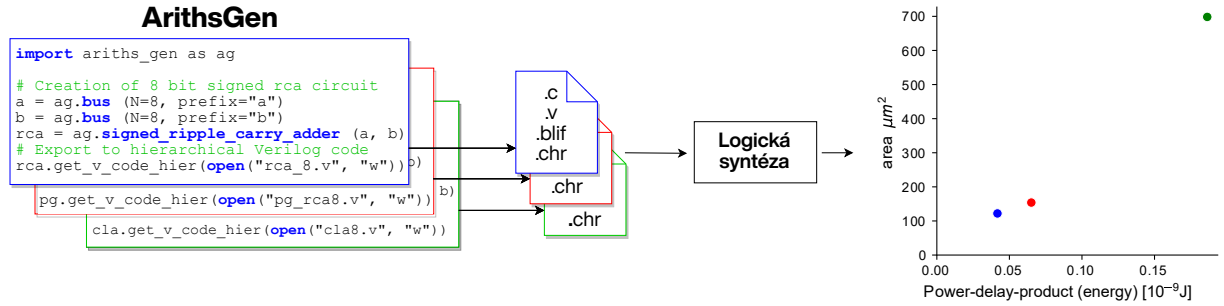
BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2021
http://excel.fit.vutbr.cz

ArithsGen: Generátor aritmetických obvodů pro HW akcelerátory

Jan Klhůfek*



Abstrakt

Aritmetické obvody jsou nedílnou součástí dnešních procesorů. Konkrétně je zde najdeme uvnitř aritmeticko-logické jednotky (ALU), jednotky pro generování adres (AGU) či v matematickém koprocesoru (FPU). Mimo procesor se ale taktéž 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.

Tato práce představuje open source nástroj, implementovaný v jazyce Python, schopný generovat aritmetické obvody a exportovat je do různých reprezentací popisu. Výstupní reprezentace pak slouží ke snadnému ověření funkčnosti navrženého obvodu (C), k popisu hardwaru a logické syntéze (Verilog), k formální verifikaci (BLIF) či ke globální optimalizaci obvodu s využitím evoluční strategie (CGP).

Klíčová slova: Generátor — Aritmetické obvody — Python — Plochý popis — Hierarchický popis — C — Verilog — BLIF — Kartézské Genetické Programování (CGP)

Přiložené materiály: Downloadable Code — Generated Circuits

*xklhuf01@stud.fit.vutbr.cz, *Fakulta informačních technologií, Vysoké učení technické v Brně*

1. Úvod

Vytváření popisu vnitřní struktury složitějších obvodů 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.

Aritmetické obvody jsou obvody provádějící aritmetické operace jako je sčítání, násobení, apod. Tyto obvody se skládají z menších funkčních bloků realizujících určité bitové operace. Nejmenšími funkčními bloky jsou logická hradla. Hlavním cílem bylo využití modularity aritmetických obvodů k dosažení jednotného generování výstupních reprezentací těchto obvodů bez ohledu na jejich vnitřní strukturu a nabídnout toto řešení komunitě.

V minulosti na tuto problematiku vznikla již řada podobných generátorů, některé jsou dnes už zastaralé, jiné naopak představují teoretický podklad k řešení, avšak implementace samotná již veřejná není, příklady takových prací jsou [1, 2]. 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 [3] využívá grafové reprezentace obvodů pomocí Arithmetic Circuit Graph (ACG) k formálnímu popisu, verifikaci a převodu do hierarchické reprezentace v HDL[1] 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 [4] 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 [5] 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[2]. Problém je v lineární vnitřní struktuře generovaných obvodů, která není hierarchická a značně ztěžuje možnost nástroj rozšířit o další výstupní reprezentace týchž obvodů.

