VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

GENERÁTOR ARITMETICKÝCH OBVODŮ

ARITHMETIC CIRCUIT GENERATOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN KLHŮFEK**
AUTHOR

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

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Klhůfek Jan**
Program: Informační technologie
Název: **Generátor aritmetických obvodů**
A Generator of Arithmetic Circuits
Kategorie: Návrh číslicových systémů

Zadání:

1. Seznamte se s hardwarovou implementací aritmetických obvodů (bezznaménkové i znaménkové sčítačky, násobičky a děličky).
2. Seznamte se s možnostmi reprezentace obvodů (VHDL, BLIF, ...)
3. Zpracujte studii na výše uvedená témata.
4. Navrhněte nástroj pro generování výše uvedených aritmetických obvodů na různých úrovních abstrakce zápisu s možností exportu do různých reprezentací.
5. Navržený nástroj implementujte.
6. Vyhodnoťte parametry navržených obvodů a diskutujte možná rozšíření 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 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem této bakalářské práce je představení návrhu a implementace generátoru aritmetických obvodů v jazyce Python umožňující export těchto obvodů do různých reprezentací popisu v plochých a hierarchických podobách. Práce se nejprve věnuje specifikaci HW struktur jednotlivých typů aritmetických obvodů a způsobům popisu těchto struktur do různých reprezentací. Následuje představení koncepce a implementace nástroje zvaného ArithsGen schopného generovat aritmetické obvody a exportovat je do různých reprezentací popisu. Výstupní reprezentace pak slouží ke snadné simulaci funkčnosti navržených obvodů (C), k popisu hardwaru a logické syntéze (Verilog), k formální verifikaci (BLIF) či ke globální optimalizaci obvodů s využitím evoluční strategie (CGP). V závěru byly generované reprezentace jednotlivě otestovány a s využitím logické syntézy vzájemně porovnány.

Abstract

The aim of this bachelor thesis is to present the design and implementation of an arithmetic circuit generator. The generator focuses on generating various output representations of arithmetic circuits in flattened and hierarchical forms using the Python programming language. The work first deals with the specification of HW structures of individual arithmetic circuits and the corresponding ways of describing these structures into various representations. Followed by an introduction to the concept and details of the implementation of a tool called ArithsGen, which is able to generate arithmetic circuits and export them to various output representations. The output representations are then used for fast and simple simulation of the designed circuits (C), to describe the hardware structures and perform logic synthesis (Verilog), to formal verify the designs (BLIF) or to globally optimize the circuits using the evolutionary strategy (CGP). Finally, the generated representations were individually tested and compared with each other using the results obtained from logic synthesis.

Klíčová slova

aritmetický obvod, generátor, logické hradlo, sčítačka, odčítačka, násobička, dělička, plochý popis, hierarchický popis, Python, C, Verilog, BLIF, Kartézské Genetické Programování (CGP)

Keywords

arithmetic circuit, generator, logic gate, adder, subtractor, multiplier, divider, flat design, hierarchical design, Python, C, Verilog, BLIF, Cartesian Genetic Programming (CGP)

Citace

KLHŮFEK, Jan. *Generátor aritmetických obvodů*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vojtěch Mrázek, Ph.D.

Generátor aritmetických obvodů

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.

.....
Jan Klhůfek
10. května 2021

Poděkování

Rád bych tímto poděkoval svému vedoucímu Ing. Vojtěchovi Mrázkovi, Ph.D. za čas strávený při konzultacích i jeho odborné vedení, rady a nápady, jež přispěly ke zhotovení této bakalářské práce.

Obsah

Kapitola 1

Úvod

V téměř všech moderních digitálních systémech najdeme aritmetické obvody řešící základní matematické operace jako je sčítání, odčítání, násobení a dělení. Aritmetické obvody jsou totiž nedílnou součástí dnešních procesorů. Konkrétně je zde najdeme uvnitř aritmeticko-logické jednotky (ALU), jednotky pro generování adres (AGU) či v matematickém koprocesoru (FPU). Mimo klasický procesor se ale taktéž ve velkém množství nachází v systolických polích, grafických akcelerátorech a akcelerátorech neuronových sítí pro paralelní rychlé výpočty. Hardwarové řešení aritmetických obvodů umožňuje vykonávat kýžené aritmetické operace mnohonásobně rychleji než softwarové řešení.

Vytváření popisu vnitřní struktury složitějších obvodů bez pomoci generátorů či generujících struktur je pro člověka náročnou a u větších obvodů takřka nereálnou disciplínou. Generování a export aritmetických obvodů značně šetří čas a lidské zdroje potřebné k vytvoření a odladění jejich popisu. Generátor je v této záležitosti nejen mnohem rychlejší, ale navíc není náchylný k zanesení chyb z nepozornosti či překlepů.

Pojem aritmetické obvody tedy souhrnně označuje hardwarové komponenty, které jsou schopné realizace některé z aritmetických operací. Patří sem zejména obvody sčítaček, odčítaček, násobiček a děliček. Pro každý ze zmíněných typů aritmetických obvodů existuje řada odlišných architektur lišících se svou vnitřní strukturou, tzn. způsobem poskládání a propojení dílčích hradel. Bližšímu popisu HW struktury aritmetických obvodů se věnuje kapitola 2.

Jednotlivé architektury sice provádějí stejnou operaci, např. sčítání, ale přitom se vůči ostatním liší v různorodých parametrech, z nichž nejsledovanějšími jsou příkon, rychlost a plocha. Právě tyto tři klíčové parametry slouží k porovnání různých architektur a ke zvolení takové, která k danému účelu použití nejvíce vyhovuje. K získání a porovnání těchto a dalších parametrů obvodů pak slouží různé reprezentace popisu jejich vnitřní struktury. Ty jsou spolu se svými specifickými účely popsány v kapitole 3.

Kapitola 4 popisuje existující řešení na danou problematiku a představuje nový nástroj zvaný ArithsGen, který umožňuje navrhovat libovolné architektury aritmetických obvodů. Je psaný v jazyce Python a s výhodou využívá principů objektového programování. Implementace je založena na RTL (*Register-transfer level*) úrovni abstrakce, tedy popisu na úrovni meziregistrových přenosů a používá dvouvstupá logická hradla. Hlavní princip generování obvodů je postaven na jejich modularitě.

Následuje kapitola 5 věnována testování a experimentování. Testování se zabývá správností popisu jednotlivých reprezentací generovaných nástrojem ArithsGen. Experimentování se pak věnuje získávání a vzájemnému porovnávání parametrů mezi rozličnými architekturami aritmetických obvodů.

Kapitola 2

Struktura aritmetických obvodů

Aritmetické obvody jsou kombinační hardwarové součástky[1] strukturovaně složené z menších komponent, z nichž nejzákladnějšími jsou logická hradla. Kaskádovitým propojováním logických hradel, jakožto nejmenších stavebních jednotek, se tvoří popisy složitějších jednobitových obvodů, jenž dále představují buďto samostatné obvody, nebo dílčí komponenty pro popis složitějších vícebitových obvodů.

Samostatná logická hradla jsou podrobně popsána v podkapitole 2.1, v podkapitole 2.2 pak následuje popis jednobitových komponent a nakonec v podkapitole 2.3 přichází popis vícebitových aritmetických obvodů.

2.1 Logická hradla

Logická hradla představují logicky nejmenší stavební bloky pro stavbu integrovaných digitálních obvodů. Jejich funkční princip je postaven na Booleově algebře, kdy se kombinací vstupních hodnot získává odpovídající výstupní hodnota. Booleově algebře a jejímu využití se věnuje například bakalářská práce [1]. Samotná kombinace vstupů je pak interně reprezentována logickou funkcí příslušející danému typu hradla. Blíže představeny jsou pouze dvouvstupá a jednovstupá logická hradla, jelikož vícevstupá se liší pouze složitější kombinací více vstupních hodnot pro tytéž logické funkce. Vícevstupá logická hradla navíc čelí problému známému jako *fan-in*, kdy se funkčnost hradla zpomaluje se zvyšujícím počtem vstupů. Větší problém však představuje *fan-out*, který je společný pro všechna hradla bez ohledu na počet jejich vstupů. *Fan-out* značí počet vstupních vodičů dalších hradel, na které je výstupní vodič hradla připojen. To ovlivňuje kapacitní zátěž hradla, jelikož kapacity připojených vstupů se sčítají. Vysoká zátěž se pak negativním způsobem odráží na rychlosti daného hradla. K řešení tohoto problému se vytvářejí hradla s posílenými koncovými prvky. Principu *fan-in*, *fan-out* a jejich dopadu na zpoždění digitálních obvodů se pak věnuje například článek [2].

2.1.1 Dvouvstupá logická hradla

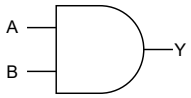
Elementárními dvouvstupými hradly jsou AND, OR, XOR a jejich komplementární členy NAND, NOR, XNOR. Kombinací hradel AND, OR a jednovstupých NOT hradel je možno popsat logickou funkci ostatních typů logických členů. Samotné NAND a NOR členy pak

[1] Obvody jejichž výstupní hodnoty jsou závislé pouze na jejich vstupních hodnotách.

představují tzv. univerzální hradla[2] a jsou samostatně funkčně úplná, což znamená, že pomocí nich lze popsat jakoukoliv Booleovskou funkci bez potřeby jiných typů hradel. Následují pravdivostní tabulky a symboly reprezentující jednotlivá dvouvstupá hradla.

Tabulka 2.1: Pravdivostní tabulka AND hradla.

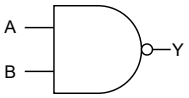
Vstupy		**Výstup**
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



Obrázek 2.1: Symbol hradla AND v normě ANSI.

Tabulka 2.2: Pravdivostní tabulka NAND hradla.

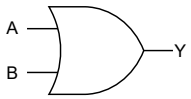
Vstupy		**Výstup**
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0



Obrázek 2.2: Symbol hradla NAND v normě ANSI.

Tabulka 2.3: Pravdivostní tabulka OR hradla.

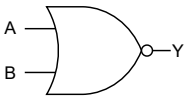
Vstupy		**Výstup**
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1



Obrázek 2.3: Symbol hradla OR v normě ANSI.

Tabulka 2.4: Pravdivostní tabulka NOR hradla.

Vstupy		**Výstup**
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

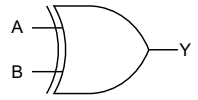


Obrázek 2.4: Symbol hradla NOR v normě ANSI.

[2]Univerzální hradla: https://www.electronics-tutorials.ws/logic/universal-gates.html

Tabulka 2.5: Pravdivostní tabulka XOR hradla.

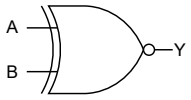
Vstupy		Výstup
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0



Obrázek 2.5: Symbol hradla XOR v normě ANSI.

Tabulka 2.6: Pravdivostní tabulka XNOR hradla.

Vstupy		Výstup
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1



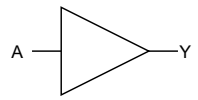
Obrázek 2.6: Symbol hradla XNOR v normě ANSI.

2.1.2 Jednovstupá logická hradla

Mezi jednovstupá hradla patří opakovač a invertor. Zejména invertor hraje důležitou roli ve stavbě ostatních logických hradel a složitějších číslicových obvodů. Následují pravdivostní tabulky a symboly reprezentující zmíněná jednovstupá hradla.

Tabulka 2.7: Pravdivostní tabulka hradla opakovače.

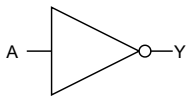
Vstup	Výstup
A	Y
0	0
1	1



Obrázek 2.7: Symbol opakovače v normě ANSI.

Tabulka 2.8: Pravdivostní tabulka NOT hradla.

Vstup	Výstup
A	Y
0	1
1	0



Obrázek 2.8: Symbol NOT hradla v normě ANSI.

2.2 Jednobitové obvody

Jednobitové kombinační obvody představují součástky, jenž mají určitý počet jednobitových vstupů a výstupů, a které realizují patřičnou bitovou operaci jako je např. sčítání či odčítání. Mohou být složené z menších logických hradel, přičemž hodnoty na výstupních vodičích jsou získány z postupné kombinace vstupních hodnot, nebo jsou popsány přímo pomocí CMOS[3] logiky. Příslušnou funkci obvodu lze popsat odpovídajícím Booleovským výrazem.

[3]Způsob vytváření logických členů s pomocí komplementárních párů MOSFET (PMOS a NMOS) tranzistorů.

Pravdivostní tabulka pak obsahuje výčet všech kombinací vstupních proměnných a jim odpovídajících výstupních hodnot. Více informací k jednobitovým kombinačním obvodům k dočtení např. v knize [3].

Poloviční 2.2.1 a úplné 2.2.2 sčítačky tvoří základní stavební kameny většiny vícebitových aritmetických obvodů. Poloviční 2.2.3 a úplné 2.2.4 odčítačky najdou své využití zejména při návrhu vícebitových děliček a odčítaček. Dvouvstupé multiplexory 2.2.5 se pak využívají např. u sčítaček s přeskočením přenosu 2.3.1.

2.2.1 Poloviční sčítačka

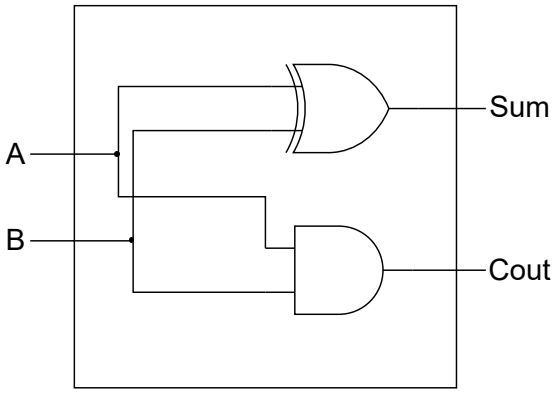
Dvouvstupá poloviční sčítačka se vstupy A a B realizuje funkci jednobitového součtu a výsledek operace vrací ve dvou jednobitových výstupech – v součtu Sum vstupních hodnot a příznaku přenosu $Cout$. Rovnice pro výpočet součtu je (2.1) a pro výpočet příznaku přenosu (2.2). Pravdivostní tabulka 2.9 ukazuje kombinace vstupních hodnot a jejich odpovídajících výstupních hodnot. Obrázek 2.9 ukazuje strukturu zapojení poloviční sčítačky a obrázek 2.10 její blokový diagram.

$$Sum = A \oplus B \tag{2.1}$$

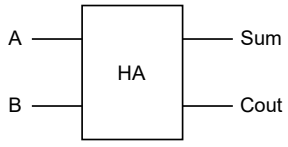
$$Cout = A \cdot B \tag{2.2}$$

Tabulka 2.9: Pravdivostní tabulka poloviční sčítačky.

Vstupy		**Výstupy**	
A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Obrázek 2.9: Vnitřní zapojení hradel tvořící poloviční sčítačku.



Obrázek 2.10: Blokový diagram poloviční sčítačky.

2.2.2 Úplná sčítačka

Úplná sčítačka obsahuje stejně jako poloviční sčítačka vstupy A, B, ale navíc rozšiřuje její logiku o třetí vstupní hodnotu Cin. Ta reprezentuje přenesení příznaku přenosu z bitového

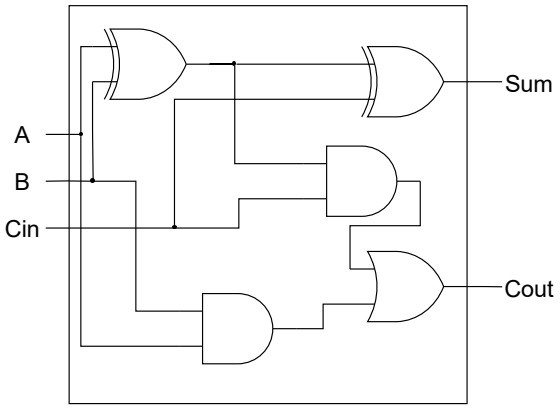
součtu předcházejícího řádu. Obvod taktéž realizuje funkci jednobitového součtu a výsledek operace vrací ve dvou jednobitových výstupech. Výpočet výstupů pro úplnou sčítačku obsahují rovnice (2.3) pro součet *Sum* a (2.4) pro příznak přenosu *Cout*. Pravdivostní tabulka s možnými kombinacemi vstupů viz 2.10. Vzhledem ke třem vstupním vodičům je vnitřní struktura obvodu složitější než u poloviční sčítačky, jak ukazuje obrázek 2.11. Obrázek 2.12 pak obsahuje blokový diagram identifikující úplnou sčítačku.

$$Sum = (A \oplus B) \oplus Cin \tag{2.3}$$

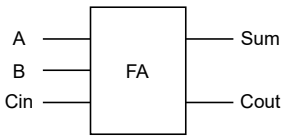
$$Cout = (A \cdot B) + ((A \oplus B) \cdot Cin) \tag{2.4}$$

Tabulka 2.10: Pravdivostní tabulka úplné sčítačky.

Vstupy			**Výstupy**	
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



Obrázek 2.11: Vnitřní zapojení hradel tvořící úplnou sčítačku.



