VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

APROXIMACE OBVODŮ V NÁSTROJI YOSYS

APPROXIMATION OF DIGITAL CIRCUITS IN YOSYS TOOL

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **LUKÁŠ PLEVAČ**
AUTHOR

VEDOUCÍ PRÁCE **MRÁZEK VOJTĚCH, Ing., Ph.D.**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání bakalářské práce

Student:	**Plevač Lukáš**
Program:	Informační technologie
Název:	**Aproximace obvodů v nástroji Yosys**
	Approximation of Digital Circuits in Yosys Tool
Kategorie:	Počítačová architektura

Zadání:

1. Seznamte se s tématem přibližného počítání, kartézským genetickým programováním a jeho využitím pro aproximaci obvodů.
2. Seznamte se se syntézním nástrojem Yosys, interní reprezentací obvodů a možnostmi jeho rozšíření.
3. Zpracujte studii na výše uvedená témata.
4. Navrhněte rozšíření nástroje Yosys, který bude umožňovat aproximovat aritmetické obvody pomocí Kartézského genetického programování.
5. Navržené rozšíření implementujte s ohledem na obecnost použité reprezentace.
6. Pomocí sady testovacích obvodů vyhodnoťte výkonnost navrženého systému.

Literatura:

- Dle pokynů vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Mrázek Vojtěch, Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	11. května 2022
Datum schválení:	29. října 2021

Abstrakt

Cílem této práce je představení rozšíření cgploss, které slouží k optimalizaci kombinačních obvodů v nástroji Yosys. V první části práce bude představena metoda Kartézského genetického programování, která lze použít na návrh a optimalizaci obvodů. Tato kapitola dále popisuje možné reprezentace kombinačních obvodů pro Kartézské genetické programování. Následuje představení nástroje Yosys z uživatelského i implementačního hlediska a popis tvorby rozšíření pro tento nástroj. Následující kapitola popisuje návrh rozšíření cgploss a jeho vnitřní struktury. Dále je popisována implementace rozšíření a jeho ovládáná. V závěru práce je otestována funkčnost nástroje a jednotlivé použité reprezentace obvodu jsou porovnány mezi sebou.

Abstract

The goal of this work is introduction of cgploss extension. This extension is extension for combinational logic circuits optimization in Yosys tool. Cartesian genetic programming is introduced in the first part of this work. Cartesian genetic programming is a design and optimization method that can be used for circuit optimization and approximation. This chapter introduces representation of combinational logic circuits for Cartesian genetic programming. The next chapter introduces Yosys tool and possibilities of the Yosys extending. The proposed 'cgploss' extension is introduced in the next chapter. The chapter also provides details about the implementation and the usage. The last chapter tests cgploss extension and compares representation of combinational logic circuits.

Klíčová slova

kombinační obvod, optimalizace, optimalizace kombinačních obvodů, logické hradlo, Kartézské genetické programování, CGP, AIG, MIG, hradlová reprezentace, And-inverter graph, Majority-Inverter Graph, Yosys, Verilog

Keywords

combinational circuit, optimization, combinational circuits optimization, logic gate, Cartesian genetic programming, CGP, AIG, MIG, logic gates representation, And-inverter graph, Majority-Inverter Graph, Yosys, Verilog

Citace

PLEVAČ, Lukáš. *Aproximace obvodů v nástroji Yosys.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Mrázek Vojtěch, Ing., Ph.D.

Aproximace obvodů v nástroji Yosys

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vojtěcha Mrázka, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Lukáš Plevač
10. května 2022

Poděkování

Rád bych tímto poděkoval svému vedoucímu Ing. Vojtěchovi Mrázkovi, Ph.D. za čas strávený při konzultacích, odborné rady k práci i za spuštění testovacích skriptů na školním clusteru.

Obsah

1 Úvod **3**

2 Kartézské genetické programování **5**
- 2.1 Genotyp 5
- 2.2 Operátory 6
  - 2.2.1 Mutace 6
  - 2.2.2 Reprodukce 6
  - 2.2.3 Křížení 7
- 2.3 Algoritmus 7
  - 2.3.1 Kroky 7
- 2.4 Reprezentace kombinačního obvodu pro CGP 8
  - 2.4.1 Úroveň hradel 9
  - 2.4.2 And-Inverter Graph 11
  - 2.4.3 Majority-Inverter Graph 12
  - 2.4.4 XOR-Majority Graph 13
  - 2.4.5 XOR-And-Inverter Graph 13
- 2.5 Využití CGP pro aproximaci obvodů 13
  - 2.5.1 Zdrojově orientovaná strategie 13
  - 2.5.2 Chybově orientovaná strategie 14
  - 2.5.3 Vícecílová strategie 14

3 Nástroj YOSYS **15**
- 3.1 Ovládání 15
- 3.2 Vnitřní formáty a reprezentace obvodu 17
  - 3.2.1 Register-Transfer-Level-Intermediate-Language (RTLIL) 17
  - 3.2.2 RTLIL::Design a RTLIL::Module 17
  - 3.2.3 RTLIL::Cell 18
  - 3.2.4 RTLIL::Wire 18
  - 3.2.5 RTLIL::SigSpec 18
  - 3.2.6 Vnitřní cell knihovna 19
- 3.3 Tvorba rozšíření pro YOSYS 19

4 Návrh rozšíření aproximaci obvodů **21**
- 4.1 Objektový návrh aplikace 21
- 4.2 Parametry 23
- 4.3 Výběr reprezentací obvodů 23
- 4.4 Návrh Genomu 23
- 4.5 Návrh reprezentace obvodu 25

4.6 Návrh generace . . . 25
4.7 Generování kombinací pro paralelní simulaci a výpočet chyby . . . 27
4.8 Jazyk popisu váhy výstupů a jeho zpracování . . . 28
4.9 Převod RTLIL reprezentace na Genom (třída) . . . 30
4.10 Převod Genomu (třída) na RTLIL reprezentaci . . . 31
4.11 Automatické testování rozšíření . . . 31
4.12 Paralelizace simulace . . . 31

5 Implementace navrženého rozšíření 33
5.1 Rozložení kódu rozšíření . . . 33
5.2 Genom . . . 33
5.3 Reprezentace . . . 34
5.3.1 AIG . . . 34
5.3.2 MIG . . . 35
5.3.3 Gates . . . 36
5.4 Generace . . . 37
5.5 Zpracování jazyka pro popis důležitosti výstupů . . . 38
5.6 Struktura cgploss . . . 38
5.7 Použití . . . 40
5.7.1 Kompilace rozšíření . . . 40
5.7.2 Spuštění a zavedení do nástroje Yosys . . . 41
5.7.3 Ukázka použití . . . 42

6 Experimentování a testování 44
6.1 Testovaný obvod: čtyřbitová sčítačka . . . 44
6.1.1 Optimalizace přesné sčítačky . . . 45
6.1.2 Optimalizace pro maximální povolenou průměrnou chybu . . . 46
6.1.3 Optimalizace pro maximální povolenou chybu jedné kombinace . . . 48
6.1.4 Parametr počtu rodičů . . . 50
6.1.5 Parametr velikosti selekce . . . 50
6.1.6 Parametr l-back . . . 51
6.2 Testovaný obvod: osmibitová sčítačka . . . 52
6.2.1 Optimalizace přesné sčítačky . . . 52
6.2.2 Optimalizace pro povolenou výstupní chybu . . . 53
6.2.3 Parametr počtu rodičů . . . 55
6.2.4 Parametr velikosti selekce . . . 55
6.2.5 Parametr l-back . . . 56
6.3 Testovaný obvod: osmibitová násobička . . . 57
6.3.1 Optimalizace přesné násobičky . . . 57
6.3.2 Optimalizace pro povolenou výstupní chybu . . . 58
6.3.3 Parametr velikosti selekce . . . 59
6.3.4 Parametr l-back . . . 60
6.4 Závěr z testování . . . 61

7 Závěr 63

Literatura 64

A Obsah přiloženého paměťového média 67

Kapitola 1

Úvod

V dnešní době se můžeme setkat s kombinačními obvody téměř v každém digitálním systému. Tyto obvody v těchto zařízeních obvykle řeší různé aritmetické a logické operace. Mezi takovéto operace patří například matematické sčítání, dělení, násobení či odečítání. Kombinační obvody jsou také součástí dnešních procesorů a bez nich by nebylo možné tyto procesory ani sestavit. Tyto kombinační obvody zde nalezneme například v ALU (aritmeticko logická jednotka) nebo v FPU (matematický koprocesor). Mimo běžný procesor se kombinační obvody také nacházejí ve spoustě specifických zařízeních, kde mohou sloužit jako hardwarové akcelerátory k procesoru. Ukázkou takového obvodu je například TPU (Tensor Processing Unit), který je vyvíjen společností Google a slouží k akceleraci výpočtů umělých neuronových sítí [14]. Cílem vytváření těchto zařízení je urychlit výpočet tím, že jeho realizaci provedeme pomocí hardwarových komponent, ze kterých sestavíme kombinační obvod, který bude schopen provádět hlavní operace mnohem rychleji než jejich softwarové řešení.

Se zvyšující se popularitou IoT a nositelné elektroniky, roste potřeba takovéto obvody optimalizovat, jak z hlediska objemnosti obvodů, tak i z hlediska jejich energetické náročnoti. Tyto potřeby se v poslední době ukazují jako velmi důležité, protože mohou ovlivnit životnost a délku výdrže baterií těchto zařízení. Dalším problémem jsou náročné výpočty umělých neuronových sítí a dalších algoritmů ze světa umělé inteligence, které se dostávají mnohem častěji do těchto zařízení. Výpočet neuronové sítě se zjednodušeně skládá z velkého množství operací násobení a operací sčítání. Každý neuron v takové síti má svoje vstupy a u každého vstupu má jeho váhu. Pokud tedy budeme mít nejednoduší variantu neuronu, jehož vnitřní funkce bude pouze suma součinů, provede tento neuron vynásobení všech vstupů jejich váhami a následně výsledná čísla sečte. Pokud tedy budeme mít neuronovou síť o sto neuronech a tyto neurony budou poskládány do sloupců po pěti neuronech, bude nutné na jednom sloupci provést $5 * 5 = 25$ operací násobení a $4 * 5 = 20$ operací sčítání na jednom sloupci za předpokladu, že se jedná o běžnou neuronovou síť typu FCNN, která má pět vstupů. Pro celou neuronovou síť tedy budeme potřebovat $20 * 25 = 500$ operací násobení a $20 * 20 = 400$ operací sčítání [20]. Z tohoto počtu operací se dá usoudit, že výpočet nebude úplně snadný a zabere nějaký čas. Takový výpočet na základních vestavěných nebude možné úplně snadno provádět v reálném čase. Pro lepší představu neuronová síť AlexNet sloužící pro rozpoznání obrázků obsahuje 9192 neuronů, což implikuje mnohem větší počet operací násobení a sčítání, přičemž zmíněných 9192 neuronů je pouze její malou částí, protože v druhou větší částí se objevuje operace konvoluce, která je mnohem výpočetně náročnější [1, 15]. Z tohoto důvodu dává smysl takovéto algoritmy akcelerovat na nějakém specializovaném obvodu. Tímto obvodem může být například GPU nebo zmí-

něné TPU. Problémem těchto obvodů je často jejich velikost nebo energetická náročnost. Z tohoto důvodu se tato práce zabývá optimalizačním nástrojem, který by byl schopen energeticky optimalizovat obvody na úkor jejich přesnosti, která ve spoustě aplikací, jako jsou například neuronové sítě, tolik důležitá není. Tomuto jevu říkáme aproximace - cíleně zavádíme chybu do obvodů (na úrovni kombinační funkce) tak, abychom dosáhly snížení energetické náročnosti výpočtů [19].

Kapitola 2 představuje metodu z oblasti genetických algoritmů, která bude v práci použita pro optimalizaci obvodů. V kapitole 3 se představuje nástroj z oblasti syntézy obvodů. Tento nástroj bude za pomoci rozšíření tato práce rozšiřovat. Kapitola 4 popisuje způsob návrhu tohoto rozšíření, které je pojmenováno jako cgploss. V kapitole 5 je popisováno implementované rozšíření cgploss a jeho implementační detaily. Předposlední kapitolou je kapitola 6, která popisuje a prezentuje výsledky testů výkonnosti rozšíření. Poslední kapitola 7 popisuje možná budoucí rozšíření této práce.

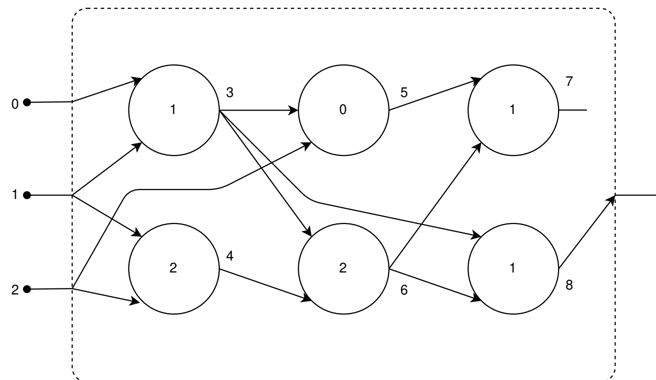
Kapitola 2

Kartézské genetické programování

Kartézské genetické programování (CGP) je variantou genetického programování, u kterého jsou kandidátní řešení reprezentována pomocí obecných orientovaných grafů [17]. Pokud je cílem navrhovat a optimalizovat, pouze kombinační obvody, jsou obecné orientované grafy zaměněny za acyklické orientované grafy [22, 21].

2.1 Genotyp

Optimalizovaná nebo generovaná struktura (matematická rovnice, obvod, ASM kód, ...) je modelována jako pole programovatelných elementů (uzlů grafu) o velikosti n_c (počet sloupců) na n_r (počet řádků). Počet primárních vstupů n_i i počet primárních výstupů n_o je pevně určen na začátku evoluce [22, 21]. Každý z uzlů reprezentuje právě jednu funkci mající až n_n argumentů, která je vybrána z množiny dostupných funkcí Γ [22, 21]. Vstupy uzlu nacházející se na i-tém sloupci mohou být připojeny buď na primární vstup obvodu nebo na libovolný uzel (pro kombinační obvody platí, že lze připojit pouze sloupce s pozicí nižší než i). Míra propojitelnosti lze ovlivnit parametrem L (L-BACK), který reprezentuje maximální vzdálenost sloupců dostupných k propojení [22, 21]. Pro $L = 1$ je propojitelnost minimální, protože je možné použít pouze okolní sloupce, naopak pro $L = n_c$ je propojitelnost maximální možná, protože bude možné připojit libovolný výstup z jakéhokoliv sloupce. Obrázek 2.1 ukazuje příklad orientovaného grafu CGP.



Obrázek 2.1: Příklad kandidátního obvodu v CGP s parametry $n_r = 2$, $n_c = 3$, $n_i = 3$, $n_o = 1$, $n_n = 2$, $L = 2$, $\Gamma = \{AND(0), OR(1), XOR(2)\}$. Uzly 7 a 5 nejsou součástí fenotypu. Reprezentovaná funkce ((0 or 1) xor (1 xor 2)) or (0 or 1).

Chromozom popisující zapojení obvodu v CGP je polem obsahující Λ_{cgp} částí (genů), které jsou reprezentovány celými čísly [22, 21].

$$\Lambda_{\mathrm{cgp}} = n_{\mathrm{r}} n_{\mathrm{c}} (n_{\mathrm{n}} + 1) + n_0$$

Tato čísla reprezentují funkci a vstupy elementů (uzlů grafu). Pro přehlednost lze tato čísla sjednotit do celků reprezentujících jeden element (uzel grafu). Vytvořené celky se poté budou skládat z čísla označující typ elementu a množiny čísel označujících vstupy elementu. Ukázka chromozomu k obrázku 2.1 s vytvořenými celky po elementech, kde prvních 6 celků je kódem vnitřních elementů a poslední celek je připojení výstupů, je níže.

$$[[0, 1, 1], [1, 2, 2], [3, 2, 0], [3, 4, 2], [5, 6, 1], [3, 6, 1], [8]]$$

Princip kódování je následující: každému primárnímu vstupu je přiřazen index $0, ..., n_{\mathrm{i}} - 1$. Každý uzel grafu je poté kódován jako série $n_{\mathrm{n}} + 1$ celočíselných hodnot. Tyto hodnoty reprezentují typ uzlu a jeho vstupy. Je předpokládáno, že prvních n_{n} hodnot reprezentuje vstupy a $n_{\mathrm{n}} + 1$ hodnota reprezentuje typ uzlu grafu [22], nicméně toto pořadí není nutné pro správnou funkci CGP a není jej tedy nutné dodržet. Po přiřazení indexů vnitřním sloupcům dochází k přiřazení indexů i primárním výstupům. Díky tomuto kódování vzniká redundance a to na několika úrovních [22].

- Některé uzly nacházející se v genotypu se ve fenotypu nemusejí využít.
- Některé vstupy uzlu se v závislosti na typu uzlu nemusejí použít.
- Některé primární vstupy nemusejí být ve fenotypu využity.

2.2 Operátory

Operátory v genetickém programování slouží na úpravu jedinců v populaci. Jediné operátory, které standardní CGP používá, jsou mutace a reprodukce [22].

2.2.1 Mutace

Mutace slouží k náhodné změně v chromozomu jedince [11]. Změna probíhá tak, že se nejdříve vygeneruje náhodná pozice v chromozomu (náhodně se vybere gen) a následně se náhodně vygeneruje jeho nová náhodná hodnota [22]. Tato nová hodnota musí být validní pro CGP reprezentaci, proto je nutné, aby náhodné číslo bylo ve validním rozsahu pro gen. Parametrem této operace je počet mutací provedených nad chromozomem, který může být opět náhodným číslem v určitém rozsahu.

2.2.2 Reprodukce

Reprodukce slouží ke klonování jedinců populace bez změny chromozomu. Vytvořený jedinec touto operací je přesnou kopijí zdrojového jedince [11]. U standardního CGP se používá tato operace pro vytvoření dostatečně velké populace, nad kterou bude následně aplikována operace mutace.

2.2.3 Křížení

Křížení je operace, která se ve standardním CGP nepoužívá [22]. Křížení je operací, která vytváří nového jedince rekombinací genotypů dvou či více existujících jedinců [9]. Vytvořený jedinec tedy nese genetickou informaci všech zdrojových jedinců. Tato operace nejdříve určí náhodnou pozici crossover pointů v chromozomu, počet crossover pointů je dán parametem operace n_c. Poté se pokračuje vytvořením nového jedice, do kterého je překopírována část chromozomu jedince, který byl vybrán pro vytvoření dané kombinace chromozomu. Toto kopírování probíhá do prvního crossover pointu, kde je opět vybrán další jedinec odpovídající vytvářené variantě kombinace chromozomu, obdobně se pokračuje až do konce chromozomu [26]. Tento postup se provede pro všechny kombinace chromozomu, přičemž každá kombinace vytvoří nového potomka. Pokud tedy máme dva crossover pointy a dva jedince A a B, pak musíme vytvořit 8 potomků (AAA, AAB, ABA, ABB, BAA, BAB, BBA, BBB). Pokud nechceme křížením vytvořit replikace rodičů, vytváříme pak pouze n_k potomků.

$$n_k = pocet_rodicu^{n_c+1} - pocet_rodicu$$

Nicméně při tomto postupu dochází i k použití méně crossover poitů než bylo operaci zadáno. Pokud tedy opravdu chceme pouze dvou crossover pointové potomky, je nutné ostatní potomky oddělat, poté zbyde pouze ABA a BAB. Je také možné implementovat jednoduší variantu, která pouze generuje jednu variantu potamka [9]. V tabulce 2.1 je ukázka křížení dvou jediců pomocí dvou crossover pointů na pozicích genu dva a genu čtyři, bez zahazování variant s jedním crossover pointem nebo žádným crossover pointem. V tabulce jsou vypsáni všichni vytvoření potomci se svými chromozomy.

Křížení je operací, která se ve standardním CGP nepoužívá, jelikož nebyla dokázána existence takového efektovního operátoru [10].

2.3 Algoritmus

Algoritmus prohledávání pracuje s populací o velikosti $\lambda+1$ jedinců. Nová populace se tvoří výběrem jedince ze staré populace s nejlepší hodnotou funkce fitness. Následně se provede jeho reprodukce do nové populace tak, aby měla $\lambda + 1$ jedinců. Následně se z populace vybere λ jedinců, nad kterými bude aplikována operace mutace. Pokud existuje více jedinců s nejlepší hodnotou funkce fitness, vybere se ten, který nebyl vybrán minulou generaci. Tím se zajištuje genetická diverzita [22]. Evoluce končí při nalezení jedince s akceptovatelnou hodnotou funkce fitness nebo po dovršení počtu generací, které byly zadány.

2.3.1 Kroky

1. Vygenerování $\lambda + 1$ inicializačních jedinců.
2. Vypočtení hodnoty funkce fitness pro všechny jedince.
3. Výběr jedinců s nejlepší hodnotou funkce fitness.
4. Výběr jedince pro následující populaci a jeho kopie do nové populace.
5. Replikace vybraného jedince s λ potomky.

