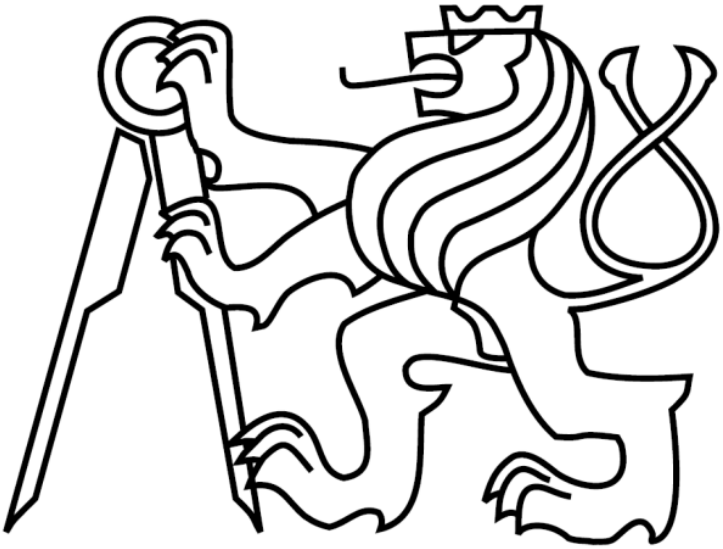
České vysoké učení technické v Praze
Fakulta elektrotechnická



Bakalářská práce

Dekompozice kombinačních logických obvodů

Jakub Zahradník

Vedoucí práce: Ing. Petr Fišer

Studijní program: Elektrotechnika a informatika strukturovaný bakalářský

Obor: Informatika a výpočetní technika

Srpen 2007

Poděkování

Rád bych poděkoval Ing. Petru Fišerovi za přínosné konzultace, poskytnutí literatury a za vedení této práce. Dále bych rád poděkoval Ing. Janu Schmidtovi, Ph.D., za poskytnutou pomoc s EDA systémem.

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 12.8.2007 …….

Anotace

Práce se zabývá převodem logických obvodů zadaných pravdivostní tabulkou na víceúrovňové logické sítě. Využívá k tomu algebraickou faktorizaci s vybranými prvky booleovské algebry. Výsledek je vyhodnocen z hlediska kvality (celkový počet hradel, počet úrovní výsledného obvodu) a srovnán s dekompozicí provedenou aplikací MVSIS. Vstupem je soubor ve formátu PLA, výstupem soubor ve formátu VHDL.

Abstract

The work deals with a transformation of logical circuits given as a truth table into a multi-level logic network. It uses algebraic factorization with some portion of Boolean algebra. The result is analyzed in term of quality (total gates count, number of levels) and compared to decomposition done by MVSIS aplication. The PLA format is used as the input format and VHDL as the output format.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

1.1 Motivace

Nedílnou součástí práce s logickými obvody je nepochybně i převod mezi různými reprezentacemi daného logického obvodu. Asi nejjednodušší reprezentací je pravdivostní tabulka, která představuje model chování logického obvodu bez jakéhokoli popisu jeho realizace. Hlavní výhodou tabulky je její snadné a rychlé vytvoření při návrhu logického obvodu. Fyzická realizace obvodu však vyžaduje znalost strukturního popisu vyjádřeného pomocí sítě hradel. Proces převodu z jedné reprezentace, tabulky, do druhé, sítě hradel, je předmětem této práce. Protože je převod velmi obtížný, zejména u rozsáhlých logických obvodů, je třeba vytvořit co nejefektivnější implementaci.

1.2 Cíl práce

Cílem této práce je navrhnout a implementovat aplikaci, jež bude převádět vstupní reprezentaci logického obvodu, tabulku (formát PLA), do výstupní, víceúrovňové sítě hradel (formát VHDL[1]), za využití stávajících funkčností EDA[2] systému (viz kapitola 2). To znamená import obvodu ze souboru ve formátu PLA do vnitřních struktur HUBu[3] (za pomoci importního podsystému EDA), jeho následnou konverzi z behaviorálního popisu na popis strukturní a následný export pomocí exportního podsystému EDA do souboru ve formátu VHDL. Aplikace bude pracovat na bázi algebraické faktorizace.

Druhým cílem práce je zhodnotit efektivitu aplikace v porovnání s existujícími řešeními, a to z hlediska kvality (celkový počet hradel, počet úrovní výsledného obvodu) a časové složitosti.

1.3 Existující řešení

1.3.1 MVSIS 2.0

Aplikace MVSIS 2.0 nahrazuje starší verze SIS a přidává nové možnosti víceúrovňové manipulace, byla vyvinuta na univerzitě v Berkeley. Aplikace podporuje vstupní formáty BLIF a PLA, umožňuje kvalitní a velmi rychlou dekompozici. Verze pro Windows je obsažena na přiloženém CD.

1.4 Základní pojmy

V této práci se pracuje s několika základními pojmy, které je nutné vysvětlit.

Logická proměnná – proměnná, která nabývá hodnot 0 a 1

Logická (booleovská) funkce – funkce se vstupy logických proměnných, nabývá hodnot 0 a 1

Logický prvek – fyzický člen realizující logickou funkci

---

[1] **V**HSIC **H**ardware **D**escription **L**anguage.
[2] **E**lectronic **D**esign **A**utomation.
[3] Jádro EDA systému.

Logický obvod – souhrn propojených logických prvků realizující složitější logickou funkci

Pravdivostní tabulka – reprezentace logické funkce, ve které je definován konkrétní vztah mezi vstupními a výstupními proměnnými; příklad pravdivostní tabulky ukazuje tabulka 1.1.

Tabulka 1.1: Pravdivostní tabulka pro funkci $f = x_1 + x_2$

x_1	x_2	f
0	0	0
0	1	1
1	0	1
1	1	1

Literál – logická proměnná nebo její negace

Term – n-tice literálů spolu svázaných logickým operátorem; podle operátoru se dělí na součinový (např. $x_1 \cdot x_2$), součtový (např. $x_1 + x_2$) a další

Krychle – součinový term

Normální disjunktivní forma – algebraická reprezentace logické funkce skládající se ze součtu součinových termů

ON-set logické funkce – množina termů, kde výstupní proměnná nabývá hodnoty 1

OFF-set logické funkce – množina termů, kde výstupní proměnná nabývá hodnoty 0

DC-set logické funkce – don't care set; množina termů, pro kterou není logická funkce specifikovaná

2 EDA systém

2.1 Úvod

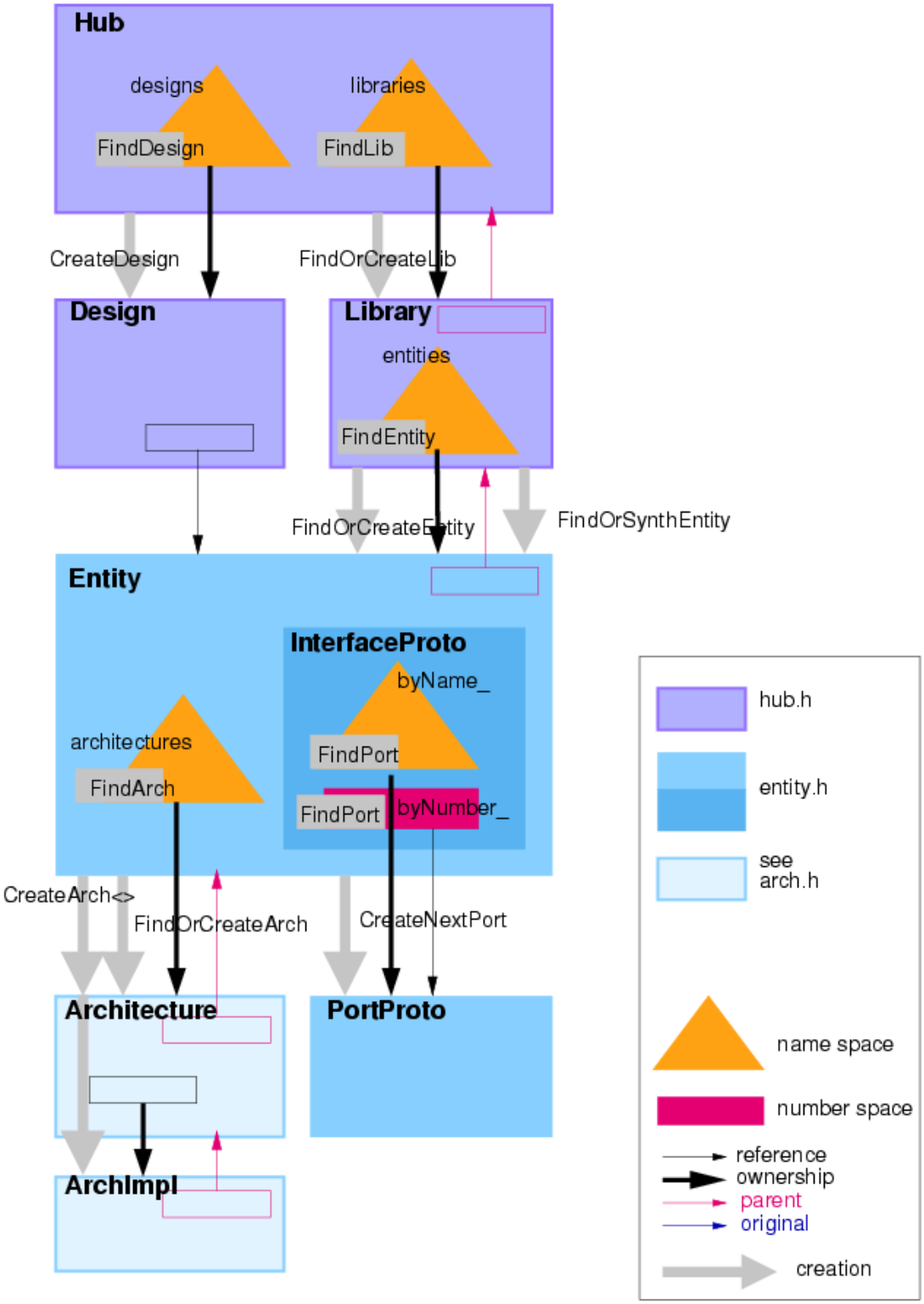
Následující kapitola se zabývá stručným popisem využitých funkcí EDA systému, vyvíjeného na katedře počítačů ČVUT FEL. Práce popisuje takovou podmnožinu systému, kterou bude implementace výsledné aplikace využívat, a je tedy nutné se s ní seznámit. Z pohledu funkčnosti se EDA systém dělí na následující moduly: jádro HUB, importní podsystém, exportní podsystém a standardní transformační procedury.