V této práci představuji nový nástroj zvaný ArithsGen, který umožňuje navrhovat libovolné aritmetické obvody. Je psaný v jazyce Python a s výhodou využívá principů objektového programování. V základní implementaci podporuje generování 1 druhu děličky (bezznaménkové), 8 druhů sčítaček (bez/znaménkových), a až 18 druhů násobiček (bez/znaménkových, využívající definované sčítačky), je však jednoduše rozšiřitelný o další typy obvodů. Implementace je založena na RTL úrovni abstrakce a používá dvouvstupá logická hradla. Hlavní princip generování obvodů je postaven na jejich modularitě. Modularita spočívá v postupném skládání obvodů z menších komponent, poskytující tak různé úrovně hierarchického zanoření.

[1] Jazyky pro popis hardware (Verilog, VHDL).

[2] https://www.concept.de/SpiceVision.html

Generovaný obvod tak přímo reflektuje návrh schématu obvodu, čímž je docíleno přímočařejšímu přepisu popisu obvodu z navrženého schématu do jeho struktury v programu. Systém je rozšiřitelný o další obvody a výstupní reprezentace a umožňuje lepší ladění jednotlivých částí. S pomocí tohoto nástroje může hardwarový návrhář využívat různé implementace aritmetických komponent, které se liší v jednotlivých parametrech, jako je příkon, rychlost či plocha. Dále výstup může sloužit k efektivnější syntéze aproximačních (přibližných) obvodů, jelikož reprezentace obvodů a jejich vstup má významný vliv na kvalitu aproximace. O aproximačních obvodech se můžete dozvědět více např. zde [6]. Navržený nástroj plánujeme po odladění a implementaci poskytnout jako open source aplikaci určenou zejména pro HW komunitu a vědeckou obec.

2. Struktura aritmetických obvodů

Aritmetické obvody představují hardwarové součástky, složené na nejnižší úrovni z logických hradel, které jsou schopné realizace aritmetických operací v binární podobě. Patří sem zejména obvody sčítaček, odčítaček, násobiček a děliček. Pro každý ze zmíněných typů aritmetických obvodů existuje řada různých architektur lišící se svou vnitřní strukturou, tzn. způsobem poskládání a propojení jednotlivých hradel. Přesnější specifikace architektur je k nalezení například v knize [7]. Různé obvody tak sice mohou provádět stejnou operaci, např. sčítání, ale přitom se lišit příkonem, plochou nebo zpožděním. Právě tyto tři klíčové parametry slouží k porovnání různých architektur a ke zvolení takové, která k danému účelu použití nejvíce vyhovuje.

2.1 Výstupní reprezentace 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[3].

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

BLIF[4] Formát určený pro jednodušší strojové zpracování a používaný zejména k formální verifikaci obvodů.

[3] Aritmetické operace jsou implementované přímo na CPU a využíváme k nim aritmetické operátory. V tomto případě však reprezentace v C slouží k otestování funkčnosti obvodu, případně k zanesení chyb aproximací. Dojde tedy ke zpomalení simulace, která je však rychlejší než simulace na úrovni RTL.

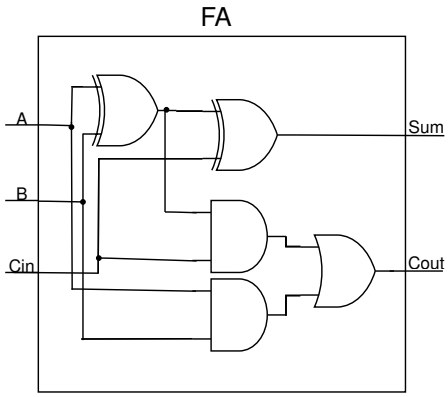
[4] Berkeley Logic Interchange Format.

CGP[5] Specializovaný formát popsaný ve formě celočíselného netlistu a používaný jako chromozom pro evoluční optimalizaci [8] (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, což také navržený nástroj umožňuje.