Obrázek 2.12: Blokový diagram úplné sčítačky.

2.2.3 Poloviční odčítačka

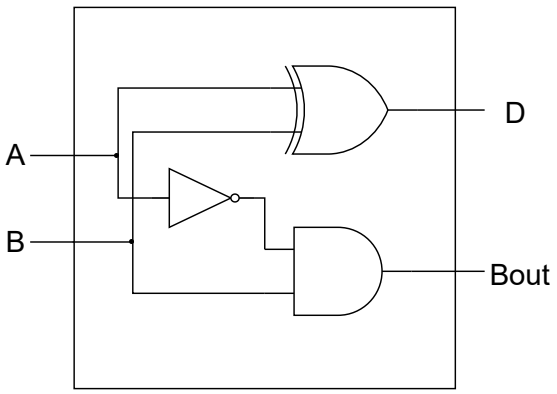
Dvouvstupá poloviční odčítačka se vstupy A a B realizuje funkci jednobitového rozdílu a výsledek operace vrací ve dvou jednobitových výstupech – v rozdílu D (*Difference*) vstupních hodnot a bitu příznaku výpůjčky z vyššího řádu *Bout*, který je roven 1 v případě, že je hodnota menšence A menší než hodnota menšitele B. Pro přehled všech možných kombinací hodnot viz pravdivostní tabulka 2.11. Vztah pro výpočet rozdílu je popsán rovnicí (2.5) a pro výpočet příznaku výpůjčky rovnicí (2.6). Obrázek 2.13 obsahuje strukturu zapojení poloviční odčítačky, zatímco obrázek 2.14 obsahuje její odpovídající blokový diagram.

$$D = A \oplus B \tag{2.5}$$

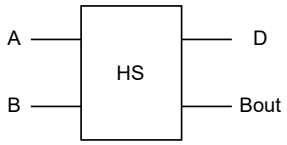
$$Bout = \overline{A} \cdot B \tag{2.6}$$

Tabulka 2.11: Pravdivostní tabulka poloviční odčítačky.

Vstupy		**Výstupy**	
A	B	D	Bout
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0



Obrázek 2.13: Vnitřní zapojení hradel tvořící poloviční odčítačku.



Obrázek 2.14: Blokový diagram poloviční odčítačky.

2.2.4 Úplná odčítačka

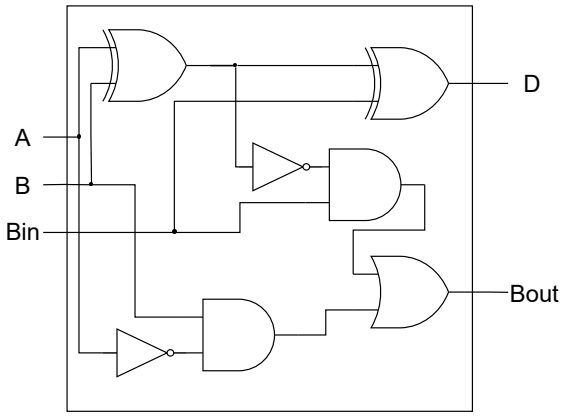
Úplná odčítačka provádí operaci odčítání kombinací vstupních hodnot A, B podobně jako poloviční odčítačka, ale navíc uvažuje i situaci přenosu příznaku výpůjčky Bin z předcházejícího nižšího řádu. Výsledek operace úplné odčítačky je uchován ve dvou jednobitových výstupech, v bitu rozdílu D (*Difference*) a bitu příznaku výpůjčky $Bout$, který je roven 1 v případě, že je hodnota menšence A menší než hodnota menšitele B nebo, když jsou si hodnoty menšence a menšitele rovny a zároveň je hodnota Bin rovna logické 1, jak vychází z pravdivostní tabulky 2.12. Výpočet D popisuje rovnice (2.7) a výpočet $Bout$ rovnice (2.8). Obrázek 2.15 popisuje vnitřní strukturu úplné odčítačky, zatímco obrázek 2.16 obsahuje její odpovídající blokový diagram.

$$D = (A \oplus B) \oplus Bin \tag{2.7}$$

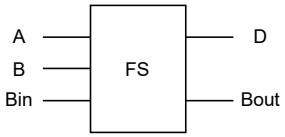
$$Bout = (\overline{A} \cdot B) + (\overline{(A \oplus B)} \cdot Bin) \tag{2.8}$$

Tabulka 2.12: Pravdivostní tabulka úplné odčítačky.

Vstupy			**Výstupy**	
A	B	Bin	D	Bout
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



Obrázek 2.15: Vnitřní zapojení hradel tvořící úplnou odčítačku.



Obrázek 2.16: Blokový diagram úplné odčítačky.

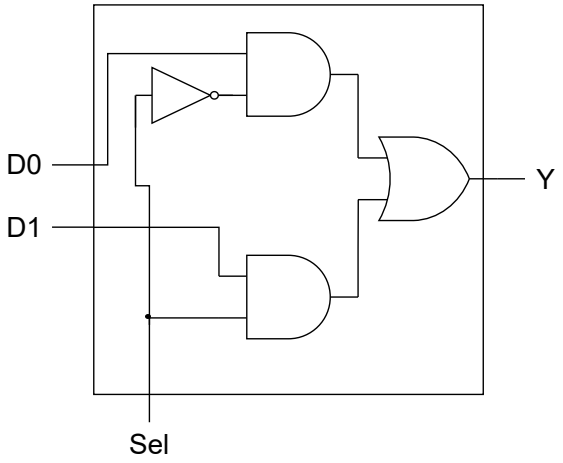
2.2.5 2:1 Multiplexor

Dvouvstupý multiplexor se vstupy $D0$ a $D1$ a jedním řídicím signálem Sel realizuje výběr připojení jednoho ze vstupních vodičů k výstupnímu vodiči Y. V případě logické hodnoty 1 signálu Sel je na výstup připojen vstup $D1$, v opačném hodnotě vstup $D0$, jak je vidět v tabulce 2.13. Vztah mezi vstupy a řídicím signálem popisuje rovnice 2.9. Na obrázku 2.17 lze vidět vnitřní zapojení pro dvouvstupý multiplexor a na obrázku 2.18 odpovídající blokový diagram. Pro podrobnější popis dvouvstupého multiplexoru viz [4].

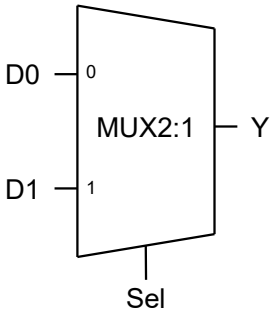
$$D = (D1 \cdot Sel) + (D0 \cdot \overline{Sel}) \tag{2.9}$$

Tabulka 2.13: Pravdivostní tabulka dvouvstupého multiplexoru.

Vstupy			**Výstup**
Sel	D0	D1	Y
0	0	–	0
0	1	–	1
1	–	0	0
1	–	1	1



Obrázek 2.17: Vnitřní zapojení hradel tvořící dvouvstupý multiplexor.



Obrázek 2.18: Blokový diagram dvouvstupého multiplexoru.

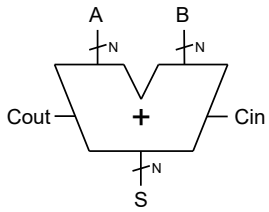
2.3 Vícebitové obvody

Mezi vícebitové aritmetické obvody se řadí obvody sčítaček 2.3.1, odčítaček 2.3.2, násobiček 2.3.3 a děliček 2.3.4. Jedním z cílů při návrhu aritmetických obvodů je zajištění co nejmenšího zpoždění *propagation delay*, tedy doby potřebné k propagaci změn vstupů k výstupům. Pro každý ze zmíněných typů aritmetických obvodů existuje řada různých architektur lišící se svou vnitřní strukturou, tzn. složením z menších funkčních bloků a celkovým poskládáním a propojením jednotlivých hradel. *Critical path* pak označuje cestu mezi vstupy a výstupy s největším možných *propagation delay* a představuje jeden z faktorů pro porovnávání různých architektur obvodů. Různé obvody tak sice mohou provádět stejnou operaci, např. sčítání, ale přitom se lišit příkonem, plochou nebo právě zpožděním. Zejména tyto tři klíčové parametry pak slouží k porovnání rozličných architektur a ke zvolení takové, která k danému účelu použití nejvíce vyhovuje. V následujících sekcích si podrobněji představíme jednotlivé typy aritmetických obvodů a u každého si popíšeme některé jejich představitele. Mimo popisované kombinační aritmetické obvody existují také sekvenční aritmetické obvody, jejichž výstupy jsou dány současnými vstupními hodnotami, ale i těmi minulými. Zastupují je například sekvenční násobičky nebo sekvenční děličky. Jimi se zde však zabývat nebudeme. Přesnější specifikace spolu s dalšími architekturami jsou k nalezení například v literatuře [5].

2.3.1 Sčítačky

Vícebitové sčítačky realizují operaci součtu mezi dvojicemi bitů ze vstupních sběrnic A, B o bitových šířkách N. Počítají i s přenesením příznaku přenosu z bitového součtu předcházejícího řádu Cin. Jednotlivé bitové součty Sum spolu s posledním příznakem přenosu $Cout$ v sobě nesou výsledek operace. Sčítání může být prováděno znaménkově či bezznaménkově, v této sekci budou uvažovány pouze bezznaménkové sčítačky. V případě znaménkového sčítání pak obvod zároveň plní i funkci odčítačky. O odčítačkách a specifikaci znaménkových sčítaček více k dočtení v sekci 2.3.2. *Propagation delay* je u obvodů sčítaček většinou závislý na výpočtu posledního výstupního bitu $Cout$. Blokový diagram reprezentující vícebitové

sčítačky viz obrázek 2.19. Různý přístup k interní realizaci dává za vznik nepřebernému množství architektur, jejichž zástupce si nyní představíme.

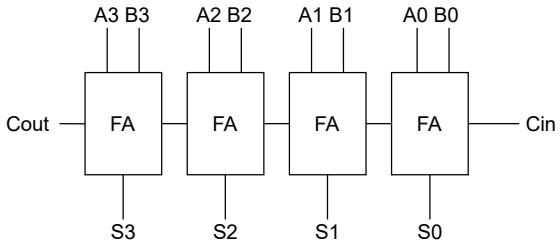


Obrázek 2.19: Blokový diagram vícebitové sčítačky.

Sčítačka s postupným přenosem

Sčítačka s postupným přenosem (*RCA*) představuje nejběžnější architekturu vícebitové sčítačky. Charakterizuje ji kaskádování jednobitových úplných sčítaček, jejichž počet je roven šířce vstupních sběrnic. Pro ukázku struktury 4 bitové sčítačky typu RCA viz obrázek 2.20.

V případě, že je hodnota vstupního bitu *Cin* konstantní logická 0, může být první úplná sčítačka nahrazena za poloviční sčítačku.

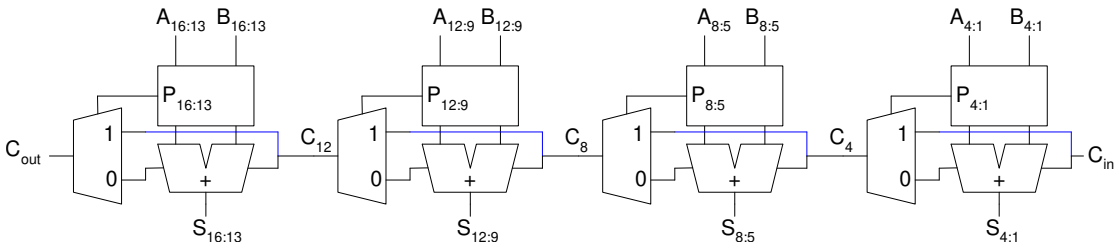


Obrázek 2.20: Struktura 4 bitové sčítačky s postupným přenosem.

Sčítačka s přeskočením přenosu

Sčítačka s přeskočením přenosu (*CSkA*) snižuje v závislosti na vstupních hodnotách zpoždění potřebné k výpočtu posledního výstupního bitu *Cout*. Její logika spočívá v rozdělení vstupní bitové šířky do menších funkčních podbloků fixních bitových šířek. Uvnitř jednotlivých bloků se paralelně určí zda má být vstupní bit přenosu pro daný blok připojen na jeho výstup a nebo je třeba výstupní bit dopočítat postupnými bitovými součty. Určení je dáno kombinací P (*propagate*) signálů vstupních dvojic bitů bloku k získání hodnoty sloužící jako řídicí signál pro dvouvstupý multiplexor, který realizuje výběr propojení patřičné hodnoty na výstup. Samotné *propagate* signály indikují zda bitová dvojice šíří (propaguje) vstupní příznak přenosu na výstup. Výpočet *propagate* signálu popisuje rovnice (2.10). Příklad zapojení 16 bitového obvodu CSkA sčítačky s využitím 4 bitových bloků je k vidění na obrázku 2.21.

$$P_i = A_i \oplus B_i, \quad kde \quad i \in \{0, \ldots, N\} \tag{2.10}$$



Obrázek 2.21: Struktura 16 bitové sčítačky s přeskočením přenosu [5].

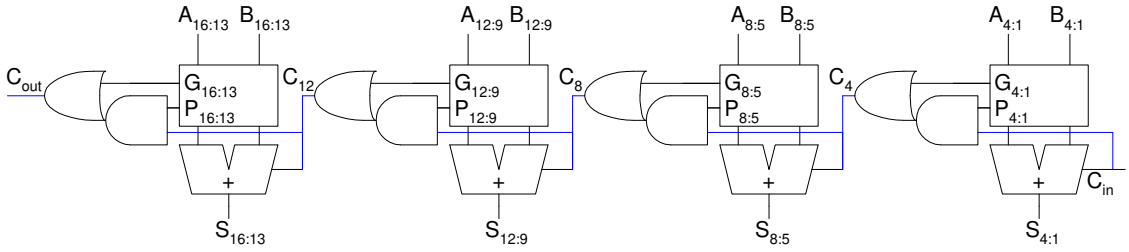
Sčítačka s predikcí přenosu

Sčítačka s predikcí přenosu (*CLA*) představuje typ tzv. *rychlé sčítačky*, která se podobně jako CSkA skládá z menších podbloků, ale navíc pro jednotlivé bitové dvojice vypočítává spolu s P (*propagate*) signály i jejich G (*generate*) signály. Hodnota signálu G indikuje, zda bitová dvojice generuje výstupní příznak přenosu. Jednotlivé paralelní předvypočítávání představuje značné zrychlení u problematického *propagation delay*. Výpočet *propagate/generate* signálů je možno realizovat s využitím vícevstupých hradel, čímž se zmenšuje zabíraná plocha, ale může nastat již dříve zmiňovaný problém *fan-in*. Hradla s mnoha vstupy u CLA sčítaček vyšších bitových šířek negativním způsobem ovlivňují rychlost výpočtu jednotlivých P/G hodnot potřebných k získání výstupního *Cout*. Právě z tohoto důvodu se využívá menších podbloků s optimální bitovou šířkou 4, jež interně využívají nejvýše 4-vstupá logická hradla. Syntézní nástroj však pak může toto zapojení upravit s ohledem na vlastnosti dané výrobní technologie. Druhou možností je nahradit vícevstupá hradla stromovou strukturou složenou z kaskády dvouvstupých hradel, kdy se sice problém *fan-in* eliminuje, ale celková plocha obvodu a náročnost na propojení jeho vodičů roste z důsledku vyššího počtu logických členů. Lze volit kompromisy mezi oběma přístupy, čímž dochází ke vzniku dalších architektur. Výpočet P, G signálů popisují rovnice (2.11), respektive (2.12). Rovnice (2.13) pak obsahuje odvození pro paralelní získání jednotlivých příznaků přenosu. Obrázek 2.22 obsahuje strukturu zapojení 16 bitové CLA sčítačky s využitím 4 bitových podbloků.

$$P_i = A_i \oplus B_i, \quad kde \quad i \in \{0, \ldots, N\} \tag{2.11}$$

$$G_i = A_i \cdot B_i, \quad kde \quad i \in \{0, \ldots, N\} \tag{2.12}$$

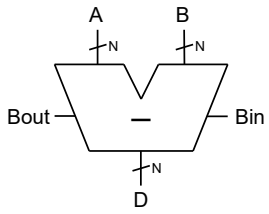
$$C_{i+1} = G_i + (P_i \cdot C_i) \tag{2.13}$$



Obrázek 2.22: Struktura 16 bitové sčítačky s predikcí přenosu [5].

