Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátové toky ako paradigma pre paralelné programovanie

Diplomová práca

2018 **Bc. Peter Kuljovský**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Dátové toky ako paradigma pre paralelné programovanie

Diplomová práca

Študijný program: Aplikovaná informatika
Študijný odbor: 2511 Aplikovaná informatika
Školiace pracovisko: Katedra Aplikovanej Informatiky
Školiteľ: Mgr. Pavel Petrovič, PhD.

2018 **Bc. Peter Kuljovský**

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Peter Kuljovský
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Dátové toky ako paradigma pre paralelné programovanie
Dataflow as the Parallel Programming Paradigm

Cieľ: Éra von Neumannovskej architektúry po 70 rokoch definitívne skončila. Nachádzame sa v ére paralelných výpočtov, ktorá však prináša základný problém: tradičné postupy zapisovania kódu sú náchylné na množstvo chýb. Riešenie poskytujú jazyky založené na modelovaní dátových tokov. Takto organizovaný kód sa dá automaticky paralelizovať a napriek tomu aj pre človeka zrozumiteľne a prirodzene zapisovať a navrhovať. Zabehnutým príkladom je programovací jazyk LabView spoločnosti National Instruments. Úlohou študenta bude prehľad existujúcich jazykov založených na modelovaní dátových tokov, návrh aj implementácia vlastného jednoduchého programovacieho jazyka založeného na dátových tokoch a empirické kvalitatívne aj kvantitatívne porovnanie paralelizácie programov zapísaných v tradičnom procedurálnom jazyku s paralelnými nadstavbami a v navrhnutom jazyku. Predpokladá sa aj vytvorenie kompilátora navrhnutého jazyka do nejakej paralelnej platformy - buď FPGA alebo GPGPU.

Literatúra: Casey Weltzin: Using Dataflow as a Solution to the Multicore Programming Challenge, Conference Procecedings of ESC 2010.
Mojo - FPGA Tutorials, dostupné online: embeddedmicro.com/tutorials/mojo/
LabVIEW Communications System Design Suite 2.0 Manual, dostupné online: www.ni.com/documentation/en/labview-comms/2.0/

Vedúci: Mgr. Pavel Petrovič, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Dátum zadania: 03.10.2016

Dátum schválenia: 17.10.2016

prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

....
študent

....
vedúci práce

Čestné vyhlásenie

Čestne prehlasujem, že som túto diplomovú prácu vypracoval samostatne s použitím uvedených zdrojov.

V Bratislave dňa 3.5.2018 ....

Poďakovanie

Ďakujem školiteľovi mojej diplomovej práce Mgr. Pavlovi Petrovičovi, PhD. za jeho čas, pripomienky a cenné rady.

Abstrakt

V práci sme navrhli jednoduchý jazyk založený na dátových tokoch, ktorý sme pomenovali DataWolf a tiež sme implementovali grafické vývojové prostredie s kompilátorom do platformy FPGA. Program v jazyku DataWolf sa prekladá do jazyka Lucid, ktorý je určený na programovanie práve tejto platformy. V našom jazyku sme naprogramovali niekoľko jednoduchých príkladov na ukážku jeho využitia. Tiež aj jeden zložitejší algoritmus, ktorý je vo veľkej miere využívaný najmä v kryptografii. Ide o hašovaciu funkciu SHA-2, ktorú sa nám podarilo s využitím nášho jazyka sparalelizovať na najvyššiu možnú mieru. Potenciál pre ďalšie využitie tohto jazyka vidíme v rozšírení knižnice príkazov, ako aj možností vývojového prostredia, ktoré umožnia ešte rýchlejší vývoj ešte zložitejších algoritmov.

Motiváciou k práci bol problematický zápis paralelných programov v štandardných programovacích jazykoch. Náš jazyk tento problém nemá, pretože program napísaný v modeli dátových tokov je úplne prirodzene paralelizovateľný. Navyše sme predviedli ako je možné dokázať, že je jazyk DataWolf univerzálny a teda má rovnakú výpočtovú silu ako Turingov stroj.

Kľúčové slová: dátové toky, grafický programovací jazyk, paralelné programovanie, FPGA, SHA-2

Abstract

In this work we designed simple dataflow language, which we have named DataWolf and we also implemented graphical development environment with compiler to FPGA platform. Program in DataWolf language is translated to Lucid language, which is dedicated to programming this platform. In our language we programmed several simple examples of how to use it. And also one more complex algorithm, which is widely used in cryptography. It is hash function SHA-2, which we have been able to parallelize with our language to the highest possible degree. The potential for further use of this language can be seen in the command library extensions as well as the development environment that will allow even faster development of even more complex algorithms.

The motivation to work was that writing parallel programs in standard programming languages is problematic. In our language it is much easier because the program written in the data flow model is completely natural to parallelize. In addition, we have demonstrated how it is possible to prove that DataWolf is universal and therefore has the same computational power as the Turing machine.

Keywords: dataflow, graphical programming language, parallel computing, FPGA, SHA-2

Obsah

Úvod **1**

1 Východiská **3**

1.1 Model dátových tokov . . . 3

1.1.1 Riadenie behu dát . . . 4

1.1.2 Teoretické spôsoby implementácie . . . 5

1.1.3 Prvé hardvérové architektúry . . . 5

1.2 Programovacie jazyky založené na dátových tokoch . . . 6

1.2.1 Vlastnosti jazyka . . . 6

1.2.2 Historický vývoj . . . 7

1.2.3 Java streams . . . 7

1.3 FPGA . . . 8

1.3.1 História . . . 9

1.3.2 Použitie . . . 9

1.3.3 Programovacie jazyky . . . 9

1.4 Použité technológie . . . 10

2 Návrh jazyka **11**

2.1 Filozofia jazyka . . . 11

2.2 Syntax . . . 12

2.2.1 Konvencie jazyka . . . 13

2.2.2 Moduly . . . 13

2.2.3 Špeciálne príkazy . . . 14

2.2.4 Operácie . . . 15

2.2.5 Informácie o prepojení operácií . . . 19

2.2.6 Príklady použitia . . . 19

2.3 Univerzálnosť jazyka . . . 25

3 Implementácia **26**
3.1 Triedy reprezentujúce operácie a ich prepojenia 26
3.1.1 Bodka 27
3.1.2 Čiara 27
3.1.3 Info 28
3.1.4 ObjektInfo 29
3.1.5 Objekt 29
3.2 Vývojové prostredie 31
3.2.1 Vstupy 31
3.2.2 Užívateľské rozhranie 33
3.2.3 Výstupy 35
3.3 Kompilátor 36
3.3.1 Spôsob kompilácie 36
3.3.2 Dôležité metódy 37

4 Použiteľnosť a testovanie **39**
4.1 Efektívnosť programovania 39
4.2 Algoritmus SHA-2 40
4.2.1 Konštanty 40
4.2.2 Predvýpočet 40
4.2.3 Pseudokód algoritmu 40
4.3 Implementácia SHA-2 v jazyku DataWolf 42
4.3.1 Optimalizácia algoritmu 42
4.3.2 Program 43
4.3.3 Komunikácia s programom 44
4.4 Porovnanie výkonu s inými jazykmi 44
4.4.1 Rýchlosť výpočtu v jazyku DataWolf 46
4.4.2 Rýchlosť výpočtu na iných platformách 48
4.5 Ďalšie možné úpravy implementácie algoritmu 49
4.5.1 Odstránenie podmienkových blokov 49
4.5.2 Rozšírenie výpočtu pre dlhšie správy 49

5 Možnosti do budúcnosti **51**
5.1 Rozšírenie knižnice jazyka 51
5.2 Vynechanie Mojo IDE z vývojového procesu 52
5.3 Rozšírenie možností vývojového prostredia 52
5.4 Debugovanie 53

Záver **55**

Literatúra **56**

Prílohy **58**

Úvod

Paralelné programovanie je s nami už dlho, no v súčasnosti sa mu venuje čoraz viac pozornosti. Jeho zmysel je urýchlenie výpočtov. Toto zrýchlenie môže byť pri niektorých úlohách veľmi razantné, a preto má rozhodne cenu sa ním zaoberať. Aj pri neustálom zvyšovaní výkonu hardvéru sú totiž rýchlosti výpočtov nedostatočné alebo až príliš energeticky náročné. Konkrétne môže ísť napríklad o spracovanie enormného množstva dát, rozpoznávanie obrazu v počítačovom videní alebo aj o kryptografiu.

Okrem štandardných viacjadrových procesorov sa paralelné výpočty môžu robiť aj na špeciálnych platformách ako je GPGPU alebo FPGA. Tieto platformy v poslednom období zaznamenali veľký rozmach a stávajú sa stredobodom pozornosti paralelného programovania. Preto je dôležité pre tieto technológie vyvinúť nové programovacie jazyky, ktoré budú ľahko naučiteľné, a tak podporia ich ďalší rozvoj. V tejto práci sa budeme zaoberať platformou FPGA.

FPGA sa čoraz častejšie objavujú napríklad v rôznych dátových centrách. FPGA dovoľuje prispôsobenie digitálnych obvodov pre danú aplikáciu. A práve toto z nich robí efektívne riešenie z hľadiska šetrenia výpočtových zdrojov a spotreby elektrickej energie. Táto spotreba býva rádovo menšia v porovnaní s bežne používanými procesormi a grafickými čipmi. Výhodou FPGA technológie je (na rozdiel od konkurenčnej ASIC), že navrhnuté obvody sa môžu kedykoľvek zmeniť. Napríklad, keď je nutné pozmeniť alebo vylepšiť logiku výpočtov [1].

Pri písaní paralelného programu v štandardnom programovacom jazyku je veľmi jednoduché urobiť chybu a celkovo je takéto programovanie veľmi náročné. Preto je vhodné použitie takzvaných dátových tokov. Takéto programovanie je špecifické tým, že program je zapísaný ako orientovaný graf dát prúdiacich medzi operáciami. Takto napísaný program je úplne prirodzene paralelizovateľný. V práci si uvedieme krátky prehľad problematiky a nejaké už existujúce riešenia. Ďalej bude našou úlohou vytvoriť jednoduchý jazyk založený na dátových tokoch a kompilátor, ktorý program napísaný v našom jazyku preloží do platformy FPGA.

V prvej kapitole si povieme o tom, čo sú to vlastne dátové toky. Uvedieme

príklady niektorých jazykov na nich založených a čím by sa každý správny takýto jazyk mal vyznačovať. Bude tu niečo málo o technológii FPGA a na konci kapitoly si rozoberieme aké všetky technológie budeme v práci využívať. V druhej kapitole nás čaká návrh jazyka. Vysvetlíme si ako sme pri tom postupovali a prečo sme zvolili dané riešenia. Popíšeme si aj syntax nášho jazyka a jednotlivé príkazy, ktoré obsahuje. Kapitola Implementácia bude rozoberať implementáciu nášho vývojového prostredia a kompilátora do FPGA platformy. Bude tu popis kľúčových použitých tried a aj užívateľského rozhrania programu. V štvrtej kapitole zhodnotíme ako sa programuje v našom jazyku. Ten otestujeme aj naprogramovaním nejakého algoritmu vhodného na paralelizáciu a porovnaním našich výsledkov s inými jazykmi a platformami. Na záver v poslednej kapitole práce uvedieme, čo ešte sa dá v budúcnosti v jazyku, respektíve v našom vývojovom prostredí, ešte zlepšiť a ako toho docieliť.

Kapitola 1

Východiská

V tejto kapitole si popíšeme prehľad teórie a technológií, ktoré budeme využívať v práci. Bude tu prehľad existujúcich programovacích jazykov založených na dátových tokoch. Zoznámime sa aj s technológiou FPGA.

1.1 Model dátových tokov

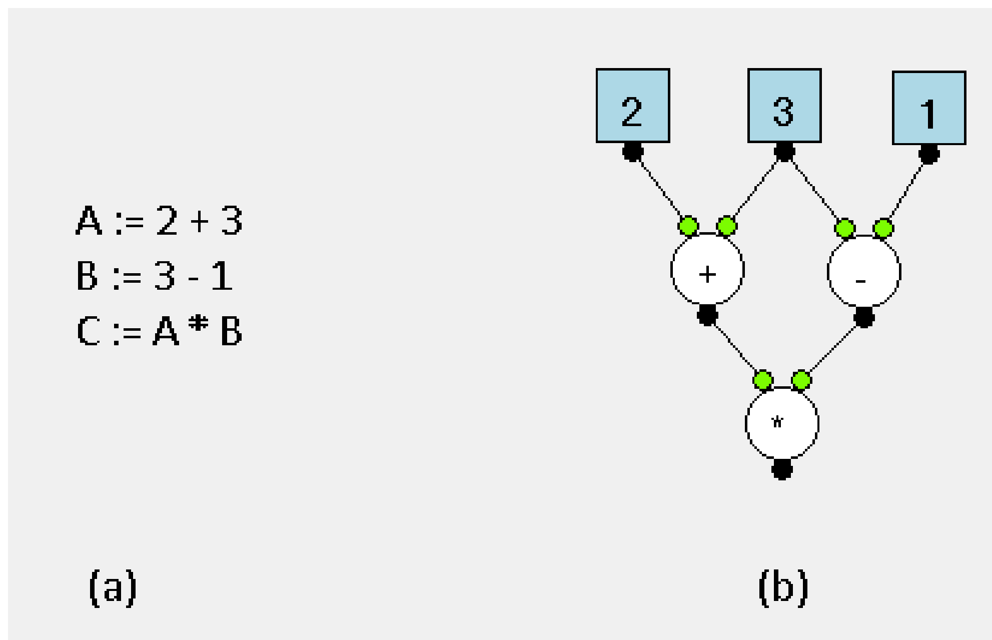
V tomto modeli je program reprezentovaný orientovaným grafom. Vrcholy tohto grafu tvoria rôzne operácie a hrany označujú pohyb dát medzi nimi. Každá operácia má jeden alebo viac vstupov. Akonáhle dostane vstupné dáta, tak je vykonaná a pošle výsledok na výstup. V prípade, že viac operácií dostane na svoj vstup všetky potrebné vstupy, tak môžeme výpočet sparalelizovať [2].

Na obrázku 1.1 môžeme vidieť porovnanie klasického von Neumannového modelu (a) a jeho ekvivalent v modeli dátových tokov (b). V prvom programe dôjde postupne k sčítaniu konštánt 2 a 3, potom k odčítaniu konštánt 3 a 1 a nakoniec k vynásobeniu výsledkov prvých dvoch krokov. Takže program trvá tri časové jednotky.

Rovnaký program napísaný v dátových tokoch má na začiatku operácie sčítania a odčítania so všetkými potrebnými vstupmi, takže tieto operácie prebehnú paralelne. Po vykonaní týchto operácií získa všetky potrebné vstupy operácia násobenia, ktorá môže byť následne vykonaná. Tento program potrebuje na dokončenie len dve časové jednotky.

Ďalšou dôležitou vlastnosťou tohto modelu je to, že takto zapísaný program v podstate môže vytvárať vlastný modul so svojimi vstupmi a výstupmi. Takže je ho možné vložiť do grafu zložitejšieho programu.

Vo von Neumannovom modeli sa naraz vykonáva iba jedna elementárna operácia, zatiaľ čo pri dátových tokoch týchto operácií môžu byť aj tisíce. No štandardný hardvér



Obr. 1.1: Jednoduchý program napísaný vo von Neumannovom modeli (a) a jeho ekvivalent v modeli dátových tokov (b)

takého niečoho nie je schopný, a preto bolo treba prísť s niečím novým. Ďalej v tejto sekcii sa pozrieme aj na to ako sa k problematike hardvéru prispôsobeného pre model dátových tokov pristupovalo v posledných desaťročiach.

1.1.1 Riadenie behu dát

Na obrázku 1.1 (b) sme mohli vidieť, že jedny dáta sú posielané viacerým operáciám. Pri tomto jave sú dáta duplikované a poslané každej operácii, ktorá tieto dáta potrebuje. Týmto je zachovaná nezávislosť dát a funkčnosť systému. Aby sa zachovala determinovateľnosť modelu, nie je dovolené svojvoľné zlučovanie dát. Na tento účel slúžia špeciálne riadiace operácie zvané brány [3].

Zlučovacia brána sa dá prirovnať k príkazu if-then-else vo von Neumannovom modeli. Máme tri vstupy. Jeden je riadiaci vstup nesúci boolovskú hodnotu Pravdivý alebo Nepravdivý a vstupy A a B. V prípade, že dáta, ktoré prídu na riadiaci vstup sú typu Pravdivý, tak na výstup sa pošlú dáta zo vstupu A. Inak sa použijú údaje zo vstupu B.

Ďalej máme prepínaciu bránu. Na rozdiel od zlučovacej brány tu je okrem riadiaceho vstupu len jeden dátový. Dáta z neho sa pošlú na základe hodnoty v riadiacom vstupe buď na výstup A alebo B.

1.1.2 Teoretické spôsoby implementácie

Pôvodne boli dáta chápané ako neaktívne prvky, ktoré tak ostávajú až do doby, kým nie sú prečítané. Neskôr sa stalo bežným javom, že dáta riadia beh programu. Existujú dva prístupy ako toto teoreticky implementovať.

Dátovo riadená architektúra

Beh programu záleží od dostupnosti dát. Operácia zostáva neaktívnou dovtedy, kým dáta prichádzajú na vstupy. Keď sú dostupné všetky dáta, zariadenie notifikuje a spustí danú operáciu.

Požiadavkami riadená architekúra

Pri tomto prístupe sa operácie aktivujú jedine vtedy, keď dostanú požiadavku na dáta od výstupu. Následne operácia pošle požiadavky na dáta svojim vstupom. Operácia sa vykoná až keď dostane všetky vstupné dáta. Na rozdiel od predchádzajúceho prístupu tento spôsob má kroky navyše. No na druhej strane umožňuje zjednodušenie programu odstránením niektorých operácií.

