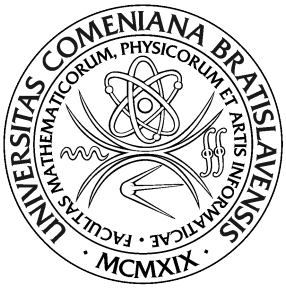
**UNIVERZITA KOMENSKÉHO V BRATISLAVE**

**FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY**



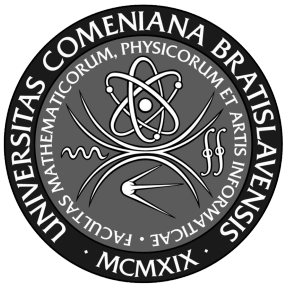

# VYŠŠÍ PROGRAMOVACÍ JAZYK DÁTOVÝCH TOKOV

Diplomová práca

2022 Bc. Martin Mašek

**UNIVERZITA KOMENSKÉHO V BRATISLAVE**

**FAKULTA MATEMATIKY, FYZIKY A INFORMATIKY**




# VYŠŠÍ PROGRAMOVACÍ JAZYK DÁTOVÝCH TOKOV

Diplomová práca

| | |
|---|---|
| Študijný program: | Aplikovaná informatika |
| Študijný odbor: | 2511 Aplikovaná informatika |
| Školiace pracovisko: | Katedra aplikovanej informatiky |
| Školiteľ: | Mgr. Pavel Petrovič, PhD. |

Bratislava, 2022 Bc. Martin Mašek


Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Bc. Martin Mašek
**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:** Vyšší programovací jazyk dátových tokov
*Higher level Data-Flow Programming Language*

**Anotácia:** Tradičný výpočtový model založený na von Neumannovej architektúre naráža na fyzikálne hranice a stáva sa neefektívnym pre moderné výpočtové aplikácie. Dátové centrá už nasadzujú do prevádzky FPGA obvody, ktoré umožňujú vysokú paralelizáciu výpočtov organizovaných v alternatívnych architektúrach. Tradičné jazyky založené na zápise riadenia toku výpočtu sú už nevyhovujúce. Je potrebné hľadať nové formalizmy a spôsoby zápisov algoritmov, ktoré sú prirodzené pre vývojárov a zároveň umožňujú efektívnu kompiláciu do nových architektúr využívajúcich paralelné výpočty. Medzi ne patria aj jazyky dátových tokov. Táto práca nadväzuje na predchádzajúcu diplomovú prácu, v ktorej bol navrhnutý nízkoúrovňový jazyk dátových tokov Datawolf. Cieľom tejto práce je preskúmať priestor rôznych verzií formalizmov pre grafický zápis dátových tokov a analyzovať ich vlastnosti. Na základe toho vybrať, navrhnúť a implementovať prototyp nového jazyka dátových tokov na vyššej úrovni a jeho vývojového prostredia s kompilátorom pre hardvérovú platformu FPGA a demonštrovať jeho využitie na príkladoch.

**Literatúra:**
Peter Kuljovský: Dátové toky ako paradigma pre paralelné programovanie, diplomová práca, FMFI UK, 2018.
Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in data-flow programming languages.ACM Computing Surveys (CSUR), 36(1):1–34, 2004.
Justin Rajewski: Learning FPGAs: Digital Design for Beginners with Mojo and Lucid HDL, O'Reilly Media, 2017.

**Kľúčové slová:** programovací jazyk dátových tokov, fpga, paralelné programovanie

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

**Dátum zadania:** 26.09.2018

**Dátum schválenia:** 26.02.2019

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



Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

.....  .....

študent  vedúci práce

Čestne prehlasujem, že túto diplomovú prácu som vypracoval samostatne len s použitím uvedenej literatúry a za pomoci konzultácií u môjho školiteľa.

.....

Bratislava, 2022 Bc. Martin Mašek

# Poďakovanie

Touto cestou by som sa chcel v prvom rade poďakovať môjmu školiteľovi Mgr. Pavlovi Petrovičovi, PhD. za jeho cenné rady a usmernenia, ktoré mi veľmi pomohli pri riešení tejto diplomovej práce. Takisto sa chcem poďakovať všetkým mojím kamarátom a celej mojej rodine za podporu počas môjho štúdia.

# Abstrakt

MAŠEK, Martin: Vyšší programovací jazyk dátových tokov [Diplomová práca]. Univerzita Komenského v Bratislave. Fakulta matematiky, fyziky a informatiky; Katedra aplikovanej informatiky. Vedúci práce: Mgr. Pavel Petrovič, PhD. Bratislava: FMFI UK, 2022. 118 strán.

Na paralelné výpočty sa v poslednej dobe kladie rastúci dôraz. Spracúva sa oveľa väčšie množstvo dát a napriek rastúcemu výkonu hardvéru je rýchlosť výpočtov neuspokojivá, prípadne veľmi energeticky náročná. Čoraz častejšie sa preto využívajú platformy ASIC a FPGA. Zozbierali sme potrebné znalosti z paralelizácie, platformy FPGA a dátových tokov. Skúmali sme rôzne verzie grafického zápisu dátových tokov, umožňujúce vytvárať paralelné programy, ktoré budú pre programátora jednoduchšie, zrozumiteľnejšie a prehľadnejšie. Analyzovali sme možnosti implementácie vyššieho jazyka na platforme FPGA. Na základe experimentov sme vybrali a navrhli nový vyšší jazyk dátových tokov s názvom Draken-I, položili základy pre vývojové prostredie a kompilátor do jazyka Verilog určeného pre konfiguráciu FPGA. Využitie a implementáciu jazyka demonštrujeme na výpočte faktoriálu.

**Kľúčové slová:** Dátové toky, FPGA, Verilog, paralelizácia

# Abstract

MAŠEK, Martin: Higher level Data-Flow programming language [Master thesis]. Comenius University, Bratislava. Faculty of Mathematics, Physics and Informatics; Department of Applied Informatics. Supervisor: Mgr. Pavel Petrovič, PhD. Bratislava: FMFI UK, 2022. 118 pages.

Recently, there has been a growing emphasis on parallel calculations. A much larger amount of data is processed and despite the growing performance of the hardware, the speed of calculations is unsatisfactory or highly energy-requiring. Therefore, the usage of ASIC and FPGA platforms is increasing. We gathered the available knowledge about parallelization, FPGA platform and dataflow. We have researched different ways of graphical data representation, which would allow the creation of parallel programs that will be easier, clearer and more understandable for the programmers. We analyzed the possibilities of implementing the proposed higher language on the FPGA platform. Based on the experiments, we selected and designed a new higher dataflow language called Draken-I. We laid the foundations for a development environment and compiler in Verilog for FPGA configuration. We demonstrate the usage and implementation of the language on the factorial calculation.

**Keywords: :** DataFlow, FPGA, Verilog, parallelization

# Obsah

# Úvod

Tradičný výpočtový model založený na riadiacom toku (control flow) von Neumannovej architektúry, vykonáva výpočet sekvenčne, inštrukciu po inštrukcii. Výpočet je možné zrýchliť zvýšením rýchlosti - frekvencie procesora. V tejto oblasti procesory v dnešnej dobe narážajú na fyzikálne hranice. Ďalšou možnosťou ako zvýšiť rýchlosť výpočtu je jeho paralelizácia.

Na paralelné výpočty sa v poslednej dobe kladie rastúci dôraz. Fyzikálne simulácie, spracovanie obrazu, veľké dáta (big data) a ďalšie oblasti informatiky prinášajú výpočty, ktoré sú zložitejšie. Spracúva sa oveľa väčšie množstvo dát a napriek rastúcemu výkonu hardvéru je rýchlosť výpočtov neuspokojivá, prípadne veľmi energeticky náročná.

Paralelné výpočty sa štandardne vykonávajú na univerzálnych viacjadrových procesoroch (general-purpose CPU), rozšírené sú aj špecializované platformy ako grafické procesory (GPU) a čoraz častejšie sa využívajú platformy ASIC a FPGA.

Paralelné programovanie prináša komplikovanejšiu tvorbu programov a algoritmov. Je potrebné prinášať nové rozšírenia do existujúcich jazykov, prípadne vytvárať úplne nové programátorské jazyky, umožňujúce vytvárať paralelné programy, ktoré budú pre programátora jednoduchšie, zrozumiteľnejšie a prehľadnejšie. Dátové toky sú jedným z možných

riešení.

V tejto práci skúmame rôzne verzie grafického zápisu dátových tokov. Analyzujeme možnosti implementácie vyššieho jazyka na platforme FPGA. Na základe experimentov navrhujeme nový vyšší jazyk dátových tokov Draken-I. Navrhujeme základy pre vývojové prostredie a kompilátor do jazyka Verilog určeného pre konfiguráciu FPGA. Využitie a implementáciu jazyka demonštrujeme na výpočte faktoriálu.

# Kapitola 1

# Východiská

## 1.1 Konkurencia, paralelizácia

Paralelné výpočty používajú na vyriešenie problému, viacero súbežne bežiacich operácií v samostatných časových líniách, prípadne jednej časovej línii v rôznych etapách.

Konkurencia, pipelining je druh súbežného (zreťazeného, prekrývaného) vykonávania inštrukcií. Namiesto úplného sekvenčného vykonania inštrukcie (vykonanie jednej je dokončené pred začiatkom druhej) je spracovanie inštrukcií rozdelené na etapy (stage). Rôzne etapy môžu byť teda vykonané súčasne. Počas konkurentného vykonávania výpočtu každá etapa vykonáva svoju úlohu nad inou inštrukciou alebo dátami.

Pri paralelnom vykonávaní je možné vykonávať naraz viacero rovnakých etáp ich duplikáciou.

Takýto druh výpočtov je možné realizovať rozdelením problému na nezávislé časti. Toto rozdelenie úzko súvisí so spracovávanými dátami a spôsobe akým sa k nim pristupuje. Na zabezpečenie správneho poradia medzi aktivitami a dátami jednotlivých operácií sa používajú synchroni-

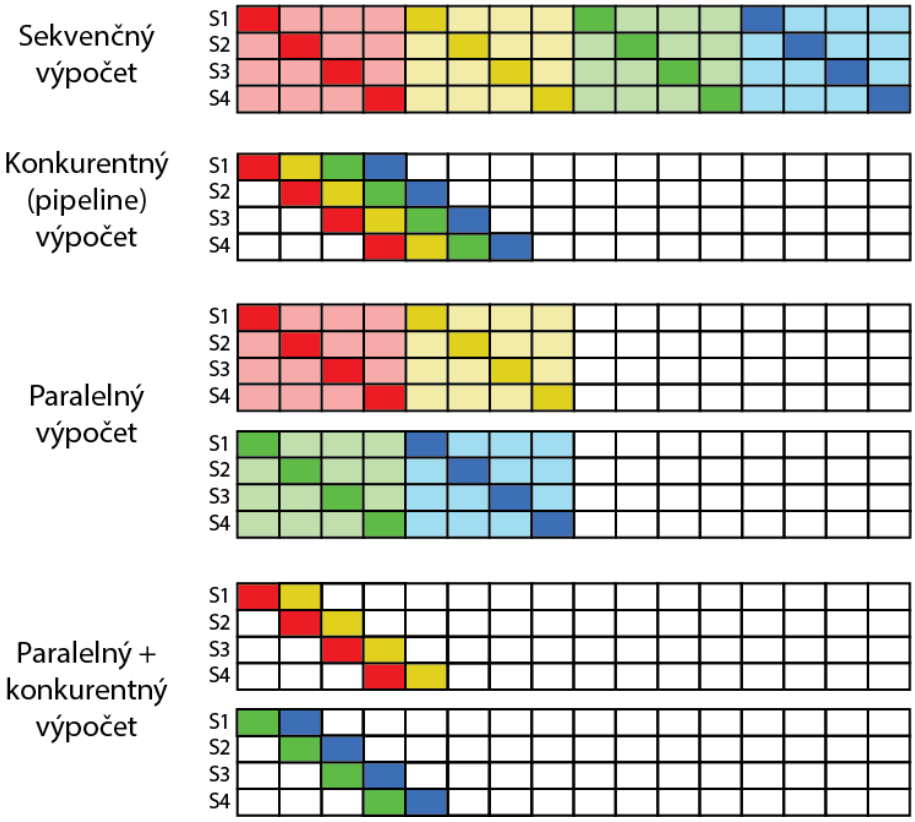
začné mechanizmy. [PH13]

Súčasné vykonávanie umožňuje vznik hneď niekoľkých nových tried potenciálnych chýb. Algoritmy využívajúce súbežné vykonávanie sú náročnejšie pre programátora počas tvorby a ladenia.

Jedným zo spôsobov prístupu k dátam je zdieľaná globálna pamäť, využívaná v architektúre riadiacich tokov. Ďalším spôsobom je použitie modelu dátových tokov. [PH13, Vee86]

Na obrázku 1.1 môžeme vidieť porovnanie sekvenčného a paralelných vykonaní. Inštrukcie zobrazené rôznymi farbami sú pri vykonávaní rozdelené na 4 etapy S1-S4 (stages), ktoré reprezentujú zjednodušený inštrukčný cyklus výpočtového modelu riadiacich tokov:

1. S1 - načítanie inštrukcie (fetch)
2. S2 - dekódovanie inštrukcie (decode)
3. S3 - vykonanie inštrukcie (execute)
4. S4 - uloženie výsledku (store)




Obr. 1.1: Porovnanie sekvenčného, konkurentného a paralelného vykonávania v modely riadiacich tokov.

Sekvenčný výpočet - nasledujúca inštrukcia sa začne vykonávať po úplnom dokončení predošlej.

Konkurentný výpočet - hneď po uvoľnení etapy S1 prvou inštrukciou, začne nasledujúca inštrukcia vykonávať etapu S1. Vykonávanie je prekrývané, v jednom momente dochádza ku konkurentnému vykonávaniu všetkých inštrukcií, každá ale vykonáva inú etapu inštrukčného cyklu.

Paralelný výpočet - etapy S1-S4 sú duplikované, vykonávanie prebieha zároveň na rovnakých etapách, avšak v dvoch samostatných paralelne vykonávaných inštrukčných cykloch, bez prekrývania.

Paralelný a konkurentný výpočet - etapy sú duplikované a zároveň

podporujú konkurenciu.

## 1.2 Výpočtové modely

Výpočtové modely špecifikujú správanie programovacích paradigiem, programovacích jazykov a hardvérových architektúr. V univerzálnych výpočtových procesoroch dominuje model riadiacich tokov, von Neumannova architektúra. Na opačnej strane množiny výpočtových modelov je model dátových tokov, používaný predovšetkým v špecifických doménach.

Architektúry sa v priebehu dejín odchýlili od pôvodných striktných výpočtových modelov riadiacich a dátových tokov. Hoci sú prístupy týchto modelov opačné, nie sú nezlučiteľné a vzájomne došlo k prienikom vedúcim k profitu na oboch stranách, v podobe zvýšenia výkonu a efektivity.

## 1.3 Model riadiacich tokov

Riadiaci tok (controlflow) je prúd inštrukcií v špecifickom poradí, ktoré vykonávajú výpočet na externých dátach. Podmienené vykonávanie, skoky a volania procedúr menia tok prúdu inštrukcií (instruction stack). Inštrukcie prúdia (operujú) nad dátami v registroch. Dáta sú statické, kým ich inštrukcia nezmení/nepresunie. Podmienka skočí na správnu vetvu výpočtového stromu, dáta sa pri tomto skoku nemenia. Riadiace toky majú presné poradie vykonania. [PH13, HH12]

Model riadiacich tokov vznikol povodne ako sériový, sekvenčný výpočet. Konkurentné (súbežné, pipeline) a paralelné vykonávanie bolo do riadiacich tokov dodatočne pridané. Paralelné procesy využívajú zdieľanie globálneho adresného priestoru, s ktorým pracujú asynchrónne pri

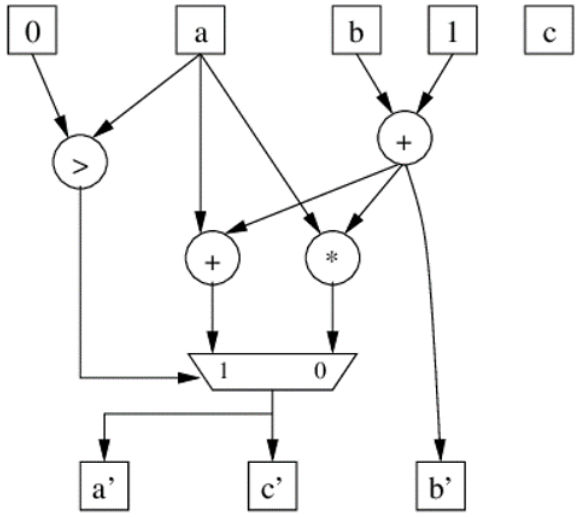
čítaní a zápise dát. Súčasný asynchrónny prístup môže vyvolať súbeh (race condition), ktorý spôsobí, že výsledky výpočtu sú pri nesprávnom poradí alebo načasovaní jednotlivých operácií nepredvídateľné. Tomuto problému je možné zabrániť pomocou synchronizačných mechanizmov ako sú semafory, monitory a zámky.

## 1.4 Model dátových tokov

Dátový tok (dataflow) je prúd dát posúvaný pri spracovaní z inštrukcie na inštrukciu. Podmienené vykonávanie, skoky a volania procedúr smerujú dáta do odlišných inštrukcií. Dáta prúdia cez statické inštrukcie. Podmienka nasmeruje dáta do konkrétnej inštrukcie. [Vee86, DK82, Ack82]

V modeli dátových tokov je program reprezentovaný pomocou orientovaného grafu, obr. 1.2, 1.3, 1.4. Globálna pamäť je zakázaná. Jednotlivé vrcholy predstavujú príkazy, inštrukcie ako aritmetické alebo porovnávacie operácie. Orientované hrany medzi vrcholmi predstavujú závislosti dát od operácií. Koncepčne dáta tečú po týchto hranách, ktoré vystupujú ako neviazané fronty FIFO (first-in, first-out).[Vee86, DK82, Ack82, JHM04]

```
int a, b, c;

void fct()
{
    b++;
    if (a > 0)
        c = a + b;
    else
        c = a * b;
    a = c;
}
```

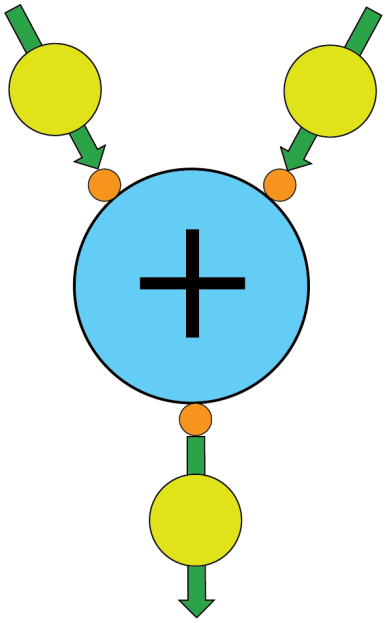



Obr. 1.2: Porovnanie textového programu riadiacich tokov a jeho grafového zápisu v dátových tokoch.

Názvy jednotlivých prvkov v dátových tokoch sa naprieč odbornou literatúrou mierne líšia. V tejto práci sa snažíme o jednotné pomenovávanie. Pre upresnenie uvádzame celý rad pomenovaní, s ktorými sa je možné stretnúť.

Na obrázku 1.3 je zobrazená operácia sčítania. Skladá sa z viacerých častí:

- vrchol, operácia, uzol, inštrukcia, (blackbox, node) - reprezentovaná modrým kruhom a operátorom plus
- hrana, spoj, spojenie, šípka, dátovod, (arc, port) - zobrazená zelenými šípkami
- port, vstup, výstup - oranžové kruhy symbolizujúce vstupné a výstupné porty operácie
- dáta, token - jednotka dát, prúdia po hranách a sú vyobrazené žltou



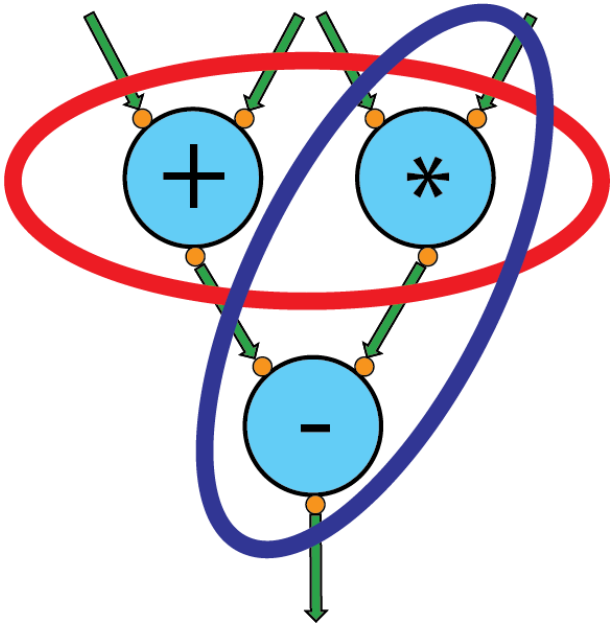
Obr. 1.3: Zobrazená operácia sčítania.

Každá operácia ma minimálne jeden vstup a jeden výstup. Operáciu je možné vykonať v momente, keď obdrží dáta na všetkých svojich vstupoch. Po skončení výpočtu pošle výsledok na výstup. Ak viaceré operácie dostanú dáta na svojich vstupoch a nie sú medzi nimi závislosti, sú vykonané paralelne. [Vee86, DK82, Ack82, JHM04]

Závislosti a paralelizáciu vizualizujeme na obr. 1.4. Červená elipsa označuje vrcholy (násobenie, sčítavanie), ktoré nie sú prepojené hranami. Neexistuje medzi nimi závislosť a môžu byť po obdržaní dát vykonané paralelne.

Prepojené vrcholy sú závislé na dodaní dát medzi sebou. Na obrázku 1.4 je modrou elipsou označená závislosť medzi vrcholmi vykonávajúcimi odčítavanie a násobenie. K rovnakej závislosti dochádza aj medzi vrcholmi sčítavanie a odčítavanie. Závislé vrcholy môžu pracovať súbežne (konkurentne). Vrchol odčítavanie spracúva dáta ktoré obdržal, súbežne

s vrcholmi + a *, ktoré spracúvajú d'alšie dáta v poradí. Podobne ako etapy pri inštrukciách v kapitole 1.1.




Obr. 1.4: Závislosti medzi operáciami.

Zároveň vidíme, že paralelné a konkurentné vykonávanie je prirodzenou vlastnosťou dátových tokov, kým do sekvenčného výpočtu riadiacich tokov bola paralelizácia doplnená neskôr.

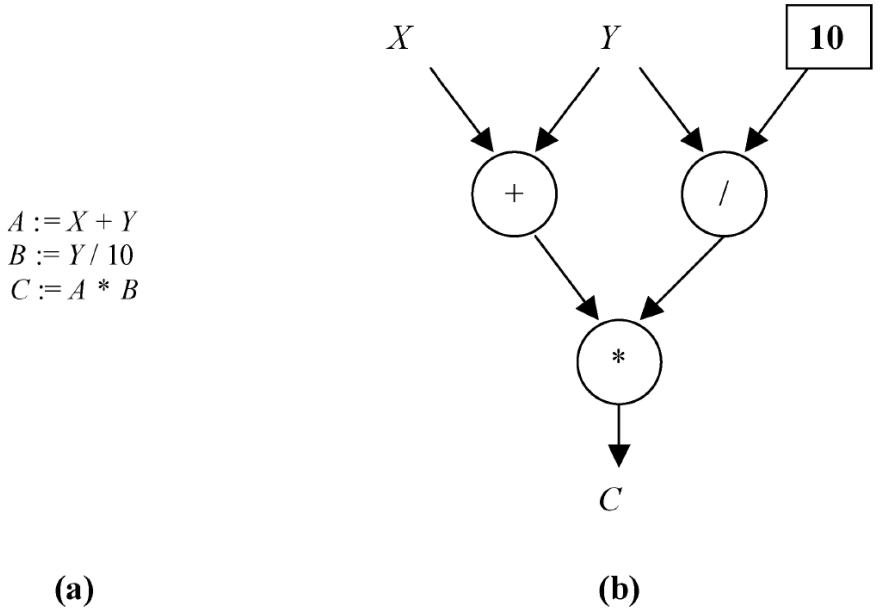
#### 1.4.0.1 Dátové vs. riadiace toky

V kontraste s dátovým tokom sú v riadiacom modeli (von Neumann) operácie vykonané až v momente, keď ukazovateľ vykonania (instruction/execution poiter) ukáže na danú operáciu, bez ohľadu na to, či operácia mohla alebo nemohla byť vykonaná skôr. Podstatnou výhodou dátových tokov je teda možnosť vykonávať viac ako jednu operáciu (inštrukciu) súčasne na základe dostupnosti dát. Tento princíp poskytuje potenciál pre masívnu paralelizáciu vykonávania operácií. [JHM04]

Obr 1.5 zobrazuje porovnanie riadiaceho modelu (a) a ekvivalent v dátových tokoch (b). Šípky predstavujú hrany, vrcholy jednotlivé operácie, štvorec konštantu, **X** a **Y** hodnoty vypočítané inými operáciami.

Program na obr. 1.5 (a) by po spustení bol vykonaný sekvenčne, počas 3 časových jednotiek. V prvej časovej jednotke sa sčítajú hodnoty premenných **X** a **Y** a výsledok je priradený do premennej **A**, následne dôjde k vypočítaniu podielu hodnoty **Y** a 10 a priradeniu výsledku do **B**. V poslednej časovej jednotke je hodnota súčinu **A** a **B** priradená do **C**. Program je vykonaný za 3 časové periódy.

V modeli dátových tokov obr. 1.5 (b) je výpočet spustený po obdržaní všetkých vstupných hodnôt. Medzi operáciami sčítania a delenia nie je závislosť (hrana). Ich vykonanie prebehne paralelne za jednu časovú jednotku. V druhej časovej jednotke prebehne operácia násobenia. Celý výpočet potrebuje na svoje vykonanie len 2 časové jednotky. [JHM04]




Obr. 1.5: Riadiaci tok vs. dátový tok. [JHM04]

Program v modeli dátových tokov môže vytvárať samostatný modul so svojimi vstupmi a výstupmi, ktorý môže byť opakovane použitý alebo môže byť vložený do zložitejšieho programu.

Počet operácií, ktoré je možné naraz vykonávať v sekvenčnom modeli je jedna resp. ak sú operácie v maximálnej možnej miere paralelizované, tak sú obmedzené počtom procesorov (výpočtových jadier procesoru). Pri modeli dátových tokov môže byť týchto operácií v teoretickej rovine neobmedzene veľa. V praxi sme limitovaní možnosťami použitého hardvéru, v tomto prípade FPGA, ktoré umožňuje rádovo tisíce takýchto operácií. [Vee86, JHM04]

### 1.4.1 Vlastnosti dátových tokov a jazykov

Teoretický model dátových tokov bol uvedený nezávisle vo viacerých publikáciách počas 60. a 70. rokov, spolu s konceptom grafového zobrazenia výpočtu. Vývoj programovacích jazykov podľa modelu dátových tokov sa uberal rôznymi smermi. V jednotlivých implementáciách sa vývoj uberal textovým alebo grafickým (grafovým) zápisom. Využili a rozšírili sa existujúce jazyky, alebo došlo k vývoju úplne nových. [Vee86, DK82, Ack82, JHM04]

Jazyky dátových tokov bývajú niekedy dávané do kontrastu s funkcionálnymi jazykmi, kvôli zdieľaniu určitých základných vlastností, ktoré sú v ostrom kontraste sekvenčných jazykov. Všetky tieto varianty majú spoločné charakteristiky, ktorými sa model a jazyky dátových tokov vyznačujú:

1. Poradie vykonávania.

   Sekvencia poradia vykonávania je ovplyvnená dátovými závislos-

ťami. Inštrukcie môžu bežať paralelne, ak nie sú vzájomne závislé - prepojené.