Tabulka 2.1: Křížení dvou jedinců s crossover pointy 2 a 4

	gen 1.	gen 2.	gen 3.	gen 4.
Jedinec A.	5	7	1	4
Jedinec B.	9	9	1	0
Potomek (AAA)	5	7	1	4
Potomek (AAB)	5	7	1	0
Potomek (ABA)	5	9	1	4
Potomek (ABB)	5	9	1	0
Potomek (BAA)	9	7	1	4
Potomek (BAB)	9	7	1	0
Potomek (BBA)	9	9	1	4
Potomek (BBB)	9	9	1	0

6. Náhodná mutace λ potomků.

7. Kontrola ukončující podmínky, pokud není splněna vrácíme se ke kroku 2.

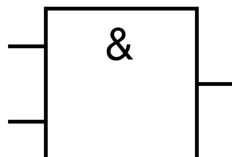
2.4 Reprezentace kombinačního obvodu pro CGP

Protože CGP pracuje pouze s orientovanými grafy, je nutné kombinační obvod převést na orientovaný graf. Kombinační obvody lze popisovat na různých urovních. Mezi nejznámější způsoby popisu patří popis RTL pouze s kombinačními bloky a bez cyklů, dále úroveň hradel (více v části 2.4.1) a následně úroveň jednotlivých elektrotechnických komponent, u kombinačních obvodů by se jednalo především o tranzistory. Vzhledem k vlastnostem CGP především kvůli problému škárovatelnosti neexistuje reprezentace, která by byla vhodná pro všechny existující obvody, proto je nutné použít více jak jednu reprezentaci. CGP na úrovni hradel je vhodné pouze pro jednoduché obvody. Pokud použijeme velmi objemné obvody, stává se CGP neefektivním. Důvodem této neefektivity je především délka chromozomu. S rostoucí velikostí chromozomu totiž roste i stavový prostor, který je nutné prohledat [22]. Pokud použijeme reprezentaci na úrovni RTL (pouze s kombinačními bloky a bez cyklů) budeme schopni rychle optimalizovat velké obvody, ale u malých obvodů opět nebudeme příliš efektivní. Důvodem je, že obvod je v reprezentaci RTL (pouze s kombinačními bloky a bez cyklů) již téměř optimální a nelze více zoptimalizovat na této úrovni. Reprezentace na úrovni elektrotechnických komponent bude efektivní na opravdu malých obvodech, další specifitou této reprezentace je její nízkoúrovňovost, která umožňuje použít pouze minimum vysokoúrovňových konvenčních součástek ve výsledném obvodu. Proto je tato reprezentace vhodná především pro výrobce integrovaných obvodů. Další reprezentace obvodu, které jsou spíše známy z odvětví optimalizace obvodů jsou And-Inverter Graph (AIG) (více v části 2.4.2), Majority-Inverter Graph (MIG) (více v části 2.4.3) a XOR-Majority Graphs (XMG)

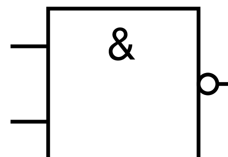
(více v části 2.4.4). Tyto reprezentace v sobě sjednocují více logických hradel do jednoho bloku, ale ne tolik jako může být viděno u RTL (pouze s kombinačními bloky a bez cyklů) reprezentace. Jsou tedy někde mezi úrovní hradel a RTL (pouze s kombinačními bloky a bez cyklů) reprezentací, nicméně narozdíl od RTL (pouze s kombinačními bloky a bez cyklů) reprezentace používají jen jednobitové cesty (hrany grafu) [22].

2.4.1 Úroveň hradel

Na úrovni hradel se logický obvod skládá z bloků nazývaných logická hradla. Logická hradla jsou elektrotechnické komponenty, jejichž princip je postaven na Booleově algebře [25, 13]. Každé logické hradlo přiřazuje na výstup logickou hodnotu, kterou má danou svojí logickou funkcí aplikovanou na svoje vstupy. Tato reprezentace je jednoduše převeditelná na orientovaný graf. Narozdíl od AIG, MIG, XMG a XAIG tento graf ale nebude mít předem určený počet vstupních hran do jednoho uzlu. Důvodem je, že existují hradla jednovstupá, dvouvstupá, trojvstupá i čtyřvstupá. Vzhledem k velkému možství netradičních logických hradel mohou existovat i vícevstupá hradla než čtyřvstupá. Nejčastěji je možné se setkat s hradly jednovstupými a dvovstupými. Mezi základní zástupce jednovstupých hradel patří především hradlo NOT, které přiřadí na svůj výstup negaci svého vstupu. Zástupci dvouvstupých hradel jsou především AND, OR, NAND, NOR, XOR a XNOR, jejichž bližší specifikace v podobě pravdivostních tabulek a schémat jsou níže [25]. Výhodou této reprezentace je její nenáročná simulace jednohu uzlu, na kterou postačí pouze jedno logické hradlo a není problém ji tedy masivně simulavat, nicméně jedná se o velmi nízkou úroveň abstrakce, která implikuje velké množství genů v chromozomu, které znamená velký stavový prostor pro prohledávání. Vzhledem k velkému množství logických hradel je nutné vybrat jen určitou jejich podmnožinu, kterou použijeme pro CGP. V některých případech je vhodné použít několik hradel netradičního typu jako je například MAJ, AOI (AND-OR-invert) či OAI (OR-AND-invert) [22].



Obrázek 2.2: Hradlo AND v normě IEC



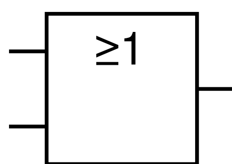
Obrázek 2.3: Hradlo NAND v normě IEC

vstup 1	**vstup 2**	**výstup**
0	0	0
0	1	0
1	0	0
1	1	1

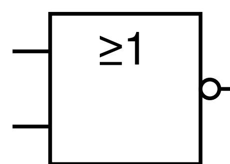
Tabulka 2.2: Pravdivostní tabulka hradla AND

vstup 1	**vstup 2**	**výstup**
0	0	1
0	1	1
1	0	1
1	1	0

Tabulka 2.3: Pravdivostní tabulka hradla NAND



Obrázek 2.4: Hradlo OR v normě IEC



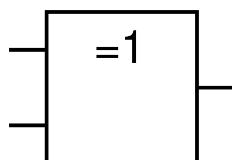
Obrázek 2.5: Hradlo NOR v normě IEC

vstup 1	**vstup 2**	**výstup**
0	0	0
0	1	1
1	0	1
1	1	1

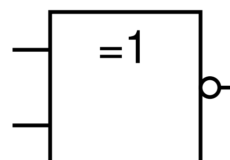
Tabulka 2.4: Pravdivostní tabulka hradla OR

vstup 1	**vstup 2**	**výstup**
0	0	1
0	1	0
1	0	0
1	1	0

Tabulka 2.5: Pravdivostní tabulka hradla NOR



Obrázek 2.6: Hradlo XOR v normě IEC



Obrázek 2.7: Hradlo XNOR v normě IEC

vstup 1	**vstup 2**	**výstup**
0	0	0
0	1	1
1	0	1
1	1	0

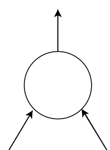
Tabulka 2.6: Pravdivostní tabulka hradla XOR

vstup 1	**vstup 2**	**výstup**
0	0	1
0	1	0
1	0	0
1	1	1

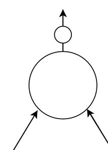
Tabulka 2.7: Pravdivostní tabulka hradla XNOR

2.4.2 And-Inverter Graph

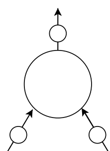
And-Inverter Graph je acyklický orientovaný graf, který reprezentuje strukturu logických funkcí nebo logických sítí [3]. Tento popis je funkčně úplný, což znamená, že pomocí něj lze popsat jakoukoliv Booleovskou funkci. Každý uzel AIG má dvě vstupní hrany a jednu výstupní. Každý uzel AIG má svůj typ, tento typ reprezentuje existenci negací na jeho vstupních a výstupních hranách. Těchto typů proto existuje osm. Interpretace jednoho uzlu je vnitřní AND (Konjunkce), který pracuje se vstupními hodnotami, které jsou negované podle typu uzlu [3]. Pomocí jednoho uzlu AIG lze zapsat většina základních hradel[1] s výjimkou hradel XOR a XNOR. Proto je nutné XOR sestavit například jako $\neg((A \wedge B) \wedge \neg(A \wedge B))$. XOR má v AIG více tříuzlových reprezentací [12]. Hlavní výhodou této reprezentace je relativně malá paměťová náročnost při simulaci obvodu a celkem rychlá simulace jednoho uzlu narozdíl od RTL (pouze s kombinačními bloky a bez cyklů). Nicméně narozdíl od hradlové úrovně je na vyšší úrovni abstrakce [3].



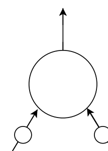
Obrázek 2.8: AND v AIG



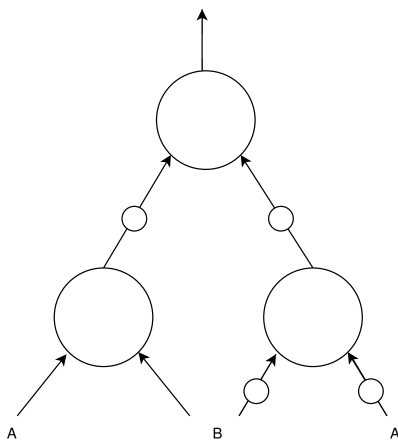
Obrázek 2.9: NAND v AIG



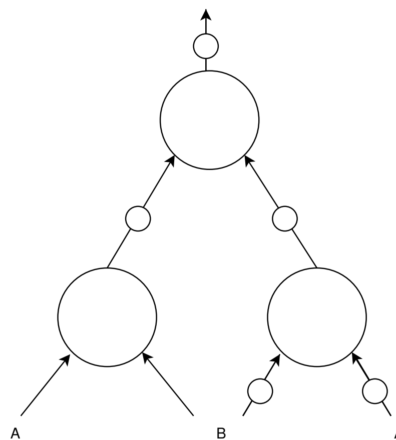
Obrázek 2.10: OR v AIG



Obrázek 2.11: NOR v AIG



Obrázek 2.12: XOR v AIG

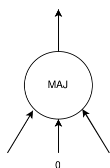


Obrázek 2.13: XNOR v AIG

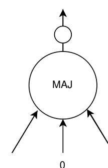
[1]NOT hradlo je reprezentováno jako NAND hradlo s identickými vstupy

2.4.3 Majority-Inverter Graph

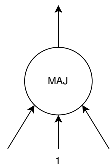
Majority-Inverter Graph je acyklický orientovaný graf, který reprezentuje strukturu logických funkcí nebo logických sítí [2]. Tento popis je funkčně úplný, což znamená, že pomocí něj lze popsat jakoukoliv Booleovskou funkci. Každý uzel MIG má tři vstupní hrany a jednu výstupní. Každý uzel MIG má svůj typ, tento typ reprezentuje existenci negací na jeho vstupních a výstupních hranách. Těchto typů proto existuje 16. Interpretace jednoho uzlu je vnitřní Majority gate (MAJ), které pracuje se vstupními hodnotami, které jsou negovány podle typu uzlu. Hradlo MAJ uvnitř uzlu realizuje funkci $(A \wedge B) \wedge (B \wedge C) \wedge (A \wedge C)$. MIG je reprezentací na vyšší úrovni než AIG. Vnitřní struktura jednohu uzlu může být realizována třemi hradly AND a dvěmi hradly OR. Proto je jeho simulace náročnější než simulace AIG, jehož jeden uzel lze reprezentovat jako jedno hradlo AND, nicméně díky tomu je MIG na vyšší úrovni abstrakce a může se proto více hodit na složitější obvody než AIG. Pomocí jehoho uzlu MIG lze zapsat většina základních hradel[2] s výjimkou hradl XOR a XNOR. Proto je nutné XOR sestavit například jako $\neg((A \wedge B) \wedge \neg(A \wedge B))$. XOR má v MIG více tříuzlových reprezentací [16].



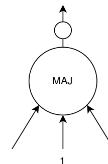
Obrázek 2.14: AND v MIG



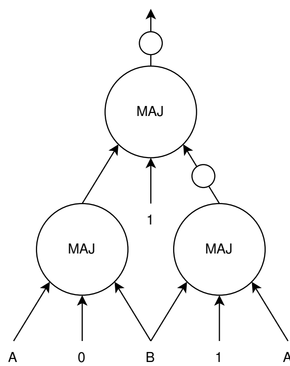
Obrázek 2.15: NAND v MIG



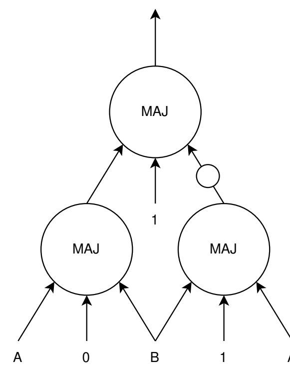
Obrázek 2.16: OR v MIG



Obrázek 2.17: NOR v MIG



Obrázek 2.18: XOR v MIG



Obrázek 2.19: XNOR v MIG

[2]NOT hradlo je reprezentováno jako NAND hradlo s identickými vstupy

2.4.4 XOR-Majority Graph

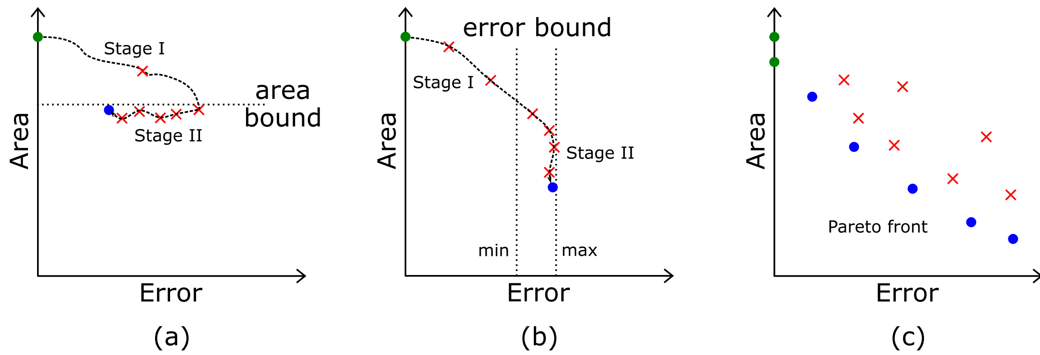
XOR-Majority Graph je acyklický orientovaný graf, který reprezentuje strukturu logických funkcí nebo logických sítí. Tento popis je funkčně úplný, což znamená, že pomocí něj lze popsat jakoukoliv Booleovskou funkci. Každý uzel XMG má tři vstupní hrany a jednu výstupní. Každý uzel XMG má svůj typ, tento typ narozdíl od MIG typu přidává hradlo XOR, díky němuž je zjednodušena reprezentace XOR a XNOR hradel v obvodech. Interpretace uzlu je téměř stejná jako u MIG, pouze je navíc přidán typ, který reprezentuje XOR, z toho důvodu pro jistý typ nebude uzel fungovat jako MAJ, nýbrž jako XOR [16].

2.4.5 XOR-And-Inverter Graph

XOR-And-Inverter Graph (XAIG) je acyklický orientovaný graf, který reprezentuje strukturu logických funkcí nebo logických sítí. Tento popis je funkčně úplný, což znamená, že pomocí něj lze popsat jakoukoliv Booleovskou funkci. Jendá se o reprezentaci AIG s přidaným hradem XOR mezi typy uzlů. Obdobným způsobem je možné do AIG přidat různé jiné bloky, jako je například multiplexor [12].

2.5 Využití CGP pro aproximaci obvodů

Při využití CGP pro aproximaci obvodů existují tři hlavní strategie [18].



Obrázek 2.20: Aproximační strategie: (a) zdrojově orientovaná, (b) chybově orientovaná (c) vícecílová. Obrázek přejat z [18]

2.5.1 Zdrojově orientovaná strategie

Tato strategie je rozdělena do dvou fází [18]. V první fázi je přesný počáteční obvod, který je redukován, dokud neobsahuje m_i komponent. Toto redukování může být implementováno jako náhodné odstranění hradel, nebo mohou být použité složitější metody. V okamžik, kdy obvod obsahuje m_i komponent nebo méně, přechází se do fáze dva.

Ve druhé fázi je CGP použito na optimalizaci výstupní chyby, zatím co počet komponent v obvodu bude menší roven m_i. Cílem je dosáhnout co nejmenší výstupní chyby.

Výhodou této metody je, že uživatel je přesně schopen určit počet komponent v obvodu (plochu i energetickou náročnost) pomocí proměnné m_i [18].

2.5.2 Chybově orientovaná strategie

Tato strategie má zadané rozpětí výstupní chyby [18]. Řekněme, že tedy máme e_{min} a e_{max}, které určují rozsah chyby. Protože je tato meteoda opět dvoufázová, provede se v první fázi redukce obvodu, dokud nebude výstupní chyba obvodu větší rovna e_{min}. Po dosažení tohoto cíle se přechází do fáze dva. Ve fázi druhé CGP optimalizuje obvod podle počtu komponent při držení chyby v rozmezí e_{min} a e_{max}. Toto držení chyby v rozmezí je často implementováno pomocí zahazování jedinců, kteří nesplní toto rozmezí.

Výhodou této metody je, že uživatel je schopen určit velikost výstupní chyby [18].

2.5.3 Vícecílová strategie

Vícecílové CGP (Multi-objective CGP) je strategií, která se zaměřuje na optimalizaci více parametrů v jednom CGP běhu [18]. Tato strategie tedy narozdíl od prvních dvou strategií, které používaly jednu fitness funkci, používá více fitness funkcí. U této strategie nás zajímají především aproximační obvody patřící do *Pareto* množiny, která obsahuje tzv. nedominantní řešení. Pokud máme dva obvody A a B, potom obvod A je lepší než obvod B pouze, pokud je lepší ve všechy optimalizovaných hlediskách. Pokud je obvod B ve všech hlediskách lepší než obvod A, pak je chápán jako lepší obvod než A. Cílem této strategie je ponechat obvody, u kterých se nedá říci, zda jsou lepší než jiné, do následující generace. Algoritmus hledání využívá upravenou variantu vícecílového genetického algoritmu, jako je Non-dominated Sorting Genetic Algorithm (NSGA-II) [18].

Kapitola 3

Nástroj YOSYS

Yosys je framework pro nejen Verilog RTL syntézu. Syntéza je proces, při kterém dochází k převodu různých RTL popisů (VHDL, Verilog, BLIF, . . .) do technologických reprezentací (LUT pro FPGA, hradla pro ASIC). V současné době podporuje Verilog-2005 a poskytuje základní sadu syntézních algoritmů pro různé aplikace. Yosys je svobodný software licencovaný pod licencí ISC. Yosys je nástroj, jehož syntéza je ovladatelná pomocí skriptů pro sytézu, které volají průchody (passes - algoritmy nástroje). Yosys také umožňuje přidání vlastního průchodu (pass) pomocí rozšíření psaném v jazyce C++ [5].

3.1 Ovládání

Nástroj Yosys je konzolovou aplikací, která se ovládá příkazy. Tyto příkazy je možné zapsat do souboru (skriptu ve formátu TCL) a nástroji Yosys tento skript předat (například přes STDIN) a tím automatizovat syntézu. Každý průchod (pass) je volán jeho jménem za nímž jsou následně předány jeho parametry. Základní průchody jsou `read_verilog` (přečtení vstupního souboru), `opt` (optimalizace obvodu), `techmap` (mapování na vnitřní cell knihovnu), `write_verilog` (zapíše obvod do Verilog souboru) a `synth` (generická syntéza) [5]. Ukázka syntézního skriptu pro ASIC je na obrázku 3.1. Další ukázky a průchody jsou k dizpozici na stránkách nástroje Yosys [5].

```
1  # přečte model obvodu z Verilog souboru
2  read_verilog mydesign.v
3
4  # provede highlevel optimalizaci
5  opt
6
7  # provede mapping na gate-level pomocí vnitřní knihovny
8  techmap
9
10
11 # vyčistí nepotřebné části obvodu
12 clean
13
14 # zapíše syntetizovaný Verilog
15 write_verilog synth.v
```

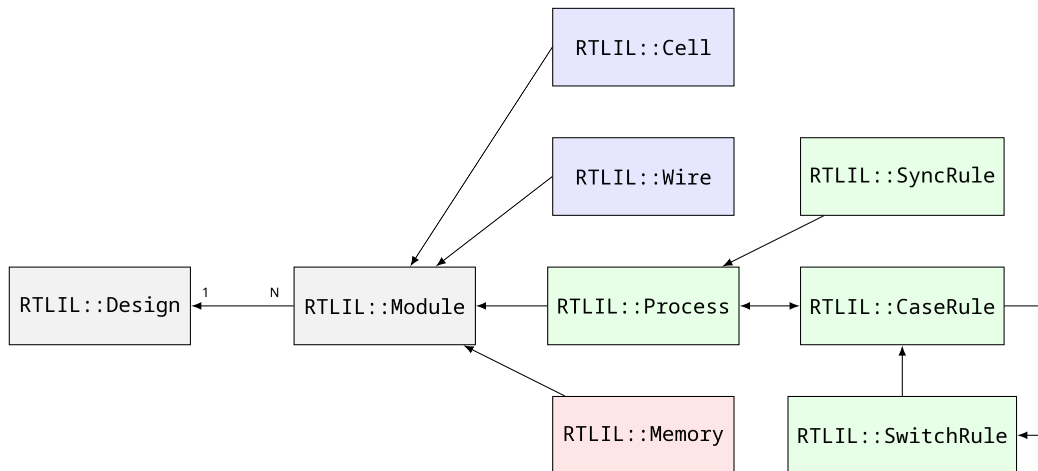
Obrázek 3.1: Ukázka syntézního skriptu Yosys

3.2 Vnitřní formáty a reprezentace obvodu

Yosys používá dva rozdílné vnitřní formáty. První formát používá na uložení abstraktního syntaktického stromu vstupního Verilog souboru. Tento formát, který je označován jako AST, je generován pomocí Verilog Frontendu. Tato data jsou používaná pouze subsystémem zvaným AST Frontend. Tento AST Frontend generuje hlavní vnitřní formát zvaný Register-Transfer-Level-Intermediate-Language, který je v kódu označován jako RTLIL. RTLIL reprezentace je používaná všemi průchody (passes) jako vstupní a výstupní struktura. Jednotlivé průchody si mohou při svém spuštění z RTLIL reprezentace vytvořit svou vlastní reprezentaci, kterou následně převedou zpět do RTLIL jako svůj výstup. RTLIL reprezentace je obecně netlist reprezentace s přidanými funkcemi. Drobnou nevýhodou této reprezentace je její složitost v podobě nepotřebných dat při gate-level úpravách [4].

3.2.1 Register-Transfer-Level-Intermediate-Language (RTLIL)

RTLIL je hlavní reprezentace, kterou nástroj Yosys využívá pro vnitřní reprezentaci obvodu. Reprezentace je založena na objektovém návrhu. V tomto návrhu existují dvě hlavní třídy a to `RTLIL::Design`, který reprezentuje celý obvod načtený z Verilog souboru. Objekt této třídy obsahuje instance třídy `RTLIL::Module`, která reprezentuje moduly ve Verilogu. Do objektů třídy `RTLIL::Module` jsou dále přidávány objekty tříd `RTLIL::Cell`, `RTLIL::Wire`, `RTLIL::Process` a `RTLIL::Memory`. Instance tříd `RTLIL::Process` a `RTLIL::Memory` budou po syntéze změněny na `RTLIL::Cell` a `RTLIL::Wire`, proto v této práci nebudou dále popisovány, nicméně jejich bližší specifikace je v dokumentaci Yosys na stránce nástroje Yosys [4].



Obrázek 3.2: Class Diagram RTLIL přejat z dokumentace Yosys [4]

3.2.2 RTLIL::Design a RTLIL::Module

`RTLIL::Design` je kontejnerem `RTLIL::Module` objektů. `RTLIL::Design` kromě `RTLIL::Module` objektů obsahuje také seznam vybraných modulů (Seznam objektů nad kterými má průchod pracovat). `RTLIL::Module` je strukturou, která popisuje jeden Verilog modul a obsahuje objekty tříd `RTLIL::Cell`, `RTLIL::Wire`, `RTLIL::Process` a `RTLIL::Memory` dále obsahuje podpůrné struktury, které jsou níže: [4].