1.1.3 Prvé hardvérové architektúry

Aj keď dátové toky vyzerajú v teórii dobre, tak v praktickej implementácii nachádzame mnoho problémov. Prvým je, že model dátových tokov počíta s neobmedzenou pamäťou, čo nie je v praxi uskutočniteľné. Ďalším problémom je predpoklad, že ľubovoľné množstvo príkazov môže bežať naraz paralelne, no opäť v praxi máme len limitovaný počet výpočtových prvkov. Kvôli tomu vznikli architektúry, ktoré nie úplne presne reflektujú čistý model dátových tokov.

Statická architektúra

V tejto architektúre každá hrana grafu môže uchovávať maximálne jeden dátový token. Takže akonáhle sú na všetkých vstupoch pripravené dáta a na výstupe žiadne dáta nie sú, tak sa spustí daný výpočet. Na implementáciu tejto architekúry sa do grafu pridávali spätné hrany, ktoré obsahovali potvrdzovací token. Výhodou tejto architektúry bola jej jednoduchá a rýchla implementácia. Takisto už počas kompilácie mohla byť alokovaná všetka pamäť, pretože každá hrana grafu mohla udržiavať nula alebo práve jeden dátový token. Nevýhodou statickej architektúry bolo spomalenie výpočtu kvôli pridaniu hrán s potvrdzovacími tokenami. Toto tiež negatívne ovplyvnilo paralelizáciu [2].

Dynamická architektúra

Alternatívny prístup spočíval v dovolení opakovaného používania podgrafov. Do pamäti sa však uložila iba jedna kópia tohto grafu a na rozlíšenie tokenov, ktoré patrili rôznym volaniam sa použili tagy, ktorým sa bežne hovorilo ako farba tokenu. Miesto iba jedného dátového tokenu na hrane grafu, táto architektúra dovoľovala na hrany umiestniť ľubovoľné množstvo tokenov, pričom každý mal svoj špecifický tag. Keďže tieto tokeny na hrane neboli usporiadané, tak mohli byť spracované aj v inom poradí v akom sa na hranu dostali. Tagy však zabezpečovali, aby nedošlo ku konfliktom. Nevýhodou dynamickej architektúry bola jej veľká komplexnosť a oveľa väčšie nároky na pamäť ako pri statickej architektúre. Výhodou bol vyšší výkon a väčšie možnosti paralelizácie [2].

1.2 Programovacie jazyky založené na dátových tokoch

Prvé takéto jazyky boli textové a nie grafické. Či už z dôvodu hardvérovej náročnosti práce s takýmto typom jazyka alebo zdĺhavého zápisu niektorých komponentov jazyka ako sú cykly alebo dátové štruktúry [4].

1.2.1 Vlastnosti jazyka

Najdôležitejšie charakteristiky, ktorými by sa mal každý takýto jazyk vyznačovať:

1. Jedno priradenie do premennej
2. Nezáleží na poradí príkazov
3. Žiadne vedľajšie efekty

Jedno priradenie do premennej

Pretože riadenie programu závisí od dát, tak je dôležité, aby sa hodnoty premenných nemenili medzi definíciou a použitím. Aby sa toho dosiahlo, tak sa znemožňuje opätovné priradenie do premennej po prvom priradení. Takže ich možno považovať skôr za hodnoty ako premenné.

Nezáleží na poradí príkazov

Každý príkaz môže byť v programe umiestnený v ľubovoľnom poradí. Výnimkou sú napríklad cykly.

Žiadne vedľajšie efekty

To znamená žiadne globálne premenné alebo možnosť modifikácie parametrov funkcií. Dá sa toho docieliť dodržiavaním pravidla 1. Problém nastáva v prípade práce s dátovými štruktúrami. Toto by sa dalo vyriešiť tak, že sa na každú modifikáciu poľa budeme pozerať ako na nové pole.

1.2.2 Historický vývoj

Obdobie textových jazykov

Tento typ jazykov zaznamenal najväčší rozmach v 70. a na začiatku 80. rokov. Prvým takýmto jazykom bol jazyk TDFL (Textual Data-Flow Language). Ďalej nasledovali napríklad jazyky LAU, Lucid, Id, LAPSE, VAL a mnoho ďalších. Okrem toho boli vytvorené kompilátory niektorých štandardných jazykov do modelu dátových tokov. Napríklad išlo o jazyky Fortran, Pascal alebo niektoré dialekty jazyka C. Koncept dátovych tokov sa nepresadil hlavne kvôli neschopnosti vytvoriť efektívny hardvér podporujúci dátove toky.

Pokusy o hybridné modely

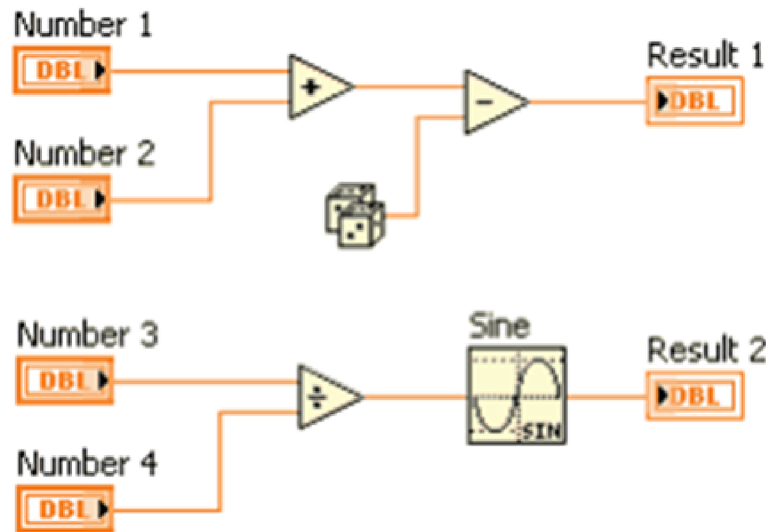
V 90. rokoch boli snahy o skombinovanie von Neumannového modelu a modelu dátových tokov. Podľa [5] je optimálna cesta niekde uprostred. Čiže použiť von Neumannovú architekúru s pár prvkami dátových tokov alebo naopak.

Nástup grafických jazykov

Vizuálne znázornenie dátových tokov prinieslo mnoho výhod. Program znázornený vo forme grafu bol tak oveľa lepšie čitateľnejší. Niektoré projekty takýmto spôsobom využívajúce dátové toky zaznamenali aj komerčný úspech. Najznámejším je v tomto ohľade asi systém LabView [6]. Na 1.3 možno vidieť ukážku toho ako tento systém vyzerá. Inšpirovali sme sa ním pri tvorbe grafického prostredia pre náš jazyk.

1.2.3 Java streams

V jave verzia 8 pribudli streamy, ktoré sa dajú chápať ako určitá forma dátových tokov. Je to abstrakcia, ktorá reprezentuje postupnosť rôznych sekvenčných alebo paralelných operácií. Nie je to dátová štruktúra, ktorá uchováva nejaké dáta. Ide len o sprostredkúvanie elementov zo zdroja cez pipeliny výpočtových operácií. Dôležitým je, že tieto streamové operácie nemodifikujú vstupné dáta. Pipeliny streamu pozostávajú



Obr. 1.2: Jednoduchý program napísaný v prostredí LabView.

zo zdroja, žiadnej alebo viacerých *intermediate* operácií a jednej *terminal* operácie, ktorá stream ukončí. Jednotlivé prvky môžu byť v streame navštívené iba raz. Zdrojom streamu môžu byť polia, kolekcie alebo aj nejaký I/O kanál. Vyhodnocovanie operácií prebieha s využitím tzv. "lazy evaluation". Čiže výpočet prebehne, až keď je na nejakom mieste programu potrebný výsledok. Toto môže prispieť k rôznym optimalizáciám. Ako sme spomenuli na začiatku, operácie na streamoch môžu prebiehať aj paralelne, čo sa dá zadefinovať pri vytváraní príslušného streamu [7].

Vďaka spomenutým vlastnostiam sa tieto streamy stali dobrou inšpiráciou pre tvorbu nášho jazyka. Na 1.3 môžeme vidieť príklad použitia streamov v jazyku Java.

```
Arrays.stream(new String[] {"1", "20", "3", "40"})
    .mapToInt(Integer::parseInt)
    .filter(x -> x > 10)
    .sum()
```

Obr. 1.3: Jednoduchý program napísaný v jazyku Java s využitím streamov.

1.3 FPGA

Field Programmable Gate Array (FPGA) alebo inak programovateľné hradlové pole je typ logického integrovaného obvodu vyrobeného tak, aby mohol byť naprogramovaný u zákazníka. Obsahuje pole programovateľných logických obvodov a umožňuje ich navzájom prepojiť.

1.3.1 História

Prvé FPGA boli vytvorené v 80. rokoch firmami Altera a Xilinx. Časom sa k samotnému FPGA začal pridávať vstavaný mikroprocesor a súvisiace periférne zariadenia. Pôvodné FPGA obsahovali v začiatkoch niekoľko tisíc hradiel. V súčasnosti sa tieto čísla pohybujú v desiatkách miliónov. Hlavnými výrobcami týchto zariadení sú naďalej firmy Altera (od roku 2015 patrí pod Intel) a Xilinx.

1.3.2 Použitie

V počiatkoch boli používané v oblasti telekomunikácií. Neskôr sa dostali do spotrebiteľských a priemyselných aplikácií. Môže sa z nich vytvoriť prakticky ľubovoľné numerické zariadenie. Napríklad veľké FPGA umožňujú dnes aj implementáciu komplikovaných procesorov.

Ďalším príkladom využitia je urýchliť vysokovýkonné, výpočtovo náročné systémy. V súčasnosti sa FPGA riešenia pre tieto systémy stávajú čoraz populárnejšie. Príkladom je firma Microsoft, ktorá testovala použitie FPGA na algoritme vyhľadávača Bing. Dosiahli skoro dvojnásobné zvýšenie výkonu [8].

Testovaná bola tiež výmena grafických procesorov za FPGA v neurónovej sieti slúžiacej na analyzovanie obrázkov. Použité boli zariadenia Altera Arria 10 a NVIDIA Tesla K40. Samostatné FPGA malo horší výkon ako použité GPU, lebo dokázalo analyzovať len 233 obrázkov za sekundu v porovnaní s 500 až 824 obrázkami u grafického procesora. No elektrická spotreba bola u FPGA až desaťkrát menšia. Vzhľadom na to, že FPGA sa dajú spolu skladať, tak bolo možné dosiahnuť rovnakého výkonu ako u GPU s použitím troch FPGA a len tretinovou spotrebou elektrickej energie [8].

1.3.3 Programovacie jazyky

Na konfiguráciu FPGA sa používa hardware description language (HDL). Najznámejšie sú VHDL alebo Verilog.

VHDL

Jazyk VHDL je navrhnutý tak, aby podporoval všetky úrovne abstrakcie používané pre návrh takých obvodov: umožňuje popísať obvod na hradlovej alebo algoritmickej úrovni. Je použiteľný aj pre návrh analogových obvodov. Programovací jazyk VHDL je silno typovaný. Umožňuje popísať aj paralelizmus.

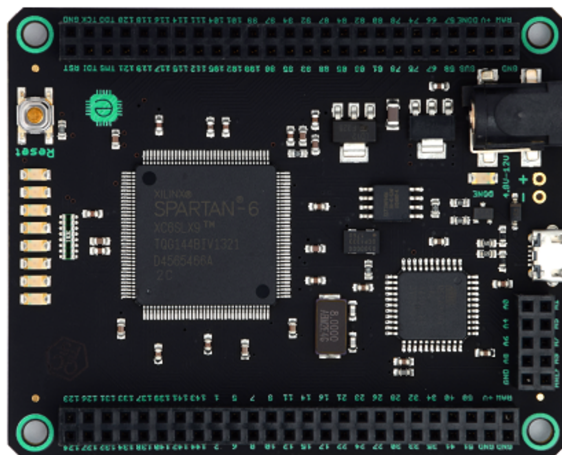
Verilog

Pri navrhovaní syntaxe tohto jazyka slúžil ako vzor jazyk C, kvôli jeho popularite a rozšírenosti. Má podobne ako jazyk C preprocesor a rovnaké hlavné kľúčové slová. Má tiež podobný mechanizmus formátovania a prioritu operátorov. Líši sa napríklad tým, že pri deklarácii premenných je nutné udať veľkosť v bitoch. Ďalej nemá zložené dátové typy alebo rekurzívne podprogramy [9].

1.4 Použité technológie

Implementačná časť diplomovej práce bude napísaná v programovacom jazyku C#, pričom budeme využívať vývojové prostredie SharpDevelop.

Výstup kompilátora nášho jazyka bude v jazyku Lucid. No nejde o jazyk založený na modeli dátových tokov, ktorý sme spomenuli v predchádzajúcej časti. Ide o dialekt Verilogu, ktorý bol vyvinutý firmou Embedded Micro [10]. Tá zostavila zariadenie, na ktorom sme robili testovanie 1.4. Keďže jazyk Lucid je odvodený od Verilogu, tak sa mu veľmi podobá, no je vhodnejší pre začiatočníkov nakoľko sa viac podobá bežnejším programovacím jazykom.



Obr. 1.4: Použité FPGA riešenie od firmy Embedded Micro. Obsahuje FPGA čip Spartan-6, ktorého výrobca je firma Xilinx.

Vygenerovaný program v Lucide sa musí ďalej spracovávať. Na to využívame jednoduché prostredie Mojo IDE (opäť od firmy Embedded Micro). To preloží náš program do Verilogu, ktorý sa ďalej spracováva pomocou softvéru ISE Design Suite. Tento softvér vyvíja firma Xilinx, ktorá je aj výrobcom FPGA čipu Spartan-6, ktorý je umiestnený na našom zariadení.

Kapitola 2

Návrh jazyka

Jedným z cieľov našej práce bolo aj navrhnúť nový jazyk založený na modeli dátových tokov. Tento jazyk sme pomenovali ako DataWolf. Meno vzniklo z anglického slovného spojenia data flow obrátením písmen v druhom slove. V rámci práce sme pracovali aj na kompilátore do platformy FPGA. Vybrali sme si preto jazyk Lucid, ktorý je užívateľsky prívetivejšou verziou známejšieho Verilogu.

2.1 Filozofia jazyka

Počas vývoja sme postupne pracovali s viacerými verziami jazyka, respektíve spôsobu jeho kompilácie. Kým sme sa pustili do zostavovania nášho jazyka, tak sme sa najskôr museli oboznámiť s jazykom Lucid. Testovaním a analyzovaním rôznych prístupov sme nakoniec dospeli k finálnej podobe.

1. verzia - Prácu sme začali myšlienkou programu preloženého do klasického von Neumanového modelu. Čiže máme jeden alebo viac procesorov, registre a vykonávame program uložený v pamäti. Pre naše zámery masívnej paralelizácie sa tento model ukázal ako neefektívny, pretože na veľký počet plnohodnotných procesorov sme spotrebovali príliš veľa zdrojov. Takisto kompilácia z nášho jazyka do Lucidu by bola veľmi náročná.

2. verzia - Pokračovali sme teda s obmenenou verziou, kde sme pracovali s nie plnohodnotnými procesormi, ale s veľkým počtom úzko špecializovaných procesorov. Tie by boli generované presne podľa požiadaviek programátorovho programu. Takže keby sme potrebovali naraz urobiť sto sčítaní, tak by sme si vygenerovali sto malých procesorov, ktoré dokážu urobiť len operáciu sčítania. Od tejto metódy sme však opäť pre jej komplikovanosť upustili.

3. verzia - Ponechali sme si myšlienku malých úzko špecializovaných procesorov, no trocha sme ju upravili. Nejde už ani tak o procesory, ale len o jednoduché moduly, ktoré majú nejaké vstupy a nejaké výstupy. Toto už pripomína aj model dátových tokov, čo nám umožní oveľa jednoduchšiu kompiláciu z nášho jazyka. V tejto verzii však ešte program napísaný v našom jazyku stále fungoval klasickým spôsobom. Mal nejaký začiatok a koniec. Tento fakt výrazne komplikoval kompiláciu, respektíve množinu programov, ktoré sme mohli v našom jazyku napísať.

4. verzia - V poslednej a finálnej verzii sme náš jazyk prispôsobili jazyku Lucid. Program nemá nejaký začiatok a koniec. Ide len o sadu operácií, ktoré sa vykonávajú pri daných podmienkach v každom cykle procesora. Táto zmena myslenia nám jednak výrazne zjednodušila kompiláciu a navyše nám umožnila písanie zložitejších programov než sme pôvodne zamýšľali. Jedinou nevýhodou je, že programátor si na toto nezvyklé myslenie musí zvyknúť.

Pri vytváraní nášho jazyka sme sa snažili dosiahnuť vlastnosti jazykov založených na dátových tokov. To sa nám do určitej miery podarilo. Náš jazyk nemá žiadne vedľajšie efekty, pretože nemá žiadne globálne premenné a nie je možné modifikovať parametre modulov. Takisto nezáleží na poradí operácií. Beh programu riadia dáta.

Ďalšiu vlastnosť, ktorá hovorila o tom, že nie je možné do premennej priradiť viackrát hodnotu sme dodržali iba z časti. Pôvodne sme v jazyku žiadne premenné nechceli, ale pri zmene filozofie jazyka do finálnej podoby nastala nutná potreba spôsobu uchovania dát medzi jednotlivými cyklami procesora. Na toto sme využili registre, ktoré by sa dali nazvať aj slovom premenné.

2.2 Syntax

Aj keď náš jazyk je hlavne grafický, tak potrebuje textovú reprezentáciu. Táto nie je primárne určená pre programátora, ale pre naše vývojové prostredie. Preto dôležitým prvkom pri zostavovaní syntaxe nebola čitateľnosť. Pôvodná verzia jazyka bola ľahko čitateľná a nebolo veľmi komplikované programovať v textovom režime. Základná operácia sa skladala z troch častí: ID operácie, meno operácie a vstupy. S ďalším vývojom a pridávaním funkcionalít sa tento zápis ukázal ako nepoužiteľný. Problém bol v poslednej časti - vstupy. Hoci všetky vstavené operácie majú len jeden výstup, tak v jazyku sme umožnili vytvoriť vlastné moduly s viacerými výstupmi. Preto nestačí pri operácii uviesť identifikátory jeho vstupov, pretože ten nemusí byť jednoznačný.