2.3.2 Odčítačky

Vícebitové odčítačky realizují operaci rozdílu mezi dvojicemi bitů ze vstupních sběrnic A, B o bitových šířkách N a navíc uvažují i s případným příznakem výpůjčky z předchozího řádu Bin. Jednotlivé bitové rozdíly D spolu s posledním příznakem výpůjčky bitu z vyššího řádu $Bout$ v sobě uchovávají výsledek operace. Operaci odčítání lze realizovat i v podobě znaménkového sčítání. V takovém případě dochází k úpravě architektur bezznaménkových sčítaček k zajištění správné znaménkovosti výstupu. Toho je možno docílit patřičnou úpravou architektur buďto negováním jejich vstupních hodnot nebo zajištěním znaménkového rozšíření výsledného součtu. Blokový diagram reprezentující vícebitové odčítačky viz obrázek 2.23. Následuje seznámení s paralelní odčítačkou, která představuje zástupce vícebitových odčítaček. Podrobnější popis úpravy bezznaménkových sčítaček do podoby znaménkových a další možné architektury odčítaček jsou k nalezení například v knize [6].

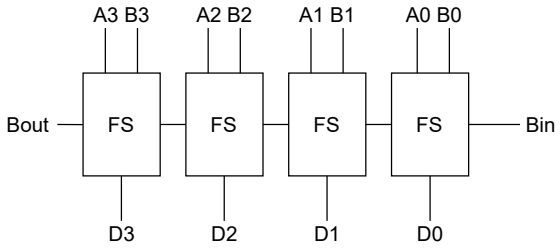


Obrázek 2.23: Blokový diagram vícebitové odčítačky.

Paralelní odčítačka

Paralelní odčítačka sdílí podobné vlastnosti týkající se sledovaného zpoždění a strukturu zapojení jako RCA sčítačka, jen ke své realizaci používá bloky jednobitových odčítaček místo sčítaček. Struktura zapojení 4 bitové paralelní odčítačky je k prohlédnutí na obrázku 2.24.

V případě, že je hodnota vstupního bitu Bin rovna logické 0, může být první úplná odčítačka nahrazena za poloviční odčítačku.

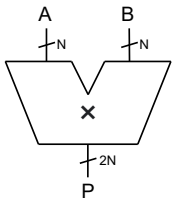


Obrázek 2.24: Struktura 4 bitové paralelní odčítačky.

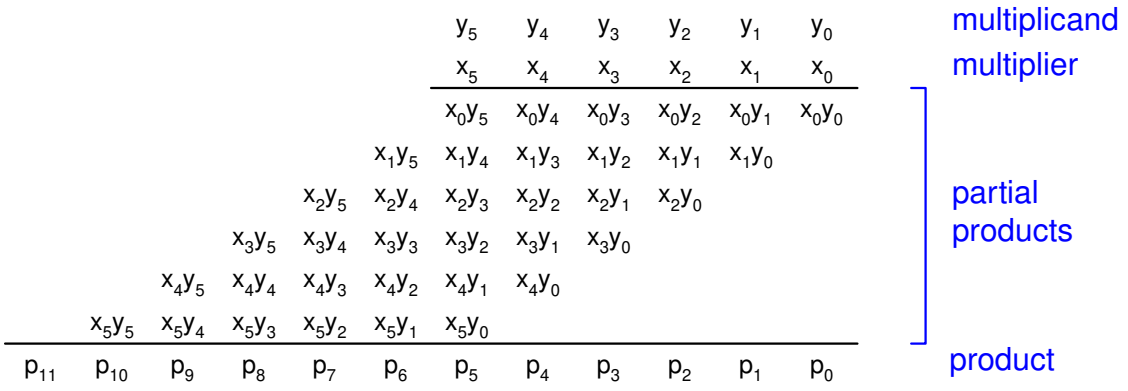
2.3.3 Násobičky

Vícebitové násobičky realizují operaci součinu mezi dvojicemi bitů ze vstupních sběrnic A, B o bitových šířkách N formováním částečných součinů P (*partial products*), jež jsou na konci sečteny k obdržení výsledného součinu. Různé architektury násobiček v sobě využívají obvody vícebitových sčítaček k finální sumaci částečných součinů. Jednotlivé součiny vstupních dvojic bitů jsou realizovány pomocí AND hradel. Blokový diagram reprezentu-

jící vícebitové násobičky viz obrázek 2.25. Proces vícebitového násobení je znázorněn na obrázku 2.26. Dále si představíme jejich zástupce.



Obrázek 2.25: Blokový diagram vícebitové násobičky.

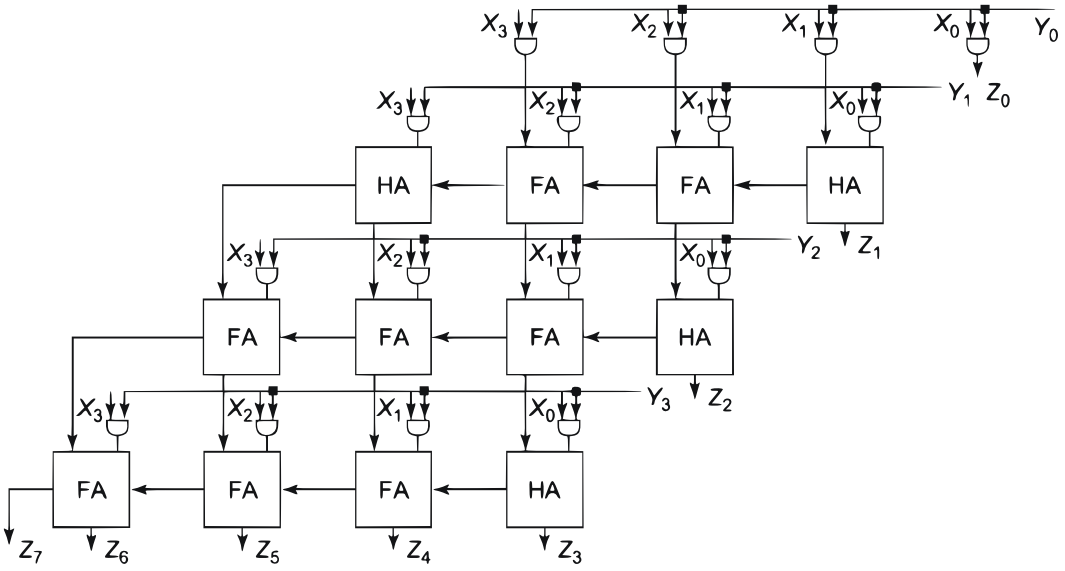


Obrázek 2.26: Princip binárního násobení [5].

Kombinační násobička

Kombinační násobička (*array násobička*) sestává z postupného získávání částečných součinů P s využitím AND hradel a jednobitových sčítaček. Struktura zapojení 4 bitové array násobičky je k prohlédnutí na obrázku 2.27.

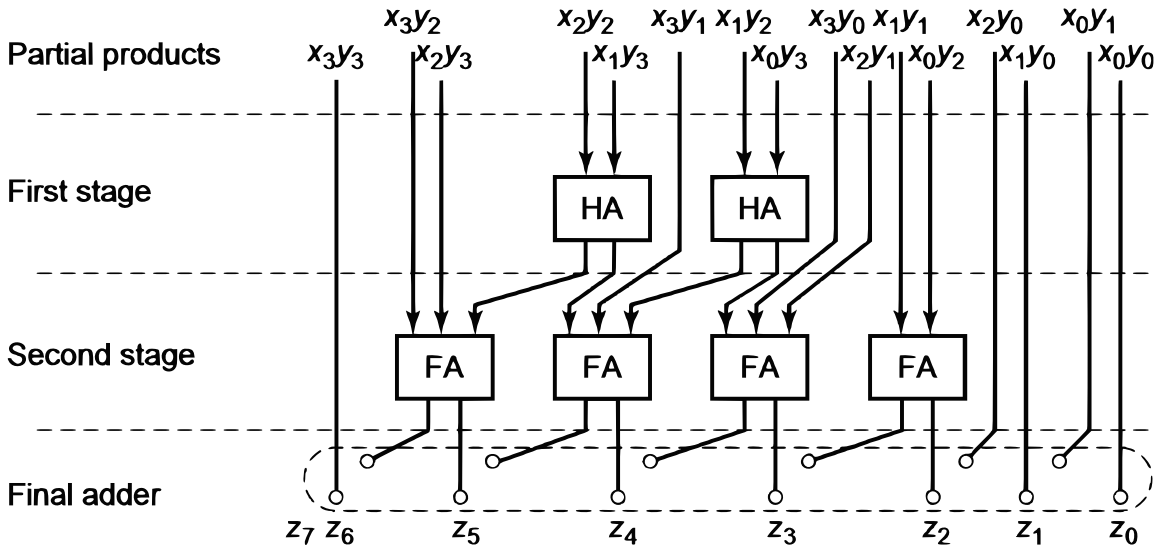
V případě popisu zapojení znaménkové kombinační násobičky se struktura částečně liší a sestává z jednobitových sčítaček, AND i NAND hradel.



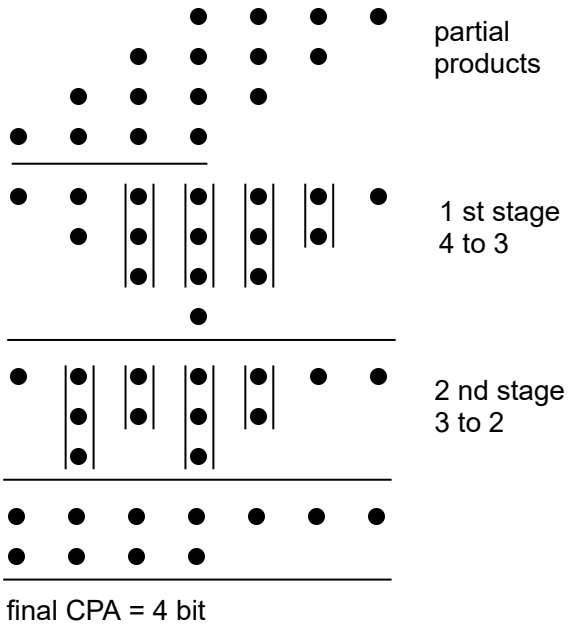
Obrázek 2.27: Struktura 4 bitové kombinační násobičky.

Wallaceova násobička

Wallaceova násobička (*Wallaceův strom*) představuje HW výhodnější realizaci vícebitové násobičky oproti kombinační násobičce. Vyznačuje se postupnými redukcemi sloupců částečných součinů s využitím jednobitových sčítaček, dokud v každém sloupci nezbudou maximálně dva bity. Nakonec se provede sečtení takto zredukovaných hodnot pomocí vícebitové sčítačky *CPA* (*Carry-propagate adder*), jenž značí souhrnné pojmenování pro obvody sčítaček, které spojuje potřeba šíření příznaku přenosu. Způsob redukce jednotlivých stupňů je znázorněn na obrázku 2.28. Obrázek 2.29 potom ukazuje postupné redukování částečných součinů pro 4 bitovou Wallaceovu násobičku.



Obrázek 2.28: Princip redukce Wallaceovy násobičky (stromu) [7].



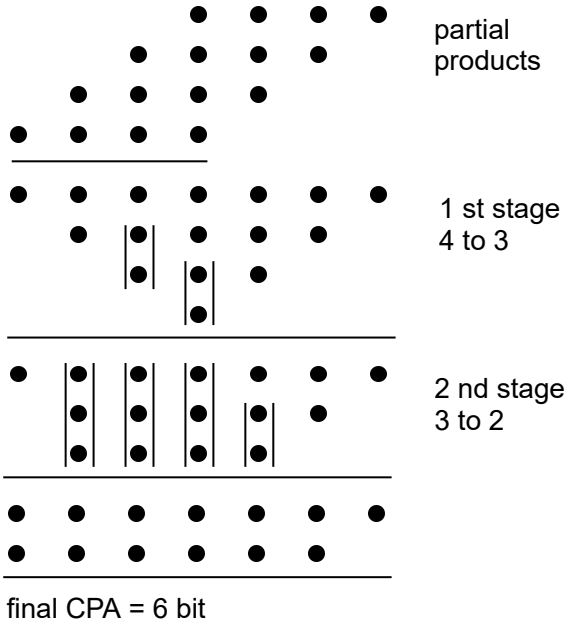
Obrázek 2.29: Ukázka redukce 4 bitové Wallace násobičky [8]. Jednotlivé tečky nejprve představují částečné součiny vstupních hodnot a v dalších fázích jejich postupné redukce. Ve sloupcích ohraničené dvojice značí jejich kombinaci pomocí poloviční jednobitové sčítačky a ve sloupcích ohraničené trojice pak jejich kombinaci pomocí úplné jednobitové sčítačky. *Sum* po redukci ve sloupci zůstává, *Cout* se přenáší do dalšího sloupce.

Dadda násobička

Dadda násobička představuje efektivnější a HW méně náročnější Wallaceovu násobičku. Princip postupných redukcí je podobný jako u Wallaceova stromu, akorát je celkový počet potřebných stupňů redukcí menší díky dodatečným pravidlům, dle nichž se redukce

řídí. Postup redukcí je řízen posloupností maximálních výšek sloupců částečných součinů d_j definovanou v rovnici (2.14). Dále se zvolí co nejvyšší počáteční redukční stupeň j, aby platilo, že $d_j < N$. Díky tomu je ke konstrukci zapotřebí menšího počtu jednobitových sčítaček oproti Wallaceovým násobičkám, což se projevuje zejména menší zabíranou plochou. Ukázka redukcí s využitím posloupnosti d_j pro 4 bitovou Dadda násobičku je vidět na obrázku 2.30. Zbytek funkčnosti je totožný s Wallaceovým stromem. Pro získání více informací k Dadda násobičkám můžete nahlédnout například do [9].

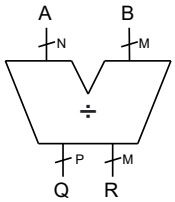
$$d_1 = 2 \quad a \quad d_{j+1} = \lfloor 1.5 * d_j \rfloor \tag{2.14}$$



Obrázek 2.30: Ukázka redukce 4 bitové Dadda násobičky [8]. Jednotlivé tečky nejprve představují částečné součiny vstupních hodnot a v dalších fázích jejich postupné redukce. Ve sloupcích ohraničené dvojice značí jejich kombinaci pomocí poloviční jednobitové sčítačky a ve sloupcích ohraničené trojice pak jejich kombinaci pomocí úplné jednobitové sčítačky. *Sum* po redukci ve sloupci zůstává, *Cout* se přenáší do dalšího sloupce.

2.3.4 Děličky

Architektury děliček se dají rozčlenit do dvou skupin. Buďto se jedná o obvody realizující pouze dělení násobky 2 s využitím bitových posuvů, tzv. *shifters*, nebo o obvody provádějící náročnější operaci dělení dvou čísel, která je založena na principu bitového posuvu *dělence* a od něj postupného odčítání hodnoty *dělitele*. Budeme se věnovat principu druhého typu děliček. Tyto vícebitové děličky realizují operaci dělení mezi dvojicemi bitů ze vstupních sběrnic A (*dělenec*), B (*dělitel*) o bitových šířkách N, respektive M. Obecně nemusí platit, že $N = M$. Výsledkem operace jsou jednotlivé bity podílu Q (*quotient*) o délce P bitů a bity zbytku po dělení R (*remainder*) o délce M bitů. Blokový diagram reprezentující popisované vícebitové děličky viz obrázek 2.31. Následuje popis architektury zastupující tyto děličky.

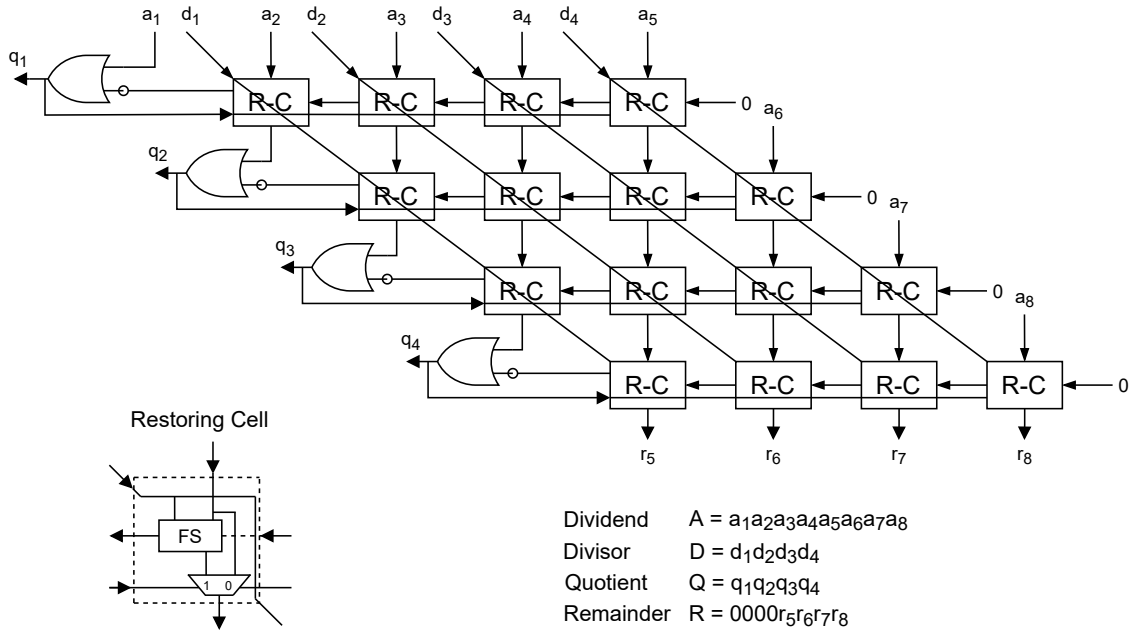


Obrázek 2.31: Blokový diagram vícebitové odčítačky.