2. Sémantika jednorazového priradenia.

   Dáta riadia výpočet, preto je po deklarácii premennej a priradení hodnoty, zakázané túto hodnotu meniť. Premenné vystupujú skôr ako konštanty. Operácie majú zakázané vstupné premenné meniť. Po absorbovaní dát zo vstupu, vytvorí operácia novú hodnotu ako výstup.

3. Lokálnosť účinku.

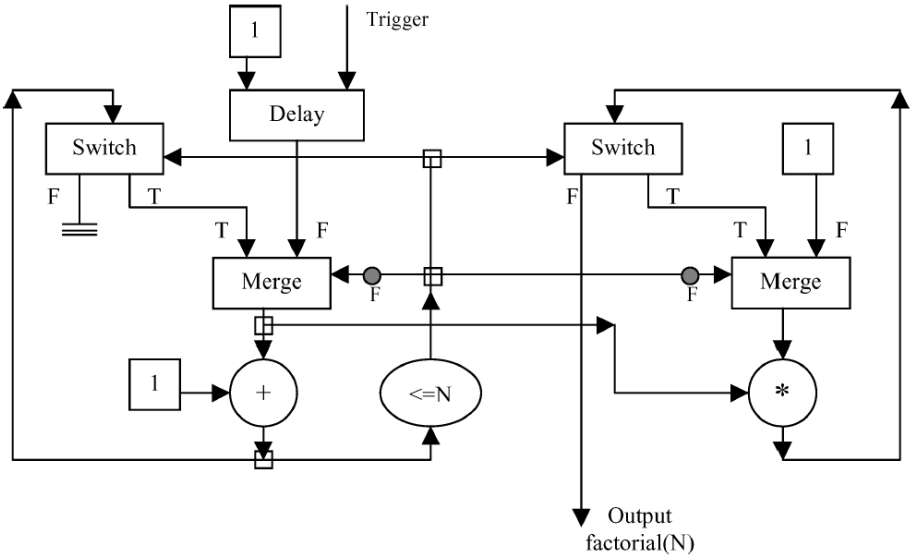
   Operácia spracúva len dáta, ktoré dostane na vstupy. Toto je dosiahnuté zakázaním globálnej pamäte, čím zároveň nevzniká miesto preťaženia (bottleneck) napr. zbernica pri komunikácii s pamäťou.

4. Žiadne vedľajšie účinky.

   Všetky operácie sa správajú funkcionálne. Kvôli zákazu modifikácií premenných, operácie nevytvárajú inde v programe vedľajšie efekty a je zaručené, že daná hodnota-premenná ostane rovnaká všade kde sa použije.

5. Chýbajúca citlivosť operácií na minulosť.

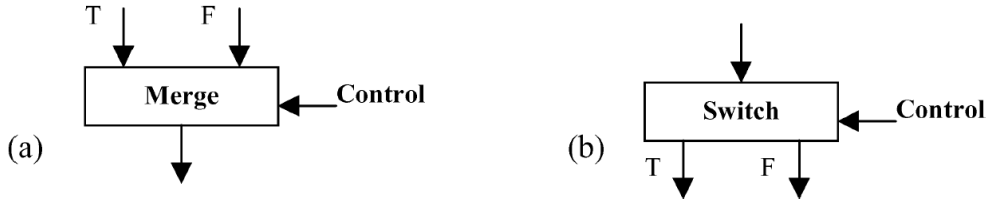
   Operácie neobsahujú žiadne stavové premenné, ktoré by umožnili uchovať údaje medzi jednotlivými spusteniami (volaniami) operácie.




Obr. 1.6: Grafový zápis výpočtu faktoriálu v modeli dátových tokov.[JHM04]

### 1.4.2 Riadenie dát

Poradie v akom sa nezávislé operácie vykonajú môže byť nedeterministické, aj keď výsledok vzhľadom na vstup deterministický je. Kvôli zachovaniu determinovateľnosti výsledku výpočtu je zakázané svojvoľné zlučovanie dát. Ak by to bolo umožnené, dáta by mohli doraziť na koncový vrchol v nesprávnom poradí a poškodiť výsledok výpočtu. Na obr. 1.5 (b) vidíme, že sú dáta premennej **Y** duplikované pri ich odosielaní operáciám. Je zrejmé, že ak sú dáta duplikované, prípadne rozdelené, tak ich bude potrebné zlúčiť.[Vee86, JHM04, Kul18, MK11] Okrem potenciónálneho zlúčenia dát operáciami (napr. hodnoty sú sčítané), poskytuje model dátových tokov na tento účel špeciálne operácie nazvané brány, zobrazené na obr. 1.7.




Obr. 1.7: Riadiace brány. a) zlučovacia, b) prepínacia. [JHM04]

Zlučovacia brána obr. 1.7 (a) funguje podobne ako podmienka. Obsahuje 3 vstupy, 1 výstup. Riadiaci vstup spracúva hodnotu boolovskej premennej pravda/nepravda a vstupy **A**, **B** spracúvajú dáta. Ak brána na riadiacom vstupe načíta hodnotu pravda, na výstup pošle dáta zo vstupu **A**, inak pošle dáta zo vstupu **B**.

Prepínacia brána obr. 1.7 (b) funguje na podobnom princípe ako zlučovacia brána. Obsahuje 2 vstupy, riadiaci a dátový; a 2 dátové výstupy **A** a **B**. Podľa hodnoty na riadiacom vstupe presmeruje dáta zo vstupu na výstup **A** (pravda) alebo **B**.

Kombinácia brán umožňuje podmienené alebo opakujúce sa vykonávanie operácií. Zoskupením 3 prepínacích brán je možné implementovať podmienené vykonávanie. Kombinácia oboch typov brán umožňuje implementáciu opakovaných výpočtov nad operáciami. [JHM04]

### 1.4.3 Riadenie výpočtu

Úvodné návrhy dátových tokov uvažovali o dátach ako o pasívnych prvkoch výpočtu, ktoré sa nachádzajú na hranách modelu do momentu, kým nie sú načítané operáciami. Rýchlo sa ale ukázalo, že je efektívne použiť dáta na kontrolu výpočtu. V teoretickej rovine boli popísané 2 spôsoby implementácie výpočtu riadeného dátami.[Vee86, DK82, Ack82, JHM04]

#### 1.4.3.1 Dostupnosťou riadený výpočet

Výpočet riadený dostupnosťou dát je prvým spôsobom implementácie. Operácia je aktívna len vtedy, ak sú pre všetky vstupy operácie dostupné všetky dáta. Pri aktivácii dôjde k absorbovaniu dát na vstupoch a spusteniu výpočtu. V prípade chýbajúcich dát na jednom, prípadne viacerých vstupoch, je operácia neaktívna. [JHM04]

#### 1.4.3.2 Dopytom riadený výpočet

Druhým spôsobom je požiadavkami riadený výpočet. Na spustenie výpočtu musí operácia dostať požiadavku o dodanie dát na výstup. Ak takúto požiadavku dostane, výpočet prebehne v momente ako budú dostupné dáta na všetkých vstupoch operácie. Ak operácia nedostane požiadavku je neaktívna, bez ohľadu na dostupnosť dát. Napriek vyšším požiadavkám na množstvo komunikácie plynúcich z doručovania požiadaviek na dodanie dát, je výhodou, že tento spôsob umožňuje odstrániť niektoré typy operácií.[JHM04]

### 1.4.4 Architektúry modelu dátových tokov

Praktická implementácia modelu dátových tokov sa ukázala ako náročná úloha. Teoretický model dátových tokov vytvára predpoklady, ktoré nie sú realizovateľné v skutočnom svete.

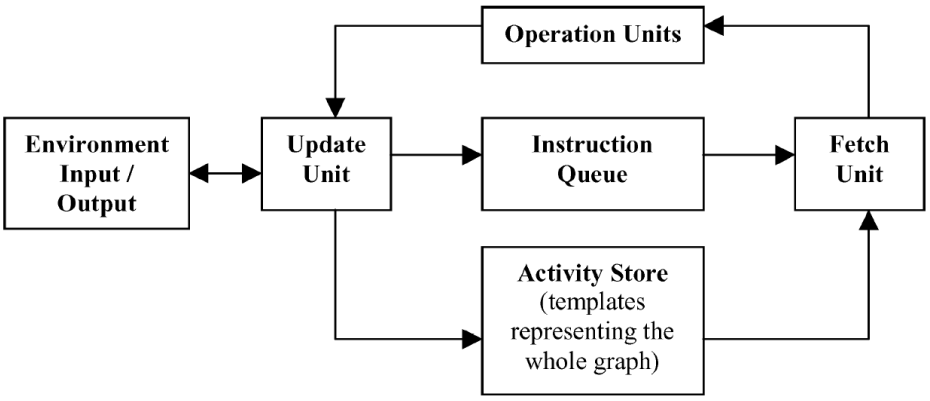
Teória predpokladá neobmedzene veľkú pamäť. Zároveň predpokladá, že existuje neobmedzený počet operácií, ktoré môžu byť vykonávané paralelne. V skutočnosti je pamäť aj počet operácií obmedzený, limitovaný možnosťami hardvéru. Tieto obmedzenia viedli k vzniku hardvérových implementácií, ktoré neimplementujú model dátových tokov úplne

presne. V skutočnosti tieto zmeny v implementácii môžu spôsobiť zablokovanie výpočtu (deadlock), v situáciách v ktorých teoretický model funguje spoľahlivo. [Vee86, Ack82, JHM04]

#### 1.4.4.1 Statická architektúra

Statická architektúra nahrádza hrany v grafe teoretického modelu jednoduchším variantom, kde každá hrana môže ukladať najviac jednu kópiu dát. Teoretický model reprezentuje hranu pomocou neobmedzenej pamäte reprezentovanej pomocou FIFO zásobníka. Výpočet prebehne v momente, keď má každý vstup potrebné dáta a výstupy operácie sú voľné. Pri implementácii boli do grafu pridané hrany idúce v opačnom smere dátových hrán, kvôli prenosu potvrdzujúcich tokenov.

Výhoda statickej architektúry spočíva v rýchlej a jednoduchej detekcii spustiteľnosti výpočtu. Zároveň vďaka tomu, že každá hrana drží práve 0 alebo 1 kópiu dát, môže byť pamäť alokovaná počas kompilácie. Tieto vlastnosti zároveň zjednodušili tvorbu hardvéru, pretože nebolo nutné implementovať zásobníkové pamäte. Pridanie hrán prenášajúcich potvrdzovací token, ale spôsobilo spomalenie výpočtu kvôli zvýšenej cirkulácii dát a čiastočne obmedzilo možnosti paralelizácie. [JHM04]




Obr. 1.8: Statická architektúra [JHM04]

#### 1.4.4.2 Dynamická architektúra

Dynamická architektúra implementuje paralelizmus viacnásobným volaním (používaním) podgrafu. V skutočnosti je v pamäti uložená iba jedna kópia podgrafu a dáta obsahujú značku (tag) na rozlíšenie ich príslušnosti k jednotlivým volaniam. Namiesto pravidla jednej kópie dát na hrane grafu ako je tomu v statickej architektúre, dynamická reprezentuje hranu ako pamäť obsahujúcu ľubovoľné množstvo balíčkov dát, každý s vlastným tagom. Takto označené dáta umožnili ich spracovanie v inom poradí ako boli na hranu zapísané, pretože ich bolo možné jednoznačne rozlíšiť.

Implementácia dynamickej architektúry je komplexnejšia, má vyššie nároky na pamäť a operácie zabezpečujúce vytváranie a párovanie tagov. Zároveň ale umožňuje maximálnu podporu simultánnej paralelizácie, spracúvanie dát v rôznom poradí pri vyššom výkone ako statická architektúra. [JHM04]

## 1.5 Hardvérová implementácia

Model dátových tokov je z hľadiska hardvérovej implantácie v silnom kontraste s von Neumanovou architektúrou. Programy vytvorené v modeli dátových tokov je možné spúšťať na procesoroch založených na architektúre von Neumann. Dochádza ale k výkonovým stratám vyplývajúcim z obmedzených možností paralelizácie a odlišnej práce s pamäťou. To viedlo k vzniku nových architektonických hardvérových platforiem, podporujúcich model dátových tokov. [Smi98]

Rozdeľujeme ich na dve hlavné časti:

1. Fixné

   Po navrhnutí a výrobe nie možné zmeniť konfiguráciu (funkcionalitu) týchto čipov. Zástupcami sú napr.: Mikrokontroléry, DSP (Digitálny signálny procesor) alebo ASIC (Integrovaný obvod pre špecifickú aplikáciu).

2. Programovateľné

   Užívateľom programovateľné logické obvody umožňujú zmenu ich konfigurácie, po tom ako boli vyrobené. V závislosti od komplexnosti konfigurácie a zamerania, poznáme rôzne programovateľné logické obvody. PLD (programovateľný logický obvod) sú vhodnejšie pre klopné kombinatorické obvody – PAL (Jednorazovo programovateľné logické pole), GAL (všeobecné logické pole). CPLD (zložitý programovateľný logický obvod) je evolúciou PLD, umožňujúci vytvárať rozsiahlejšie kombinatorické obvody. FPGA (programovateľné hradlové pole) je oproti PLD vhodnejšie na rozsiahlu sekvenčnú logiku.

### 1.5.1 FPGA

FPGA (field programable gate array) programovateľné hradlové pole je typ polovodičového zariadenia, založeného na matici konfigurovateľných logických blokov (CLB) prepojených prostredníctvom programovateľných prepojení (routing matrix). [PH13, HH12, Dig] FPGA sa skladá z:

1. Routing Matrix Smerovacia matica prepája CLB a LUT bloky naprieč celým FPGA.

2. CLB Complex Logic Block (komplexný logický blok) – každý obsahuje 2 SLICES.

3. SLICES Plátky sú vytvorené zoskupením LUT a Flip-Flop.

4. LUT Look-Up Tables (vyhľadávacie - náhľadové tabuľky) definujú naprogramovanie logiky, môžu byť pripojené k flip-flopom.

5. Flip-Flop - Flip-Flop alebo LATCH je základná stavebná jednotka, má dva stabilné stavy. Môže sa použiť na ukladanie informácií ako pamäť alebo na synchronizáciu propagácie signálu - prepojenie LUT s hodinovým signálom daného FPGA. 1 flip-flop má kapacitu 1 bit. Flip-flopy sú označované aj ako distribuovaná pamäť (ďalej označovaná ako DRAM) prípadne registre, kvôli obmedzenej kapacite a vysokej rýchlosti sú vhodnejšie na ukladanie menšieho množstva dát ako sú medzi-výsledky a pod.

6. Block RAM - ďalej ako BRAM alebo aj vstavaná (embedded) pamäť RAM, je štandardne diskrétnou súčasťou FPGA. Vhodná na konštrukciu väčších dátových úložísk, zásobníkov, vyrovnávacích pamätí, registrov.

Takt (clock) - periodický štvorcový signál, pracujúci na logických úrovniach 0 a 1 počas určitých časových intervalov, je ďalšou dôležitou súčasťou mnohých digitálnych systémov. Zabezpečuje synchrónnu prácu operácií. Logické operácie sa vykonávajú pri nábežnej hrane (zmena signálu, prechod z logickej 0 na 1) alebo klesajúcej hrane signálu. Použitie taktu zabezpečí synchronizáciu operácií a perióda určí rýchlosť vykonávania.[PH13, HH12, Dig]

FPGA a HDL jazyky umožňujú tvorbu obvodov oboch typov, odporúčané je však používať synchronizované operácie. Latch je príkladom nesynchronizovaného - netaktovaného (non-clocked) obvodu slúžiaceho na uloženie dvoch stabilných stavov, logickej 0 a 1. Pridaním taktu, synchronizáciu k obvodu latch, získame flip-flop.

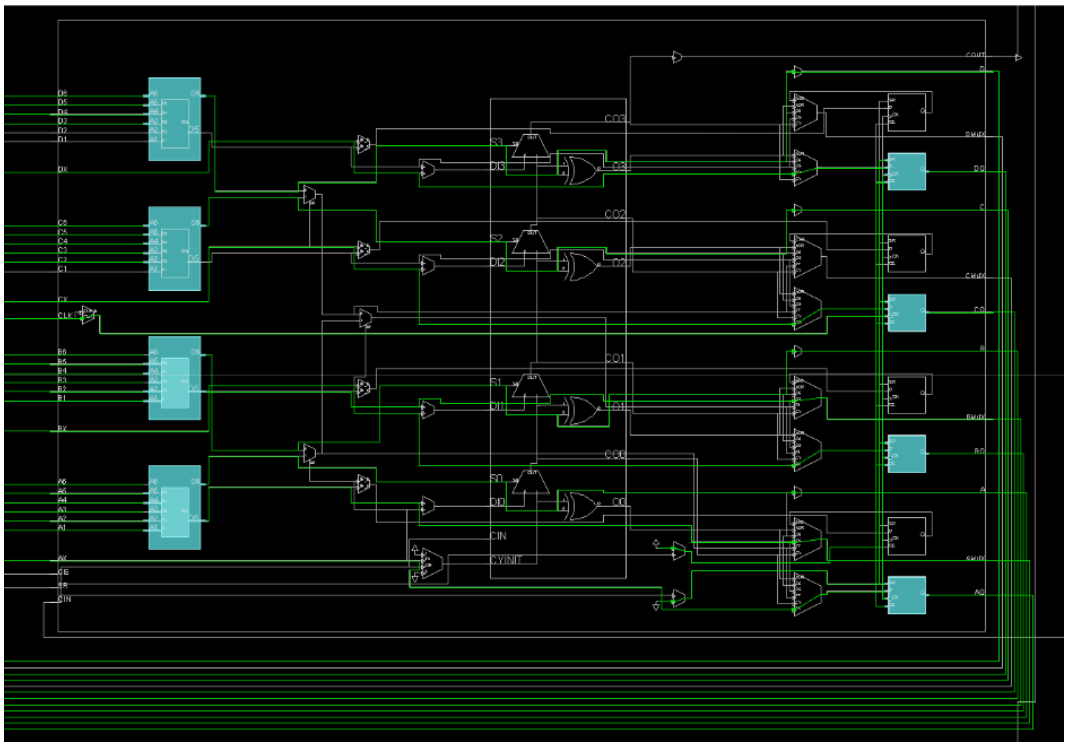
FPGA môže byť naprogramované podľa konkrétnych požiadaviek aplikácie alebo funkcie, ľubovoľne veľa krát. Táto vlastnosť odlišuje FPGA od integrovaných obvodov špecifických pre aplikáciu (ASIC), ktoré sú vyrábané na mieru pre konkrétne výpočtové úlohy, a po ich vytvorení už nie je možné funkcionalitu ASICu zmeniť.[PH13, HH12, Dig]

Medzi lídrov v odbore patrí Altera (od r. 2015 súčasť Intelu) a Xilinx (vlastnený AMD od 02/2022). Vďaka svojmu programovateľnému charakteru sú FPGA ideálne pre mnoho rôznych použití. Najviac používané boli v počiatku v telekomunikáciách a postupne prenikajú do oblasti priemyselných, vedeckých a spotrebiteľských aplikácií. Využívané sú predovšetkým v oblastiach: spracovania audia a videa, káblovej komunikácie, výpočtových simuláciách, návrhoch počítačových čipov (ASIC, CPU, GPU), medicínskych prístrojoch.

Pri mikrokontroléroch, procesoroch má programátor kontrolu nad softvérom. FPGA umožňuje vytvoriť si vlastné digitálne obvody. Programátor

má kontrolu nad hardvérom.

FPGA neobsahuje žiadny procesor na spustenie softvéru, kým programátor dané FPGA nenastaví. Konfigurácia FPGA môže byť rozličná, od niečoho jednoduchého (logická brána), až po zložité (viacjadrový procesor). Pri vytváraní vlastného dizajnu, sa používa HDL (Hardware Description Language).



Obr. 1.9: Zobrazenie zapojenia obvodov v FPGA v softvéri Vivado

### 1.5.2 HDL

Dva najpopulárnejšie HDL (Hardware Description Language, jazyk popisujúci hardvér) sú Verilog a VHDL. Platforma FPGA široko podporuje tieto jazyky, vrátane používaných vývojových prostredí a kompilátorov. Kód, algoritmus, program napísaný v jazykoch HDL, popisuje hardvér,

mení zapojenie jednotlivých obvodov.

#### 1.5.2.1 VHDL

VHDL je jazyk popisujúci hardvér, používaný pri tvorbe a automatizácii elektronického návrhu na opis digitálnych systémov a systémov so zmiešaným signálom, ako sú programovateľné hradlové polia a integrované obvody. Implementuje abstrakcie na úrovni prenosu registrov (RTL) a je silne typovaný. VHDL môže byť tiež použitý ako univerzálny paralelný programovací jazyk. [UT17, Smi98]

#### 1.5.2.2 Verilog, SystemVerilog

Verilog je podobne ako VHDL jazyk popisujúci hardvér. Syntax je podobná jazyku **C**. Verilog je oproti VHDL slabšie typovaný, všetky typy sú preddefinované a každý má bitovú reprezentáciu. Podporuje polia-/vektory (arrays, vectors), nepodporuje zložené dátové štruktúry. [oEE16, Sut01, Pal03]

SystemVerilog vznikol z jazyka Superlog, pričom verifikačné funkcie sú založené na jazyku OpenVera. Ako norma IEEE bol prijatý v roku 2005. V roku 2009 bola norma rozšírená o normu Verilog. Všetky funkcie Verilogu sú dostupné v SystemVerilogu, Verilog je teda podmnožinou SystemVerilogu. [SM14, Pal03]

Verilog umožňuje 3 typy modelovania hardvéru: [LaM19, TM08]

1. Štrukturálne modelovanie Každý prvok použitý pri modelovaní by mal byť definovaný ako štruktúra. Logické hradlá, ktoré sú používané vo veľkom rozsahu, boli definované vopred. Táto metóda sa preto nazýva aj modelovanie na úrovni hradiel.

2. Modelovanie dátovým tokom Táto metóda sa nazýva aj funkcionálne modelovanie. Kľúčová konštrukcia je nazvaná assign (priradiť). Výstupom musí byť skalárna veličina alebo vektor. Vstupom je funkcia, tvorená napríklad logickými hradlami. Takéto modelovanie umožňuje kompaktnejší zápis.

3. Behaviorálne modelovanie Behaviorálne modelovanie) Používajú sa podmienkové, prípadne rekurzívne príkazy, ktoré sú aktivované zmenou sledovaného signálu (signálov, hodnôt). Na začiatku v čase 0 môžeme vykonať inicializáciu (príkaz initial). Zmeny signálu sleduje príkaz always. Na poradí príkazov, ktoré budú vykonané nezáleží. Behaviorálne modelovanie pozná 2 typy priradení. Blokujúce a neblokujúce. Blokujúce priradenie používa operátor -". Vykonanie ostatných priradení v poradí je blokované, kým aktuálne priradenie nie je dokončené. Vykonanie teda prebieha postupne, sekvenčne. Operátor «=öznačuje neblokujúce priradenie. Príkazy s neblokujúcim operátorom sa vykonajú súbežne.

Jednotlivé typy využívajú niektorý variant premenných, pripadne oba: [LaM19, TM08]

- Net (wire) - slúži iba na transport informácie medzi časťami návrhu
- Variable (reg, register) - slúži ako dočasné úložisko údajov

Verilog pozná 4 hodnoty, ktoré signál môže nadobudnúť v 3 stavoch: [LaM19, TM08]

- deklarovaná a definovaná hodnota, signál 0 (logická 0 alebo nepravdivá podmienka) a 1 (logická 1, pravdivá podmienka)
- deklarovaná a nedefinovaná alebo neznáma hodnota, signál X

- nedeklarovaná a nedefinovaná alebo nezapojená, signál Z (vysoká impedancia, trojstavový alebo plávajúci signál)

S hodnotami 0, 1, X je možné pracovať v FPGA počas výpočtu, sú plne syntetizovateľné, signál Z je dostupný tiež ale s obmedzeniami.

Výrobcovia FPGA a ich partneri vytvárajú do HDL jazykov rozšírenia vo forme hardvérových blokov (obvodov) - knižníc duševného vlastníctva IP (intellectual property). IP bloky sa prispôsobujú použitému FPGA a konfigurujú podľa užívateľských potrieb. Ponúkajú špecifické bloky na spracovanie signálov z meracích prístrojov, audia, videa, základné bloky ako sú registre, akumulátory, zbernice až celé procesorové celky napr. procesoru RISC-V. [LaM19, TM08]

## 1.6 Existujúce riešenia

### 1.6.1 Programovacie jazyky dátových tokov

V začiatkoch sa dátové toky zaznamenávali pomocou grafov. Uprostred 70. rokov začali vznikať programovacie jazyky dátových tokov. Na začiatku sa jednalo predovšetkým o textové jazyky ako TDFL, Lapse, Hasal, LAU, Lucid, neskôr Sisal, SAC a mnohé ďalšie. Zároveň vznikali rozšírenia štandardných jazykov o dátové toky, ktoré boli v danom období populárne ako Fortran, Pascal a **C**. [WP94]

Zápis v textovej forme ťažkopádnejšie vyjadroval dátové závislosti, ktoré sú v grafovej reprezentácii ihneď zjavné.

To viedlo k vzniku vizuálnych programovacích jazykov, ktoré vychádzali z pôvodných grafových zápisov modelu dátových tokov. Medzi ne patria napr. DDL, FDL, GPL, ProGraph, LabVIEW.

### 1.6.2 Vizuálne programovacie jazyky dátových tokov

Vizuálnych programovacích jazykov dátových tokov vzniklo veľké množstvo. Veľká časť z nich je špecificky zameraná na konkrétnu úlohu ako spracovanie obrazu, vedecké vizualizácie, operácie nad databázovými systémami, analýzu a spracovanie veľkého množstva dát, kryptografiu, smerovanie sietí, tvorbu hudby, matematické výpočty, riadenie strojov výrobných liniek a mnohé iné. [WP94]

Vizuálne programovacie jazyky so všeobecným zameraním nezaznamenali veľké a dlhotrvajúce úspechy. Za ich menším rozšírením stojí predovšetkým slabá až neexistujúca podpora paralelizácie v existujúcich populárnych počítačových systémoch a ich procesoroch v minulých desaťročiach. Posun smerom k paralelným systémom nastal koncom 90. rokov a trvá dodnes.

Najväčšie úspechy zaznamenali jazyky ProGraph a LabVIEW, ktoré sme spolu s jazykom DataWolf analyzovali a získané poznatky aplikovali pri návrhu vyššieho jazyka dátových tokov v tejto práci.

Dátové toky umožňujú čitateľnejší a presnejší zápis paralelizácie narozdiel od klasických programovacích jazykov. Funkcionálny a deklaratívny zápis v klasických jazykoch detaily paralelizácie skrýva, bez ich využitia je programovanie a ladenie ťažšie a zdĺhavé.

#### 1.6.2.1 Úroveň živosti

Vizuálne programovacie jazyky sa snažia poskytnúť používateľovi určitú mieru abstrakcie a zároveň ponúknuť nástroje, ktoré budú vhodne reprezentovať program a manipuláciu so spracúvanými dátami.

Vizuálny zápis si kladie za jeden z cieľov poskytovať programátorovi možnosť prirodzenejšieho, atraktívnejšieho a rýchlejšieho pochope-

nia programu oproti textovému zápisu. Tieto ciele boli postupne aplikované aj na spracúvané dáta. Snaha o poskytovanie spätnej väzby programátorovi počas programovania viedla k vzniku predpokladu živosti (liveness assumption) programu, objektov, metód a dát v pracovnom priestore editora. [Tan13]

Ďalším z cieľov je zmenšenie oneskorenia medzi akciou programátora a zobrazením jej vplyvu na beh programu. Zjednodušenie rozhodovania sa pri prideľovaní priority úlohám, ktoré vyvolali zmenu pri behu, napr. chybu. **A** podpora výučby programovania pomocou skorej vizualizácie vykonaných zmien programu.

Podľa stupňa živosti (level of liveness) spätnej odozvy programátorovi boli vo vizuálnych jazykoch definované 4 úrovne, neskôr doplnené o ďalšie 2.

6 úrovní živosti programu[Tan13]:

1. 1. Informatívna

   Na tejto úrovni je vizuálny program vnímaný ako pomocný vývojový diagram.