Tento problém sám o sebe by sa dal vyriešiť, ale do vývojového prostredia sme navyše ešte pridali možnosť spájať operácie pomocou vlastných ciest. Čiže už nešlo len o jednoduché úsečky ale o zložitejšie lomené čiary. Ich zápis sa úplne osamostatnil od samotných operácií, takže vzniklo duplikovanie informácií. Z tohto dôvodu má aktuálna podoba nášho jazyka tvar základných operácií vo forme: ID operácie a meno operácie. Pričom ešte pridávame informáciu o umiestnení v grafickej ploche alebo niektoré ďalšie parametre, ktoré si rozoberieme u jednotlivých operácií.

2.2.1 Konvencie jazyka

Zatiaľ čo vývojové prostredie bude, rovnako ako samotná písomná práca, vytvorené v slovenskom jazyku, tak kľúčové slová jazyka DataWolf sme navrhli v anglickom jazyku. To zabezpečí jednoduchšiu prácu so zdrojovými súbormi vďaka tomu, že sa nebudeme musieť zaoberať so špecifickými znakmi slovenskej abecedy.

Identifikátory jednotlivých operácií je nutné písať bez diakritiky a bez medzier. Je možné používať písmená malej a veľkej anglickej abecedy ako aj čísla. Mená nemôžu začínať veľkými písmenami a číslami a viacslovné pomenovania je možné oddeliť pomocou podčiarkovníka. Ďalšie symboly, ktoré pozná jazyk sú všetky tri druhy zátvoriek, ktorých význam si preberieme pri operáciách, ktorých sa týkajú. Nakoniec je tu ešte symbol #, ktorý slúži na oddeľovanie dodatočných parametrov za menom operácie.

Pred tým ako si podrobne popíšeme jednotlivé operácie, je nutné ešte vysvetliť si použitú terminológiu. Všetky dáta, s ktorými pracujeme sú nejaké polia bitov. Veľkosť tohto poľa označujeme v práci ako bitová dĺžka. Napríklad pokiaľ chceme mať dáta rozmerov klasického integeru, tak bude táto bitová dĺžka mať hodnotu 32. Ďalšou vlastnosťou dát je bitová šírka. V prípade, že je väčšia ako jedna, tak už ide o pole polí. Čiže pokiaľ máme bitovú šírku 10 a bitovú dĺžku 32, tak je to desaťprvkové pole integerov. V práci často používame pojem *boolovská hodnota*. Ide o hodnotu, ktorej bitová šírka a dĺžka je jedna.

2.2.2 Moduly

Moduly, ako sme jednotlivé podprogramy nazvali, sú hlavnou štruktúrnou jednotkou jazyka. Každý program obsahuje modul *main*, v ktorom sa zabezpečuje vonkajšia komunikácia, ktorá prebieha po sériovom porte. Zápis modulu začína kľúčovým slovom *def*, za ktorým nasleduje jej meno. Potom sme použili hranaté zátvorky [a] na označenie tela modulu. Toto telo má 3 časti, ktoré musia byť v danom poradí. Avšak

operácie v rámci týchto celkov môžu už byť napísané ľubovoľne poprehadzované, pretože náš jazyk je založený na modeli dátových tokoch. Beh programu riadia dáta a nie poradie v akom sa jednotlivé operácie napíšu. Toto menšie obmedzenie s rozdelením na 3 celky je len z dôvodu jednoduchšieho načítania zdrojového súboru.

1. časť - Na začiatku sú spísané všetky vstupy, výstupy a registre, ktoré daný modul obsahuje.

2. časť - Tu sa zadefinujú všetky operácie, ktoré sa v module použijú.

3. časť - Na konci sa nachádzajú informácie o prepojení operácií.

2.2.3 Špeciálne príkazy

Nejde o samostatné operácie ale len o definície pre vstupno-výstupné operácie a registre. Tieto nie sú nijako zobrazené na grafickej ploche vo vývojovom prostredí, pretože sú to len informácie, na ktoré sa potom iné operácie odvolávajú. Preto ani nie je nutné, aby obsahovali ID operácií a súradnice umiestnenia v grafickej ploche. Operácia je zložená z typu špeciálnej operácie input, output alebo register, mena konkrétneho vstupu, výstupu alebo registra a na konci sa ešte nachádza informácia o bitovej šírke a dĺžke.

Vstupno-výstupné operácie

Všetky programátorom vytvorené moduly majú možnosť vytvoriť si vlastný počet vstupov a výstupov. Na to slúžia samostatné okná vo vývojovom prostredí v menu „Modul". V tomto menu je možné si jednotlivé vstupy a výstupy pomenovať, zmazať alebo vytvoriť nové. Tiež je možné si upraviť ich bitovú šírku a dĺžku.

Špeciálne postavenie má hlavný modul *main*. Ten má preddefinované vstupy a výstupy, ktoré sa už viac nedajú meniť, pretože sa jedná o vonkajšiu komunikáciu programu. Ide o vstupy *read_new*, *read_data* a *write_busy*. Prvý obsahuje boolovskú hodnotu, ktorá je pravdivá práve vtedy, keď sa dajú zo vstupu čítať nejaké dáta. Tie sú uložené v 8-bitovom vstupe read_data. V prípade ak nie je možné zapisovať na výstup, tak je tu ešte jeden boolovský vstup *write_busy*. Výstupy modulu *main* sú *write_new* a *write_data*. Analogicky ako pri vstupoch, do prvého sa zapisuje pravdivá boolovská hodnota práve vtedy, keď chceme v ďalšom cykle procesora zapísať na výstup dáta, ktoré budú vo výstupe *write_data*.

Registre

Zmysel registrov je uchovať si dáta z jedného výpočtového cyklu procesora do druhého. Na prácu s registrami slúžia tri operácie. Čítaj z registra, Zapíš do registra a Zapíš do registra na danom indexe. Keďže je predpoklad, že s jedným registrom sa môže intenzívne pracovať, tak sme kvôli prehľadnejšiemu zobrazeniu na grafickej ploche, pridali možnosť použitia týchto operácií ľubovoľný početkrát. To znamená, že napríklad viaceré operácie čítania z registra A vytvoria síce na grafickej ploche viac objektov, ale logika programu bude rovnaká, ako keby tam táto operácia bola iba jedna.

2.2.4 Operácie

Každá operácia začína zloženými zátvorkami { a }, v ktorých je uvedené unikátne ID operácie. Na to sa potom budeme odvolávať, keď budeme chcieť ďalej pracovať s touto operáciou. Za ID nasleduje kľúčové slovo operácie. Napríklad „add“ pre operáciu sčítanie. Potom nasleduje oddeľovač #, za ktorým sa u niektorých operácií môžu vyskytnúť dodatočné parametre. Za ďalším oddeľovačom # sú uvedené súradnice v celých číslach, ktoré značia, kde sa daná operácia zobrazí na grafickej ploche vývojového prostredia. Popíšeme si ich postupne v skupinách v akých budú zatriedené v našom vývojovom prostredí.

Hodnoty

- Konštanta - Ide o obyčajné pole bitov alebo je možné zostrojiť aj pole polí bitov. Kľúčové slovo je *const*. Za ním nasledujú v obyčajných zátvorkách hodnoty. Tie sú oddelené medzerou alebo aj novým riadkom. Hodnoty môžeme uvádzať klasicky v desiatkovej sústave, no keďže v prostredí nie je nutné tieto hodnoty vyhodnocovať, tak sme dovolili zadávať hodnoty aj v tvare, ktorý akceptuje jazyk Lucid. Napríklad je môžné zadať hodnotu v šesťnástkovej alebo aj dvojkovej sústave. Avšak pokiaľ sa nejedná o sústavu desiatkovú, tak je nutné uviesť pred samotnou hodnotou znak, ktorý špecifikuje danú číselnú sústavu. Sú to znaky b pre dvojkovú a h pre šesťnástkovú sústavu.

- Prvok z poľa - Pomocou tejto operácie môžeme pristupovať k jednotlivým bitom hodnoty. Na prvom vstupe má nejaké pole bitov, na druhom je index prvku, ktorý chceme dostať a výstup. V prípade, že na vstup pošleme pole polí, tak na výstupe dostaneme pole. Kľúčové slovo tejto operácie je *index*.

Vstupno-výstupné operácie

V časti *Špeciálne operácie* sme si uviedli operácie *input* a *output*. Tie len zadefinovali daný vstup alebo výstup. K manipulácii s týmito dátami nám ale slúžia iné operácie. Majú kľúčové slová *in* pre vstupy a *out* pre výstupy. Za nimi nasleduje meno vstupu alebo výstupu, na ktorý sa operácia odvoláva. V module sme umožnili použitie viacerých týchto operácií, aby sme tak docielili sprehľadnenie programu v jeho grafickej podobe. V prípade, že chceme na viacerých miestach použiť operáciu pre výstup, tak je nutné ich dať do podmienky, inak sa na výstup pošlú dáta len z jednej operácie.

Aritmetické operácie

Tieto operácie vypočítajú výsledok danej aritmetickej operácie s dvoma vstupmi (respektíve iba s jedným vstupom pri inkrementácii a dekrementácii). Nepočítajú s poliami, takže ich bitová šírka musí byť jedna. Avšak výpočet vykonajú aj s hodnotami s rozdielnou bitovou dĺžkou. Bitová dĺžka výstupu je maximum bitových dĺžok vstupov.

- Sčítanie - add
- Odčítanie - sub
- Násobenie - mul
- Celočíselné delenie - div
- Zvyšok po delení - mod
- Inkrementácia - inc
- Dekrementácia - dec

Bitové operácie

Ide o operácie, ktoré vykonajú danú logickú operáciu so všetkými jednotlivými bitmi vstupných hodnôt.

- Konjunkcia - bit_and
- Disjunkcia - bit_or
- Exkluzívna disjunkcia - bit_xor
- Negácia - bit_not

- Bitový posun doľava - shift_left
- Bitový posun doprava - shift_right

Logické operácie

Vstupy týchto operácií sú len boolovské hodnoty. Výstup je opäť len boolovská hodnota.

- Konjunkcia - log_and
- Disjunkcia - log_or
- Negácia - log_not

Porovnávacie operácie

Tieto operácie porovnávajú hodnoty z dvoch vstupov. Neporovnávajú polia, takže ich bitová šírka musí byť jedna. Avšak porovnanie vykonajú aj s hodnotami s rozdielnou bitovou dĺžkou. Na výstupe je boolovská hodnota.

- Väčší - gt
- Väčší alebo rovný - gteq
- Menší - lt
- Menší alebo rovný - lteq
- Rovná sa - eq
- Nerovná sa - noteq

Ďalšie operácie

Do tejto sekcie sme zaradili rôzne ďalšie operácie, ktoré sa nedali umiestniť do nejakej skupiny.

- Zreťazenie - Táto operácia má dva vstupy a spojí ich do jednej hodnoty. Je však nutné, aby mali obidva rovnakú bitovú dĺžku. Kľúčové slovo je *concat*.
- Otočenie bitov - Kľúčové slovo *reverse*. Operácia otočí bity hodnoty zo vstupu. V prípade, že dostane na vstup pole, tak otočí jeho prvky.

- Zmena bitovej šírky - Pri písaní programov v našom jazyku sme zistili, že je veľmi užitočné mať operáciu, ktorá napríklad jednu 256-bitovú hodnotu rozdelí na pole 8-bitových hodnôt. Operácia funguje aj druhým smerom, takže pokiaľ máme dvojprvkové pole 16-bitových hodnôt, môže ho zmeniť na jednu 32-bitovú hodnotu. Kľúčové slovo pre túto operáciu je *change*.

- Podmienkový blok - Táto operácia je špecifická tým, že je mimo modelu dátových tokov. Na riadenie progamu v tomto modeli sa v teórii používajú mechanizmy popísané v úvodnej kapitole (brány switch a merge). Avšak ich používanie je veľmi nepraktické a program by s nimi vyzeral neprehľadne (kto by ich aj napriek tomu chcel využívať, tak je možné si vytvoriť vlastné moduly práve pomocou operácie *Podmienkový blok*). Podobne ako aj iné operácie má *Podmienkový blok* klasický vstup. Konkrétne ide o jednu boolovskú hodnotu. Ak je táto hodnota pravdivá, tak sa vykonajú operácie, ktoré sú ohraničéné hraničnou čiarou tejto operácie. Praktický zmysel umiestnenia nejakej operácie do tohto podmienkového bloku je len pri výstupoch modulu a zapisovaním do registra. Iné operácie aj tak nebudú vykonané v rámci podmienky, pretože by to nemalo žiadny efekt. Napriek tomu je možné tieto operácie do podmienkového bloku umiestniť, aby sa tak mohli opticky odlíšiť skupiny operácií. Kľúčové slovo je *if* a v časti vyhradenej pre súradnice na grafickej ploche má oproti iným operáciám navyše dvojicu čísel, ktoré uvádzajú pravý dolný roh hranice podmienkového bloku.

Operácie s registrami

Podobne ako pri špeciálnych operáciách *input* a *output* slúži operácia *register* len na zadefinovanie registra. Na prácu s ním využívame až tri rôzne operácie. Tiež ich môžeme pre jeden register použiť viac. Netreba však zabúdať na ich umiestnenie v podmienkách (v prípade, že ide o zapisovanie).

- Čítaj z registra - Použitie tejto operácie je podobné ako pri operácií *Konštanta*. Kľúčové slovo je *regC*.

- Zapíš do registra - Operácia slúžiaca na zmenu hodnoty registra. Kľučové slovo je *regZ*.

- Zapíš do registra na indexe - Táto operácia sa trocha podobá na operáciu *Prvok z poľa*. Prvý vstup však nie je pole (to predstavuje daný register) ale dáta, ktoré uložíme v registri na danom indexe. Čiže operácia nám umožní zmeniť iba

časť registra (narozdiel od predchádzajúcej operácie, ktorý zmení celý register). Kľúčové slovo je *regZIndex*.

Vlastné moduly

Všetky moduly, ktoré v projekte vytvoríme (okrem modulu *main*) môžeme kdekoľvek v programe ľubovoľný početkrát použiť. Avšak nie je možná rekurzia - nemôžeme použiť modul A v programe modulu A. Operácia slúžiaca na použitie modulu má kľúčové slovo *use* (v staršej verzií jazyka to bolo *call*) a za ním nasleduje meno použitého modulu.

2.2.5 Informácie o prepojení operácií

Ako sme spomenuli v úvode kapitoly, tak prepojenia operácií zapisujeme osobitne. Deje sa to v tretej poslednej časti tela modulu. Prvý riadok prepojení začína znakom # a je vyhradený pre zoznam súradníc bodov, cez ktoré prechádzajú prepojenia. Jediná funkcionalita týchto bodov je sprehľadnenie grafického znázornenia programu, takže pre správny chod programu ich použitie nie je nutné.

Za zoznamom bodov nasledujú už prepojenia. Každé prepojenie začína znakom #. Za ním nasleduje informácia o vstupe nejakej operácie. Tá pozostáva z indexu vstupu a identifikátora operácie, do ktorého dáta vchádzajú. Ďalej voliteľne nasleduje zoznam indexov bodov, cez ktoré prechádza toto prepojenie. Na záver je index výstupu a identifikátor operácie, z ktorého dáta vychádzajú.

2.2.6 Príklady použitia

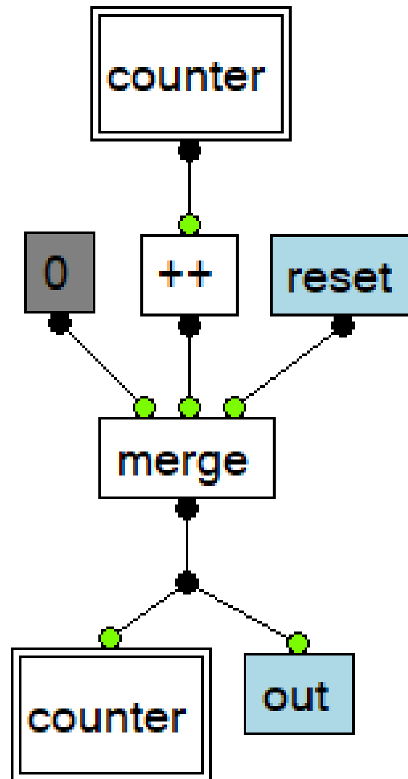
Na záver kapitoly si uvedieme niekoľko jednoduchých príkladov napísaných v našom jazyku. Každý príklad bude obsahovať zdrojový kód a aj jeho grafické znázornenie.

Počítadlo cyklov

Pokiaľ chceme zistiť ako dlho sa bude vykonávať náš program a nedokážeme to ručne spočítať, tak potrebujeme nejaké počítadlo cyklov procesora. Na obrázku 2.1 môžeme vidieť grafický zápis takéhoto programu. Modul má jeden vstup a jeden výstup. Pokiaľ na boolovský vstup s názvom *reset* príde hodnota 1, tak to znamená, že chceme počítadlo resetovať a nastaviť jeho hodnotu na 0. Na zapamätanie si hodnoty počtu cyklov nám slúži register *counter*. V prípade ak nechceme počítadlo resetovať, tak sa hodnota, ktorú tento register obsahuje v aktuálnom cykle procesora zväčší o jedna.

Takto upravená hodnota sa pošle na výstup s názvom *out* a opäť sa zapíše do registra *counter*.

Na rozlíšenie toho aké hodnoty sa majú poslať na výstup a zapísať sa do registra sme použili vlastný modul *merge*, ktorý má rovnakú funkcionalitu ako zlučovacia brána z teórie modelu dátových tokov, ktorú sme si rozobrali v kapitole 1.