Restoring array dělička

Proces tzv. *restoring* dělení (volně přeloženo jako „obnovovací“ dělení) představuje jeden z možných přístupů realizace dělení dvou čísel. Spolu s dalšími jako jsou *non-restoring* či *SRT algoritmus* patří mezi **pomalé** algoritmy dělení. **Rychlejšími** a složitějšími variantami jsou potom algoritmy *Newton–Raphson* a *Goldschmidt*. Obnovovací dělení je iterativní proces, jejíž princip využívá bitových posuvů a odčítání. Při každé iteraci dochází k bitovému posuvu dělence A a odečtení jeho hodnoty od dělitele B. Výsledek každé iterace je uložen do odpovídajícího výstupního bitu podílu na bitovou pozici dané iterace. Při poslední iteraci představují zbylé bity rozdílu zbytky po dělení. Více informací k jednotlivým algoritmům dělení naleznete např. v knize [10].

Restoring array dělička představuje typ pomalé kombinační děličky dvou čísel, jejíž struktura sestává zejména z úplných odčítaček a dvouvstupých multiplexorů jako hlavních stavebních bloků. Struktura zapojení 4 bitové restoring děličky je k prohlédnutí na obrázku 2.32.



Obrázek 2.32: Struktura restoring děličky s 8 bitovým dělencem a 4 bitovým dělitelem [11].

Kapitola 3

Způsoby popisu vnitřní reprezentace aritmetických obvodů

Aritmetické obvody lze popsat pomocí různých reprezentací. Jednotlivé reprezentace slouží specifickým účelům vůči popisovanému obvodu.

Jazyk C K rychlé simulaci očekávaného chování poslouží jazyk C díky své jednoduché kompilaci a spuštění.

HDL jazyky Slouží k popisu hardware, zobrazení RTL schématu a logické syntéze obvodu. Příkladem je například jazyk Verilog či VHDL.

BLIF[1] Formát určený pro jednodušší strojové zpracování a používaný zejména k formální verifikaci obvodů.

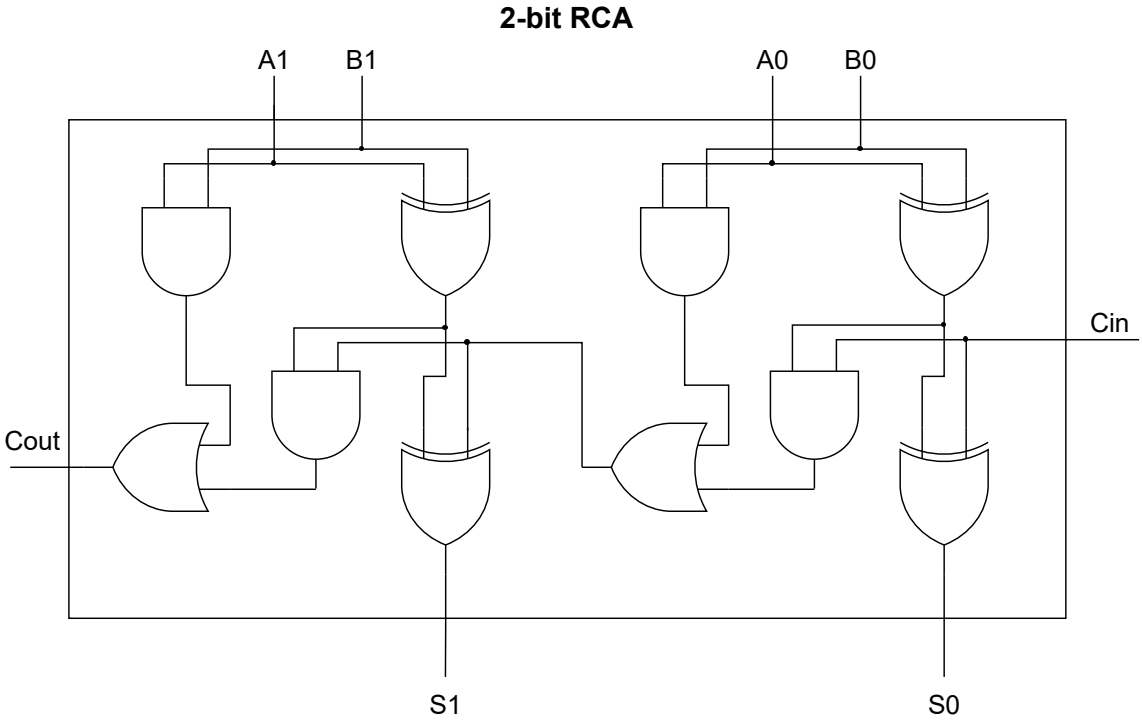
CGP[2] Specializovaný formát popsaný ve formě celočíselného netlistu a používaný jako chromozom pro evoluční optimalizaci [12] (např. pomocí evoluční strategie).

Přestože jsou reprezentace vzájemně odlišné samotným popisem, dovedou charakterizovat chování stejného obvodu. Některé reprezentace je možné popsat dvěma různými způsoby, tzv. *plochým* a *hierarchickým* popisem. Kromě chromozomu CGP lze všechny reprezentace popsat oběma způsoby. Při optimalizaci pomocí CGP by se pak spíše optimalizovaly jednotlivé moduly samostatně.

3.1 Plochý popis

Plochý znamená, že je obvod popsán na úrovni nejnižších komponent, tj. logických hradel. Vzniklé reprezentace tak mohou být velké vzhledem k nutnosti namapovat všechny interní vodiče a hradla. Zploštění seskupuje veškerou funkční logiku do jediné komponenty a umožňuje provádět globální optimalizace např. pomocí kartézského genetického programování (CGP). Ukázku schématu plochého obvodu najdete na obrázku 3.1.

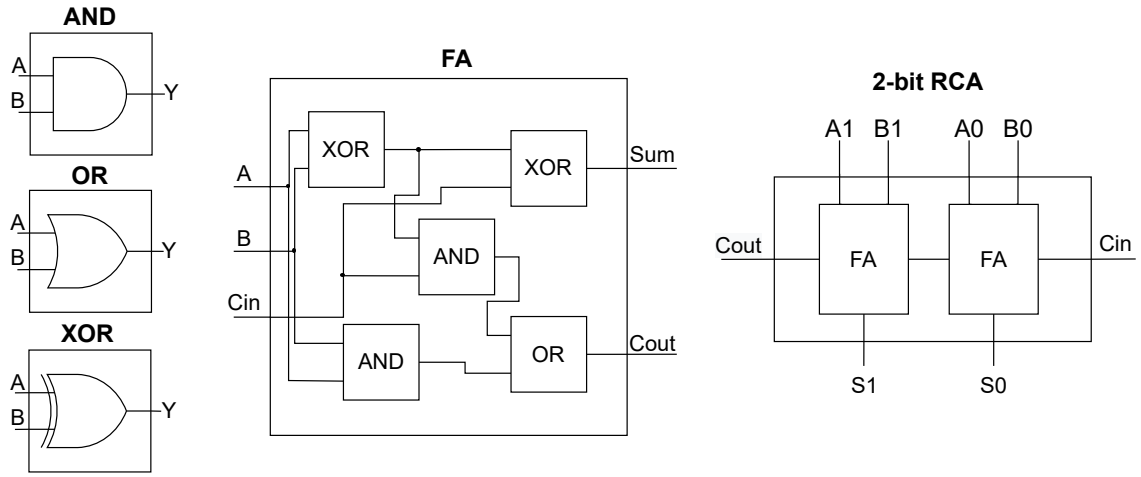
[1] Berkeley Logic Interchange Format.
[2] Kartézské Genetické Programování.



Obrázek 3.1: Plochá reprezentace 2 bitové bezznaménkové RCA sčítačky.

3.2 Hierarchický popis

Hierarchický popis se vyznačuje znovupoužitelností jednou definovaných funkčních bloků. Bloky se mohou vzájemně necyklicky zanořovat. Funkční blok reprezentuje ucelenou komponentu realizující specifickou bitovou operaci: jednotlivá logická hradla, jednobitová úplná sčítačka apod. Vzniklá reprezentace je díky seskupení logiky do větších celků a jejich opakovanému použití mnohem menší než plochá. Právě vícenásobné použití stejných funkčních bloků umožňuje provádět lokální optimalizace jednou definovaných bloků. Ukázku schématu s využitím hierarchie funkčních bloků najdete na obrázku 3.2.



Obrázek 3.2: Hierarchická reprezentace 2 bitové bezznaménkové RCA sčítačky.

3.3 Bližší specifikace jednotlivých reprezentací popisu

V této sekci si podrobněji představíme jednotlivé reprezentace popisu, abychom získali představu o tom, jak vypadají, čím se navzájem liší a k čemu slouží.

3.3.1 Jazyk C

Pomocí nízkoúrovňového jazyka C můžeme jednoduše popisovat struktury aritmetických obvodů a jejich funkční logiku seskupovat do *funkcí*. Vodiče i sběrnice jsou reprezentovány pomocí proměnných. Jazyk C ovšem neumožňuje přímo přistupovat k jednotlivým bitům, a proto si musíme pomoct bitovými posuny. Za účelem vykonání logických operací dílčích hradel provádíme s bity příslušné bitové operace. Při hierarchickém popisu je každý typ komponenty popsán vlastní funkcí, která je pak hierarchicky volána z funkce patřící nadřazenému obvodu.

Běžné aritmetické operace se provádí přímo v aritmeticko-logické jednotce CPU a využíváme k nim aritmetické operátory. Jiná reprezentace těchto operací je pak nevýhodná, jelikož každou musíme simulovat. V tomto případě však reprezentace v C slouží k otestování funkčnosti obvodu, případně k zanesení chyb aproximací, kdy dochází ke zpomalení simulace, která je však rychlejší než simulace na úrovni RTL.

3.3.2 HDL jazyky

Mezi specializované jazyky používané k popisu struktur a chování HW patří Verilog a VHDL. HDL jazyky představují vyšší úroveň abstrakce, zvanou RTL, oproti schématickému popisu. Vyjadřovací schopnosti obou jazyků jsou stejné, pouze jazyk VHDL je odvozený z jazyka ADA a tím je méně kompaktnější než Verilog.

Poslouží zejména k logické syntéze obvodů, kdy se popis HW převede do netlistu logických hradel a jejich vzájemného propojení. To umožňuje získat řadu parametrů o popisovaných obvodech jako jsou zpoždění, příkon či plocha.

Verilog

Popis struktury aritmetických obvodů pomocí jazyka Verilog se v některých aspektech podobá popisu v jazyce C. Vodiče jsou uchovány ve speciálním datovém typu *wire* a sběrnice, sestávající z mnoha vodičů, v datovém typu *bus*. Přistupovat k jednotlivým bitům sběrnice je možno pomocí hranatých závorek (podobně jako pole v C). Verilog seskupuje popis jednotlivých komponent do ucelených bloků – *modulů* a k vykonání funkcí jednotlivých hradel využívá stejné bitové operátory jako jazyk C. Při hierarchickém popisu je každý typ komponenty popsán uvnitř vlastního modulu, který je pak hierarchicky volán z modulu popisující nadřazený obvod.

VHDL

Proces popisu obvodu pomocí jazyka VHDL je rozdělen do dvou částí. Nejdříve je třeba uvnitř konstrukce *entity* definovat název, vstupní a výstupní rozhraní popisovaného obvodu. Následně na řadu přichází popis vnitřní struktury a chování vytvořené entity uvnitř konstrukce *architecture*. Architekturu entity je ve VHDL možno popsat různými způsoby. Aby mohl být popsaný obvod syntetizován, používají se tzv. *behaviorální*, *data-flow* nebo *strukturální* způsoby modelování. Pro popis kombinačních obvodů však typicky používáme *data-flow* nebo *strukturální* způsob.

Data-flow modelování slouží k popsání architektury plochým způsobem, kdy jsou k jednotlivým výstupním vodičům přiřazeny odpovídající hodnoty s využitím funkcí logických hradel. K jejich vykonání se využívají vestavěné logické operátory pojmenováné dle typu hradla („and“, „nand“, . . .).

Strukturální modelování slouží k popisu hierarchické reprezentace. Nejdříve jsou samostatně definovány entity nejnižších funkčních komponent s využitím *data-flow* modelování. Popis architektury nadřazené entity (obvodu) potom obsahuje deklarace rozhraní použitých komponent, deklarace signálů (vodičů, sběrnic) pokud jsou třeba a nakonec funkční kód sestávající z instanciací jednotlivých komponent.

3.3.3 BLIF

BLIF představuje ne příliš známý formát reprezentující zploštělý hierarchický popis obvodu, jenž má být implementován na cílovém FPGA[3]. Hierarchii je možno použít ve smyslu volání definovaných funkčních bloků, avšak interně se popis uchovává v netlistu všech hradel, který představuje ono zploštění.

Struktura obvodu je definována uvnitř tzv. *modelů*, jež mají určené vstupní a výstupní vodiče. BLIF neumožňuje deklaraci sběrnice, jednotlivé vodiče jsou reprezentovány svým jménem. Klíčová slova jsou uvozena tečkou, např. „.model“, „.names“, „.subckt“.

Klíčové slovo *.model* slouží k vytvoření názvu popisovaného modelu. Za ním následuje řádek začínající slovem *.inputs* se jmény vstupů modelu a řádek začínající slovem *.outputs* s názvy výstupů modelu. Samotné tělo modelu pak sestává z konstrukcí tvořených s pomocí klíčového slova *.names*. Ten slouží k realizaci logických funkcí jednotlivých hradel i k přiřazení logické hodnoty k výstupnímu vodiči. Funkce logických hradel jsou určeny kombinací vstupních hodnot, jež jsou pro každé hradlo definovány zvlášť, a které jsou vyvoditelné z pravdivostních tabulek dílčích hradel viz sekce 2.1. K ukončení definice modelu slouží klíčové slovo *.end*.

[3] Programovatelné hradlové pole.

Hierarchie je pak implementována přes volání podmodelu přes pokyn „.subckt“ následovaný jménem podmodelu a přiřazením vodičů k příslušným názvům vstupních, výstupních vodičů volaného podmodelu.

BLIF se používá převážně v akademických nástrojích např. pro mapování obvodů na LUT (*vyhledávací tabulka*), formální verifikaci a podobně. S tímto formátem pracují nástroje jako je *ABC*, *Cirkit* nebo *Yosys Open SYnthesis Suite*, o kterém se ještě zmíníme a více o něm můžete nalézt zde [13].

Detailní popis BLIF formátu naleznete zde [14]. Ukázku plochého popisu reprezentace poloviční sčítačky v BLIF formátu obsahuje obrázek 3.3 a ukázku jejího hierarchického popisu pak obrázek 3.4.

Plochá reprezentace obsahuje na řádku 1-3 popis názvu modelu poloviční sčítačky spolu s definicemi jejích vstupů a výstupů. Následují dvě konstrukce *.names*, jež popisují funkce dvouvstupého XOR hradla (řádky 4-6) a dvouvstupého AND hradla (řádky 7-8). Zároveň provádějí přiřazení k výstupním vodičům.

Hierarchická reprezentace obsahuje definici hlavního modelu poloviční sčítačky (řádky 1-6) a definice podmodelů jejích podkomponent – AND hradla (řádky 8-13) a XOR hradla (řádky 15-21). Dílčí definice podmodelů hradel obsahují příslušnou konstrukci *.names* z ukázky ploché reprezentace. Hlavní model však místo konstrukcí *.names* využívá volání vytvořených podmodelů pomocí konstrukce uvozené klíčovým slovem *.subckt*. Za ním následuje jméno daného podmodelu, přiřazení vodičů k jeho příslušným vstupním parametrům a přiřazení výstupního vodiče podmodelu k určenému vodiči uvnitř hlavního modelu.

```
1 .model f_ha
2 .inputs a b
3 .outputs f_ha_xor0 f_ha_and0
4 .names a b f_ha_xor0
5 01 1
6 10 1
7 .names a b f_ha_and0
8 11 1
9 .end
```

Obrázek 3.3: Plochý popis poloviční sčítačky v BLIF reprezentaci.

```
.model h_ha
.inputs a b
.outputs h_ha_xor0 h_ha_and0
.subckt xor_gate a=a b=b out=h_ha_xor0
.subckt and_gate a=a b=b out=h_ha_and0
.end

.model and_gate
.inputs a b
.outputs out
.names a b out
11 1
.end

.model xor_gate
.inputs a b
.outputs out
.names a b out
01 1
10 1
.end
```

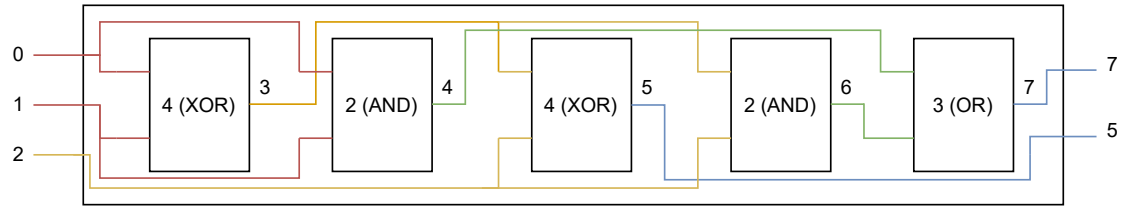
Obrázek 3.4: Hierarchický popis poloviční sčítačky v BLIF reprezentaci.