2. 2. Spustiteľná (informatívna)

   Spustiteľný vývojový diagram. Zapísaný program je možné spúšťať. Po vykonaní úprav je nutné program opätovne spustiť. (spĺňajú prakticky všetky známe programovacie jazyky)

3. 3. Responzívna (spustiteľná a informatívna)

   Úpravami riadené vykonávanie. Program alebo jeho časť je vykonaná automaticky zakaždým ako programátor pozmení niektorú časť a tá spĺňa podmienky vykonateľnosti.

4. 4. Aktívna (responzívna, spustiteľná a informatívna)

   Úpravami a dátami riadené vykonávanie. Program sa neustále vykonáva, reaguje na zmeny v programe podľa vstupov od programátora a v dátach.

5. 5. Takticky prediktívna

   Pridávanie operácií/príkazov výberom z predpovedaných variant programu. Počítač predpovedá nasledujúce kroky programátora na základe predošlých aktivít, za použitia strojového učenia. Z predpovedaných blízkych verzií programu je možné jednu alebo viaceré spúšťať. Programátor si vyberie, ktoré moduly/operácie budú použité prípadne ručne doupravené. (dopĺňanie kódu v aktuálne programovanom riadku)

6. 6. Strategicky prediktívna

   Pridávanie rozsiahlej funkcionality výberom a úpravami z predpovedaných variant. Vývojové prostredie by odhadovalo zámer programátora a ponúkalo rozsiahlejšiu funkcionalitu na úrovni celých funkcií/modulov v súlade so zámerom programátora a programovanou časťou.

### 1.6.3 ProGraph

Prograph [CGP89] je vizuálny multi-paradigmový objektovo-orientovaný programovací jazyk dátových tokov, s vykonávaním riadeným dostupnosťou dát. Najväčší úspech dosahoval na prelome 80. a 90. rokov. Bol dostupný na operačných systémoch spoločnosti Apple, neskôr Microsoft a

fungoval na procesoroch architektúry riadiacich tokov, čo značne obmedzovalo možnú paralelizáciu.

Operácie nad dátami sú reprezentované pomocou symbolov a ikon, prepojených pomocou orientovaných hrán. Špecifická synchronizačná hrana umožňovala vytváranie závislosti medzi dátovo nezávislými vrcholmi, a zabezpečovala oneskorenie vo vykonávaní jedného z prepojených vrcholov voči druhému. Prograph obsahoval interpreter aj kompilátor. Vývojové prostredie vďaka tomu umožňovalo vykonávanie programu počas jeho upravovania a ladenia, čo umožňovalo opravu chýb bez neustálej potreby opätovnej kompilácie. Pri ladení podporoval zároveň obvyklé mechanizmy ako je prerušenie vykonávania (breakpoint) a krokovanie. Počas ladenia boli práve vykonávané operácie zvýrazňované a pomocou kurzora mohol programátor zobrazovať dáta na jednotlivých hranách. Prograph implementoval 1. až 4. úroveň živosti programu.

Napriek opravám a vylepšeniam v novších verziách nedostatky Prographu neboli úplne eliminované. Komentáre metód pomocou štítkov a dokumentácia bola nedostatočná a nevhodne realizovaná. Zabudované metódy a objekty nemali dostatočne popísané typy dát pre vstupy a funkcionalitu. Smerovanie zapojenia a komentovanie vstupno-výstupných portov vyžadovalo pozornosť vývojárov, aby vizuálny kód zostal prehľadný a nevznikal zo zapojení špagetový kód. Programátor mohol elementy ručne popresúvať, ale v zásade neexistoval spôsob ako sa tomu vyhnúť.

Veľké množstvo okien bol ďalší problém. Pri editácii kódu a otváraní vnútra vytvorených metód alebo objektov (podgrafov) sa detaily obsahu otvárali v nových oknách a často dochádzalo k prekrývaniu pôvodne upravovanej metódy.

### 1.6.4 LabVIEW

LabVIEW (Laboratory Virtual Instrument Engineering Workbench) [JdlC19] je platforma a vývojové prostredie s vizuálnym programovacím jazykom G založeným na výpočtovom modeli dátových tokov. Poskytuje širokú variabilitu použitia, vrátane vedeckých výpočtov, zberu údajov, ovládanie prístrojov a priemyselnú automatizáciu. Pracuje pod väčšinou existujúcich operačných systémov a vybrané moduly je možné kompilovať do platformy FPGA. Model vykonávania je riadený dostupnosťou údajov. Funkcie a operácie jazyka sú reprezentované pomocou symbolov a ikon, prepojených pomocou hrán.

Štandardne používané nástroje na vytváranie, sledovanie a porovnávanie verzií kódu nie je možné použiť ako v prípade textových jazykov, nakoľko je jazyk G vizuálny. Prostredie Labview obsahuje nástroje, ktoré túto funkcionalitu dopĺňajú.

### 1.6.5 DataWolf

DataWolf [Kul18] je nižší vizuálny programovací jazyk založený na dátových tokoch, s grafickým prostredím a kompilátorom do jazyku Lucid (dialekt Verilogu) spoločnosti Embedded Micro, ktorý je následne použitý na konfiguráciu FPGA. Obsahuje sadu nízko-úrovňových operácií a modulov s jasnými definíciami a jednoznačnou textovou reprezentáciou používanou pri kompilácii.

Praktickým spôsobom pomocou lomených prepojení a duplikačného uzla je vyriešený problém špagetizácie prepojení v kóde. Datawolf vizuálne, tvarovo i farebne slabo rozlišuje jednotlivé operácie a elementy jazyka. V dnešných jazykoch a editoroch je štandardne použitá farebná

téma. Jednotlivé kľúčové slová (príkazy, volania, podmienky, cykly, komentáre a ostatné elementy) textového jazyka majú priradenú konkrétnu farbu, ktorá zlepšuje čitateľnosť, prehľadnosť a zrýchľuje orientáciu v rozsiahlom kóde.

## 1.7 Použité technológie, hardvér

Na komunikáciu s FPGA cez UART sme používali softvér Realterm.

Kompilátor pre vyšší jazyk dátových tokov je navrhnutý v jazyku Javascript. Navrhnutý vyšší jazyk bude prekladaný do Verilogu.

### 1.7.1 Vivado

Vivado design suite je softvérový balík vyrábaný spoločnosťou Xilinx na syntézu a analýzu návrhov v HDL. Spoločnosť Xilinx je zároveň dodávateľom použitého FPGA Arty. Podporované sú jazyky VHDL a Verilog, SystemVerilog nie je plnohodnotne podporovaný.

Distribuovanú pamäť používa Verilog štandardne pri vytvorení premennej. Block RAM je možné vytvoriť pomocou IP blokov v prostredí Vivado alebo pomocou špeciálnej šablóny prispôsobenej pre použitý model FPGA. Hardvérová implementácia Block RAM sa môže líšiť naprieč rôznymi modelmi FPGA, preto je nutné zabezpečiť v oboch prípadoch jej konfiguráciu prostredníctvom správnych parametrov. [Xilc, Xila, Xilb]

Verilog poslúži pri experimentoch a ako výstupný jazyk nášho kompilátora. Tento výstup bude ďalej spracovávaný v prostredí Vivado.

Navrhnutý dizajn FPGA vo Verilogu je možné ladiť pomocou simulátora. Vivado počas ladenia, simulácie podporuje použite prerušení (breakpoint) a krokovania algoritmu - fungovania obvodov.

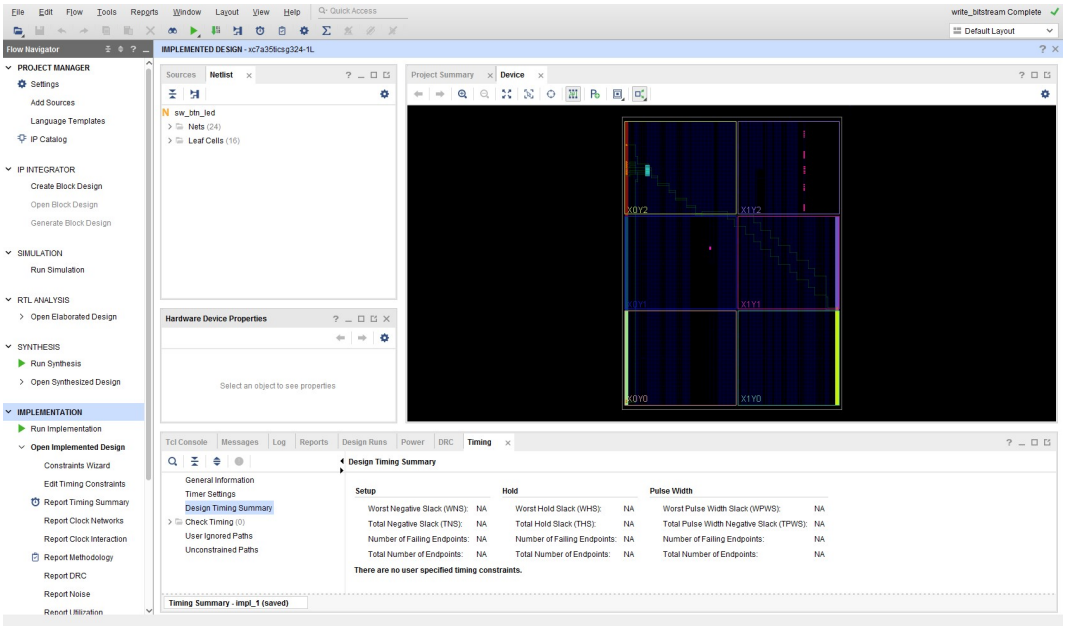
Po overení je možné daný dizajn kompilovať do FPGA:

1. Syntetizácia vytvorí z HDL popisu hardvéru obvody s definovanou funkcionalitou a vykoná sériu optimalizácií.
2. Implementácia vytvorený logický návrh preusporiada a naviaže (namapuje) do existujúceho hardveru - FPGA.
3. Bitstream konvertuje implementáciu do konfiguračného súboru.
4. Nastavenie FPGA prebieha pomocou bitstream súboru 2 spôsobmi.

   Prostredníctvom prostredia Vivado a pod.

   Konfiguračnou flash pamäťou - nakonfiguruje FPGA pri získaní napájania.

Všetky základné funkcie Vivado je možné ovládať prostredníctvom Tcl konzoly [Xilc], bez použitia GUI prostredníctvom skriptov. Pri konfigurácii je možné používať rozsiahle knižnice IP spoločnosti Xilinx, spolu so šablónami pre konfiguráciu špecifických obvodov daného FPGA.




Obr. 1.10: Vivado design suite

### 1.7.2 FPGA ARTY

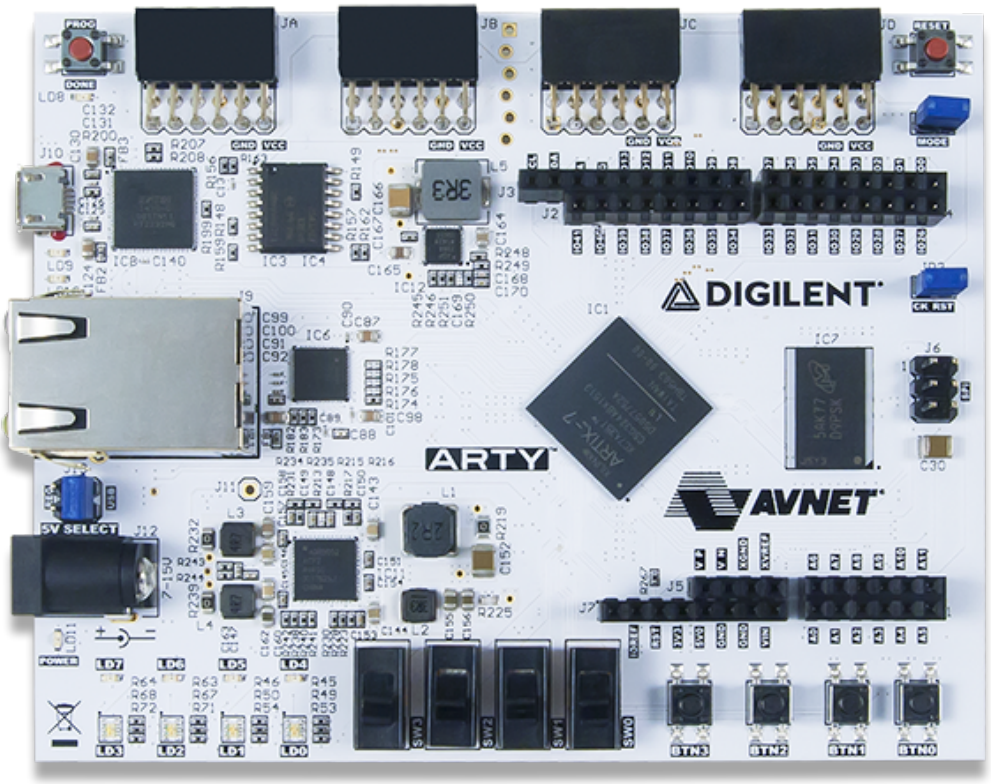
FPGA Arty [Dig] je vývojová doska obsahujúca FPGA čip Artix-7, ktorú používame v tejto práci. Použitý model FPGA je Arty A7-35T Xilinx XC7A35TICSG324-1L. Tento čip obsahuje štandardne sa vyskytujúce hardvérové elementy v FPGA. Technické parametre:

1. 5 200 Slices obsahujúcich štyri 6-vstupové LUTy a 8 flip-flops
2. 1 800 Kbits rýchlej block RAM pamäte
3. 90 DSP slices
4. Interný hodinový signál (clock speed) do rýchlosti 450MHz (štandardne nastavený na 100MHz)
5. Analógovo digitálny prevodník (XADC)

6. 256MB DDR3L (externej) pamäte, 16-bit zbernica @ 667MHz
7. 16MB Quad-SPI Flash pamäte (umožňuje ukladanie konfigurácie FPGA)

Pamäť na použitej vývojovej doske ARTY sa delí na 2 skupiny, 3 rôzne druhy:

1. Externá pamäť (ERAM) - mimo čipu FPGA - DDR3L o kapacite 256MB so 16-bit zbernicou s frekvenciou 667MHz
2. Interná pamäť - umiestnená v čipe FPGA:
    - Block RAM (BRAM) - celková kapacita 1800 Kbit (225 KB) FPGA Arty obsahuje 50 kusov Block RAM 1 ks Block RAM obsahuje 36 Kbit alebo 2 * 18 Kbit pamäte o šírke 64 bitov Zdržanie pri čítaní: 2 cykly hodinového signálu.
    - Distribuovaná pamäť (DRAM) - celková kapacita 40,625 Kbit 1 flip-flop ukladá 1 bit. 5200 (slices) * 8 (flip-flops) = 41 600 bitov = 5200 bajtov (5,2 KB). Zdržanie pri čítaní: 1 cyklus hodinového signálu.




Obr. 1.11: FPGA Arty

# Kapitola 2

# Návrh

V tejto kapitole uvádzame návrh, analýzu a experimenty s navrhnutými konštrukciami jazyka Draken-I. Počas vývoja sme niektoré konštrukcie viackrát menili. Navrhované konštrukcie a komponenty jazyka sme počas navrhovania testovali programovaním na papieri a zároveň sme skúmali možnosti implementácie daných komponentov vo Verilogu.

Vo svete hardvéru a jeho navrhovania je jedným z najdôležitejších prvkov signál a jeho zmena. Celé FPGA je riadené taktom (clock), hodinovým signálom resp. jeho zmenou. Signál sa opakuje - FPGA teda stále cyklí. Clock loop (d'alej len CL, 1CL) - jeden cyklus/takt, počas ktorého celé FPGA obdrží signál a vykoná určitú činnosť. Zmeny taktu alebo zmeny vlastných signálov (dát, premenných) umožňujú riadiť výpočet spôsobom zhodujúcim sa s filozofiou dátových tokov.

Počas návrhu sme brali ohľad na niekoľko vlastností, ktoré sme chceli v našom jazyku dosiahnuť. Informácie v tejto kapitole sú radené podľa možnosti chronologicky a organizované do logických celkov jednotlivých oblastí výskumu. Kľúčové slová a komponenty jazyka sú v anglickom jazyku.

Vizualizácie, ktoré v kapitole uvádzame, nemusia byť vždy syntetizovateľné do FPGA, nakoľko sa jedná o pracovné verzie, rôzne upravované v priebehu vývoja návrhu a experimentov.

Dáta s ktorými vo Verilogu pracujeme majú veľkosti udávané v bitoch. DW - data width - dátová šírka označuje veľkosť dát. Napr.: 1 bit, bajt (šírka 8 bitov), integer (šírka 32 bitov). DH - data depth - dátovú hĺbku, určuje koľko prvkov (napr. o šírke 8 bitov, teda bajtov ) uložíme. V tomto prípade sa jedná o pole (pole bajtov)

## 2.1 Komponenty

Komponenty jazyka, kľúčové oblasti, ktoré sme skúmali:

- hrany
- pamäte
- aritmetické vyjadrenia
- buffre
- podmienky
- cyklickosť
- multiplexovateľnosť
- duplikovanosť

## 2.2 Vlastnosti

Vlastnosti a parametre jazyka Draken-I, ktoré sme sa snažili dodržať pri návrhu:

- čitateľnosť jazyka
- prehľadnosť jazyka
- funkcionálnosť
- deklaratívnosť
- rýchlosť - snažili sme sa dosiahnuť maximálnu mieru paralelizácie a efektívnosti
- optimalizovanosť - snaha o optimalizovanú spotrebu hardvérových prostriedkov
- kompilovateľnosť - v maximálnej miere hľadať a využívať komponenty a prednosti Verilogu pri kompilácií z Draken-Ijazyka

V neposlednom rade sme chceli vytvoriť jazyk, ktorý by bol praktický, páčil sa nám a chceli by sme ho používať v praxi.

## 2.3 Syntax

Na začiatku vývoja sme spísali základné vlastnosti a konštrukcie, ktoré by každý programovací jazyk mal mať. Následne sme ich rozvíjali a pridávali nové. Na to aby sme mohli s jazykom pracovať potrebujeme splniť niekoľko základných bodov - vytvoriť spôsob reprezentácie:

- hodnôt - dát s ktorými pracujeme

- pamätí - dáta potrebujeme skladovať a distribuovať
- operácií - základných - logických, aritmetických, relačných, porovnávacích a rozšírených - bitových, redukčných a ďalších

### 2.3.1 Hodnoty

Hodnoty, dáta sú nevyhnutnou častou každého programovacieho jazyka. Verilog podporuje slabé typovanie s nutnosťou uviesť typ číselnej sústavy, a ich veľkosť v bitoch - dátovú šírku (data width).

#### 2.3.1.1 Číselné hodnoty

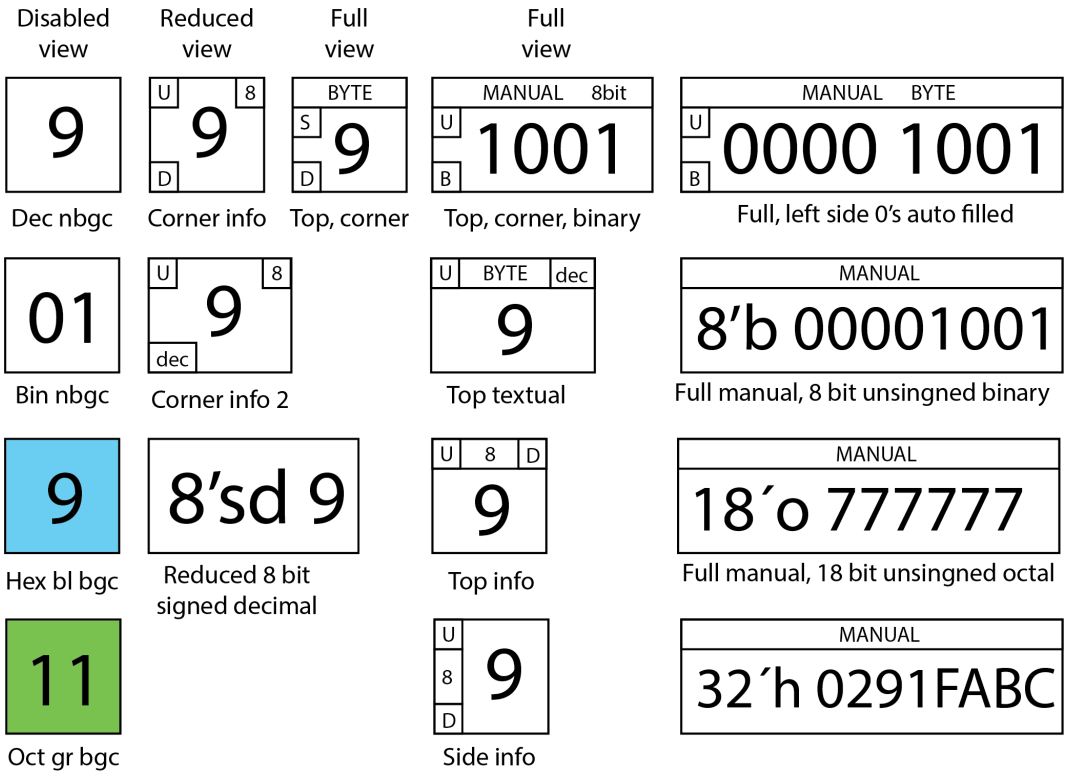
Verilogom podporované sú desiatková, binárna, šestnástková a osmičková sústava. Zadávanie v týchto sústavách umožňuje aj náš jazyk. Pokiaľ nie je uvedený typ sústavy, je automaticky používaná desiatková sústava a bez-znamienkový (unsigned) vstup. Veľkosť dát je možné zadávať manuálne alebo výberom z definovaných typov vrátane znamienkového (signed) a bez-znamienkového vstupu:

- MANUAL bit width N signed/unsigned
- BIT bit width 1
- NIBBLE bit width 4 signed/unsigned
- BYTE bit width 8 signed/unsigned
- SHORT bit width 16 signed/unsigned
- INT bit width 32 2´s complement
- LONG bit width 64 signed/unsigned

- FLOAT bit width 64 floating point representation

Pri vizuálnej forme sme experimentovali s rôznymi úrovňami detailov zobrazenia vid'. obr 2.1:

- deaktivovaný výpis detailov (disabled)
- redukovaný (reduced)
- plnohodnotný (full)




Obr. 2.1: Varianty zápisu číselných hodnôt. u,U-unsigned, s,S-signed, d,D-decimal, b,B-binary, h,H-hexadecimal, o,O-octal

Pri deaktivovanom detaile výpisu je rozlišovanie zabezpečené pomocou farebného pozadia. Dec ngbc - decimálny zápis, bez farebného pozadia. Bin ngbc - binárny zápis, bez farebného pozadia. Hex bl bgc - hexadecimálny, modré pozadie. Oct gr bgc - osmičkový zápis, zelené pozadie.

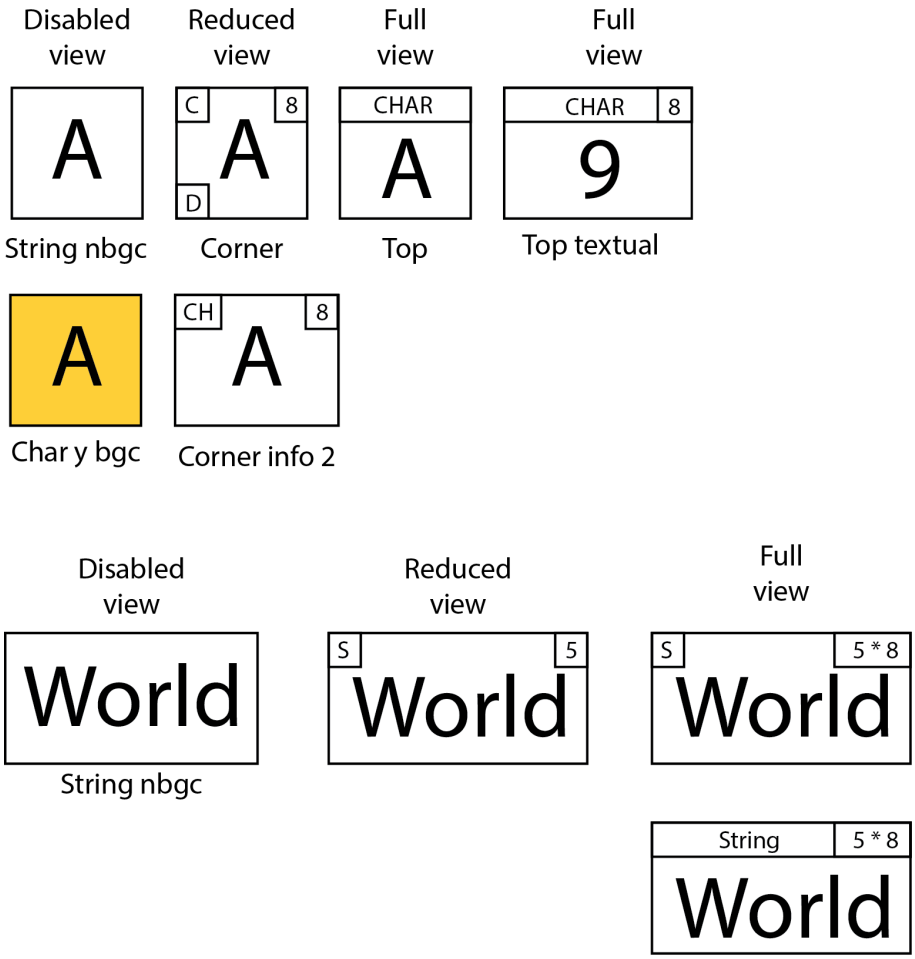
Zápis decimálnych čísiel je možný bez zadávania veľkosti a číselnej sústavy, v takomto prípade je automaticky použitá decimálna sústava, znamienkový vstup veľkosti 32 bitov.

Pri testoch a experimentoch jazyka najčastejšie uvádzame hodnoty v deaktivovanom detaile, desiatkovej sústavy. Manuálne zadávanie veľkosti dát je v našom jazyku odporúčané. Hardvérové zdroje v FPGA sú limitované a pri väčších návrhoch (aplikáciách) je nutné optimalizovať spotrebu zdrojov, vrátane pamätí. Spôsob zadávania veľkostí dát, premenných (width, depth) sme sa rozhodli ponechať zhodnú s Verilogom. Charakteristika práce s hardvérom, kompilácie vyššieho jazyka do Verilogu a použitie FPGA môže vyžadovať dodatočné úpravy kódu, návrhu vo Verilogu. Identický zápis premenných zabezpečí užívateľom jednoduchší prechod medzi jazykmi v prípade potreby.

#### 2.3.1.2 Textové hodnoty

Znaky a textové reťazce sú vo Verilogu reprezentované pomocou ASCII tabuľky. Na uloženie jedného znaku (charakteru, char) je potrebných 8 bitov miesta. Textový reťazec je uložený ako pole definovanej šírky (width, 8 bitov na 1 znak) a hĺbky dát (data depth) - podľa počtu znakov v reťazci. Možné reprezentácie sú na obr. 2.2.

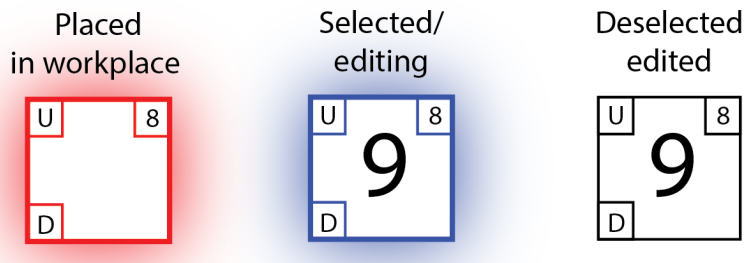
Pre rozlíšenie reťazca a znaku pri deaktivovanom detaile výpisu, má znak žlté pozadie.




Obr. 2.2: Varianty zápisu znakov a reťazcov.

### 2.3.2 Úpravy

Pri návrhu sme sa zameriavali aj na vizualizáciu elementov pri práci s nimi v pracovnom prostredí editora. Umiestnené a nedefinované elementy sú zobrazené červeným podsvietením. Vybrané alebo práve upravované elementy sú podsvietené modrou. Správne deklarované a definované komponenty majú podsvietenie neaktívne.




