VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

HLEDÁNÍ REGULÁRNÍCH VÝRAZŮ S VYUŽITÍM TECHNOLOGIE FPGA

FAST REGULAR EXPRESSION MATCHING USING FPGA

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JURAJ KUBIŠ**
AUTHOR

VEDOUCÍ PRÁCE **Ing. DENIS MATOUŠEK**
SUPERVISOR

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Kubiš Juraj**
Obor: Informační technologie
Téma: **Hledání regulárních výrazů s využitím technologie FPGA**
Fast Regular Expression Matching Using FPGA
Kategorie: Počítačové sítě

Pokyny:

1. Nastudujte možnosti hardwarové akcelerace hledání regulárních výrazů s využitím technologie FPGA.
2. Analyzujte možnosti optimalizace automatů s cílem zrychlit proces vyhledávání a minimalizovat hardwarové zdroje potřebné pro realizaci operace vyhledávání.
3. Navrhněte a implementujte programové vybavení, které převede zadanou množinu regulárních výrazů do podoby automatů a provede optimalizace pro zajištění multigigabitové propustnosti na technologii FPGA. Soustřeďte se na úsporu paměťových prostředků na čipu.
4. Vytvořenou implementaci ověřte nad množinami regulárních výrazů používaných v různých síťových aplikacích.
5. V závěru diskutujte vlastnosti navrženého řešení a možnosti použití v různých síťových aplikacích.

Literatura:

- Fast and Scalable Pattern Matching for Network Intrusion Detection Systems, by Sarang Dharmapurikar and John W. Lockwood, IEEE Journal on Selected Areas in Communications, (JSAC) Oct. 2006, Volume: 24, Issue: 10, pp. 1781- 1792
- Podle pokynů vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Splnění bodů 1 a 2 zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Matoušek Denis, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt

Bakalárska práca sa zaoberá možnosťami hardvérovej akcelerácie vyhľadávania regulárnych výrazov. Obsahom práce je analýza už existujúcich hardvérových architektúr a zhodnotenie ich pozitívnych a negatívnych vlastností. Na základe týchto poznatkov je navrhnutá architektúra. Tá je založená na deterministických konečných automatoch s implicitnými prechodmi (D^2FA), je implementovaná v jazyku VHDL a je vykonaná jej syntéza. Výsledky syntézy sú analyzované za účelom zistenia celkovej priepustnosti architektúry. Je navrhnuté programové vybavenie na prevod regulárnych výrazov do podoby D^2FA a na optimalizovanie tohoto automatu s cieľom minimalizovania pamäťových nárokov. Implementácia je overená a je zhodnotený prínos jednotlivých optimalizačných techník na redukciu pamäťových nárokov.

Abstract

Bachelor thesis deals with the possibility of hardware acceleration of regular expression matches. The content of the thesis is to analyze existing hardware architectures and evaluate their positive and negative properties. Based on this knowledge, the architecture is designed. It is based on deterministic finite automata with implicit transitions (D^2FA), is implemented in VHDL and is synthesized. The synthesis results are analyzed to determine the overall throughput of the architecture. It is designed software to convert regular expressions into a D^2FA and to optimize this automaton in order to minimize memory requirements. The implementation is verified and the benefits of individual optimization techniques to reduce memory requirements are evaluated.

Kľúčové slová

Regulárne výrazy, konečné automaty, konečné automaty s implicitnými prechodmi, FPGA, FSM, NFA, DFA, D2FA

Keywords

Regular expressions, finite automata, finite automata with default transitions, FPGA, FSM, NFA, DFA, D2FA

Citácia

KUBIŠ, Juraj. *Hledání regulárních výrazů s využitím technologie FPGA*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Denis Matoušek

Hledání regulárních výrazů s využitím technologie FPGA

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Denise Matouška. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....
Juraj Kubiš
17. mája 2018

Poďakovanie

Ďakujem pánovi Ing. Denisovi Matouškovi za poskytnuté konzultácie a rady pri vypracovávaní tejto práce.

Obsah

Kapitola 1

Úvod

Regulárne výrazy nachádzajú uplatnenie v mnohých aplikáciách a jedným z kľúčových odvetví ich uplatnenia je aj oblasť počítačových sietí. Tu sú využívané napríklad na monitorovanie prevádzky v sieti a zaistenie jej bezpečnosti, nakoľko otázka bezpečnosti sietí nebola nikdy tak aktuálna, ako je dnes, keďže internet, čoby najväčšia počítačová sieť na svete, zasahuje do každého odvetia ľudskej činnosti, alebo sú využívané v aplikáciách pracujúcich na aplikačnej vrstve, ako je *load balancing na HTTP* [17].