2.2 Jádro HUB

Jádro HUB (dále jen HUB) je soubor datových struktur a metod, jejichž pomocí lze popsat logický obvod a manipulovat s ním, je základním stavebním kamenem EDA systému. HUB se dělí na tři části, na vrchní vrstvu, starající se o organizaci a správu datových struktur, na část popisující behaviorální chování obvodu a na část popisující jeho strukturu.

2.2.1 Vrchní vrstva

Hierarchicky nejvýše umístěnou třídou, reprezentující vrchní vrstvu, je třída *Hub*. Všechny funkčnosti HUBu lze využít jen za pomoci této třídy. Předpokládá se, že v aplikaci bude jen jedna její instance, z které je možné organizovat všechny ostatní objekty, není to však nutností. Třída *Hub* přímo organizuje knihovny a návrhy. Architekturu vrchní vrstvy detailně zobrazuje obrázek 2.1.



Obrázek 2.1: HUB – vrchní vrstva (Schmidt)

2.2.1.1 Knihovna

Knihovna je reprezentována třídou *Library* a má jediný účel, a sice spravovat jmenný prostor entit. Správa zahrnuje jejich vytváření a vyhledávání dle jména nebo funkce. Samotná knihovna je také pojmenována, takže k ní lze z nadřazené třídy *Hub*

přistupovat. Třída je odpovědná za rušení entit při uvolnění z paměti. V praxi má jen organizační uplatnění pro rozřazení entit do několika knihoven, nejčastěji dle jejich funkce.

2.2.1.2 Návrh

Návrhem se rozumí pojmenovaný odkaz na entitu. Je reprezentován třídou *Design*. Pro účely této práce je nepodstatný.

2.2.1.3 Entita

Entita, vyjádřená třídou *Entity*, spravuje jmenný prostor architektur a rozhraní portů. Ve fyzickém světě představuje návrh abstraktního obvodu, jenž může být instanciován a použit ve strukturním popisu jiné entity. Entita je pravděpodobně nejdůležitějším prvkem celého systému.

2.2.1.4 Rozhraní portů

Rozhraním portů (*InterfaceProto*) se rozumí množina všech vstupních, výstupních, popř. vstupně-výstupních portů (*PortProto*) dané entity. Tyto porty popisují rozhraní, jehož pomocí entita (obvod) interaguje s okolím. Každý port je jednoznačně identifikovatelný, a to jménem, číslem a funkcí. Rozhraní portů je nedílnou součástí entity, bez něj je entita užitečná asi jako Medvěděvův konečný automat[4].

2.2.1.5 Architektura

Zatímco rozhraní portů definuje vstupy a výstupy obvodu, architektura, zastoupená třídou *Architecture*, popisuje chování obvodu. Entita může obsahovat několik architektur, jejichž implementace (poděděné z abstraktní třídy *ArchImpl*) mohou být behaviorálního i strukturního charakteru, tj. obvod může být popsán např. jak pravdivostní tabulkou, tak sítí hradel.

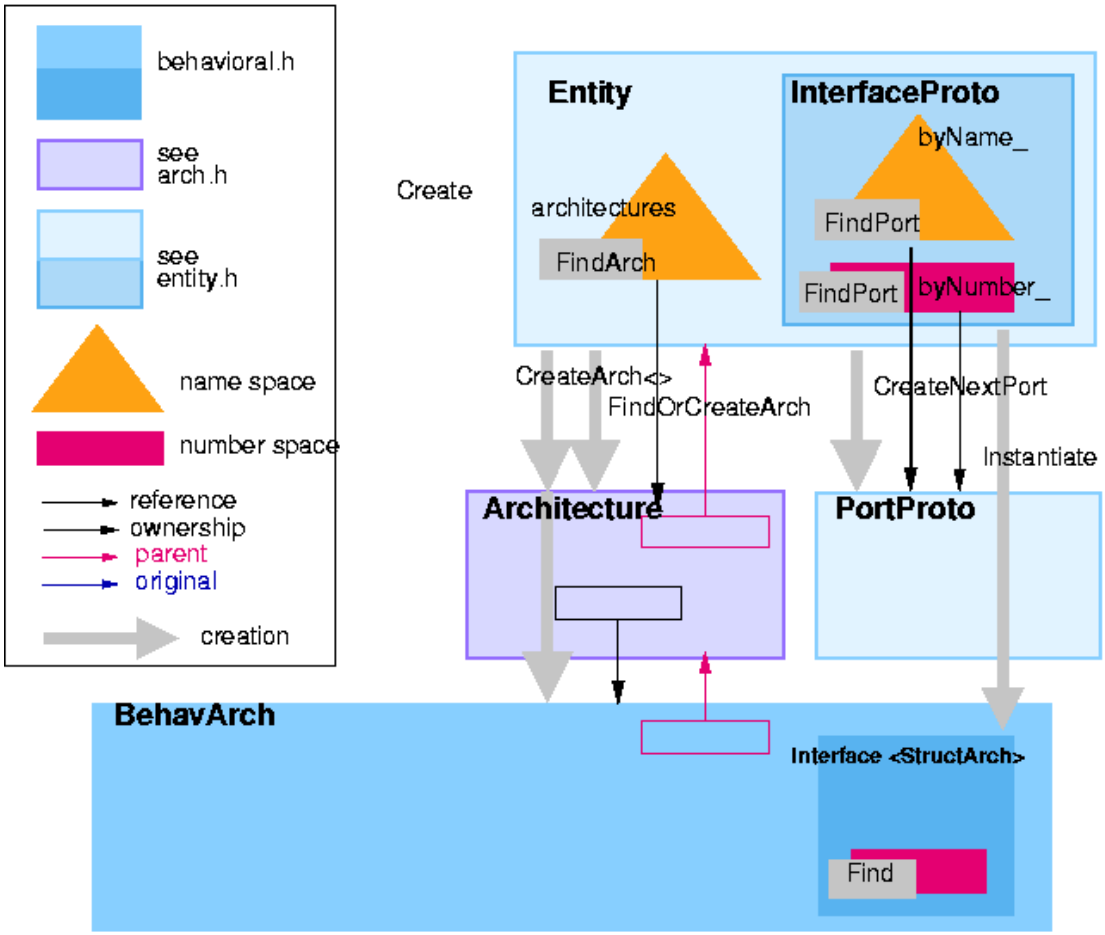
2.2.2 Behaviorální popis

Behaviorální popis, popis chováním, specifikuje chování entity (obvodu) bez podrobnějších znalostí jeho realizace. Je implementován třídou *BehavArch* (viz obrázek 2.2). Popisů chování existuje více: symetrickou funkcí, pravdivostní tabulkou, konečným automatem a dalšími. Základní hradla a konstanty jsou řešeny symetrickou funkcí, implementovány jsou následující:

- funkce bez vstupů: 0 (GND) a 1 (VCC),
- funkce s jedním vstupem: negace (INV) a buffer (BUF),
- funkce se dvěma a více vstupy: AND, OR, XOR, NAND, NOR a XNOR.

Pravdivostní tabulka je reprezentována maticí vstupů a výstupů, podobně jako ve formátu PLA (viz kapitola 3.2.1).

[4] Konečný automat, který nemá výstup.

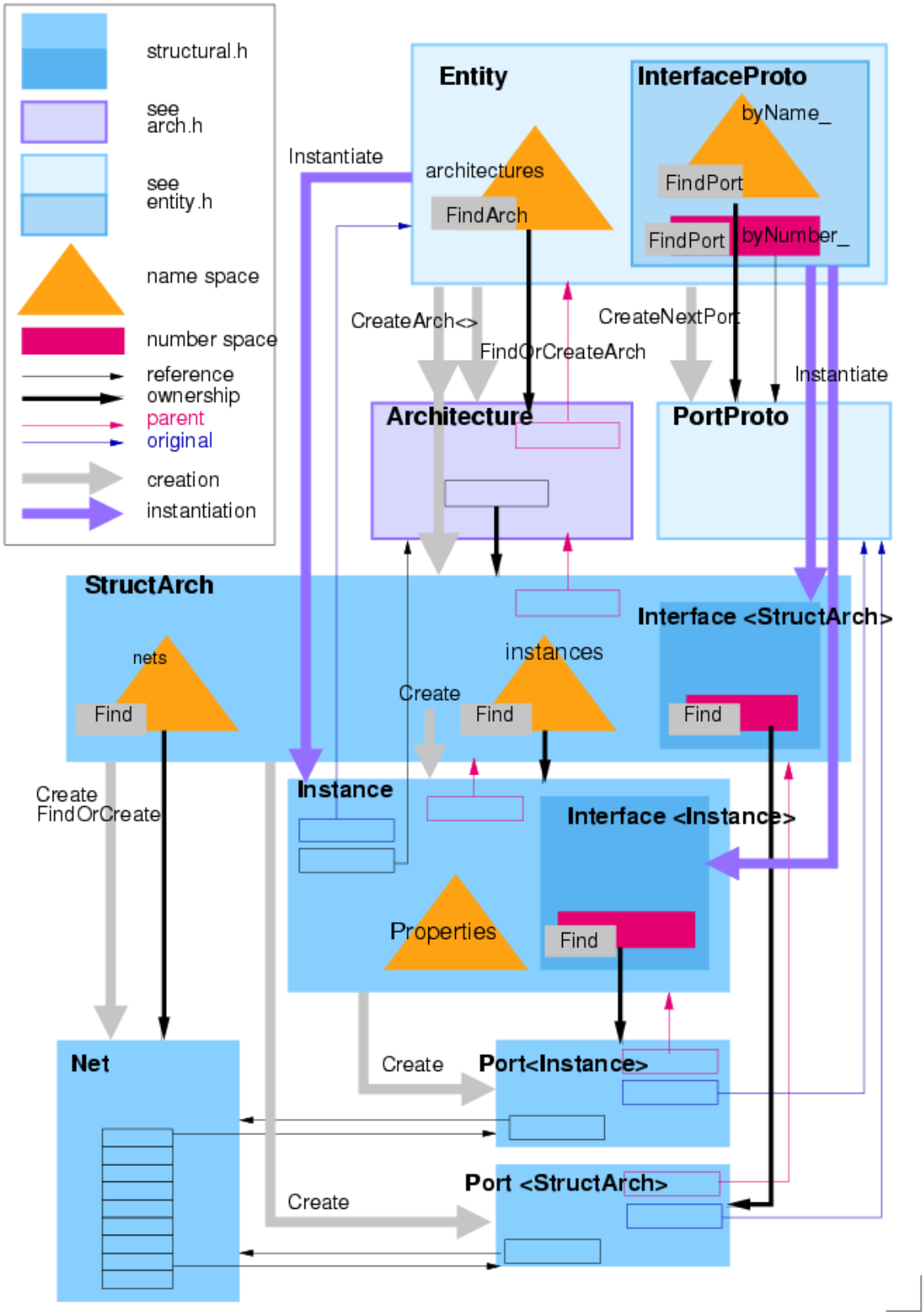


Obrázek 2.2: HUB – behaviorální popis (Schmidt)