Obr. 2.3: Ukážka úpravy číselného komponentu v pracovnom priestore.

### 2.3.3 Pamäť

Pamäť je možné realizovať v FPGA viacerými spôsobmi 1.5.1. Použité FPGA umožňuje použitie 1.7.2 3 rôznych pamätí. Pri práci s dátovými tokmi sa zameriavame na 2 hlavné typy pamätí:

- Hrany (pipes, connections)
- Premenné, hodnoty (registers, values, variables)

#### 2.3.3.1 Hrany

Hrany v dátových tokoch slúžia na distribúciu dát medzi operáciami, prípadne ich dočasné uloženie. Navrhujeme 3 typy hrán.

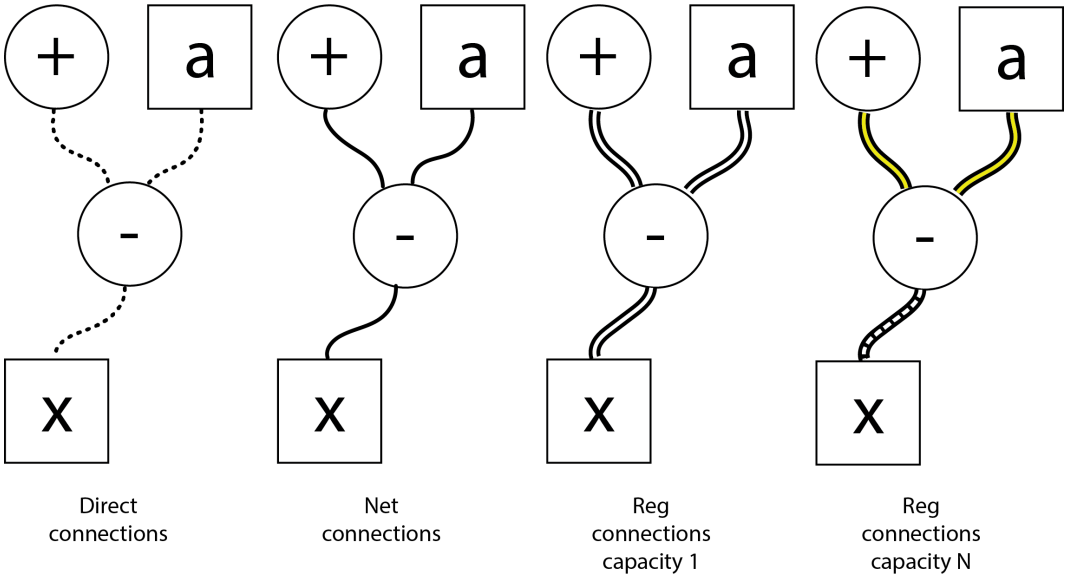
**Priame** Direct connection - priame prepojenie. Operácie sú prepojené priamo, výstup operácie **A** je priamo zapojený na niektorý zo vstupov operácie **B**. Hrana takéhoto zapojenia je reprezentovaná bodkovanou čiarou, viď. obr. 2.4 Direct connections.

**Sieťové** Net connection, wire connection - sieťové prepojenie, reprezentuje dáta, kým nie sú zmenené. Neukladajú hodnotu, majú definovanú šírku dát (DW) aby dáta dokázali preniesť. Zmeny dát na vstupe tohto zapojenia sú ihneď spropagované na výstup (v rámci toho istého taktu) - vo Verilogu implementovateľné prostredníctvom kontinuálneho priradenia (continuous assigment). Tieto hrany sú vizualizované jednoduchou spojitou čiarou, viď. obr. 2.4 Net connections.

**Kapacitné** Reg, register connection - prepojenie pomocou registra, ktoré má definovanú kapacitu úložného priestoru. Register si pamätá hodnotu medzi priradeniami, je ju možné zapísať a neskôr prečítať. Šírka DW je nastavená podľa dát, hĺbka DH je štandarne 1. Pri väčšej hĺbke ako 1 sa hrana chová ako zásobník FIFO (first in - first out) aby bolo zachované poradie postupnosti dát.

Hrany s kapacitou 1 tokenu dát sú zobrazené dvojitou čiarou, obr. 2.4 Reg connections capacity 1.

Pri hranách s kapacitou N tokenov dát, sme experimentovali s viacerými zobrazeniami. Vstupy operácie mínus sú zobrazené dvojitou čiarou so žltým podfarbením obr. 2.4 Reg connections capacity N, vstupy operácie odčítanie. Vhodnejšou verziou zápisu je ale z hľadiska lepšej viditeľnosti použitie hrany s okienkami, viď. obr. 2.4 Reg connections capacity N, výstup operácie odčítanie.




Obr. 2.4: Hrany. Priame zapojenie, sieťové zapojenie, kapacitné (s veľkosťou 1), kapacitné (veľkosť N).

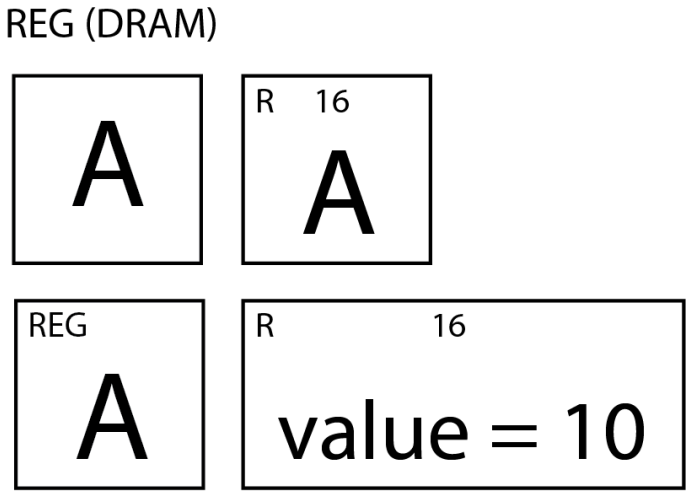
#### 2.3.3.2 Premenné

FPGA štandardne obsahujú 2 typy pamätí 1.5.1. Čip, ktorý obsahuje použitá vývojová doska podporuje oba tieto typy pamätí 1.7.2 a navyše aj externú RAM pamäť. Navrhované pamäte delíme podľa hĺbky na premenné a polia. Oba typy je možné implementovať vo Verilogu prostredníctvom distribuovanej alebo blokovej pamäte. Skúmali sme rôzne vizuálne zápisy, implementácie, ako aj možnosť grafického vyjadrenia inicializácie.

Pamäť je štandardne inicializovaná v plnej veľkosti na hodnoty X definovanej šírky. Je možné zadať tiež inicializáciu na hodnoty 0, prípadne konkrétne inicializačné dáta.

**Premenné** Premenné majú hĺbku 1, môžu teda uložiť iba jeden prvok, o určenej šírke. Implementovateľné sú pomocou DRAM, BRAM aj ERAM. Predvolene používame distribuovanú pamäť (DRAM) vzhľadom na jej vý-

kon, pomenovanú ako REG (register). Alternatívne je možné použiť aj blokovú pamäť (BRAM) pomenovanú BREG. Externú pamäť v tomto prípade nepoužívame. Dáta potrebujeme zapísať a prečítať z premennej, nepotrebujeme adresu na indexovanie poľa a podobne, môžeme však indexovať konkrétny bit v premennej, prípadne vykonať rez.




Obr. 2.5: Premenné, register, rôzne formy zápisu.

**Polia** Polia je možné implementovať pomocou všetkých typov pamätí, ktoré máme k dispozícií. Štandardne o všetkých typoch pamätí uvažujeme ako o RAM - umožňuje čítanie aj zápis dát. ROM - umožňuje iba čítanie a je pri návrhu považovaná iba za podmnožinu variantov RAM. Táto charakteristika platí aj v hardvéri, kde pri vytvorení ROM, sú prvky obvodov vytvárajúce RAM deaktivované.

**Distribuovaná pamäť** Distribuovaná pamäť - DRAM môže byť vytvorená 3 spôsobmi [Xilb] - pomocou príkazu **reg** vo Verilogu, jazykovej šablóny vygenerovanej výrobcom použitého FPGA alebo pomocou IP generátora prostredia Vivado.

Dátová hĺbka (DH) DRAM je vo všetkých FPGA (spoločnosti Xilinx) možná v rozsahu 16 – 65536 blokov (dátových slov) v násobkoch 16 pri konfigurácii pomocou IP generátora. Dátová šírka (DW) každého bloku sa môže pohybovať v rozsahu 1 – 1024 bitov. Pri vytváraní DRAM Verilogom hĺbka začína na veľkosti 1 (variant, ktorý je registrom - premennou) a jej zväčšovanie je možné po 1 kroku hĺbky - nie je obmedzené na násobky 16.

Pri použití IP generátora je v závislosti na konfigurácii použitej pamäti možné dosiahnuť oneskorenie 0 alebo 1 takt.

Z hľadiska počtu portov rozdeľujeme distribuovanú pamäť na 3 kategórie:

- Single port - pamäť poskytuje na komunikáciu 1 port, ktorým je možné vykonávať čítanie alebo zápis dát. Vytvorenie prostredníctvom Verilogu, šablón alebo IP.
- Dual port - 2 porty na paralelné čítanie a 1 port slúžiaci na zápis. Vygenerovanie možné pomocou šablón alebo IP.
- Multi port - 4 porty, 3 na paralelné čítanie a 1 kombinovaný port umožňujúci čítanie alebo zápis. Generovanie možné pomocou šablón.

Obr. 2.6 zobrazuje v hornej časti možné zápisy distribuovanej pamäte DRAM spolu s označeniami šírky, hĺbky a počtu portov (S-single/D-dual port).

**Bloková pamäť** Bloková pamäť (BRAM) - môže byť vytvorená 2 spôsobmi, šablónami alebo IP generátorom ktorý poskytuje viac než bohaté možnosti konfigurácie BRAM [Xila]. BRAM sú rozdeľované na single-

port RAM, simple dual-port RAM, true dual-port RAM, single-port ROM, dual-port ROM.

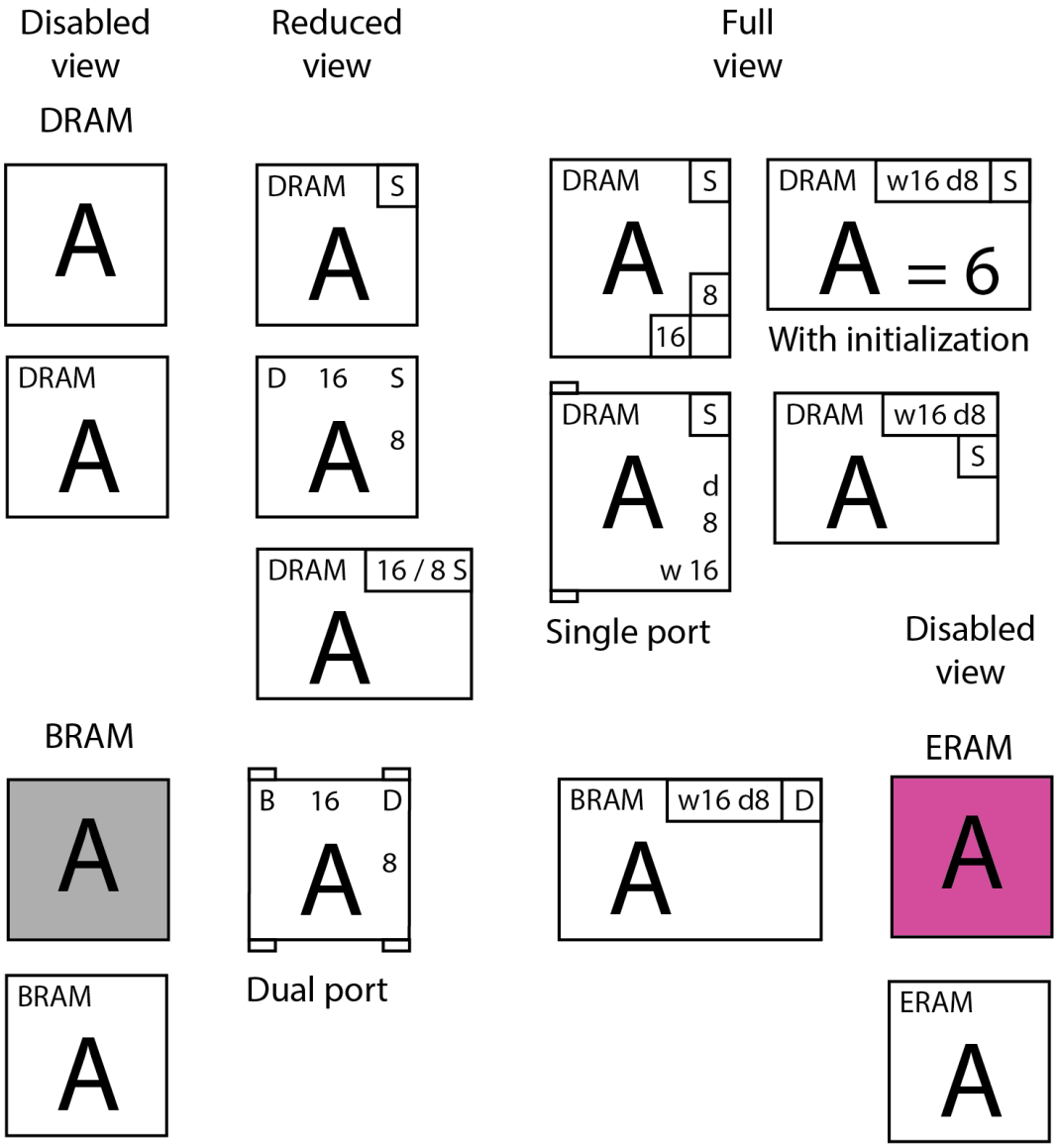
Podľa použitej BRAM je šírka a hĺbka dát veľmi rozmanitá, obzvlášť v prípade konfigurácie IP generátorom. Možné rozsahy uvádzame nižšie pri 2 konkrétnych variantoch.

Pri použití IP generátora je v závislosti na konfigurácii použitej DRAM je možné dosiahnuť rôzne oneskorenie pri práci s pamäťou, priemerne 2 takty (CL).

Z hľadiska počtu portov rozdeľujeme distribuovanú pamäť na 2 kategórie:

- Single port RAM - pamäť poskytuje na komunikáciu 1 port, ktorým je možné vykonávať čítanie alebo zápis dát. Konfigurácia prostredníctvom IP. DW BRAM je v rozsahu 1 - 4608 bitov. DH BRAM je v rozsahu 2 - 1 048 576 blokov (slov).

- Dual port (true dual port RAM) - pamäť poskytuje na komunikáciu 2 porty, možné paralelné čítanie, paralelný zápis dát na oboch, zápis na prvom a čítanie na druhom porte a naopak. Správanie pamäte je možné konfigurovať v prípadoch kolízie (zhodných) adries pri zápise na oboch portoch, prípadne zápis a čítanie zhodnej adresy.




Obr. 2.6: Distribuovaná, bloková, externá pamäť.

### 2.3.4 Operácie

Pri návrhu operácií sme vychádzali z grafického zápisu dátových tokov, ktoré sme videli v použitej literatúre a existujúcich riešeniach. Pri viacerých z týchto jazykov sme nadobudli pocit slabšieho vizuálneho rozlíše-

nia jednotlivých komponentov a konštrukcií, ktoré tieto jazyky umožňujú. Pri návrhu sme si definovali galériu základných geometrických tvarov z ktorých sme vychádzali s cieľom lepšej vizuálnej čitateľnosti jazyka, pomocou vizuálneho odlíšenia jednotlivých komponentov. Pri návrhu komponentov sme zároveň vychádzali z existujúcich vo Verilogu.

#### 2.3.4.1 Aritmetické

Väčšina operácií v našom jazyku má iba jeden výstup (výstupný port), aritmetické nie sú výnimkou.

**Unárne** Operácie s jedným vstupom. Zvolený tvarom je kosoštvorec nakoľko skvele odpovedá tvaru s 1 vstupom a 1 výstupom, zobrazené na obr. 2.7 Unary. Aritmetický mínus, konvertuje kladnú hodnotu na zápornú a naopak. Increment a decrement operácie zvačšujú a zmenšujú hodnotu o 1.

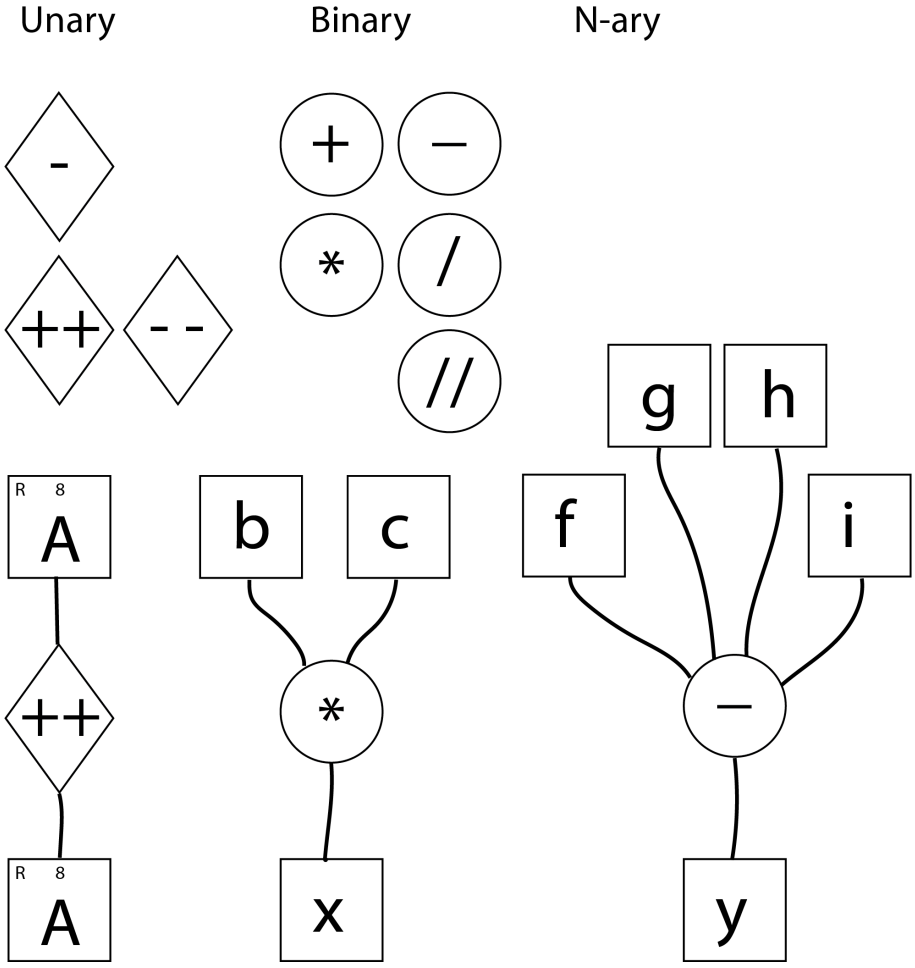
**Binárne** 2 vstupy, 1 výstupný port. V prípade binárnych aritmetických operácií sme zvolili kruh, ktorý nám pripadá najvhodnejší, viď. obr. 2.7 Binary. Zároveň zhodujeme s väčšinou existujúcich jazykov dátových tokov.

Sčítanie, odčítanie, násobenie, delenie, celočíselné delenie sú operácie, ktoré zavádzame do nášho jazyka a sú zároveň nutným základom, nedeliteľnou súčasťou väčšiny všetkých jazykov bez ohľadu na paradigmu alebo výpočtový model.

V prípade nekomutatívnych operácií (odčítanie, delenie, celočíselné delenie) záleží na poradí zapojenia hrán a portov. Poradie zapojenia portov zľava doprava je zhodné s matematickým zápisom. Matematický zá-

pis operácie násobenia z obr. 2.7 Binary je $x = b * c$.

**N-árne** N-árne aritmetické operácie N vstupov, kde N > 2. Jedná sa o rozšírenie binárnych o viacero vstupov, z dôvodu zrýchlenia zápisu a zlepšenia čitateľnosti jazyka. Nekomutatívne N-árne operácie majú zhodné pravidlá zapájania ako binárne. Zápis operácie odčítanie na obr. 2.7 N-ary je $y = f - g - h - i$.




Obr. 2.7: Unárne, binárne a N-árne aritmetické operácie a ich možné zapojenia.

#### 2.3.4.2 Logické operácie

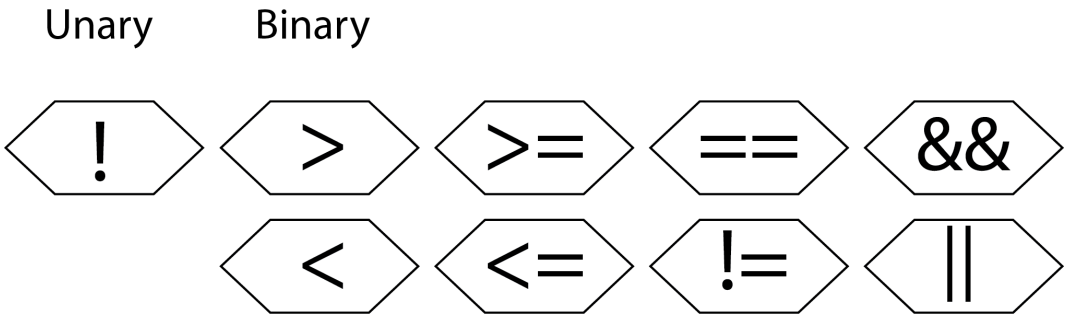
Logické operácie spolu s operátormi rovnosti a relacií boli pôvodne súčasťou komponentu IF a neboli zapájané priamo 2.3.5.1. Počas experimentovania s jazykom sa postupne transformovali do samostatného komponentu. Rozdeľujeme ich podľa počtu hodnôt, ktoré spracúvajú.

**Unárne** Logický zápor NOT symbol - "!". Zmení vstupnú pravdivostnú hodnotu na opačnú.

**Binárne** Logické a porovnávacie operátory, používame rovnaké aké pozná Verilog, vrátane väčšiny ostatných jazykov:

- vačší $>$
- menší $<$
- vačší rovný $>=$
- menší rovný $<=$
- rovný $==$
- nerovný $!=$
- súčasne $\&\&$
- alebo $||$

Vizuálna reprezentácia 6 uholníkom vznikla počas vývoja komponentu IF, pôvodne opticky prepájajúcim 2 porovnávané hodnoty vid'. obr 2.14.




Obr. 2.8: Logické a porovnávacie operátory.

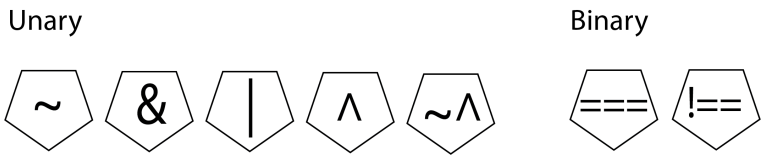
#### 2.3.4.3 Bitwise a Identity

Podmnožina logických operácií, s prácou nad jednotlivými bitmi. Reprezentované 5 uholníkmi.

**Unárne**

- negácia $\sim$
- and $\&$
- or $|$
- xor $\wedge$
- xnor $\sim \wedge$

**Binárne** Bitové porovnanie totožnosti vstupov.

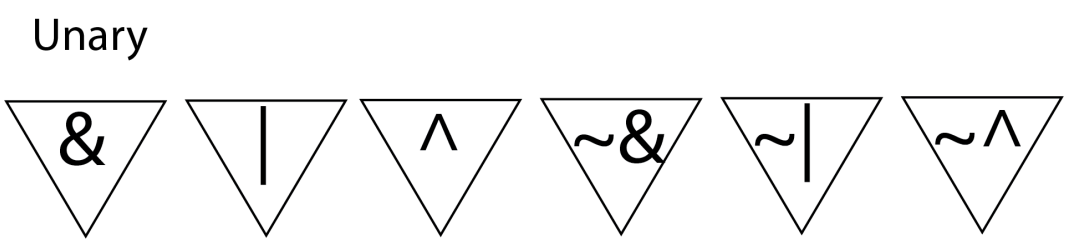



Obr. 2.9: Bitwise operátory.

#### 2.3.4.4 Redukčné

1 vstup, 1 výstup. Redukujú vstup na výstup o veľkosti 1 bit podľa logickej operácie.

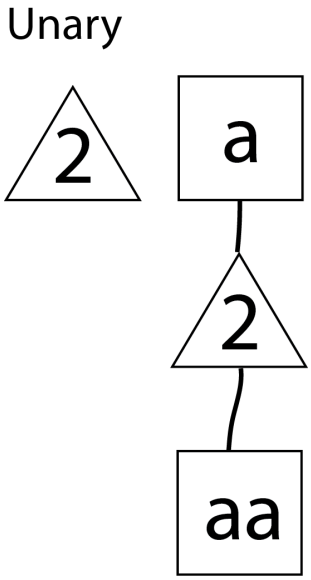
- and &
- or |
- xor ^
- nand ~ &
- nor ~ |
- xnor ~ ^




Obr. 2.10: Redukčné operátory.

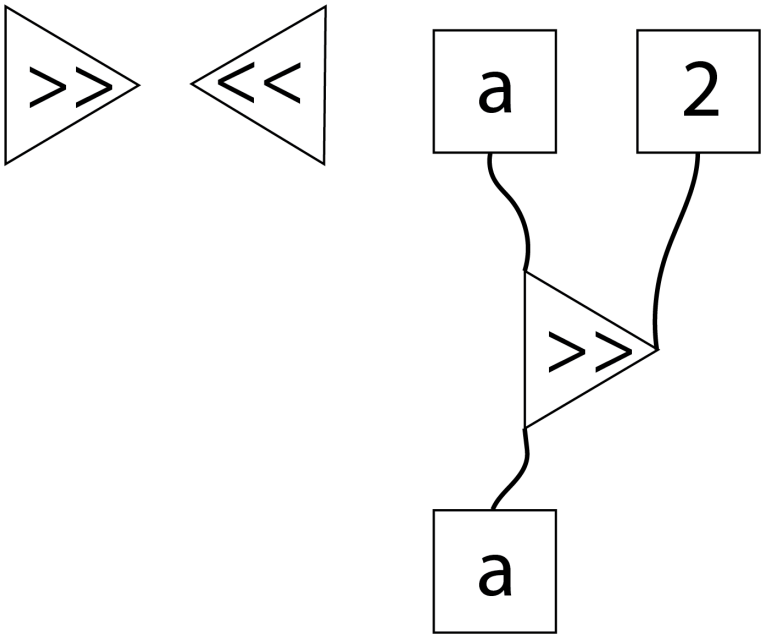
#### 2.3.4.5 Replikačné

Duplikujú dáta N krát, kde N >= 2. Výsledkom je zreťazený vstup N krát.




Obr. 2.11: Replikačné operátory.

#### 2.3.4.6 Shiftovacie



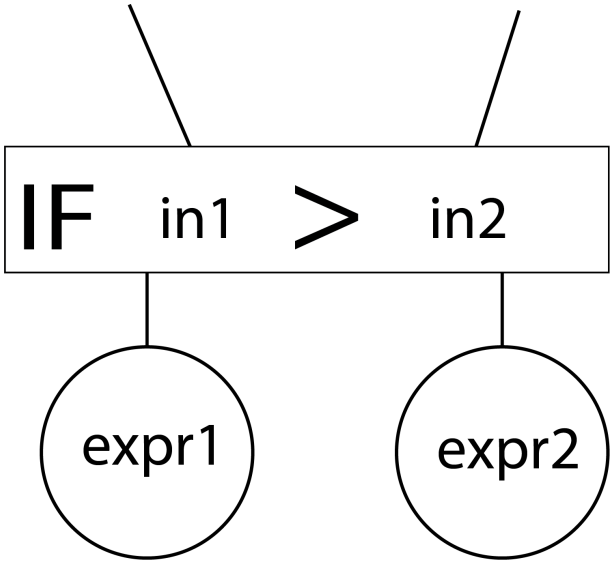

Obr. 2.12: Shiftovacie oprátory a možné zapojenie.

### 2.3.5 Podmienky

Operácia umožňujúca podmienku prešla najväčším vývojom.