Z tohoto dôvodu je kladený veľký dôraz na vývoj technológií na monitorovanie sietí a na detekciu hrozieb a útokov. Veľká časť týchto technológií je založená na vyhľadávaní určitých vzorov a podobností, ktoré sú charakteristické práve pre uvedenú nebezpečnú komunikáciu a jedným z prostriedkov pre definovanie takýchto vzorov sú práve *regulárne výrazy*. V prípade zaisťovania bezpečnosti počítačových sietí majú regulárne výrazy naozaj kľúčové postavenie, keďže ich použitie má oveľa vyššiu mieru úspešnosti detekcie bezpečnostných incidentov [7].

Prenosová kapacita počítačových sietí má vzrastajúcu tendenciu, o čom napríklad svedčia novo prijaté štandardy technológie *Ethernet*[1], a spolu s ňou súvisí aj nárast objemu prenášaných dát. Táto skutočnosť vyžaduje neustále zrýchľovanie vyhľadávania vzorov s cieľom včas detegovať potencionálnu hrozbu.

V dnešnej dobe už nie sú ničím nezvyčajné siete, ktorých priepustnosť dosahuje rádovo stovky Gbit/s. Vyhľadávanie vzorov na sieťach s takouto priepustnosťou však naráža na výkonnostné limity bežných procesorov a jediná cesta k ďalšiemu zrýchľovaniu spracovania dátového toku je jeho paralelizácia.

Riešením je akcelerácia vyhľadávania pomocou špeciálneho hardware, ktorý dosahuje oveľa lepšie výsledky najmä vďaka masívnej paralelizácii. Nevýhoda takéhoto hardware zase spočíva v jeho neprispôsobivosti. Toto sa však netýka čipov *FPGA*, ktoré stále poskytujú vysoký výkon, ale zároveň umožňujú pružne reagovať na aktuálnu situáciu a meniť dizajn hardvérového obvodu podľa aktuálnej potreby.

Obsahom tejto práce preto je navrhnutie hardvérovej architektúry, ktorá dokáže zaistiť detegovanie vzorov v dátovom toku, s možnosťou mieniť vyhľadávanú sadu regulárnych výrazov bez rekonfigurácie čipu.

Druhým výsledkom práce bude implementácia programového vybavenia, ktoré dokáže transformovať sadu regulárnych výrazov do podoby konečných automatov a optimalizovať ich s cieľom minimalizovať pamäťové nároky architektúry.

Text práce je členený na niekoľko kapitol, ktoré predstavujú ucelené etapy riešenia problému a ktorých poradie a náväznosť kopíruje postup práce na riešenej problematike.

[1] Prehľad jednotlivých štandardov je dostupný na: http://www.ieee802.org/3/

Kapitoly 2 a 3 ozrejmujú čitateľovi základné výrazy a termíny a poskytujú teoretické informácie potrebné pre pochopenie tejto práce. Kapitola 4 analyzuje existujúce riešenia riešenej problematiky spolu s ich výhodami a nevýhodami. Obsahom kapitoly 5 je popis navrhnutej architektúry, ozrejmenie princípu jej činnosti a popis jej VHDL implementácie. V kapitole 6 je potom opísané vytvorené programové vybavenie a vysvetlený princíp činnosti jednotlivých optimalizačných techník. Náplňou kapitoly 7 je sumarizácia a prezentovanie dosiahnutých výsledkov, konkrétne hardvérové nároky navrhnutej architektúry, jej teoretická priepustnosť a prínos jednotlivých optimalizačných techník k redukcii počtu prechodov automatu. Kapitola 8 predstavuje záver celej práce, v ktorom sú rekapitulované dosiahnuté výsledky.

Kapitola 2

Regulárne jazyky

Táto kapitola zhrňuje terminológiu používanú v ďalších častiach práce. Definuje predovšetkým pojmy *regulárny výraz*, *konečný automat*, ako aj ďalšie, menej obecné termíny.

2.1 Základné pojmy a definície

Všetky definície v tejto sekcii sú prebrané z knihy *Automata and Languages* [5] od *prof. Meduny*.

Definícia 2.1.1 (Abeceda a symbol)**.** *Abeceda* je neprázdna konečná množina prvkov, nazývaných *symboly*.