2.2.3 Strukturní popis

Strukturním popisem se rozumí vlastní implementace entity (obvodu) za pomoci instancí jiných entit a spojů (třída *Net*), reprezentující galvanické propojení mezi porty

instanciovaných entit či mezi porty rozhraní implementované entity. Při znalosti tohoto popisu je entita snadno přenositelná a zkonstruovatelná v reálném světě. Programovou implementaci architektury popisuje třída *StructArch* (viz obrázek 2.3).



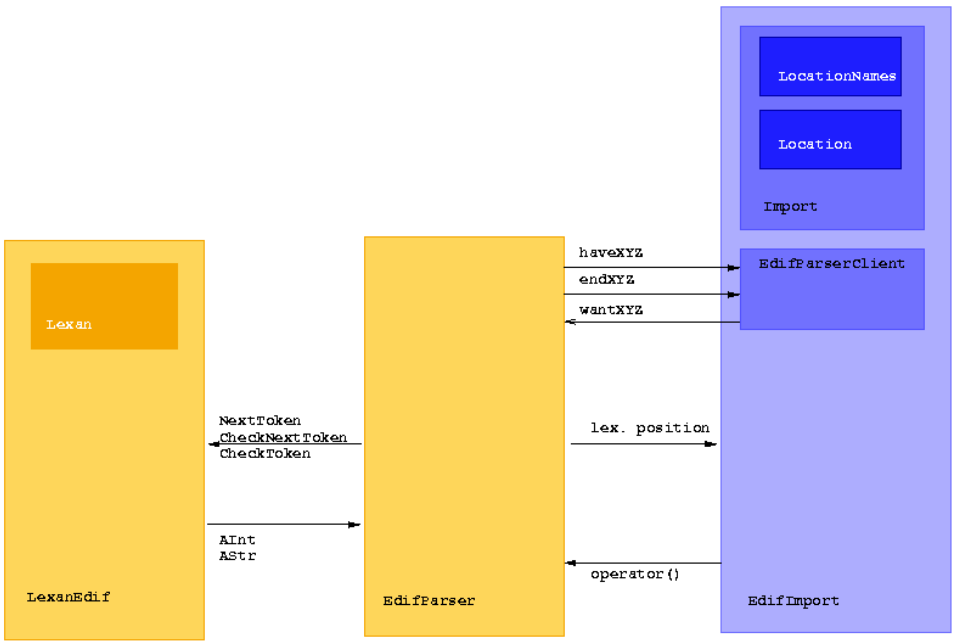
Obrázek 2.3: HUB – strukturní popis (Schmidt)

2.3 Importní podsystém

Importní podsystém, jak již název napovídá, slouží k importu specifikací logických obvodů. Rozhraní importu představuje třída *Import*, jež zahrnuje odkazy na parser (zpravidla syntaktický analyzátor), definice sémantických metod a klienta, který celý import kontroluje. V podsystému v současnosti existují třídy pro import následujících formátů:

- EDIF,
- BLIF,
- BENCH,
- VHDL,
- PLA,
- KISS.

Na obrázku 2.4 je znázorněna architektura třídy *EdifImport*, starající se o import z formátu EDIF. V implementaci dekompoziční aplikace je využíván import z formátu PLA, třída *PlaImport*.



Obrázek 2.4: EDA – EDIF import (Schmidt)

2.4 Exportní podsystém

K exportu z vnitřní reprezentace HUBu slouží exportní podsystém. Má podobnou strukturu jako importní podsystém. Oproti importnímu systému, jenž by měl být schopen známý nepoškozený formát vždy naimportovat, nemusí export zaznamenat úspěch. Příčinou je existence behaviorálního a strukturního popisu entity, které nelze implicitně převádět z jedné podoby do druhé. Tato neschopnost nepřímo iniciovala zpracování této práce. Podsystém v současnosti podporuje export do následujících formátů:

- BLIF,
- VHDL,
- PLA,
- KISS,
- kontrolní výstup DBG.

Aplikace, jež je předmětem této práce, využívá export do formátu VHDL.

3 Analýza problému

3.1 Algebraické dělení

Algebraické dělení je operace, která vypočítá rovnici podílu (dále jen podíl) Q a rovnici zbytku (dále jen zbytek) R ze zadané rovnice F a rovnice dělitele (dále jen dělitel) D tak, že F a $Q \cdot D + R$ jsou ekvivalentní rovnice. Protože podíl Q a zbytek R nejsou určeny jednoznačně, používá se algebraické dělení nazývané slabé dělení[5], které najde nejmenší zbytek R.

Před popsáním algoritmu slabého dělení je třeba definovat dělení krychlí $F \,/\, D$, kde F je rovnice a D je krychle.

$$F/D = \{(F_i - D) | F_i \in F, F_i \supset D\}$$

Obrázek 3.1 zobrazuje pseudokód algoritmu slabého dělení. Z obrázku je patrné, že podíl F/D je průnikem všech podílů F/D_i.

```
WEAK_DIV(F, D) {
      Q = ⋂_{D_i∈D} F/D_i
      R = F − D · Q
      return(Q, R)
}
```

Obrázek 3.1: Algoritmus slabého dělení (Hassoun, a další, 2002)

3.2 Kernel, co-kernel, úroveň kernelu

Pro nalezení dělitelů hraje důležitou roli pojem kernel. Pro vysvětlení tohoto pojmu je nutné definovat, kdy je rovnice *cube-free*. Rovnici nazýváme *cube-free*, pokud neexistuje krychle, která dělí danou rovnici beze zbytku. *Cube-free* rovnice se tedy musí skládat minimálně ze dvou krychlí. Primárními děliteli[6] rovnice F, značené $D(F)$, je množina podílů získaná vydělením rovnice F všemi možnými krychlemi. Kernelem rovnice F, značeným $K(F)$, je takový primární dělitel, který je *cube-free*. Krychli, jež je použita k získání kernelu, se říká co-kernel. Dalším neméně důležitým pojmem je pojem úroveň kernelu. Úroveň kernelu je definována pomocí následujících pravidel:

- pokud kernel neobsahuje žádný další kernel, je jeho úroveň rovna 0,
- pokud kernel obsahuje kernely úrovní n – 1, ale zároveň neobsahuje žádný kernel s úrovní vyšší než n – 1, je jeho úroveň rovna n.

Obrázek 3.2 obsahuje pseudokód algoritmu pro získání kernelů. Funkce KERNEL$(0, F)$ vrátí všechny kernely rovnice F. Tato funkce může být rovněž použita k vygenerování všech co-kernelů či úrovní jednotlivých kernelů. Podmínka „$\exists k \leq i, l_k \in$ all cubes of G/l_i“ je hlavním optimalizačním faktorem. Zároveň zaručuje, že žádný co-kernel nebude použit vícekrát. Proměnná l s dolním indexem představuje příslušný literál, kde n je počet unikátních literálů v rovnici G.

[5] Z anglického WEAK DIVISION.
[6] Z anglického primary divisors.

```
KERNEL(j, G) {
      R ⇐ O
      if (G is cube-free) R ⇐ {G}
      for i = j + 1, …, n {
            if (l_i appears only in one term) continue
            else {
                  if (∃k ≤ i, l_k ∈ all cubes of G/l_i) continue
                  else R ⇐ R ∪KERNEL(i, G/l_i)
            }
      }
      return R
}
```

Obrázek 3.2: Algoritmus pro získání kernelů

Tabulka 3.1 zobrazuje kernely, co-kernely a úrovně kernelů získané z následující rovnice, pod níž je uvedena ekvivalentní rovnice ve faktorizované formě.

$$F = adf + aef + bdf + bef + cdf + cef + bfg + h$$

$$F = \big((a + b + c)(d + e) + bg\big)f + h$$

Tabulka 3.1: Ukázka kernelů, co-kernelů a úrovní kernelů (Hassoun, a další, 2002)

kernel	co-kernel	úroveň
$d + e$	af, cf	0
$d + e + g$	bf	0
$a + b + c$	df, ef	0
$(a + b + c)(d + e) + bg$	f	1
$\big((a + b + c)(d + e) + bg\big)f + h$	1	2

3.3 Faktorizace

Faktorizace je proces, při kterém se vstupní rovnice rozkládá na součin menších částí. Následující příklad představuje motivaci pro faktorizaci. Formě rovnice, jež prošla faktorizací, se říká faktorizovaná forma. Z motivačního příkladu je vidět, že faktorizovaná forma je jednodušší než zadaná forma. Laicky řečeno je faktorizovaná forma uzávorkovaný algebraický výraz.

$$F = abc + abd + cd = ab(c + d) + cd$$

Proces faktorizace je znázorněn algoritmem na obrázku 3.3. Uvedená podmínka kontroluje, jestli rovnici F lze faktorizovat, například nemá-li jen jednu krychli nebo neobsahuje-li žádný literál vícekrát. Následná operace CHOOSE_DIVISOR vybírá dělitel, na jehož základě bude rovnice faktorizována. Faktorizace se provádí ve funkci dělení (DIVIDE). Výběrem dělitele se zabývá kapitola 3.5. Výstupem faktorizace je rovnice $Q \cdot D + R$, která je ekvivalentní k původní rovnici F.

```
FACTOR(F) {
      if (F has no factor) return F
      D ⟸ CHOOSE_DIVISOR(F)
      (Q, R) ⟸ DIVIDE(F, D)
      return FACTOR(Q)·FACTOR(D) + FACTOR(R)
}
```

Obrázek 3.3: Algoritmus faktorizace

Při algebraické faktorizaci se pro dělení nejčastěji používá algoritmus slabého dělení (viz výše). Jiným příkladem je booleovská faktorizace, která na rozdíl od algebraické bere v potaz proměnnou i její negaci (v algebraické faktorizaci jsou proměnná a její negace chápány jako dvě nezávislé proměnné), počítá tedy v rámci booleovské algebry. Algoritmus faktorizace je totožný, rozdíl je v použitém algoritmu dělení, a tedy i ve výběru dělitele.