- Jméno modulu,
- seznam atributů,
- seznam propojení mezi RTLIL::SigSpec,
- volitelný frontend callback pro řízení parametrizovaných variant modulu.

3.2.3 RTLIL::Cell

`RTLIL::Cell` je datová struktura, která reprezentuje funkční bloky z cell knihovny. Jednotlivé objekty `RTLIL::Cell` se propojují pomocí `RTLIL::Wire`. Spolu s `RTLIL::Wire` reprezentují netlist obvodu. Objekt `RTLIL::Cell` obsahuje následující atributy [4].

- Jméno cell,
- typ cell,
- seznam atributů,
- seznam parametrů (parametrizované varianty),
- porty cell a signály připojené na porty.

3.2.4 RTLIL::Wire

`RTLIL::Wire` je datová struktura, která reprezentuje propojení mezi funkčními bloky z cell knihovny. Spolu s `RTLIL::Cell` reprezentují netlist obvodu. Vícebitové cesty jsou reprezentovány pouze pomocí jednoho objektu `RTLIL::Wire` s nastavenou šířkou větší než 1. Objekt `RTLIL::Wire` obsahuje následující atributy [4].

- Jméno wire,
- seznam atributů,
- šířka (pro více bitové cesty větší než 1),
- seznam parametrů (parametrizované varianty),
- pokud je wire portem, tak obsahuje jeho číslo a směr (IN, OUT, INOUT).

3.2.5 RTLIL::SigSpec

`RTLIL::SigSpec` reprezentuje signál v obvodu. Slouží k přidání bližší specifikace k `RTLIL::Wire`, případně k jeho nahrazení konstantou. Objekty typu `RTLIL::Cell` používají tento typ pro specifikaci připojení na port (Na porty se přiřazují `RTLIL::SigSpec`/`RTLIL::SigBit`). Propojení více signálů dohromady se provádí pomocí `RTLIL::SigSig`, které je definováno jako pár dvou objektů `RTLIL::SigSpec`. `RTLIL::SigSpec` obsahuje vektor `RTLIL::SigBit`, které popisují jeden bit v `RTLIL::Wire`.[4].

3.2.6 Vnitřní cell knihovna

Vnitřní cell knihovna je rozdělena na dvě části. Jedna část je RTL, která pracuje s RTL bloky jako jsou sčítačky, násobičky, bitové posuvy a podobně. Druhou částí je gate level knihovna, která se používá po provedení systézy z RTL úrovně na úroveň hradel. Níže je tabulka dostupných hradel na hradlové úrovni [4].

<table>
<tr><th>C++ kód</th><th>Cell typ</th></tr>
<tr><td>y = a</td><td>$_BUF_</td></tr>
<tr><td>y = ~a</td><td>$_NOT_</td></tr>
<tr><td>y = a & b</td><td>$_AND_</td></tr>
<tr><td>y = ~(a & b)</td><td>$_NAND_</td></tr>
<tr><td>y = a & ~b</td><td>$_ANDNOT_</td></tr>
<tr><td>y = a | b</td><td>$_OR_</td></tr>
<tr><td>y = ~(a | b)</td><td>$_NOR_</td></tr>
<tr><td>y = a | ~b</td><td>$_ORNOT_</td></tr>
<tr><td>y = a ^ b</td><td>$_XOR_</td></tr>
<tr><td>y = ~(a ^ b)</td><td>$_XNOR_</td></tr>
<tr><td>y = ~((a & b) | c)</td><td>$_AOI3_</td></tr>
<tr><td>y = ~((a | b) & c)</td><td>$_OAI3_</td></tr>
<tr><td>y = ~((a & b) | (c & d))</td><td>$_AOI4_</td></tr>
<tr><td>y = ~((a | b) & (c | d))</td><td>$_OAI4_</td></tr>
<tr><td>y = s ? b : a</td><td>$_MUX_</td></tr>
<tr><td>y = ~(s ? b : a)</td><td>$_NMUX_</td></tr>
<tr><td>2bit mux</td><td>$_MUX4_</td></tr>
<tr><td>3bit mux</td><td>$_MUX8_</td></tr>
<tr><td>4bit mux</td><td>$_MUX16_</td></tr>
</table>

Tabulka 3.1: Yosys vnitřní cell knihovna - hradlová úroveň

3.3 Tvorba rozšíření pro YOSYS

Rozšíření pro nástroj Yosys se píše v jazyce C++ jako realizace abstraktní třídy `Pass` (rozšíření je realizací jednoho nebo více průchodů). Kompilace rozšíření probíhá pomocí nástroje `yosys-config`, který je součástí vývojářské sady k nástroji Yosys. Při kompilaci vznikne soubor sdílené knihovny (.so), který je možné při spouštení zavést do nástroje Yosys. Při použití makra `USING_YOSYS_NAMESPACE` bude v rozšíření možné pracovat se všemi datovými strukturami, které byly zmíněny výše. Pokud bude použito makro pro zavedení jmeného prostoru Yosys, je nutné kvůli oddělení kódů kód rozšíření psát mezi makra `PRIVATE_NAMESPACE_BEGIN` a `PRIVATE_NAMESPACE_END`.

Abstraktní třída `Pass`, kterou rozšířením implementujeme má dvě hlavní metody, které je nutné realizovat. Jednou z metod je metoda `execute`, která je volána při spuštění průchodu. Tento průchod je součástí psaného rozšíření. Druhou z metod je metoda `help`, která není metodou povinou pro implementaci a slouží k vyvolání reakce na zavolání nápovědy pro průchod. Protože se jedná o realizaci abstraktní třídy, je nutné při volání konstruktoru průchodu zavolat konstruktor třídy jí nadřazené, kterému předáme jméno průchodu, pod kterým bude průchod volatelný. Metoda `execute` má dva parametry, těmi jsou std::vector<std::string>

args a RTLIL::Design *design, kde args je seznamem parametrů průchodu a design je designem obvodu, nad kterým má průchod pracovat. Ukázka kódu `hello.cpp` Hello Word rozšíření.

```
1  #include "kernel/yosys.h"
2
3  USING_YOSYS_NAMESPACE
4  PRIVATE_NAMESPACE_BEGIN
5
6  struct HelloWorldPass : public Pass {
7      // Konstruktor rozšíření,
8      // který registruje rozšíření s jménem hello_world
9      HelloWorldPass() : Pass("hello_world") { }
10
11     // minimalní verze execute která vypíše Hello Word!
12     void execute(vector<string>, Design*) override {
13         log("Hello World!\n");
14     }
15 } HelloWorldPass;
16
17 PRIVATE_NAMESPACE_END
```

Obrázek 3.3: Ukázka implementace jednoduchého rozšíření Yosys

Ke kompilaci této ukázky dojde pomocí nástroje `yosys-config`, tak jako bylo zmíněno výše. Tento kompilátor zavede veškeré nutné závislosti nástroje Yosys a provede kompilaci.

```
1  yosys-config --exec --cxx --cxxflags --ldflags -o hello.so \\
2  -shared hello.cpp --ldlibs
3  # nebo
4  yosys-config --build hello.so hello.cpp
```

Obrázek 3.4: Kompilace rozšíření Yosys

Po dokončení kompilace je možné nástroj Yosys s rozšířením spustit za pamoci načtení sdílené knihovny, která toto rozšíření obsahuje. V případě nutnosti je také možné zkompilovat celý nástroj Yosys s rozšířením a poté není nutné rozšíření samostatně zavádět jako sdílenou knihovnu.

```
1  # spuštení nástroje Yosys s rozšířením hello.so
2  yosys -m hello.so
3
4  # spuštení průchodu hello_world
5  yosys> hello_world
```

Obrázek 3.5: Spuštění rozšíření Yosys

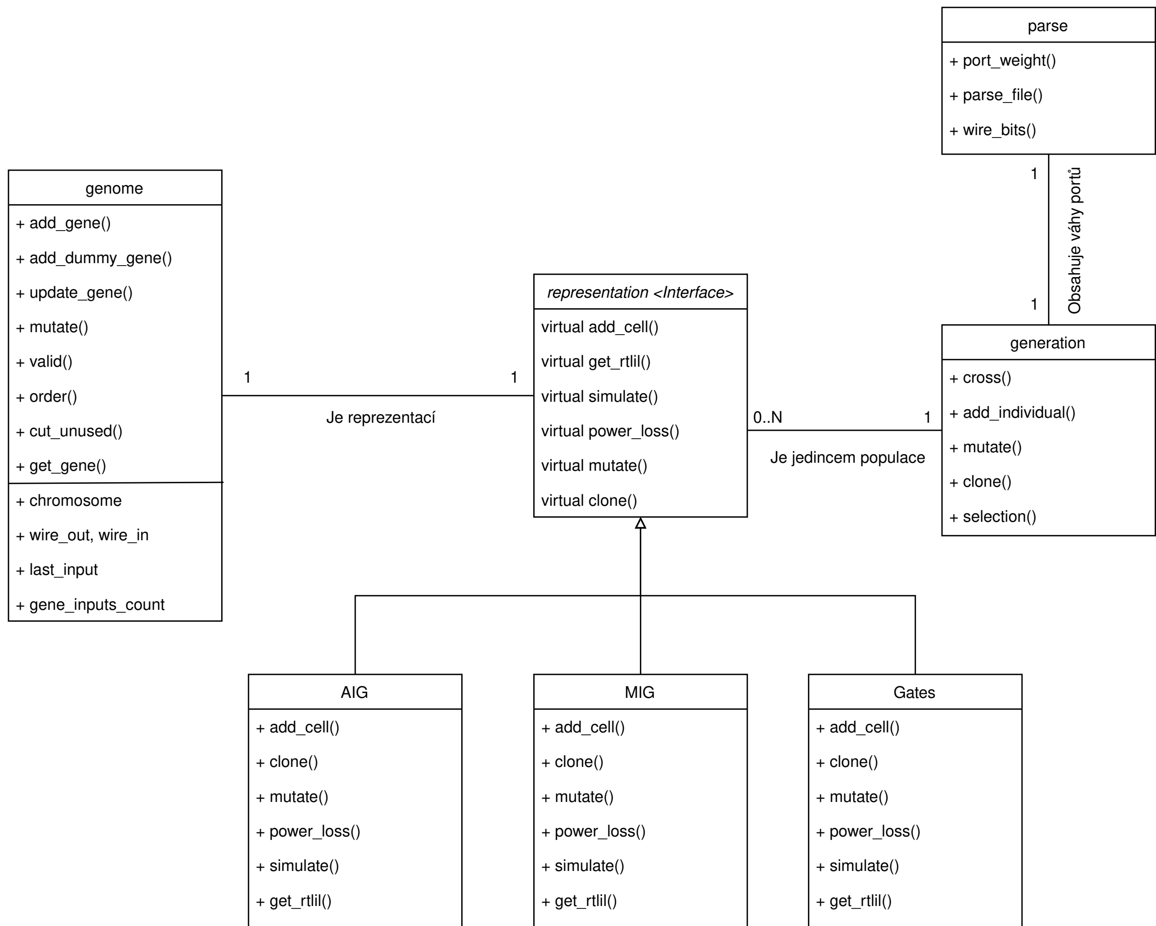
Kapitola 4

Návrh rozšíření aproximaci obvodů

Rozšíření cgploss bude po vytvoření sloužit k optimalizaci kombinačních logických obvodů za pomoci Kartézského genetického programování, které bude pracovat s obvody načtenými v nástroji Yosys. Bude se tedy jednat o průchod nástroje Yosys, který bude provádět optimalizaci obvodu. Cílem optimalizace bude vytvořit obvody, které budou mít nižší nároky na elektrickou energii za cenu vyšší chybovosti výstupu. Problematikou optimalizace se zabývá hned několik nástrojů a dokonce i několik průchodů samotného nástroje Yosys. Jendá se o nástroje jako je například abc, které slouží k logické syntéze a optimalizaci. Jako svou vnitřní reprezentaci používá AIG [23]. Dalším nástrojem je průchod nástroje Yosys jménem `opt`, který provede triviální optimalizace a čištění obvodu [5]. Nástrojů zabývajících se optimalizací, která by zohledňovala chybu výstupu na úkor nějakého jiného paremetru obvodu, již tolik není, u existujících nástrojů je hlavní problém jejich optížná propojitelnost s klasickými syntézními nástroji. Ukázkou jednoho takového nástroje je BLASYS [8]. Nicméně tento nástroj používá metodu Boolean Matrix Factorization, která je rozdílná od zamýšlené metody CGP.

4.1 Objektový návrh aplikace

Objektový návrh vycházející z částí v této kapitole zahrnuje základní potřeby cgploss rozšíření. Diagram tříd, který je na obrázku 4.1, ukazuje všechny základní třídy a jejich základní funkce a datové struktury. Třídy po implementaci budou obsahovat více funkcí a struktur, které budou podpůrného charakteru pro vypsané funkce v digramu.



Obrázek 4.1: Diagram tříd rozšíření cgploss

4.2 Parametry

Rozšíření cgploss bude obsahovat několik parametrů pro změnu vlastností CGP, aby bylo možné dosáhnout optimálních výsledků na různých obvodech.

```
-wire-test                  pouze načtení a uložení obvodu bez CGP [DEBUG]
-save_individuals=file      uložit chromozomy jedinců do souboru [DEBUG]
-ports_weights=file         soubor vah portů
-selection_size=size        velikost selekce (počet jedinců, kteří přežijí)
-generation_size=size       velikost jedné generace
-max_one_error=0..inf       maximální chyba pro jednu generaci
-generations=count          počet generací
-mutations_count=count      střed normálního rozdělení počtu mutací
-mutations_count_sigma=num  směrodatná odchylka normálního rozdělení počtu mutací
-parents=1..2               počet rodičů (dva rodiče - Experimentální rozšíření CGP)
-power_accuracy_ratio=0..1  poměr energetické náročnosti a chybovosti
-max_abs_error=num          maximální průměrná chyba
-cross_parts=2..inf         počet crossover pointů při křížení (dva rodiče)
-l-back                     maximální povolený rozsah mutace vstupu genu
-status                     průběžně vypisuje skóre jedinců
-profile                    vypíše jedince ve formátu {energie}-{chyba}-{skóre};
-max_duration=num           maximální doba optimalizace v minutách

-representation={aig, gates, mig}  reprezentace obvodu
```

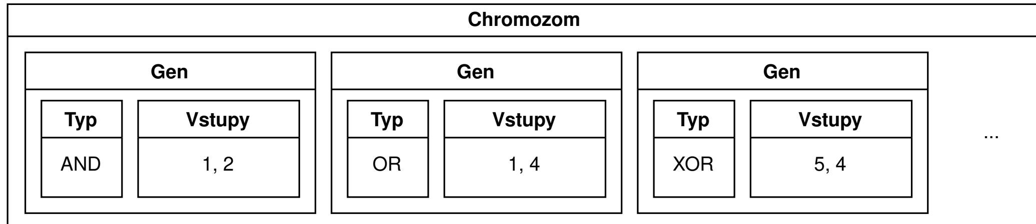
4.3 Výběr reprezentací obvodů

Vzhledem k tomu, že není možné použít všechny reprezentace obvodu pro CGP, je nutné vybrat jen jejich malou podmnožinu, kterou bude možné vytvořit v rámci této práce, a proto bude u vytvoření těchto reprezentací nutné sledovat především jejich snadnou rozšířitelnost. Pro dosažení tohoto cíle bude nutné reprezentace implementovat jako realizaci abstraktní třídy, tím bude zajištěno snadné přidávání dalších reprezentací v budoucnu. Vzhledem k tomu, že má reprezentace RTL (pouze s kombinačními bloky a bez cyklů) velkou množinu typů bloků, nebude implementována v cgploss, nicméně vzhledem k návrhu rozšíření nebude problém ji později přidat. Reprezentace na úrovni hradel bude jednou z implementovaných, protože se nachází na nejnižší úrovni abstrakce, kterou nástroj Yosys podporuje. Jako abstrakce na nejvyšší úrovni bude implementována reprezentace MIG, která je vzhledem k její úrovni abstrakce částečnou náhradou za RTL (pouze s kombinačními bloky a bez cyklů) reprezentaci. Jako třetí a zároveň poslední byla vybrána reprezentace AIG, která je vzhledem k její úrovni abstrakce mezi MIG a hradlovou úrovní. Varianty XAIG a XMG nebyly zvoleny kvůli jejich větší množině typů, která implikuje složitější implementaci a testování, nicméně je možné tyto varianty přidat později.

4.4 Návrh Genomu

Genom bude datová struktura, která bude na nejnižší úrovni abstrakce vzhledem ke genotypu. Genom bude tedy třídou jejichž instance budou nositelem genů, potažmo genetické informace jedince. Tato třída bude také obsahovat funkce na čtení a úpravu genů. Tato třída

tedy bude mít metody na klonování sebe sama, přidání prázdného genu, seřazení chromozomu, spočítání aktivních genů, získávání genů, odstranění nepoužívaných genů, zjištění validity sama sebe a zapsání genů. Gen bude reprezentován jako struktura mající typ a seznam vstupů, který má pevně danou velikost. Funkce pracující s geny budou pracovat s touto strukturou (jako gen zde nebude označeno jedno číslo v chromozomu ale skupina čísel, která realizuje hradlo). Chromozom poté bude realizován jako pole genů. Ukázka chromozomu je na obrázku 4.2.



Obrázek 4.2: Ukázka datové struktury chromozom s ukázkovými daty

Chromozom bude jednou ze struktur, která bude součástí Genomu (datová struktura). Protože gen bude popisovat pouze vnitřní hradla (vstupy ani výstupy nebudou součástí chromozomu), budou vstupy i výstupy zapsány v pomocné datové struktuře, která ponese identifikátor vstupu nebo výstupu a také index portu, který je použit v RTLIL reprezentaci. Funkce a datové struktury obsažené v genomu tedy budou:

- `wire_out` reprezentující výstupy,
- `wire_in` reprezentující vstupy,
- `chromosome` pole genů,
- `get_gene()` získání genu na pozici,
- `cut_unused()` odstranění nepoužitých genů,
- `used_cost()` zjištění počtu použitých genů se započítáním náročnosti genu pomocí předané funkce,
- `valid()` kontrola validity genu pro CGP,
- `mutate()` provedení mutace chromozomu,
- `update_gene()` aktualizace/změna určitého genu na určené pozici,
- `add_dummy_gene()` přidání nicnedělajícího genu,
- `clone()` vytvoření kopie sebe sama,
- `order()` seřazení chromozomu pro správnou funkčnost CGP.

4.5 Návrh reprezentace obvodu

Vzhledem k výběru zvolených reprezentacích, který je popsán v části 4.3, bude nutné vytvořit abstraktní třídu, kterou budou jednotlivé reprezentace implementovat. V části 4.3 došlo k výběru tří reprezentací, které budou implementovány. Tyto reprezentace jsou AIG, MIG a úroveň hradel, která bude dále označována jako reprezentace GATE. Abstraktní třída, která bude reprezentovat obecnou reprezentaci bude na vyšší úrovni abstrakce než třída Genom. Tato třída bude mít za úkol zapouzdřit třídu Genom do sebe a zajistit vyšší úroveň abstrakce, která umožní pracovat na úrovni hradel nikoli genů. Tato třída bude použita při generováni výstupního RTLIL i při načítání vstupního RTLIL do Genomu. Pro tyto účely bude mít třída dvě metody, které budou volány částmi kódu zajištujících RTLIL/GENOM převod. Jednou z metod sloužících pro tyto účely bude **add_cell**, která přidá (přepíše) jedno hradlo (buňku) z RTLIL reprezentace do Genomu. Druhou metodou bude poté `get_rtlil`, která naopak přidá (přepíše) jeden gen z chromozomu v Genomu do modulu v RTLIL reprezentaci. Obě dvě tyto funkce budou virtuálními funkcemi a jejich realizace budou provedeny pro jednotlivé reprezentace samostatně. Pro potřeby CGP, které potřebuje počítat funkci fitness, je nutné být schopen provádět simulaci obvodu. Tato simulace bude implementována také na stejné úrovni jako přepis RTLIL hradla do Genomu, to bude provedeno pomocí virtuální metody `simulate`, která provede simulaci obvodu nad předanými vstupy. Realizace této funkce bude opět provedena v jednotlivých reprezentacích samostatně. Vzhledem k nutnosti propagace funkce mutace na tuto úroveň musí být přidána i virtuální metoda `mutate`, která bude realizována v jednotlivých reprezentacích samostatně a bude předávat původní metodě mutace z Genomu validní rozsah typů genů. Objekty třídy reprezentace budou také provádět interpretaci hodnot chromozomu a tedy i budou chápat co dané číslo v genu znamená narozdíl od objektů třídy Genom, které budou pracovat pouze s čísly a význam těchto čísel dále chápat nebudou. Funkce a datové struktury obsažené v reprezentaci tedy budou:

- `chromosome` genom reprezentace,
- `add_cell` přidání RTLIL hradla do genomu,
- `get_rtlil` zapsání genu do RTLIL modulu,
- `simulate` simulace obvodu,
- `power_loss` energetická náročnost obvodu v reprezentaci (počet tranzistorů),
- `mutate()` provedení mutace chromozomu,
- `clone()` vytvoření kopie sebe sama.

4.6 Návrh generace

Generace bude datovou strukturou uchovávající jednotlivé reprezentace obvodu, které bude chápat jako jedince. Nad jedinci jedné generace bude možné provádět operace, jako je křížení, mutace a replikace. Jedince bude možné do populace přidávat, či je z ní odebírat. Hlavním úkolem generace, která bude implementována jako třída, bude řídit ohodnocování jedinců. Ohodnocování jedinců bude prováděno za pomoci volání metody `simulate()` v objektu třídy reprezentace. Tato simulace se bude provádět společně s referenčním obvodem, který bude obvodem nepozměněným a obvodem, který je jedincem

populace, následně se provede operace XOR mezi výstupy referenčního obvodu a výstupy jedince generace. Tento výsledek operace XOR bude použit pro spočítání absolutní chyby jedince. Pro spočtení absolutní chyby jedince bude nutné obvod spustit se všemi kombinacemi vstupních signálů. Pro urychlení celé simulace se bude tato simulace provádět paralelně pro 256 kombinací. Tohoto urychlení bude dosaženo za pomoci použití vektorových instrukcí AVX2, které umožní pracovat s vektory o velikosti 256 bitů. Všem jedincům po dokončení simulace bude vypočteno jejich skóre (hodnota funkce fitness) a podle tohoto skóre budou seřazeni v populaci. To znamená, že jedinec s indexem 0 bude mít nejlepší skóre v populaci, jedinec s indexem, který je maximální v celé populaci, bude mít skóre nejhorší v celé populaci. Pokud hodnota skóre u některého z jedinců bude mimo povolený rozsah, nastaví se skóre na nekonečno. Operace křížení bude rozšířením klasického CGP a bude schopná křížit dva jedince s proměnným počtem crossover pointů. Tato operace nebude zahazovat méně crossover pointové potomky. Bližší popis operace křížení je v části 2.2.3. Operace replikace bude provádět zkopírování jedince pomocí metody `clone()` v objektu třídy reprezentace. Poslední operací je mutace, ta bude implementována pomocí volání funkce `mutate()` v objektu třídy reprezentace a toto volání bude provedeno nad množinou jedinců, která bude metodě předána. Funkce a datové struktury obsažené v generaci tedy budou:

- `individuals` pole párů hodnot funkce fitness a objektů třídy reprezentace,
- `reference` referenční reprezentace,
- `generation_size` velikost populace,
- `max_one_loss` maximální povolená chyba pro jednu kombinaci,
- `max_abs_loss` maximální povolená průměrná chyba,
- `power_accuracy_ratio` poměr váhy přesnosti výstupů a energetické náročnosti ve skóre,
- `cross()` operace křížení,
- `mutate()` operace mutace,
- `clone()` operace replikace,
- `selection()` provedení ohodnocení jedinců spojené se zachováním N jedinců s nejlepším skóre v populaci.