Obr. 2.1: Počítadlo cyklov procesora

Keď sa pozrieme na zdrojový kód tohto modulu, tak môžeme vidieť podrobnosti implementácie. Na riadkoch 3-5 sú zadefinovaný vstup a výstup modulu a tiež aj jediný register, ktorý modul používa. Za kľúčovými slovami a názvami týchto objektov sa nachádza oddeľovač. Za ním sú uvedené bitové rozmery. Na vstup *reset* nám stačí iba jeden bit, lebo reprezentuje len boolovskú hodnotu. Výstup *out* a register *counter* obsahujú informáciu o počte cyklov, táto informácia je 32-bitové číslo.

Na riadkoch 6-12 sú zaznačené jednotlivé operácie, ktoré sme mohli vidieť aj na obrázku 2.1.

Riadok 13 obsahuje súradnice jediného bodu, ktorý nie je súčasťou nejakého objektu. Jeho funkcionalita je len pre prehľadnejšie grafické znázornenie.

Riadky 14-19 nesú informácie o prepojeniach medzi operáciami. Každé prepojenie sa skladá z troch častí.

1. Začiatok - značí vstup nejakej operácie. Ten je zložený z indexu vstupu a z identifikátora operácie.

2. Cesta - indexy bodov z riadka 13, cez ktoré prechádza dané spojenie.

3. Koniec - označuje výstup nejakej operácie, z ktorej čerpá dáta operácia z prvej časti. Tiež obsahuje index výstupu a identifikátor operácie.

Index vstupu a výstupu má spoločné číslovanie. Teda pokiaľ má operácia jeden vstup a jeden výstup. Tak tento vstup sa označuje indexom 0 a výstup indexom 1.

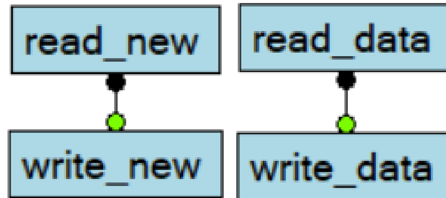
```
1  def pocitadlo
2  [
3    input reset # 1 1
4    output out # 1 32
5    register counter # 1 32
6    {regC0} regC counter # 212 74
7    {inc0} inc # 212 150
8    {regZ0} regZ counter # 182 316
9    {out0} out out # 253 308
10   {const0} const (0) # 1 32 # 163 149
11   {merge0} use merge # 211 219
12   {in0} in reset # 270 150
13   # 211 266
14   # 0 inc0 0 regC0
15   # 0 regZ0 0 3 merge0
16   # 0 out0 0 3 merge0
17   # 0 merge0 0 const0
18   # 1 merge0 1 inc0
19   # 2 merge0 0 in0
20 ]
```

Listing 2.1: Zdrojový kód počítadla cyklov.

Echo server

V druhom príklade si ukážeme ako funguje vonkajšia komunikácia programu. Jediné čo robí program znázornený na obrázku 2.2 je, že hodnoty zo vstupov pošle na výstup.

Teda pokiaľ je na vstupe *read_new* pravdivá hodnota, tak táto bude aj na výstupe *write_new*. V tom prípade sa dáta zo vstupu *read_data* zapíšu na výstup *write_data* a my ich môžeme vidieť na monitore sériového portu, ktorý má prostredie Mojo IDE. Tento program nerieši vstup *write_busy*, ktorý hovorí, či je možné zapisovať. Takže v prípade veľkej frekvencie vstupov môže dôjsť k tomu, že o niektoré dáta prídeme.



Obr. 2.2: Program posiela na výstup všetky dáta, zo vstupu.

Tentokrát nám ukážka zdrojového kódu ukazuje modul main. Ten má vždy zadefinované vstupy a výstupy z riadkov 3-7. Ich bitové rozmery sme si popísali na začiatku tejto kapitoly.

Na riadkoch 8-11 sú operácie, ktoré môžeme vidieť aj na obrázku 2.2. Riadok 12 tentokrát neobsahuje súradnice žiadneho bodu, pretože ani v grafickom znázornení sme žiadny nepoužili.

Na koniec sú v riadkoch 13-14 uvedené spojenia medzi operáciami. Ich význam sme si rozobrali pri prvom príklade.

```
1  def main [
2    input read_new # 1 1
3    input read_data # 1 8
4    input write_busy # 1 1
5    output write_new # 1 1
6    output write_data # 1 8
7    {in0} in read_new # 106 80
8    {in1} in read_data # 210 79
9    {out0} out write_new # 106 136
10   {out1} out write_data # 210 137
11   #
12   # 0 out0 0 in0 # 0 out1 0 in1
13 ]
```

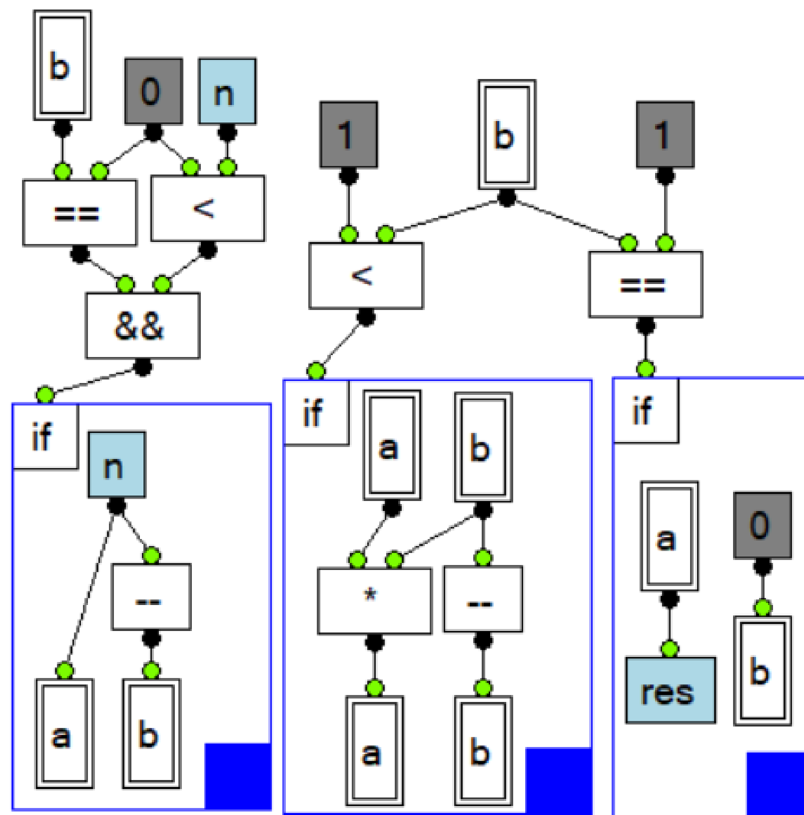
Listing 2.2: Zdrojový kód echo servera.

Faktoriál

V tomto poslednom príklade si ukážeme ako môžeme v našom jazyku zostrojiť program pre výpočet faktoriálu. Ten môžeme vidieť na obrázku 2.3. Program má tri stavy.

Do prvého stavu sa dostaneme, keď register b má hodnotu nula a zároveň je na vstupe n nastavená hodnota väčšia ako nula, čo znamená, že existujú nejaké dáta, ktoré môžeme spracovať. Tieto vstupné dáta uložíme do registra a a ich dekrementovanú hodnotu uložíme do registra b. V druhom stave prebieha samotný výpočet. Vynásobia sa hodnoty registrov a a b a výsledok sa uloží opäť do registra a. Hodnota v registri b sa dekrementuje. Do tretieho konečného stavu sa dostaneme ak už register a obsahuje výsledok výpočtu. To zistíme tak, že v registri b bude hodnota jedna. V tomto stave sa na výstup *res* uloží výsledok výpočtu.

Takto zostavený program môže vypočítať faktoriál pre hodnoty väčšie alebo rovné ako 2.



Obr. 2.3: Program pre výpočet faktoriálu.

Keď sa pozrieme na zdrojový kód, tak vidíme, že ako registre, tak aj vstupy a výstupy sú 64-bitové. Takže toto číslo nám ohraničuje zhora možnosti výpočtu. Pokiaľ by sme chceli vyrátať faktoriál pre väčšie čísla, tak musíme zväčšiť bitovú dĺžku týchto objektov. Z dôvodu kompaktnosti sme niektoré riadky zlúčili. Napriek tomu je aj takto

upravený kód korektný.

```
def faktorial [
  input n # 1 64 output res # 1 64
  register a # 1 64 register b # 1 64
  {in0} in n # 224 328 {dec1} dec # 240 389
  {in1} in n # 275 156 {if0} if # 191 315 280 472
  {const0} const (1) # 1 1 # 477 176
  {eq0} eq # 207 212 {log_and0} log_and # 236 264
  {regZ0} regZ a # 200 452 {regC0} regC b # 199 144
  {regZ1} regZ b # 240 452 {lt1} lt # 339 241
  {const1} const (1) # 1 1 # 331 176
  {regC2} regC b # 404 176 {if1} if # 316 304 428 474
  {regC3} regC a # 351 319 {regC4} regC b # 393 320
  {regZ2} regZ a # 343 460 {regZ3} regZ b # 393 460
  {mul0} mul # 343 391 {dec0} dec # 393 390
  {eq1} eq # 468 245 {if2} if # 468 303 530 476
  {out0} out res # 479 432 {regC5} regC a # 479 361
  {const2} const (0) # 1 1 # 241 156
  {lt0} lt # 266 210 {regZ4} regZ b # 522 423
  {const4} const (0) # 1 1 # 522 356
  #
  # 0 dec1 0 in0 # 0 if0 2 log_and0
  # 0 eq0 0 regC0 # 1 eq0 0 const2
  # 0 log_and0 2 eq0 # 1 log_and0 2 lt0
  # 0 regZ0 0 in0 # 0 regZ1 1 dec1
  # 0 lt1 0 const1 # 1 lt1 0 regC2
  # 0 if1 2 lt1 # 0 regZ2 2 mul0
  # 0 regZ3 1 dec0 # 0 mul0 0 regC3
  # 1 mul0 0 regC4 # 0 dec0 0 regC4
  # 0 eq1 0 regC2 # 1 eq1 0 const0
  # 0 if2 2 eq1 # 0 out0 0 regC5
  # 0 lt0 0 const2 # 1 lt0 0 in1 # 0 regZ4 0 const4 ]
```

Listing 2.3: Zdrojový kód výpočtu faktoriálu.

2.3 Univerzálnosť jazyka

Naším cieľom bolo navrhnúť jazyk, ktorý by bol univerzálny. Teda taký, ktorý by bol rovnako silný ako Turingov stroj. Na dokázanie tejto vlastnosti musíme v našom jazyku zostrojiť program, ktorý by umožnil tento Turingov stroj simulovať. Mohol by vyzerať napríklad nasledovne.

Na vstupe by sme mali 6 polí, pričom 5 z nich by predstavovalo prechodovú funkciu a posledné šieste pole by bola vstupná páska. Okrem toho by sme ešte na vstupe potrebovali jednu hodnotu, ktorá by označovala akceptačný stav programu. Výstup programu by bol iba jeden a ten by reprezentoval pásku po vykonaní programu Turingovho stroja.

Prechodová funkcia na základe dvoch vstupov:

1. stav Turingovho stroja
2. symbol na páske, nad ktorou je hlava Turingovho stroja

vydá tri výstupy:

1. stav, do ktorého sa Turingov stroj dostane
2. hodnota, ktorá sa zapíše na pásku, nad ktorou je hlava
3. smer, ktorým sa má hlava posunúť

Z tohto dôvodu je nutné mať v našom programe spomenutých päť polí. Pričom všetky ich prvky s rovnakým indexom reprezentujú jeden prechod.

Na uchovanie informácii o aktuálnom stave alebo prípadne ďalších údajov, ktoré by sme v praktickej implementácií potrebovali, by nám poslúžili registre.

Z vyššie uvedeného predpokladáme, že jazyk DataWolf je univerzálny.

Kapitola 3

Implementácia

V tejto kapitole si popíšeme ako sme postupovali pri implementácii grafického vývojového prostredia ako aj kompilátora do jazyka Lucid. Na začiatku sme premýšľali nad oddelením vývojového prostredia a kompilátora. Pričom kompilátor by mohol byť napísaný aj v inom jazyku. Tento postup by mal isté výhody ako napríklad to, že by v jednotlivých častiach bol menší počet tried a ich atribútov a metód. Zdrojový kód by tak bol o niečo prehľadnejší, avšak za cenu zbytočného duplikovania niektorých úkonov a zhoršenia výkonu kompilácie (aj keď tento faktor je zanedbateľný kvôli vysokej časovej náročnosti kompilácie z jazyka Lucid ďalej). Nakoniec sme sa však rozhodli spojiť obe funkcionality (vývojové prostredie aj kompilátor) do jediného programu. Ten sme napísali v jazyku C# vo vývojovom prostredí SharpDevelop. Program sme pomenovali rovnako ako náš jazyk, teda DataWolf.

3.1 Triedy reprezentujúce operácie a ich prepojenia

Program využíva veľké množstvo tried, pričom vo veľkej miere sme využívali dedičnosť. Na nasledujúcich stranách sa pokúsime uviesť tie najdôležitejšie z nich a niektoré ich významené atribúty a metódy. Menné konvencie sú nasledovné:

- Čiara - prepojenie medzi operáciami. Môže ísť o jednoduchú úsečku alebo o lomenú čiaru.
- Bodka - všetky čiary prechádzajú cez nejakú bodku. Tá symbolizuje nejaký vstup, výstup alebo nejaký bod medzi nimi, cez ktorý prechádza čiara.
- Objekt - spoločné označenie všetkých možných operácií jazyka, ktoré majú nejakú vizuálnu reprezentáciu.

- Info - oznaženie pre rôzne štruktúry, v ktorých sú zoskupené informácie o rôznych typoch Objektov.

3.1.1 Bodka

Abstraktná trieda, ktorá má dve podtriedy. Dôležité atribúty sú *stav*, ktorý je typu *Brush* a reprezentuje farbu bodky. Celočíselné atribúty x a y označujú umiestnenie bodky v grafickej ploche. Avšak toto umiestnenie môže pri odvodených triedach znamenať niečo iné. Napríklad v našom prípade relatívnu alebo absolútnu pozíciu. Z tohto dôvodu nie sú tieto atribúty verejné a na ich získanie alebo úpravu používame gettery a settery.

- ObjektBodka - Táto trieda slúži na reprezentovanie vstupov a výstupov objektov. Atribúty x a y nereprezentujú absolútne súradnice bodky ale len relatívne vhľadom k objektu ku ktorému daná bodka patrí. Referencia na ňu je v atribúte *obj*. Atribút *stav* môže nadobúdať hodnoty čierna v prípade výstupov objektov, červená v prípade vstupov, ku ktorým nie sú priradené žiadne dáta alebo zelená znázorňujúca, že k danej bodke sú už dáta pripojené. Atribút *id* hovorí o pozícii objektovej bodky v poli *bodky* daného objektu. Posledný atribút *info* je typu *BodkaInfo* a sú v ňom uložené informácie o vstupe alebo výstupe ku ktorému táto bodka patrí.

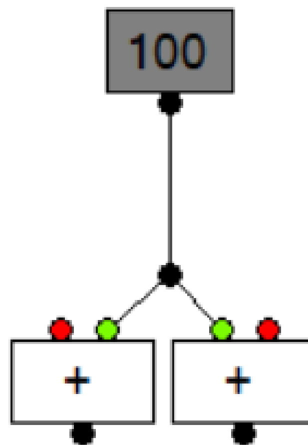
- CiaraBodka - Keďže čiary medzi objektami môžu byť aj lomené čiary, tak na reprezentáciu ich vrcholov slúži táto trieda. Atribúty x a y tentokrát slúžia na vyjadrenie absolútnej pozície bodky na grafickej ploche, keďže čiarová bodka nemusí prislúchať len jednému objektu. Z toho dôvodu si táto bodka musí pamätať aj referencie na všetky čiary, ktorých je súčasťou. Tie sú uložené v zozname *ciary* a v prípade, že je tento zoznam prázdny, tak sa bodka zmaže. V atribúte *ifBlok* je ešte referencia na najbližší podmienkový blok ku ktorému bodka patrí. Štandardne má hodnotu *null*.

3.1.2 Čiara

Na reprezentáciu spojenia dvoch objektov používame túto triedu. V atribútoch *bodkaVstup* a *bodkaVystup* si uchováva referenciu na bodky vstupu jedného objektu a výstupu druhého objektu. Keďže operácie nemusíme spájať len obyčajnými úsečkami ale aj lomenou čiarou, tak je potrebné si tiež pamätať aj jednotlivé body, cez ktoré táto čiara prechádza. Tie sú uložené v atribúte *cesta*. Atribút *stav* je typu *Pen* a môže mať

čiernu alebo červenú farbu. Pri každej zmene v programe prebehne kontrola správnosti a v prípade, že sa snažíme dostať dáta príliš veľkých rozmerov do vstupu operácie, ktorá nemá také rozmery, tak stav nadobudne červenú farbu a čiara sa zafarbí na červeno, čím dá hneď vedieť programátorovi, že niečo nie je v poriadku.

Pri vytváraní čiar je možné postupovať vždy len zdola nahor. Čiže pre vytvorenie čiary treba kliknúť najskôr na vstup nejakého objektu. Druhý klik urobíme na výstupe iného objektu (alebo aj na bode čiary, ktorá z tohto objektu už vychádza ako môžeme vidieť na obrázku 3.1). Každý klik, ktorý urobíme medzi tým, vytvorí nový bod, z ktorého môžeme viesť čiaru ďalej novým smerom. Vytváranie čiar môžeme kedykoľvek zrušiť stlačením pravého tlačítka myši.



Obr. 3.1: Na obrázku sú dve operácie sčítania. Obidve majú priradený iba jeden vstup. Tieto bodky sú zafarbené na zeleno, zatiaľ čo tie druhé na červeno. Spojenia objektov na obrázku sú vytvorené pomocou jedného pomocného bodu, cez ktorý prechádzajú obidve čiary, vďaka čomu nemáme v programe nakreslené 2 dlhé čiary.