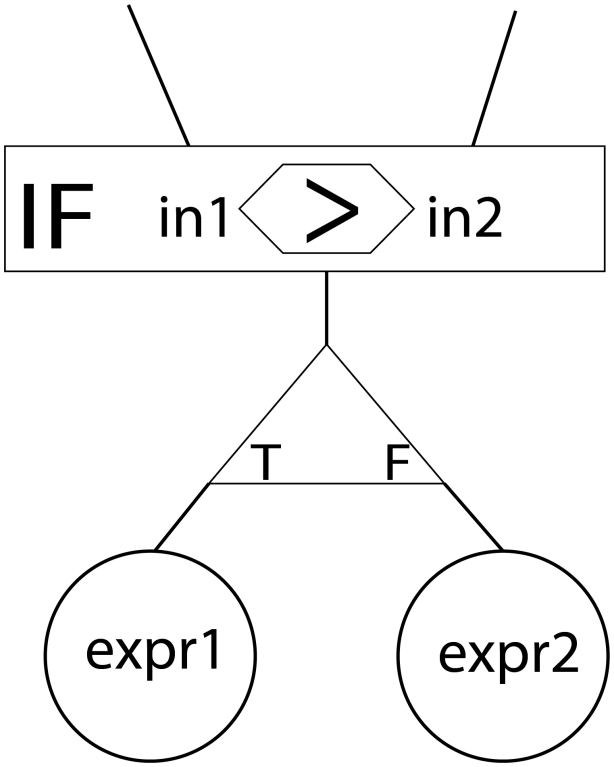
#### 2.3.5.1 IF

Na začiatku sme začali s variantom 1, a jeho skôr logickým nákresom obr. 2.13.




Obr. 2.13: Podmienka v.1, logický nákres.

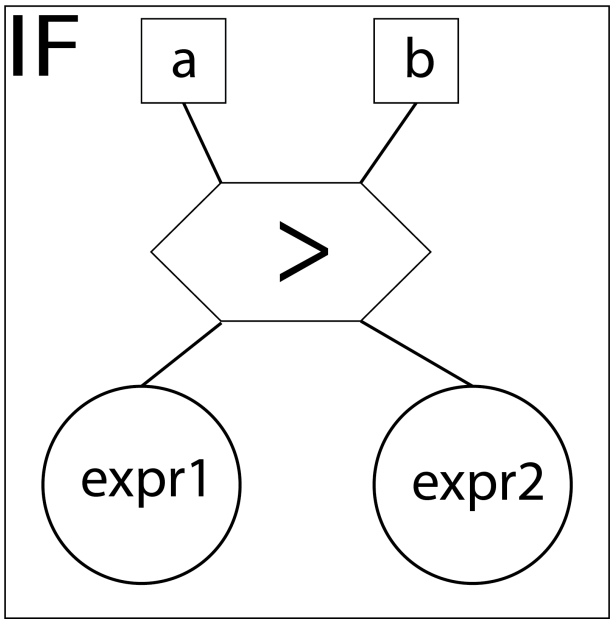
Pomerne rýchlo sme sa dostali k variantu 2, ktorý vracal logickú hodnotu, následne slúžiacu na aktiváciu výrazov (expressions).




Obr. 2.14: Podmienka v.2, logický nákres.

Oba spôsoby nám pripomínali až príliš riadiace toky a zápis podmienok v nich, čo sme nechceli.

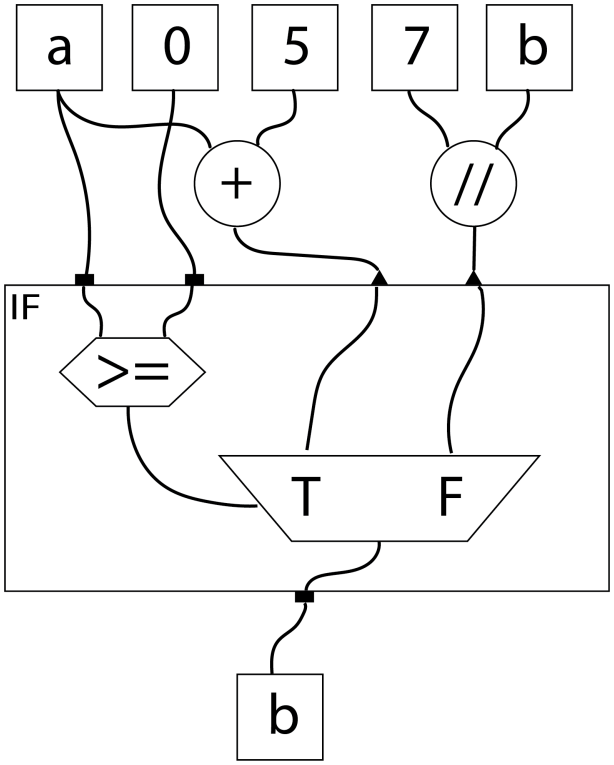
Z verzie 2 sme teda odstránili všetko a ponechali iba 6 uholník a hrany začali zapájať priamo naň, čím sa podmienkový zápis IF začal transformovať na volič selector.




Obr. 2.15: Podmienka v.3, logický nákres.

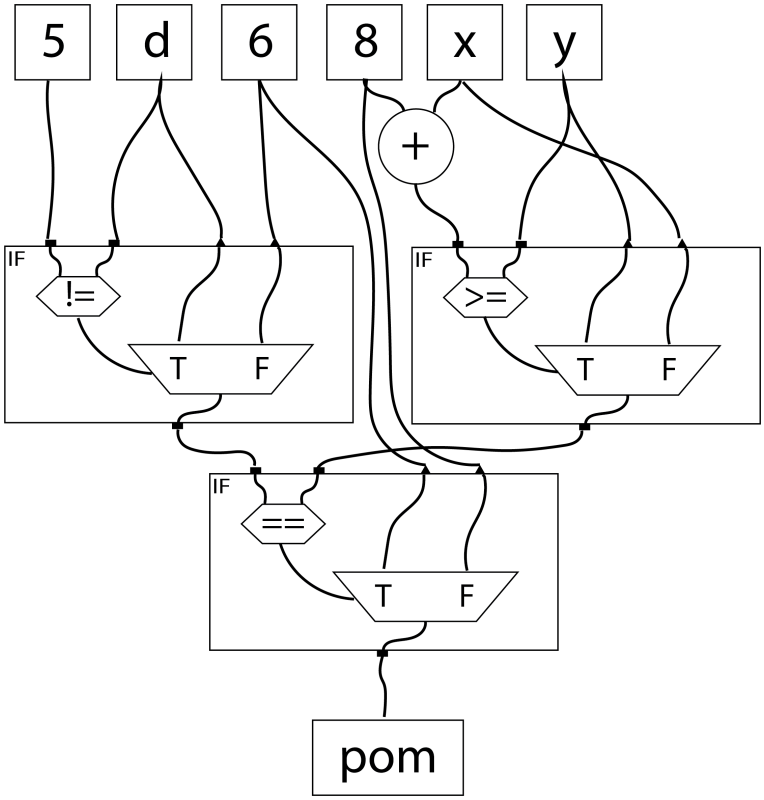
#### 2.3.5.2 IF Selector

Rozpracovali sme selector, ten vracia zakaždým pravdivostnú hodnotu a pridali zlučovaciu bránu 1.7 z teórie dátových tokov (merge gate), nazvanú mergor. Označenie IF sme stále ponechali, zapojenie a celý zápis je na obr. 2.16.




Obr. 2.16: IF Selector, v.1, spolu so zapojením.

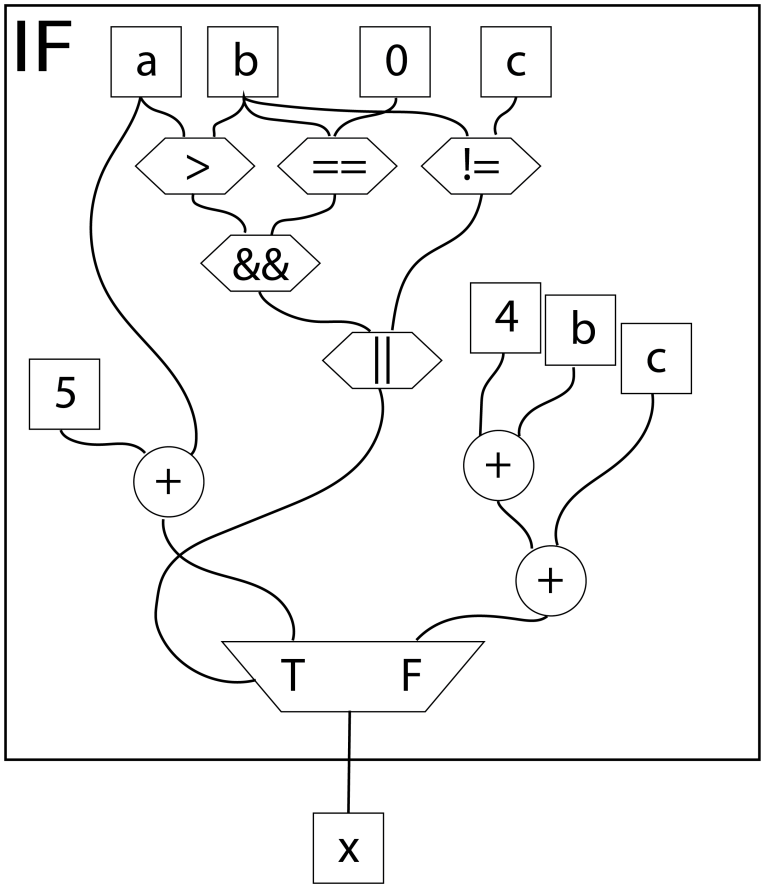
Tento typ podmienky je plnohodnotne použiteľný. Navyše umožňuje paralelné vykonávanie operácií zapojených pred mergor modulom. Minimalizuje sa doba nutná na vykonanie, nakoľko v čase kedy sa vykoná mergor, môžu byť dáta oboch vetiev podmienky vypočítané. Vykonali test viacerých podmienok a možností ich zápisu.




Obr. 2.17: Test 1, viaceré podmienky IF Selector v.1.

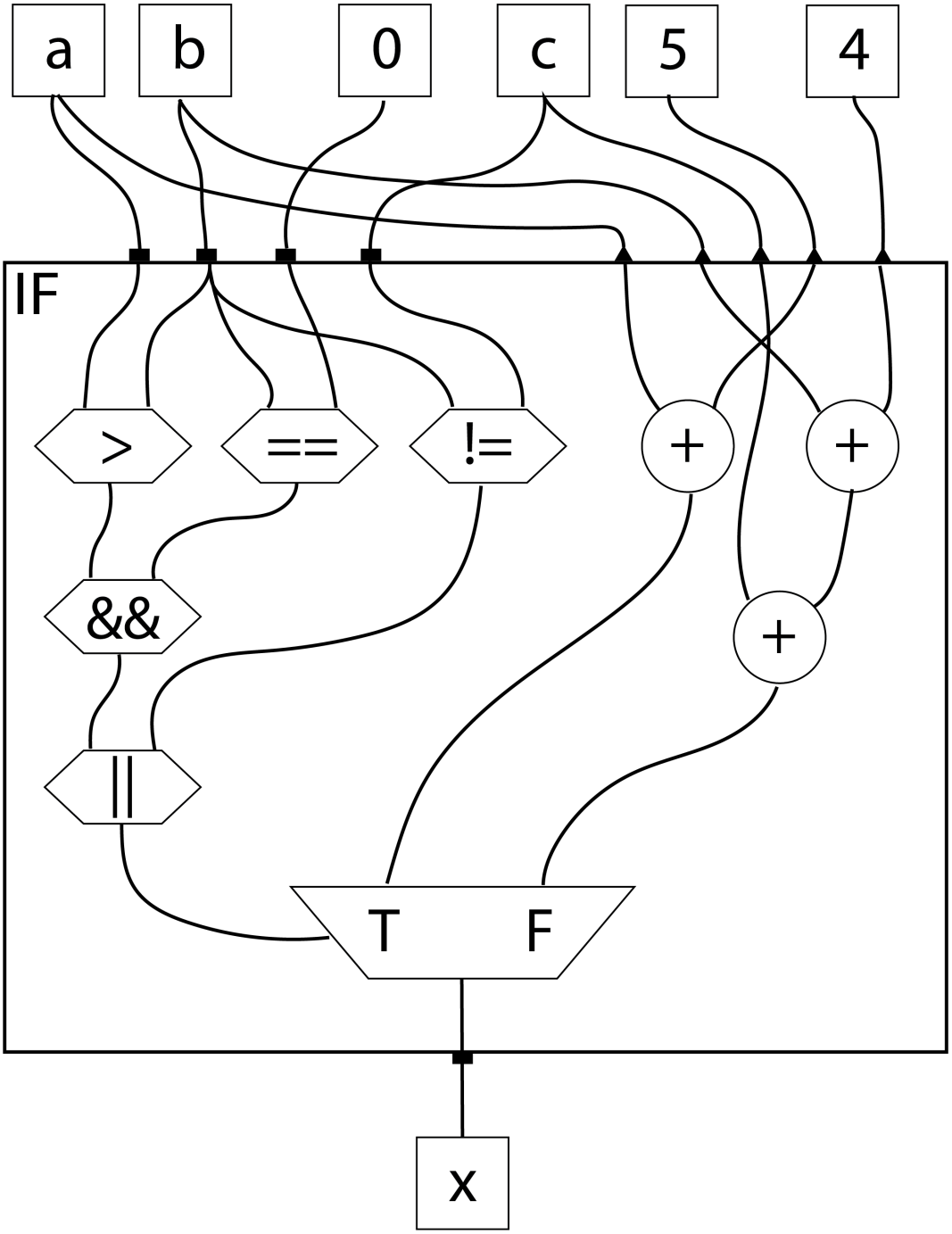
Z testov ako je na obr. 2.17 vidíme mierny pokles prehľadnosti, navyše konštrukcia podmienky zafixovanej kombinácie selector-mergor pri tomto zápise zaberá značné množstvo miesta.

Rušíme fixáciu, a skúmame zapojenie rôzneho množstva selectorov za sebou obr. 2.18.




Obr. 2.18: Test 2, upraviteľná podmienka IF Selector v.2, spolu so zapojením.

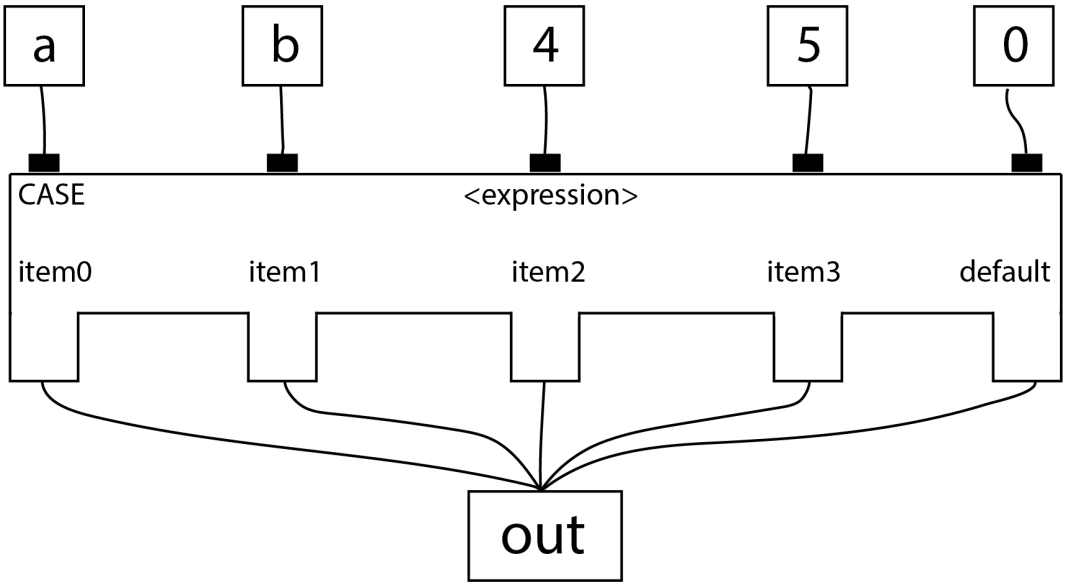
Takýto zápis sa ukazuje byť efektívnejší, avšak máme pomiešané dáta spolu s operáciami. Na obr. 2.19 skúmame možnosť oddelenia dát. Hrany sa natˈahujú, dáta sa vzďalˈujú operáciám a sledovať ich príslušnosť k operáciám sa komplikuje.




Obr. 2.19: Test 3, upraviteľná podmienka IF, dáta sú oddelené.

#### 2.3.5.3 CASE

Skúmali sme aj možnosti operátora switch-case, na obr. 2.20 zobrazujeme možný schematický zápis a zapojenie. Tento operátor je vo Verilogu dostupný.




Obr. 2.20: Case operátor, schematický zápis a možné zapojenie.

### 2.3.6 Cykly

Cykly a generátory sú prakticky nedeliteľnou súčasťou programovacích jazykov. V teoretickom modely dátových tokov sa nevyskytujú, nakoľko iterácie je možné vykonať naraz paralelne alebo sekvenčne v prípade existencie závislosti dát jednotlivých iterácií od predošlých.

Opätovne a detailne skúmame vo Verilogu konštrukciu for cyklu. Nepracuje v zmysle ako ju poznáme zo softvérových jazykov, slúži primárne na replikáciu hardvéru. S konfiguráciou FPGA ale nie je možné dynamicky manipulovať počas výpočtu, preto musí byť počet iterácií for cyklu fixný a známy pred kompiláciou. Všetky iterácie for cyklu následne zbehnú

za 1 takt.

V prípade že je počet iterácií dynamický vypočítaný počas vykonávania v FPGA, je for cyklus Verilogu nepoužiteľný. Celé FPGA sa ale nachádza v neustálom taktom riadenom cykle. Pomocou taktu a aritmetických operácií je možné zostrojiť generátor pracujúci s dynamicky vypočítanou hodnotou počas vykonávania a následne ju o nastavený krok meniť. Samozrejme takúto konštrukciu je možné zapísať aj pomocou nižších operácií, použitie generátora je ale elegantnejšie a umožňuje vznik nových druhov funkcií v našom jazyku.

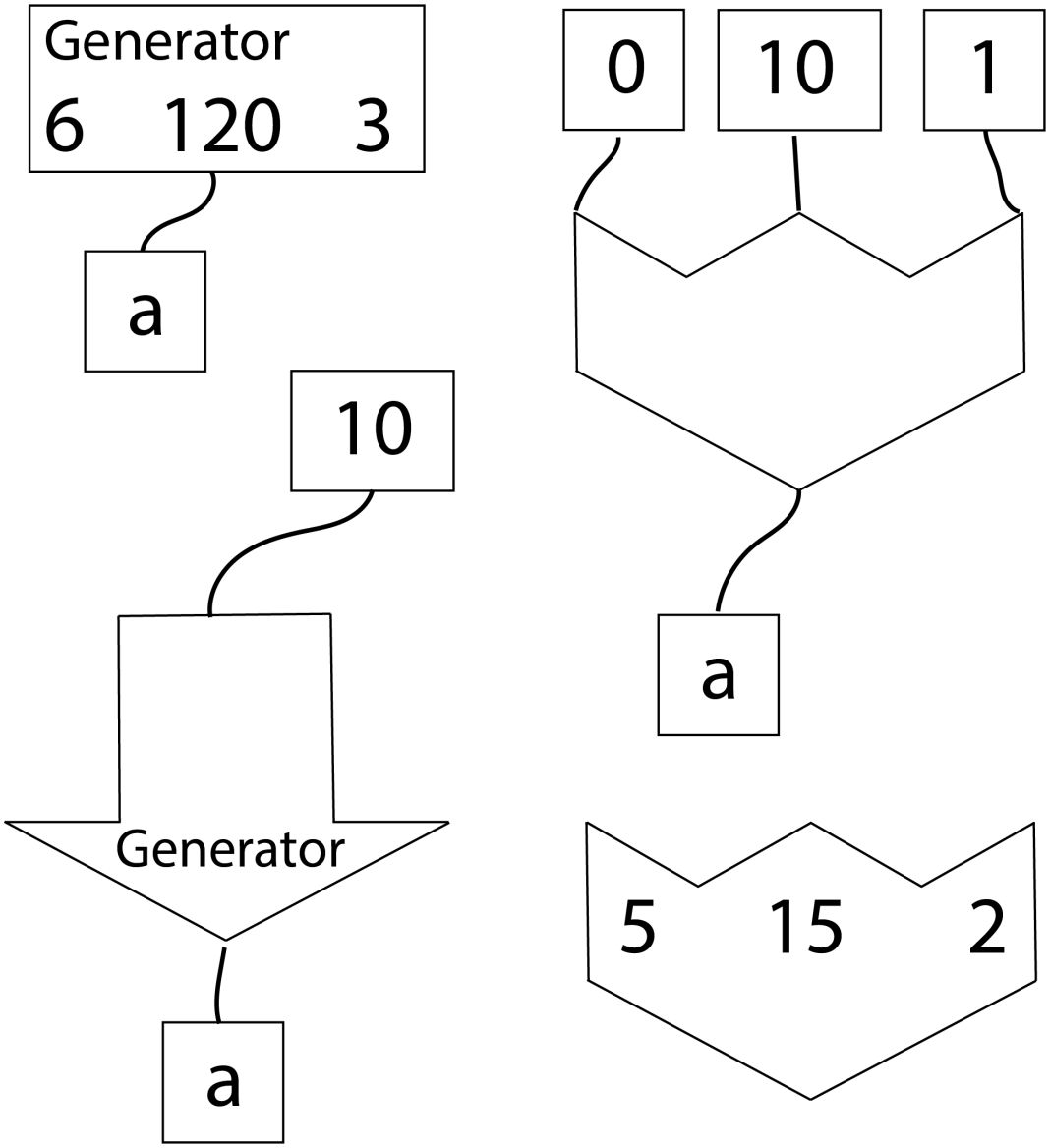
Generátor je konfigurovateľný 3 parametrami:

- od (from) - prvá hodnota ktorú generátor vyšle na výstup, predvolene nastavená na 0
- do N-1 (to N-1) - ohraničenie po ktoré sa iterácie vykonajú, posledná hodnota na výstupe je N mínus 1
- krok (step) - číslo o ktoré je hodnota na výstupe menená, predvolene 1

Príklad generovaných hodnôt:

| from | to | step | output |
|---|---|---|---|
| | 10 | | 0,1,2,...,7,8,9 |
| 5 | 10 | 2 | 5, 7, 9 |
| 10 | 2 | | 10,9,...,3,2,1 |

Na obr. 2.21 sú vyobrazené rôzne verzie zápisu, finálna podoba generátora je na pravej polovici. Hranami pripojené dáta demonštrujú dynamickú konfiguráciu generátora, v prípade statickej je možné hodnoty zapísať priamo do generátora a vizuálne znížiť počet hrán a pripojených hodnôt.

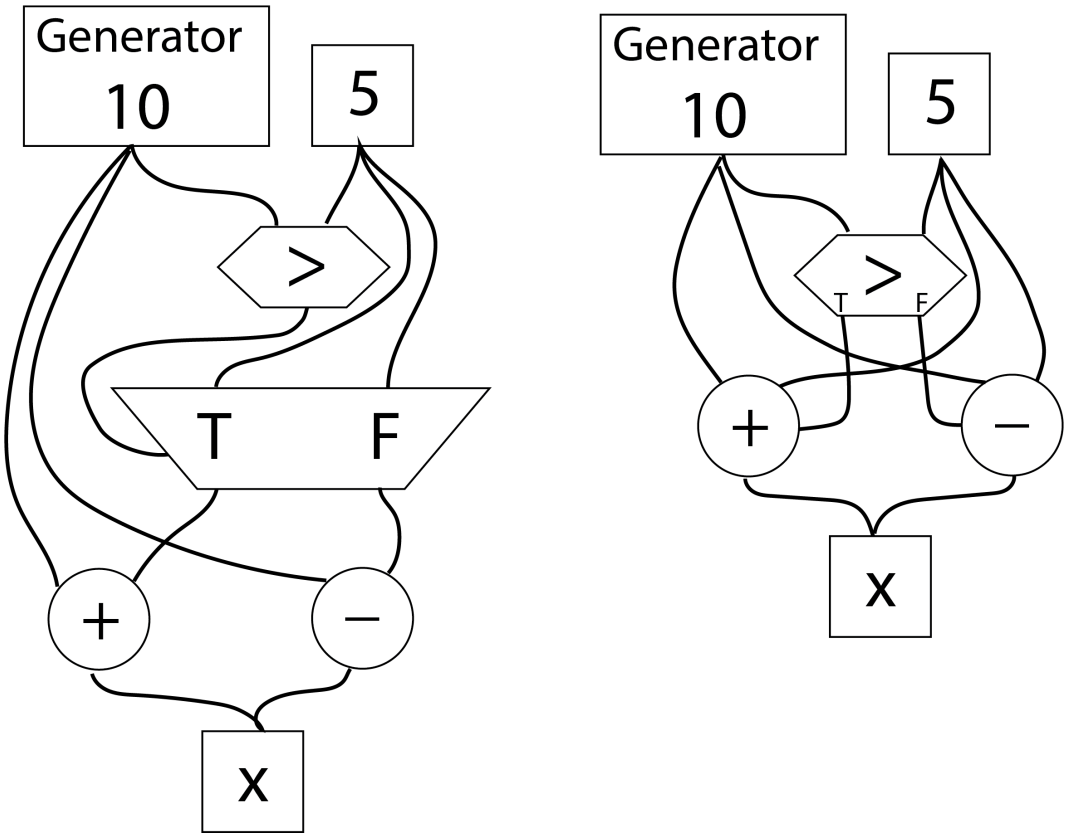



Obr. 2.21: Generátor, rôzne verzie zápisu a možné zapojenia.

### 2.3.7 Selector-mergor, trigger-selector

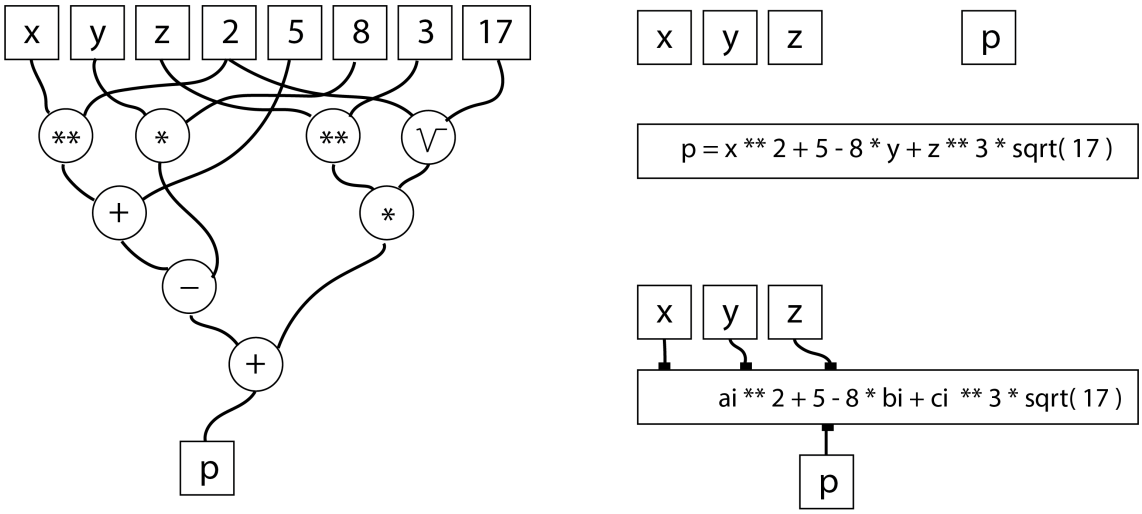
Počas experimentov s generátorom sme upravili aj selector (2.3.5). Zlučovacia brána sa stáva súčasťou selector komponentu, ktorý má miesto jedného pravdivostného výstupu dva, jeden pre každú hodnotu. Zároveň

operácie získavajú špeciálny spúšťací (trigger) port. Na obrázku 2.22 sú zobrazené obe varianty, vľavo je selector so zlučovacou bránou a operáciami riadenými dostupnosťou dát, na pravo kratší zápis no operácie sú riadene navyše riadiacim portom.