4.7 Generování kombinací pro paralelní simulaci a výpočet chyby

Generování kombinací pro paralelní simulaci musí být prováděno s vysokou rychlostí, aby nedocházelo ke zpomalování simulace. Protože paralelizace simulace probíhá po řádcích vstupů kombinací nikoli sloupcích, je nutné vytvořit rychlý způsob jak generovat kombinace od určitého řádku do jiného určitého řádku a zároveň být schopen zapisovat tyto kombinace atomicky z pohledu N-bitových vektorů, aby nebylo nutné pracovat s jednotlivými bity jednotlivých vektorů. Pro tento účel je nutné vytvořit generátor čísel, který bude schopen vytvořit nutné kombinace na úrovni bitů ale bez použití bitových operací nad N-bitovými vektory. Ukázka vygenorovaných kombinací pro simulaci čtyřvstupového obvodu za pomoci osmibitových vektorů, které jsme schopni paralelně simulovat, je na obrázku 4.3.

Simulace první

Vstup A 8b vektor	*Vstup B 8b vektor*	*Vstup C 8b vektor*	*Vstup D 8b vektor*
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

Simulace druhá

Vstup A 8b vektor	*Vstup B 8b vektor*	*Vstup C 8b vektor*	*Vstup D 8b vektor*
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

Obrázek 4.3: Ukázka kombinací pro simulaci čtyřvstupového obvodu za pomoci osmibitových vektorů

Na obrázku 4.3 je možné si povšimnout, že vektory vstupů B, C a D zůstávají stejnými pro obě dvě simulace. Tento jev není jevem náhodným a bude se opakovat u jakékoliv kombinace vstupů a velikosti vektorů, pouze se bude měnit počet konstantních vektorů. Tento počet je ovlivněn tím, kolik kombinací jsme schopni simulovat při jedné simulaci.

Pro vektor o velikosti osm bitů je to osm kombinací bez ohledu na to, kolik máme vstupů. Osm kombinací je počtem všech kombinací, které existují pro třívstupový kombinační obvod, proto jsou tři vektory konstantní. Díky tomu bude možné tyto tři vektory uvést jako konstanty do kódu a nebude je nutné generovat. Počet těchto vektorů lze vypočítat jako $log_2(velikost_vektoru)$. Pro vektor o velikosti 256 bitů tedy těchto konstant bude osm. Proto bude nutné generovat jen vektor A. Na obrázku 4.3 je vidět, že tento vekto má při první simulaci svou hodnotu rovnou nule, při druhé simulaci je jeho hodnota hodnotou maximální, tedy v binárním formátu se jedná o samé jedničky. Tehoto efektu lze dosáhnout pomocí jeho binární negace, tato negace je operací, která pracuje se všemi jednotlivými bity vektoru zároveň, tedy se jedná o operaci atomickou nad tímto vektorem. Stačí tedy už jen zjistit, kdy tuto negaci provést. U vstupu D si můžeme povšimnout, že k negaci bitu dochází každou jednu kombinaci. U vstupu C dochází k negaci bitu každé dvě kombiace. U vstupu B dochází k negaci bitu každé čtyři kombinace a u vstupu A dochází k negaci bitu každých osm kombinací, což je zároveň délka našeho simulačního vektoru. Z toho plyne, že k negaci tohoto vektoru by mělo docházet při každé simulaci. Obdobně jsme schopni odvodit chování i pro další vstup, který by svůj vektor negoval každou druhou simulaci (jednou za šestnáct kombinací). Pro libovolný vstup lze poté periodu negace, která vyjde v počtu simulací, vypočítat jako $\frac{2^{index_vstupu}}{velikost_vektoru}$, kde index prvního vstupu je nula. Tedy pro vstup s indexem tři (vstup A) a s velikostí simulačního vektoru osm bitů je tato perioda $\frac{2^3}{8} = \frac{8}{8} = 1$, což znamená, že k negaci bude docházet každou jednu simulaci. Tato funkcionalita lze tedy implementovat jako negace, která se provede po dopočítání počítadla nad určitým vstupem.

Aritmetická chyba výstupu se následně spočítá pomocí paralelní simulace referenční reprezentace, která je bezchybná, a reprezentace testované. U jednotlivých výstupů jednotlivých reprezentací se následně provede operace XOR. Výsledkem této operace jsou jedničkové bity, které určují chyby na výstupu. Protože jeden bit odpovídá jedné kombinaci, je nutné tento výsledek číst bit po bitu. Přečteme tedy nejdříve všechny první bity z výstupu XOR a vynásobíme je vahou portu, jehož jsou výstupem. Tyto součiny následně sečteme. Výsledný součet je chybou obvodu pro jednu kombinaci vstupů. Obdobně se pokračuje pro ostatní bity ve výstupu operace XOR.

4.8 Jazyk popisu váhy výstupů a jeho zpracování

Protože generace (třída) při hodnocení jedinců bude potřebovat váhy jednotlivých výstupů, bude nutné, aby existovala datová struktura, která bude schopna objektům třídy generace říci, jakou váhu má který výstup. K tomuto účelu byl navržen jazyk sloužící pro popis důležitosti jednotlivých výstupů a jeho parser, který bude mít za úkol komunikovat s instancemi třídy generace. Jazyk pro popis váhy výstupů je založen na základech jazyka yaml a tedy i soubory tohoto jazyka se zapisují s koncovkou .yaml kvůli podbarvování syntaxe v editorech a jeho přeložitelnosti jako yaml. Nicméně tento jazyk má jisté niance oproti jazyku yaml. Navržený jazyk má jednoduchou syntaxi, která má každý řádek ve formátu `\jmeno_portu: vaha_portu`. Dále je do syntaxe přidán znak `#`, který symbolizuje začátek komentáře a makra msb-first a lsb-first. Velikost znaků je při používání jazyka nedůležitá. Jazyk má také podporu vícebitových portů, jejichž váha se zapisuje pomocí čísel oddělených mezerou. Ukázka tohoto jazyka je na obrázku 4.4.

```
1  # generická hodnota
2  \sum: lsb-first
3  # makro hodnoty
4  \spi_data: msb-first
5  \spi2_data: LSB-FIRST
6
7  # vlastní hodnota
8  \uart_rx: 1024
9  # vlastní hodnoty pro vícebitový port
10 # LSB MSB
11 \uart_tx: 2 4 8 10
```

Obrázek 4.4: Návrh jazyku pro popis vah portů

Pokud v souboru napsaném v jazyce výše nebude některý z portů uveden, bude se k tomuto portu přistupovat, jako by byl msb-fisrt (MSB bit je na pozici nula). Obdobně se bude přistupovat k situaci, kdy bude definováno prvních N bitů pro M-bitový port, kdy M > N. Pokud by nastala situace, kdy v souboru je port s lsb-fisrt (LSB je na pozici 0) nebo msb-fisrt a port se v obvodu nenachází, bude vyvolána chyba. Samostatný parser jazyka se bude nacházet ve třídě `parse`. Tato třída bude mít funkce sloužící k parsování souboru s váhami portů, získání váhy signálu z RTLIL modelu a spočtení bitové velikosti portu. Funkce a datové struktury obsažené v parseru tedy budou:

- `ports` přiřazení vah jménům portů,
- `port_weight()` zjištění váhy portu,
- `parse_file()` zpracování souboru jazyka vah portů,
- `wire_bits()` zjištění bitové velikosti portu.

4.9 Převod RTLIL reprezentace na Genom (třída)

Převod z RTLIL reprezentace na Genom (třída) bude řízen uvnitř hlavního souboru rozšíření. Cílem tohoto převodu bude přemapovat jednotlivé `RTLIL::SigBit` signály na indexy v chromozomu ukazující na hradlo, na které jsou napojeny jako výstupní signály. Tohoto bude dosaženo za pomoci procházení RTLIL modelem po jednotlivých hradlech (buňkách), nad jejichž porty provedeme mapping a nad samotnou buňkou spustíme funkci `add_cell()` z třídy reprezentace. Funkce `add_cell()` nám vrátí index výstupního genu (jedno hradlo může být přeloženo jako více genů v chromozomu), tento index použijeme při mappingu signálu, který je nastaven jako signál výstupní. Vstupní signály budeme mapovat tak, že se nejdříve podíváme, zda jsme je již někdy mapovaly na index, pokud ano použijeme tento index, pokud ne vytvoříme nový gen, který označíme jako vstupní a jeho index použijeme jako index pro signál. Pokud by tento index byl v budoucnu použit u výstupu hradla, bude jeho označení jako vstupní zrušeno. Výstupní indexy jsou hledány jako indexy, které nebyly nikdy použity jako vstup hradla. Před mappingem signálů je nutné všechny signály spojené pomocí `RTLIL::SigSig` přepsat na jeden signál. Po projití všech hradel (buněk) v modulu RTLIL, bude z instance třídy Genom zavolána funkce `order()`, která slouží k seřazení genomu pro správnou funkcionalitu CGP. Toto seřazení posune všechny vstupní geny na začátek chromozomu a nastaví index pro jejich přeskočení při simulaci. Jednotlivé geny seřadí tak, aby každý gen měl všechny svoje vstupní indexy menší, než je jeho vlastní index, pokud tato podmínka při řazení nemůže být splněna, dojde k vyvolání chyby. Po dokončení seřazení se z mapy indexů a `RTLIL::SigBit` vytvoří mapa vstupů a výstupů chromozomu, která obsahuje indexy z chromozomu a `RTLIL::SigBit` z RTLIL modulu. Nakonec bude nutné odstranit všechna propojení `RTLIL::SigSig`, `RTLIL::Wire` a `RTLIL::Cell` z RTLIL modulu. `RTLIL::Wire` které jsou IN/OUT porty ostraněny nebudou.

4.10 Převod Genomu (třída) na RTLIL reprezentaci

Převod Genomu (třída) na RTLIL reprezentaci bude řízen uvnitř hlavního souboru rozšíření. Cílem tohoto převodu bude přemapovat jednotlivé geny v chromozomu na hradla (buňky) v RTLIL reprezentaci. Tohoto bude dosaženo za pomoci čtení jednotlivých genů z chromozomu v instanci třídy Genome. Toto čtení bude prováděno od prvního genu, který není vstupem a ukončeno bude na konci chromozomu. Každý přečtený gen bude předán funkci `get_rtlil()` ze třídy reprezentace. Této funkci budou předány vstupy a výstupy genu přemapované na `RTLLIL::SigBit`. Toto přemapování bude probíhat následovně. Zjistí se, zda index vstupu nebo genu (v případě výstupu) se nachází v mapě vstupů, výstupů nebo vnitřních propojení. Mapy vstupů a výstupů jsou součástí Genomu (třídy) a byly vytvořeny při převodu RTLIL reprezentace na Genom. Mapa vnitřních propojení je lokální mapou sloužící potřeby převodu a na začátku převodu je prázdná. Pokud se index z chromozomu bude v jedné z těchto map nacházet, bude použit `RTLIL::SigBit` přiložený k tomuto indexu. Pokud by index z chromozomu nebyl ani v jedné z map, bude vytvořen nový `RTLIL::SigBit`, který bude vložen zároveň s indexem z chromozomu do mapy vnitřních propojení. Po provedení tohoto postupu nad všemi geny bude nad modulem 0 zavolána funkce `fixup_port`. Pokud by vybraný Design obsahoval více modulů, rozšíření bude zapisovat pouze do modulu prvního. Ostaní moduly ponechá nezměněné.

4.11 Automatické testování rozšíření

Testování funkcionality rošíření bude rozděleno do několika částí. První částí bude testování načtení a zapsání obvodu bez použití CGP. Pro potřeby tohoto testu bude mít rozšíření parametr, který, když bude použit, způsobí přeskočení CGP části. Samotné testování pak bude probíhat pomocí syntézního skriptu. Tento syntézní skript načte zkoušený obvod, provede techmap, spustí cgploss s parametrem pro přeskočení CGP a zapíše výstupní Verilog. Nad výstupním Verilogem následně budou spuštěny testy pomocí programu `iverilog`, který prozkouší základní funkcionalitu výstupního Verilog souboru, zda nebyla požkozena. Testování CGP části bude probíhat obdobně jako testování načtení a zapsání s tím rozdílem, že se nebude přeskakovat CGP část, která bude v tomto případě spuštěna s maximální povolenou chybou, která bude nastavena na nulu (optimalizace bez povolení chyb na výstupu). Posledním typem testů budou jednotkové testy, ty budou prováděny pomocí zaměnění hlavního souboru rozšíření za soubor řídící test. Pokud test selže, ukončí Yosys s návratovým kódem rozdílným od nuly. Testy budou řízeny z Makefile a jejich typy budou rozeznány podle pojmenování složek, ve kterých budou umístěny.

4.12 Paralelizace simulace

Paralelizace simulace, jak již bylo zmíněno, bude probíhat pomocí instrukcí AVX2. Protože není vhodné, aby projekt byl závislí na jedné konkrétní platformě, jsou istrukce AVX2 zaměněny za SIMD instrukce, které jsou dostupné v GCC jako obecné vektorové instrukce, které podporují mnoho platforem, mezi které patří MMX, 3DNow, SSE, AVX či AVX2. Použítí těchto instrukcí je pomocí vytvoření speciálního datového typu, který kompilátoru GCC říká, že je vektorem. Ukázkový typ je na obrázku 4.5.

V příkladu výše je datový typ vektorem osmi hodnot typu int, za předpokladu 32b integeru. Číslo 32 uvnitř `vector_size` sděluje velikost vektoru v bajtech. Proto $32 \cdot 8/32 = 8$,

```
typedef int v8int __attribute__ ((vector_size (32)));
```

Obrázek 4.5: Ukázka definice vektorového datového typu

kde první číslo 32 je velikost vektoru, první číslo osm je počet bitů v bajtu, druhé číslo 32 je velikost vnitřního datového typu ve vektoru v bitech a druhé číslo osm je počet hodnot (čísel) ve vektoru. S proměnnými dotového typu `v8int` lze pak provádět většinu aritmetickologických operací a samotné GCC se stará o to, zda bude použito AVX2 nebo zda se bude pracovat s jednotlivými hodnotami v cyklu. Potože tento vektor bude 32bajtový, tedy bude 256bitový a lze s ním pracovat jen jako s celkem, nejsou operace, jako je přístup ke konkrétní hodnotě, úplně možné. Pokud bychom s ním chtěli pracovat i jako s datovým typem `int[8]`, je možné toto chování zajistit pomocí datové struktury `union` [24].

```
typedef union {
        v8int vector;
        int values[8];
} vec256;
```

Obrázek 4.6: Ukázka definice union pro datový typ v8int

Samotná paralelizace simulace bude prováděna pomocí logických instrukcí, které jsou těmto vektorům také dostupné a používají se stejně jako u standartních datových typů. Díky těmto instrukcím jde jednotlivé bity vektoru použít jako jednotlivé kombinace vstupu. Díky tomu bude možné dosáhnout paralelní simulace 256 a více kombinací. Tento postup využívá toho, že pokud provedeme logickou operaci nad jednobitovým číslem, bude výsledek stejný, jako kdybychom k tomuto číslu přidali další náhodnou hodnotu a z výsledku pak četli na odpovídajícím místě, kde je naše první hodnota. Například pokud máme biární číslo 1 a druhé 0 a provedeme operaci and, máme výsledek 0, pokud budeme mít číslo 110010 a provedeme logický and s 010110, výsledek bude 010010, kde na prvním místě zleva je výsledek našeho prvního příkladu [22].

Kapitola 5

Implementace navrženého rozšíření

Rozšíření cgploss bylo implementováno s ohledem na jeho návrh, který je popsán v kapitole 4. V této kopitole budou popsány implementační datily a niance oproti návrhu.

5.1 Rozložení kódu rozšíření

Zdrojové kódy rozšíření byly rozděleny na hlavičkové a zdrojové soubory. Hlavičkové soubory se nacházejí ve složce `include`. Každá třída je navíc s datovými strukturami, které využívá, zabalena do jmenných prostorů. Tyto jmenné prostory náleží problematikám, které tyto třídy řeší. Hlavičkové soubory obsahují především definice typů, tříd a deklarace funkcí. Zdrojové soubory se nacházejí ve složce `src` a obsahují především implementace tříd a funkcí. Složka `src` také obsahuje soubor `main.cpp`, který je hlavním souborem implementace a vkládá do sebe všechny ostatní soubory a knihovny. Výjimkou je soubor `convert.cpp`, který obsahuje funkce sloužící k převodu Genomu na RTLIL a opačně. Tyto funkce byly vyjmuty z `main.cpp` z důvodu velkého množství kódu provádějícího tyto přepisy. Nicméně toto vytknutí kódu bylo provedeno pouze do bočního souboru, který se do `main` přímo vkládá pomocí `include`, z totoho důvodu je tento soubor ve složce include. Adresář `tests` obsahuje kódy automatických testů, které mají každý svoji vlastní složku, ve které se nacházejí. Podle konce jména složky se rozeznává typ testu. Jedním z typů testů je `{cokoli}-`, který znamená, že tento test se nemá použít. Dálším typem je `{cokoli}-unit`, který značí jednotkový test. Všechna ostatní jména jsou posledním typem testu, kterým je Verilog test (test celé aplikace pro libovolný Verilog soubor). Složka *yosys* je složkou obsahující zdrojový kód nástroje Yosys a vznikla jako `git submodul`. Tato složka slouží jako zdroj hlavičkových souborů nástroje Yosys a také jako zdroj kompilátoru pro rozšíření.

5.2 Genom

Třída genome je implementována v jmenném prostoru genome. Obsah třídy genome je velmi podobný jejímu obsahu z návrhu, ale bylo přidáno pár podpůrných funkcí. Tyto funkce jsou `swap_genes`, `is_gene_ins_eqbelow`, `sort_asc_by_ins` a funkce pro převod genomu na String pro zápis do souboru při ladění aplikace. Funkce `swap_genes` slouží na prohození dvou genů v chromozomu za zachování jeho spojení s ostatními geny. Funkce `is_gene_ins_eqbelow` slouží jako pomocná funkce pro zjištění validity chromozomu pro CGP a zjišťuje, zda gen má všechny indexy svých vstupů menší než svůj vlastní index. Funkce `sort_asc_by_ins` je pomocnou funkcí pro `order` a provede vzestupné řa-

zení chromozomu podle indexů vstupů genů. Samotný chromozom je implementován jako `std::vektor` datové struktury `gene_t`. Struktura `gene_t` je implementována jako struktura obsahující `type` a `inputs[MAX_INPUTS]`. `Type` je implementován jako neznaménkové šestnáctibitové číslo. `Inputs` je implementován jako pole neznaménkových třicetidvoubitových čísel o velikosti `MAX_INPUTS`. `MAX_INPUTS` je makrem nastaveno na tři. Toto číslo bylo vybráno, protože žádná implementovaná reprezentace v chromozomu nepotřebuje více, jak třívstupý gen. Toto číslo slouží k určení alokované velikosti pole, nicméně používaná velikost je určena pomocí hodnoty proměnné `gene_inputs_count`, kterou používají všechny funkce pracující se vstupy genů k určení délky pole vstupů. Tato proměnná je nastavena objektem třídy reprezentace při jeho vytvoření. Funkce mutace, která nebyla v kapitole návrhu blíže popsána, byla implementována následovně. Funkci mutace budou předány parametry hodnoty pro střed normálního rozdělení, směrodatná odchylka normálního rozdělení, minimální hodnota typu genu, maximální hodnota typu genu a povolený rozsah mutace vstupu genu. Prvním krokem je vytvoření náhodného počtu mutací v chromozomu pomocí `std::normal_distribution`. Poté je provedeno N mutací, kde N je vygenerované náhodné číslo. Provádění jednotlivých mutací probíhá pomocí vygenerování náhodné pozice v chromozomu, následně se z náhodné pozice vybere gen. Zjistí se, zda gen je vnitřím genem nebo reprezentuje výstup. Pokud je výstupem, vytvoří se náhodné číslo v rozsahu velikosti chromozomu. Pokud se jedná o vnitřní gen, je vygenerována náhodná pozice v genu (libovolný vstup nebo typ genu) a na vygenerovanou pozici se zapíše nové náhodné číslo, které je generováno v povoleném rozsahu. Pokud se jedná o vstup, je využito parametru povoleného rozsahu, který se použije při generování náhodného čísla pro určení maxima a minima. Toto maximum a minimum bude určeno jako $sucasna_pozice + -povoleny_rozsah$, nicméně tato funkce bude použita pouze, pokud je povolený rozsah v parametru větší než nula, jinak bude použit obecný povolený rozsah pro CGP.

5.3 Reprezentace

Třída representation je implementována ve jmenném prostoru representation. Obsah třídy representation je velmi podobný jejímu obsahu z návrhu, ale bylo přidáno pár podpůrných funkcí. Tyto funkce jsou `set_rtlil_port` a `get_rtlil_port`. Obě dvě funkce slouží k zjednodušení práce s porty v RTLIL reprezentaci a jejich implementace se nachází v hlavičkové funkci. Funkce `set_rtlil_port` slouží k nastavení portu RTLIL buňky (hradla) za pomoci identifikátoru portu v chromozomu. Funkce `get_rtlil_port` slouží k získání `RTLIL::SigBit` náležícímu k danému indexu v chromozomu. Poslední pomocnou dodanou funkcí je funkce `save`, která slouží k uložení celé reprezentace včetně chromozomu do soboru. Tato funkce byla přidána kvůli potřebám ladění rozšíření.