3.4 Dekompozice

Dekompozice je proces velmi podobný faktorizaci, tedy rozklad logické sítě na menší celky za účelem zjednodušení. Protože je fyzická realizace mnohavstupových hradel velmi složitá, je kladen důraz na kvalitní dekompozici, jež převede logický obvod na víceúrovňovou strukturu s jednoduššími hradly. Dekomponované řešení je lépe realizovatelné a ve výsledku i mnohem efektivnější.

Jediným rozdílem oproti faktorizaci je, že je dělitel zaveden jako nový prvek do logické sítě. Jednoduchý algoritmus pro dekompozici je uveden na obrázku 3.4.

```
DECOMP(f_i) {
      K ⟸ CHOOSE_KERNEL(f_i)
      if (K == 0) return
      f_{m+j} = K
      f_i = (f_i/K)y_{m+j} + R
      DECOMP(f_i)
      DECOMP(f_{m+j})
}
```

Obrázek 3.4: Algoritmus dekompozice

Výběrem kernelu (dělitele; funkce CHOOSE_KERNEL) se zabývá kapitola 3.5. Po výběru je zaveden nový logický prvek, který je v následujícím kroku navázán do právě dekomponujícího se prvku f jako nová proměnná y. R reprezentuje zbytek rovnice, který nebyl kernelem nijak ovlivněn.

3.5 Výběr dělitelů

Výběr správného dělitele je nejvýznamnější faktor ovlivňující výslednou kvalitu v procesech faktorizace a dekompozice. Použitím správného dělitele se logický obvod mnohdy několikanásobně zjednoduší. Vzhledem k iterativnímu charakteru operací faktorizace a dekompozice se ale většinou výběrem špatného dělitele zabrání další faktorizaci či dekompozici. Celý problém faktorizace a dekompozice se tedy redukuje

na problém vybrání správného dělitele. Existuje několik známých řešení, každé má své výhody i nevýhody.

3.5.1 Výběr nejfrekventovanějšího literálu

Za dělitele je zvolen jediný literál, a to takový, který se vyskytuje v rovnici nejčastěji. Pokud takových literálů existuje více, může být použit jakýkoli z nich. Tento způsob výběru dělitele je velice rychlý, avšak kvalitativně velmi neefektivní. Příklad faktorizace pomocí výběru nejfrekventovanějšího literálu je zobrazen na obrázku 3.5.

```
zadaná forma (24 literálů):
```

$$x = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + df + dg$$

```
faktorizovaná forma (16 literálů):
```

$$x = a(c + d + e + g) + b(c + d + e + f) + c(e + f) + d(f + g)$$

Obrázek 3.5: Faktorizace nejfrekventovanějším literálem

3.5.2 Výběr prvního kernelu úrovně 0

Jako dělitel je zvolen prvně nalezený kernel úrovně 0. Tento způsob výběru je velmi rychlý, dekompozice je kvalitativně efektivnější než při výběru nejfrekventovanějšího literálu, ovšem stále ne dostatečně. Obrázek 3.6 zobrazuje faktorizaci pomocí výběru prvního nalezeného kernelu úrovně 0, pro srovnání se stejným zadáním jako na obrázku 3.5.

```
zadaná forma (24 literálů):
```

$$x = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + df + dg$$

```
faktorizovaná forma (14 literálů):
```

$$x = g(a + d) + (a + b)(c + d + e) + c(e + f) + f(b + d)$$

Obrázek 3.6: Faktorizace prvním nalezeným kernelem úrovně 0

3.5.3 Výběr kernelu s největším ziskem

Kvalitativně efektivnějším řešením je výběr takového kernelu, který znamená pro danou rovnici největší zisk. Složitost rovnice se dá do jisté míry reprezentovat počtem literálů, pokud se podaří snížit počet literálů, výsledná rovnice bude jednodušší. Ziskem se rozumí počet literálů, který ušetříme výběrem daného kernelu. Zisk je definován následujícím vztahem:

$$G = kC \cdot ckL + ckC \cdot kL - ckL - kL,$$

kde G je zisk kernelu, kC je počet krychlí kernelu, ckL je počet literálů ve všech co-kernelech, ckC je počet co-kernelů a kL je počet literálů v kernelu. Tento způsob výběru dělitele je kvalitativně efektivnější, avšak časové náročnější. Obrázek 3.7 zobrazuje faktorizaci pomocí kernelu s největším ziskem, pro srovnání je zadáním stejná rovnice jako na obrázcích 3.5 a 3.6.

zadaná forma (24 literálů):

$$x = ac + ad + ae + ag + bc + bd + be + bf + ce + cf + df + dg$$

faktorizovaná forma (14 literálů):

$$x = (c + d + e)(a + b) + f(b + c + d) + g(a + d) + ce$$

Obrázek 3.7: Faktorizace kernelem s nejvyšším ziskem

3.6 Skupinová dekompozice

Skupinová dekompozice se od dříve uváděné liší tím, že se kernel vybírá a aplikuje na více rovnic najednou. Obrázek 3.8 zobrazuje algoritmus skupinové dekompozice.

```
DECOMP(F) {
     while true {
          K ⟸ CHOOSE_KERNEL(F)
          if (K == 0) return
          for each f_i in F {
               f_i = (f_i/K)y_j + R
          }
          F = F ∪ K
     }
}
```

Obrázek 3.8: Algoritmus skupinové dekompozice

Výběr kernelu je opět hlavní faktor ovlivňující výslednou kvalitu dekompozice. Jednou metodou výběru kernelu pro skupinovou dekompozici je tzv. obdélníkové pokrytí[7].

Obdélníkové pokrytí využívá k nalezení kernelu matici co-kernelů[8], v které řádky reprezentují co-kernely a sloupce jednotlivé krychle nalezených kernelů. Prvkem této matice, M_{ij}, je krychle původní rovnice, která je ekvivalentní s násobkem odpovídajících co-kernelů a kernelů. Tabulka 3.2 zobrazuje matici co-kernelů pro následující rovnice:

$$F = af^1 + bf^2 + ag^3 + cg^4 + ade^5 + bde^6 + cde^7$$

[7] Z anglického rectangle covering; viz (Hassoun, a další, 2002).
[8] Z anglického co-kernel cube matrix; viz (Hassoun, a další, 2002).

$$G = af^8 + bf^9 + ace^{10} + bce^{11}$$

$$H = ade^{12} + cde^{13}$$

Kernely a co-kernely rovnice F jsou: $(de + f + g)/a$, $(de + f)/b$, $(a + b + c)/de$, $(a + b)/f$, $(de + g)/c$, $(a + c)/g$; rovnice G: $(ce + f)/\{a, b\}$, $(a + b)/\{f, ce\}$; a rovnice H: $(a + c)/de$.

Tabulka 3.2: Matice co-kernelů (Hassoun, a další, 2002)

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	.	.	.	.	5	1	3
F	b	2	.	.	.	.	6	2	.
F	de	3	5	6	7	.	.	.	.
F	f	4	1	2	.	.	.	.	.
F	c	5	.	.	.	.	7	.	4
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	10	.	8	.
G	b	8	.	.	.	11	.	9	.
G	ce	9	10	11	.	.	.	.	.
G	f	10	8	9	.	.	.	.	.
H	de	11	12	.	13	.	.	.	.

Obdélník (R, C) je množina řádků R a sloupců C, kde platí následující:

$$\forall i \in R, \forall j \in C, M_{ij} \neq 0$$

Při použití obdélníku (R, C) v dekompozici je zavedena nová rovnice, odpovídající součtu krychlí asociovaných se sloupci uvedenými v množině C (např. $a + b$ pro $C = \{1,2\}$), a nová proměnná, která tuto rovnici reprezentuje v již existujících. Poté jsou ze zavedených rovnic odstraněny krychle vyznačené obdélníkem (R, C) a přidány krychle co-kernelů asociovaných s řádky uvedenými v množině R, které jsou vynásobeny s nově zavedenou proměnnou (např. $dex + fx$ pro $R = \{3,4\}$, kde x je nově zavedená proměnná).

Pro vybrání nejlepšího obdélníku se zavádí ohodnocení zisku $G(R, C)$. Při odebrání krychlí, vybraných obdélníkem (R, C), z původních rovnic získáme hodnotu $V(R, C)$ takovou, že platí následující:

$$V(R, C) = \sum_{i \in R, j \in C} V_{ij},$$

kde V_{ij} je počet literálů krychle M_{ij}. Představuje tedy počet literálů, jež odebereme z původních rovnic. Aby ale byla zachována jejich ekvivalence, je nutné některé literály přidat. Jejich počet je reprezentován následujícími vztahy:

$$W_i^r = (\text{počet literálů co-kernelu spojeného s řádkou } i) + 1$$

$$W_j^c = \text{počet literálů krychle spojené se sloupcem } j$$

Výsledný počet přidaných literálů $W(R,C)$ je dán následujícím vztahem:

$$W(R,C) = \sum_{i \in R} W_i^r + \sum_{j \in C} W_j^c$$

Celkový zisk obdélníku $G(R,C)$ je pak definován následovně:

$$G(R,C) = V(R,C) - W(R,C)$$

Vybrán by měl být takový obdélník, který má nejvyšší zisk $G(R,C)$.