2.2 Plochý popis

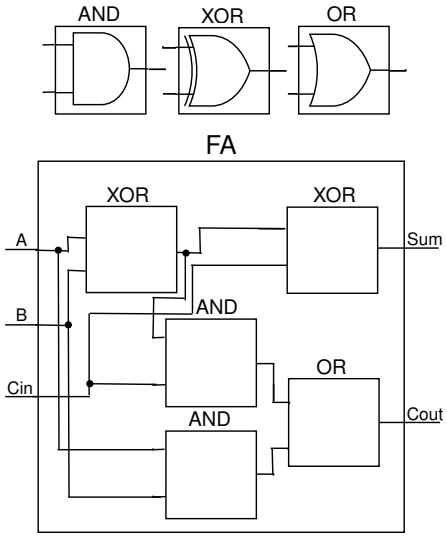
Plochý znamená, že je obvod popsán na úrovni nejnižších komponent, tj. logických hradel. Vzniklé reprezentace tak mohou být velké vzhledem k nutnosti namapovat všechny interní vodiče a hradla. Zploštění seskupuje veškerou funkční logiku do jediné komponenty a umožňuje provádět globální optimalizace např. pomocí kartézského genetického programování (CGP). Ukázku schématu plochého obvodu najdete na obrázku 1.



Obrázek 1. Plochá reprezentace úplné jednobitové sčítačky.

2.3 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 2.



Obrázek 2. Hierarchická reprezentace úplné jednobitové sčítačky.

3. Nástroj ArithsGen

ArithsGen modeluje s využitím možností jazyka Python 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. Navíc díky systému dědičnosti jazyka je nástroj lépe rozšiřitelný.

3.1 Vnitřní struktura obvodu

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.

Vodič představuje jednobitový spoj sloužící k propojení jednotlivých komponent mezi sebou. Uchovává o sobě jméno a případně i index, na kterém se nachází ve vícebitové sběrnici. Sběrnice pak sestává z několika vodičů dle zvolené bitové šířky specifikované při její instanciaci.

Nejmenšími implementovanými funkčními součástkami jsou dvouvstupá logická hradla a jednovstupý invertor. Dvouvstupá hradla na vstupu očekávají dva vodiče, se kterými provedou příslušnou bitovou operaci a výsledek vrátí prostřednictvím výstupního vodiče. V případě jednovstupého invertoru se pak očekává pouze jeden vstup a jeho invertovaný vodič je vyveden na výstup.

[5] Kartézské Genetické Programování.