Obr. 2.22: Generator selector, selector absorbuje mergor.

### 2.3.8 Operácie aritmetické výrazy




Obr. 2.23: Výrazy aritmetických operácií.

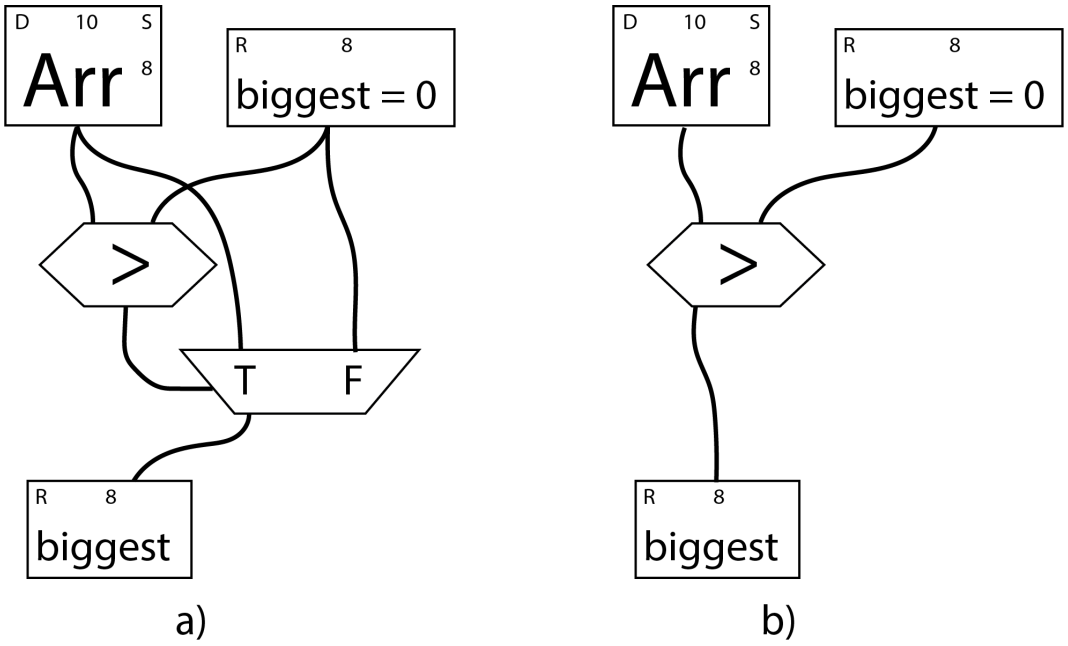
Zložitejšie aritmetické výrazy sú pri použití binárnych aritmetických operácií zdĺhavejšie na zápis, obr. 2.23 vľavo. Experimentujeme s možnosťami textového zápisu aritmetického výrazu a následného prepojenia tohoto výrazu s premennými. Na obr. 2.23 vpravo hore so zapojením bez hrán pomocou priameho použitia premenných vo výraze, vpravo dole pomocou zapájania premenných ako vstupov v poradí ich výskytu vo výraze.

### 2.3.9 Data-Selector

Upustili sme od trigger-selectora využívajúceho riadiaci signál a skúmali možnosti, ktoré by selector priblížili dátovým tokom. Vznikol dátový selector. U tohoto selectora je výstup dátový, podľa podmienky a hodnôt na vstupe, podľa tabuľky pre data-selector väčší:

| left input | right input | data output of > |
|---|---|---|
| 5 | 1 | 5 |
| 1 | 3 | 3 |

Na obr. 2.24 môžeme vidieť zápis rovnakej funkcie pomocou rôznych selectorov.




Obr. 2.24: Selector-mergor a data-selector, schematický zápis.

### 2.3.10 Najväčší prvok poľa

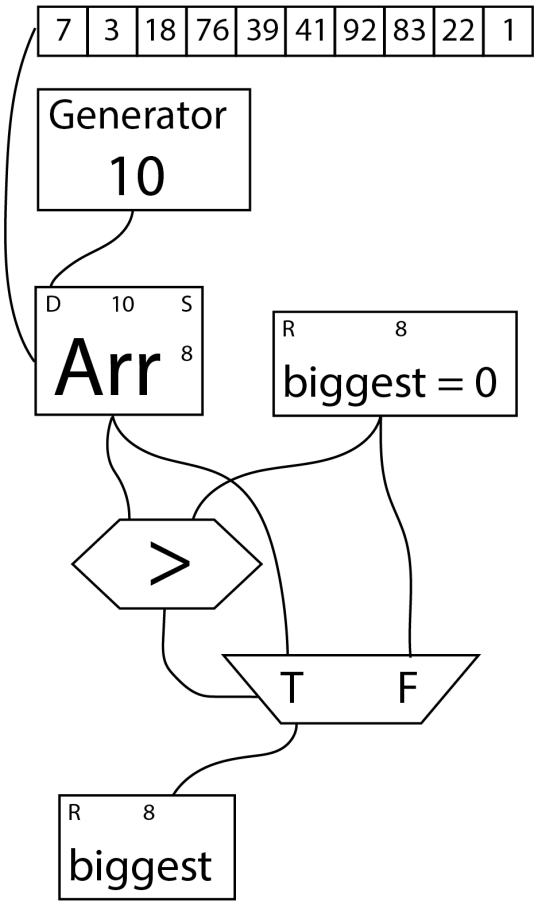
Data-selector a pôvodný selector-mergor sme testovali na jednoduchých príkladoch hľadania najväčšieho prvku poľa, z dôvodov overenia vhodnosti vizuálneho zápisu i implementačných variantov.

#### 2.3.10.1 Sekvenčný výpočet najväčšieho prvku poľa

Na obr. 2.25 uvádzame sekvenčnú implementáciu hľadania najväčšieho prvku v poli s použitím selector-mergor operácie. Premenná **Arr** je po

inicializácií indexovaná hodnotami z generátora. Na výstupe sa v každom cykle objavia príslušné dáta, ktoré sú porovnávané s hodnotou v registry **biggest**, inicializovanom na 0.

Ak podmienka v selectore platí, mergor brána na hranu odošle konkrétnu hodnotu, ktorá sa uloží do registra **biggest**. V d'alšom cykle je hodnota z nasledujúceho indexu porovnávaná s hodnotou uloženou v registry v predošlom takte. Ak podmienka neplatí, hodnota v registry **biggest** sa nemení.

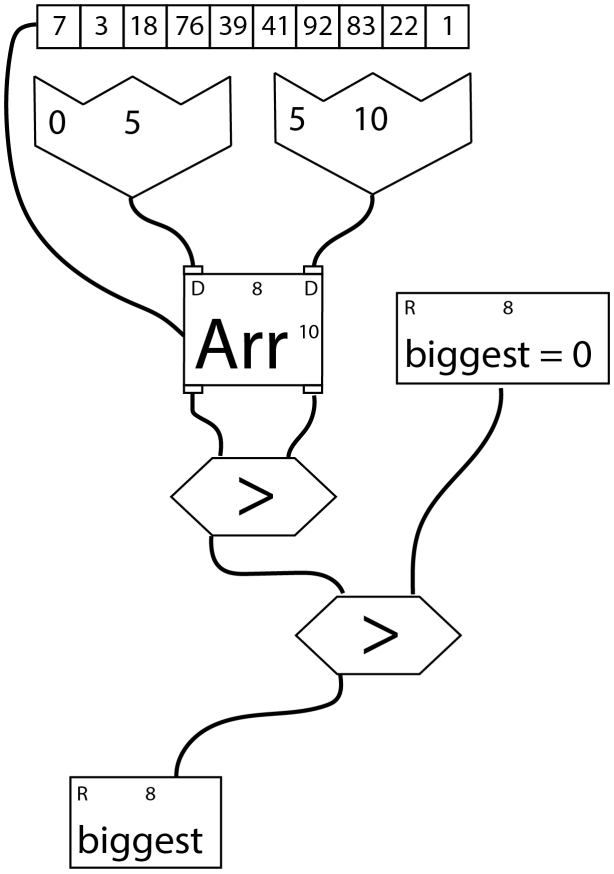



Obr. 2.25: Najväčší prvok v poli, sekvenčný výpočet, selector-gator.

#### 2.3.10.2 Čiastočne paralelný výpočet najväčšieho prvku poľa

Dátové toky sú predurčené na tvorbu paralelných algoritmov, experimenty pokračujú návrhom paralelného výpočtu najväčšieho prvku poľa.

FPGA podporuje, viac portové pamäte, s najrozšírenejším typom dual-port. S ich použitím vytvárame algoritmus na obr. 2.26. Premenná **Arr** je vytvorená pomocou dual-port distribuovanej pamäte, ktorá je paralelne indexovaná dvoma generátormi. Pamäť je logicky rozdelená na dve polovice. Hodnoty oboch idexov sú porovnávané prvým data-selectorom. Na jeho výstupe je väčšia z hodnôt porovnávaná ešte s hodnotou v registry **biggest**. Takýto výpočet skončí za polovičný počet cyklov oproti sekvenčnému 2.3.10.1.




Obr. 2.26: Najväčší prvok v poli, paralelizácia, 2 zdroje dát, data-selector.

Výpočet je možné d'alej zrýchl'ovať zvyšovaním paralelizácie, vzniká potreba paralelnej pamäte, umožňujúcej čítanie na väčšom počte portov ako 2.

### 2.3.11 Paralelná RAM

Paralelná pamäť sa ukazuje ako nevyhnutná podmienka na dosiahnutie maximálnej paralelizácie. Napriek tomu, že FPGA priamo nedisponuje multi-portovou pamäť'ou, je možné ju z logického hl'adiska simulovať prostredníctvom dostupných pamätí. Uvádzame niekol'ko možných riešení paralelnej pamäte, dostupných je však väčšie množstvo.

#### 2.3.11.1 Veľké množstvo registrov

Paralelná pamäť by bola pri kompilácii z nášho vyššieho jazyka vytvorená pomocou veľkého množstva registrov distribuovanej pamäte.

#### 2.3.11.2 Veľké množstvo dual-port RAM

Paralelná pamäť by bola pri kompilácii z nášho vyššieho jazyka vytvorená pomocou veľkého množstva dual-port block RAM.

#### 2.3.11.3 Veľká šírka

Pomocou multiplexora pracujúceho s pamäťou o veľmi veľkej šírke realizovateľnej distribuovanou tak aj blokovou pamäťou. Načítané pamäťové slovo by následne rozdelil podľa definície užívateľom napríklad na X, N-bitových hodnôt, doručených k jednotlivým operáciám. Pri zápise by naopak dané hodnoty boli zreťazené za sebou, do jedného veľkého slova.

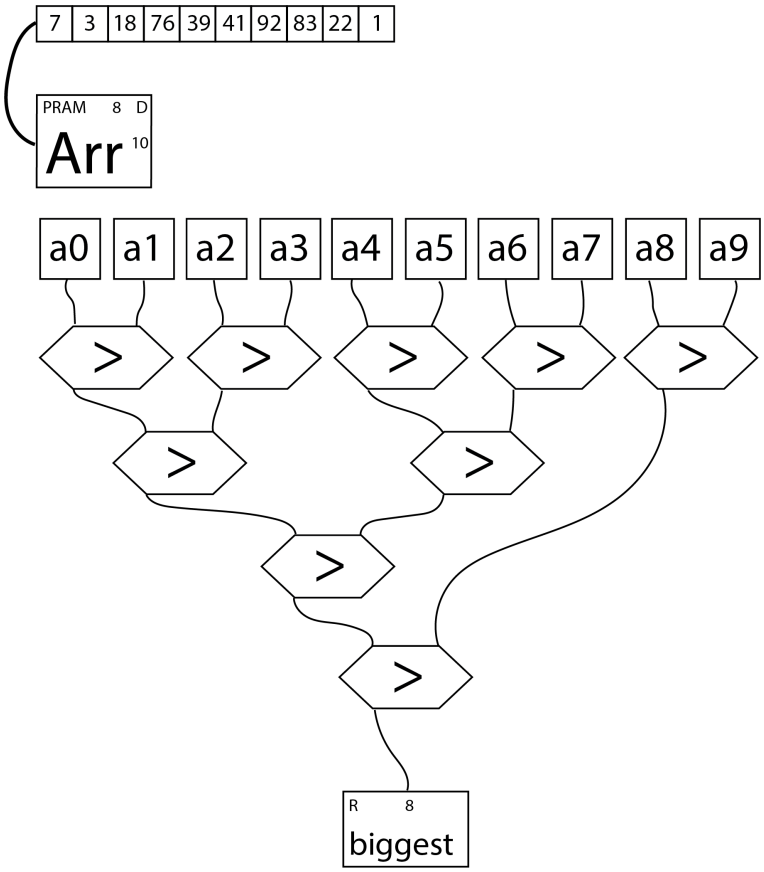
#### 2.3.11.4 BRAM na vyššej frekvencii

Dual-port BRAM môže pracovať na vyššej frekvencii ako výpočet v FPGA. Spojením takejto pamäte a vhodného multiplexora môžeme vytvárať paralelnú pamäť, s násobkom portov dual-port BRAM. Ak napríklad bude dual-port pamäť operovať na 3 násobnej frekvencii hlavného výpočtu získame 6 portov.

### 2.3.12 Paralelný výpočet najväčšieho prvku poľa

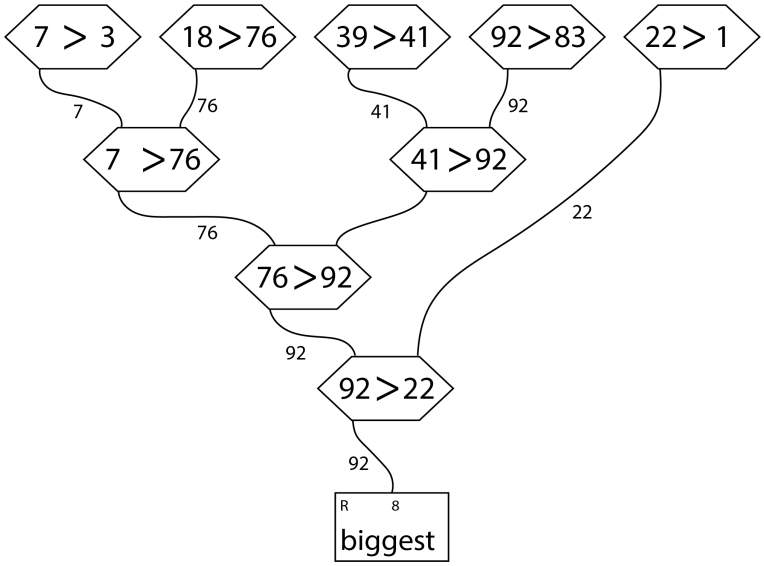
Použitím veľkého množstva registrov môžeme zvýšiť paralelizáciu na maximálnu možnú úroveň a porovnať všetky hodnoty spracúvaného poľa po

dvojiciach. Na obr. 2.27 zobrazujeme výpočet najväčšieho prvku poľa s maximálnou paralelizáciou s použitím data-selectorov.




Obr. 2.27: Najväčší prvok poľa, najväčšia paralelizácia, 10 zdrojov dát, schematický zápis.

Ukážka toku dát v takomto výpočte je zobrazená na obr. 3.1.




Obr. 2.28: Najväčší prvok poľa, najväčšia paralelizácia, ukážka toku dát vo výpočte.

### 2.3.13 Duplikovanosť

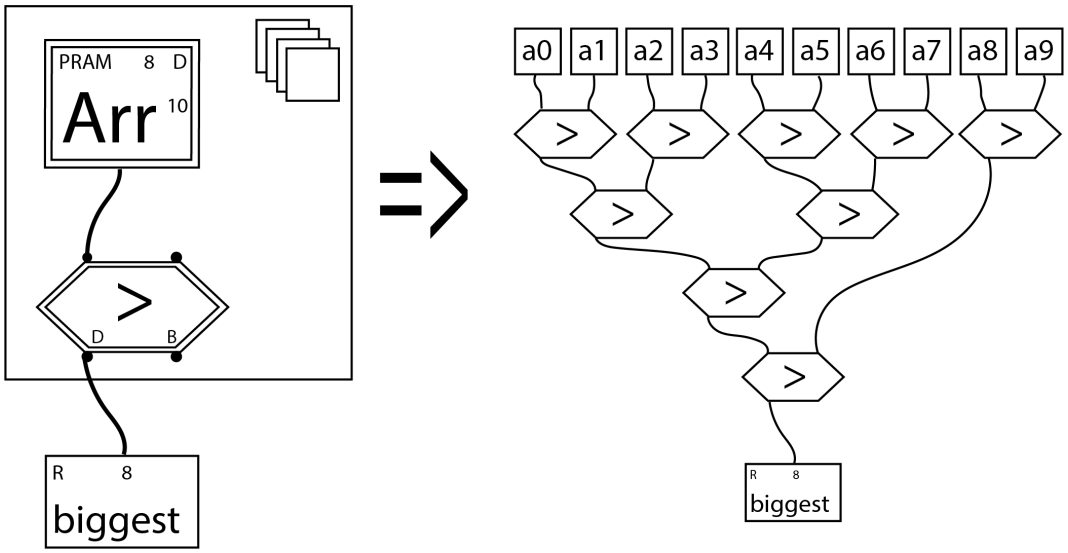
Pri pokusoch s paralelizáciou a hľadaním najväčšieho prvku poľa sme pri opakovanom zápise veľkého množstva operácií dospeli k potrebe operácie v našom jazyku umožňujúcej zápis duplikovanosti. Duplikovanosť môžeme rozdeliť podľa rovnorodosti rozdeliť na homogénnu a heterogénnu.

#### 2.3.13.1 Homogénna duplikovanosť

V homogénnej duplikovanosti skracujeme zápis rovnorodých štruktúr. Duplikované operácie majú zdvojené orámovanie a celé sú navyše ohraničené. Programátor vo vývojovom prostredí určí hlavný element a nastaví fixnú šírku duplikovanosti.

**Stromová duplikácia** Stromová duplikácia prebieha od listov ku koreňu. Užívateľ nastaví šírku (mohutnosť) duplikácie listom, od ktorých podľa použitých operácií v jednotlivých vrcholoch výpočet smeruje ku koreňu.

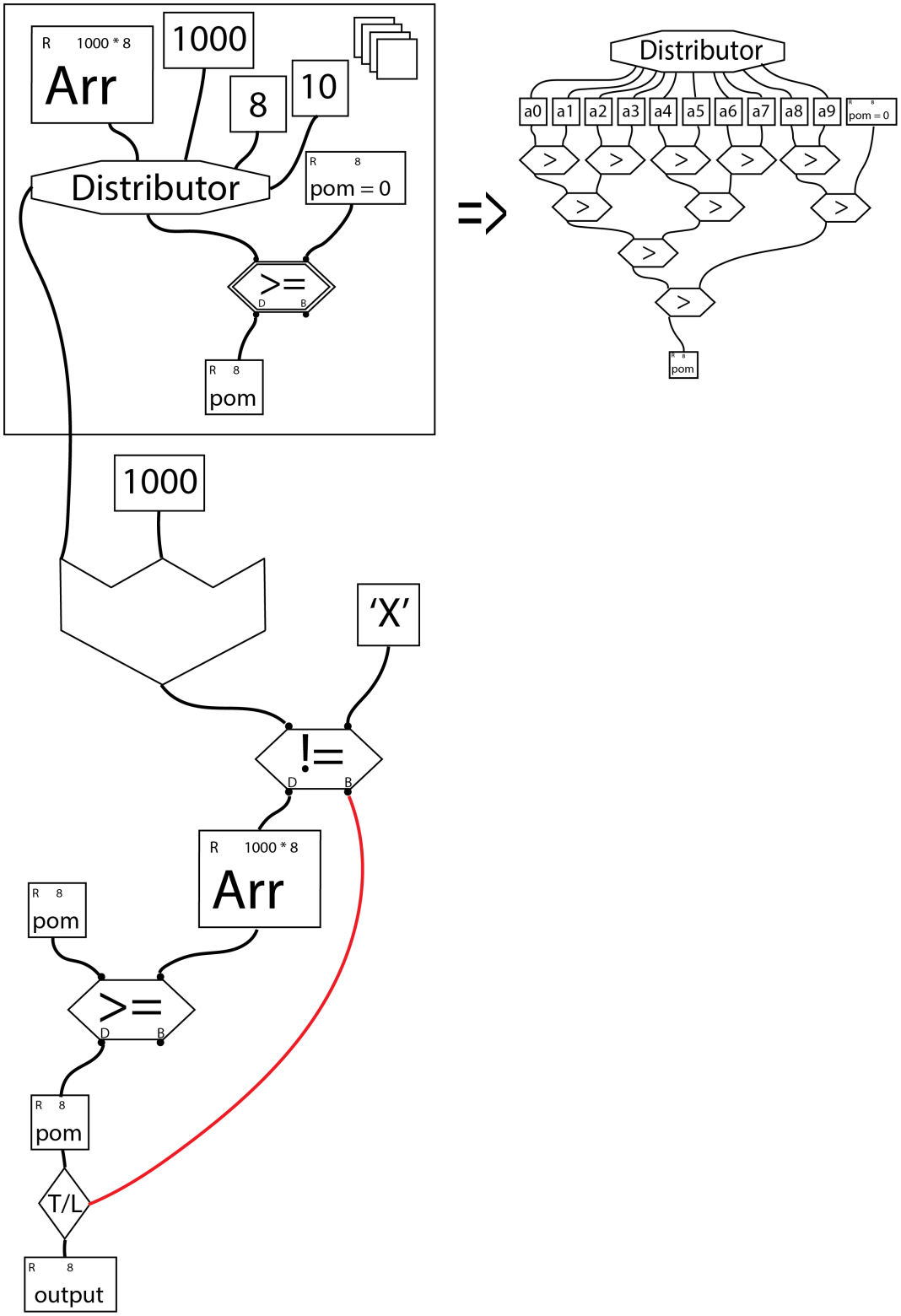
Na obr. **??** je zobrazený schematický zápis homogénnej duplikovanosti, hľadania najväčšieho prvku v poli. Listy tvoria registre paralelnej pamäti **Arr** a majú nastavenú mohutnosť 10.




Obr. 2.29: Homogénna stromová duplikovanosť, schematický zápis a vnútorná reprezentácia.

### 2.3.13.2 Šírka paralelizácie

Využitím duplikovanosti je možné pracovať so šírkou paralelizácie. Pomocou jej zmeny je možné meniť rýchlosť výpočtu, zároveň aj spotrebu hardvérových prostriedkov.


R 1000 * 8
Arr
1000
8
10
Distributor
R 8
pom = 0
>=
D B
R 8
pom
=>
Distributor
a0
a1
a2
a3
a4
a5
a6
a7
a8
a9
R 8
pom = 0
R 8
pom
1000
'X'
!=
D B
R 1000 * 8
Arr
R 8
pom
>=
D B
R 8
pom
T/L
R 8
output

Obr. 2.30: .

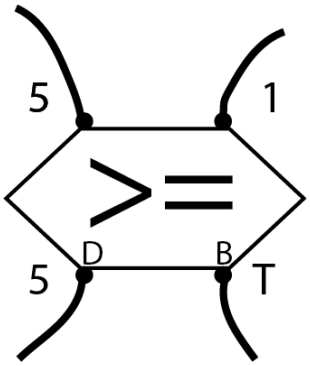
#### 2.3.13.3 Heterogénna duplikácia

Rôznorodé štruktúry navrhujeme duplikovať pomocou zlučovania častí programov - podgrafov do kondenzovaných operácií. Tieto operácie by následne mohli byť duplikované a daná časť výpočtu paralelizovaná. Synchronizáciu poradia vstupných dát voči výstupným by zabezpečovali multiplexory.

### 2.3.14 Data-Boolean-Selector

Počas testov jazyka sa ukázala potreba opäť pozmeniť data-selector, ktorý sme obohatili aj o pravdivostný (boolean) riadiaci výstup. Ukazovala sa totiž potreba synchronizácie dát pri výpočte alebo rozhodovania pri zlučovaní dát. Táto potreba sa pri riešení implementácie prejavila v plnej miere.

Na obr. 2.31 vidíme zápis DBselectora spolu so vstupnými a výstupnými hodnotami. Nakoľko je podmienka splnená, na dátovom výstupe sa objaví hodnota 5, na pravdivostnom T ako true (pravda).




Obr. 2.31: DBselector.

### 2.3.15 Špeciálne operácie

Implementácia si vyžiadala d'alší typ operácií. S FPGA a syntetizovaným návrhom je potrebné komunikovať. Spôsob komunikácie a použitá zbernica je ovplyvnená použitým FPGA, prípadne samotnou vývojovou doskou, ktorej je súčasťou. Zbernica UART je jedným z najdostupnejších komunikačných prostriedkov v FPGA a je dostupná aj na použitej vývojovej doske Arty.

#### 2.3.15.1 UART

UART reprezentujeme ako operáciu zloženú z dvoch častí, vstupnej a výstupnej. Pomocou nich zabezpečíme komunikáciu medzi FPGA a počítačom.

## 2.4 Sekvenčný faktoriál

Experimenty so sekvenčným faktoriálom, testujeme rôzne spôsoby vizuálnych reprezentácií, riadenia výpočtov a implementácií.

Na obr. 2.32 zobrazujeme schematický zápis sekvenčného zápisu faktoriálu. Hrany označené červenou farbou nesú iba pravdivostné hodnoty. Vstupnú hodnotu s UARTu porovnávame s 0, ak podmienka platí hodnota je uložená v registry **n**, a hodnota **True** aktivuje TActivator a výpočet pokračuje. Ak podmienka neplatí, do registra **out** je priradená hodnota **-1** znamenajúca neplatný vstup. V TActivatore v l'avej strednej časti sú 3 podmienky, porovnávajúce hodnotu registra **n** s hodnotami 0, 1, a 2. Všetky podmienky sa vykonajú zároveň.

Ak je platné porovnanie s 0, hodnota 0 vystupujúca zo selectora je zvýšená na hodnotu 1 v unárnej operácii zvýšenie (increment) a uložená

do registra **out**.

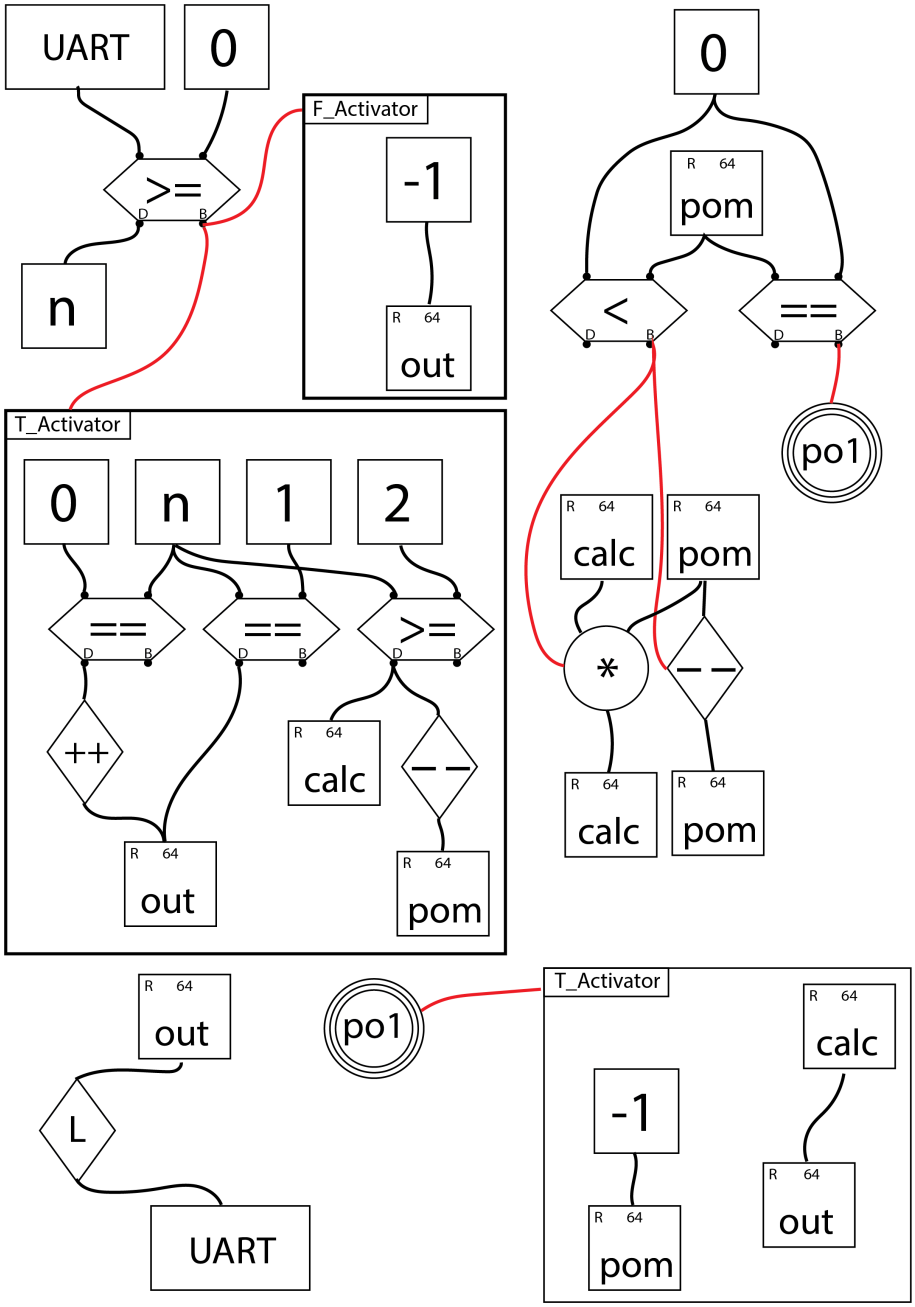
Platné porovnanie s hodnotou 1 uloží 1 do registra **out**.