Definícia 2.1.2 (Reťazec)**.** Nech Σ je abeceda.

- ϵ je *reťazec* nad abecedou Σ.
- Ak x je *reťazec* nad abecedou Σ a $a \in \Sigma$, potom xa je *reťazec* nad abecedou Σ.

Definícia 2.1.3 (Konkatenácia reťazcov)**.** Nech x a y sú dva reťazce nad abecedou Σ. Potom xy je *konkatenácia reťazcov* x a y.

Pre každý reťazec x,

$$x\epsilon = \epsilon x = x$$

.

Definícia 2.1.4 (Jazyk)**.** Nech Σ je abeceda, Σ^* množina všetkých reťazcov nad abecedou Σ a $L \subseteq \Sigma^*$. Potom L je *jazyk* nad abecedou Σ.

Definícia 2.1.5 (Konkatenácia jazykov)**.** Nech L_1 a L_1 sú dva jazyky. Potom L_1L_2 je *konkatenácia jazykov* L_1 a L_2 definovaná ako

$$L_1L_2 = \{xy : x \in L_1 \text{ a } y \in L_2\}$$

.

Definícia 2.1.6 (Mocnina jazyka)**.** Nech L je jazyk. Pre $i \geqslant 1$, *i-ta mocnina jazyka*, L^i, je definovaná ako

- $L^0 = \epsilon$
- pre všetky $i \geqslant 1$, $L^i = LL^{i-1}$.

Definícia 2.1.7 (Uzáver jazyka)**.** Nech L je jazyk. *Uzáver* L, L^*, je definovaný ako

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

.

Okrem hore uvedených operácií, môžeme nad jazykmi vykonávať aj všetky bežné množinové operácie, ako je *zjednotenie*, *prienik*, *rozdiel* a pod.

2.2 Regulárne výrazy

Zjednodušene sa dá povedať, že *regulárny výraz* je sekvencia znakov, ktoré definujú určitý vyhľadávací vzor. Každému *regulárnemu výrazu* teda odpovedá množina reťazcov (jazyk), ktoré danému vyhľadávaciemu vzoru vyhovujú.

Definícia 2.2.1 (Regulárny výraz)**.** Nech Σ je abeceda. *Regulárne výrazy* nad Σ a jazyky generované týmito výrazmi sú definované rekurzívne nasledovne:

- $\varnothing$ je *regulárny výraz* generujúci prázdnu množinu.
- ϵ je *regulárny výraz* generujúci $\{\epsilon\}$.
- a, kde $a \in \Sigma$ je *regulárny výraz* generujúci $\{a\}$.
- Ak r a s sú *regulárne výrazy* generujúce jazyky R a S, potom:
  - $(r.s)$ je *regulárny výraz* generujúci RS.
  - $(r + s)$ je *regulárny výraz* generujúci $R \cup S$.
  - (r^*) je *regulárny výraz* generujúci R^*.

Všetky jazyky, ktoré sú generovateľné pomocou regulárnych výrazov, patria do kategórie, ktorú nazývame *regulárne jazyky*.

2.3 Konečné automaty

Konečné automaty sú ďalším prostriedkom pre definíciu regulárnych jazykov. Táto ekvivalencia regulárnych výrazov a konečných automatov je známa ako *Kleeneho teorém* [8], ktorého autorom je americký matematik *Stephen Cole Kleene*.

Definícia 2.3.1 (Nedeterministický konečný automat)**.** *Nedeterministický konečný automat* (NFA) je pätica:

$$(Q, \Sigma, \delta, s, F)$$

kde

- Q je konečná množina stavov
- Σ je vstupná abeceda taká, že $\Sigma \cap Q = \varnothing$
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ je prechodové zobrazenie (funkcia)
- $s \in Q$ je počiatočný stav
- $F \subseteq Q$ je množina koncových stavov

2.3.1 Princíp činnosti

Činnosť konečného automatu je v princípe veľmi jednoduchá. Automat postupne číta symboly z reťazca na vstupe a snaží sa vykonať *„prechod“* z aktuálneho stavu do stavu nasledujúceho. Nasledujúci stav je určený aktuálnym stavom a vstupným symbolom. Toto sú dva parametre prechodovej funkcie a prípade, že je pre túto dvojicu prechodová funkcia definovaná, jej výsledok určuje stavy nasledujúce (v opačnom prípade sa automat ocitne v tzv. *„zaseknutom“* stave).