Konstruktoru třídy obvodu se předají parametry **a** a **b** reprezentující jeho vstupní rozhraní (vodiče u jednobitových, sběrnice u vícebitových obvodů). Uvnitř konstruktoru se dle typu obvodu a bitových šířek vstupů určí bitová šířka výstupní sběrnice. Při skládání obvodů (kromě samostatných hradel) se postupně vkládají do seznamu **components** objekty reprezentující menší podkomponenty. Seznam umožňuje přistoupit k již přidaným komponentám k docílení propojení výstupů předchozích komponent na vstupy následujících. Nakonec se připojí výstupní vodiče vytvořených podkomponent na příslušné bitové pozice výstupní sběrnice. Na obrázku 3 je ukázka implementace bezznaménkové sčítačky s postupným přenosem.

```
class unsigned_ripple_carry_adder(arithmetic_circuit):
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("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 = half_adder(a=self.a.get_wire(input_index),
                b=self.b.get_wire(input_index),
                prefix=self.prefix+"_ha")
        # Rest adders are full adders
        else:
            obj_adder = full_adder(a=self.a.get_wire(input_index),
                b=self.b.get_wire(input_index),
                c=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 3. Implementace bezznaménkové sčítačky s postupným přenosem.

K vygenerování výstupu zvoleného aritmetického obvodu je třeba nejprve instanciovat jemu náležející třídu se zvolenými vstupními parametry. Volenými parametry jsou vstupní sběrnice s vlastními názvy a bitovými šířkami a cílový název, který identifikuje vytvářený obvod. Nad vytvořenou instancí obvodu, jenž představuje reprezentaci obvodu uvnitř programu, stačí zavolat metodu k vygenerování výstupu do kýžené reprezentace popisu.

3.2 Vnitřní optimalizace

ArithsGen dovede na úrovni hradel rozpoznat, když má některý ze vstupních vodičů konstantní logickou hodnotu a v takovém případě zjednodušit strukturu generovaného hradla dle zbývajících možných kombinací vstupních hodnot. Vztahy mezi vstupy se dají vyvodit z pravdivostní tabulky patřící danému hradlu. Ve výsledku se pak některá hradla vůbec negenerují a jen se zajistí propojení vnitřně určeného vodiče k dalším komponentám.

3.3 Export do 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.

3.4 Princip exportu do plochých reprezentací

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 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.

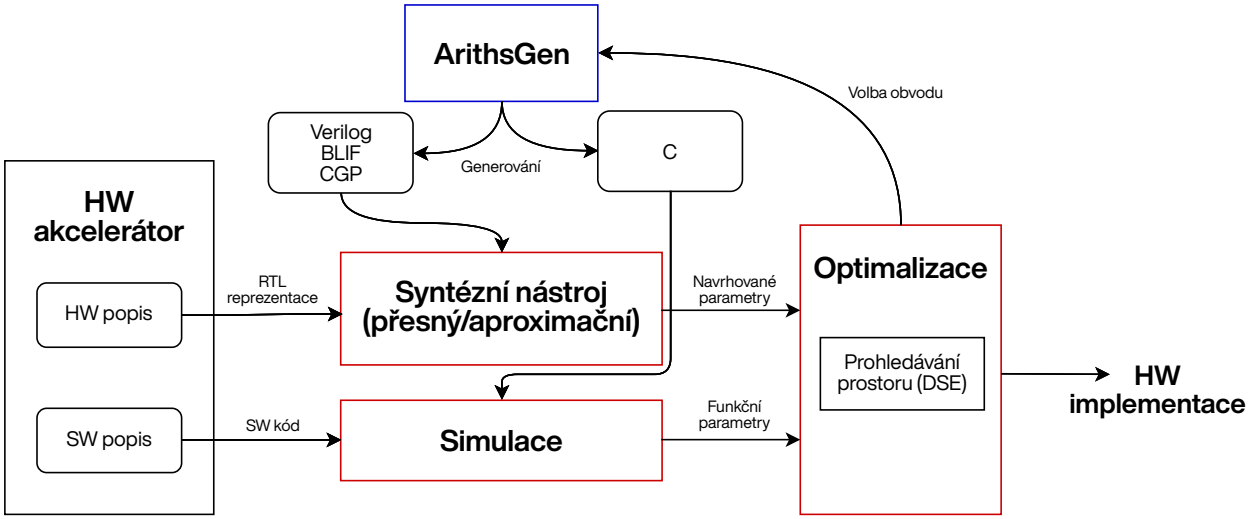