3.3.4 Kartézské genetické programování

Kartézské genetické programování (CGP) je formou *genetického programování* (*GP*), které je často využíváno k návrhu a optimalizaci číslicových obvodů. Tyto obvody jsou reprezentovány fixní 2D mřížkou uzlů a jejich propojení. Mřížka nabývá rozměrů $N_r \times N_c$, tedy počtu řádků krát počtu sloupců. Číslo N_i potom značí počet vstupů a číslo N_o počet výstupů.

CGP se primárně využívá k popisu výpočetních struktur (obvody, matematické rovnice, apod.), jež jsou popsány ve formě genotypu složeného z posloupnosti jednotlivých uzlů (*chromozomů*). Uzly představují celočíselný seznam, reprezentující určitou operaci nad vstupními daty (např. funkce logických hradel). Celá čísla slouží k identifikaci jednotlivých funkcí uzlů, ke kterým jsou pevně přiřazena a také slouží k jednoznačné identifikaci vstupů a výstupů, kterým jsou čísla sekvenčně přidělována dle jejich pozice v grafu. Všechny uzly mají stejný počet vstupů a výstupů napříč celým genotypem. Bližší specifikace CGP viz [15].

CGP se ukázalo být přínosem pro evoluční optimalizace obvodů s cílem vytvoření aproximačních (přibližných) obvodů, které jsou cíleně navrženy ke snížení plochy, zpoždění a příkonu. O aproximačních obvodech se můžete dozvědět více např. zde [16, 17]. Obrázek 3.5 obsahuje strukturu úplné jednobitové sčítačky popsané pomocí CGP.



Obrázek 3.5: CGP reprezentace úplné jednobitové sčítačky, kde $N_i = 3$, $N_o = 2$ a rozměry mřížky jsou 1×5. (Pozn. barvy propojů nemají bližší význam, slouží pouze k jejich rozlišení.)

Kapitola 4

ArithsGen – generátor aritmetických obvodů

V minulosti na tuto problematiku vznikla již řada generátorů, některé jsou dnes už zastaralé, jiné naopak představují teoretický podklad k řešení, avšak implementace samotná již veřejná není, příklady takových prací jsou [18, 19]. V současnosti pak existuje jen pár volně dostupných nástrojů schopných generovat aritmetické obvody pro konkrétní druhy reprezentací. Dále se podíváme na volně dostupné nástroje pro generování aritmetických obvodů.

Arithmetic Module Generator [20] využívá grafové reprezentace obvodů pomocí Arithmetic Circuit Graph (ACG) k formálnímu popisu, verifikaci a převodu do hierarchické reprezentace v HDL jazycích. Výhodou je široká podpora pro různé typy obvodů s možností bližší specifikace obvodu jako např. šířka vstupních sběrnic, znaménkovost apod. Vzhledem ke svému webovému rozhraní však nástroj neumožňuje automatizované generování většího počtu obvodů a není uživatelsky rozšiřitelný ani upravitelný. Navíc negeneruje samotný ACG či netlist všech obsažených komponent vygenerovaného obvodu, čímž neumožňuje lepší globální optimalizaci pomocí evolučních algoritmů.

VHDLMultGenerator [21] je program, který s využitím grafického uživatelského rozhraní dovede generovat rozličné obvody násobiček a exportovat je do VHDL reprezentace. Nedostatkem je zejména omezení se pouze na násobičky a jediný výstupní formát: VHDL.

Dříve vzniknuvší bakalářská práce [22] na tuto problematiku poskytuje podporu generování široké škály aritmetických obvodů do ploché či hierarchické reprezentace v jazyce C a umožňuje obvod vizualizovat ve formě schématu programem SpiceVision[1]. Problém je v lineární vnitřní struktuře generovaných obvodů, která není hierarchická a značně ztěžuje možnost nástroj rozšířit o další výstupní reprezentace týchž obvodů.

4.1 Nástroj ArithsGen

4.1.1 Koncepce nástroje ArithsGen

V této práci navrhuji nový nástroj zvaný ArithsGen, který umožňuje navrhovat libovolné aritmetické obvody a generovat jejich výstupní reprezentace do plochého i hierarchického způsobu popisu. Podporovanými výstupními reprezentacemi jsou jazyk C, Verilog, BLIF a CGP. Z HDL jazyků byl zvolen Verilog vzhledem k jeho větší kompaktnosti oproti VHDL.

[1] SpiceVision PRO: https://www.concept.de/SpiceVision.html

Implementace je založena na RTL úrovni abstrakce a používá dvouvstupá logická hradla. Při návrhu generování aritmetických obvodů bylo hlavním cílem využití modularity aritmetických obvodů k dosažení jednotného generování výstupních reprezentací těchto obvodů bez ohledu na jejich vnitřní strukturu. Modularita spočívá v postupném skládání obvodů z menších komponent, poskytující tak různé úrovně hierarchického zanoření. Zároveň byl kladen důraz na umožnění jednoduchého rozšíření nástroje o další architektury a výstupní reprezentace.

Pro implementaci byl zvolen jazyk Python, díky jehož možnostem generátor modeluje modularitu aritmetických obvodů k popisu jejich vnitřních struktur. Objektová orientace jazyka Python umožňuje pohodlně popsat struktury i složitějších komponent, než by bylo možné pomocí jiných programovacích paradigmat. Popis vnitřní reprezentace obvodu v programu navíc přímo reflektuje návrh schématu, čímž je docíleno přímočařejšího přepisu popisu obvodu z navrženého schématu do jeho struktury v programu. Díky tomu a díky systému dědičnosti jazyka je pak nástroj lépe rozšiřitelný.

4.1.2 Balíček nástroje ArithsGen

Implementace generátoru se nachází v balíčku nejvyšší úrovně `ariths_gen`, jehož strukturu naleznete v příloze A.1. Podbalíček `core` obsahuje moduly popisující obecné struktury obvodů spolu s principy generování jejich výstupních reprezentací. Podbalíčky `multi_bit_components` a `one_bit_components` pak obsahují moduly pro popis struktur jednotlivých architektur obvodů, menších komponent a logických hradel. Nakonec podbalíček `wire_components` obsahuje popis komponent spojů sloužících k vzájemnému propojení obvodů mezi sebou.

4.1.3 Implementace obecných typů obvodů a generování jejich výstupních reprezentací

Způsob generování výstupních reprezentací je popsán v třídách modulů podbalíčku `core`. Tyto třídy představují popis obecných typů obvodů spolu s jejich metodami a atributy, které se využívají k popisu vnitřních struktur jednotlivých obvodů a taktéž ke generování jejich výstupních reprezentací. Z těchto tříd postupně dědí jednotlivé podtřídy popisující konkrétní typy obvodů (komponent). Implementaci struktur obvodů se podrobněji věnuje podsekce 4.1.4.

Společné atributy a metody vícebitových obvodů spolu s principy exportu jejich výstupních reprezentací popisuje třída `ArithmeticCircuit` modulu `arithmetic_circuit.py`. Třída `MultiplierCircuit` uvnitř modulu `multiplier_circuit.py` dědí vlastnosti z nadtřídy `ArithmeticCircuit`, ale navíc obsahuje metody sloužící k implementaci vnitřních struktur obvodů násobiček.

Atributy a metody jednobitových obvodů spolu s principy exportu jejich výstupních reprezentací jsou popsány v třídách pro dvouvstupé obvody `TwoInputOneBitCircuit` a třívstupé obvody `ThreeInputOneBitCircuit` uvnitř modulů `two_input_one_bit_circuit.py`, respektive `three_input_one_bit_circuit.py`. Třída `ThreeInputOneBitCircuit` přitom dědí některé společné vlastnosti z třídy `TwoInputOneBitCircuit` a liší se pouze v počtu vstupních vodičů.

Definici společných Atributů a metod a principu exportu logických hradel potom obsahují třídy `TwoInputLogicGate`, `TwoInputInvertedLogicGate` a `OneInputLogicGate` vyskytujících se v modulu `logic_gate_circuit.py`. `TwoInputInvertedLogicGate` i `OneInputLogicGate` pak dědí určité vlastnosti z nadtřídy `TwoInputLogicGate` a první z dvojice

slouží k popisu negovaných dvouvstupých hradel, zatímco druhá třída se věnuje obecnému popisu jednovstupých hradel.

Implementace generování výstupních reprezentací popisu

Výše popsané třídy obsahují definice atributů a metod, které se používají ke dvěma různým účelům. Jednak umožňují popsat vnitřní struktury jednotlivých architektur obvodů a za druhé slouží k exportu definovaných struktur do jejich rozličných výstupních reprezentací.

Vzhledem k odlišnosti výstupních reprezentací se proces generování pro jednotlivé druhy liší. Využívá se však stejného principu. Díky seznamu objektů **components** z nichž se obvod skládá a schopnosti hierarchického zanoření do seznamů objektů podkomponent onoho obvodu je tak získán přehled a přístup ke všem komponentám tvořící výsledný obvod.

Při generování *plochých* reprezentací se postupuje následovně:

1. Vytvoření hlavičky popisovaného obvodu obsahující název spolu s definicí jeho vstupních/výstupních rozhraní (vodičů, sběrnic).
2. Unikátní deklarace interních vodičů obvodu, je-li to pro danou reprezentaci potřeba, k předejití jmenných kolizí.
3. Přiřazení hodnot deklarovaným vodičům k výsledku bitové operace mezi dvěma jinými propoji.
4. Přiřazení hodnot výstupních vodičů k příslušným bitovým pozicím výstupní sběrnice.

Hierarchické reprezentace vyžadují nejprve definovat všechny typy funkčních bloků, které hierarchicky tvoří výsledný obvod. Postup generování je v tomto případě následující:

1. Vytvoření unikátní reprezentace jednotlivých typů funkčních bloků, z nichž se hlavní obvod skládá, podobně jak u plochého popisu, avšak k přiřazení hodnot vnitřním vodičům se využívají výstupní hodnoty z volání již definovaných menších bloků.
2. Definice popisu hlavního obvodu probíhá podobně jako u předešlých bloků. Tzn. zploštělý princip deklarace hlavičky, deklarace vnitřních vodičů, přiřazení hodnot vodičů k výstupům z volání již dříve definovaných funkčních celků.
3. Přiřazení výstupních vodičů k příslušným bitovým pozicím výstupní sběrnice.

4.1.4 Implementace vnitřních struktur jednotlivých obvodů

Každý typ součástky počínaje vodiči, sběrnicemi a logickými hradly je popsán pomocí vlastní třídy. Logická hradla představují nejzákladnější logicky samostatné funkční jednotky, z nichž se vytvářejí struktury složitějších obvodů. K popisu tříd složitějších obvodů se využívá instanciace objektů tříd reprezentujících nižší stavební komponenty. Patřičné vodiče jednotlivých komponent jsou pak vzájemně propojeny s využitím jejich vstupně-výstupních rozhraní.

Komponenty propojů

Jednobitové vodiče a vícebitové sběrnice propojují obvody mezi sebou přes jejich vstupně-výstupní rozhraní. Jejich struktury jsou implementovány v modulech `wires.py`, `buses.py` podbalíčku `wire_components`.

Vodič je implementován uvnitř třídy `Wire` a uchovává o sobě parametry `name`, `prefix`, `value`, `index` a `parent_bus`. Parametr `name` slouží k jeho jednoznačné identifikaci ve výstupních reprezentacích. Parametr `prefix` nese název sběrnice v případě, že je některé vodič součástí, jinak nabývá stejné hodnoty jako `name`. Hodnota `value` (implicitně 0) představuje počáteční hodnotu vodiče využívanou při generování jeho deklarace a inicializace v jazyce C. Číselná hodnota `index` (implicitně 0) pak značí pozici vodiče ve sběrnici. Parametr `parent_bus` je objektem nadřazené sběrnice, které je vodič součástí nebo *None*, jestliže je vodič samostatným spojem.

Mimo obecný vodič `Wire` existují i z něj odvozené třídy `ConstantWireValue1` a `ConstantWireValue0`, jež reprezentují vodiče připojené ke zdroji napětí (konstantní logická hodnota 1), respektive k zemi (konstantní logická hodnota 0). Tyto vodiče mají neměnné vlastnosti týkající se jejich pojmenování i hodnot, které je reprezentují ve výstupních reprezentacích.

Třída popisující vícebitové sběrnice nese název `Bus` a uchovává o sobě parametry `prefix`, `N` a `wires_list`. Parametr `prefix` jednoznačně identifikuje název sběrnice. Číslo `N` (implicitně 1) udává počet vodičů ve sběrnici. Volitelný parametr `wires_list` slouží k předání explicitního seznamu vodičů, z nichž se má sběrnice skládat (v takovém případě je parametr `N` ignorován a délka sběrnice bude určena délkou vstupního seznamu). Dle kombinace vstupních parametrů se naplní vnitřní seznam `bus` vodiči, a to buď postupnou instanciací vodičů podle specifikovaného počtu `N`, a nebo vodiči z explicitního seznamu `wires_list`.

Při generování výstupních CGP reprezentací je ke každému vodiči navíc zjištěna jeho pozice v popisovaném obvodu. Konstantní vodiče mají pevně přiřazenou hodnotu, jenž odpovídá logické hodnotě, které jsou nositelem (0, 1). Pro obecný typ vodičů se pak vždy jedná o unikátní hodnotu, přičemž indexování začíná od číslice 2 vzhledem k rezervovaným indexům pro konstantní spoje.

Komponenty logických hradel

Dvouvstupá a jednovstupá logická hradla jsou definována odlišným způsobem vzhledem k různému počtu svých vstupních vodičů. Popis chování každého dílčího hradla je popsán uvnitř vlastní třídy zděděné z nadtřídy popisující danou skupinu n-vstupých hradel. Tyto nadtřídy se vyskytují v modulu `logic_gate_circuit.py`, nacházejícím se v podbalíčku `core/logic_gate_circuits`. Samotné struktury hradel jsou potom implementovány v modulu `logic_gates.py` podbalíčku `one_bit_circuits/logic_gates`.

Jednotlivá dvouvstupá hradla 2.1.1 reprezentují třídy `AndGate`, `NandGate`, `OrGate`, `NorGate`, `XorGate`, `XnorGate`. Jejich konstruktorům se předávají parametry `a`, `b`, `prefix` a `parent_component`. Parametry `a` a `b` představují vstupní vodiče hradla. Parametr `prefix` se používá k pojmenování výstupního vodiče a slouží k jeho jednoznačné identifikaci napříč zbytkem obvodu. Objekt `parent_component` určuje nadřazený obvod, jehož je objekt hradla součástí a používá se v případě vnitřních optimalizací při generování výstupních reprezentací.

Z jednovstupých hradel 2.1.2 je v programu implementován pouze invertor, který je pro návrh aritmetických obvodů významný, a reprezentuje jej třída `NotGate`. Její konstruktor přijímá parametry `a`, `prefix` a `parent_component`. Význam parametrů je totožný jako u dvouvstupých hradel. Liší se tedy pouze v počtu vstupních vodičů.

Přiřazení celých čísel identifikujících příslušné funkce logických hradel pro výstupní reprezentace uzlů ve formátu CGP obsahuje tabulka 4.1.

Tabulka 4.1: Celá čísla funkcí logických hradel pro CGP reprezentaci.

Typ hradla	Číslo funkce
NOT	1
AND	2
OR	3
XOR	4
NAND	5
NOR	6
XNOR	7

V závislosti na typu vstupních vodičů (obyčejný vodič/konstanta) a jejich kombinací může dojít k zjednodušení struktury hradel a tím docílení vnitřní optimalizace návrhu obvodů. Pro popis principu optimalizace na úrovni hradel viz podsekce 4.1.6.

Komponenty jednobitových obvodů

Jednobitové obvody se podobně jako hradla dělí do různých skupin dle počtu jejich vstupních hodnot. Implementované jsou základní jednobitové obvody 2.2 se dvěma a třemi vstupy. Struktury dvouvstupých komponent jsou implementovány v modulu `two_input_one_bit_components.py` a struktury třívstupých komponent pak v modulu `three_input_one_bit_components.py` podbalíčku `one_bit_circuits/one_bit_components`.

Mezi dvouvstupé jednobitové obvody patří `HalfAdder`, `HalfSubtractor` a `PGLogicBlock`, jež reprezentuje blok pro zisk hodnot signálů *propagate*, *generate* a *sum* využívaný k popisu struktury CLA sčítačky. Konstruktorům všech tříd se předávají parametry `a`, `b` a `prefix`. Parametry `a` a `b` představují vstupní vodiče obvodu. Parametr `prefix` se používá k pojmenování a jednoznačné identifikaci patřičné komponenty uvnitř obvodu a taktéž je předáván při instanciaci jednotlivých složkových hradel jako součást jejich `prefixů`.