5.3.1 AIG

Reprezentace AIG je implementována jako implementace abstraktní třídy representation. Celá implementace reprezentace AIG se nachází ve jmenném prostoru representation. Vzhledem k rozsáhlosti kódu této reprezentace je její implementace rezdělena do tří zdrojových souborů. Jeden ze souborů se zabývá její manipulací s chromozomem a převodem RTLIL na chromozom. Druhou částí je část zabývající se opačným převodem něž část první, tedy převodem chromozomu na RTLIL reprezentaci. Poslední částí je část řídící simulaci. První část se nachází v souboru `aig-genome.cpp` a jejím hlavním úkolem je implementovat `add_cell` funkci z reprezentace, nicméně také implementuje funkci `mutate`, která na této úrovni volá

funkci `mutate` z genome a předává jí svoji maximální a minimální hodnotu typu genu, která je určena samotnou implementací AIG reprezentace. Dále se v tomto souboru nacházejí podpůrné funkce, které slouží k přidání hradel, které nelze do chromozomu přidat jako jedn gen. Jedná se o funkce `add_mux`, `add_nmux`, `add_aoi3`, `add_oai3`, `add_aoi4`, `add_oai4`, `add_xor` a `add_xnor`. Tyto funkce přidávají do chromozomu více jak jeden gen podle typu hradla. Na pořadí přidaných genů nezáleží, protože po dokončení těchto funkcí bude nad chromozomem volána funkce `order`, která zajistí správné pořadí genů. Díky tomu záleží jen na správném propojení pomocí indexů ve vstupech genu. Nevýhodou těchto vícegenových hradel je to, že při opačném převodu nebudou sestaveny jako hradlo, které bylo původně v RTLIL reprezentaci, ale tak jak jsou nyní popsány v chromozomu, to znamená více jak jedním hradlem. Tento problém měl být ošetřen pomocí optimalizátoru, který by se spustil po dokončení CGP, bohužel jeho implementace nebyla z časových důvodů provedena. Nicméně tento problém se dá snadno vyřesit spuštěním průchodu abc, který tuto optimalizaci provede. U reprezentace AIG je jako typ použito číslo, jehož jednotlivé binární kombinace určují existence negací. Toto číslo reprezentuje negace pomocí prvních tří bitů. První bit reprezentuje negaci vstupu A, druhý negaci vstupu B a třetí negaci výstupu. Pokud je bit nulovým bitem, negace se na danném portu nenachází. V opačném případě, tedy pokud danný bit je bitem jedničkovým, je daný port portem negovaným. Druhá část se nachází v souboru `aig-rtlil.cpp` a jeho hlavním úkolem je implementovat funkci `get_rtlil`. Tato funkce má další podpůrné funkce, které slouží k přidání genů, u kterých buď jejich typ v RTLIL cell (Hradlu) není a je nutné jej vyjádřit pomocí více RTLIL cell (Hradel), nebo jejich typy mohou být přeloženy jako dva rozdílné typy cell v RTLIL (hradel). Mezi tyto funkce patří `rtlil_add_andnota`, `rtlil_add_ornota`, `rtlil_add_nand`. U funkcí `rtlil_add_andnota`, `rtlil_add_ornota` je důvodem to, že hradlo v genu má negaci na portu, kde RTLIL cell typ neumožňuje. U funkce `rtlil_add_nand` je důvodem, že je možné typ NAND v genu přeložit jako NOT nebo NAND (podle vstupů) v reprezentaci RTLIL. Poslední částí je část popisující simulace, tato část se nachází v souboru `aig-sim.cpp`, jehož obsahem je funkce `simulate`, která obsahuje popis jednotlivých typů genů, za pomoci binárních operací nad simulačním vektorem.

5.3.2 MIG

Reprezentace MIG je implementována jako implementace abstraktní třídy representation. Celá implementace reprezentace MIG se nachází ve jmenném prostoru representation. Vzhledem k rozsáhlosti kódu této reprezentace je její implementace rezdělena do tří zdrojových souborů. Jeden ze souborů se zabývá její manipulací s chromozomem a převodem RTLIL na chromozom. Druhou částí je část zabývající se opačným převodem něž část první, tedy převodem chromozomu na RTLIL reprezentaci. Poslední částí je část řídící simulaci. První část se nachází v souboru `mig-genome.cpp` a jejím hlavním úkolem je implementovat `add_cell` funkci z reprezentace, nicméně také implementuje funkci `mutate`, která na této úrovni volá funkci `mutate` z genome a předává jí svoji maximální a minimální hodnotu typu genu, která je určena samotnou implementací MIG reprezentace. Dále se v tomto souboru nacházejí podpůrné funkce, které slouží k přidání hradel, které nelze do chromozomu přidat jako jedn gen. Jedná se o funkce `add_mux`, `add_nmux`, `add_aoi3`, `add_oai3`, `add_aoi4`, `add_oai4`, `add_xor` a `add_xnor`. Tyto funkce přidávají do chromozomu více jak jeden gen podle typu hradla. Na pořadí přidaných genů nezáleží, protože po dokončení těchto funkcí bude nad chromozomem volána funkce `order`, která zajistí správné pořadí genů. Díky tomu záleží jen na správném propojení pomocí indexů ve vstupech genu. Nevýhodou těchto více-

genových hradel je to, že při opačném převodu nebudou sestaveny jako hradlo, které bylo původně v RTLIL reprezentaci, ale tak jak jsou nyní popsány v chromozomu, to znamená více jak jedním hradlem. Tento problém měl být ošetřen pomocí optimalizátoru, který by se spustil po dokončení CGP, bohužel jeho implementace nebyla z časových důvodů provedena. Nicméně tento problém se dá snadno vyřesit spuštěním průchodu abc, který tuto optimalizaci provede. U reprezentace MIG je jako typ použito číslo, jehož jednotlivé binární kombinace určují existence negací. Toto číslo reprezentuje negace pomocí prvních čtyř bitů. První bit reprezentuje negaci vstupu A, druhý negaci vstupu B, třetí negaci vstupu C a čtvrtý negaci výstupu. Pokud je bit nulovým bitem, negace se na danném portu nenachází. V opačném případě, tedy pokud daný bit je bitem jedničkovým, je daný port portem negovaným. Druhá část se nachází v souboru `mig-rtlil.cpp` a jeho hlavním úkolem je implementovat funkci `get_rtlil`. Tato funkce má další podpůrné funkce, které slouží k přidání genů, u kterých buď jejich typ v RTLIL cell (Hradelu) není a je nutné jej vyjádřit pomocí více RTLIL cell (Hradel), nebo jejich typy mohou být přeloženy jako dva rozdílné typy cell v RTLIL (hradel). Mezi tyto funkce patří `rtlil_add_maj3`, `rtlil_add_and`, `rtlil_add_or`. U funkce `rtlil_add_maj3`, je důvodem to, že hradlo, které je typem genu (Majority Gate), RTLIL cell nezná. U funkcí `rtlil_add_and` a `rtlil_add_or` je důvodem, že jejich rozeznaný typ, který nezahrnuje negace, bude po zahrnutí negací možné přeložit na velkou množinu RTLIL cell typů. Poslední částí je část popisující simulace, tato část se nachází v souboru `mig-sim.cpp`, jehož obsahem je funkce `simulate`, která obsahuje popis jednotlivých typů genů, za pomoci binárních operací nad simulačním vektorem.

5.3.3 Gates

Reprezentace Gates je implementována jako implementace abstraktní třídy representation. Celá implementace reprezentace Gates se nachází ve jmenném prostoru representation. Vzhledem k rozsáhlosti kódu této reprezentace je její implementace rezdělena do tří zdrojových souborů. Jeden ze souborů se zabývá její manipulací s chromozomem a převodem RTLIL na chromozom. Druhou částí je část zabývající se opačným převodem něž část první, tedy převodem chromozomu na RTLIL reprezentaci. Poslední částí je část řídící simulaci. První část se nachází v souboru `gates-genome.cpp` a jejím hlavním úkolem je implementovat `add_cell` funkci z reprezentace, nicméně také implementuje funkci `mutate`, která na této úrovni volá funkci `mutate` z genome a předává jí svoji maximální a minimální hodnotu typu genu, která je určena samotnou implementací Gates reprezentace. Dále se v tomto souboru nacházejí podpůrné funkce, které slouží k přidání hradel, které nelze do chromozomu přidat jako jedn gen. Jedná se o funkce `add_mux`, `add_nmux`, `add_aoi3`, `add_oai3`, `add_aoi4`, `add_oai4`. Tyto funkce přidávají do chromozomu více jak jeden gen podle typu hradla. Na pořadí přidaných genů nezáleží, protože po dokončení těchto funkcí bude nad chromozomem volána funkce `order`, která zajistí správné pořadí genů. Díky tomu záleží jen na správném propojení pomocí indexů ve vstupech genu. Nevýhodou těchto vícegenových hradel je to, že při opačném převodu nebudou sestaveny jako hradlo, které bylo původně v RTLIL reprezentaci, ale tak jak jsou nyní popsány v chromozomu, to znamená více jak jedním hradlem. Tento problém měl být ošetřen pomocí optimalizátoru, který by se spustil po dokončení CGP, bohužel jeho implementace nebyla z časových důvodů provedena. Nicméně tento problém se dá snadno vyřesit spuštěním průchodu abc, který tuto optimalizaci provede. U reprezentace Gates je jako typ použito číselná proměnná, jejíž jednotlivé hodnoty reprezentují jednotlivé typy. Čisla jednotlivým typům jsou přiřazena zapomoci maker v hlavičkovém souboru. Druhá část se nachází v souboru `gates-rtlil.cpp` a jeho

hlavním úkolem je implementovat funkci `get_rtlil`. Tato funkce má další podpůrné funkce, které slouží k přidání genů, nicméně se jedná pouze o funkce, které přímo přiřadí genu jeho RTLIL reprezentaci. Důvodem neexistence podpůrných funkcí (narozdíl od MIG a AIG), které by řešily neexistující typy RTLIL Cell, je ten, že reprezentace Gates mezi svými typy nepoužívá žádný typ, který nemá svoji obdobu v typu RTLIL Cell. Poslední částí je část popisující simulace, tato část se nachází v souboru `gates-sim.cpp`, jehož obsahem je funkce `simulate`, která obsahuje popis jednotlivých typů genů, za pomoci binárních operací nad simulačním vektorem.

5.4 Generace

Třída reprezentující generaci se v implementaci nazývá `generation`. Tato třída je umístěna v jmenném prostoru `evolution`. Obsah třídy `generation` je velmi podobný jejímu obsahu z návrhu, pouze bylo přidáno pár podpůrných funkcí. Tyto funkce jsou `create_kid`, `sort_individual_score_asc` a `score_individual`. Funkce `create_kid` slouží k vytvoření jednoho potomka za pomoci pole crossover pointů, indexů obou rodičů v generaci a popisu použití chromozomu rodičů v potomkovi. Tento popis je proveden pomocí neznaménkového šedesátičtyřbitového čísla, jeho jednotlivé bity se používají na popis jednotlivých částí chromozomu. Pokud máme tři crossover pointy, znamená to, že chromozom potomka bude rozdělen na čtyři části. Pokud v tomto případě bude mít číslo popisující chromozomy rodičů v potomkovi hodnotu `0b1010`, bude vytvořený potomek mít svůj chromozom složen z částí ABAB, kde A reprezentuje rodiče prvního a B rodiče druhého. V tomto čísle na binární úrovni je rodič A reprezentován nulou a Rodič B reprezentován jedničkou. Toto křížení chromozomů je implementováno pomocí kopírování genů od crossover pointu do crossover pointu, přičemž se kopíruje pouze chromozom rodiče B, protože potomek je nazačátku operace vytvořen jako replikace rodiče A, tedy obsahuje pouze chromozom rodiče A. Funkce `sort_individual_score_asc` slouží k porovnání dvou jedinců v generaci podle jejich skóre. Jedná se o funkci, která bude volána v rámci řazení jedinců podle skóre. Funkce `score_individual` je funkcí provádějící ohodnocení jedince spolu s jeho simulací. Samotná simulace začíná vytvořením dvou vektorů obsahujících simulační vektory, velikost vytvářených vektorů odpovídá počtu genů v chromozomu. Jeden vektor bude sloužit pro testovaného jedince a druhý pro jedince, který je jedincem referenčním, tedy má nulovou chybovost výstupu. Následně simulační vektory, které mají indexy odpovídající genům reprezentující vstupy obvodu, budou vyplněny kombinačními hodnotami pro paralelní simulaci, jak je popsáno v části 4.7. Po vyplnění vstupních hodnot je vyvolána funkce `simulate` nad hodnoceným jedincem, který je instancí reprezentace. Funkce `simulate` je taktéž i volána pro referenčního jedince. Po dokončení funkcí `simulate` je vytvořen vektor o velikosti počtu výstupů obvodu, do kterého je uložen výsledek operace xor mezi geny v hodnoceném a referenčním jedinci, které jsou označeny jako výstupní. V tomto vektoru se následně spočítá počet jedničkových bitů (počet kombinací jejichž výstup se lišil v referečním jedinci a hodnoceném jedinci). Z počtu jedničkových bitů je odvozena maximální chyba pro jednu kombinaci a absolutní chyba pro všechny kombinace. Pokud by chyba pro jednu kombinaci vyšla vetší než je povolena, bude simulace ihned ukončena se skóre, které bude rovno nekonečnu. Pokud ale splní tuto podmínku, dojde k spočtení další vstupní kombinace (pokud je dostupná), tak jak je popsáno v části 4.7. Tyto úkony se opakují, dokud nejsou vyčerpány všechny kombinace. Po dokončení všech simulací je spočtena průměrná chyba obvodu a je opět porovnána se svým povoleným maximem, pokud nesplní podmínku je jedinec ohodnocen nekonečným skóre. V opačném případě bude vypočteno skóre $score_{indiv}$.

$$score_{indiv} = (1 - power_accuracy_ratio) \cdot abs_error + power_accuracy_ratio \cdot power_loss$$

Kde `power_accuracy_ratio` je desetinným číslem v rozsahu nula až jedna a reprezentuje poměr důležitosti chybovosti a energetické náročnosti. `abs_error` reprezentuje průměrnou chybu obvodu a `power_loss` je hodnotou funkce, která je volána z reprezentace a vrací počet tranzistorů v dané reprezentaci. Funkce `score_individual` je podpůrnou funkcí funkce `selection`, která funkci `score_individual` volá nad všemi jedinci a následně podle vypočteného skóre jedince seřadí a ponechá v generaci N jedinců s nejlepším skóre. Ostatní jedince odstraní.

5.5 Zpracování jazyka pro popis důležitosti výstupů

Parser jazyka byl implementován ve třídě `parse` uvnitř jmenného prostoru `config`. Hlavním úkolem této třídy zůstává, jak bylo navrženo v návrhu, určování váhy jednotlivých porů. K tomuto účelu byla implementována datová struktura, která nese informace o jméně a váze portu. Tato struktura se nazývá `ports` a je implementována jako `std::map`, která mapuje `std::string` na `std::vector<unsigned>`. `std::string` ve struktuře označuje jméno portu ve vnitřní reprezentaci Yosys, tedy se jedná o jméno portu z Verilog souboru doplněné o znak \. Pokud tedy máme port se jménem `rx` ve Verilog souboru, bude tento port v této struktuře označen jako \`rx`. `std::vector<unsigned>` ve struktuře označuje váhu jednotlivých bitů výstupního portu. Pokud je port `rx` portem s velikostí osm bitů, bude mít tento vektor osm hodnot, reprezentujících váhy jednotlivých bitů. Tato struktura je naplněna ze vstupního souboru, který je zpracován pomocí funkce `parse_file`. Tato funkce čte soubor po řádcích a nad každým řádkem spustí funkci `parse_line`, která z řádku oddělí jméno portu a zbytek předá funkci `parse_desc`, která zajistí zpracování vah portu. Tyto funkce očekávají, že jména portů ve vstupním souboru budou označena pomocí jmen ve formátu Yosys, tedy budou začínat znakem \. Pro zpracování maker `msb-fisrst` a `lsb-fisrst` jsou ve třídě podpůrné funkce `generate_msb_first` a `generate_lsb_first`, které vytvoří podle jména portu vektor o velikosti bitové šířky portu. Tento vektor obsahuje mocniny dvojky. Bitovou šířku portů funkce zjistí pomocí volání funkce `wire_bits`, která se dotáže na velikost portu v chromozomu. Rozdíl mezi `generate_msb_first` a `generate_lsb_first` je pořadí těchto bitů. Pro `generate_msb_first` bude na nultém indexu ve vektoru hodnota 2^{N-1}, kde N je počet bitů v portu. U `generate_lsb_first` bude na nultém indexu ve vektoru hodnota 2^0. Mocniny čísla dva jsou v těchto funkcích počítány pomocí binárních posuvů. K přístupu k vahám dochází pomocí funkce `port_weight`, která z parametru, kterým je `RTLIL::SigBit`, vyčte jméno portu a jeho offset (o kolikátý bit ve vstupu se jedná). Poté se funkce zeptá, zda toto jméno je v mapě, pokud není, vrací generickou LSB-FIRST reprezentaci (hodnotu tedy vypočte jako 2^{offset}), pokud v ní je, vrací hodnotu z vektoru na pozici offset.

5.6 Struktura cgploss

Struktura cgploss je hlavní strukturou implementace a je implementována jako implementace rozhraní Pass v nástroji Yosys. Obsahuje pouze dvě funkce, které implementuje z rozhraní Pass. První funkcí je funkce `help`, která obsahuje vypsání základní nápovědy. Druhou funkcí je funkce `execute`, která obsahuje celou implementaci rozšíření. Tato funkce se dá

rozdělit do několika základních funkčních bloků. Prvním blokem je blok, který řeší načtení argumentů průchodu a jejich kontrolu korektnosti. Pokud nějaký argument nebude korektní, vypíše varování na STDOUT nástroje Yosys za pomoci funkce `log` a hodnotu nahradí za hodnotu generickou. Po zpracování parametrů následuje vytvoření prázdného chromozomu (objektu třídy genome) a prázdného `config_prasru`. Pokud bude nastaven parametr se souborem obsahujícím váhy portů, bude nad tímto souborem spuštěna funkce `parse_file` z vytvořeného `config_prasru`. V opačném případě bude u všech portů použito LSB-FIRST, tedy nultý bit v libovolném portu bude mít nejnižší váhu. Dále se pokračuje vytvořením nové reprezentace nad vytvořeným chromozomem (objekt třídy genome), která je typu určeného parametrem. Pokud by parametr nebyl zadán, bude použita reprezentace AIG. Nyní se přechází k části převádějící RTLIL reprezentaci na chromozom. Tento proces probíhá, za pomoci funkce `design2genome`, která bude volat funkci pro převod jedné buňky RTLIL na gen, tak jak bylo popsáno v části 4.9. Po dokončení převodu se dostáváme k samotné části, která řeší chod CGP. V této části je nejdříve vytvořena nová generace s parametry z části, která řešila zpracování parametrů. Do generace je vložena kopie vytvořené reprezentace, která byla vytvořena v předchozí části. Reprezentace, která byla vytvořena v předchozí části, je označena jako referenční reprezentace pro simulaci. Následně docházi ke klonování vloženého jedince v generaci, aby generace obsahovala požadovaný počet jedinců. Po dokončení klonování se nad všemi jedinci, kteří byli takto klonováni, aplikuje mutace, z tohoto důvodu má současná generace jednu reprezentaci shodnou s referenční a dalších N reprezentací, které jsou jejími mutovanými kopiemi. V tuto chvíli se provede selekce, která ponechá tolik jedinců, kolik bylo zadáno parametrem. Následuje replikace nebo křížení, to určuje parametr parents. Pokud hodnota parametru bude dva, použije se křížení, pokud bude jeho hodnota jedna bude použita replikace. Replikováni nebo kříženi budou všichni jedinci v populaci, kteří přežili selekci. Nicméně se v krajním případě může stát, že nebudou použiti všichni, protože maximální velikost generace je moc malá na to, aby se tam všichni vešli, v takovém případě budou replikování a klonováni jedinci s lepší hodnotou skóre přednostně. Následuje mutace, která bude aplikována na všechny potomky, kteří vznikli replikací nebo klonováním. Dále se pokračuje opět selekcí a celý ciklus se opakuje, dokud nebude dosaženo zadaného počtu generací. Po dovršení počtu generací je vybrán nejlepší jedinec ze současné populace a je nad ním volána funkce `cut_unused`, která odstraní nepoužité geny v chromozomu. Výsledný chromozom je převeden zpět do RTLIL pomocí funkce `genome2design`, tak jak bylo popsáno v kapitole 4.10

5.7 Použití

5.7.1 Kompilace rozšíření

Pro kompilaci rozšíření je nutné provést inicializaci a aktualizaci git submodulů. Toho dosáhneme pomocí následujících příkazů. Těmito příkazy bude dosaženo stažení nástroje Yo-

```
1  git submodule init
2  git submodule update
```

Obrázek 5.1: Inicializace a aktualizace git submodulů

sys, který je projektu přidán jako git submodul. Po přidání git submodulu můžeme provést samotný build rozšížení. Pro build aplikace je nutné mít nainstalované závislosti pro bu-

```
1  make
2  # nebo pro multicore verzi
3  make multicore
4
5  # nebo sestavení pomocí GCC (nedoporučuje se)
6  gcc -Wall -Wextra -ggdb -I/usr/local/share/yosys/include \
7  -MD -MP -D_YOSYS_ -fPIC -I/usr/local/include -std=c++11 \
8  -Os -DYOSYS_ENABLE_READLINE -DYOSYS_ENABLE_PLUGINS \
9  -DYOSYS_ENABLE_GLOB -DYOSYS_ENABLE_ZLIB -I/usr/include/tcl8.5 \
10 -DYOSYS_ENABLE_TCL -DYOSYS_ENABLE_ABC -DYOSYS_ENABLE_COVER \
11 -rdynamic -o cgploss.so -shared src/aig-genome.cpp \
12 src/aig-rtlil.cpp src/aig-sim.cpp src/config-parse.cpp \
13 src/gates-genome.cpp src/gates-rtlil.cpp src/gates-sim.cpp \
14 src/generation.cpp src/genome.cpp src/main.cpp src/mig-genome.cpp \
15 src/mig-rtlil.cpp src/mig-sim.cpp src/simulation.cpp \
16 -I yosys/ -I include/ -lstdc++ -lm -lrt -lreadline -lffi -ldl -lz -ltcl
```

Obrázek 5.2: Kompilace rozšíření

ild nástroje Yosys, který se také spustí tímto příkazem. Tyto balíčky lze na debian-base systémech nainstalovat pomocí Samotné rozšíření nevyžaduje žádné další balíčky. Pokud

```
1  sudo apt-get install build-essential clang bison flex \
2      libreadline-dev gawk tcl-dev libffi-dev git \
3      graphviz xdot pkg-config python3 libboost-system-dev \
4      libboost-python-dev libboost-filesystem-dev zlib1g-dev
```

Obrázek 5.3: Instalace závislostí pro kompilaci rozšíření

ale bude použit multicore build, bude nutné navíc nainstalovat balíček `libomp-dev`, který obsahuje OpenMP. Po dokončení příkazu make bude vytvořen soubor `cgploss.so`, který je sdílenou knihovnou obsahující build rozšíření cgploss.