3.5 Princip exportu do hierarchických reprezentací

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.

3.6 Zasazení ArithsGen do kontextu návrhu HW akcelerátorů

Na obrázku 4 je znázorněno využití nástroje ArithsGen v praxi se zasazením do metodiky návrhu HW akcelerátoru. HW návrhář volí vstupní sledované parametry (např. příkon) pro popisovaný obvod. Tato práce pak věnuje pozornost modře znázorněnému bloku, reprezentující generátor, jenž v celkovém návrhu slouží



Obrázek 4. Metodika návrhu HW akcelerátoru s využitím ArithsGen.

návrháři k vhodné volbě obvodu a následnému generování do zvolené reprezentace popisu. Dále je nad parametry získanými ze syntézy/simulace provedena analýza pomocí DSE[6], která slouží k nalezení nejoptimálnějšího řešení.

4. Formální verifikace vygenerovaných obvodů

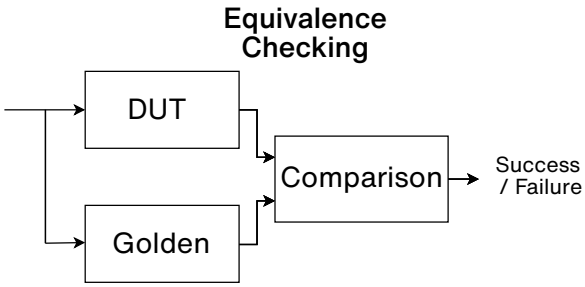
Jednotlivé výstupní reprezentace jsou generovány nezávisle na sobě a je tudíž potřeba ověřit jejich správnost. Samotné simulaci nad všemi vstupy v jazyce C nebo Verilogu může u netriviálních obvodů zabrat spoustu času k ověření správné funkčnosti. Ověření však lze dosáhnout i s využitím chytřejších heuristik. Typickým příkladem je převod na SAT[7] problém do CNF[8] formy a řešení pomocí pokročilého solveru.

K ověření byl využit nástroj Yosys open synthesis suite [9], který poskytuje řadu syntézních algoritmů zejména pro Verilog návrhy obvodů. Umožňuje ale také formálně verifikovat správnost návrhu a taktéž porovnat ekvivalenci vůči jiným reprezentacím popisu jako je např. BLIF.

Pomocí příkazů **equiv_*** bylo dosaženo formální shodnosti mezi Verilog a BLIF reprezentacemi s užitím přímé kontroly splnitelnosti využitím SAT přístupu verifikace. Princip verifikace je ilustrován na obrázku 5.

5. Dosažené výsledky

K porovnání různých architektur aritmetických obvodů bylo potřeba provést logickou syntézu pro ASIC[9] ob-



Obrázek 5. Princip formální verifikace mezi Design Under Test a referenčním / jiným Golden modelem.

vody nad jejich Verilog reprezentacemi. Ta byla provedena nad obvody s šířkou 8, 12, 16, 24 a 32 bitů s využitím nástroje Synopsys Design Compiler[10] pro 45nm FreePDK[11] technologickou knihovnu a získala nám 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[12], který získáme vynásobením příkonu vůči zpoždění, a jenž nám 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.

[6] Design space exploration.
[7] Problém splnitelnosti booleovské formule.