Mezi třívstupé jednobitové obvody patří `FullAdder`, `FullSubtractor`, `TwoOneMultiplexer` a `FullAdderPG`, jež odpovídá `PGLogicBlock` se třemi vstupy. Konstruktorům tříd jsou předány parametry `a`, `b`, `c` a `prefix`. Význam parametrů je totožný jako u dvouvstupých jednobitových obvodů. Liší se pouze třetím vstupním vodičem `c` jež reprezentuje buď *cin*, *bin* nebo řídicí signál *sel*.

Komponenty vícebitových obvodů

Vícebitové obvody jsou jednotlivě popsány ve svých vlastních třídách, které dědí z nadtřídy `ArithmeticCircuit` (kromě obvodů násobiček – ty dědí z nadtřídy `MultiplierCircuit`) nacházející se v podbalíčku `core/arithmetic_circuits`. Samotné struktury vícebitových obvodů jsou implementovány uvnitř svých vlastních modulů nacházejích se v příslušných adresářích `adders`, `multipliers` a `dividers` podbalíčku `multi_bit_circuits`.

Konstruktorům tříd obvodů se předávají parametry `a`, `b` a `prefix`. Parametry `a` a `b` představují vstupní sběrnice obvodu. Z nich se pomocí volání `max(a.N, b.N)` zjistí, která sběrnice má větší bitovou šířku `N`, jež slouží k určení bitové šířky popisovaného obvodu a jeho výstupní sběrnice. V případě, že jsou vstupní sběrnice rozdílných délek, dojde k rozšíření kratší z nich na jednotnou délku `N`. Parametr `prefix` se používá k pojmenování a jednoznačné identifikaci obvodu ve výstupních reprezentacích a taktéž je předáván při instanciaci jednotlivých podkomponent jako součást jejich `prefixů`.

Bezznaménkové sčítačky 2.3.1 implementují třídy `UnsignedRippleCarryAdder`, `UnsignedCarrySkipAdder`, `UnsignedCarryLookaheadAdder`, `UnsignedPGRippleCarryAdder`, jež představuje variaci RCA čítačky s využitím PG logiky. Konstruktory tříd obvodů CLA a CSkA navíc přijímají volitelný parametr `cla_block_size`, respektive `bypass_block_size` (oba implicitně 4), které určují velikost jednotlivých funkčních podbloků.

Znaménkové sčítačky, zastupující též obvody odčítaček 2.3.2, se liší svou vnitřní strukturou a pojmenováním odpovídajících tříd, kde místo slova `Unsigned` obsahují ve svém názvu slovo `Signed`.

Bezznaménkové Násobičky 2.3.3 implementují třídy `UnsignedArrayMultiplier`, `UnsignedWallaceMultiplier` a `UnsignedDaddaMultiplier`. Konstruktory tříd implementujících Wallaceovu a Dadda násobičku navíc přijímají parametr `unsigned_adder_class_name`, který slouží ke specifikaci třídy vícebitové bezznaménkové sčítačky, jež má být použita k závěrečnému sečtení zredukovaných dvojic bitů částečných součinů.

Znaménkové násobičky se stejně jako znaménkové sčítačky od svých bezznaménkových variant liší svou vnitřní strukturou a pojmenováním odpovídajících tříd (`Signed` místo `Unsigned`).

Zástupce bezznaménkových děliček 2.3.4 implementuje třída `ArrayDivider`. Ta pracuje se vstupními sběrnicemi shodných délek a jako výsledek vrací pouze bity podílu Q.

4.1.5 Princip využití modularity k popisu vnitřních struktur obvodů

Při instanciaci jednobitových a vícebitových obvodů se nejprve konstruktoru třídy předají vstupní parametry. Následně se dle typu obvodu a bitových šířek vstupů určí bitová šířka výstupní sběrnice. Při skládání obvodů se postupně metodou `add_component` vkládají do vlastního seznamu `components` objekty reprezentující menší podkomponenty. Seznam umožňuje přistoupit k již přidaným komponentám k docílení propojení výstupů předchozích komponent na vstupy následujících. Ono propojení realizuje metoda `connect`, která náleží instanci sběrnice, a jež specifikuje index k němuž má být připojen objekt vodiče `Wire`. Nakonec se připojí vodiče z výstupních rozhraní vytvořených podkomponent na příslušné bitové pozice výstupní sběrnice vytvářeného obvodu.

Ukázka popsaného principu na implementaci bezznaménkové RCA sčítačky je vidět na obrázku 4.1. Konstrukce na řádku 4 určuje ze vstupních sběrnic bitovou šířku popisovaného obvodu. Na řádcích 10 a 11 se provádí bitové rozšíření obou vstupních sběrnic na shodnou bitovou šířku, k čemuž dojde v případě, že je jedna ze sběrnic menší. Vytvoření výstupní sběrnice a určení její bitové šířky je popsáno na řádku 14. Na řádcích 17-28 se provádí postupné instanciování objektů jednobitových sčítaček a jejich přidávání do vnitřního seznamu podkomponent `components` s využitím metody `add_component`. Na řádku 26 je vidět volání metody `connect` k připojení výstupního bitu součtu *Sum* objektu jednobitové sčítačky na příslušnou bitovou pozici výstupní sběrnice. Nakonec řádek 28 provede s pomocí volání metody `connect` přiřazení výstupního příznaku přenosu *Cout* z objektu poslední jednobitové sčítačky k poslednímu bitu výstupní sběrnice obvodu.

```
class UnsignedRippleCarryAdder(ArithmeticCircuit):
  def __init__(self, a: Bus, b: Bus, prefix: str = "u_rca"):
    super().__init__()
    self.N = max(a.N, b.N)
    self.prefix = prefix
    self.a = Bus(prefix=a.prefix, wires_list=a.bus)
    self.b = Bus(prefix=b.prefix, wires_list=b.bus)

    # Bus sign extension in case buses have different lengths
    self.a.bus_extend(N=self.N, prefix=a.prefix)
    self.b.bus_extend(N=self.N, prefix=b.prefix)

    # Output wires for N sum bits and additional cout bit
    self.out = Bus(self.prefix+"_out", self.N+1)

    # Gradual addition of 1-bit adder components
    for input_index in range(self.N):
      # First adder is a half adder
      if input_index == 0:
        obj_adder = HalfAdder(self.a.get_wire(input_index),
            self.b.get_wire(input_index), prefix=self.prefix+"_ha")
      # Rest adders are full adders
      else:
        obj_adder = FullAdder(self.a.get_wire(input_index),
            self.b.get_wire(input_index),
            self.get_previous_component().get_carry_wire(),
            prefix=self.prefix+"_fa"+str(input_index))

      self.add_component(obj_adder)
      self.out.connect(input_index, obj_adder.get_sum_wire())
      if input_index == (self.N-1):
        self.out.connect(self.N, obj_adder.get_carry_wire())
```

Obrázek 4.1: Popis bezznaménkové RCA sčítačky v nástroji ArithsGen.

4.1.6 Vnitřní optimalizace obvodů nástrojem ArithsGen

Při generování můžeme rovnou provádět některé optimalizace. Budeme se soustředit pouze na ty základní, protože zbytek dále vyřeší syntézní nástroj. Nástroj automaticky provádí vnitřní optimalizace generovaných obvodů na úrovni logických hradel. Ta spočívá v analýze vstupních vodičů při instanciaci jednotlivých hradel. Jestliže nese některý ze vstupních vodičů konstantní hodnotu, tzn. že je připojen buďto ke zdroji napájení, nebo k zemi, pak patřičné hradlo na základě vyvození zbylých možných kombinací určí hodnotu výstupního vodiče. Může tedy dojít k situaci, kdy např. k hradlu AND přivedeme na jeden ze vstupních vodičů konstantní hodnotu 0. Potom můžeme dle zbylých kombinací pravdivostní tabulky 2.1 AND hradla určit, že hodnota jeho výstupního vodiče bude vždy nabývat logické hodnoty 0 – tedy bude rovna hodnotě vstupního uzemněného vodiče. V těchto případech se danému hradlu nastaví atribut `disable_generation` na *True*, a tím se zamezí jeho ná-

slednému generování při exportu do výstupních reprezentací. V jiných případech může také dojít k situaci, že je výstupní hodnota hradla rovna negaci některého z jeho vstupů. Pak se při výsledném generování výstupních reprezentací místo popisovaného hradla použije jednodušší NOT hradlo. Toto NOT hradlo je ale potřeba přidat do vlastního seznamu obsažených podkomponent `components` nadřazeného obvodu, jehož je popisované hradlo součástí. K tomuto účelu poslouží objekt `parent_component` předávaný při instanciaci hradla a odpovídající objektu nadřazeného obvodu.

Další optimalizací by bylo využití strukturního hashování, kdy by se pomocí kanonického popisu dalo v tabulce vyhledat, zda už daný vodič není někde definovaný. Tuto činnost ale provádějí samy syntézní nástroje a navíc by to nemělo zásadní vliv na stávající generované struktury, jelikož se exportují pouze nezbytně nutné vodiče.

4.1.7 Generování výstupních reprezentací obvodu

K vygenerování výstupu zvoleného aritmetického obvodu je třeba nejprve instanciovat jemu náležející třídu se zvolenými vstupními parametry. Následně stačí nad vytvořenou instancí obvodu, jež představuje reprezentaci obvodu uvnitř programu, zavolat metodu k vygenerování výstupu do kýžené reprezentace popisu.

Pro logická hradla je metoda tvaru **get__X__code** a u ostatních komponent pak tvaru **get__X__code__Y**, kde X značí výstupní reprezentaci a jedná se buď o „c“, „v“, „blif“ nebo „cgp“ a Y potom označuje způsob popisu výstupní reprezentace a může být „flat“ či „hier“. Ukázku popsaného generování nástrojem ArithsGen ukazuje obrázek 4.2.

```
# Definition of circuit's input buses and name
a = Bus(N=8, prefix="a")
b = Bus(N=8, prefix="b")
name = "f_u_cla8"

# 8-bit unsigned CLA instantiation and generation to flat output
    representations
circuit = UnsignedCarryLookaheadAdder(a=a, b=b, prefix=name)
circuit.get_c_code_flat(open(f"{name}.c", "w"))
circuit.get_v_code_flat(open(f"{name}.v", "w"))
circuit.get_blif_code_flat(open(f"{name}.blif", "w"))
circuit.get_cgp_code_flat(open(f"{name}.chr", "w"))
```

Obrázek 4.2: Ukázka generování 8 bitové bezznaménkové CLA sčítačky do plochých reprezentací popisu podporovaných nástrojem ArithsGen.

Kapitola 5

Experimenty a testování

Jak již bylo v kapitole 3 zmíněno, jednotlivé reprezentace popisu slouží svým specifickým účelům. K ověření funkčnosti generátoru proto bylo potřeba se intenzivně věnovat kontrole správnosti vygenerovaných reprezentací. Té se věnují jednotlivé následující sekce. Za porovnání stojí také doba potřebná k vygenerování rozličných výstupních reprezentací různých obvodů. Testování bylo provedeno na stolním počítači s procesorem Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz se 4 jádry, 8.0 GB RAM operační pamětí a 64-bitovým operačním systémem Microsoft Windows 10 Pro. Pro jednotlivé časy generování je uváděna průměrná hodnota z 10 měření. Tabulka 5.1 obsahuje dobu generování výstupních reprezentací pro 32 bitové bezznaménkové sčítačky a tabulka 5.2 dobu generování výstupních reprezentací pro 32 bitové bezznaménkové násobičky. Z naměřených hodnot je evidentní, že všechny reprezentace až na CGP zaberou přibližně stejnou dobu potřebnou k jejich vygenerování. Program při generování výstupní CGP reprezentace totiž stráví spoustu času nad voláním generátorových konstrukcí, které by mohly být urychleny. U sčítaček typu CLA lze také pozorovat různou dobu generování v závislosti na zvolené velikosti použitých funkčních bloků.

Tabulka 5.1: Srovnání doby generování výstupních reprezentací do plochého a hierarchického způsobu popisu pro 32 bitové bezznaménkové architektury sčítaček.

Architektura sčítačky	**Čas [s]**						
	C		**Verilog**		**BLIF**		**CGP**
	Ploché	**Hier.**	**Ploché**	**Hier.**	**Ploché**	**Hier.**	**Ploché**
RCA	0.003	0.005	0.003	0.005	0.003	0.005	0.019
PG_RCA	0.003	0.005	0.003	0.005	0.003	0.005	0.019
CSkA (4 bit bloky)	0.005	0.007	0.005	0.007	0.005	0.007	0.034
CSkA (8 bit bloky)	0.005	0.007	0.004	0.007	0.004	0.006	0.032
CSkA (32 bit bloky)	0.004	0.006	0.004	0.007	0.004	0.006	0.031
CLA (4 bit bloky)	0.012	0.014	0.012	0.015	0.012	0.013	0.071
CLA (8 bit bloky)	0.044	0.045	0.048	0.049	0.043	0.044	0.258
CLA (32 bit bloky)	3.905	3.986	3.946	3.971	3.893	3.881	18.073

Tabulka 5.2: Srovnání doby generování výstupních reprezentací do plochého a hierarchického způsobu popisu pro 32 bitové bezznaménkové architektury násobiček.

Architektura násobičky	Čas [s]						
	C		Verilog		BLIF		CGP
	Ploché	Hier.	Ploché	Hier.	Ploché	Hier.	Ploché
Kombinační násobička	0.086	0.135	0.075	0.137	0.076	0.129	12.103
Wallace (RCA)	0.227	0.277	0.217	0.281	0.215	0.270	12.345
Wallace (PG_RCA)	0.225	0.278	0.23	0.29	0.212	0.264	12.038
Wallace (CSkA)	0.24	0.3	0.22	0.3	0.22	0.277	13.133
Wallace (CLA)	0.261	0.34	0.25	0.379	0.264	0.334	14.362
Dadda (RCA)	0.211	0.268	0.202	0.275	0.2	0.251	13.169
Dadda (PG_RCA)	0.21	0.262	0.2	0.271	0.199	0.253	12.75
Dadda (CSkA)	0.218	0.28	0.209	0.284	0.211	0.264	13.484
Dadda (CLA)	0.254	0.334	0.237	0.359	0.234	0.317	14.31

5.1 Simulace chování obvodů v C, Verilog reprezentacích

Nad *plochými* i *hierarchickými* reprezentacemi obvodů v jazyce C a Verilog byly prováděny simulační běhy k otestování jejich očekávaného chování. Samotná simulace však může být u netriviálních obvodů časově velmi náročná, a tak bylo testování zejména C reprezentací omezeno jen do šířky 12 bitů, protože pro n-bitovou sčítačku/násobičku je potřeba 2^{2n} testování. Princip generování vnitřních struktur a jejich následný export do výstupních reprezentací je stejný i pro vyšší bitové šířky. To však jednoznačně neprokazuje jejich správnost, pouze připouští tento úsudek.