Hodnota väčšia rovná ako 2, aktivuje tretí dbselector a výstup je uložený do registra **calc** a zmenšený o hodnotu 1 v unárnom operátore zmenšenie (decrement) a uložený do **pom** registra.

Zmena hodnoty registra **pom** spustí výpočet na pravej strane, kde v cykle vďaka zmenšovaniu **pom** dôjde k výpočtu faktoriálu. Binárna operácia násobenia a operácia zmenšenia obdržia zo selectora menší po riadiacej hrane pravdivostnú hodnotu, ktorá riadi ich vykonávanie.

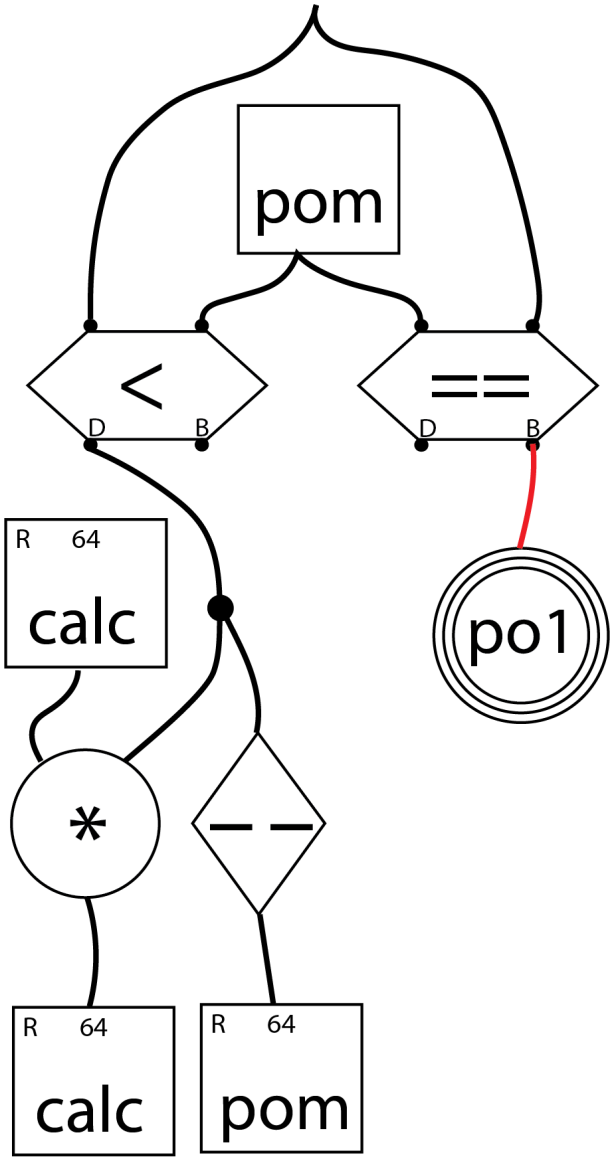
Selector porovnávajúci **pom** hodnotu 0, svojim vykonaním zabezpečí doručenie výsledku na výstup. Riadiacou hranou aktivuje priradenie hodnoty **calc** na **out**. Operácia **po1** je akýmsi portálom, čisto vizuálneho charakteru, bez ovplyvnenia výpočtu. Ide o pokus na zlepšenie prehľadnosti programu, ktorý vizuálne preruší hranu, ktorá by sa inak tiahla na veľkú vzdialenosť.

Hodnota **out** registra je pri zmene pomocou unárnej operácie Listener následne odoslaná na spracovanie operácií UART, ktorá zabezpečí jej vysielanie do pripojeného zariadenia.



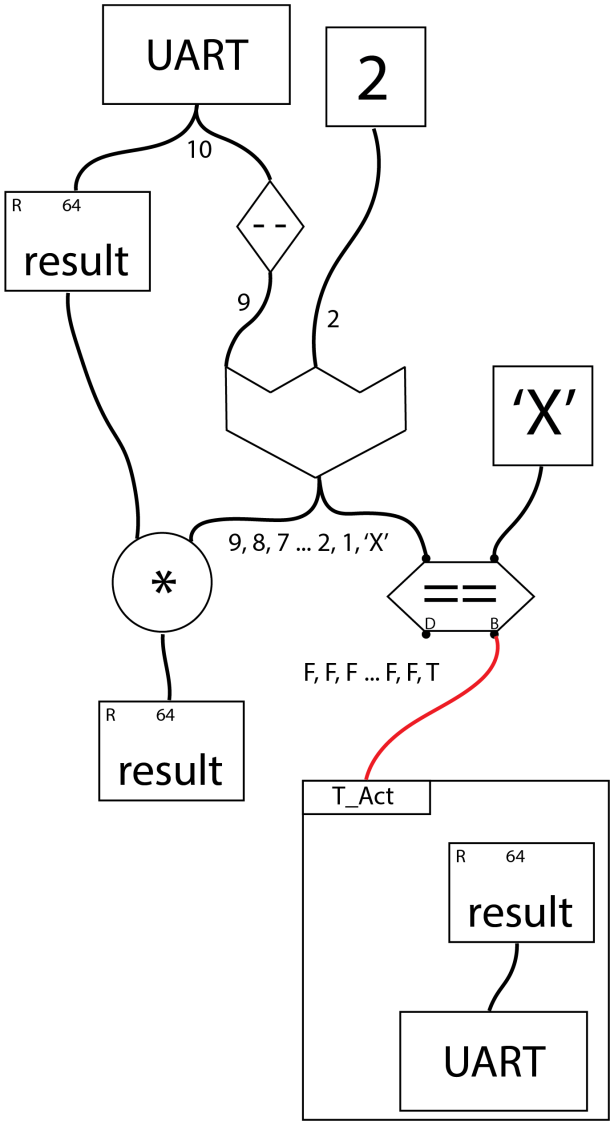

Obr. 2.32: Sekvenčný výpočet faktoriálu, použitie aktivátorov na riadenie výpočtu, schematický zápis.

Na obr. 2.33 testujeme možnosť minimalizovania používania riadiacich hrán.

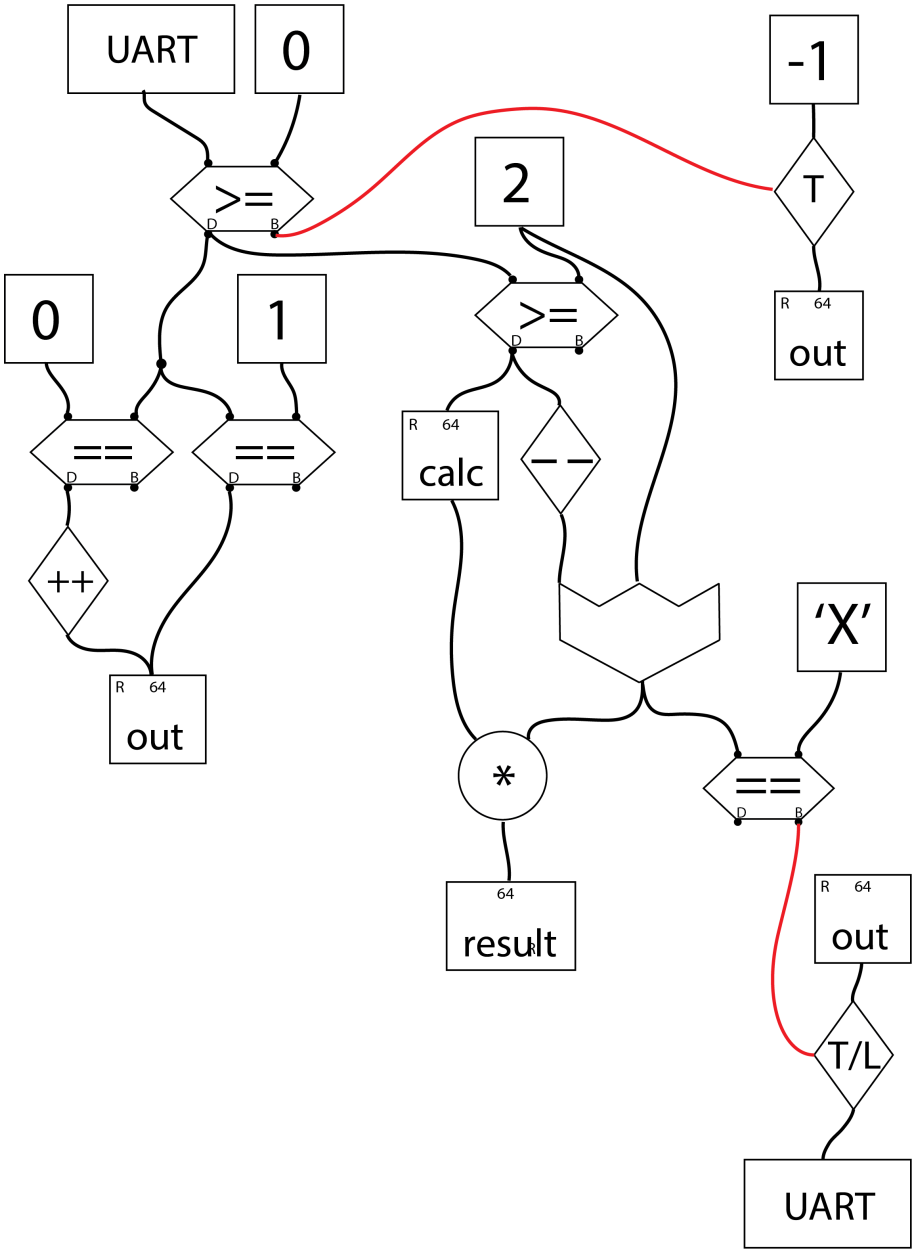



Obr. 2.33: Sekvenčný faktoriál, úprava na dátami riadený výpočet, schematický zápis.

Program na výpočet faktoriálu s využitím generátora a testom práce so signálom **X** môžeme vidieť na obr. 2.34 a v rozvinutejšej verzii na obr. 2.35. Práca so signálom X a Z sa vo Verilogu ukázala ako nepraktická a od tejto formy zápisu a implementácie sme upustili.




Obr. 2.34: Sekvenčný faktoriál s využitím generátora a signálu X, schematický zápis.




Obr. 2.35: Sekvenčný faktoriál s využitím generátora a signálu X, schematický zápis.

# Kapitola 3

# Implementácia

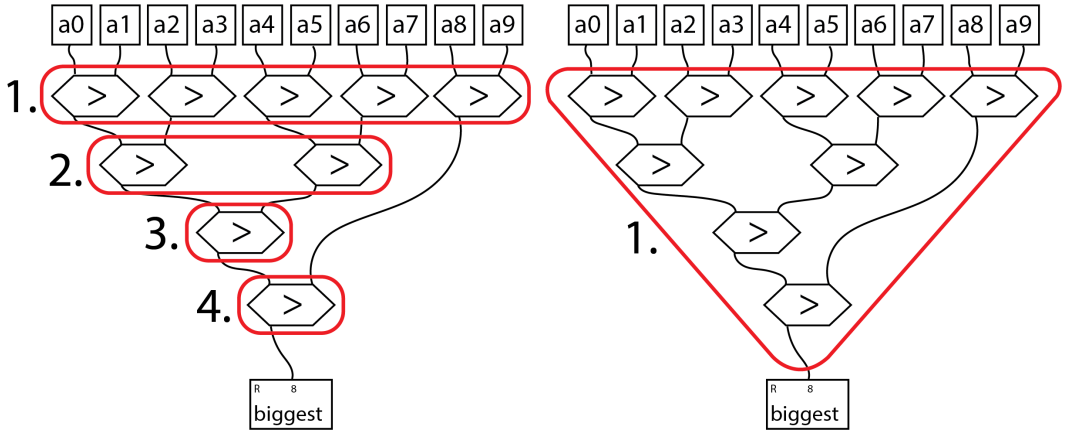
Verilog pozná 3 druhy modelovania hardvéru, ktoré umožňujú viaceré implementácie, dosahujúce rôzne úrovne výpočtovej rýchlosti pri danej paralelizácii. Na výpočte najväčšieho prvku poľa demonštrujeme možné implementácie.

Zdrojové kódy - hardvérové popisy uvedené v tejto kapitole sú simulovateľné a plne syntetizovateľné - na základe týchto popisov je v FPGA zapojený obvod, ktorý sa správa podľa charakteristiky daného modelovania a konkrétneho popisu.

## 3.1 Najväčší prvok poľa

Pri experimentoch počas návrhu jazyka sme sa dopracovali k dvom odlišným spôsobom vykonávania. Dáta počas výpočtu prúdia medzi etapami, časti výpočtu môžu byť rozdelené na etapy alebo zlúčené do jednej etapy.

Na obr. 3.1 je zobrazené porovnanie rôzneho vykonávania, v ľavej polovici je hlavná časť výpočtu rozdelená na etapy, v pravej zlúčená do jednej.




Obr. 3.1: Paralelné vykonávanie, viacero etáp a jedna etapa.

Synchronizácia operácií v rámci etapy a synchronizácia etáp vzhľadom na použité modelovanie a paralelná distribúcia dát spolu súvisia a sú kľúčovými prostriedkami pri návrhu a riešení implementácie navrhovaného vyššieho jazyka.

### 3.1.1 Viacero etáp

Všetky etapy sú vykonávané zároveň v jednotlivých taktoch FPGA. Dáta sa v každom cykle presúvajú z etapy na etapu. Samotný výpočet trvá 4 cykly, dáta z 1. etapy sa v nasledujúcom takte presunú do 2. etapy a zároveň dôjde k ich vykonaniu.

Pri začiatku výpočtu, trvá prvej sade dát výpočet 4 cykly, následne je nový výsledok výpočtu doručený v každom cykle.

Všetky operácie v tomto druhu výpočtu sú synchronizované a riadené taktom. Tok dát medzi operáciami je taktiež riadený taktom. Použitý dataselector v tomto výpočte zobrazujeme v zdrojovom popise hardvéru 3.1, vytvoreného pomocou behaviorálneho modelovania 1, používajúci taktom riadený výpočet (always @(posedge clk)) s neblokujúcim priradením.

```
module selector_b(data_out, lin, rin, clk);

input clk;
input [7:0] lin;
input [7:0] rin;
output reg [7:0] data_out;

always @(posedge clk) begin
    if (lin > rin) begin
        data_out <= lin;
    end
    else begin
        data_out <= rin;
    end
end
endmodule
```

Listing 3.1: Taktom riadený behaviorálne modelovaný data-selector

### 3.1.2 Jedna etapa

Všetky operácie sú v tomto type výpočtu vykonané v rámci jedného taktu. Prvá sada dát je vypočítaná v 1 cykle, v každom ďalšom cykle je doručený výsledok nasledujúcej sady dát.

Operácie, výpočet samotný v jednotlivých etapách je riadený zmenou dát, etapy navzájom a prúdenie dát medzi nimi môže byť synchronizované zmenou dát alebo taktom.

V zdrojovom popise 3.2 uvádzame behaviorálne modelovanie 1, zmenou dát na vstupoch operácie (always @(*)) riadený výpočet s neblokujúcim priradením.

```
module data_selector_b(data_out, lin, rin, clk);
```

```
2
3  input clk;
4  input [7:0] lin;
5  input [7:0] rin;
6  output reg [7:0] data_out;
7
8  always @(*) begin
9      if (lin > rin) begin
10         data_out <= lin;
11     end
12     else begin
13         data_out <= rin;
14     end
15 end
16 endmodule
```

Listing 3.2: Zmenou dát riadený behaviorálny data-selector

Výpočet vykonaný v rámci jednej etapy je možné vytvárať vo Verilogu aj pomocou modelovania dátovým tokom. V zdrojovom popise 3.3 uvádzame modelovanie dátovým tokom 2, neustále prebiehajúci výpočet s kontinuálnym priradením, s okamžitou reakciou na zmenu vstupu.

```
1  module data_selector_b(data_out, lin, rin, clk);
2
3  input clk;
4  input [7:0] lin;
5  input [7:0] rin;
6  output wire [7:0] data_out;
7
8  assign data_out = (lin > rin) ? lin : rin;
9
```

```
endmodule
```

Listing 3.3: Data-selector modelovaný dátovým tokom s neustálym priradením

## 3.2 Finálny návrh jazyka

Vo finálnom návrhu jazyka uvádzame operácie a komponenty jazyka z návrhu, ktoré boli vybrané spolu s implementačnými informáciami. Zredukovali a ujednotili sme vizuálne zápisy, vynechali niektoré operácie a doplnili pár implementačných. Vo vývojovom prostredí obohacujeme reprezentáciu grafických návrhov o porty a vypúšťame zobrazenie s plnými detailami (full view).

Operácie implementujeme pomocou behaviorálneho modelovania, riadené zmenou dát alebo taktom, používajúc neblokujúce priradenie.

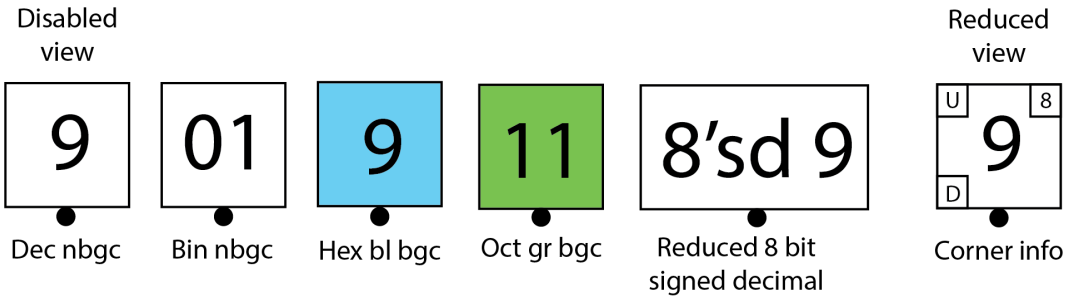
## 3.3 Syntax a implementácia

### 3.3.1 Hodnoty

Vizuálna reprezentácia, zápis hodnôt sa oproti návrhu mení minimálne. V tejto verzii jazyka znemožňujeme v hodnotách používanie signálov X a Z **??**, nakoľko prinášajú ďalší rozmer zložitosti pri ich používaní. V budúcich verziách jazyka ich použitie ale nevylučujeme, nakoľko sú súčasťou Verilogu.

#### 3.3.1.1 Číselné hodnoty

Reprezentáciu číselných hodnôt nemeníme **??**, vynechávame INT (reprezentácia dvojitým doplnkom, 2′complement form) a FLOAT (repr. plávajúcou desatinnou čiarkou), nakoľko pre ich použitie je nutná rozsiahlejšia úprava aritmetických operátorov. Pripájame reprezentáciu typov z návrhu vo Verilogu, všetky reprezentácie môžu byť zadané aj manuálne.



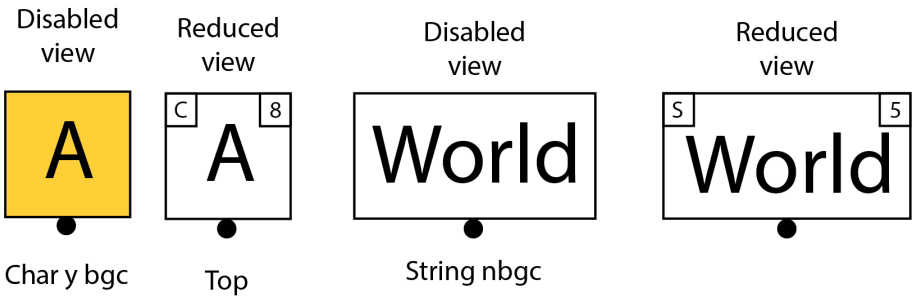

Obr. 3.2: Hodnoty, výsledné formy zápisu.

```
1'b1;         //BIT 1bit binary
4'd12;        //NIBBLE unsigned 4bit decimal
8'b00001001; //BYTE unsigned 8bit binary
16'sh53F1;    //SHORT signed 16bit hexadecimal
64'b0110;     //LONG unsigned 64 bit, zero auto-filled in missing bits
```

Listing 3.4: Verilog reprezentácia číselných hodnôt

#### 3.3.1.2 Textové hodnoty

Znaky a textové reťazce sú bez technických zmien, aj tu vynechávame zápis obsahujúci všetky detaily.




Obr. 3.3: Textové hodnoty.

```
"a";                //char
"Hello, World.";    //string
```

Listing 3.5: Verilog reprezentácia znaku a textového reťazca

### 3.3.2 Pamäť

Uvádzame finálne typy hrán a pamätí.

#### 3.3.2.1 Hrany

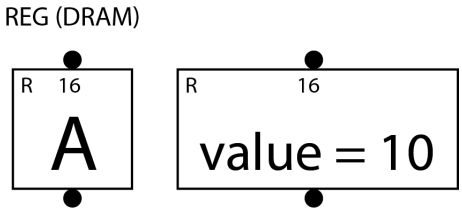
Pre distribúciu dát a logických hodnôt medzi operáciami implementujeme sieťové hrany. Ostatné typy je možné v budúcnosti doplniť spolu so synchronizačnými mechanizmami.

**Sieťové** Šírka hrán je definovaná automatizovane podľa šírky výstupných portov, na ktoré je hrana pripojená. Je možná ručná úprava užívateľom. Vizuálna reprezentácia jednoduchou spojitou čiarou z návrhu ostáva zachovaná, riadiace čiary nesúce iba pravdivostnú hodnotu sú odlíšené červenou farbou.

#### 3.3.2.2 Premenné

Premenné implementujeme prostredníctvom distribuovanej pamäte kompiláciou do Verilogu, umožňujeme aj manuálne použite blokovej pamäte, avšak je vyžadovaná ručná konfigurácia užívateľom, pomocou IP generátora prostredia Vivado. V budúcnosti je možné tento problém vyriešiť pomocou TCL scriptu, ktorý bude počas kompilácie nášho jazyka vytvorený a vygeneruje podľa nastavení potrebnú blokovú pamäť.

**Premenné** Na obr. **??** sú zobrazené premenné, implementované pomocou registrov. Na vrchnej časti obsahujú port na zápis dát, na spodnej na čítanie. Premenná **value** je inicializovaná na hodnotu 10. Šírka uložených dát je zobrazená v hornej časti.




Obr. 3.4: Premenné, registre s portami zapojenia, šírka 16 bitov.

```
reg [15:0] A;                //variable A from picture, size 16 bit
reg [15:0] value = 16'd10;   //variable pom from picture, size 16 bit
reg [7:0] data_out1;         //variable data_out1, size (data width) 8 bit
reg [31:0] pom;              //variable pom, size (data width) 32 bit
```

Listing 3.6: Verilog reprezentácia premenných

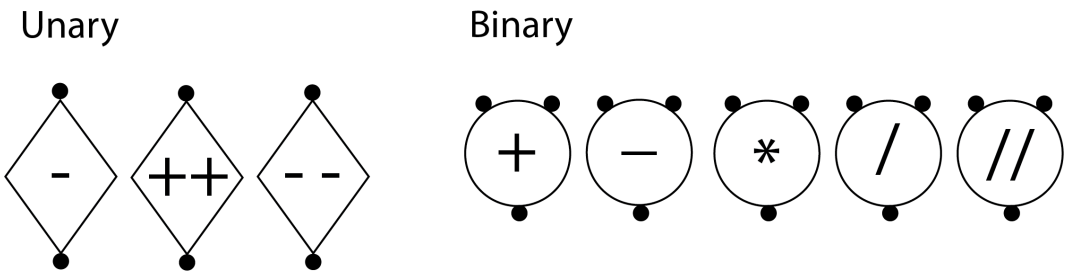
### 3.3.3 Operácie

Všetkým operáciám je možné vo vývojovom prostredí nastaviť šírku dát s ktorými majú pracovať. Implementácia behaviorálnym programovaním,

zmenou dát riadený výpočet.

#### 3.3.3.1 Aritmetické

N-árne operácie sme vynechali a pridali sme možnosť výrazov.




Obr. 3.5: Unárne a binárne operácie.

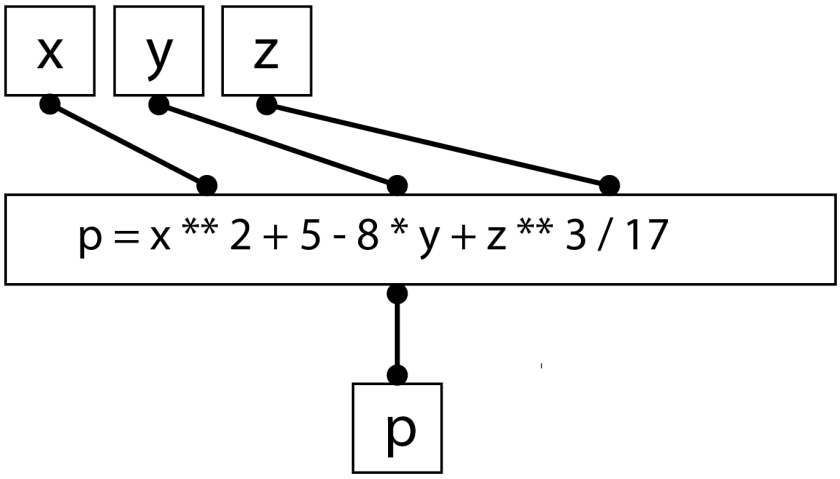
```
module arithmetic_unary_inc(dataout, datain);
input wire [7:0] datain;
output reg [7:0] dataout;

always @(*) begin
    dataout <= datain + 8'b00000001;
end
endmodule
```

Listing 3.7: Verilog implementácia unárneho zvýšenia (increment)

```
module arithmetic_binary_mul(data_out, lin, rin);
input wire [7:0] lin;
input wire [7:0] rin;
output reg [7:0] data_out;

always @(*) begin
    data_out <= lin * rin;
end
```

```
endmodule
```

Listing 3.8: Verilog implementácia binárneho násobenia (multiply)




Obr. 3.6: .