Pri vykonaní prechodu sa nasledujúci stav stáva stavom aktívnym a na vstupe automatu sa nachádza ďalší symbol z reťazca (v prípade, že sa nejedná o tzv. *epsilon prechod*, viz ďalej). Ak aktívny stav patrí do množiny koncových stavov, reťazec, ktorý bol po jednotlivých symboloch prečítaný automatom označujeme ako *reťazec prijímaný automatom* a tento reťazec patrí do jazyka definovaného týmto automatom.

2.3.2 Nedeterministický a deterministický konečný automat

Deterministický konečný automat (DFA) je špeciálnym prípadom NFA, od ktorého sa odlišuje len dvoma obmedzeniami. Zobrazenie δ priraďuje, v prípade NFA, niektorým usporiadaným dvojiciam z množiny $Q \times (\Sigma \cup \{\epsilon\})$ ľubovoľnú podmnožinu množiny stavov Q.

U DFA sú všetky podmnožiny priraďované zobrazením jednoprvkové – zobrazenie teda priraďuje iba jeden konkrétny stav patriaci do Q.

Druhým obmedzením je, že automat neobsahuje tzv. *epsilon prechody*. To znamená, že hore spomenuté usporiadané dvojice neobsahujú ϵ.

Definícia *prechodového zobrazenia* DFA je teda:

$$\delta : Q \bigtimes \Sigma \to Q$$

2.3.3 Determinizácia konečného automatu

Michael O. Rabin a Dana Scott vo svojom článku [6] dokázali, že ľubovoľný NFA je možné previesť na ekvivalentný DFA (ekvivalencia v zmysle, že oba automaty definujú (prijímajú) rovnaký regulárny jazyk).

Toto zistenie je dôležité najmä z dôvodu, že z praktického pohľadu má DFA oveľa lepšie vlastnosti:

- Maximálny počet aktívnych stavov DFA v ľubovoľnom okamihu je jedna, nakoľko prechodové zobrazenie DFA priraďuje iba vždy iba jeden stav.
- Maximálny počet vykonaných prechodov DFA potrebných na rozhodnutie, či vstupný reťazec patrí do jazyka prijímaného automatom, je rovný dĺžke tohto reťazca. To je zaručené tým, že DFA neobsahuje epsilon prechody a pri každom prechode je vždy prečítaný jeden symbol zo vstupného reťazca.

Okrem hore spomenutých výhod však determinizácia spôsobuje aj jednu závažnú nevýhodu a tou je nárast počtu stavov automatu. Počet stavov DFA, ktorý je ekvivalentný NFA s počtom stavov n, totiž môže byť až 2^n [10].

2.3.4 Minimalizácia konečného automatu

Minimalizácia konečného automatu je jeho transformácia na ekvivalentný konečný automat, ktorý má minimálny počet stavov. Takýto automat potom označujeme ako *minimálny*.

Základným princípom minimalizácie je odstránenie stavov, ktoré sú:

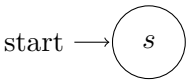
- *nedostupné* tzn., že neexistuje žiadny vstupný reťazec, ktorý by dostal automat do daného stavu zo stavu počiatočného
- *neukončujúce* tzn., že neexistuje žiadny vstupný reťazec, ktorý by dostal automat z daného stavu do stavu koncového

Existuje viacero algoritmov na minimalizáciu, všetky majú ale jedno spoločné obmedzenie a tým je, že je možné ich aplikovať iba na DFA [2].

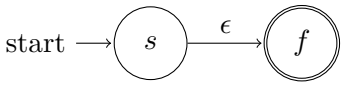
2.4 Transformácia regulárneho výrazu na konečný automat

Ako už bolo povedané, regulárne výrazy ako aj konečné automaty sú dva základné modely pre definíciu regulárnych jazykov. Z tohto faktu vyplýva, že každý regulárny výraz je možné transformovať na ekvivalentný konečný automat (ekvivalencia v zmysle, že oba definujú rovnaký jazyk) a jedným zo spôsobov tejto transformácie je algoritmus *Thompsonova konštrukcia* [12].

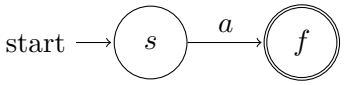
Základný princíp algoritmu je založený na rekurzívnom rozdeľovaní komplexného regulárneho výrazu na vstupe na jednotlivé základné regulárne výrazy. Pre každý takýto výraz potom skonštruuje ekvivalentný konečný automat podľa nasledovných pravidiel (viz obrázky 2.1, 2.2 a 2.3).