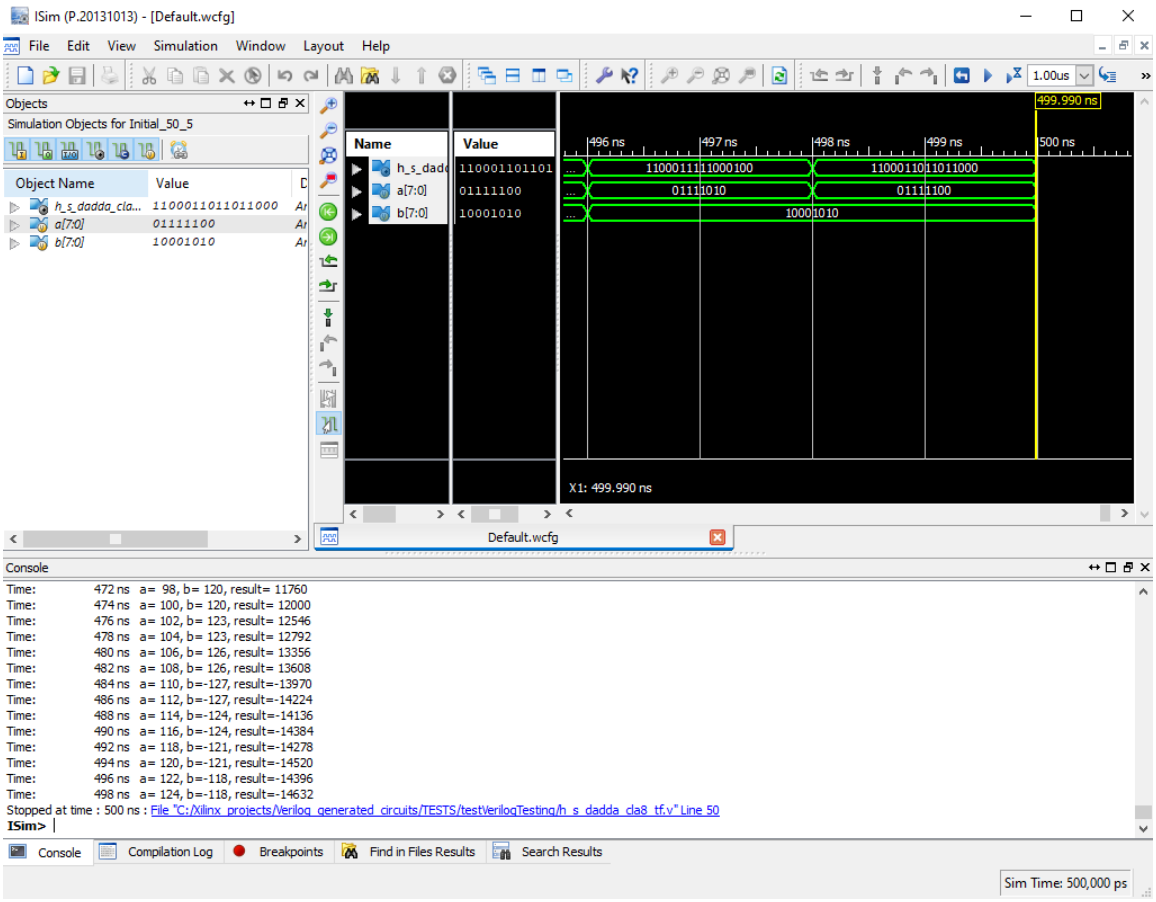
Testování obvodů v jazyce C bylo provedeno s vnořenými *for* cykly iterujících přes rozsah přípustných vstupních hodnot obvodu, odkud byla zanořeně volána funkce popisující daný aritmetický obvod s hodnotami řídicích proměnných smyček. Výsledek funkce byl dále pomocí makra *assert* porovnán s výsledkem referenčních aritmetických operací v jazyce C nad stejnými vstupními hodnotami. V případě odlišnosti porovnávaných hodnot by došlo k výpisu chybového hlášení na standardní chybový výstup. Upravené soubory k výše popsanému testování logických hradel, jednobitových komponent a 4, 8 bitových obvodů se nachází v přiložené složce `c_circuits_simulation_tests`. V ní se také vyskytuje skript `c_tests.sh`, napsaný v jazyce `bash`, který slouží k automatické kompilaci a následnému spuštění jednotlivých souborů, a jenž na standardní výstup vypisuje výsledek jednotlivých simulací.

K otestování očekávaného funkčního chování obvodů popsaných v jazyce Verilog byl využit nástroj Xilinx ISE Design Suite[1]. Prostředí ISE umožňuje nad reprezentacemi obvodů provádět mimo samotné simulační běhy také syntaktickou kontrolu jejich popisu a zobrazit jejich RTL schéma pro ověření odpovídající struktury popisovaného druhu obvodu. Pro samotný simulační běh bylo nutné k testovanému Verilog souboru vytvořit testovací soubor `Verilog Test Fixture`. Ten obsahuje testovací modul, v němž se vytvoří instance testovaného modulu UUT[2]. Následně se v testovacím modulu určí počáteční vstupní hodnoty, nastaví se jejich změny v průběhu simulace a pro debugovací účely se s využitím systémové

[1] Xilinx ISE (Integrated Synthesis Environment) Design Suite: https://www.xilinx.com/products/design-tools/ise-design-suite.html

[2] Unit under test – testovaná jednotka.

úlohy[3] `$monitor` vypíše obsah jednotlivých vstupů/výstupů do konzole v případě, že dojde ke změně některé z nich. Ke spuštění simulace slouží simulátor ISim[4], který obsahuje průběh a výsledky simulačního běhu. Popsané testování slouží k simulaci očekávaného chování vygenerovaných Verilog reprezentací před jejich logickou syntézou. Ukázku simulace dadda násobičky v prostředí ISim obsahuje obrázek 5.1.



Obrázek 5.1: Simulace chování Verilog reprezentace 8 bitové znaménkové dadda násobičky popsané v simulátoru ISim.

5.2 Formální verifikace Verilog, BLIF reprezentací

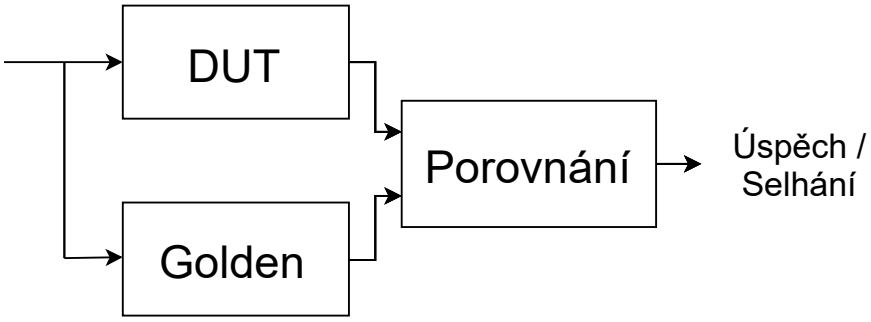
Jednotlivé výstupní reprezentace jsou generovány nezávisle na sobě a je tudíž potřeba ověřit jejich správnost. Samotné simulaci nad všemi vstupy v jazyce C nebo Verilogu může u netriviálních obvodů zabrat spoustu času k ověření správné funkčnosti. Ověření však lze dosáhnout i s využitím chytřejších heuristik. Typickým příkladem je převod na SAT[5] problém do CNF[6] formy a řešení pomocí pokročilého solveru.

K ověření byl využit nástroj *Yosys Open SYnthesis Suite*, který poskytuje řadu syntézních algoritmů zejména pro Verilog návrhy obvodů. Umožňuje ale také formálně verifikovat

[3] Úlohy používané ke generování vstupu a výstupu během simulace.
[4] ISE Simulator – vestavěný HDL simulátor používaný k simulaci FPGA návrhů v prostředí ISE.
[5] Problém splnitelnosti booleovské formule.
[6] Konjunktivní normální forma.

správnost návrhu a taktéž porovnat ekvivalenci vůči jiným reprezentacím popisu jako je např. BLIF.

Pomocí příkazů **equiv_*** byla kontrolována formální shodnost mezi Verilog a BLIF reprezentacemi pro různé architektury obvodů s užitím přímé kontroly splnitelnosti využitím SAT přístupu verifikace. Přiložený skript `yosys_equiv_check.sh`, napsaný v jazyce `bash`, umožňuje provést tuto formální shodnost mezi specifikovanými Verilog a BLIF reprezentacemi. Princip verifikace je ilustrován na obrázku 5.2. Spočívá v porovnání klíčových míst, ve kterých jsou oba modely předpokládány za logicky identické s ohledem na stejné vstupy. Pro tato místa je poté verifikováno, že se pro stejné vstupy u obou modelů získá stejný výstup. Toto porovnání bývá provedeno pomocí XOR hradel, jejichž výstupy jsou pomocí logického součtu (OR) sloučeny do jednoho výstupu, jehož splnitelnost značí to, že si obvody DUT a Golden neodpovídají. Verifikační nástroj musí převést tuto obvodovou reprezentaci do SAT konjuktivní normální formy, což je možné např. Tseytinovou transformací [23].



Obrázek 5.2: Princip formální verifikace mezi Design Under Test a referenčním/jiným Golden modelem.

5.3 Testování CGP reprezentací

Otestování generovaných CGP reprezentací bylo provedeno s využitím skriptu `chr2c.py`, jenž převádí vstupní *.chr* soubor do ploché reprezentace v jazyce C. U převedených reprezentací bylo simulací testováno jejich očekávané chování. Mimo testování je možné také pozorovat srovnání mezi ArithsGen přímo generovanými plochými reprezentacemi v jazyce C a těmi převedenými do nich z popisu CGP.

5.4 Logická syntéza a porovnání získaných parametrů

K porovnání různých architektur aritmetických obvodů bylo potřeba provést logickou syntézu pro ASIC[7] obvody nad jejich Verilog reprezentacemi. Ta byla provedena nad obvody s šířkou 8, 12, 16, 24 a 32 bitů s využitím nástroje Synopsys Design Compiler[8] pro 45nm FreePDK[9] technologickou knihovnu a získala řadu parametrů, pomocí nichž bylo možné různé architektury realizující stejnou činnost porovnat. Z výsledků logické syntézy máme např. informace o příkonu, zpoždění či ploše. Dalším významným parametrem je PDP[10],

[7] Application Specific Integrated Circuit.

[8] Synopsys Design Compiler: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html

[9] FreePDK: https://free-pdk.github.io/

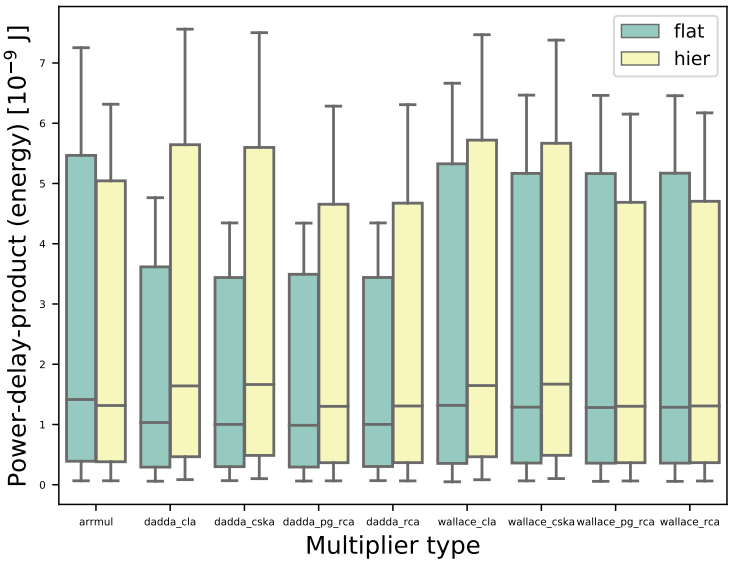
[10] Power-delay product – součin zpoždění-napájení.

který se získá vynásobením příkonu vůči zpoždění, a jenž udává energetickou spotřebu jedné operace v Joulech.

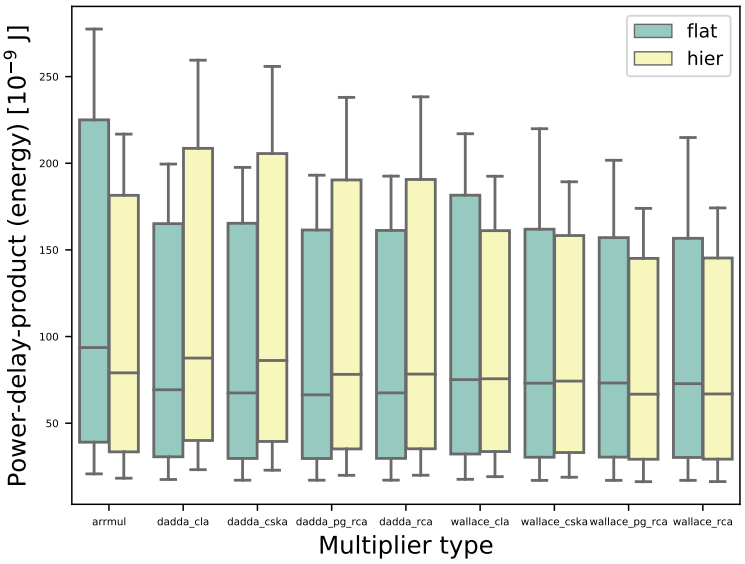
Logická syntéza mimo jiné provádí i optimalizaci popisů navržených nástrojem ArithsGen, sestávajících jen z dvouvstupých logických hradel, kdy některé části rozšiřuje nebo částečně i nahrazuje jinými členy z technologické knihovny jako jsou multiplexory, jednobitové sčítačky či vícevstupá hradla. Při syntéze však tyto optimalizace byly omezené, aby nedošlo k výrazné změně struktury obvodů.

5.4.1 Porovnání PDP u znaménkových násobiček

U menších a větších bitových šířek obvodů znaménkových násobiček byly pozorovány závislosti mezi jejich plochými a hierarchickými reprezentacemi vůči PDP. Porovnání obvodů s menšími bitovými šířkami je vidět na obrázku 5.3, zatímco obvody s vyššími bitovými šířkami jsou na obrázku 5.4. Z porovnání výsledků je patrné, že některé typy násobiček jsou vůči PDP efektivnější v ploché variantě a jiné zase v hierarchické. Například kombinační **array** násobičky si drží stejný trend, kdy je hierarchická reprezentace výhodnější napříč všemi bitovými šířkami, naopak je tomu pak u **dadda** násobiček, u kterých se jako výhodnější jeví spíše plochý popis. U **wallace** násobiček se zase hierarchické reprezentace stávají výhodnějšími až s vyššími bitovými šířkami jak ukazují jejich maximální hodnoty a taktéž hodnoty mediánů napříč oběma grafy. Dalším zjištěním bylo, že nehledě na zvolené reprezentaci jsou **dadda** násobičky u menších rozměrů výhodnější oproti zbylým typům. Např. u 12 bitových obvodů jsou až o 26.7 % výhodnější oproti druhým nejoptimálnějším **wallace** násobičkám. U větších obvodů jsou to ale právě **wallace** násobičky, které jsou tou nejlepší volbou pro sledovanou hodnotu PDP.



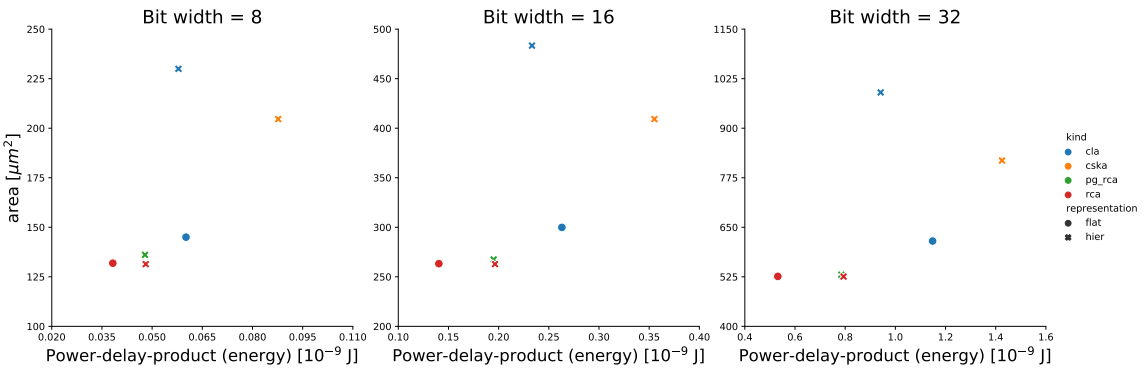
Obrázek 5.3: Porovnání PDP různých znaménkových násobiček s bitovými šířkami 4, 8, 12.



Obrázek 5.4: Porovnání PDP různých znaménkových násobiček s bitovými šířkami 16, 24, 32.

5.4.2 Srovnání znaménkových sčítaček

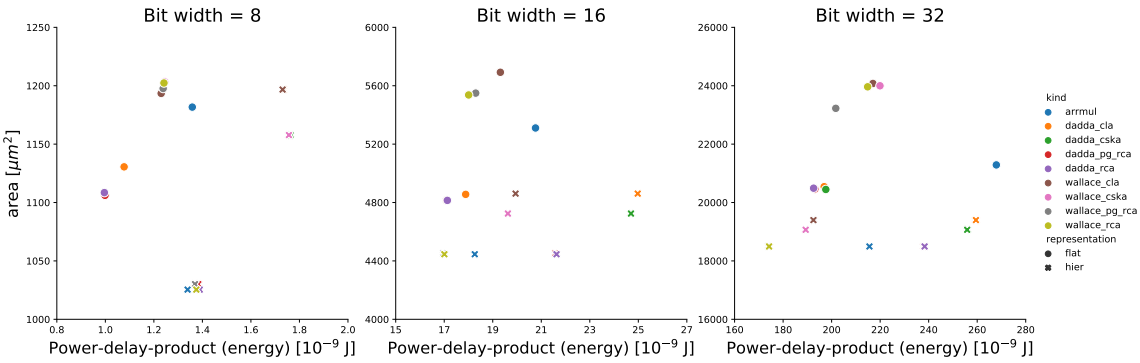
Srovnání PDP vůči zabrané ploše u různých architektur obvodů znaménkových sčítaček je k vidění na obrázku 5.5. Nutno podotknout, že ploché **RCA**, **PG_RCA** a **CSkA** sčítačky mají všechny shodné hodnoty. To může být způsobeno vlivem optimalizací ze strany syntézního nástroje, jelikož samotné vygenerované soubory popisující jednotlivé obvody jsou navzájem odlišné. Obecně se v porovnání plochy vůči PDP pro sledované bitové šířky zdají být nejoptimálnější sčítačky s postupným přenosem (**RCA**), avšak nemusí tomu tak být i u jiných parametrů. **RCA** sice zabírají menší plochu, jejich zpoždění je však vyšší než třeba u konstantního zpoždění **CLA** sčítaček nebo než zpoždění u **CSA**.