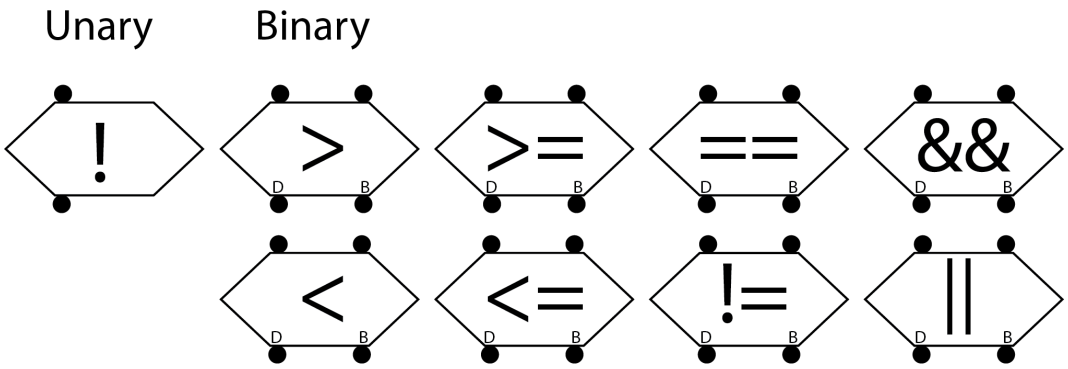
```
module arithmetic_exp(p, x, y, z);
input wire [31:0] x;
input wire [31:0] y;
input wire [31:0] z;
output reg [31:0] p;

always @(*) begin
    p <= x ** 2 + 5 - 8 * y + z ** 3 / 17;
end
endmodule
```

Listing 3.9: Verilog implementácia výrazu (expression)

#### 3.3.3.2 Logické operácie

**Unárne** Logický zápor má iba po jednom dátovom vstupe a výstupe.

**Binárne** DBselector, dva dátové vstupy v hornej časti, v l'avej dolnej časti (označený malým D) je dátový výstup, na pravej strane logický výstup (bool, označený B).




Obr. 3.7: Logické operátory, dbselector.

```
module selector_be(data_out, bool_out, lin, rin);
input [7:0] lin;
input [7:0] rin;
output reg [7:0] data_out;
output reg bool_out;

always @(*) begin
    if (lin >= rin) begin
        data_out <= lin;
        bool_out <= 1'b1;
    end
    else begin
        data_out <= rin;
        bool_out <= 1'b0;
    end
end
```

```
endmodule
```

Listing 3.10: Verilog implementácia dbselectora bigger-equal (väčší-rovný)

Dátovo pravdivostná tabuľka dbselectora väčší rovný (>=):

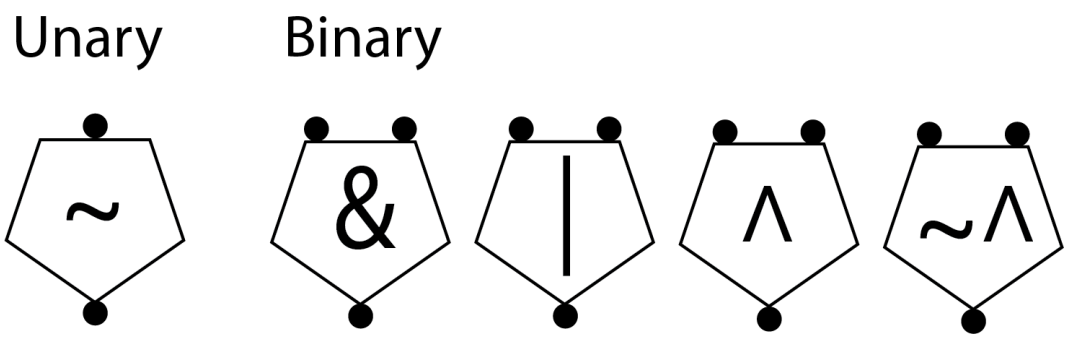
| lin | rin | data out | bool out |
|---|---|---|---|
| 5 | 1 | 5 | True |
| 1 | 3 | 3 | False |

Dátovo pravdivostná tabuľka dbselectora rovný (==):

| lin | rin | data out | bool out |
|---|---|---|---|
| 5 | 5 | 5 | True |
| 1 | 3 | 3 | False |

#### 3.3.3.3 Bitwise a Identity

Bitwise operátory používame rovnaké ako Verilog. Unárnu negáciu, binárne AND, OR, XOR, XNOR.




Obr. 3.8: Bitwise operátory.

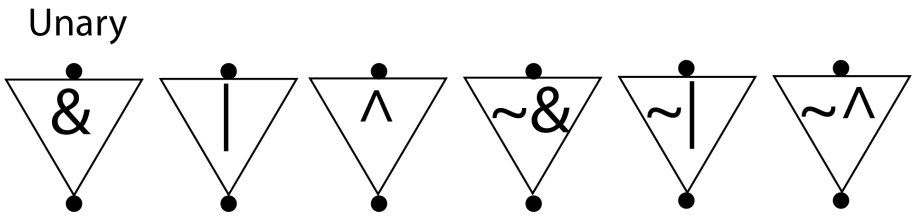
```
module bitwise_xor(data_out, lin, rin);
input [7:0] lin;
input [7:0] rin;
output reg [7:0] data_out;

```

```
always @(*) begin
    data_out <= lin ^ rin;
end
endmodule
```

Listing 3.11: Verilog implementácia bitwise XOR

#### 3.3.3.4 Redukčné

Implementujeme všetky redukčné operátory, ktoré Verilog obsahuje, AND, NAND, OR, NOR, XOR, XNOR.



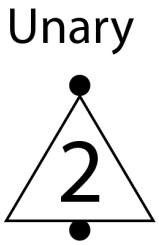

Obr. 3.9: Redukčné operátory.

```
module reduction_nand(data_out, data_in);
input [7:0] data_in;
output reg data_out;

always @(*) begin
    data_out <= ~&data-in;
end
endmodule
```

Listing 3.12: Verilog implementácia reduction NAND

#### 3.3.3.5 Replikačné

Počet replikácií sa zapisuje priamo do operátora. Na obr. 3.10 zobrazujeme replikáciu dvomi.



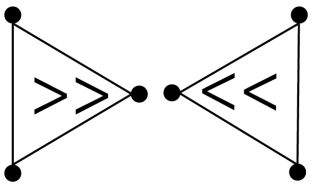

Obr. 3.10: Replikačný operátor.

```
module replication(data_out, data_in);
input [7:0] data_in;
output reg [2*7:0] data_out;

always @(*) begin
    data_out <= {2{data_in}};
end
endmodule
```

Listing 3.13: Verilog implementácia replication by 2

#### 3.3.3.6 Shiftovacie

Horný a dolný port sú dátové, bočný určuje o koľko bitov shiftnutie prebehne, voľné pozície sú doplnené nulami.




Obr. 3.11: Shiftovacie operátory.

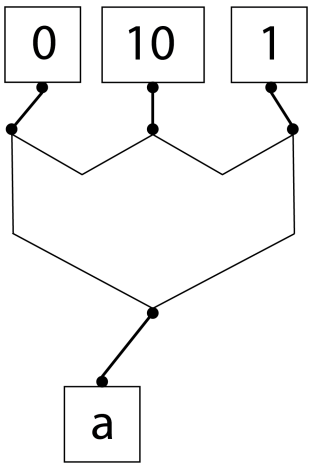
```
module shift_r(data_out, data_in, shift_by);
input [7:0] data_in;
input [7:0] shift_by;
```

```
4 output reg [7:0] data_out;
5
6 always @(*) begin
7     data_out <= data_in >> shift_by ;
8 end
9 endmodule
```

Listing 3.14: Verilog implementácia shift right

### 3.3.4 Generátor

Generátor technicky nemeníme, redukujeme vizuálne formy na tvar zloženej šípky, obr. 3.12, s externým definovaním krokov. Všetky porty musia byť zapojené. Implementácia behaviorálnym programovaním, taktom riadený výpočet.




Obr. 3.12: Generátor a možné zapojenie.

```
1 module shift_r(data_out, from, to, step, clk);
2 input clk;
3 input [7:0] from;
4 input [7:0] to;
```

```
input [7:0] step;
output reg [7:0] data_out = from;

always @(posedge clk) begin
    if (data_out < to) begin
        data_out <= data_out + step;
    end
end
endmodule
```

Listing 3.15: Verilog implementácia shift right

### 3.3.5 Špeciálne operácie

#### 3.3.5.1 UART

UART je zobrazený pomocou jednoduchého obdĺžnika s nápisom. Vývojové prostredie umožňuje jeho detailnejšie nastavenie. V predvolenom nastavení spracúva dáta v tvare jedného bezznamienkového bajtu, ihneď ako sú vysielané počítačom. Po prijatí dát sa dostane do stavu pripraveného vysielať výsledok ako jeden bezznamienkový bajt.

## 3.4 Kompilátor

Operácie, hrany, premenné nášho jazyka sú reprezentované pomocou JSONov. V reprezentácii oddeľujeme logické parametre od vizuálnych.

Logické parametre:

- typ operácie
- symbol operácie

- vstupy
- výstupy
- zapojenie vstupov
- zapojenie výstupov

Vizuálne parametre:

- typ operácie
- súradnice X a Y reprezentujúce pozíciu v ploche
- súradnice portov
- farba operácie/hrany
- stav operácie

Pri kompilácii je vytvorený hlavný modul obsahujúci zapojenie a inštancie jednotlivých operácií, skompilovaných do samostatných súborov. Hlavný modul zároveň obsahuje porty potrebné pre vstup hodinového signálu, komunikáciu a použitie d'alších hardvérových súčastí ako napr. LED diód.

## 3.5 Prostredie

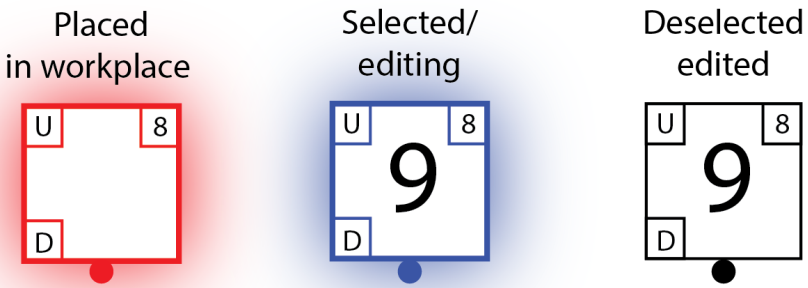
Vývojové prostredie sme sa kvôli kompatibilite rozhodli implementovať ako webovú aplikáciu. Pracovná plocha a operácie jazyka sú reprezentované pomocou JSONu. Prostredie vytvorený program ukladá a načítava ako textové súbory s príponou **.draken**. Pri kompilácii vznikajú súbory jazyka Verilog, s ktorými je možné zatiaľ manuálne d'alej pracovať v prostredí Vivado.

### 3.5.1 Menu

Menu je umiestené v hornej lište. Umožňuje vytváranie, ukladanie a načítavanie projektov v našom prostredí. Súbory sa ukladajú na disk používateľa. Toto menu obsahuje aj potrebné tlačidlá na kompiláciu programu, nastavenia prostredia a zobrazenia operácií a elementov jazyka ako napr. prepínanie medzi redukovaným a deaktivovaným detailom informácií v zobrazení operácií.

### 3.5.2 Pracovná plocha

Operácie a hrany je možné v pracovnej ploche ľubovoľne umiestňovať a editovať. Na obr. 3.13 sú tri hlavné stavy každej operácie jazyka: umiestnený v ploche, práve zvolený alebo editovaný, po dokončení úprav.




Obr. 3.13: Operácie v pracovnej ploche, rôzne stavy úprav.

### 3.5.3 Bočný panel

Obsahuje všetky operácie, ktorými jazyk disponuje, rozdelenými do kategórií pre lepšiu orientáciu. Nové operácie sa pridávajú spôsobom ťahaj a pusť z bočného panela do plochy. Po pustení je automaticky zmenený ob-

sah konfiguračného panela. Užívateľ môže už umiestnené operácie jednotlivo alebo po skupinách ľubovoľne presúvať.

### 3.5.4 Konfiguračný panel

Operáciu umiestnenú do pracovnej plochy je možné upravovať pomocou konfiguračného panela.

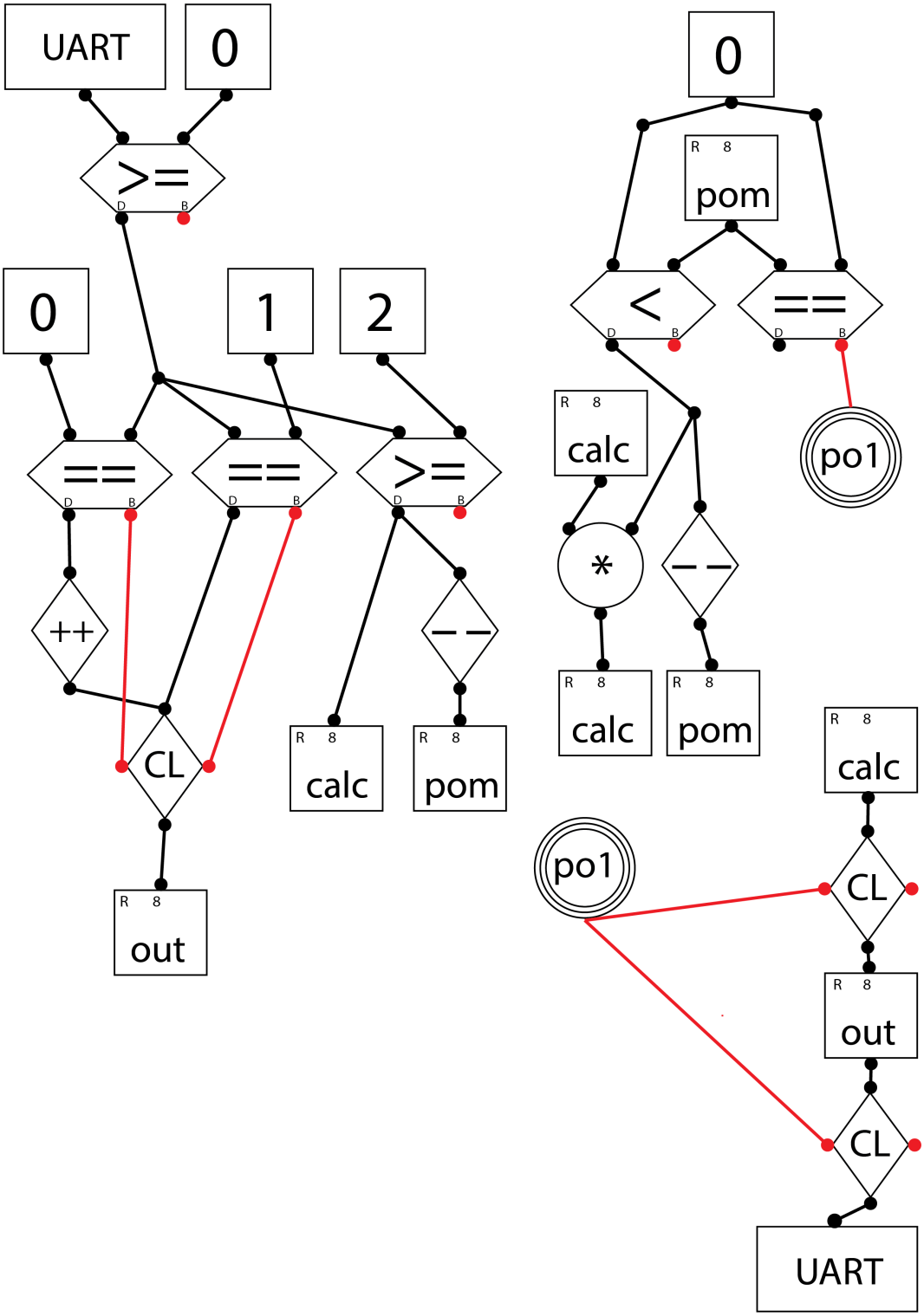
# Kapitola 4

# Výsledky

Implementáciu a operácie nášho jazyka sme priebežne testovali a ladili. Výpočet faktoriálu preveril väčšinu kategórií operácií spoločne so sekvenčným taktom riadeným výpočtom.

## 4.1 Faktoriál

Na obrázku 4.1 je zobrazený výsledný výpočet faktoriálu, ktorý je možné syntetizovať.




Obr. 4.1: Sekvenčný výpočet faktoriálu, syntetizovateľný zápis.

```
module factorial_top(clk, RsRx, RsTx, led);
input clk;
input RsRx;
output RsTx;
output [3:0] led;
reg ready;

wire [7:0] received_data;
reg [7:0] r_broadcast_data;
reg r_broadcast_allowed;

wire [7:0] sel1out;
wire [7:0] sel2out;
wire [7:0] auinc1out;
wire [7:0] sel3out;

wire [7:0] sel4out;
wire [7:0] audec1out;

wire [7:0] sel5o;
wire [7:0] audec2o;
wire [7:0] abmul1o;

wire sel1b;
wire sel2b;
wire sel3b;
wire sel4b;
wire sel5b;
wire sel6b;

reg [7:0] calc;
reg [7:0] pom;
reg [7:0] out;
```

```
34
35 UART_rx_tx_master rx_tx(.broadcast_data(r_broadcast_data),.
       broadcast_allowed(r_broadcast_allowed), .received_data(received_data),
        .RsRx(RsRx), .RsTx(RsTx), .led(led), .clk(clk));
36
37 selector_be selector1(.data_out(sel1out), .bool_out(sel1b), .lin(
       received_data), .rin(8'b00000000), .clk(clk));
38
39 selector_e selector2(.data_out(sel2out), .bool_out(sel2b), .lin(sel1out),
        .rin(8'b00000000), .clk(clk));
40
41 arithmetic_unary_inc auinc1(.dataout(auinc1out), .datain(sel2out));
42
43 selector_e selector3(.data_out(sel3out), .bool_out(sel3b), .lin(sel1out),
        .rin(8'b00000001), .clk(clk));
44
45 selector_be selector4(.data_out(sel4out), .bool_out(sel4b), .lin(sel1out)
       , .rin(8'b00000010), .clk(clk));
46
47 arithmetic_unary_dec audec1(.dataout(audec1out), .datain(sel4out));
48
49 selector_s selector5(.data_out(sel5o), .bool_out(sel5b), .lin(8'b00000000
       ), .rin(pom), .clk(clk));
50
51 arithmetic_unary_dec audec2(.dataout(audec2o), .datain(sel5o));
52
53 arithmetic_binary_mul abmul1(.data_out(abmul1o), .lin(calc), .rin(sel5o))
       ;
54
55 selector_e selector6(.data_out(), .bool_out(sel6b), .lin(pom), .rin(8'
       b00000000), .clk(clk));
56
57 always @(posedge clk) begin
```

```
58      if (sel2b == 1'b1) begin
59          out <= auinc1out;
60          ready = 1'b1; end
61
62      if (sel3b == 1'b1) begin
63          out <= sel3out;
64          ready = 1'b1;end
65
66      if (sel4b == 1'b1 && sel5b == 1'b0) begin
67          calc <= sel4out;
68          pom <= audec1out; end
69
70      if (sel4b == 1'b1 && sel5b == 1'b1) begin
71          pom <= audec2o;
72          calc <= abmul1o; end
73
74       if (sel4b == 1'b1 && sel6b == 1'b1) begin
75          out = calc;
76          r_broadcast_allowed = 1'b1;
77          r_broadcast_data = out; end
78  end
79  endmodule
```

Listing 4.1: Verilog implementation sequential factorial main modul

Uvedený faktoriál vypočíta najvyššiu hodnotu 5!, pretože pracuje s dátami o šírke 8 bitov. Pre výpočet väčších faktoriálov by bolo nutné zväčšiť veľkosť premenných a sieťových hrán na napr. 64 bitov. Zároveň je potrebné upraviť modul **UART rx tx master** aby dokázal vysielať hodnoty väčšie ako 8 bitov. To je možné vyriešiť rozdelením 64 bitového výsledku na 8 bitové hodnoty a ich následné odvysielanie za sebou.

# Kapitola 5

# Záver

Vo východiskách sme sa oboznámili s potrebnými detailami dátových tokov, HDL jazykov a platformy FPGA.

V druhej kapitole sme skúmali rôzne verzie grafického zápisu dátových tokov.

V tretej kapitole sme analyzovali možnosti implementácie vyššieho jazyka na platforme FPGA. Na základe množstva experimentov sme vybrali a navrhli vyšší jazyk dátových tokov s názvom Draken-I. Navrhujeme základy pre vývojové prostredie a kompilátor do jazyka Verilog.

Využitie a implementáciu jazyka demonštrujeme na výpočte faktoriálu, zobrazeného v štvrtej kapitole. Zdrojový popis uvedeného faktoriálu po kompilácií je v digitálnej prílohe tejto práce.

## 5.1 Možné rozšírenia

Navrhnutý jazyk je možné rozširovať viacerými smermi.

Pokračovať vo vývoji jazyka samotného, hľadať a experimentovať s homogénnou a heterogénnou duplikovanosťou. Vylepšovať existujúce

operácie a moduly ako napríklad rozšíriť generátor o generovanie znakov. Hľadať a pridávať nové operácie súvisiace s paralelizáciou a výpočtovými možnosťami, tak aj hardvérovými možnosťami platformy ako externá RAM pre ukladanie veľkého množstva dát, LAN port pre rozšírenie komunikácie.

Navrhujeme taktiež rozšíriť jazyk o interpreter prípadne kompilátor do klasickej PC architektúry. Umožnilo by to širšie možnosti použitia, vylepšilo by sa ladenie chýb a zároveň by to umožnilo vizualizáciu dát pre účely výučby dátových tokov.

# Literatúra

[Ack82] Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982.

[CGP89] Philip T. Cox, F. R. Giles, and Tomasz Pietrzykowski. Prograph: a step towards liberating programming from textual conditioning. *[Proceedings] 1989 IEEE Workshop on Visual Languages*, pages 150–156, 1989.

[Cum00] Clifford Cummings. Nonblocking assignments in verilog synthesis, coding styles that kill. 01 2000.

[Dig] Digilent. *Arty FPGA Board Reference Manual*. Digilent.

[DK82] Alan Davis and Robert Keller. Data flow program graphs. *Computer*, 15:26 – 41, 03 1982.

[DKS21] Roopa M. Druva Kumar S. Design and analysis of multiple read port techniques using bank division with xor method for multi-ported-memory on fpga platform. *International Journal of Electrical and Computer Engineering (IJECE)*, 11:4785–4793, 12 2021.

[EGRG12] Laforest Charles Eric, Liu Ming G., Rapati Emma Rae, and

Steffan J. Gregory. Multi-ported memories for fpgas via xor. 2012.

[EZT+14] Laforest Charles Eric, Li Zimo, O'rourke Tristan, Liu Ming G., and Steffan J. Gregory. Composing multi-ported memories on fpgas. *ACM Transactions on Reconfigurable Technology and Systems*, 7, sep 2014.

[HH12] David Harris and Sarah Harris. *Digital Design and Computer Architecture, Second Edition*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2012.

[Hil92] Daniel D Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69–101, 1992.

[JdlC19] Richard Jennings and Fabiola de la Cueva. *LabVIEW graphical programming*. McGraw-Hill Education, 2019.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 3 2004.

[Kul18] Peter Kuljovský. Dátové toky ako paradigma pre paralelné programovanie. Master's thesis, Univerzita Komenského v Bratislave, Fakulta matematiky, fyziky a informatiky, 2018.

[LaM19] Brock LaMeres. *Quick Start Guide to Verilog*. 01 2019.

[LL16] Bo-Cheng Lai and Jiun-Liang Lin. Efficient designs of multiported memory on fpga. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25:1–12, 06 2016.

[MK11] Maija Marttila-Kontio. *Visual data flow programming languages: challenges and opportunities*. PhD thesis, Väitöskirja :, Kuopio, 2011. Artikkeliväitöskirjan yhteenveto-osa ja 8 eripainosta.

[NE18] Muthumanickam T. Navagiri E., Sheela T. Design of lvt based multiported memory in fpga. *International Journal of Pure and Applied Mathematics*, 119(14):121–125, 2018.

[oEE16] School of Electrical and Computer Engineering. *Tutorial 4: Verilog Hardware Description Language*, May 5, 2016.

[Pal03] Samir Palnitkar. *Verilog Hdl: A Guide to Digital Design and Synthesis, Second Edition*. Prentice Hall Press, 2 edition, 2003.

[PH13] David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[Raj17] J. Rajewski. *Learning FPGAs: Digital Design for Beginners with Mojo and Lucid HDL*. O'Reilly, 2017.

[RGHJ15] Townsend Kevin R., Attia Osama G., Jones Phillip H., and Zambreno Joseph. A scalable unsegmented multiport memory for fpga-based systems. *International Journal of Reconfigurable Computing vol. 2015*, 2015, 2015.

[SM14] Stuart Sutherland and Don Mills. Dvcon-2014. In *Can My Synthesis Compiler Do That?*, page 24, 2014. What ASIC

and FPGA Synthesis Compilers Support in the SystemVerilog-2012 Standard.

[Smi98] Douglas J. Smith. *Hdl Chip Design: A Practical Guide for Designing, Synthesizing & Simulating Asics & Fpgas Using Vhdl or Verilog*. Doone Pubns, 1998.

[Sut01] Stuart Sutherland. *Quick Start Guide to Verilog*. Sutherland HDL, Inc., 2001.

[Tan13] Steven L. Tanimoto. A perspective on the evolution of live programming. LIVE '13, page 31–34. IEEE Press, 2013.

[TM08] D. E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Springer, 5 edition, 2008.

[UT17] C. Unsalan and B. Tar. *Digital System Design with FPGA: Implementation Using Verilog and VHDL*. McGraw-Hill Education, 2017.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, dec 1986.

[WA85] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. A.P.I.C. studies in data processing. Academic Press, 1985.

[Wil88] Timothy J. Wilson. A visual programming language for data flow systems. Master's thesis, Rochester Institute of Technology, October 14, 1988. Accessed: 2022-3-4.

[WP94] P.G. Whiting and R.S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, 1994.

[Xila] Xilinks. *Block Memory Generator*. Xilinks.

[Xilb] Xilinks. *Distributed Memory Generator*. Xilinks.

[Xilc] Xilinks. *Vivado Design Suite Tcl Command Reference Guide*. Xilinks.

# Prílohy

# Digitálna príloha

Priložené CD k tejto práci obsahuje simulovateľný a syntetizovateľný zdrojový popis faktoriálu uvedeného v kapitole výsledky. Obsahom sú aj experimentálne popisy hardvéru, ktoré vznikali pri návrhu jazyka, overovaní vytvoriteľnosti operácií a následne aj návrhu implementácie. Prikladáme aj experimentálne implementácie faktoriálu, ktoré slúžili pri hľadaní riešení synchronizácií operácií a etáp.

# Názov jazyka

Počas vývoja sme jazyk označovali skratkou názvu témy - VPJDT.

Nad názvom jazyka sme príležitostne premýšľali od samého začiatku, chceli sme niečo výstižné.

Názov prešiel viacerými zmenami. Autor práce obľubuje programovací jazyk Python. Tu sa objavuje jeden z ďalších zdrojov inšpirácie - zvieracia ríša. Pri premýšľaní sme identifikovali mnohé vlastnosti tejto témy ako aj neobyčajne agresívnu tendenciu rásť v oblastiach robustnosti a zložitosti.

Spojením s paralelizáciou, dátovými tokmi a charakteristikou práce sa nám v mysli zjavila bájna príšera, Cthulhu od spisovateľa H. P. Lovecrafta. Cthulhu je vhodne desivý, veľký, nebezpečný. Jeho meno je však komplikované, nehovoriac o výslovnosti. Je tu však ďalšia príšera s chápadlami.

Kraken, obrovský hlavonožec, sťahujúci na morské dno celé lode. Z jeho mena berieme základ názvu a vytvárame naň odkaz. K meníme za D ako odkaz na dáta z DataFlow a na D-FlipFlop ako základnú stavebnú jednotku FPGA. Vzniká Draken.

Chápadlá reprezentujú paralelizáciu, ich paralelné ovládanie výpočty. Náš jazyk je zároveň vizuálny. I v názve znamená ink - atrament a získavame celý názov **Draken-I**. Atrament už veľmi dlho slúži ľudstvu na

zobrazovanie - v našom prípade na vizualizáciu dát. Hlavonožce v nebezpečenstve, pri úniku vypúšťajú látku zvanú atrament. Zmiznú a ostane po nich atramentový mrak. Týmto odkazujeme na to, čo ostane po tom ako všetky výpočty skončia - dáta. Znak I je preto v názve oddelený pomlčkou.