Obr. 2.1: Konečný automat ekvivalentný regulárnemu výrazu $\varnothing$.

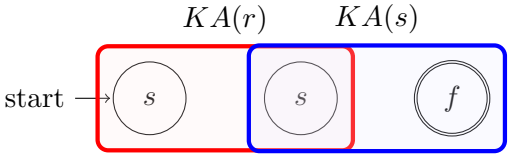


Obr. 2.2: Konečný automat ekvivalentný regulárnemu výrazu ϵ.

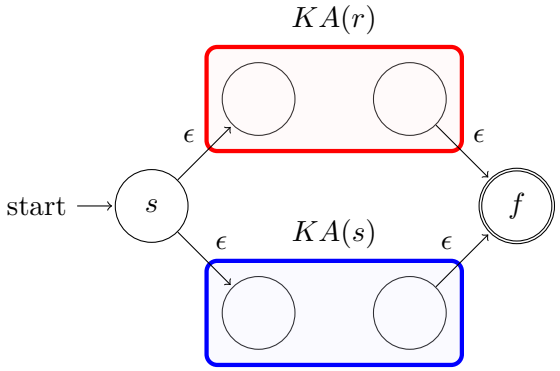


Obr. 2.3: Konečný automat ekvivalentný regulárnemu výrazu a.

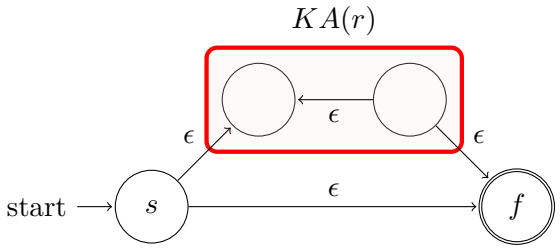
Následne sú tieto „malé“ konečné automaty opätovne spájané do väčších automatov podľa nasledovných pravidiel ($KA(r)$ a $KA(s)$ sú konečné automaty odpovedajúce výrazom r a s) (viz obrazky 2.4, 2.5 a 2.6).



Obr. 2.4: Konečný automat ekvivalentný regulárnemu výrazu $(r.s)$.



Obr. 2.5: Konečný automat ekvivalentný regulárnemu výrazu $(r + s)$.



Obr. 2.6: Konečný automat ekvivalentný regulárnemu výrazu (r^*).

Výsledkom Thompsonovej konštrukcie je NFA ekvivalentný regulárnemu výrazu na vstupe algoritmu. Automat v tejto podobe je možné priamo implementovať, jeho vlastnosti sú ale väčšinou na efektívne implementovanie nevhodné a preto ďalej býva takýto automat determinizovaný a minimalizovaný.

2.5 Zhrnutie

V tejto kapitole sme si predstavili dva základné modely regulárnych jazykov - regulárne výrazy a konečné automaty. Každý z modelov má svoje výhody a nevýhody, ale v praxi sú používané oba.

Výhoda konečných automatov spočíva v ich jednoduchej implementácii či už vo softvéri, alebo v hardvéri, ich nevýhodou zase je, že sú pre človeka neprehľadné. Pri regulárnych výrazoch je to presne naopak. Vyznačujú sa totiž oveľa vyššou čitateľnosťou, no ich praktická implementácia je náročná.

Kapitola 3

Programovateľné hradlové polia

Programovateľné hradlové pole (anglicky *Field-programmable gate array*, FPGA) je špeciálny typ integrovaného číslicového obvodu, ktorého funkcionalitu je možné programovo meniť. Tento názov je zložený z dvoch pojmov:

field-programmable – v preklade *programovateľné v poli*, čo znamená, že zariadenie je prispôsobené na zmenu svojej vnútornej konfigurácie bez nutnosti jeho demontáže, alebo jeho zaslania výrobcovi [13],