[8] Konjunktivní normální forma.
[9] Application Specific Integrated Circuit.
[10] Synopsys Design Compiler: https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/design-compiler-nxt.html
[11] FreePDK: https://free-pdk.github.io/
[12] Power-delay product – součin zpoždění-napájení.

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 8, zatímco obvody s vyššími bitovými šířkami jsou na obrázku 9. Z porovnání výsledků je patrné, že některé typy násobiček jsou vůči PDP efektivnější v ploché variantě a jiné zase v hierarchické. Například **array**[13] násobičky si drží stejný trend, kdy je hierarchická reprezentace výhodnější napříč všemi bitovými šířkami, naopak je tomu pak u **dadda**[14] násobiček, u kterých se jako výhodnější jeví spíše plochý popis. U **wallace**[15] 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.

Nepopíratelnou výhodou hierarchických reprezentací vůči těm plochým je, že čím větší je zvolená bitová šířka pro daný obvod, tím se počet vygenerovaných řádků snižuje. 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 reprezentacemi viz tabulka 1.

Tabulka 1. Srovnání doby generování Verilog souborů spolu s počty obsažených řádků mezi plochým a hierarchickým popisem.

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

Srovnání PDP vůči zabrané ploše u různých architektur obvodů znaménkových sčítaček je k vidění na obrázku 6. Nutno podotknout, že ploché **RCA**[16], **PG_RCA**[17] a **CSkA**[18] sčítačky mají všechny shodné hodnoty. To může být způsobeno vlivem optimalizací ze strany syntézního nástroje, jelikož samotné vygenerované soubory popisující jednotlivé obvody jsou navzájem rozličné. 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**[19] sčítaček nebo než zpoždění u **CSkA**.

Se stejnými parametry byly srovnány taktéž znaménkové násobičky. Výsledky lze vidět na obrázku 7, 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.

6. Závěry

Cílem bylo vytvořit snadno rozšiřitelný open source generátor aritmetických obvodů pro HW komunitu, který umožňuje parametrizovatelné a automatizované generování výstupních reprezentací aritmetických obvodů k další práci s nimi.

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.

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.

Nástroj lze rozšířit o více architektur obvodů, případně i další reprezentace jako je např. jazyk VHDL. Do budoucna je uvažována vestavěná podpora k lepší optimalizaci generovaných reprezentací a přidání Boothova překódování.