Obrázek 5.5: Porovnání PDP vůči zabrané ploše u plochých a hierarchických znaménkových sčítaček s bitovými šířkami 8, 16, 32.

5.4.3 Srovnání znaménkových násobiček

Se stejnými parametry byly srovnány taktéž znaménkové násobičky. Výsledky lze vidět na obrázku 5.6, kde jsou hierarchické reprezentace ve všech případech evidentně výhodnější s ohledem na velikost zabrané plochy. V případě hodnot PDP jsou však výsledky pro různé bitové šířky rozličné. Obecně se ploché násobičky jeví jako vhodnější volba pro popis menších obvodů, zatímco u větších se jednotlivé typy vyrovnávají. U větších bitových šířek jsou na tom zdaleka nejlépe **wallace** násobičky v hierarchické podobě, v těsném závěsu jsou pak **dadda** násobičky v obou reprezentacích.



Obrázek 5.6: Porovnání PDP vůči zabrané ploše u plochých a hierarchických znaménkových násobiček s bitovými šířkami 8, 16, 32.

5.4.4 Porovnání rychlosti generování a počtu řádků mezi popisy

Nepopíratelnou výhodou hierarchických reprezentací vůči těm plochým je obecně menší počet řádků potřebných pro popis stejné architektury obvodu. Ta se však začíná projevovat až s narůstající bitovou šířkou obvodů. Pro menší obvody jsou ploché reprezentace v tomto směru výhodnější. Vnitřní režie spojená s generováním a voláním jednotlivých funkčních bloků u hierarchických popisů se však nepříznivě podepisuje na době potřebné k jejich vygenerování. Porovnání rychlostí generování a počtu řádků mezi oběma způsoby popisu viz tabulka 5.3.

Tabulka 5.3: Srovnání doby generování Verilog souborů spolu s počty obsažených řádků mezi plochým a hierarchickým popisem.

	Ploché		**Hierarchické**	
Obvod	**Čas [ms]**	**Řádky**	**Čas [ms]**	**Řádky**
s_cla4	1.5	79	2	89
s_cla16	5	367	7	353
s_cla32	12.4	751	15.7	705
s_dadda_cla4	3.4	188	6.7	218
s_dadda_cla16	46.7	3212	73.8	1910
s_dadda_cla32	240.6	12620	365.1	6406

5.4.5 Shrnutí výsledků syntézy

Syntézou bylo porovnáno 324 obvodů rozličných typů. Jednalo se o znaménkové a bezznaménkové sčítačky, násobičky a bezznaménkové děličky s různou bitovou šířkou popsaných v ploché i hierarchické reprezentaci.

Ukázalo se, že v některých bitových šířkách a u některých architektur jsou pro určité sledované parametry výhodnější ploché reprezentace oproti těm hierarchickým a jinde zase naopak. Například pro hodnoty PDP jsou **dadda** násobičky výhodnější v plochém způsobu popisu pro všechny sledované bitové šířky, zatímco **wallace** násobičky jsou výhodnější v ploché variantě pro menší bitové šířky, ale u vyšších naměřených šířek se stávají výhodnější jejich hierarchické reprezentace. U naměřených hodnot zabírané plochy se dle očekávání v porovnání s ostatními sčítačkami objevila **CLA** velmi vysoko vzhledem k vysokému počtu logických hradel používaných k paralelnímu výpočtu P/G signálů. Zajímavé však je, že je to právě její hierarchická reprezentace, která má nejvyšší zabíranou plochu a nikoliv ta plochá. U násobiček je však dle očekávání zabíraná plocha hierarchických reprezentací menší v porovnání s plochými popisy a s narůstající bitovou šířkou obvodů se rozdíly navíc prohlubují.

Příčinou neočekávaných výsledků parametrů u některých reprezentací architektur mohou být nadměrné optimalizace ze strany syntézního nástroje, případně i volba parametrů se kterými byla syntéza spuštěna. Například ploché popisy obvodů **RCA**, **PG_RCA** či **CSkA** před syntézou nepředstavují stejnou architekturu sčítačky a po syntéze se mylně zdá, že ano.

Z analýzy výsledků bylo zjištěno, že vhodným zvolením obvodu můžeme např. pro 32 bitovou znaménkovou násobičku ušetřit až 23.2 % plochy, 31 % zpoždění nebo 33.2 % příkonu. Což může být přínosem pro hardwarové akcelerátory či aproximační syntézu.

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a implementovat snadno rozšiřitelný generátor aritmetických obvodů v jazyce Python, který umožňuje parametrizovatelné a automatizované generování jejich výstupních reprezentací sloužících k další práci s nimi.

Stěžejní částí bylo implementovat modularitu aritmetických obvodů do struktury programu, a tím tak docílit jednoduchého přidávání nových komponent a jednotného způsobu generování výstupních reprezentací. Dále bylo důležité pro jednotlivé druhy popisu ověřit správnost generovaných obvodů po syntaktické i funkční stránce. Významný přínos pro generování pak představuje vestavěná optimalizace na úrovni hradel.

Nad získanými obvody v jazyce Verilog byla provedena logická syntéza, která o popisovaných obvodech získala řadu parametrů, pomocí nichž bylo možné porovnat jednotlivé architektury. S pomocí tohoto nástroje může hardwarový návrhář využívat různé implementace aritmetických komponent, které se liší v jednotlivých parametrech, jako je příkon, rychlost či plocha. Dosažené výsledky pak mohou být přínosem pro HW akcelerátory. Dále také výstup může sloužit k efektivnější syntéze aproximačních (přibližných) obvodů, jelikož reprezentace obvodů a jejich vstup má významný vliv na kvalitu aproximace.

K výsledné podobě navrženého generátoru významným podílem přispěly též zkušenosti získané z účasti na studentské konferenci *Excel@FIT* 2021. Díky ní byla také získána zpětná vazba s návrhy na další možná rozšíření předkládané práce.

Možných rozšíření generátoru je celá řada. Od přidání dalších vnitřních optimalizací pro generování výstupních reprezentací obvodů až po jiná rozšíření jako je algoritmus Boothova překódování či podpora vícevstupých hradel. Větší důraz může být kladen na lepší parametrizovatelnost – například volba vstupního *Cin* pro sčítačku s postupným přenosem, kdy se použije blok úplné sčítačky místo poloviční. Další možností je také návrh grafického uživatelského rozhraní se zachováním cílené parametrizovatelnosti, ale také umožňující klíčovou automatizaci generování. Navíc by rozhraní mohlo např. vizualizovat generovanou CGP reprezentaci do 2D mřížky uzlů. V neposlední řadě lze nástroj doplnit o více architektur obvodů jako např. stromové sčítačky, případně i další výstupní reprezentace jako je např. jazyk VHDL. Navržený nástroj se po odladění a implementaci může využít jako open source aplikace určená zejména pro HW komunitu a vědeckou obec.

Literatura

[1] Zrůcký, J. *Booleova algebra a její aplikace* [[online]]. Brno, 2016. Bakalářská práce. Masarykova univerzita. Pedagogická fakulta. https://is.muni.cz/th/vsreg/.

[2] Ji, L. a Heuring, V. P. Impact of gate fan-in and fan-out limits on optoelectronic digital circuits. *Appl. Opt.* OSA. Jun 1997, sv. 36, č. 17, s. 3927–3940. DOI: 10.1364/AO.36.003927. Dostupné z: http://ao.osa.org/abstract.cfm?URI=ao-36-17-3927.

[3] Hussein, Q. Chapter 5: Combinational Logic Circuit. In:. Srpen 2020, s. 154–176. Dostupné z: https://www.researchgate.net/publication/343361502_chapter_5_Combinational_Logic_Circuit.

[4] N, Y. Design the 2X1 MUX with 2T Logic and Comparing the Power Dissipation and Area with Different Logics. *International Journal of Advanced Research in Electrical, Electronics and Instrumentation Engineering.* Březen 2015, sv. 04, s. 1284–1290. DOI: 10.15662/ijareeie.2015.0403016.

[5] Weste, N. a Harris, D. *CMOS VLSI Design: A Circuits and Systems Perspective.* 4th. USA: Addison-Wesley Publishing Company, 2010. 429-496 s. ISBN 0321547748.

[6] Mano, M. M. *Digital design : with an introduction to the Verilog HDL / M. Morris Mano, Michael D. Ciletti.* 5th ed., International ed. / contributions by B.R. Chandavarkar. Upper Saddle River, NJ: Pearson, 2013 - 2013. ISBN 9780273764526. Dostupné z: http://www.portcity.edu.bd/files/636444791235373856_Digitallogicdesign.pdf.

[7] A.C, S., T, Y., J., P. a A, R. A Proposed Wallace Tree Multiplier Using Full Adder and Half Adder. *IJIREEICE.* Květen 2016, sv. 4, s. 472–474. DOI: 10.17148/IJIREEICE.2016.45110.

[8] Kaushik, V. a Saini, H. Comparative Analysis of Proposed Parallel Digital Multiplier with Dadda and other Popular Multipliers. Červen 2017, sv. 4.

[9] Kaushik, V. a Saini, H. The Proposed Full-Dadda Multipliers. *Research & Reviews: Journal of Engineering and Technology.* 2018, sv. 7.

[10] Obermann, S. a Flynn, M. Division algorithms and implementations. *Computers, IEEE Transactions on.* Září 1997, sv. 46, s. 833 – 854. DOI: 10.1109/12.609274.

[11] Sayedsalehi, S., Rahimi Azghadi, M., Angizi, S. a Navi, K. Restoring and Non-Restoring Array Divider Designs in Quantum-dot Cellular Automata. *Information Sciences.* Srpen 2015, sv. 311, s. 86–101. DOI: 10.1016/j.ins.2015.03.030.

[12] Sekanina, L., Vašíček, Z., Růžička, R., Bidlo, M., Jaroš, J. et al. *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Academia, 2009. 328 s. Edice Gerstner. ISBN 978-80-200-1729-1. Dostupné z: https://www.fit.vut.cz/research/publication/9123.

[13] Wolf, C. *Yosys Open SYnthesis Suite* [[online]]. Dostupné z: http://www.clifford.at/yosys/.

[14] University of California. Berkeley Logic Interchange Format (BLIF). In:. 1992. Dostupné z: https://course.ece.cmu.edu/~ee760/760docs/blif.pdf.

[15] Končal, O. *Geometrické sémantické genetické programování* [[online]]. Brno, 2018. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií. http://hdl.handle.net/11012/84915.

[16] Mrazek, V., Vasicek, Z. a Hrbacek, R. Role of circuit representation in evolutionary design of energy-efficient approximate circuits. *IET Computers & Digital Techniques*. 2018, sv. 12, č. 4, s. 139–149. DOI: https://doi.org/10.1049/iet-cdt.2017.0188. Dostupné z: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/iet-cdt.2017.0188.

[17] Dvořáček, P. Evoluční návrh pro aproximaci obvodů. [[online]]. 2015. Dostupné z: http://excel.fit.vutbr.cz/submissions/2015/081/81.pdf.

[18] Watanabe, Y., Homma, N., Aoki, T. a Higuchi, T. Arithmetic module generator with algorithm optimization capability. *2008 IEEE International Symposium on Circuits and Systems*. 2008, s. 1796–1799.

[19] Sugawara, Y., Ueno, R., Homma, N. a Aoki, T. System for Automatic Generation of Parallel Multipliers over Galois Fields. In: *2015 IEEE International Symposium on Multiple-Valued Logic (ISMVL)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2015, s. 54–59. DOI: 10.1109/ISMVL.2015.15. ISSN 0195-623X. Dostupné z: https://doi.ieeecomputersociety.org/10.1109/ISMVL.2015.15.

[20] Homma Laboratory, RIEC, Tohoku University. *About Arithmetic Module Generator (AMG)* [[online]]. Homma Laboratory, RIEC, Tohoku University. https://www.ecsis.riec.tohoku.ac.jp/topics/amg/.

[21] Kulisz, J. a Mikucki, J. An IP-Core Generator for Circuits performing Arithmetic Multiplication. *IFAC Proceedings Volumes*. 2013, sv. 46, č. 28, s. 320–325. ISSN 1474-6670. 12th IFAC Conference on Programmable Devices and Embedded Systems. Dostupné z: https://www.sciencedirect.com/science/article/pii/S1474667015373468.

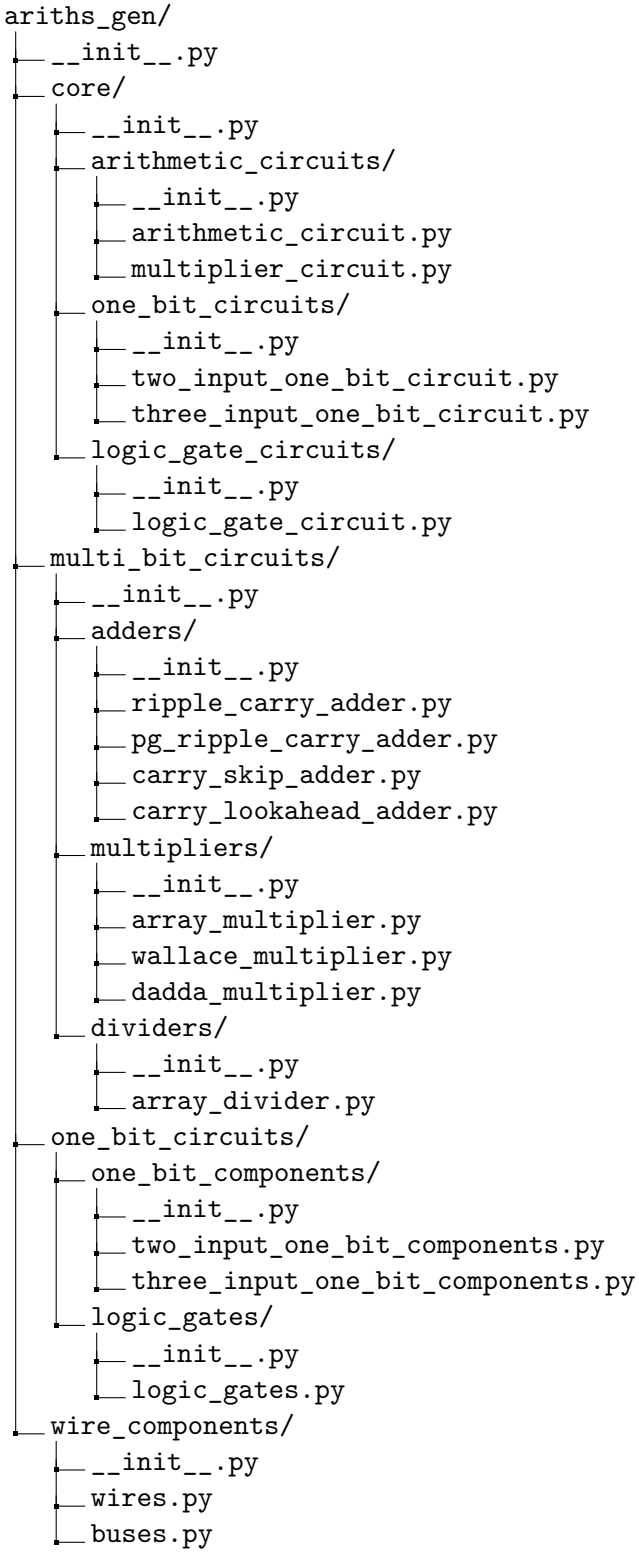
[22] Bolješik, M. *Generátor aritmetických obvodů* [[online]]. Brno, 2015. Bakalářská práce. Vysoké učení technické v Brně. Fakulta informačních technologií. http://hdl.handle.net/11012/52481.

[23] Tseytin, G. S. On the Complexity of Derivation in Propositional Calculus. In:. September 1966. Presented at the Leningrad Seminar on Mathematical Logic. Dostupné z: http://www.decision-procedures.org/handouts/Tseitin70.pdf.

Příloha A

Obsah přiloženého paměťového média

- source_codes – adresář obsahující zdrojové kódy a pomocné skripty generátoru
  - ariths_gen – balíček nejvyšší úrovně obsahující implementaci generátoru
  - c_circuits_simulation_tests – adresář obsahující simulace C obvodů se skriptem pro jejich otestování
  - documentation – adresář s vygenerovanou programovou dokumentací
  - generated_circuits – adresář s vygenerovanými výstupními reprezentacemi obvodů
  - ariths_gen.py – skript pro generování obvodů a jejich výstupních reprezentací
  - chr2c.py – skript pro převod CGP do C k otestování funkčnosti
  - yosys_equiv_check.sh – pomocný skript sloužící k formální shodnosti Verilog a BLIF reprezentací
  - README.md – textový soubor popisující nástroj ArithsGen spolu s příklady jeho použití
- bachelor's_thesis – adresář obsahující zdrojové kódy technické zprávy v LaTeXu a její vysázenou verzi ve formátu *pdf*



Obrázek A.1: Struktura balíčku nejvyšší úrovně *ariths_gen* implementující nástroj Ariths-Gen.