3.1.3 Info

Info je abstraktná trieda, od ktorej dedia triedy, slúžiace na uchovanie informácií o registroch alebo vstupoch a výstupoch. Trieda s informáciami o objektoch je samostatná entita nakoľko nemá podobné informácie, ktoré by mohla zdielať. Atribút *meno* obsahuje meno registra, vstupu alebo výstupu a *bitSirka* a *bitDlzka* hovoria o rozmeroch týchto dát. Pre generovanie zdrojového kódu nášho jazyka je užitočný ešte atribút *typ*.

- RegisterInfo - Jediná zmena oproti rodičovskej triede je v konštruktore, kde sa nastavuje atribút *typ* na hodnotu *register*.

- BodkaInfo - táto trieda má navyše dve boolovské atribúty *bitSirkaZmena* a *bitDlzkaZmena* značiace, či je možné bitové rozmery dát objektu meniť. Všetky objekty majú nejaké štandardné bitové rozmery. Napríklad operácia sčítania predpokladá, že na vstup mu prídu dve integerovské hodnoty. V prípade ak chceme zrátať 64-bitové hodnoty, tak sa kontrolná metóda pozrie na to, či je možné zmeniť bitovú dĺžku objektu a ak áno, tak ju vhodne upraví. Táto trieda sama o sebe nehovorí, či ide o bodku patriacu nejakému vstupu alebo výstupu. Preto pre generovanie korektného zdrojového kódu nášho jazyka (a aby sme sa vyhli potrebe zavedenia niekoľkých ďalších tried) je atribút *typ* nastavený až pri samotnom generovaní kódu, pretože vtedy dokážeme bez problémov určiť, či ide o vstup alebo výstup (a v inej situácii nám táto informácia nie je k úžitku).

3.1.4 ObjektInfo

Na uloženie vlastností jednotlivých objektov (ako vstavaných aj užívateľsky vytvorených) slúži trieda ObjektInfo. Stringové atribúty *znamienko* a *text* si pamätajú pod akým názvom sa zobrazuje operácia na pracovnej ploche, respektíve v menu pridania nových objektov. V aktuálnej verzii vývojového prostredia sú pre vlastné moduly tieto atribúty totožné, takže užívateľ si nemôže zadefinovať skratku pre svoj modul. Ďalšie atribúty tejto triedy sú zoznamy *vstupy* a *vystupy*. Obsahujú elementy typu *BodkaInfo*, ktoré hovoria o charaktere vstupov a výstupov operácie. Posledným atribútom je boolovská hodnota *clkRstSignal*, ktorú využívame pri kompilácii do jazyka Lucid, aby sme určili či daná operácia potrebuje hodinový a resetový vstupný signál. Tie sú potrebné, napríklad keď v danom module pracujeme s registrami.

3.1.5 Objekt

Objekt je abstraktná trieda, od ktorej sa odvíjajú všetky operácie jazyka s nejakou vizuálnou reprezentáciou. Dôležité atribúty sú polia *bodky* a *ciary*. V nich sú uložené referencie na všetky vstupné a výstupné bodky a čiary, ktoré do objektu vchádzajú. Stringový atribút *op* obsahuje názov danej operácie. Je tu aj *ifBlok*, ktorý má štandardne hodnotu *null*. V prípade, že táto operácia prislúcha nejakému podmienkovému bloku, tak je tu na neho uložená referencia.

Atribút *id* je unikátny identifikátor operácie. Generuje sa automaticky pomocou metódy *getOpId*, ktorá je súčasťou triedy *Modul*. Toto ID je zložené z kľúčového slova operácie v našom jazyku a poradového čísla. Keďže v priebehu vytvárania projektu môže dôjsť k rôznym zmenám (opakované mazanie a pridávanie operácií), tak nie je

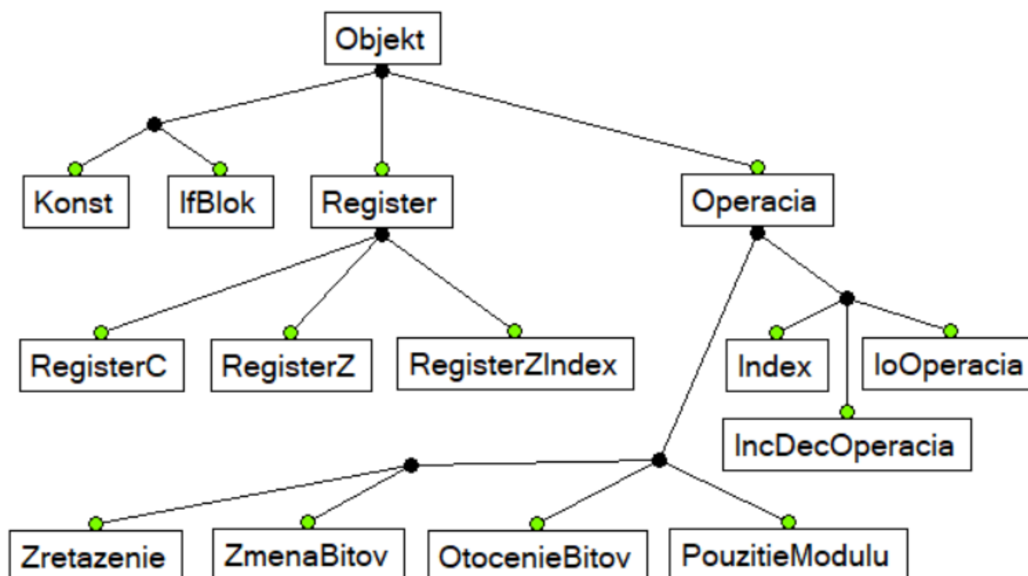
jednoduché hneď na prvýkrát vygenerovať správne ID. Preto sa postupne skúšajú ID s poradovými číslami od 0 po 999. V prípade, že sa nepodarí vygenerovať nové ID, tak sa vráti hodnota hovoriaca o prekročení limitu počtu konkrétnej operácie v module. Keď sa stane toto, je nutné veľký modul rozdeliť na menšie, prípadne pomenovať si operácie vlastnými identifikátormi.

Súradnice objektu, ktoré sme si načítali zo zdrojového súboru hovoria o strede objektu na grafickej ploche. Sú uložené v atribútoch x a y. Okrem nich si v rx a ry pamätáme aj polomery v smere osi x a osi y. Pre grafickú časť programu je ešte dôležitý atribút bg označujúci farbu pozadia objektu. Väčšina štandardných operácií má obyčajné biele pozadie. Operácia *Konštanta* má sivú farbu a vstupno-výstupné operácie zase svetlo modrú. Tieto sme sa totiž rozhodli odlíšiť od ostatných, pretože sa pre ne počas kompilácie negenerujú vlastné moduly a ich použitie môže teda byť opakované bez zvýšenia náročnosti programu. Dôvod na toto bol ten, aby bolo zakreslenie programu prehľadnejšie. Podobne sa správajú aj operácie pracujúce s registrami, tam sme ale na odlíšenie použili miesto farby dvojité orámovanie.

Keďže ide o najviac používanú triedu v programe, tak má aj mnoho metód. Najdôležitejšie sú *kresli* a *isClick*. Prvá vykreslí na grafickú plochu objekt spolu s jeho bodkami. Druhá vracia boolovskú hodnotu informujúcu o tom, či sme klikli na objekt myšou. Všetky objekty majú obdĺžnikový tvar, takže výpočet je celkom jednoduchý a nie je nutné núto metódu v podtriedach preťažovať.

Za zmienku stojí ešte skupina metód, ktoré sa starajú o zobrazovanie vlastností v pravom bočnom paneli vývojového prostredia. To môžeme vidieť na obrázku 3.9. Ide o metódy *nastavEditovaciePrvky*, *zrusEditovaciePrvky* a *ulozEditovaneVlastnosti*. Keď klikneme na niektorý z objektov, tak sa nám vďaka ním zobrazia v paneli informácie o objekte. Niektoré z nich tu môžeme priamo editovať. Ide napríklad o presné umiestnenie v grafickej ploche, zmenu identifikátora alebo bitových rozmerov pri niektorých operáciách. V prípade vstupno-výstupných operácií alebo operácií pracujúcich s registrami tu môžeme zmeniť aj daný vstup, výstup alebo register za iný.

Podtried tejto triedy Objekt je veľmi veľa a nemá veľký zmysel si ich podrobne rozoberať. Na obrázku 3.2 môžeme vidieť aspoň ich prehľad.



Obr. 3.2: Na obrázku môžeme vidieť štruktúru tried odvodených od triedy Objekt. Graf bol vytvorený v našom vývojovom prostredí za použitia vlastných modulov, čo znamená, že samotný program (alebo aspoň jeho časť) má aj ďalšie praktické využitie.

3.2 Vývojové prostredie

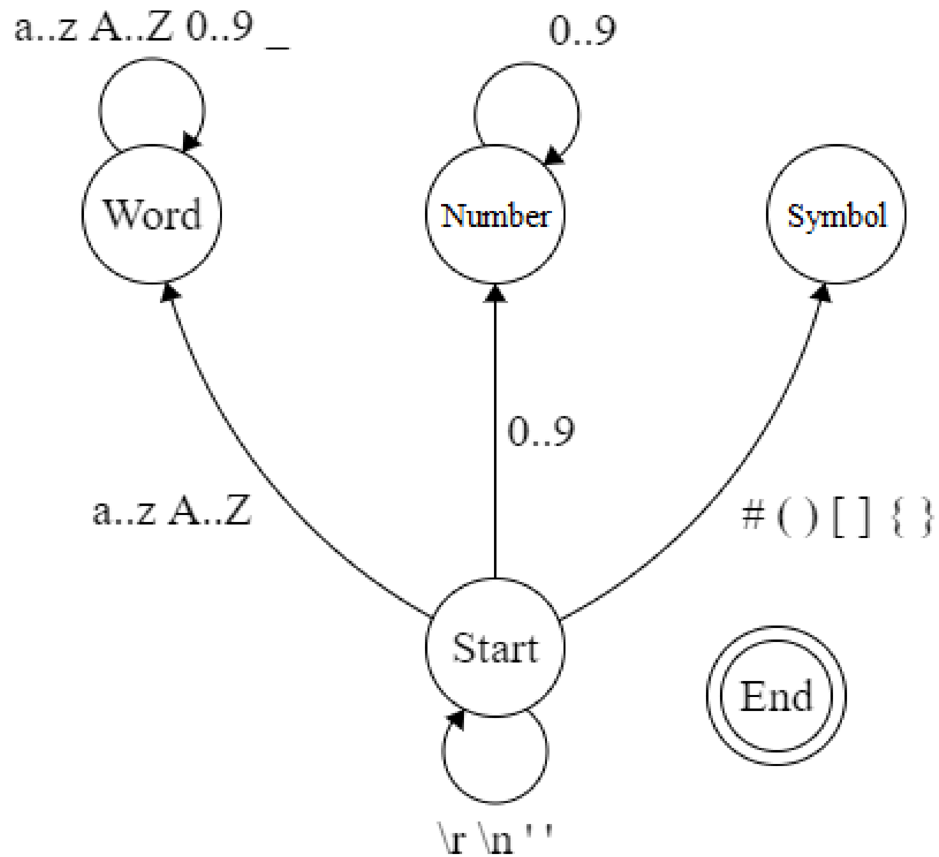
3.2.1 Vstupy

Prvou činnosťou, ktorú musí naše prostredie zvládať je načítanie zdrojových súborov. Ide o obyčajné textové súbory, ktorým sme priradili príponu *dw* podľa skratky mena jazyka DataWolf. Tento proces sa zabezpečuje v troch triedach:

1. Vstup - číta vstup a pomocou metódy *next* generuje postupnosť znakov, ktoré sa zapisuje do atribútu *look*.

2. Analyzator - táto trieda slúži ako lexikálny analyzátor. Testuje obsah atribútu *look* inštancie predchádzajúcej triedy alebo si pýta ďalší znak volaním metódy *next*. Pomocou metódy *scan* generuje postupnosť jazykových prvkov, ktoré zapisuje do atribútu *token*. Táto trieda má formu konečného automatu, ktorý vyzerá takto 3.3.

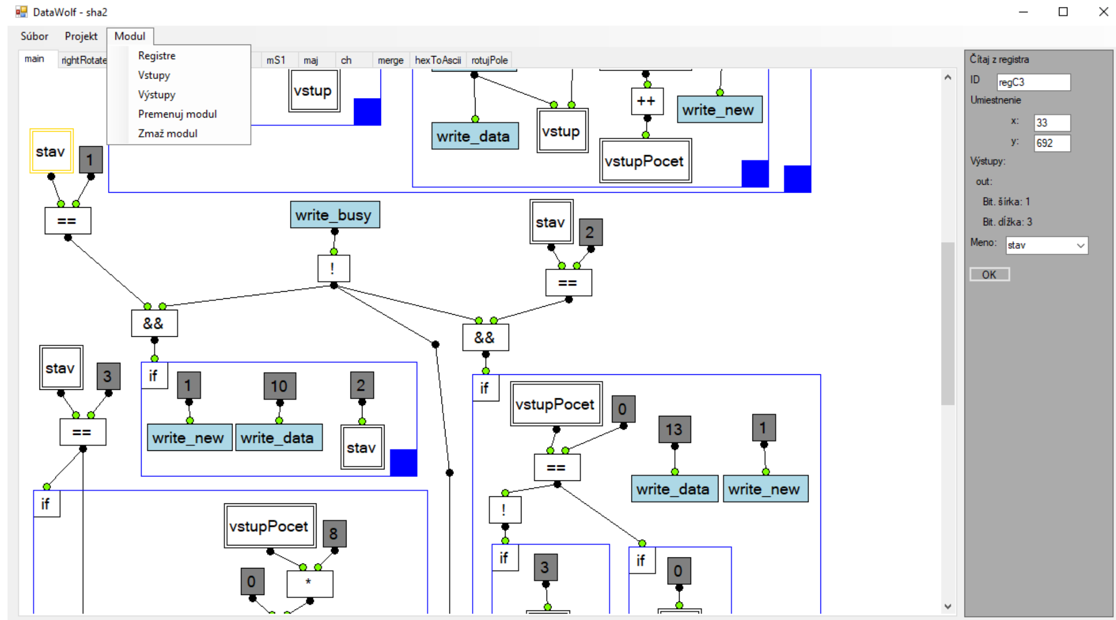
3. Projekt - ide o rozsiahlu triedu, ktorá okrem iných dôležitých modulov plní aj rolu syntaktického analyzátora. Testuje atribút *token* inštancie triedy Analyzator alebo si pýta ďalšiu lexému volaním metódy *scan*.



Obr. 3.3: Konečný automat ukazujúci ako funguje lexikálny analyzátor. Všetky zmeny stavov nezaznačené na obrázku vedú do stavu *End*. V pôvodnej verzii jazyka bol tento automat oveľa rozsiahlejší, lebo umožňoval písať klasické operátory pre základné operácie ako sčítanie a odčítanie. No dospeli sme k záveru, že to jazyk už nepotrebuje nakoľko písanie programu v textovej podobe sa dosť skomplikovalo.

3.2.2 Užívateľské rozhranie

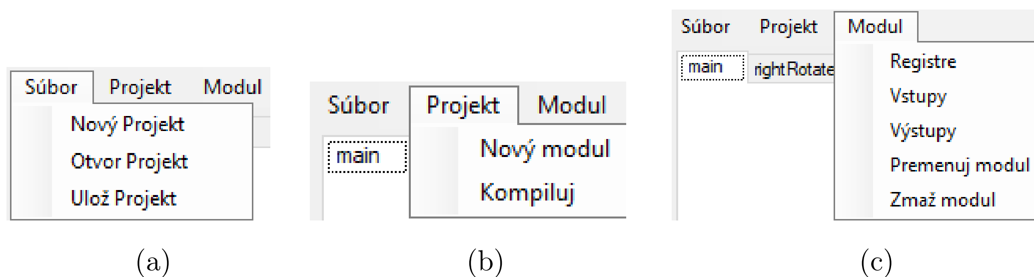
Na obrázku 3.4 môžeme vidieť kompletné užívateľské rozhranie nášho vývojového prostredia. V ďalších častiach tejto sekcie si podrobne popíšeme jednotlivé elementy.



Obr. 3.4: Užívateľské rozhranie.

Menu

Na obrázkoch 3.5a až 3.5c sú rozbalené položky hlavného menu. Nachádzajú sa tu všetky operácie potrebné na vývoj programu. Menu *Projekt* a *Modul* sú dostupné iba v prípade, že máme otvorený nejaký projekt. Pokiaľ máme ako aktívny nastavený modul *main*, tak nie je možné pristupovať k položkám *Premenuj Modul* a *Zmaž modul*. Tiež u tohto modulu sú okná pre *Vstupy* a *Výstupy* iba informatívne, nakoľko v tomto module nie je možné meniť vstupy a výstupy.

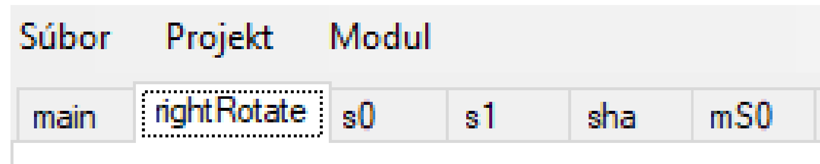


(a) (b) (c)

Obr. 3.5: Menu programu DataWolf.