5.7.2 Spuštění a zavedení do nástroje Yosys

Somotné načtení nástroje Yosys s rozšířením cgploss je možné provést pomocí příkazu make run, který spustí zkompilovaný submodul Yosys se samotným rozšířením cgploss. Druhou variantou je zavést rozšíření ručne pomocí parametru, při spouštění nástroje Yosys. Druhá

```
1    make run
2    #nebo
3    yosys -m cgploss.so
```

Obrázek 5.4: Spuštění rozšíření

varianta bude používat Yosys nainstalovaný v systému počítače, na kterém je tento příkaz spouštěn.

5.7.3 Ukázka použití

Pro potřeby této ukázky použijeme jednobitovou úplnou sčítačku. Tato sčítačka je popsána Verilog souborem `adder.v`, jehož obsah je na obrázku 5.5.

```
1  module fulladder (input a,
2                    input b,
3                    input c_in,
4                    output c_out,
5                    output sum);
6     assign {c_out, sum} = a + b + c_in;
7  endmodule
```

Obrázek 5.5: Obsah souboru adder.v

Nejdříve tedy spustíme rozšíření cgploss s nástrojem Yosys. Provedeme načtení Verilog

```
1  make run
2  #nebo
3  yosys -m cgploss.so
```

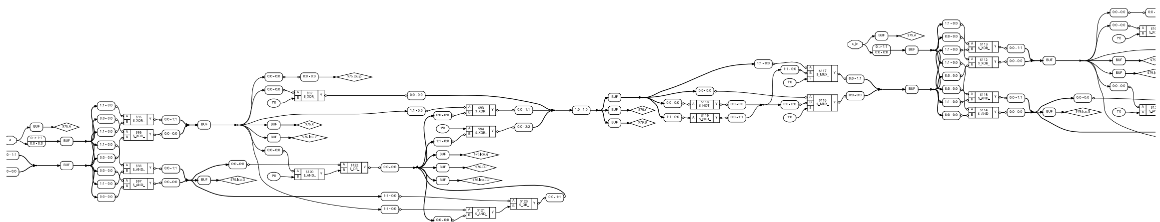
Obrázek 5.6: Spuštění rozšíření

souboru a jeho techmap na gate úroveň pomocí vnitřní cell knihovny. Pomocí příkazu

```
1  yosys> read_verilog code.v
2  yosys> techmap
```

Obrázek 5.7: Načtení verilog souboru a jeho techmap

`show` zobrazíme jak Yosys vysyntetizoval gate úroveň. Ukázka výstupu příkazu `show` je na obrázku 5.8. Protože je digram příliš velký ukazuje obrázek pouze část digramu.



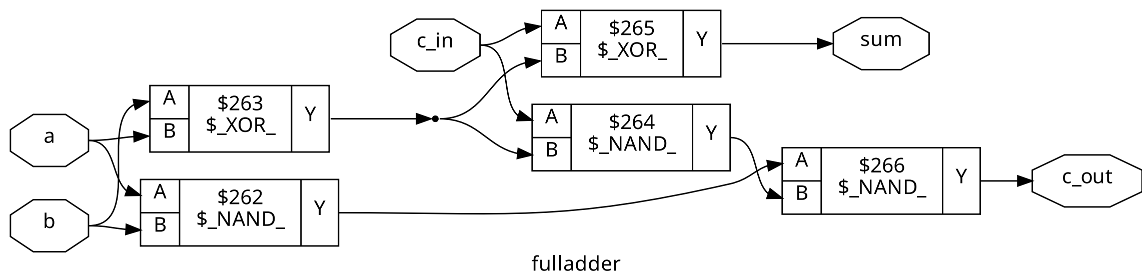
Obrázek 5.8: Výstup show po průchodu techmap

Důvodem nadměrné velikosti digramu je, že Yosys pomocí techmap namapoval naši jednobitovou celou sčítačku na celý ALU blok pomocí alumacc, ze kterého použije pouze sčítačku a z této sčítačky pouze malou část. Odstranění přebytečných bloků se provádí pomocí optimalizace, kterou implementuje například průchod `opt` nebo rozšíření `cgploss`. V následujícím kroku tedy pomocí `cgploss` zoptimalizujeme jednobitovou sčítačku bez povolení chyb na výstupu. Spuštění rozšíření cgploss lze různě kombinovat mezi sebou a i mezi ostatními průchody nástroje Yosys. V tomto případě došlo ke kombinaci všech

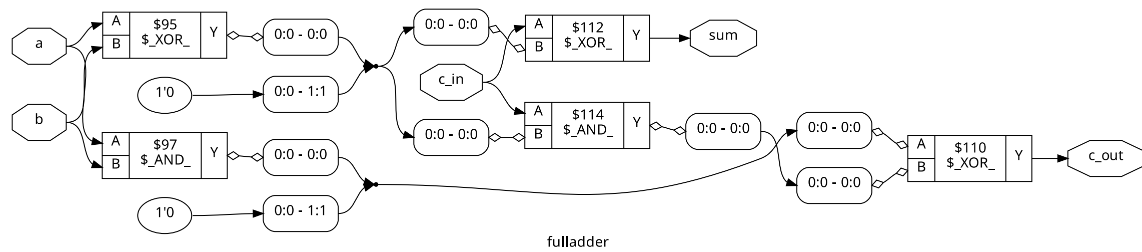
```
1    yosys> cgploss -generations=1000 -representation=mig -status
2    yosys> cgploss -generations=1000 -representation=aig -status
3    yosys> cgploss -generations=1000 -representation=gates -status
```

Obrázek 5.9: Spuštění průchodu cgploss třikrát po sobě s rozdílnými reprezentacemi

dostupných reprezentací. Po dokončení byl pomocí příkazu `show` opět vypsán obvod, který byl uvnitř vnitřní reprezentace Yosys. Tento obvod je na obrázku 5.10. Pro srovnání na obrázku 5.11 je výstup průchodu `opt` nad stejným obvodem.



Obrázek 5.10: Výstup show po průchodech cgploss



Obrázek 5.11: Výstup show po průchodu opt

Kapitola 6

Experimentování a testování

Rozšíření cgploss bylo testováno na několika kombinačních obvodech s cílem zjistit, jaký mají jednotlivé parametry CGP vliv na výsledek optimalizace. Vybrané obvody byly testovány pomocí bashového scriptu `profile_script.sh`, který spustil paralelně několik nástrojů Yosys s rozšířením cgploss a tomuto nástroji předal syntézní skripty vytvořené souborem `generateScripts.py`. Po dokončení byly uloženy informace o jedincích v generacích pomocí parametru $-profile$. Vytvořené soubory byly zanalyzovány scriptem `plotOutput.py` a z dat obsažených v souborech byly vytořeny grafy, které jsou součástí této práce. Samotné testování bylo provedeno na několika nezávislých počítačích. Jedním z počítačů, který prováděl testování, je stolní počítač s procesorem AMD Ryzen 5 2400G @ 3.6GHz s 8 jádry, 16 GB RAM a 64bitovým operačním systémem Arch linux. Dalším testovacím počítačem byl pronajatý výpočetní server s procesorem AMD EPYC 7B13 @ 2.5GHz s 8 jádry, 32 GB RAM a 64bitovým operačním systémem Debian. Posledním použitým počítačem byl výpočetní cluster, který je založen na blade serverech IBM a Dell [6]. Obvody vybrané pro testování byly čtyřbitová sčítačka, osmibitová sčítačka a osmibitová násobička. Mezi testovanými parametry byly především různé reprezentace a různé maximální povolené chyby. Maximální průměrná povolená chyba (MAE) byla testována ve větším rozsahu než maximální povolená chyba pro jednu kombinaci (WCE). Cílem testovaní bylo zjistit, jaká reprezentace je vhodnější pro daný typ obvodu a pro danou maximální chybu (WCE i MAE). Hotnota funkce fitness je v experimentech hodnocena tak, že když je její hodnota nižší, jedná se o lepší řešení. Grafy a hodnoty se vždy týkají nejlepšího jedince v generaci. Každý experiment byl proveden pětkrát a jeho hodnoty byly zprůměrovány. K spuštění rozšíření cgploss docházelo v experimentech hned po dokončení průchodu `techmap`, testovaný obvod tedy nebyl nijak optimalizován.

6.1 Testovaný obvod: čtyřbitová sčítačka

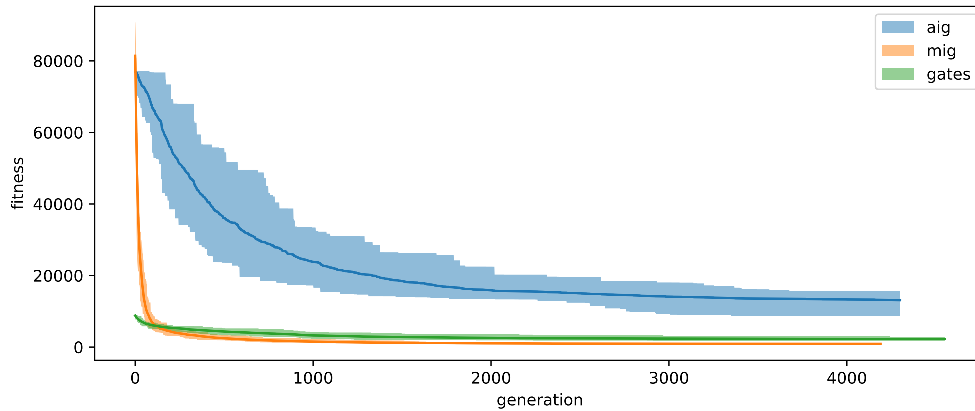
Čtyřbitová sčítačka byla testována na počítači s procesorem AMD Ryzen 5 2400G @ 3.6GHz s 8 jádry, 16 GB RAM a 64bitovým operačním systémem Arch linux. Chod jedné konfigurace byl nastaven na čtyři minuty. Celkem se jednalo o 1440 kombinací, které testovaly především maximální povolené MAE a to konkrétně z množiny $\{0, 10, 20\}$, maximální povolené WCE z množiny $\{0, 5, 10\}$ a reprezentace $\{"mig", "aig", "gates"\}$.

6.1.1 Optimalizace přesné sčítačky

Optimalizace bez chyby na výstupu byla provedena pomocí parametrů

```
-generations=10000 -max_duration=4 -representation={"mig", "aig", "gates"}
-max_abs_error=0 -max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75 -profile
-selection_size=1.
```

Graf na obrázku 6.1 ukazuje průměr hodnoty fitness s jejím rozpětím u jednotlivých reprezentací v průběhu generací.



Obrázek 6.1: Hodnota funkce fitness v průběhu generací

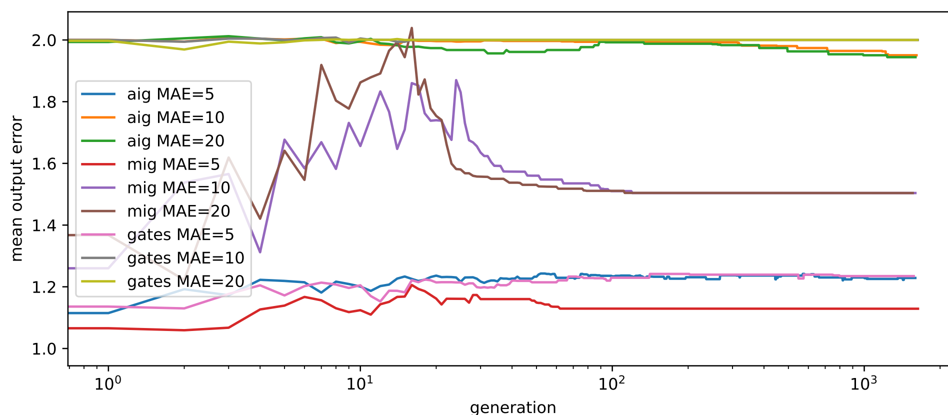
Jedna generace u každé reprezentace trvala jinou časovou dobu, protože má každá jiný počet simulovaných prvků. Z tohoto důvodu každá reprezentace končí na grafu na jiné generaci. Počáteční hodnota funkce fitness je pro AIG i MIG téměř shodná, ale pro reprezentaci gates je jiná, tento jev je zapříčiněn velkým množstvím hradel xor v optimalizovaném obvodu, které v genomu AIG i MIG musejí být zapsány jako kombinace několika genů narozdíl od reprezentace GATES. Na grafu se dá pozorovat celkem malý optimalizační potenciál reprezentace gates, která v tomto obvodu provedla celkem malou optimalizaci narozdíl od AIG a MIG, nicméně tato reprezentace i tak dosáhla lepšího výsledku než reprezentace AIG, protože měla menší počet genů. Dále se dá z grafu vyčíst, že pro optimalizaci čtyřbitové sčítačky stačilo prvních 1000 generací, protože v dalších generacích již nedocházelo k markantnímu zlepšení. Jako reprezentace s největším potenciálem se v tomto případě ukázalo MIG, které dosáhlo lepšího výsledku než GATES, přestože mělo na začátku mnohem horší hodnotu funkce fitness.

6.1.2 Optimalizace pro maximální povolenou průměrnou chybu

Optimalizace s chybou na výstupu byla provedena pomocí parametrů

```
-generations=10000 -max_duration=4 -representation={"mig", "aig", "gates"}
-max_abs_error={5,10,20} -max_one_error=999999999 -generation_size=300
-mutations_count=10 -mutations_count_sigma=9 -parents=1
-power_accuracy_ratio=0.75 -profile -selection_size=1
```

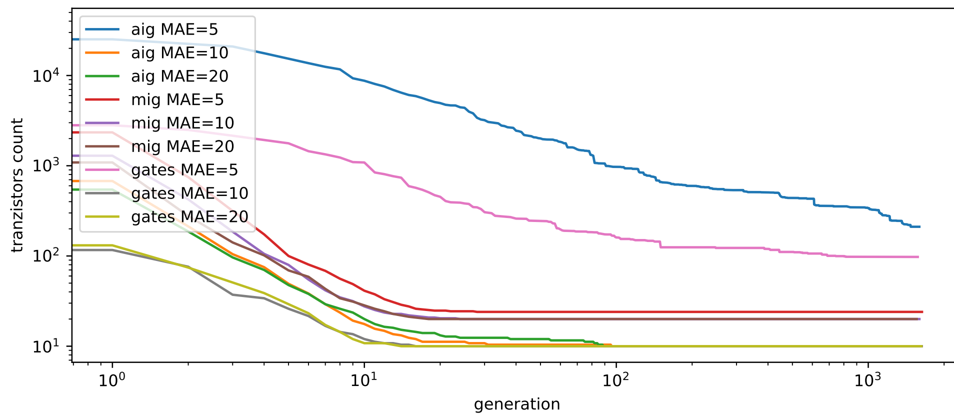
Protože zkoumáme jednotlivé parametry maximálního povoleného MAE, je parametr pro maximální povolené WCE ponechán jako maximum, kterého tento obvod ani nemůže dosáhnout. Graf s osou X v logaritmickém měřítku na obrázku 6.2 ukazuje průměrnou výstupní chybu v průběhu generací.



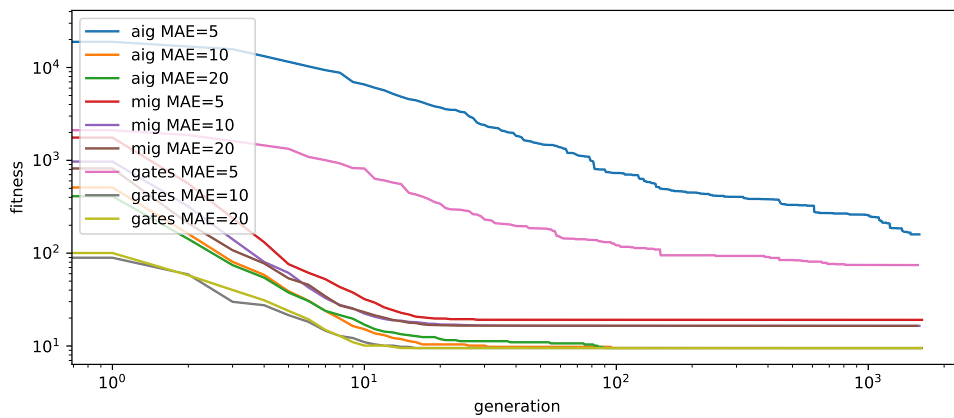
Obrázek 6.2: MAE v průběhu generací

Na obrázku 6.2 lze pozorovat, že všechny reprezentace zvládly využít možnost výstupní chyby za účelem vyšší optimalizace. Reprezentace AIG a GATES využívají průměrnou výstupní chybu zhruba ve stejné míře. Reprezentace MIG využívá výstupní chyby mnohem méně. Bohužel, jak můžeme vidět na obrázku 6.3, který ukazuje počet použitých tranzistorů, a na obrázku 6.4, který ukazuje hodnotu fitness funkce, nedosahuje tato reprezentace lepších výsledků než reprezentace AIG i GATES. Nicméně je možné pozorovat, že čím je povolená chyba menší, tím je MIG úspěšnější v porovnání s AIG a GATES.

Na obrázku 6.3 lze pozorovat, že reprezentace AIG, která používá zhruba stejně velkou výstupní chybu jako reprezentace GATES, v případě MAE=5 GATES dosáhne mnohem lepšího výsledku ve funkci fitness. Nejpravděpodobnějším důvodem je, že pro MAE=10 a MAE=20 existují snadno nalezitelná optima, zato v případě MAE=5 toto snadné optimum neexistuje a optimalizace je pomalejší. Z tohoto důvodu je reprezentace MIG v případě MAE=5 nejefektivnější.



Obrázek 6.3: počet tranzistorů v průběhu generací



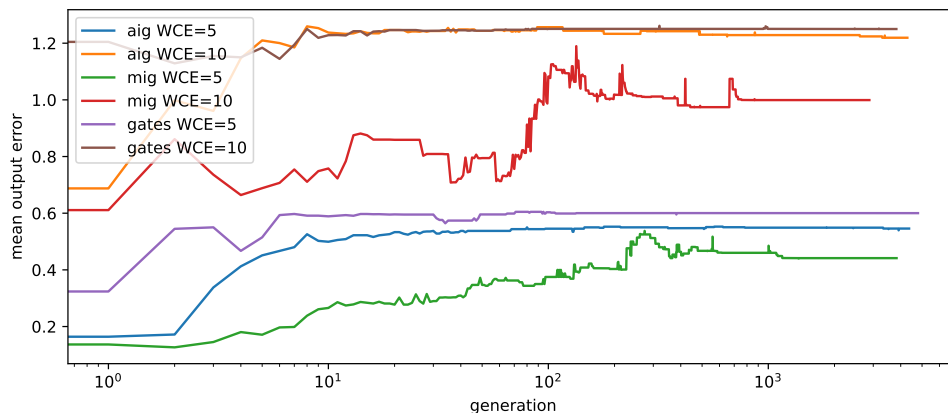
Obrázek 6.4: hodnota funkce fitness v průběhu generací

6.1.3 Optimalizace pro maximální povolenou chybu jedné kombinace

Optimalizace s chybou na výstupu byla provedena pomocí parametrů

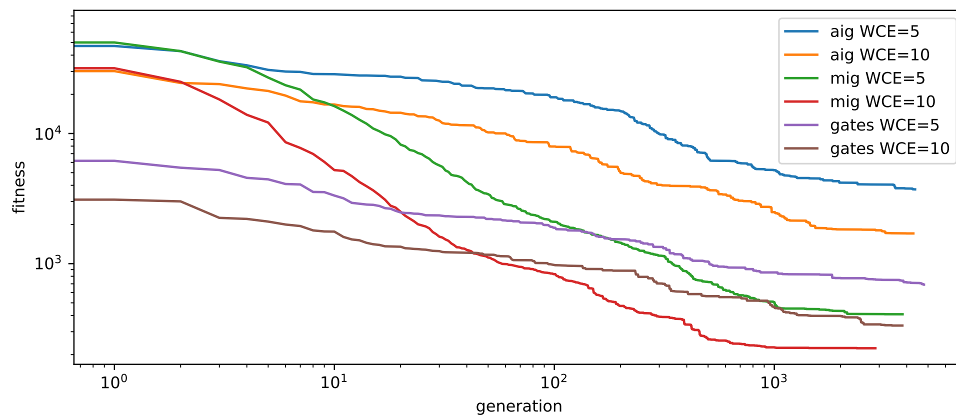
```
-generations=10000 -max_duration=4 -representation={"mig", "aig", "gates"}
-max_abs_error=999999999 -max_one_error={5, 10} -generation_size=300
-mutations_count=10 -mutations_count_sigma=9 -parents=1
-power_accuracy_ratio=0.75 -profile -selection_size=1
```

Protože zkoumáme jednotlivé parametry maximálního povoleného WCE, je parametr pro maximální povolené MAE ponechán jako maximální. Toto maximum není obvod schopen dosáhnout. Graf na obrázku 6.5 ukazuje průměrnou výstupní chybu v průběhu generací.



Obrázek 6.5: MAE v průběhu generací

Graf na obrázku 6.5 ukazuje opět, že z pohledu průměrné chyby bylo WCE používáno nejméně častěji v reprezentaci MIG. U reprezentací AIG a GATES opět dochází k podobné míře využití. Nicméně narozdíl od optimalizace pro maximální povolené MAE je zde neje-fektivnější reprezentací MIG, jak je možné vidět na grafu hodnot funkce fitness. Důvodem je nejspíše to, že v předchozím případě bylo optimum, které našli reprezentace AIG a GATES snadno, špatně vyjádřitelné v reprezentaci MIG.



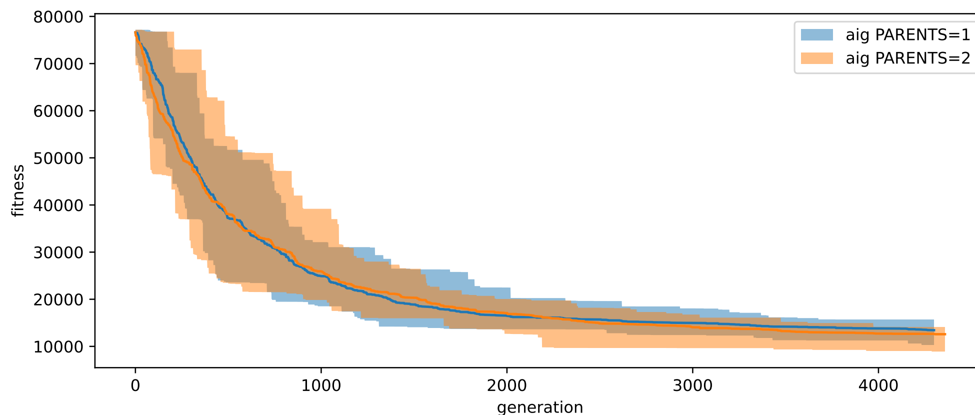
Obrázek 6.6: hodnota funkce fitness v průběhu generací

6.1.4 Parametr počtu rodičů

Test pro zjištění, jak parametr ovlivňuje úspěšnost CGP, byl proveden s parametry

```
-generations=10000 -max_duration=4 -representation="aig" -max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -power_accuracy_ratio=0.75 -profile
-selection_size=1 -parents={1,2}
```

Cílem testu bylo zjistit, zda u tohoto obvodu povede operace křížení k nějakému zlepšení nebo ne. Graf na obrázku 6.7 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.