Při uvažování příkladu z tabulky 3.4 má největší zisk obdélník $(\{3,4,9,10\},\{1,2\})$, a to 8. Proto upravíme původní soustavu rovnic na následující, kde X je nově zavedená rovnice a x je její proměnná reprezentace.

$$F = fx + ag + cg + dex + cde$$

$$G = fx + cex$$

$$H = ade + cde$$

$$X = a + b$$

S nově zavedenou proměnnou x se v příštích iteracích zachází stejně jako s každou jinou proměnnou. Dekompozice se provádí do té doby, dokud je obdélník (R,C) nenulový.

4 Návrh řešení

4.1 Volba programovacího jazyka

Vzhledem k nutnosti použití EDA systému, jenž je napsán v jazyku C++, byl pro implementaci zvolen taktéž jazyk C++. Ten poskytuje výhody objektového programování a standardní knihovnu STL, což značně ulehčí implementaci. Jako vývojový nástroj bylo zvoleno Microsoft Visual Studio 2005.

4.2 Schéma aplikace

Aplikace nahraje PLA soubor, vytvoří z něj množinu rovnic typu součet krychlí a provede dekompozici. Volitelně následuje konverze na dvouvstupová hradla a převod na NAND hradla. Po všech úpravách se redukují redundance a výsledek je exportován do VHDL souboru. Toto schéma je zobrazeno na obrázku 4.1.

```
DECOMPOSE(PLA, b, n) {
     F ⇐ IMPORT_PLA(PLA)
     DECOMP(F)
     if (b) CONVERT_TO_2INPUT(F)
     if (n) CONVERT_TO_NAND(F)
     REDUCE(F)
     EXPORT_VHDL(F)
}
```

Obrázek 4.1: Základní schéma aplikace

4.3 Algoritmus dekompozice

Pro dekompozici byl zvolen algoritmus skupinové dekompozice popisovaný výše s několika úpravami. Aby se minimalizoval výpočet kernelů a co-kernelů, je vybíráno ze seznamu více kernelů. Seznam je průběžně aktualizován. Pseudokód popisující algoritmus dekompozice je uveden na obrázku 4.2. Algoritmus výpočtu kernelů a co-kernelů (GET_KERNELS) je shodný s algoritmem uvedeným v kapitole 3.3.

```
DECOMP(F) {
     while true {
          K ⇐ GET_KERNELS(F)
          if (K == 0) return
          while k ⇐ CHOOSE_KERNEL(K) {
               for each fi in F {
                    fi = (fi/k)yj + R
               }
               F = F ∪ k
               REMOVE_KERNELS(K, k)
          }
     }
}
```

Obrázek 4.2: Použitý algoritmus dekompozice

4.4 Algoritmus výběru kernelu

V aplikaci budou implementovány dva druhy výběru kernelů, mezi kterými se bude přepínat.

První algoritmus výběru kernelu se od výběru kernelu s nejvyšším ziskem (uvedeného v kapitole 3.5.3) liší jen minimálně. I tentokrát je vybrán kernel s nejvyšším ziskem, ale zisk je vypočítán dle následujícího vztahu:

$$G = kG(kC \cdot ckL + ckC \cdot kL - ckL - kL),$$

kde G je zisk kernelu, kC je počet krychlí kernelu, ckL je počet literálů ve všech co-kernelech, ckC je počet co-kernelů, kL je počet literálů v kernelu a kG je počet nezávislých kernelů, tedy kernelů, které jsou získány z odlišných krychlí rovnice. Uvažujeme-li následující příklad:

$$F = abc^1 + abd^2 + acd^3 = ab(c^1 + d^2) + acd^3 = ac(b^1 + d^3) + abd^2,$$

pak kernely rovnice F jsou $(c^1 + d^2)$ a $(b^1 + d^3)$. Indexy nad jednotlivými krychlemi určují krychle původní rovnice, z nichž byl kernel derivován. Je zřejmé, že po užití kernelu $(c^1 + d^2)$ již nepůjde aplikovat kernel $(b^1 + d^3)$, a to právě kvůli krychli abc^1, která se angažuje v obou kernelech. Kernely $(c^1 + d^2)$ a $(b^1 + d^3)$ jsou tedy závislé.

Zvolený algoritmus je časově náročnější, ale měl by maximalizovat počet vybraných kernelů v jediné iteraci dekompozice. Tento efekt by měl vést k menšímu počtu kernelů v dalších iteracích, což by mělo časovou složitost naopak zredukovat. Tento algoritmus byl nově navržen v rámci této práce.

Druhý implementovaný algoritmus bude výběr s ohodnocením bez proměnné kG, popisovaný v kapitole 3.5.3.

4.5 Algoritmus aktualizace kernelů

Algoritmus aktualizace kernelů odebere ze seznamu všechny závislé kernely (viz kapitola 4.4) na posledně vybraném kernelu. Jeho schéma je zobrazeno na obrázku 4.3.

```
REMOVE_KERNELS(K, k) {
     for each k_i in K {
          if (k_i depends on k) K ⇐ K \ k_i
     }
}
```

Obrázek 4.3: Použitý algoritmus aktualizace kernelů

4.6 Algoritmus konverze na dvouvstupová hradla

Cílem práce je provést dekompozici na dvouvstupová hradla. Dekompozice popisovaná výše ale nezaručí, že výsledná hradla budou dvouvstupová. Proto je zapotřebí po dekompozici provést konverzi, která to zajistí. Vstupem převodu jsou vždy hradla AND a OR (výstup dekompozice), převod pro jiná hradla tedy není nutný. Algoritmus převodu popisuje obrázek 4.4.

```
CONVERT_TO_2INPUT(G) {
     if (|G| > 2) switch (GATETYPE(G))
     {
     case AND:
          G0 ⇐ G0 · G2 · G4 · …
          G1 ⇐ G1 · G3 · G5 · …
          G ⇐ G0 · G1
     case OR:
          G0 ⇐ G0 + G2 + G4 + ⋯
          G1 ⇐ G1 + G3 + G5 + ⋯
          G ⇐ G0 + G1
     }

     for each Gi in G {
          CONVERT_TO_2INPUT(Gi)
     }
}
```

Obrázek 4.4: Použitý algoritmus převodu na dvouvstupová hradla

4.7 Algoritmus převodu na NAND hradla

Algoritmus převodu na NAND hradla je omezen na vstupní hodnoty AND a OR hradel. Konverze se provádí ze znalostí, že AND je ekvivalentní s negovaným NAND hradlem a OR je ekvivalentní s NAND hradlem s negovanými vstupy. Použitý algoritmus je zobrazen na obrázku 4.5.

```
CONVERT_TO_NAND(G) {
     switch (GATETYPE(G))
     {
     case AND:
          G ⇐ ¬(G0 ⊼ G1 ⊼ G2 ⊼ … ⊼ Gn)
     case OR:
          G ⇐ ¬G0 ⊼ ¬G1 ⊼ ¬G2 ⊼ … ⊼ ¬Gn
     }

     for each Gi in G {
          CONVERT_TO_NAND(Gi)
     }
}
```

Obrázek 4.5: Použitý algoritmus převodu na NAND hradla

4.8 Algoritmus redukce redundancí

Po dekompozici, převodu na dvouvstupová hradla a konverzi na NAND hradla vznikne v logické síti mnoho redundancí. To je zapříčiněno především tím, že je použita algebraická dekompozice, nikoli booleovská. To znamená, že proměnná a její negace se chovají jako dvě nezávislé proměnné. Z toho důvodu byl navržen algoritmus, který

některé redundance odstraní. Jedná se zejména o odebrání dvou po sobě jdoucích negací ($x = \bar{\bar{x}}$), o náhradu násobení nulou ($0 \cdot x = 0$), o sčítání jedničkou ($1 + x = 1$), o sčítání a násobení proměnné a její negace ($x \cdot \bar{x} = 0$ a $x + \bar{x} = 1$). Schéma algoritmu je zobrazeno na obrázku 4.6.

```
REDUCE(G) {
      for each G_i in G {
            REDUCE(G_i)
      }

      switch (GATETYPE(G))
      {
      case INV:
            if (G == x̄̄) G = x
      case AND:
            if (G contains 0 or x·x̄) G = 0
      case OR:
            if (G contains 1 or x+x̄) G = 1
      case NAND:
            if (G contains 0 or x·x̄) G = 1
      }
}
```

Obrázek 4.6: Použitý algoritmus redukce redundancí

5 Popis implementace

5.1 Metoda main()

Metoda *main()* implementuje základní schéma aplikace. Kromě zjišťování aktivních přepínačů aplikace vytváří instanci třídy *Hub*, a poté importuje vstupní soubor PLA pomocí třídy *PlaImport*, tedy využívá funkcí EDA systému. V dalším kroku je vytvořena instance třídy *Decomposer*, která je zodpovědná za vlastní dekompozici; ta se provede zavoláním operátoru (); třída *Decomposer* byla nově navržena a implementována v rámci této práce. Po úspěšném dokončení je výsledek exportován do VHDL souboru pomocí třídy *VhdlExport*, která je opět součástí EDA systému. Třída *Decomposer* vyžaduje několik pomocných struktur, mezi které patří zejména třída *Gate*.