Panel modulov

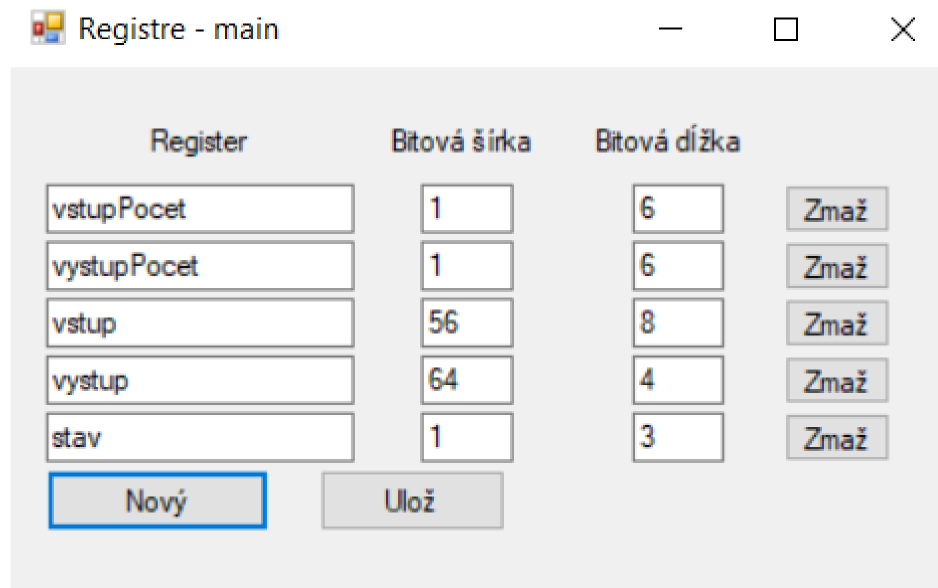
Pokiaľ má náš projekt viac modulov, tak prepínať medzi nimi môžeme v lište pod hlavným menu 3.6.



Obr. 3.6: Lišta modulov.

Úprava parametrov modulov

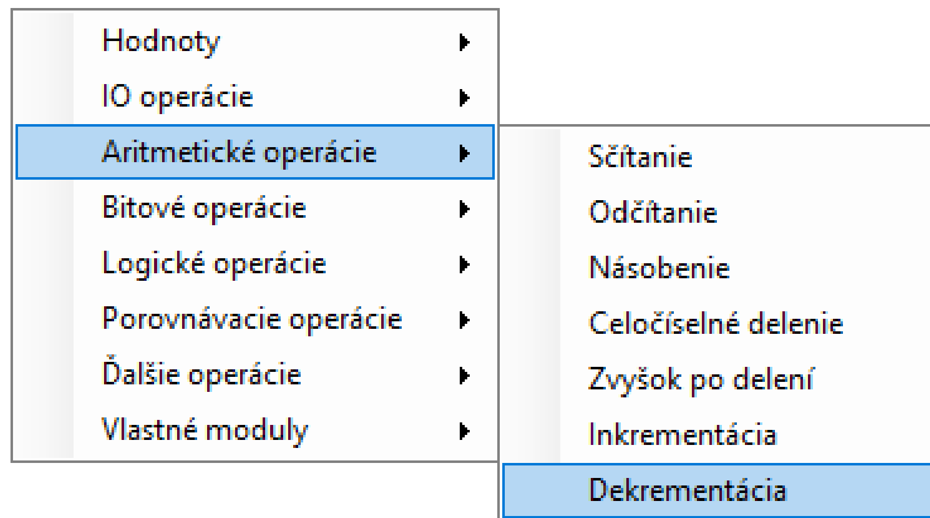
Každý novovytvorený modul začína bez zadefinovaných vstupov, výstupov alebo registrov. Nastaviť si ich môžeme v menu *Modul*, v častiach *Registre*, *Vstupy* a *Výstupy*. Jednotlivé okná sú veľmi podobné a vyzerajú nejako takto 3.7. Pokiaľ chceme, aby sa prejavili vykonané zmeny, tak je nutné stlačiť tlačidlo *Ulož* alebo okno zavrieť (uloží sa automaticky). Pokiaľ robíme výrazné úpravy pri vstupoch a výstupoch modulu, tak môže nastať na iných miestach programu situácia, kde už použitie tohto modulu má priradené vstupy. V takom prípade sa všetky vstupy (respektíve výstupy) použitia modulu zrušia, aby sa predišlo možným komplikáciám.



Obr. 3.7: Okno registrov modulu.

Pridávanie operácií

Nové operácie je možné pridať pomocou pravého tlačítka myši na grafickej ploche programu 3.8. Toto riešenie má výhodu v tom, že novovzniknutá operácia sa umiestni presne na lokáciu, ktorú si vyberie užívateľ a tak nie je nutné túto operácia dodatočne presúvať. Toto kontextové menu obsahuje jednotlivé operácie roztriedené v kategóriách, ktoré sme si popísali v kapitole 2. V prípade, že modul nemá napríklad žiadne registre, tak možnosť pridania operácií, ktoré pracujú s registrami sa ani neobjaví.



Obr. 3.8: Menu pridávania operácií.

Vlastnosti operácií

Niektoré operácie vyžadujú okrem priradenia vstupných dát aj dodatočné parametre. Môže ísť napríklad o operáciu na zmenu bitovej šírky dát alebo aj o obyčajnú konštantu. Tieto parametre sa dajú nastaviť v pravom bočnom paneli 3.9. Okrem rôznych parametrov sa tam dajú meniť aj základné údaje ako je identifikátor operácie alebo súradnice na grafickej ploche. Niektoré vlastnosti tu nie je možné meniť. Slúžia len na informovanie užívateľa.

3.2.3 Výstupy

Výstupom nášho programu je kód skompilovaný z nášho jazyka do jazyka Lucid. Aby mohol byť program pohodlne spustený v prostredí Mojo IDE, tak sa vytvorí adresár s názvom "DataWolfProjekt_[Názov projektu]". Ten obsahuje súbor "[Názov projektu].mojo". To je xml súbor, v ktorom sú informácie o projekte. Ďalej je tu ešte adresár "source", ktorý obsahuje sériu lucidovských súborov, ktoré reprezentujú

Obr. 3.9: Panel vlastností pre operáciu Konštanta.

jednotlivé používané moduly. Pri každej kompilácii nášho kódu sú všetky dáta zmazané a nahradené novými.

3.3 Kompilátor

Súčasťou vývjojového prostredia je aj kompilátor do jazyka Lucid. Ten je upravenou verziou známejšieho jazyka Verilog. V našej práci sme pracovali s Lucidom, pretože bol viac vhodnejší pre začiatočníkov. Rozdiely oproti Verilogu sú len veľmi malé, ale pomohli nám v úvode práce lepšie pochopiť novú programovaciu paradigmu. O preklad z Lucidu do Verilogu sa stará vývojové prostredie *Mojo IDE*, ktoré navrhli výrobcovia nami testovaného FPGA zariadenia. Rovnako v tomto prostredí je možné aj komunikovať s FPGA zariadením pomocou sériového portu.

3.3.1 Spôsob kompilácie

Na kompiláciu využívame naše triedy typu *Objekt* z predchádzajúcej sekcie. Skoro každý objekt predstavuje samostatný modul. Pri niektorých operáciách sa modul

nevytvára. Ide napríklad o operácie odvodené od triedy *Register* a *IoOperacie*. Nie vždy sa však vytvorí samostatný súbor predstavujúci modul. Každý objekt v sebe nosí informáciu o bitovej šírke a dĺžke vstupov a prípadne aj výstupov (pokiaľ je to potrebné). V prípade, že viac objektov má rovnaké parametre, tak sa vygeneruje iba jeden súbor s modulom, ktorý potom využívajú všetky objekty s rovnakými parametrami.

Toto bol aj jeden z hlavných dôvodov prečo sa v programe dbá na rozmery všetkých dát. Programátor si totiž po počiatočnom nastavení nemusí robiť starosti s tým aké sú veľké vstupy a výstupy modulov. Všetko sa to vypočíta automaticky, a tak aby sa zbytočne neplýtvalo zdrojmi. Je síce možné, že pri optimalizáciách robených pri konečnom navrhovaní dizajnu FPGA sa vykonávajú podobné procesy, no my sme toto chceli mať viac pod kontrolou a nespoliehať sa, že to niekto urobí za nás.

Pri kompilácii sa čo najviac využívajú možnosti jazyka Lucid, ktorý podporuje väčšinu základných operácií ako je napríklad sčítanie alebo bitové operácie. Pri niektorých špeciálnych operáciách sme však museli vygenerovať väčšie množstvo kódu. Ide o prevažne operácie zo skupiny *Ďalšie operácie* ako je *Zmena bitových rozmerov* alebo *Otočenie bitov*.

3.3.2 Dôležité metódy

Na vygenerovanie výsledného kódu používame sériu metód, pričom vo veľkej miere využívame vlastnosti dedičnosti tried, aby sme dosiahli výsledok s čo najmenším množstvom kódu, ktorý by sa naprieč triedami opakoval.

Hlavnou metódou je metóda *vytvorModul*. Tá má dva stringové parametre, ktoré hovoria o umiestnení výsledného súboru a jeho mene. V tejto metóde sa vygenerujú vstupy a výstupy modulu ako aj ďalšie časti, ktoré majú všetky moduly rovnaké. Výsledok sa potom zapíše do súboru.

Na vygenerovanie hlavnej časti programu sa volá metóda *programModulu*. Túto metódu veľa z operácií preťažuje vlastným kódom avšak štandardne sa správa tak, že využíva uloženého znamienka pre danú operáciu. Ten potom umiestni medzi dva vstupy. V prípade jedného vstupu znamienko umiestni dopredu pred tento vstup. Toto zabezpečuje funkcionalitu všetkých základných operácií.

Okrem metód určených na generovanie súborov s modulmi tu máme ešte trojicu metód, ktoré sa používajú na zapisovanie dát do týchto modulov a ich čítanie. Ide o metódy *generujLucidZapisovanieDat* a *generujLucidCitanieDat*. Tie dostávajú ako parameter ID bodky, z ktorej chceme dáta načítať alebo ich naopak zapísať. Štandardne

pri operáciách s vlastným modulom sa vráti stringová hodnota obsahujúca názov ID modulu a meno danej bodky. Metódu *generujLucidKod* používame na zapísanie všetkých vstupných hodnôt do modulov.

Okrem týchto hlavných a niektorých pomocných metód je významná ešte metóda *vlastnyModul*. Tá nemá žiadne parametre, ale vracia len boolovskú hodnotu informujúcu kompilátor, či má pre objekt vytvárať samostatný modul.

Kapitola 4

Použiteľnosť a testovanie

V tejto kapitole práce sa pozrieme na to aký výsledok prinieslo naše snaženie. Pozrieme sa na to ako sa prakticky programuje v jazyku *DataWolf* a ako sa nám v ňom podarilo naprogramovať hašovací algoritmus SHA-2. Tiež si ukážeme akých výkonov dosahuje naša implementácia v porovnaní s inými jazykmi a platformami.

4.1 Efektívnosť programovania

Rýchlosť programovania v našom jazyku sa ukázala ako celkom dobrá. Tak ako aj pri iných jazykoch závisí od dostupných knižníc operácií. Tá je pri našom jazyku skromná, takže sa môžu vyskytnúť situácie, keď niečo naprogramovať sa stane veľmi zdĺhavým. V testovacom príklade je takým prípadom modul *rotujPole*, ktorý zoberie 16 prvkové pole, zahodí prvý prvok a zvyšné posunie, čím vznikne 15 prvkové pole. Implementácia tohto modulu nebola komplikovaná z dôvodu náročnosti algoritmu, ale z nedostatku vhodných nástrojov.

Ako sme si už spomenuli, tak operácie jazyka boli priebežne dopĺňané práve na implementácii algoritmu *SHA-2*. Avšak pri pôvodnej implementácii algoritmu priamo v jazyku Lucid sme túto funkcionalitu nepotrebovali. Tam sa nám podarilo program vytvoriť dostatočne malý, aby sa zmestil na naše FPGA zariadenie. Kód vygenerovaný naším kompilátorom túto vlastnosť však veľmi tesne nespĺňal. Toto sme úspešne vyriešili niekoľkými drobnými optimalizáciami, ktoré si neskôr popíšeme. Pravdepodobný dôvod, prečo sme tento problém zaznamenali, bol ten, že pri písaní priamo v jazyku Lucid sú možnosti ako niektoré úkony urobiť o trocha efektívnejšie. Toto sme však nepovažovali za veľký problém, pretože je bežná vlastnosť programovacích jazykov, že kód napísaný vo vyššom programovacom jazyku je menej efektívny ako kód napísaný v nižšom.

Optimalizácia algoritmu sa týkala hlavne pamäťovej náročnosti. Použitie registrov na uchovanie dát je totižto veľmi náročné na zdroje. Treba však ešte poznamenať, že keby sme sa nesnažili dosiahnuť maximálny možný výkon a uspokojili sa s tým, že výpočet bude trvať dlhšie, tak aj v knižnici operácií nášho jazyka boli prostriedky, ako zmienený modul *rotujPole* navrhnúť jednoduchšie a rýchlejšie.

4.2 Algoritmus SHA-2

Ide o hašovaciu funkciu, ktorá je vo veľkej miere používaná v kryptografii. Aj keď pochádza ešte z roku 2001 a už má nástupcu SHA-3, tak jej výskyt je stále ešte veľmi bežný. Konkrétne využitie je v rôznych bezpečnostných protokoloch, pri hašovaní hesiel alebo aj pri niektorých kryptomenách ako je Bitcoin. Má niekoľko variant, ktoré udávajú počet bitov výsledného hašu. My sme pri testovaní použili 256-bitovú verziu.

Na nasledujúcich riadkoch si popíšeme dôležitú terminológiu, ktorú budeme ďalej používať a aj ako tento algoritmus vlastne vyzerá.

4.2.1 Konštanty

Každá verzia algortimu SHA-2 používa nejakú sadu konštánt. 256-bitová verzia používa ako iniciálne hodnoty (označované ako *h0* až *h7*) prvých 32 bitov desatinnej časti druhých odmocnín prvých ôsmych prvočísel (2 až 19). Výpočet prebieha v 64 krokoch a v každom je ešte použitý jeden prvok z poľa konštánt k. To obsahuje prvých 32 bitov desatinných častí tretích odmocnín prvých 64 prvočísiel (2 až 311).

4.2.2 Predvýpočet

Zadanú správu, ktorú chceme hašovať musíme nasledovne upraviť. K pôvodnej správe dĺžky L bitov pridáme jeden bit, ktorý má hodnotu 1. Následne pridáme K nulových bitov, kde k >= 0, také že L + 1 + k + 64 je nejakým násobkom čísla 512. Následne pridáme ešte počet bitov originálnej správy L ako 64-bitový big-endian integer. Týmto predvýpočtom upravená správa má teda dĺžku, ktorá je násobkom čásla 512.

4.2.3 Pseudokód algoritmu

Samotný algoritmus začína rozdelením správy, ktorú sme si upravili na bloky po 512 bitoch. Potom postupne v cykle prejdeme cez všetky tieto bloky. Spracovanie každého bloku začína vytvorením 64 prvkového poľa w zloženého z 32-bitových slov. Dáta zo

spracovávaného bloku skopírujeme do prvých 16 slov poľa *w*. Potom na základe týchto prvých 16 prvkov dopočítame zvyšných 48.

```
1 for i from 16 to 63
2   s0 := (w[i-15] >>> 7) xor (w[i-15] >>> 18) xor
       (w[i-15] >> 3)
3   s1 := (w[i-2] >>> 17) xor (w[i-2] >>> 19) xor (w[i-2]
       >> 10)
4   w[i] := w[i-16] + s0 + w[i-7] + s1
```

Listing 4.1: Výpočet zvyšných prvkov poľa w

Nasleduje inicializácia premenných *a* až *h* hodnotami *h0* až *h7*. Tie obsahujú, pri spracovaní prvého bloku správy, konštanty uvedené v predchádzajúcej sekcii. Pri ďalších blokoch sú tam už uložené čiastočné výsledky hašovania.

```
1  for i from 0 to 63
2    S1 := (e >>> 6) xor (e >>> 11) xor (e >>> 25)
3    ch := (e and f) xor ((not e) and g)
4    temp1 := h + S1 + ch + k[i] + w[i]
5    S0 := (a >>> 2) xor (a >>> 13) xor (a >>> 22)
6    maj := (a and b) xor (a and c) xor (b and c)
7    temp2 := S0 + maj
8
9    h := g
10   g := f
11   f := e
12   e := d + temp1
13   d := c
14   c := b
15   b := a
16   a := temp1 + temp2
```

Listing 4.2: Hlavný výpočet hašovacej funkcie

Po tomto kroku už prichádza na rad cyklus, v ktorom prebieha hlavný výpočet. Ten beží 64-krát a v každom jeho chode sa aktualizujú premenné *a* až *h* na základe

hodnôt z predchádzajúcich chodov. To znamená, že toto je limit maximálnej možnej paralelizácie. Čiže aj keď môžeme spraralelizovať jednotlivé výpočty, ktoré sa dejú v tomto cykle, tak cyklus musí aj tak prebehnúť 64-krát.

Na záver spracovania každého bloku sa vypočítané hodnoty v premenných a až h pripočítajú k hodnotám *h0* až *h7*.

Po spracovaní všetkých blokov správy výslednú hodnotu hašu získame zreťazením hodnôt *h0* až *h7*.

4.3 Implementácia SHA-2 v jazyku DataWolf

Výpočet hašu prebieha v module *sha*, ktorého časť môžeme vidieť na obrázku 4.5 neskôr v tejto kapitole. Teraz si popíšeme ako sme postupovali pri programovaní tohto modulu.

4.3.1 Optimalizácia algoritmu

Pri implementácii tohto algorimtu do nášho jazyka sme narazili na niekoľko problémov, pričom všetky spočívali v tom, že naše testovacie zariadenie obsahovalo FPGA čip s malým počtom programovateľných obvodov. Problém bol najmä v tom, že sme mali nedostatok miesta na ukladanie dát. Použili sme registre, ktoré sú oveľa náročnejšie na zdroje ako použitie špeciálnej pamäte. Tú sme ale použiť nechceli z dôvodu výraznej straty výkonu.

Dôvod bol nasledovný. Pristupovať do registrov je veľmi jednoduché. V každom cykle procesora sme mohli načítať z registrov ľubovoľné množstvo dát a ľubovoľné množstvo dát do nich aj zapísať. No v prípade tejto špeciálnej pamäte sme boli limitovaný na jednu pamäťovú operáciu za procesorový cyklus. Takže keby sme v programe potrebovali počas jedného kroku výpočtu päťkrát pristúpiť k našim uloženým dátam, tak s využitím tejto pamäte by sme na to potrebovali päť procesorových cyklov. Toto by sa ale odchýlilo od našej predstavy maximálnej možnej paralelizácie.