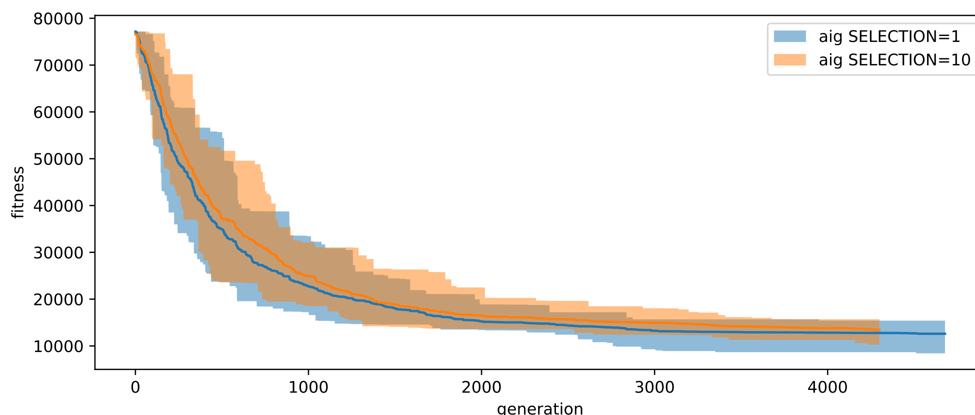


Obrázek 6.7: hodnota funkce fitness v průběhu generací

Z grafu na obrázku 6.7 se nepodařilo prokázat, že by varinta s křížením provedla velké zlepšení nebo zhoršení oproti jednorodičové variantě (pouze replikace a mutace). Proto je jedno jaká varianta vytváření potomků se u tohoto typu obvodu zvolí.

6.1.5 Parametr velikosti selekce

Test pro zjištění, jak parametr ovlivňuje úspěšnost CGP, byl proveden s parametry -generations=10000 -max_duration=4
-representation="aig"-max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75 -profile
-selection_size={1,10} . Cílem testu bylo zjistit, zda u tohoto obvodu provede změna velikosti selekce nějaké razantní změny. Graf na obrázku **??** ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.8: hodnota funkce fitness v průběhu generací

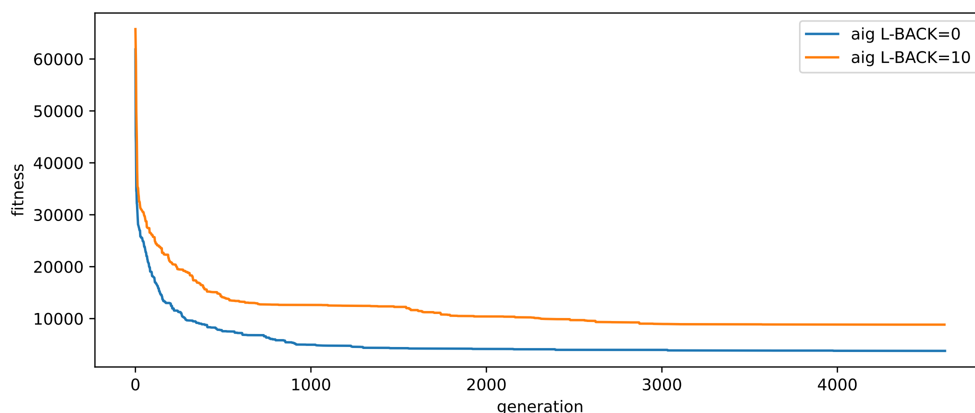
Z grafu na obrázku 6.8 se nepodařilo prokázat, že by velikost selekce měla pozitivní či negativní vliv na hodnotu funkce fitness v průběhu generací.

6.1.6 Parametr l-back

Cílem tohoto parametru je zajistit lokálnější vliv mutace na geny. Očekávané chování je tedy takové, že CGP bude pomaleji hledat optimální řešení, ale díky lokálním změnám budou prudké změny v zapojení trvat déle. Díky tomu budou upřednostňovány změny, které nebudou tolik měnit obvod. Pro ověření této teorie byl spuštěn test s parametry

```
-generations=10000 -max_duration=4 -representation="aig" -max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75 -profile
-selection_size=1 -l-back={0, 10}
```

U tohoto parametru hodnota nula znamená, že nemá být použit, pokud je jeho hodnota vyšší než nula, bude povolena mutace v rozsahu +-hodnota. Graf na obrázku 6.9 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.9: hodnota funkce fitness v průběhu generací (l-back=0 -> unlimited)

Na grafu na obrázku 6.9 je vidět, že parametr l-back opravdu zpomalil optimalizaci, nicméně výsledek byl u tohoto obvodu horší než u varianty bez l-back.

6.2 Testovaný obvod: osmibitová sčítačka

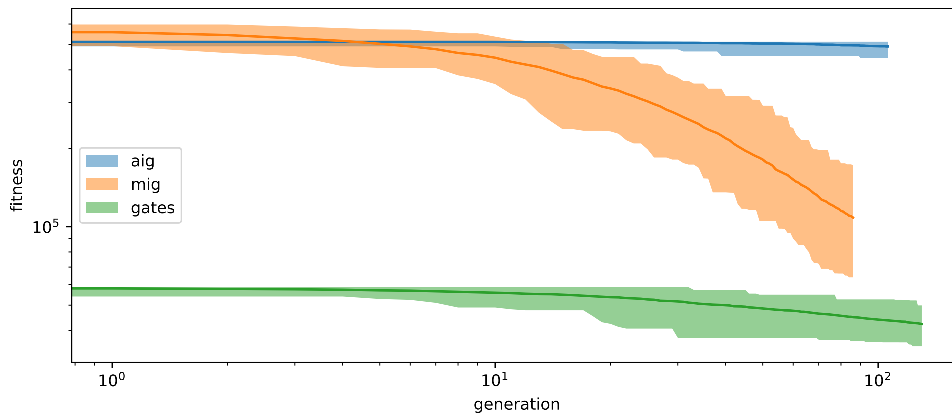
Osmibitová sčítačka byla testována na školním clusteru. Chod jedné konfigurace byl nastaven na třicet minut. Celkem se jednalo o zhruba 3000 kombinací, které testovaly především maximální povolené MAE a to konkrétně z množiny {0, 5, 10, 20, 50, 100}, maximální povolené WCE z množiny {0, 5, 10, 20, 50} a reprezentace {"aig", "gates", "mig"}.

6.2.1 Optimalizace přesné sčítačky

Optimalizace bez chyby na výstupu byla provedena pomocí parametrů

```
-generations=10000 -max_duration=30 -representation={"mig", "aig", "gates"}
-max_abs_error=0 -max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75
-profile -selection_size=1
```

Graf na obrázku 6.10 ukazuje průměr hodnoty fitness s jejím rozpětím u jednotlivých reprezentací v průběhu generací.



Obrázek 6.10: hodnota funkce fitness v průběhu generací

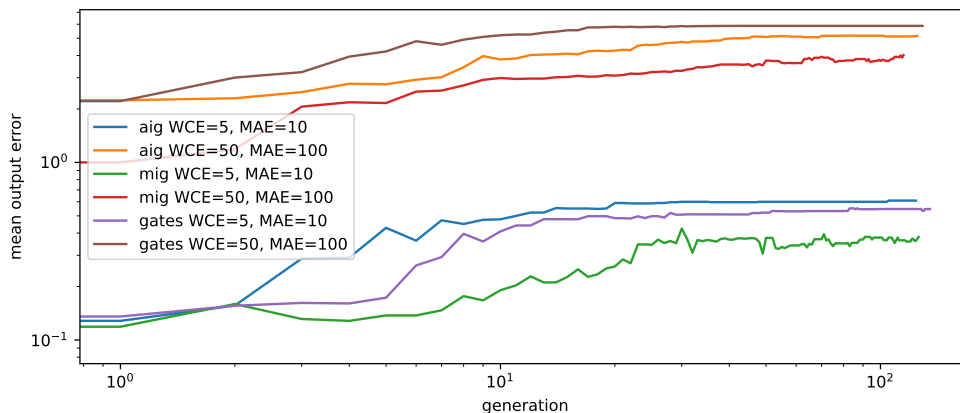
Jedna generace u každé reprezentace trvala jinou časovou dobu, protože má každá jiný počet simulovaných prvků. Z tohoto důvodu každá reprezentace končí na grafu na jiné generaci. Počáteční hodnota funkce fitness je pro AIG i MIG téměř shodná, ale pro reprezentaci GATES je jiná, tento jev je zapříčiněn velkým množstvím hradel xor v optimalizovaném obvodu, které v genomu AIG i MIG musejí být zapsány jako kombinace několika genů narozdíl od reprezentace GATES. Na grafu se dá pozorovat celkem malý optimalizační potenciál reprezentace AIG, která v tomto obvodu provedla celkem malou optimalizaci narozdíl od GATES a MIG narozdíl od čtyřbitové sčítačky, kde AIG nebylo nejpomalejší reprezentací v optimalizaci. Reprezentace GATES v těchto bězích skončila s nejlepším výsledkem, nicméně optimalizovala mnohem pomaleji než MIG a využívala své výhody, kterou je existence xor hradel ve svých typech.

6.2.2 Optimalizace pro povolenou výstupní chybu

Optimalizace s chybou na výstupu byla provedena pomocí parametrů

```
-generations=10000 -max_duration=3 -representation={"mig", "aig", "gates"}
-max_abs_error={5,10,20,50,100} -max_one_error={5,10,20,50}
-generation_size=300 -mutations_count=10 -mutations_count_sigma=9
-parents=1 -power_accuracy_ratio=0.75 -profile -selection_size=1
```

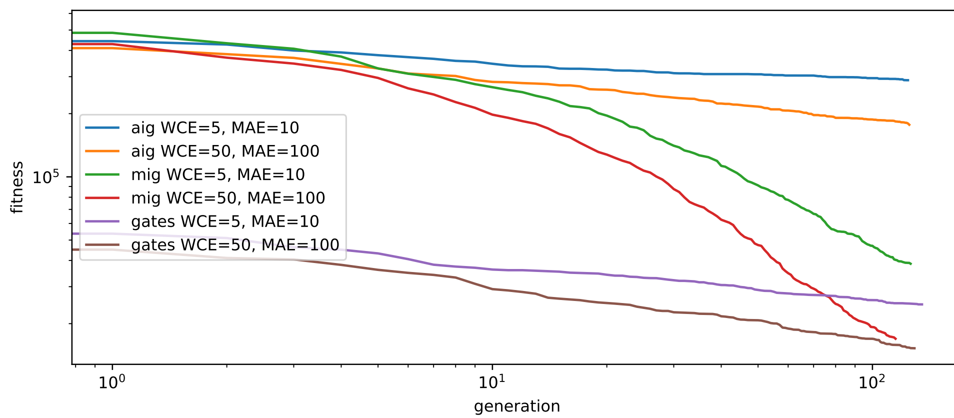
Kvůli velkému množství kombinací, byly do grafu vneseny pouze ty kombinace, které měly více rozdílné hodnoty od ostatních. Graf s osou X a Y v logaritmickém měřítku na obrázku 6.11 ukazuje průměrnou výstupní chybu v průběhu generací.



Obrázek 6.11: MAE v průběhu generací

Na obrázku 6.11 lze pozorovat, že všechny reprezentace zvládly využít možnost výstupní chyby za účelem vyšší optimalizace. Reprezentace MIG využívá výstupní chyby mnohem méně než ostatní, nicméně jak můžeme vidět na obrázku 6.18, který ukazuje hodnotu fitness funkce, dosahuje tato reprezentace lepších optimalizačních skoků než reprezentace AIG i GATES. Z grafu lze také pozorovat, že reprezentace GATES je nejefektivnější v zavádění chyb do výstupu.

Na obrázku 6.18 lze pozorovat, že reprezentace AIG, která používá zhruba stejně velkou výstupní chybu jako reprezentace GATES, dosahuje mnohem horších výsledků, nicméně je nutné započítat, že začínala na mnohem horší hodnotě funkce fitness.



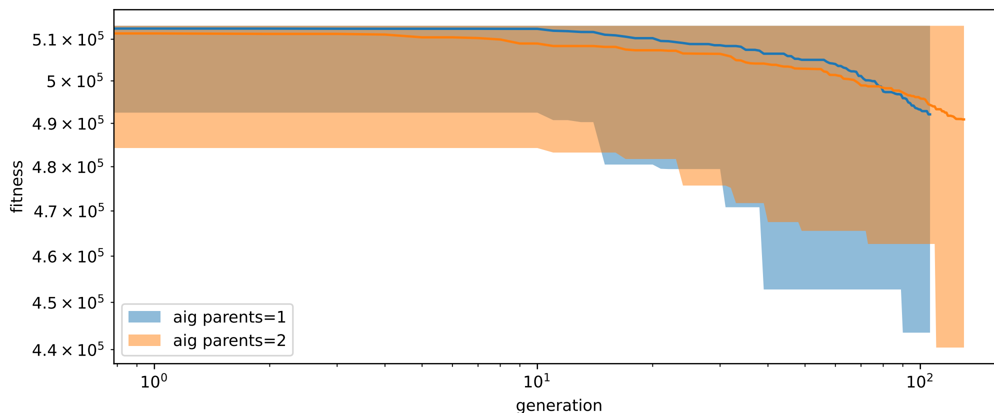
Obrázek 6.12: hodnota funkce fitness v průběhu generací

6.2.3 Parametr počtu rodičů

Test pro zjištění, jak parametr ovlivňuje úspěšnost CGP, byl proveden s parametry

```
-generations=10000 -max_duration=30 -representation="aig"
-max_abs_error=0 -max_one_error=0 -generation_size=300
-mutations_count=10 -mutations_count_sigma=9
-power_accuracy_ratio=0.75 -profile -selection_size=1 -parents={1,2}
```

Cílem testu bylo zjistit, zda u tohoto obvodu povede operace křížení k nějakému zlepšení nebo ne. Graf na obrázku 6.13 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.13: hodnota funkce fitness v průběhu generací

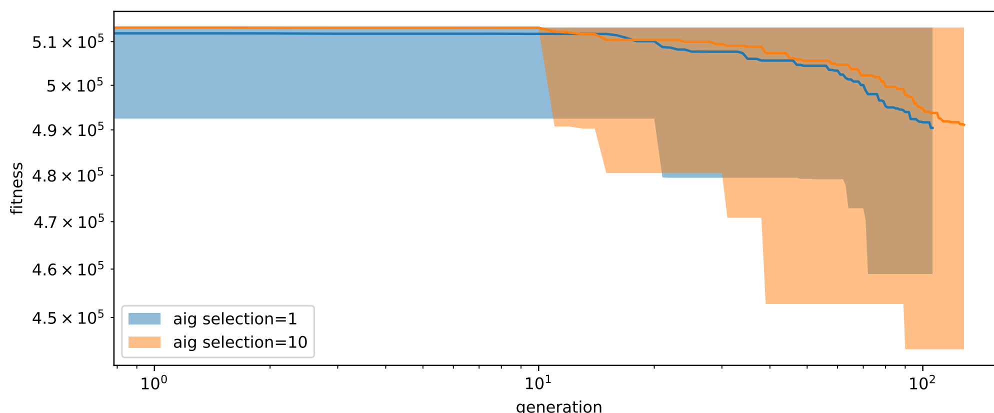
Z grafu na obrázku 6.13 se nepodařilo prokázat, že by varianta s křížením provedla velké zlepšení nebo zhoršení oproti jednorodičové variantě (pouze replikace a mutace). Proto nezáleží na tom, jaká varianta vytváření potomků se u tohoto typu obvodu zvolí.

6.2.4 Parametr velikosti selekce

Test pro zjištění, jak parametr ovlivňuje úspěšnost CGP byl proveden s parametry

```
-generations=10000 -max_duration=30 -representation="aig" -max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75 -profile
-selection_size={1,10}
```

Cílem testu bylo zjistit, zda u tohoto obvodu provede změna velikosti selekce nějaké razantní změny. Graf na obrázku 6.14 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.14: hodnota funkce fitness v průběhu generací

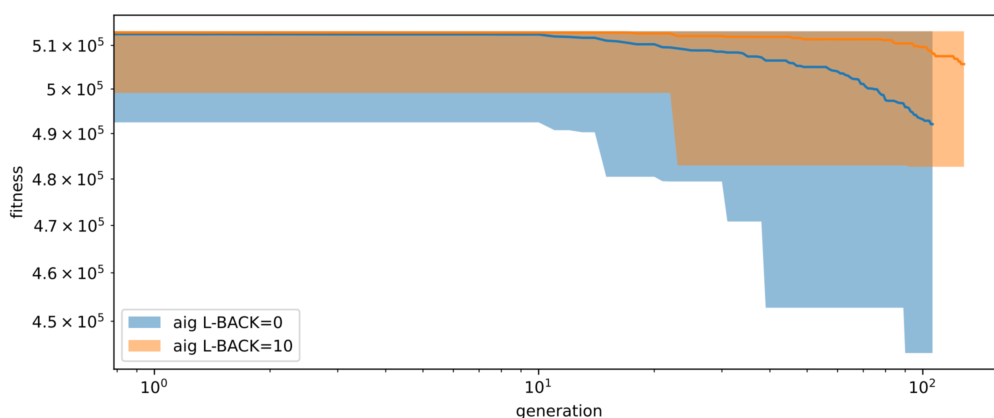
Z grafu na obrázku 6.14 se nepodařilo prokázat, že by velikost selekce měla pozitivní či negativní vliv na hodnotu funkce fitness v průběhu generací.

6.2.5 Parametr l-back

Pro ověření teorie popsané u parametru l-back u čtyřbitové sčítačky byl spuštěn test s parametry

```
-generations=10000 -max_duration=4 -representation="aig" -max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75 -profile
-selection_size=1 -l-back={0, 10}
```

U tohoto parametru hodnota nula znamená, že nemá být použit, pokud je jeho hodnota vyšší než nula, bude povolena mutace v rozsahu +-hodnota. Graf na obrázku 6.15 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.15: hodnota funkce fitness v průběhu generací (l-back=0 -> unlimited)

Na grafu na obrázku 6.15 je vydět že parametr l-back opravdu zpomalil opitimalizaci, nicméně výsledný výsledek byl u totoho obvodu horší než u varinaty bez l-back.

6.3 Testovaný obvod: osmibitová násobička

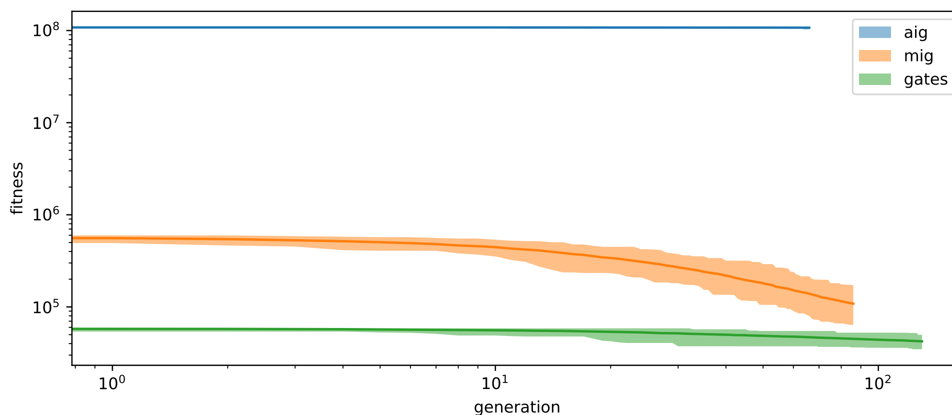
Osmibitová násobička byla testována na školním clusteru. Chod jedné konfigurace byl nastaven na šedesát minut. Celkem se jednalo o zhruba 3000 kombinací, které testovaly především maximální povolené MAE a to konkrétně z množiny {0, 5, 10, 20, 50, 100}, maximální povolené WCE z množiny {0, 5, 10, 20, 50} a reprezentace {"aig", "gates", "mig"}.

6.3.1 Optimalizace přesné násobičky

Optimalizace bez chyby na výstupu byla provedena pomocí parametrů

```
-generations=10000 -max_duration=60 -representation={"mig", "aig", "gates"}
-max_abs_error=0 -max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75
-profile -selection_size=1
```

Graf na obrázku 6.16 ukazuje průměr hodnoty fitness s jejím rozpětím u jednotlivých reprezentací v průběhu generací.



Obrázek 6.16: hodnota funkce fitness v průběhu generací

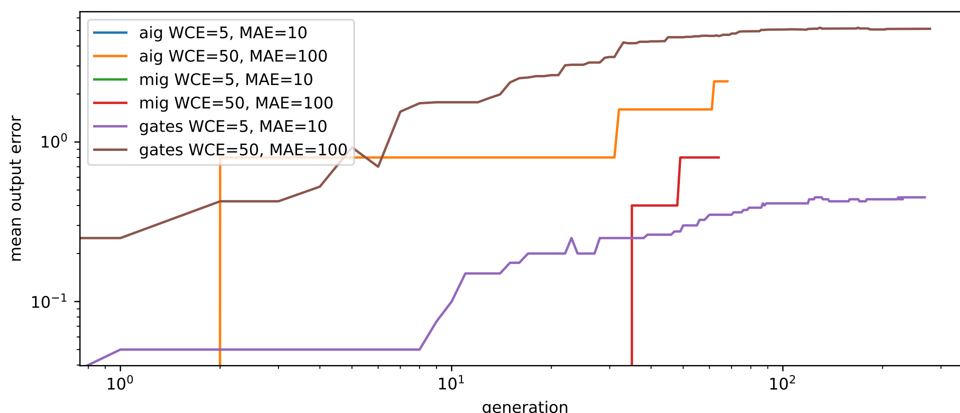
Jedna generace u každé reprezentace trvala jinou časovou dobu, protože má každá jiný počet simulovaných prvků. Z tohoto důvodu každá reprezentace končí na grafu na jiné generaci. Počáteční hodnoty funkce fitness jsou u jednotlivých reprezentací rozdílé. Tento jev je zapříčiněn velkým množstvím hradel xor v optimalizovaném obvodu, které v genomu AIG i MIG musejí být zapsány jako kombinace několika genů narozdíl od reprezentace GATES. Dalším důvodem je, že graf ukazuje hodnoty od výsledků z první generace. Pokud tedy v první generaci došlo k nějaké optimalizaci, na grafu to nebude vidět. Na grafu se dá pozorovat malý optimalizační potenciál reprezentace AIG, která v tomto obvodu provedla malou optimalizaci narozdíl od GATES a MIG narozdíl od čtyřbitové sčítačky, kde AIG nebylo nejpomalejší reprezentací v optimalizaci. Reprezentace GATES v těchto bězích skončila s nejlepším výsledkem, nicméně optimalizovala mnohem pomaleji než MIG a využívala své výhody, kterou je existence xor hradel ve svých typech.