5.2 Třída *Gate*

Základní třídou reprezentující hradlo je třída *Gate*. Tato třída byla navržena v rámci této práce. Vstupy hradla jsou ukazatele na třídu *Gate*, jedná se tedy spíše o reprezentaci celé logické sítě než jednoho hradla. Tato třída obsahuje všechny potřebné údaje o hradlu, stejně jako všechny potřebné funkce, jejichž pomocí se s hradlem manipuluje. Pro implementaci třídy byly zavedeny nové typy, jako výčtový typ *GateType* a seznam hradel *Gates*. Aby byla hradla snadněji porovnávána, co se týče struktury, jsou jejich vstupy řazeny dle typu a portů. Obrázek 5.1 zobrazuje deklaraci výčtového typu *GateType* a definici typu *Gates*.

```
enum GateType
{
      FALSE=0,
      TRUE,
      PORT,
      INVPORT,
      BUF,
      INV,
      AND,
      OR,
      XOR,
      NAND,
      NOR,
      XNOR
};

class Gate;

struct GateComp
{
      bool operator()(const Gate* u,const Gate* v) const;
};

typedef std::set<Gate*,GateComp> Gates;
```

Obrázek 5.1: Výčtový typ *GateType* a typ *Gates*

Struktura *GateComp* je zavedena jen za účelem srovnávání struktury hradel, k čemuž používá statickou funkci *gateComparator* třídy *Gate*. Ta porovnává v první řadě typy hradel; pokud jsou stejná, bere se ohled na již seřazená vstupní hradla. Tabulka 5.1 popisuje jednotlivé proměnné třídy *Gate*.

Tabulka 5.1: Proměnné třídy *Gate*

datový typ	**název**	**popis**
GateType	type	určuje typ hradla, nabývá hodnot FALSE, TRUE, PORT, INVPORT, BUF, INV, AND, OR, XOR, NAND, NOR, XNOR; některé typy nejsou funkcemi podporovány, protože se v dekompozici nevyskytují
int	portNumber	jestli je hradlo typu PORT, nebo INVPORT, přesněji identifikuje port číslem
Gates	inputs	obsahuje seznam vstupních hradel
bool	disabled	proměnná určující, jestli je hradlo propojeno, používá se při výpočtu kernelů, při dělení
std::set<int>	disablingPorts	jestliže hradlo není propojeno (viz disabled výše), obsahuje seznam portů, jež hradlo blokují
*Net**	net	slouží k určení spoje ve strukturním popisu entity; využívá se při stavění této struktury
GateCache	cache	obsahuje hodnoty, které se za normálních okolností počítají, např. jestli je hradlo součin, součet součinů, cube-free atd.

Všechny funkce, manipulující s třídou *Gate*, jsou založené na rekurzivním volání pro všechna vstupní hradla; některé vypočtené hodnoty jsou uloženy do cache, aby nemusely být při příštím volání počítány znovu. Mezi stěžejní funkce patří funkce *reduceGates*, *convertTo2Input* a *convertToNAND*, jež jsou popsány v návrhu řešení. Implementace se od návrhu nijak neliší.

5.3 Třída *Decomposer*

Stěžejní třídou, jejímž návrhem a implementací se zabývá tato práce, je třída *Decomposer*, která je zodpovědná za provedení dekompozice, obsahuje funkce pro výpočet kernelů, jejich ohodnocení, výběr a následné nahrazení v síti hradel. Neméně důležitými funkcemi jsou i funkce zodpovědné za vytvoření sítě hradel z behaviorálního popisu entity a vytvoření strukturního popisu entity z této sítě hradel (reprezentované třídou *Gate*). Všechny funkce jsou implementovány přesně dle návrhů v kapitole 4. Pro účely testování přibyla funkce, která vypočítá výstupní logickou hodnotu hradla pro zadané vstupní hodnoty.

Datové struktury popisující krychli, kernel, co-kernel a další jsou uvedeny na obrázku 5.2. Využitím struktur standardní knihovny je způsobena velká časová složitost, jež je minimálně redukována definicí vlastních komparátorů.

```
typedef std::set<int> Cube;

struct CubeComp
{
     bool operator()(const Cube& x,const Cube& y) const;
};

typedef std::set<Cube,CubeComp> Kernel;
typedef std::set<Cube,CubeComp> CoKernels;

struct KernelComp
{
     bool operator()(const Kernel& x,const Kernel& y)
const;
};

struct KernelValue
{
     KernelValue();

     Kernel kernel;
     Gate* function;
     CoKernels coKernels;
     Gates representation;
     int gainCache;
};

typedef std::map<Gate*,KernelValue> KernelValues;
typedef std::map<Kernel,KernelValues,KernelComp> Kernels;

struct KernelLiteralComp
{
     bool operator()(const Kernel& x,const Kernel& y)
const;
};

typedef std::set<Kernel,KernelLiteralComp> KernelsSize;
```

Obrázek 5.2: Pomocné datové struktury

Datovou strukturou pro seznam kernelů je typ *Kernels*. Jedná se o mapu, kde klíčem je kernel a hodnotou jsou struktury *KernelValue*, roztříděné dle jednotlivých dekomponovaných rovnic. Časově nejnáročnější je vyhledávání nejlepšího možného kernelu v tomto seznamu typu *Kernels*, proto je zavedena optimalizace, která omezuje počet vypočtených kernelů tak, že upřednostňuje jednodušší (kratší) kernely. Ty jsou dle zkušeností efektivnější pro proces dekompozice. K tomu je zaveden pomocný typ *KernelsSize*, datový typ seznamu kernelů řazeného dle složitosti. Vyhledávání v malém počtu kernelů je pak relativně rychlé.

6 Testování

Testy byly prováděny na nejrůznějších testovacích úlohách, jež jsou obsaženy na přiloženém CD. Výsledné struktury byly vždy ekvivalentní k vstupním PLA souborům, a lze je tedy považovat za akceptovatelné. Toto tvrzení se opírá o statistický výstup aplikace, který obsahuje i hodnotu, jež značí, jestli je obvod ekvivalentní k vstupní pravdivostní tabulce. Tato hodnota je vypočítána algoritmem, při němž se postupně zkusí dosadit jednotlivé řádky vstupní tabulky a výstupní hodnota se porovná s očekávanou hodnotou, jež je opět uvedena ve vstupní tabulce PLA. Srovnáním efektivity se zabývá kapitola 7. Následuje výpis ukázkového vstupu (viz obrázek 6.1) a výstupu (viz obrázek 6.2). Statistický soubor pro tento testovací soubor je k dispozici v příloze D.4.

```
.i 3
.o 2
000 00
001 00
010 00
011 00
100 10
101 11
110 10
111 11
```

Obrázek 6.1: Výpis testovacího vstupního PLA souboru

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;

entity test is
  port (
    I0 : in std_logic;
    I1 : in std_logic;
    I2 : in std_logic;
    O0 : out std_logic;
    O1 : out std_logic
  );
end entity test;

architecture gatenet of test is
  signal net_0 : std_logic;
  signal net_1 : std_logic;
begin
  net_0 <= not net_1;
  net_1 <= i0 nand i2;
  O0 <= i0;
  O1 <= net_0;
end architecture gatenet;
```

Obrázek 6.2: Výpis testovacího výstupního VHDL souboru

7 Srovnání s existujícími řešeními

V rámci srovnání implementovaných algoritmů bylo použito několik vybraných testovacích úloh pro dekompozici. Srovnání probíhalo na platformě MS Windows. Aplikace decompose byla spouštěna z příkazové řádky příkazem na obrázku 7.1 (významy jednotlivých přepínačů je uveden v příloze D).

```
decompose –b –s –i fileName [-g]
```

Obrázek 7.1: Spuštění dekompozice v implementované aplikaci

Pro spuštění dekompozice v MVSIS 2.0 byl do aplikace zadán sled příkazů zobrazený na obrázku 7.2.