Aby sme mohli algoritmus implementovať, tak sme museli pristúpiť k niektorým optimalizáciám. Prvou bolo, že sme obmedzili dĺžku správy na 55 znakov, pričom každý znak mal veľkosť jeden bajt. Po vykonaní predvýpočtu dostaneme ďalších deväť bajtov čím naša správa získa veľkosť 64 bajtov alebo 512 bitov, čo je presne veľkosť jedného bloku. Neskôr v tejto kapitole si rozoberieme ako by bolo možné upraviť náš program, tak aby zvládol spracovať aj väčšie správy.

Druhá optimalizácia spočívala v spojení dvoch cyklov algoritmu. Výpočet prvkov

poľa w sme vložili do hlavného cyklu s tým, že prebieha iba od 16. prvku vyššie, keďže tie prvé počítať nie je nutné. Pri tejto optimalizácii nešlo o šetrenie pamäte, ale o zrýchlenie výpočtu.

V poslednej veľkej optimalizácii sme zmenšili pole w na 16 prvkov. Prišli sme totiž na to, že nie je nutné si pamätať všetkých 64 prvkov. Vždy na výpočet ďalšieho elementu stačilo posledných 16 prvkov. Takže v každom ďalšom kroku cyklu po šestnástom sme prvý prvok poľa zahodili, zvyšné prvky posunuli a nový vypočítaný prvok umiestnili na koniec poľa.

Po všetkých úpravách sa výpočet hašu zmenšil na jeden jediný cyklus, ktorého pseudokód vyzeral nasledovne:

```
1  for i from 0 to 63
2    if i <= 15
3      w_value = w[i]
4    else
5      s0 := (w[1] >>> 7) xor (w[1] >>> 18) xor (w[1] >> 3)
6      s1 := (w[14] >>> 17) xor (w[14] >>> 19) xor (w[14]
          >> 10)
7      w_value := w[0] + s0 + w[9] + s1
8      w = w[0:14] append w_value
9    ...
10   temp1 := h + S1 + ch + k[i] + w_value
11   ...
```

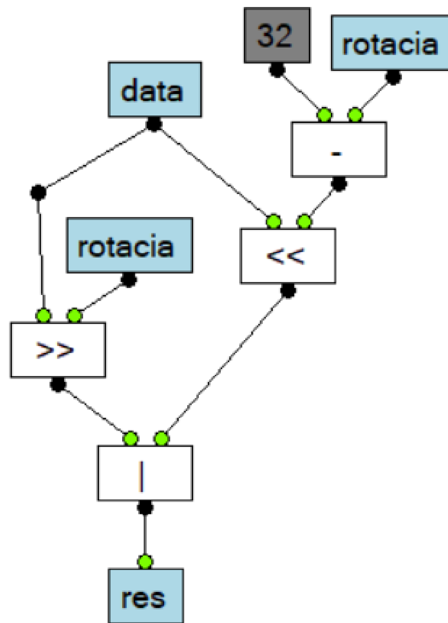
Listing 4.3: Výpočet hašu po našich úpravách. Zaznačené sú len zmenené riadky.

4.3.2 Program

Naša implementácia pozostáva z mnohých modulov, takže podrobné popísanie programu asi nie je vhodné, pretože by to zabralo veľký priestor práce. Napriek tomu si ukážeme aspoň niekoľko ukážok.

Na obrázku 4.1 môžeme vidieť implementáciu bitovej rotácie doprava. Vo výpočte sme túto operáciu niekoľkokrát potrebovali, a keďže sme ho nemali v knižnici jazyka, tak sme si ho museli sami navrhnúť. Nejde o veľmi zložitý výpočet, ale možno by bolo v budúcnosti vhodné túto operáciu do nášho jazyka zaradiť.

V algoritme máme niekoľko modulov, ktoré počítajú rôzne medzivýsledky.



Obr. 4.1: Modul ktorý zrotuje doprava *data* o *rotacia* počet bitov.

Napríklad modul *s0* vyráta hodnotu jedného sčítanca súčtu potrebného na výpočet hodnôt poľa *w*. Ako vyzerá môžeme vidieť na obrázku 4.2. Veľmi podobne vyzerajú aj ďalšie moduly, ktoré sme v programe využili.

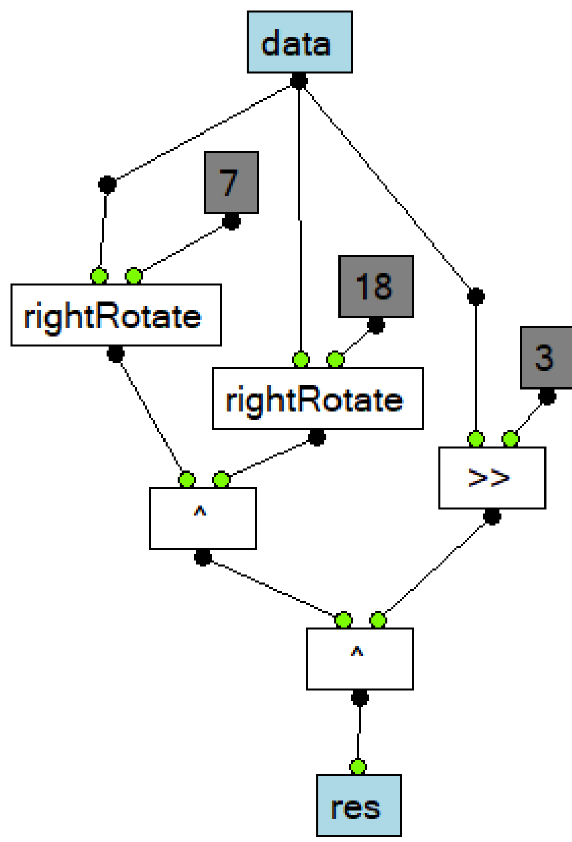
Na koniec sa ešte pozrieme na modul *hexToAscii*, ktorého graf je na obrázku 4.3. Ten použijeme pri konečnom vypísaní výsledku hašovania.

4.3.3 Komunikácia s programom

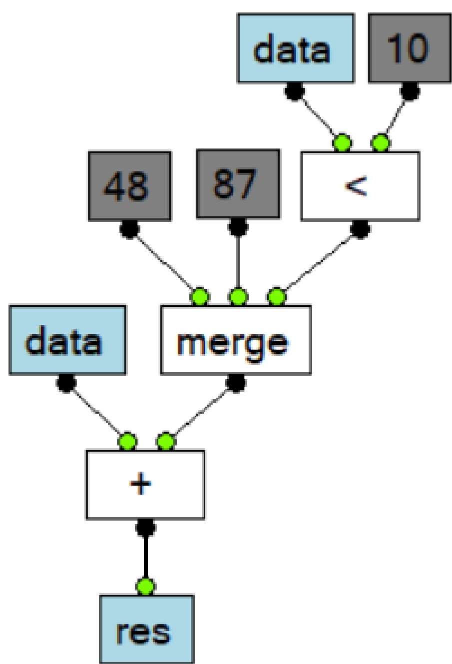
Zabezpečenie vstupov a výstupov je v module *main*. Aby sme mohli jednoducho s výsledným programom pracovať a testovať jeho funkčnosť, tak sme komunikáciu urobili interaktívnu. Teda v prostredí Mojo IDE je monitor sériového portu, v ktorom sa dajú zadávať vstupné dáta. FPGA je nadizajnované tak, aby zadané dáta ihneď vrátilo na výstup, čo nám umožní vidieť vložený text. Po stlačení klávesy enter alebo po dosiahnutí maximálneho počtu znakov sa spustí výpočet a vypíše sa výsledok. Toto môžeme vidieť na obrázku 4.4.

4.4 Porovnanie výkonu s inými jazykmi

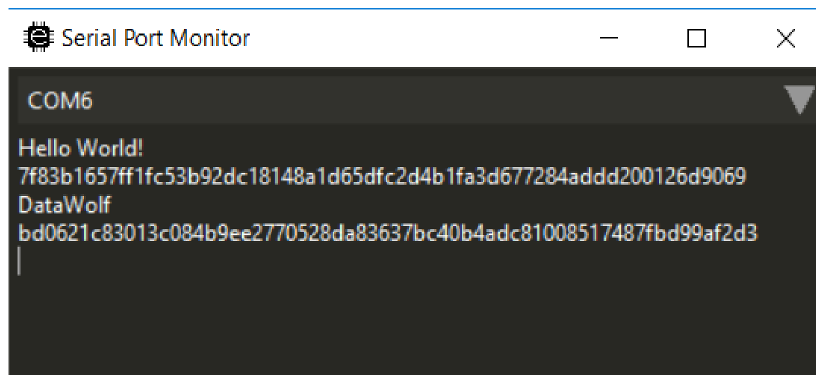
Implementovať algoritmus SHA-2 sme si zvolili z dôvodu, že je to ideálny príklad použitia nášho jazyka. Nemá veľké pamäťové požiadavky a obsahuje dosť výpočtov, ktorých paralelizáciou sa dalo dosiahnuť veľkého zrýchlenia.



Obr. 4.2: Výpočet hodnoty *s0*.



Obr. 4.3: Prevod hexadecimálnej hodnoty do ASCII.



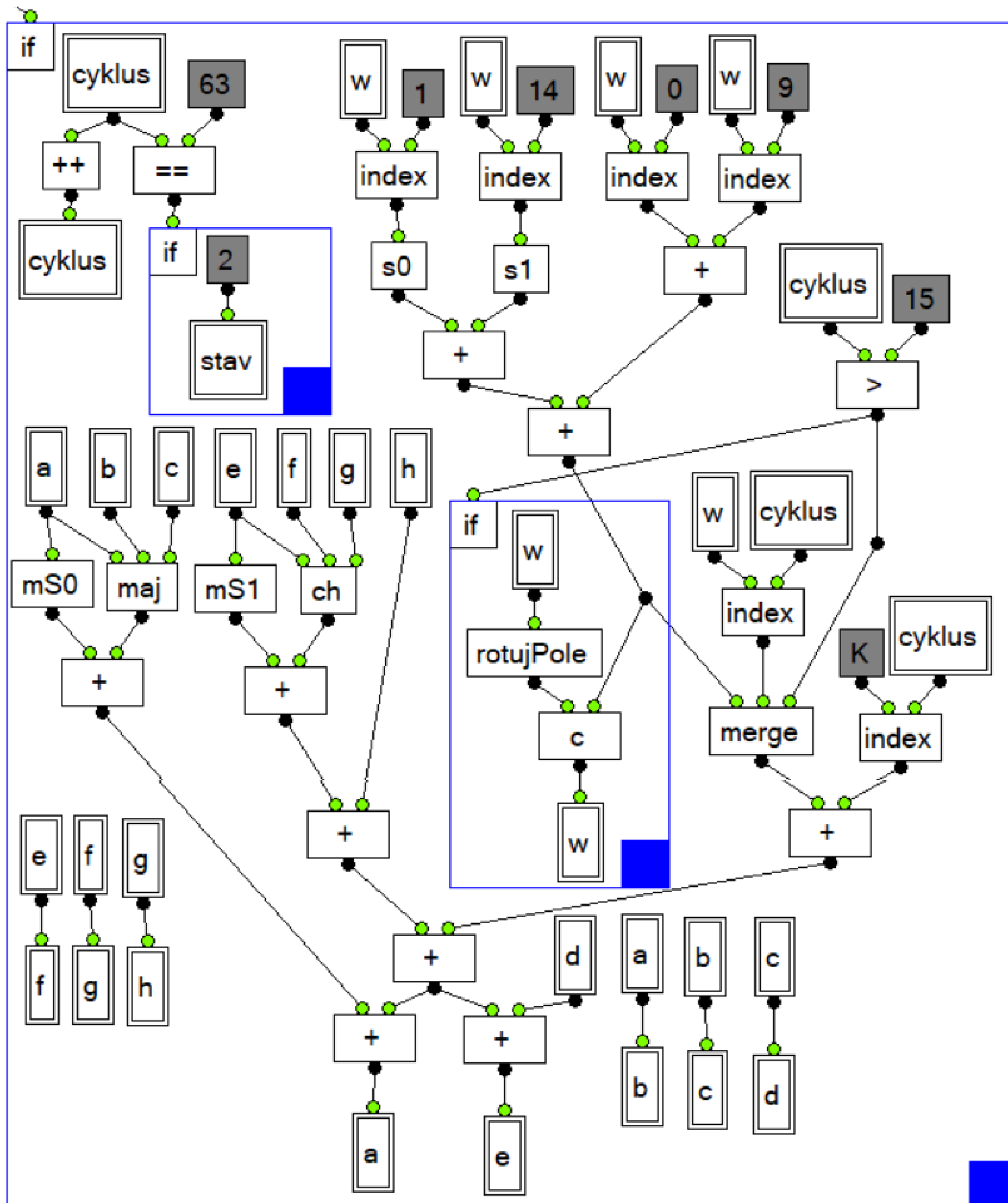
Obr. 4.4: Komunikácia s FPGA a testovanie hašovania.

4.4.1 Rýchlosť výpočtu v jazyku DataWolf

Náš jazyk umožňuje zakomponovať do programu vlastné počítadlo cyklov procesora. Avšak teraz sme niečo také nepotrebovali. Pri testovanom algoritme sa to dá presne spočítať už pred vytvorením programu. Ak nebudeme rátať zadanie vstupu a vypísanie konečného výsledku, tak sa nám stačí pozrieť na modul *sha*. Ten je zložený z troch častí. Prvá a posledná nie je veľmi zaujímavá, takže stačí keď si ich iba zbežne opíšeme. V tej prvej sa v prípade, že z hlavného programu v module *main* príde informácia o tom, že vstupné dáta sú už pripravené, inicializujú všetky potrebné hodnoty. Tretia a posledná časť výsledok výpočtu len vhodne pripraví na výstup. Obidve tieto časti trvajú presne jeden cyklus procesora.

Významná je práve druhá časť. Tu bude prebiehať hlavný výpočet hašu. Môžeme ho vidieť na obrázku 4.5. Register *cyklus* je ekvivalentom premennej cyklus i v pseudokóde. Výpočet niektorých medzivýsledkov sme rozdelili do modulov *s0*, *s1*, *mS0*, *mS1*, *maj* a *ch*. Snažili sme sa pri tom dodržať menné konvencie z pseudokódu. V kapitole 2 sme spomínali, že nie je nutné, alebo dokonca zbytočné, dávať základné operácie do podmienkových blokov. Napriek tomu sme sa pri vytváraní programu rozhodli to urobiť. Vďaka tomu je program prehľadnejší a umožňuje ľahšiu orientáciu v jednotlivých častiach programu.

Táto časť výpočtu trvá presne 64 cyklov procesoru. Čo je ako sme si vysvetlili v predchádzajúcej sekcii maximálna možná miera paralelizácie, ktorá sa dala dosiahnuť. Spolu nám teda výpočet hašu jedného 64 bajtového bloku trvá 66 cyklov procesora. Čo je po prepočte 1,03 cyklu na bajt dát.



Obr. 4.5: Výpočet hašu v hlavnej časti modulu *sha*

4.4.2 Rýchlosť výpočtu na iných platformách

Využitie algoritmu SHA-2 je veľmi široké a potreba počítať veľké množstvo hašov za čo najkratší možný čas spôsobila, že vzniklo mnoho rôznych implementácií, ktoré sa tento najkratší čas snažili docieliť. Na klasických procesoroch sú výsledky dosť ovplyvnené inštrukčnými sadami, ktoré tieto procesory majú. No v štandardnom von Neumannovom modeli sú možnosti paralelizácie u tohto algoritmu veľmi obmedzené, takže aj tie najlepšie implementácie dosahujú len okolo desiatich cyklov na bajt dát [11] (čo je výrazne menej ako pri našom riešení).

V praxi sa preto používa paralelizácia na hašovanie viacerých nezávislých vstupných dát. To je možné vďaka veľkému množstvu dát, ktoré sú procesory alebo grafické karty schopné načítať (narozdiel napríklad od nášho FPGA zariadenia).

Pre predstavu o tom aké najlepšie možné výsledky sa v prípade tohto algoritmu môžu dosahovať, sme sa pozreli na jednu z najvýkonnejších grafických kariet súčasnosti Nvidia GTX 1080 Ti. V tomto teste [12] použili softvér Hashcat verzie 3.4, ktorý slúži na prelamovanie hesiel. Uvedený benchmark pre 256-bitovú verziu algoritmu SHA-2 dosahoval výsledkov 4453 MH/s (miliónov hašov za sekundu).

Nami použité FPGA nie je s týmto samozrejme porovnateľné. Beží totiž len na frekvencii 50 MHz, čo nám teoreticky dáva rýchlosť 0,7575 MH/s. To je výrazne menej ako pri vyššie spomenutej grafickej karte. Rozdiel vo výkone je daný aj tým, že pri teste grafickej karty prebiehalo paralelne hašovanie viacerých nezávislých blokov. Keby sme týmto spôsobom upravili našu implementáciu, tak by sme dostali oveľa lepšie výsledky. Problém je však pri zadávaní vstupov a získavaní výstupov. Náš spôsob komunikácie totiž umožňuje rýchlosť iba jeden bajt za cyklus procesora.

Ako sme v práci viackrát spomenuli, tak je v súčasnosti pri výpočtovej technike dôležitým faktorom aj energetická náročnosť. Naše FPGA zariadenie nadizajnované podľa vytvoreného programu pre výpočet hašu dosahovalo maximálnu spotrebu 0,029 W. Na meranie sme použili softvér od výrobcu FPGA čipu Xilinx XPower Analyzer. Výstup merania môžeme vidieť na obrázku 4.6.

On-Chip	Power (W)	Used	Available	Utilization (%)
Clocks	0.005	2	---	---
Logic	0.004	2298	5720	40
Signals	0.002	2884	---	---
DSPs	0.000	2	16	13
IOs	0.001	22	102	22
Leakage	0.018			
Total	0.029			

Obr. 4.6: Meranie spotreby FPGA nadizajnovaného na výpočet SHA-2.