6.3.2 Optimalizace pro povolenou výstupní chybu

Optimalizace s chybou na výstupu byla provedena pomocí parametrů

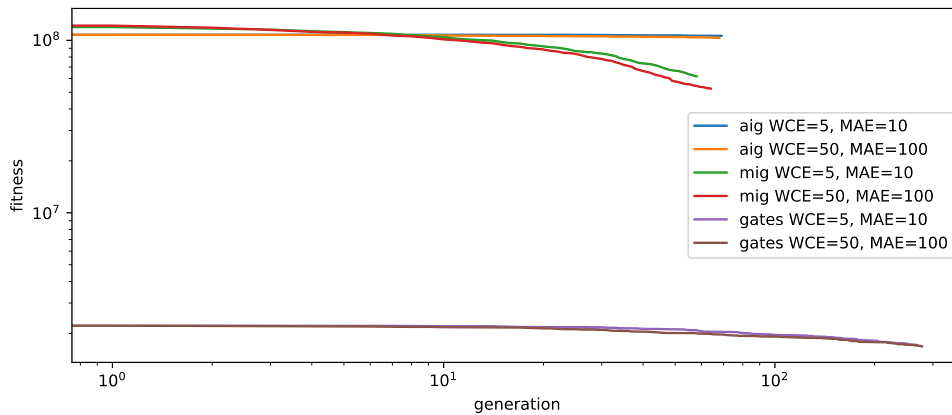
```
-generations=10000 -max_duration=3 -representation={"mig", "aig", "gates"}
-max_abs_error={5,10,20,50,100} -max_one_error={5,10,20,50}
-generation_size=300 -mutations_count=10 -mutations_count_sigma=9
-parents=1 -power_accuracy_ratio=0.75 -profile -selection_size=1
```

Kvůli velkému množství kombinací byly do grafu vneseny pouze ty kombinace, které měly více rozdílné hodnoty od ostatních. Graf s osou X a Y v logaritmickém měřítku na obrázku 6.17 ukazuje průměrnou výstupní chybu v průběhu generací.



Obrázek 6.17: MAE v průběhu generací

Na obrázku 6.17 lze pozorovat, že všechny kombinace běhů nezvládly využít možnost výstupní chyby za účelem vyšší optimalizace. Důvodem je nejspíše velká komplexnost optimalizovaného obvodu. Reprezentace MIG využívá výstupní chyby mnohem méně než ostatní, nicméně jak můžeme vidět na obrázku 6.18, který ukazuje hodnotu fitness funkce, dosahuje tato reprezentace lepších optimalizačních pokroků než reprezentace GATES (GATES je zvýhodněn díky xor). Reprezentace GATES je nejefektivnější v zavádění chyb do výstupů (podle grafu). AIG v tomto případě nedokázalo udělat velké optimalizační pokroky.



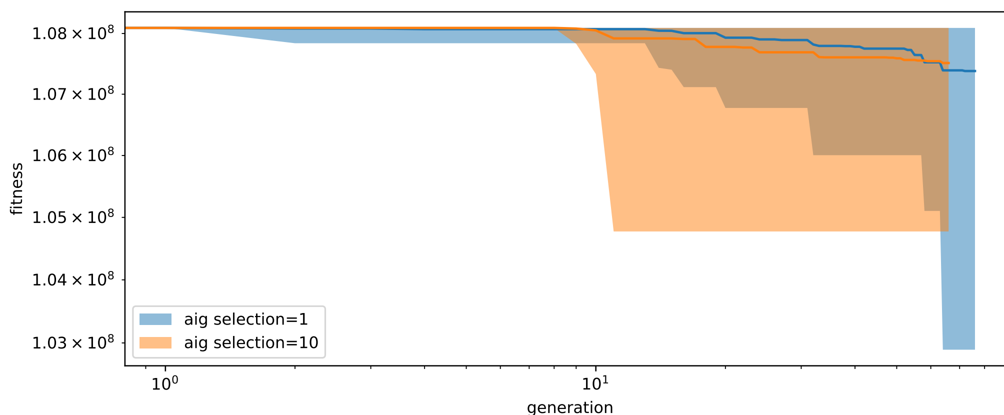
Obrázek 6.18: hodnota funkce fitness v průběhu generací

6.3.3 Parametr velikosti selekce

Test pro zjištění, jak parametr ovlivňuje úspěšnost CGP, byl proveden s parametry

```
-generations=10000 -max_duration=60 -representation="aig" -max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75
-profile -selection_size={1,10} -max_one_error=0
```

Cílem testu bylo zjistit, zda u tohoto obvodu provede změna velikosti selekce nějaké razantní změny. Graf na obrázku 6.19 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.19: hodnota funkce fitness v průběhu generací

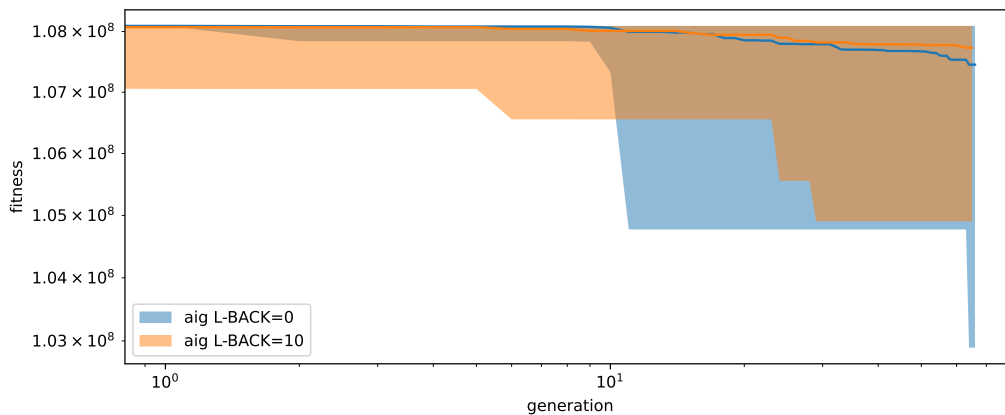
Z grafu na obrázku 6.19 se nepodařilo prokázat, že by velikost selekce měla markantně pozitivní či negativní vliv na hodnotu funkce fitness v průběhu generací. Nicméně drobný pozitivní vliv se dá sledovat.

6.3.4 Parametr l-back

Pro ověření teorie, popsané u parametru l-back u čtyřbitové sčítačky, byl spuštěn test s parametry

```
-generations=10000 -max_duration=60 -representation="aig" -max_abs_error=0
-max_one_error=0 -generation_size=300 -mutations_count=10
-mutations_count_sigma=9 -parents=1 -power_accuracy_ratio=0.75
-profile -selection_size=1 -l-back={0, 10}
```

U tohoto parametru hodnota nula znamená, že nemá být použit, pokud je jeho hodnota vyšší než nula, bude povolena mutace v rozsahu +-hodnota. Graf na obrázku 6.20 ukazuje průměrnou hodnotu a rozsah hodnot funkce fitness pro jednotlivé generace.



Obrázek 6.20: hodnota funkce fitness v průběhu generací (l-back=0 -> unlimited)

Na grafu na obrázku 6.20 je vidět, že parametr l-back opravdu zpomalil optimalizaci, nicméně výsledek byl u tohoto obvodu horší než u varianty bez l-back.

6.4 Závěr z testování

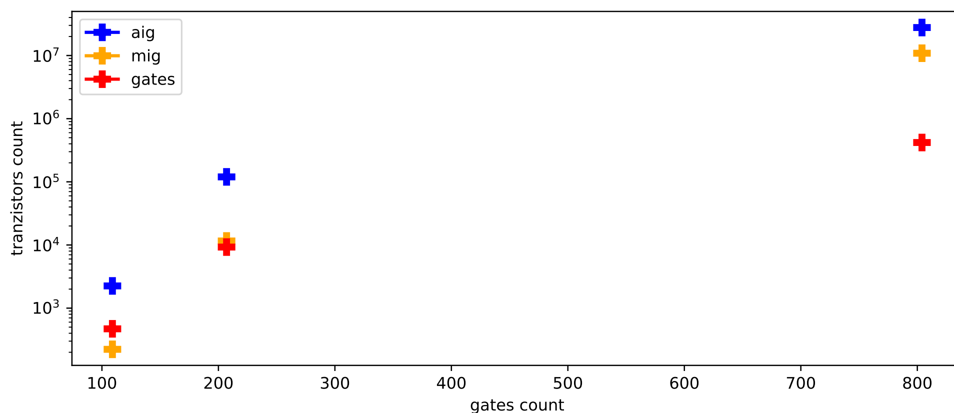
Z testování vyplynulo, že čím je genom větší, tím hůře se provádí optimalizace, což odpovídá předpokladu, který vyplývá z teorie uvedené v kapitole 2. Jako nejefektivnější se ukázala reprezentace MIG, která byla schopna na všech testovaných obvodech provádět největší optimalizační skoky. Vzhledem k vlastnostem jednotlivých reprezentací je vhodné během optimalizace tyto reprezentace střídat a tím částečně jejich problémy vyvažovat. Parametr $l - back$ se ukázal jako vhodný parametr pro zajištění optimalizace pouze v jistém okolí jednotlivých genů, jak je ukázáno v grafech z testování. U operace křížení se nepodařilo prokázat velké zlepšení. U velikosti selekce stejně jako u křížení se nepodařilo prokázat markantní zlepšení. Graf na obrázku 6.21 ukazuje počet tranzistorů jednotlivých obvodů v závislosti na počtu hradel v obvodu při optimalizaci bez chyby. Graf na obrázku 6.22 ukazuje počet tranzistorů jednotlivých obvodů v závislosti na počtu hradel v obvodu při optimalizaci s chybou (nejlepší kombinace maximálního MAE a WCE). V tabulkách 6.1 a 6.2 je procentuální optimalizace obvodu při použití jednotlivých reprezentací. Z těchto dat vyplývá, že rozšíření je schopno provádět relativně velké optimalizace. Tyto grafy a tabulky jsou vytvořeny z dat experimentů, které byly popsány v této kapitole. Optimalizace v těchto experimentech byla časově omezena, proto výsledky v těchto grafech jsou pouze minimální optimalizací z časového pohledu. Pokud by optimalizace probíhala déle, je možné dosáhnout lepších výsledků.

Počet hradel	**GATES**	**MIG**	**AIG**
802	32,644%	75,391%	4,601%
205	71,360%	96,701%	66,509%
107	92,022%	99,746%	98,679%

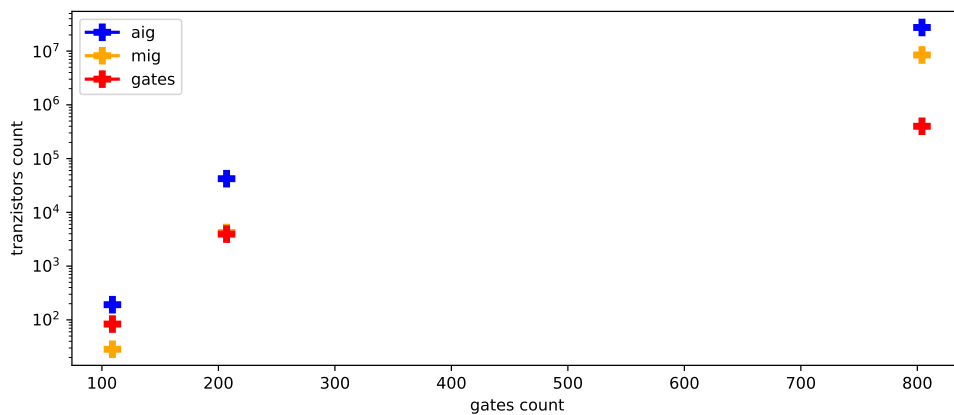
Tabulka 6.1: Procentuální optimalizace obvodů pro jednotlivé reprezentace (s povolenou výstupní chybou)

Počet hradel	**GATES**	**MIG**	**AIG**
802	29,314%	67,938%	3,745%
205	40,687%	92,736%	12,512%
107	80,471%	99,032%	40,687%

Tabulka 6.2: Procentuální optimalizace obvodů pro jednotlivé reprezentace (bez povolené výstupní chyby)



Obrázek 6.21: Počet tranzistorů v závislosti na počtu hradel (bez povolené výstupní chyby)



Obrázek 6.22: Počet tranzistorů v závislosti na počtu hradel (s povolenou výstupní chybou) GATES a MIG jsou překryty v x=205

Kapitola 7

Závěr

Cílem této práce bylo navrhnout a implementovat snadno rozšířitelný optimalizátor kombinačních obvodů, který bude ditribuován jako rozšíření nástroje Yosys. Toto rozšíření by má za úkol optimalizovat kombinační obvody na úkor jejich přesnosti výstupu.

Stěžejní částí projektu bylo implementovat převod mezi RTLIL reprezentací nástroje Yosys a reprezentací, kterou používá CGP, aby rozšíření bylo schopné pracovat s obvody ve vnitřní reprezentaci nástroje Yosys. Dále bylo také důležité implementovat jednotlivé reprezentace co nejvíce modulárně, aby bylo do budoucna zajištěno snadné přidávání dalších reprezentací obvodů pro CGP a tím rozšíření zajistit snadnou rozšířitelnost. Díky tomu je implementované CGP schopné optimalizovat jakoukoliv možnou reprezentací, která bude implementována. Po dokončení implementace, bylo nad rozšířením spuštěno několik testů, které měly za cíl odhalit jeho úspěšnost v optimalizaci a také vhodnost jednotlivých reprezentací obvodů pro různé typy obvodů. Tyto data byla následně analyzována a vnesena do grafů v této práci.

S pomocí tohoto rozšíření může hardwarový návrhář provést optimalizaci, která je schopná do výstupů zanášet drobnou chybu za účelem větší optimalizace tohoto obvodu. Použití CGP také zajišťuje možnost vytvoření opravdu inovativních obvodů, ke kterým se návrhář při použití konvenčních technik nedostane. Takové řešení pak může být přínosem v hardwarových akcelerátorech nebo v nositelné elektronice. Cgploss také může sloužit jako optimalizátor, který bude optimalizovat obvody bez zavádění chyby na výstupech, pokud si návrhář bude přát.

Hlavní nevýhodou zvoleného přístupu optimalizace je jeho časová náročnost, která byla popsána v části 6, a častá rozdílnost výstupu při každém běhu optimalizace. Tato rozdílnost je typickou vlastností většiny genetických algoritmů, ze které plyne jejich problém s nalezením úplného optima [7].

Možných rozšíření práce je celá řada. Vhodné by bylo implementovat paralelizaci a akceleraci simulace obvodů za pomaci GPU. Další možnosti rozšíření se skrývají v části, která řeší různé typy reprezentací obvodů. Bylo by možné do rozšíření přidat RTL reprezentaci či další jiné, jako je XMG nebo XAIG, které byly zmíněny v části 2.4. V neposlední řadě je možné rozšíření doplnit o podporu cyklických orientovaných grafů a tím začít podporovat kromě kombinačních obvodů i obvody sekvenční. Navržený nástroj bude po uveřejnění práce dostupný jako open source aplikace, která bude dostupná v repozitáři na službě Github a bude určene především pro hardwarovou komunitu a vědeckou obec.

Literatura

[1] Alom, M. Z., Taha, T. M., Yakopcic, C., Westberg, S., Sidike, P. et al. The history began from alexnet: A comprehensive survey on deep learning approaches. *ArXiv preprint arXiv:1803.01164*. 2018.

[2] Amarú, L., Gaillardon, P.-E. a Micheli, G. Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization. Červen 2014.

[3] Brummayer, R. a Biere, A. Local two-level and-inverter graph minimization without blowup. *Proc. MEMICS*. 2006, sv. 6, s. 32–38.

[4] Wolf, C. *Yosys Manual* [online]. 2022 [cit. 2022-17-04]. Dostupné z: https://github.com/YosysHQ/yosys-manual-build/releases/download/manual/manual.pdf.

[5] Wolf, C. *Yosys Open SYnthesis Suite* [online]. 2022 [cit. 2022-17-04]. Dostupné z: http://bygone.clairexen.net/yosys/about.html.

[6] Brně, F. informačních technologií VUT v. *VÝPOČETNÍ CLUSTER* [online]. 2022 [cit. 2022-17-04]. Dostupné z: https://www.fit.vut.cz/units/cvt/cluster/.cs.

[7] Luner, P. *Jemný úvod do genetických algoritmů* [online]. 2022 [cit. 2022-17-04]. Dostupné z: https://cgg.mff.cuni.cz/~pepca/prg022/luner.html.

[8] Hashemi, S., Tann, H. a Reda, S. BLASYS: Approximate Logic Synthesis Using Boolean Matrix Factorization. In:. červen 2018, s. 1–6. DOI: 10.1109/DAC.2018.8465702.

[9] Hordějčuk, V. *Genetický algoritmus* [online]. 2022 [cit. 2022-17-04]. Dostupné z: http://voho.eu/wiki/geneticky-algoritmus/.

[10] Husa, J. a Kalkreuth, R. A Comparative Study on Crossover in Cartesian Genetic Programming. In: Castelli, M., Sekanina, L., Zhang, M., Cagnoni, S. a García Sánchez, P., ed. *Genetic Programming*. Cham: Springer International Publishing, 2018, s. 203–219. ISBN 978-3-319-77553-1.

[11] Hynek, J. *Genetické algoritmy a genetické programování*. Grada Publishing as, 2008.

[12] Háleček, I. Logická syntéza s nativní podporou XOR hradel. In: [online]. Brno Kraví Hora: Fakulta informačních technologií, ČVUT, 2016 [cit. 2022-17-04]. PAD 2016. Dostupné z: http://www.fit.vutbr.cz/events/pad2016/download/sbornik/17-Halecek.pdf.

[13] Jan, K. *Generátor aritmetických obvodů* [online]. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vojtěch, M. P. Dostupné z: https://www.fit.vut.cz/study/thesis-file/23300/23300.pdf.

[14] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G. et al. In-datacenter performance analysis of a tensor processing unit. In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, s. 1–12.

[15] Tien Ju Yang, V. S. *Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning* [online]. 2017 [cit. 2022-17-04]. Massachusetts Institute of Technology. Dostupné z: https://arxiv.org/pdf/1611.05128.pdf.

[16] Micheli, Z. C. M. S. Y. X. L. W. G. D. *Structural Rewriting in XOR-Majority Graphs* [online]. 2019 [cit. 2022-17-04]. Dostupné z: https://msoeken.github.io/papers/2019_aspdac.pdf.

[17] Miller., J. F. *CARTESIAN GENETIC PROGRAMMING* [online]. 2018 [cit. 2022-17-04]. Dostupné z: https://www.cartesiangp.com/.

[18] Mrázek, V. *Automated Design Methodology for Approximate Low Power Circuits*. Brno, CZ, 2018. Disertační práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/phd-thesis/841/.

[19] Han, J. a Orshansky, M. *Approximate computing: An emerging paradigm for energy-efficient design* [online]. 2013 [cit. 2022-17-04]. In the 18th IEEE European Test Symposium, pp. 1-6. Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.701.4955&rep=rep1&type=pdf.

[20] Publications, M. *Neural Network Architectures* [online]. 2020 [cit. 2022-17-04]. Dostupné z: https://manningbooks.medium.com/neural-network-architectures-74527000a798.

[21] Sekanina, L., Vašíček, Z. a Mrázek, V. Automated Search-Based Functional Approximation for Digital Circuits. In: *Approximate Circuits - Methodologies and CAD*. Springer International Publishing, 2019, s. 175–203. DOI: 10.1007/978-3-319-99322-5_9. ISBN 978-3-319-99322-5. Dostupné z: https://www.fit.vut.cz/research/publication/11679.

[22] Sekanina, L. *Evoluční hardware: od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. 1. vyd. Academia, 2009. ISBN 978-80-200-1729-1.

[23] Synthesis, B. L. a Group, V. *ABC: A System for Sequential Synthesis and Verification* [online]. 2022 [cit. 2022-17-04]. Dostupné z: https://people.eecs.berkeley.edu/~alanmi/abc/.

[24] Free Software Foundation, I. *Using Vector Instructions through Built-in Functions* [online]. 2022 [cit. 2022-17-04]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html.

[25] Hordějčuk, V. *Logické hradlo* [online]. 2022 [cit. 2022-17-04]. Dostupné z: http://voho.eu/wiki/logicke-hradlo/.

[26] Vrajitoru, D. Crossover improvement for the genetic algorithm in information retrieval. *Information processing & management.* Elsevier. 1998, sv. 34, č. 4, s. 405–415.

Příloha A

Obsah přiloženého paměťového média

- source_codes – adresář obsahující zdrojové kódy, výsledky testů a pomocné skripty pro zpracování výsledků testů
  - yosys-cgploss – adresář obsahující zdrojové kódy rozšíření cgploss
  - experiments - adresář obsahující výsledky experimentů včetně pomocných skriptů pro generování nových testů a zpracování výsledků testů
  - README.md - manuál obsahující popis kompilace rozšíření cgploss a ukázka jeho základního použití
- bachelor's_thesis – adresář obsahující zdrojové kódy písemné zprávy v LATEXu a její verzi ve formátu pdf

```
.
├── include
│   ├── aig.h
│   ├── config-parse.h
│   ├── convert.cpp
│   ├── gates.h
│   ├── generation.h
│   ├── genome.h
│   ├── mig.h
│   ├── representation.h
│   └── simulation.h
├── LICENSE
├── Makefile
├── README.md
├── src
│   ├── aig-genome.cpp
│   ├── aig-rtlil.cpp
│   ├── aig-sim.cpp
│   ├── config-parse.cpp
│   ├── gates-genome.cpp
│   ├── gates-rtlil.cpp
│   ├── gates-sim.cpp
│   ├── generation.cpp
│   ├── genome.cpp
│   ├── main.cpp
│   ├── mig-genome.cpp
│   ├── mig-rtlil.cpp
│   ├── mig-sim.cpp
│   └── simulation.cpp
├── tests
└── yosys
```

Obrázek A.1: Obsah složky obsahující rozšíření cgploss