```
read_pla fileName
decomp
simplify
mfs
```

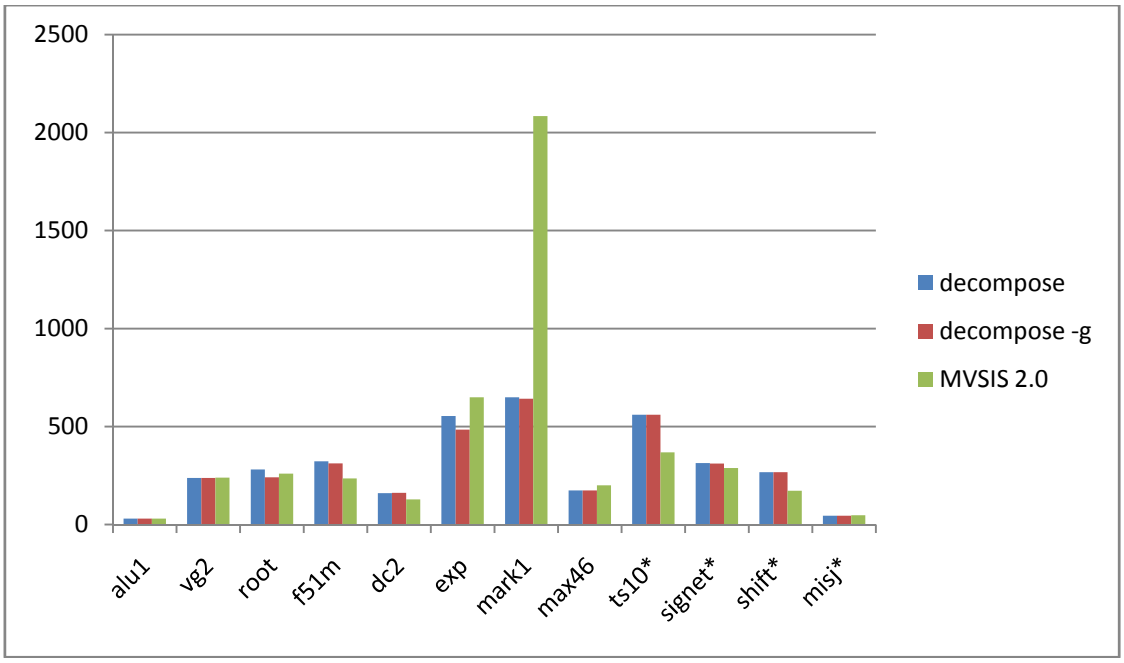
Obrázek 7.2: Spuštění dekompozice v aplikaci MVSIS 2.0

Výsledky testování jsou uvedeny v tabulce 7.1. Výsledky implementované práce jsou pojmenovány jako „decompose“; jednou za využití nově navrženého algoritmu (bez přepínače –g) a jednou za využití algoritmu převzatého z literatury (s přepínačem –g). Benchmarky označené hvězdičkou jsou označené jako velmi náročné pro dekompozici.

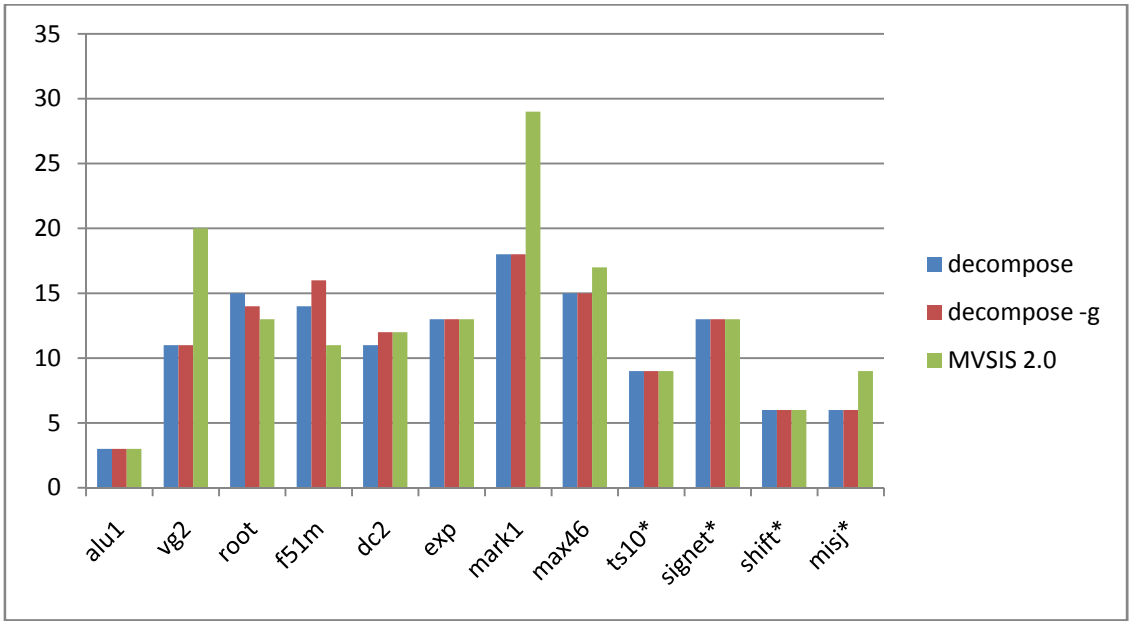
Tabulka 7.1: Výsledky srovnávání

		decompose			**decompose -g**			**MVSIS 2.0**		
	literálů	hradel	úrovní	čas [s]	hradel	úrovní	čas [s]	hradel	úrovní	čas [s]
alu1	41	30	3	<0,001	30	3	<0,001	30	3	<0,001
vg2	757	238	11	0,39	238	11	0,37	239	20	0,13
root	4920	281	15	13,33	241	14	12,39	260	13	0,73
f51m	8192	323	14	51,25	312	16	18,00	235	11	0,81
dc2	548	160	11	0,30	162	12	0,28	128	12	0,03
exp	2376	554	13	2,76	484	13	2,06	649	13	1,08
mark1	1580	649	18	43,90	642	18	43,73	2084	29	2,63
max46	395	174	15	1,03	174	15	1,14	200	17	0,28
ts10*	896	560	9	0,23	560	9	0,22	368	9	0,34
signet*	590	314	13	0,42	311	13	0,39	288	13	1,01
shift*	512	267	6	0,13	267	6	0,09	172	6	0,08
misj*	77	45	6	0,02	45	6	0,02	47	9	0,02

Z uvedené tabulky je patrné, že obě uvedené aplikace dosahují velmi podobných výsledků v kvalitě, ale časově je aplikace MVSIS 2.0 lépe optimalizována. Pro lepší přehlednost je tabulka převedena do sady grafů na obrázcích 7.3, 7.4 a 7.5.

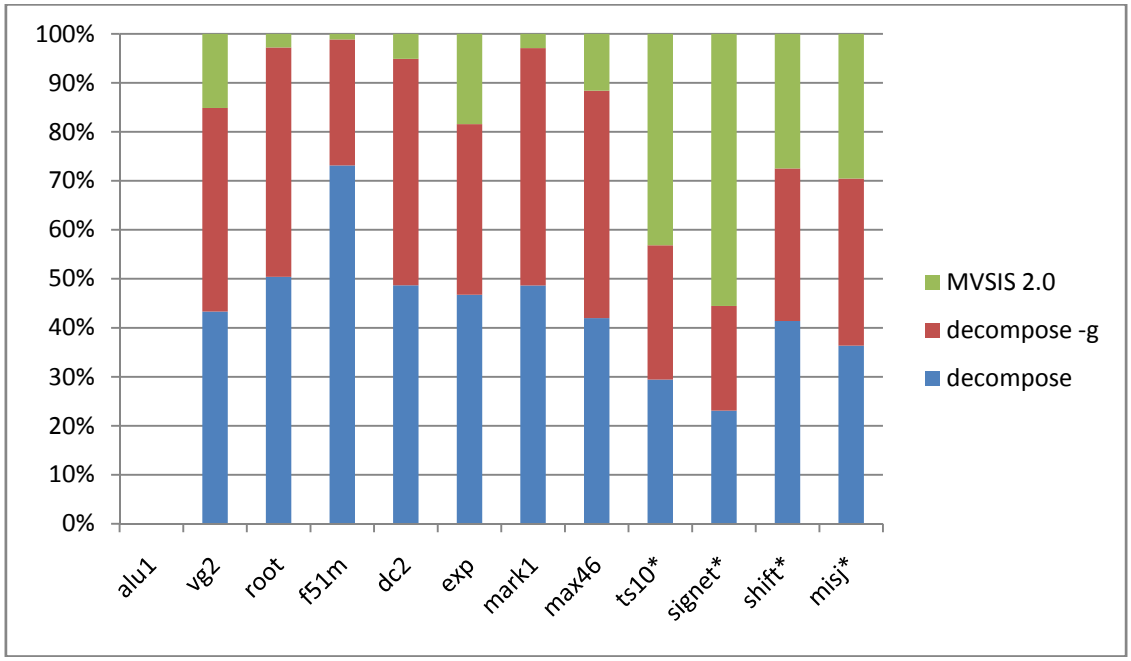


Obrázek 7.3: Výsledné počty hradel



Obrázek 7.4: Výsledné počty úrovní

Následující graf, zobrazující časovou složitost jednotlivých aplikací, je vynesen tak, aby byl jasně a zřetelně vidět poměr mezi výslednými časy dekompozice jednotlivých aplikací.



Obrázek 7.5: Výsledná časová složitost

Z porovnání nově navrženého algoritmu výběru kernelů a algoritmu převzatého z literatury vyšel lépe algoritmus uváděný v literatuře, a to jak z časového, tak z kvalitativního hlediska. Nemyslím si však, že je rozdíl nějak podstatný.

8 Závěr

V průběhu řešení práce byl navržen a implementován nový algoritmus v rámci algebraické faktorizace. Do aplikace, jež byla náplní práce, byl zaimplementován tento nově navržený algoritmus stejně jako algoritmus uváděný v literatuře a celá aplikace byla zaintegrována do EDA systému v rozsahu zadání práce. Aplikace byla otestována a splnila všechny funkční požadavky, tedy dekompozici na dvouvstupová hradla a napojení na importní a exportní podsystémy EDA systému. Výsledná síť hradel byla vždy ekvivalentní k zadané pravdivostní tabulce.

Druhým cílem bylo implementovanou aplikaci srovnat s existujícími řešeními, zastoupenými aplikací MVSIS 2.0. Z výsledků testování se ukázala srovnatelná efektivita kvality dekompozice s navrženou aplikací, na druhou stranu byla aplikace MVSIS 2.0 časově efektivnější. Rozdíl mezi nově navrženým algoritmem a algoritmem uváděným v literatuře byl zanedbatelný.

Časová optimalizace je námětem pro budoucí zefektivnění aplikace, především využitím jiných datových struktur.

Hlavní zkušeností, kterou jsem si z práce odnesl, je spolupráce s EDA systémem, tedy s aplikací, která je neustále ve vývoji a je psaná jinou formou, než jsem zvyklý z vlastních aplikací.

9 Seznam použité literatury

MCNC. [Online] [Citace: 3. 7 2007.] ftp://ic.eecs.berkeley.edu.

File Format: espresso. [Online] [Citace: 15. 7 2007.] http://www.cs.indiana.edu/classes/c421/man/espresso.5.html.

Halamíček, M. 2006. *Vstupní a výstupní konverze VHDL.* Praha : Bakalářská práce, 2006.

Hassoun, S., Sasao, T. a Brayton, R. K. 2002. *Logic synthesis and verification.* Boston/Dordrecht/London : Kluwer Academic Publishers, 2002.

MVSIS: Gao, M., a další. June 2001. In the Notes of the International Workshop on Logic Synthesis. Tahoe City, June 2001.

Schmidt, J. Dokumentace k EDuArdovi.

Vhdl language reference guide. [Online] [Citace: 15. 7 2007.] https://service.felk.cvut.cz/doc/vhdl/Refguide.htm.

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

A. Obsah přiloženého CD

- data/ testované PLA, aplikace MVSIS 2.0
- exe/ přeložený program, exotické .dll, uživatelský manuál
- src/ zdrojové soubory aplikace
- text/ text této práce
- readme.txt základní informace

B. Formát PLA

Formát PLA reprezentuje logickou funkci pravdivostní tabulkou. Jedná se o textový formát, který je načítán po řádcích. Znak # označuje zbytek řádku jako komentář. Na začátku souboru je několik klíčových slov určujících strukturu a vlastnosti logické funkce. V tabulce B.1 je uveden seznam nejdůležitějších klíčových slov.

Tabulka B.1: Klíčová slova formátu PLA

klíčové slovo	**význam**
.model [s]	definuje jméno popisovaného obvodu
.i [d]	označuje počet vstupních proměnných
.o [d]	označuje počet výstupních proměnných
.ilb $i_1, i_2, i_3, \ldots, i_n$	pojmenovává vstupní proměnné
.ob $o_1, o_2, o_3, \ldots, o_n$	pojmenovává výstupní proměnné
.type [s]	určuje typ zadání logické funkce
.p [d]	určuje celkový počet termů
.e (.end)	označuje konec souboru

Jednotlivé typy zadání logické funkce uvádí tabulka B.2. Libovolná logická funkce je kompletně popsána pomocí ON-setu, OFF-setu a DC-setu.

Tabulka B.2: Typy zadání logické funkce ve formátu PLA

typ	**popis zadání logické funkce**
f	popisuje logickou funkci pomocí ON-setu, OFF-set je dopočítán, DC-set prázdný
fd	výchozí typ; popisuje logickou funkci pomocí ON-setu a DC-setu, OFF-set je dopočítán
fr	popisuje logickou funkci pomocí ON-setu a OFF-setu, DC-set je dopočítán
fdr	popisuje logickou funkci pomocí ON-setu, OFF-setu i DC-setu

Klíčová slova v záhlaví souboru jsou následována termy v součinovém tvaru. První část řádku je určena pro vektor vstupních proměnných v termu. Druhá část řádku je vyhrazena pro vektor výstupních proměnných. Vstupní a výstupní proměnné mohou nabývat těchto hodnot: „0“, „1“ (synonymem je „4“), „-“ (bez významu, synonymem je „2“), „~“ (don’t care, synonymem je „3“). Obrázek 6.1 uvádí výpis ukázkového PLA souboru.