Spotreba grafickej karty Nvidia GTX 1080 Ti, ktorú kvôli testu ešte pretaktovali na maximálnu možnú mieru, dosahovala hodnotu 300 W. Keby sme výsledky prepočítali na 1 W, tak naše riešenie pri tejto spotrebe dokáže zrátať viac ako 26 MH/s, zatiaľ čo grafická karta len necelých 15 MH/s. Takže z toho vyplýva, že aj keď je nami použité riešenie výrazne pomalšie, tak má aspoň nižšiu spotrebu. Treba však ešte zobrať do úvahy aj to, že išlo o jedno z najlacnejších FPGA na trhu. Zatiaľ čo spomenutá grafická karta je jedným z najvýkonnejších modelov súčasnej generácie grafických kariet. Výsledky porovnania FPGA a grafických kariet v kapitole 1 boli podobného charakteru.

Takže sme zistili, že naše riešenie síce nedosahuje najlepších výsledkov, no napriek tomu má veľký potenciál do budúcnosti. Rozhodne aspoň pri niektorých typoch výpočtov.

4.5 Ďalšie možné úpravy implementácie algoritmu

4.5.1 Odstránenie podmienkových blokov

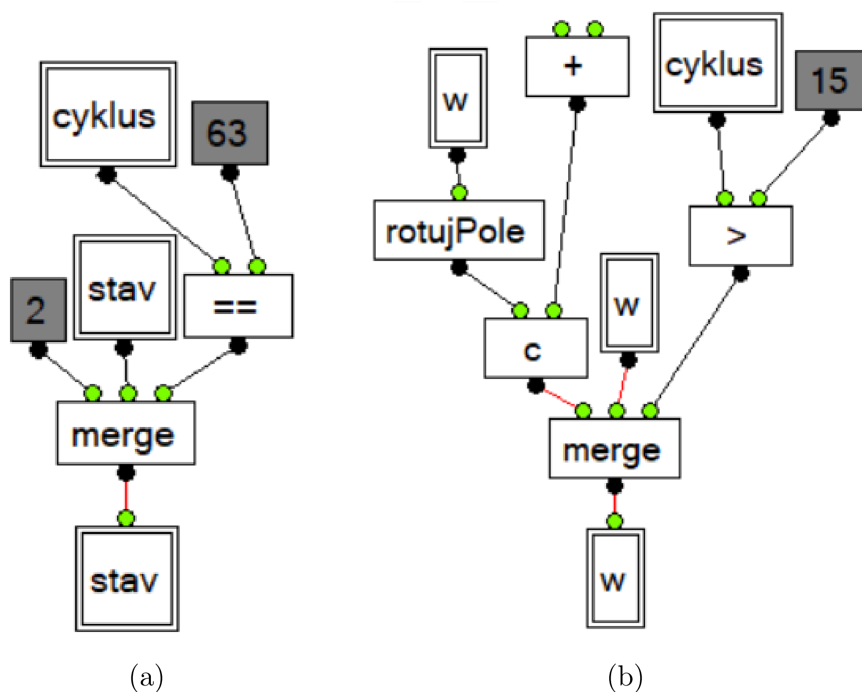
Keďže použitie podmienkových blokov sa odchyľuje od myšlienky dátových tokov, tak by bolo dobré preukázať, že ich používanie nie je nevyhnutné. Môžeme sa o tom presvedčiť napríklad odstránením dvoch vnútorných podmienkových blokov výpočtu hašu z obrázka 4.5.

Výsledok môžeme vidieť na obrázkoch 4.7a a 4.7b. Červeno označené spojenia znázorňujúce chybné priradenie dát sú spôsobené tým, že sme v ukážke použili operáciu *merge* navrhnutú priamo v našom vývojovom prostredí. Takéto operácie aktuálne nemajú možnosť dynamicky prispôsobovať si veľkosť rozmerov vstupných a výstupných dát.

Keďže pri tejto úprave išlo len o málo operácií, ktoré sa v týchto blokoch vykonávali, tak bol prevod jednoduchý. Keby sme ale chceli zmeniť týmto spôsobom niektorý z väčších podmienkových blokov, tak by program bol omnoho komplikovanejší a neprehľadnejší. To dostatočne ospravedlňuje narušenie paradigmy dátových tokov.

4.5.2 Rozšírenie výpočtu pre dlhšie správy

Dalo by sa namietať, že naše dobré výsledky boli spôsobené len zjednodušením algoritmu na správy dĺžky maximálne 55 znakov. Faktom ale je, že aj s dlhšími správami by bolo možné získať obdobné výsledky. Program fungoval tak, že po načítaní vstupných dát, ktoré trvalo od 1 po 55 cyklov procesora (podľa dĺžky správy) nasledoval



Obr. 4.7: Úprava časti hašovacieho modulu nahradením podmienkových blokov operáciou *merge*.

výpočet trvajúci 66 cyklov. Keby sme chceli dlhšie správy, tak by sa dalo postupovať tak, že zatiaľ čo beží výpočet, tak je možné načítavať ďalší vstup. Toto načítanie by bolo o niečo kratšie ako výpočet, takže by sme museli pridať nejaké zdržanie. Trocha zložitejšie by bolo zabezpečenie správneho ukončenia správy, ktoré v predvýpočte algoritmu získava niekoľko bajtov navyše, ako je napríklad celkový počet bitov správy.

Vo výpočtovej časti by bolo treba zmeniť iba inicializačný a koncový blok operácií. V prípade, že by sme hašovali iba jeden z blokov dlhšej správy, tak by sme nemohli niektoré registre nastaviť na iniciálne hodnoty, lebo by sme prišli o čiastočný výsledok hašovania. Tiež pri výstupe by sme potrebovali výsledný haš až na konci správy, respektíve by bolo treba nejako zabezpečiť, aby sa čiastočný výsledok neposlal na výstup modulu *main*.

Kapitola 5

Možnosti do budúcnosti

Náplňou našej práce bolo hlavne získať prehľad o modeli dátových tokov a jeho využití. Ďalej sme navrhli jednoduchý jazyk založený na tomto modeli a grafické vývojové prostredie s kompilátorom do platformy FPGA. Tento jazyk má však stále priestor pre zlepšenie. Rovnako aj naše vývojové prostredie, z ktorého sa postupne stal veľmi mohutný program, čo znamená, že môže obsahovať rôzne nedokonalosti. V tejto časti si popíšeme, ktoré vylepšenia by v budúcnosti stáli za zmienku.

5.1 Rozšírenie knižnice jazyka

Ako sme spomenuli v predchádzajúcej kapitole, tak je pri programovacích jazykoch dôležitá paleta dostupných operácií. Jazyk *DataWolf* síce obsahuje všetky základné, no bolo by veľkým prínosom, keby sme ho rozšírili o niektoré ďalšie dôležité, ktoré by uľahčili vývojový proces. Ide napríklad o nasledovné operácie:

- Merge a switch - ide o operácie z teórie modelu dátových tokov z kapitoly 1. Počas implementácie jazyka sme ich nahradili podmienkovým blokom. No napriek tomu je užitočné mať takéto operácie priamo ako súčasť jazyka. Konkrétne operáciu merge sme využili aj v našom testovacom príklade algoritmu SHA-2. Tam sme túto operáciu naprogramovali ako vlastný modul, ktorý používa 2 podmienkové bloky.

- Operácie s poliami - tieto operácie sa v súčasnej podobe jazyka dajú síce jednoducho naprogramovať, no len v sériovej a nie paralelnej verzii. Preto sme sa v niektorých častiach testovacieho príkladu priveľmi opakovali. Program však napriek tomu fungoval rýchlo a správne. Vyžadovalo to len vyššie úsilie pri programovaní.

- Premenlivý počet vstupov operácií - občas je príliš časovo náročné zapísať napríklad desať sčítaní alebo násobení. Preto by bolo užitočné u takýchto typov operácií doplniť možnosť viacerých vstupov.

Časť uvedenej funkcionality bola už pripravená na implementáciu, no k tej nedošlo z dôvodu nedostatku času. Ten sme radšej využili na odladenie aktuálnej verzie, pretože sme uprednostnili dobre fungujúci program s menšou funkcionalitou pred programom s viacerými modulmi, ale nedostatočne otestovanými.

5.2 Vynechanie Mojo IDE z vývojového procesu

V prípadných ďalších verziách jazyka *DataWolf* by bolo pravdepodobne vhodnejšie urobiť kompilátor do Verilogu. To by zabezpečilo lepšiu použiteľnosť aj na zariadeniach iných výrobcov. V súčasnosti sa však pre tieto prípady dá využiť aj prostredie *Mojo IDE*. Jedinou ďalšou funkcionalitou tohto prostredia, ktorú sme využívali je komunikácia s FPGA zariadením, ktorá by sa takisto dala preniesť do nášho vývojového prostredia. Týmito krokmi by sme vynechali prostredný článok medzi naším jazykom a dizajnovaním FPGA pomocou Verilogu.

Zmienili by sme ešte, že jedna z operácií sa už teraz prekladá priamo do Verilogu (*Mojo IDE* podporuje prácu s jazykom Lucid ako aj Verilog). Bolo to nutné, nakoľko sme až pri záverečnom testovaní zistili, že jazyk Lucid nepodporuje operátor modulo. Tento sa dá síce zostaviť pomocou iných operácií, no keďže sme už mali definovanú knižnicu jazyka konkrétne s touto operáciou, tak sme sa rozhodli vzniknutý problém vyriešiť kompiláciou daného modulu priamo do Verilogu, ktorý zmienený operátor podporuje. Takže sme položili základy pre budúcu transformáciu kompilátora do ďalšieho jazyka.

5.3 Rozšírenie možností vývojového prostredia

Keďže jazyk, ktorý sme navrhli je primárne grafický, tak sme na zobrazenie programov napísaných v tomto jazyku vyvinuli vývojové prostredie. Toto však plní len základnú funkcionalitu. Priestor pre zlepšenie vidíme v doplnení možností rýchleho vývoja vďaka hromadnému označovaniu operácií, čo by umožnilo tieto operácie pohodlne presúvať, mazať a prípadne aj kopírovať. Pri programovaní by sme tieto moduly rozhodne ocenili, pretože by to výrazne zrýchlilo vývojový proces.

Spomenuté hromadné označovanie by mohlo fungovať na podobnom princípe ako podmienkové bloky. Ich presúvaním sa zároveň hýbu aj operácie v nich obsiahnuté.

Takže možné riešenie tohto problému by bolo napríklad v triede odvodenej od triedy *IfBlok*. Objekt tejto triedy by sa samozrejme vykreslil inak a bol by iba dočasný.

Ďalším vylepšením by mohla byť dôkladnejšia analýza zdrojových súborov. V súčasnosti v prípade poškodeného súboru nie je možné otvoriť ani časť projektu, ktorá poškodená nebola. Užitočným by bol aj pokus o opravu takéhoto poškodeného súboru, čo by sa dalo využiť v situácii, keď program alebo jeho časť chceme napísať ručne. Takýto program by bol po grafickej stránke neprehľadný, ale bolo by ho možné skompilovať.

Zobrazovanie nie syntaktických ale logických chýb nie je v aktuálnej verzii programu takisto úplne dokonalé. Pod pojmom logická chyba teraz rozumieme prípady, keď sa napríklad snažíme vložiť dáta väčších rozmerov na nejaké miesto menších rozmerov. Neustále v programe prebieha kontrola takýchto situácií, no je zobrazovaná len pomocou zčervenania príslušných spojení. Dobrým zlepšením by bol aj nejaký celkový prehľad naprieč všetkými modulmi projektu, ktorý by programátorovi jednoznačne prezradil miesto chyby, ktorá by tak mohla byť rýchlo opravená.

Problémom je aj menenie vstupov a výstupov modulov. Pokiaľ sa menia len informácie o ich rozmeroch alebo názvy, tak to až taký problém nie je. No keď sa zmení počet týchto vstupov alebo výstupov, tak by to mohlo spôsobovať rôzne komplikácie. Keďže elegentné riešenie by bolo veľmi náročné, tak sme sa rozhodli situáciu vyriešiť tak, že sa v celom projekte vo všetkých použitiach daného modulu resetujú vstupné, respektíve výstupné spojenia operácie.

Posledným navrhovaným zlepšením by bolo určite lepšie kontrolovanie užívateľských vstupov. Niektoré parametre jednotlivých operácií sa editujú ručne a v takom prípade sa ešte môže vyskytnúť situácia, že sa zadajú údaje, ktoré nemusia byť úplne korektné. Prípady, keď sa na nejaké číselné vstupy zadá nečíselná hodnota sú síce ošetrené, program nespadne a mal by užívateľovi vypísať informáciu o neuskutočnenej operácii. Avšak keď ide napríklad o pomenovávanie modulov alebo samotných projektov môžu stále nastať situácie, keď bude názov na pohľad korektný, no po skompilovaní sa zistí, že daný názov je neplatný. Preto je nevyhnutné v súčasnosti dodržiavať menné konvencie popísané v kapitole Návrh.

5.4 Debugovanie

Testovanie funkčnosti a hľadanie chýb v programe sa ukázalo ako najslabší článok nášho jazyka. Ten sme však nemohli veľmi ovplyvniť, nakoľko išlo o veľkú zdĺhavosť vytvorenia dizajnu FPGA z kódu vo Verilogu (preklad z Lucidu do Verilogu je takmer

okamžitý). Tento proces vykonáva softvér priamo od výrobcu FPGA čipu a obsahuje mnoho rôznych úkonov. Nebolo v našich možnostiach do toho veľmi zasahovať. Je tiež otázne, či by sa dali nejako razantne zrýchliť procesy, ktoré vymysleli ľudia, ktorí sa tým dlhodobo zaoberajú a dané zariadenie zostrojili.

Pri ďalšom vývoji jazyka by bolo asi nevyhnutné vyriešiť problém s debugovaním. Jednou z navrhovaných možností ako toto v budúcnosti vyriešiť je vytvorenie emulátora, ktorý by simuloval beh programu. Takýto emulátor by nemusel byť veľmi efektívny. Stačilo by, aby programátor mal prehľad o tom, čo sa deje v jeho programe.

Záver

Výsledkom tejto práce je navrhnutie nového jazyka založeného na dátových tokoch. Ide o grafický jazyk, takže súčasťou našej práce bola aj implementácia grafického vývojového prostredia s kompilátorom do platformy FPGA. Okrem toho sme v tomto jazyku naprogramovali algoritmus SHA-2, aby sme predviedli využiteľnosť jazyka pri paralelizácii výpočtov. Naše vývojové prostredie je napísané v jazyku C# a je voľne k dispozícii v online repozitári na stránke `https://github.com/px13/DataWolf` a tiež aj na médiu priloženom pri tejto práci.

Táto práca obsahuje prehľad teórie týkajúcej sa dátových tokov ako aj základné informácie o technológii FPGA. Ďalej sme sa zaoberali popisom toho ako sme postupovali pri navrhovaní jazyka a knižnice jeho príkazov. Uvedených je tu niekoľko príkladov využitia jazyka pri rôznych praktických výpočtoch. Navrhli sme aj spôsob ako dokázať, že je jazyk DataWolf univerzálny a má teda rovnakú výpočtovú silu ako Turingov stroj. Tiež sme si v práci rozobrali akým spôsobom prebiehala implementácia vývojového prostredia a kompilátora.

Testovanie naprogramovaného algoritmu SHA-2 ukázalo využiteľnosť jazyka aj pri riešení rôznych reálnych problémov. Takisto sme si potvrdili, že FPGA platforma prináša výraznú energetickú úspornosť, tak ako sme o tom hovorili v kapitole 1. Vývoj v našom jazyku sa navyše ukázal ako oveľa rýchlejší v porovnaní s použitím štandardných programovacích jazykov pre FPGA ako je Lucid alebo Verilog.

Priestor pre pokračovanie práce vidíme hlavne v rozšírení jazyka o ďalšie možnosti, ktoré by umožnili jednoduchší a rýchlejší vývoj. Tiež aj v ďalších návrhoch uvedených v záverečnej kapitole.

Literatúra

[1] Gustavo Alonso. Fpgas in data centers. *acmqueue*, 16(2):52–57, 2018.

[2] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[3] Alan L Davis and Robert M Keller. Data flow program graphs. *IEEE Comput.*, 15(2):26–41, 1982.

[4] Paul G Whiting and Robert SV Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, 1994.

[5] T Sterling. Studies on optimal task granularity and random mapping. *Advanced Topics in Dataflow Computing and Multithreading, 1995*, pages 349–365, 1995.

[6] Jeffrey Travis and Jim Kring. *LabVIEW for everyone: graphical programming made easy and fun.* Prentice-Hall, 2007.

[7] Xueyuan Su, Garret Swart, Brian Goetz, Brian Oliver, and Paul Sandoz. Changing engines in midstream: A java stream computational model for big data processing. *Proceedings of the VLDB Endowment*, 7(13):1343–1354, 2014.

[8] Philavanh V. Fpgas in neural networks. *Arrow Electronics*, 2017. Získané 26. apríla 2018, z https://www.arrow.com/en/research-and-events/articles/fpgas-in-neural-networks.

[9] Donald Thomas and Philip Moorby. *The Verilog® Hardware Description Language.* Springer Science & Business Media, 2008.

[10] Justin Rajewski. Learning fpgas: Digital design for beginners with mojo and lucid hdl. 2017.

[11] Gopal V Guilford J, Yap K. Fast sha-256 implementations on intel® architecture processors. *Intel Corporation*, 2012. Získané 26. apríla 2018,

z https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/sha-256-implementations-paper.pdf.

[12] Nvidia gtx 1080 ti hashcat benchmarks. 2018. Získané 26. apríla 2018, z https://gist.github.com/epixoip/973da7352f4cc005746c627527e4d073.

Prílohy

- CD obsahujúce zdrojový kód a elektronickú verziu práce vo formáte pdf