[13] Kombinační (array) násobička.

[14] Dadda násobička: https://en.wikipedia.org/wiki/Dadda_multiplier

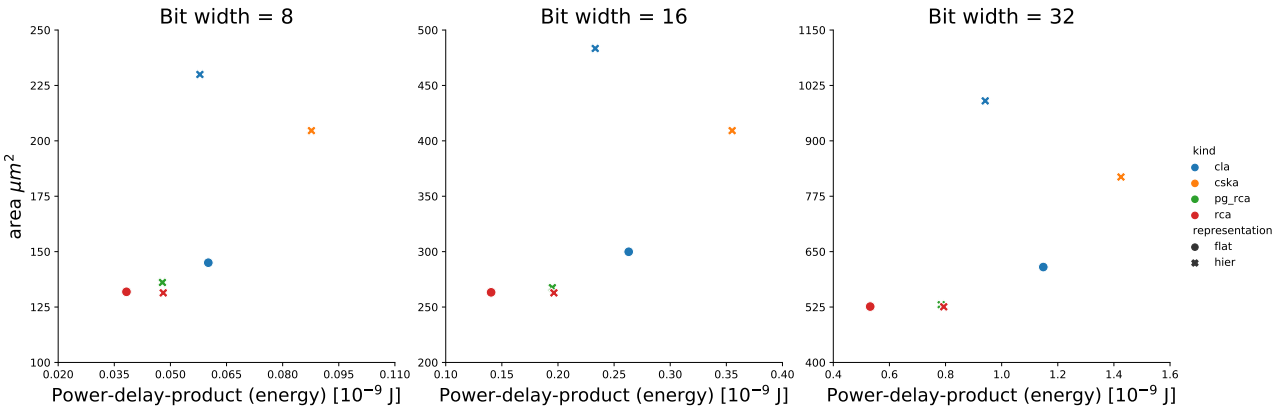
[15] Wallaceova násobička: https://en.wikipedia.org/wiki/Wallace_tree

[16] Ripple-carry adder – sčítačka s postupným přenosem.

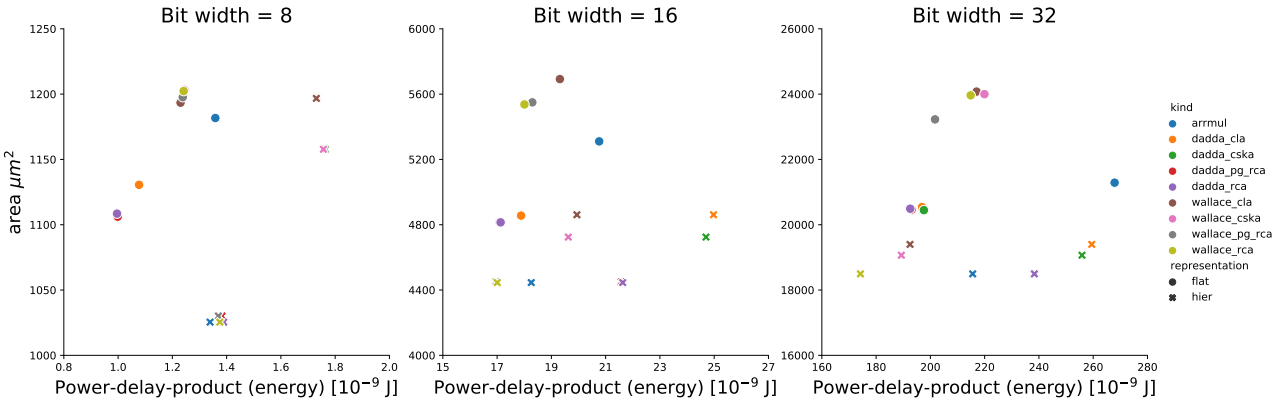
[17] Propagate/generate ripple-carry adder – sčítačka s postupným přenosem využívající propagate/generate logiku.

[18] Carry-skip/bypass adder – sčítačka s přeskočením přenosu.

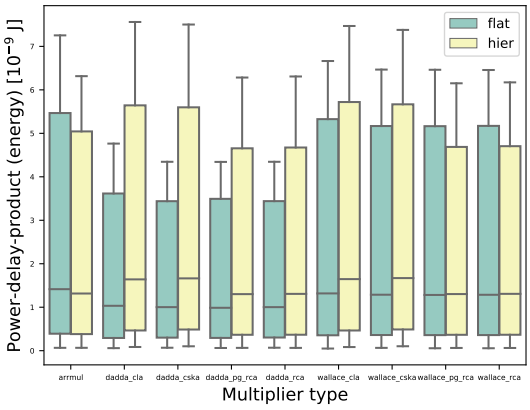
[19] Carry-lookahead adder – sčítačka s predikcí přenosu.



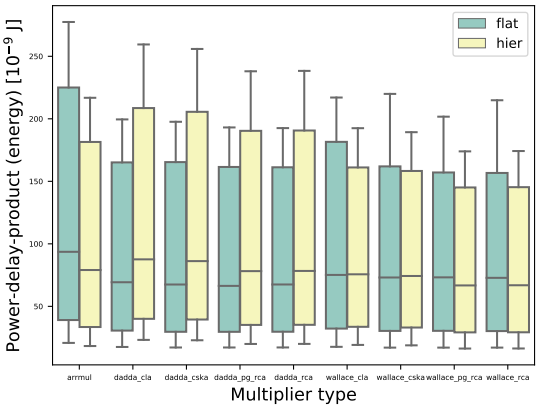
Obrázek 6. Porovnání PDP vůči zabrané ploše u plochých a hierarchických znaménkových sčítaček s bitovými šířkami 8, 16, 32.



Obrázek 7. Porovnání PDP vůči zabrané ploše u plochých a hierarchických znaménkových násobiček s bitovými šířkami 8, 16, 32.



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



Obrázek 9. Porovnání PDP různých znaménkových násobiček s bitovými šířkami 16, 24, 32.

Poděkování

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

Literatura

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

[2] Y. Sugawara, R. Ueno, N. Homma, and T. Aoki. System for automatic generation of parallel mul-

tipliers over galois fields. In *2015 IEEE International Symposium on Multiple-Valued Logic (ISMVL)*, pages 54–59, Los Alamitos, CA, USA, may 2015. IEEE Computer Society.

[3] About arithmetic module generator (amg). [online]. https://www.ecsis.riec.tohoku.ac.jp/topics/amg/.

[4] Józef Kulisz and Jerzy Mikucki. An ip-core generator for circuits performing arithmetic multiplication. *IFAC Proceedings Volumes*, 46(28):320–325, 2013. 12th IFAC Conference on Programmable Devices and Embedded Systems.

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

[6] Vojtech Mrazek, Zdenek Vasicek, and Radek Hrbacek. Role of circuit representation in evolutionary design of energy-efficient approximate circuits. *IET Computers & Digital Techniques*, 12(4):139–149, 2018.

[7] Neil Weste and David Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley Publishing Company, USA, 4th edition, 2010.

[8] Lukáš Sekanina, Zdeněk Vašíček, Richard Růžička, Michal Bidlo, Jiří Jaroš, and Petr Švenda. *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Edice Gerstner. Academia, 2009.

[9] Clifford Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.