```
# 2-bit by 2-bit binary adder (with no carry input)
.i 4
.o 3
.ilb a1 a0 b1 b0
.ob s2 s1 s0
0000  000
0001  001
0010  010
0011  011
0100  001
0101  010
0110  011
0111  100
1000  010
1001  011
1010  100
1011  101
1100  011
1101  100
1110  101
1111  110
```

Obrázek B.1: Ukázkový PLA soubor

C. Formát VHDL

Jazyk formátu VHDL je velmi rozsáhlý, proto bude popsána jen malá podmnožina nutná k ověření správnosti převodu. Odkaz na kompletní popis formátu VHDL je uveden v použité literatuře.

Jazyk VHDL je určen k použití ve všech fázích návrhu elektronických systémů. Umožňuje popisovat logické obvody na různé úrovni abstrakce. Základní jednotkou abstrakce je entita a její architektura, struktura je velmi podobná EDA systému (viz kapitola 2), který se nechal inspirovat právě jazykem VHDL.

Podobně jako jiné objektové jazyky nabízí i jazyk VHDL možnost používat již vytvořené komponenty seskupené do knihoven (library) a balíků (package). Pro užití komponent v určitém balíku je zapotřebí nejprve definovat knihovnu (příkazem library) a posléze i balík (příkazem use), z kterého se bude čerpat, jak je vidět na obrázku C.1.

```
library IEEE;
use ieee.Std_Logic_1164.all;
use ieee.numeric_std.all;
```

Obrázek C.1: Formát VHDL – použití knihoven

C.1 Deklarace entity

Deklarace entity zahrnuje jméno entity a rozhraní, jehož pomocí komunikuje s okolím. Rozhraní je reprezentováno množinou portů s následujícími parametry:

- název portu,
- mód portu,
- datový typ portu.

Následující příklad (obrázek C.2) deklaruje entitu jménem „AND2“, jejíž rozhraní tvoří vstupní porty *I1* a *I2* a výstupní port *O*. Všechny porty jsou datového typu *std_logic*.

```
entity AND2 is
  port (
    I1 : in std_logic;
    I2 : in std_logic;
    O : out std_logic
  );
end entity AND2;
```

Obrázek C.2: Formát VHDL – deklarace entity

C.2 Deklarace architektury

Architektura, podobně jako v EDA systému, popisuje vnitřní uspořádání entity. K realizaci využívá již vytvořené či knihovní komponenty (entity). Jazyk VHDL je tedy hierarchický. Deklarace architektury zahrnuje jméno architektury, deklaraci signálů,

reprezentujících galvanické spoje, a tzv. *data-flow* popis architektury. Signál má podobně jako port svůj název a datový typ.

Následující příklad (obrázek C.3) deklaruje architekturu „gatenet“ popisující entitu „AND2“. Je deklarován signál *out* datového typu *std_logic*. Tomuto signálu (proměnné) je přiřazena hodnota odpovídající součinu vstupů entity a poté je tato hodnota dále propagována do výstupního portu entity. Signál *out* je v tomto případě zcela zbytečný, hodnota se může přiřadit přímo na výstup, v příkladu je jen pro ilustraci.

```
architecture gatenet of AND2 is
  signal out : std_logic;
begin

  out <= I1 and I2;
  O <= out;
end architecture gatenet;
```

Obrázek C.3: Formát VHDL – deklarace architektury

D. Uživatelský manuál

Aplikace decompose provádí dekompozici ze zadané pravdivostní tabulky PLA do sítě hradel zastoupené formátem VHDL. K tomuto účelu využívá algebraickou faktorizaci.

D.1 Komponenty

Aplikace decompose obsahuje následující části systému EDuArd:

- hub,
- import,
- export,
- util.

D.2 Spuštění dekompozice

Dekompozice se spouští souborem decompose.exe, jehož předvolby jsou uvedeny v tabulce D.1.

Tabulka D.1: Přepínače aplikace decompose

přepínač	**popis**
-i fileName	vstupní soubor ve formátu PLA
-b	určuje, jestli se provede převod na dvouvstupová hradla
-d	do statistického souboru jsou přidána data o jednotlivých iteracích dekompozice (musí být použito s přepínačem -s)
-g	použije se klasické ohodnocení kernelů, viz kapitola (3.5.3)
-h	zobrazí nápovědu programu
-n	určuje, jestli se provede konverze na NAND hradla
-o %d	číslo definující stupeň optimalizace; maximální počet kernelů v seznamu; -1 pro vypnutí
-s	vygeneruje statistický soubor s údaji o provedené dekompozici

Windows:
Knihovna Z (v současné distribuci ZLIB1D.DLL) musí být dostupná.

D.3 Generovaný statistický soubor (-s)

Výstup dekompozice je kromě souboru s příponou *.vhdl* (obsahujícího síť hradel ekvivalentní s vstupním souborem ve formátu PLA) i soubor s příponou *.txt,* ve kterém jsou zobrazeny nejrůznější statistické informace o dekompozici. Struktura tohoto souboru je popsána v následujících odstavcích.

Jako první je v souboru uvedena legenda, která popisuje označení použitých logických operátorů. Následuje přepis tabulky PLA do algebraické formy, přesněji do součtu krychlí. Každá výstupní proměnná má u sebe statistické údaje o počtu literálů, počtu hradel (včetně hradel potřebných k negaci vstupních proměnných) a úrovni logické sítě.

V případě použití přepínače –d následuje výpis jednotlivých iterací dekompozice. Každá iterace je označena počtem nalezených kernelů, počtem použitých kernelů, časem provádění iterace a výpisem jednotlivých rovnic, doplněných o jednotlivé

kernely. V poslední iteraci je zobrazen výpis výstupních rovnic před redukcí redundancí opět se statistickými údaji o počtu literálů, hradel a úrovni logické sítě.

Následuje výpis výstupních rovnic po aplikaci algoritmu redukce, označené jako *PRE-OUTPUTS*. Poslední výpis výstupních rovnic, po převodu na dvouvstupová hradla, konverzi na NAND hradla a po poslední redukci redundancí, je označen jako *FINAL OUTPUTS*.

Posledním záznamem v souboru jsou obecné informace o provedené dekompozici. Patří sem celkový počet hradel, počet úrovní, čas potřebný pro dekompozici a údaj, jestli výsledná struktura prošla ověřením funkčnosti, tj. jestli odpovídá vstupnímu PLA souboru.

D.4 Výpis ukázkového statistického souboru

```
Legend:
INV  ... !
AND  ... *
OR   ... +
XOR  ... ^
NAND ... &
NOR  ... |
XNOR ... @

------------------------------------------------------------
INPUTS:
------------------------------------------------------------

o0 = ((a*b*c)+(a*b*!c)+(a*c*!b)+(a*!b*!c))

literals count [o0]:     12
gates count [o0]:        7
levels [o0]:             2

o1 = ((a*b*c)+(a*c*!b))

literals count [o1]:     6
gates count [o1]:        4
levels [o1]:             2

------------------------------------------------------------
----->***DEBUG INFO***
------------------------------------------------------------

***NEXT ITERATION
kernels found:      3
kernels used:       1
```

```
iteration time:     0 ms

o0 = ((a*c*d)+(a*d*!c))
o1 = ((a*c*d))
d = ((b)+(!b))

***NEXT ITERATION
kernels found:      1
kernels used:       1
iteration time:     0 ms

o0 = ((a*d*e))
o1 = ((a*c*d))
d = ((b)+(!b))
e = ((c)+(!c))

***NOT REDUCED
o0 = ((a*((b)+(!b))*((c)+(!c))))

literals count [o0]:     5
gates count [o0]:        12
levels [o0]:             4

o1 = ((a*c*((b)+(!b))))

literals count [o1]:     4
gates count [o1]:        7
levels [o1]:             4

------------------------------------------------------------
<-----***DEBUG INFO***
------------------------------------------------------------
------------------------------------------------------------
PRE-OUTPUTS:
------------------------------------------------------------
o0 = a

literals count [o0]:     1
gates count [o0]:        0
levels [o0]:             0

o1 = (a*c)

literals count [o1]:     2
```

```
gates count [o1]:        1
levels [o1]:             1

------------------------------------------------------------
FINAL OUTPUTS:
------------------------------------------------------------
o0 = a

literals count [o0]:     1
gates count [o0]:        0
levels [o0]:             0

o1 = !(a&c)

literals count [o1]:     2
gates count [o1]:        2
levels [o1]:             1

------------------------------------------------------------
GENERAL INFO:
------------------------------------------------------------
total gates count:  2 = 2 (gates) + 0 (inv ports)
levels:             1
decomposition time: 0 ms
verified:           1
```