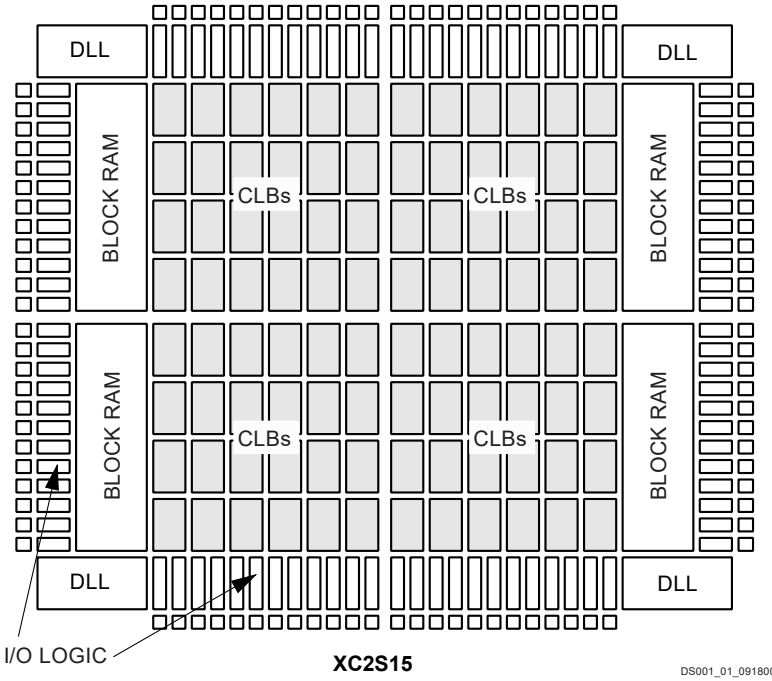
gate array – predstavuje prefabrikovaný ASIC (z angl. *Application Specific Integrated Circuit*) čip, ktorého komponenty (logické hradlá, preklápacie obvody a pod.) sú prepojené až dodatočne, podľa konkrétnej potreby a účelu, aký má takéto zariadenie plniť [14].

Programovateľné hradlové pole je teda také hradlové pole, ktoré je uspôsobené na jednoduchú a opakovanú zmenu vnútorného prepojenia jeho komponent, tzv. *rekonfiguráciu*.

FPGA kombinuje výhody procesorov, ktorou je možnosť programovo meniť funkcionalitu univerzálneho čipu a ASIC obvodov, ktoré vynikajú hlavne výhodným pomerom medzi výpočtovým výkonom a elektrickým príkonom čipu.

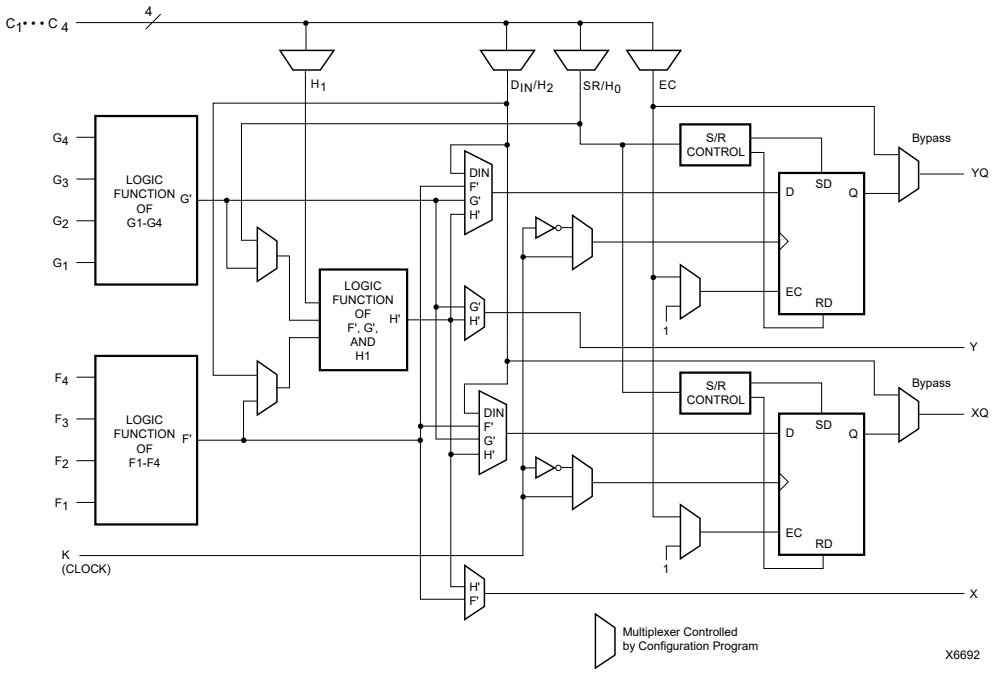
3.1 Architektúra FPGA

Obrázok 3.1 znázorňuje základnú architektúru FPGA čipu, v ktorej možno rozoznať základné stavebné bloky. Sú to konfigurovateľné logické bloky (anglicky *Configurable logic block*, CLB), konfigurovateľné vstupno-výstupné bloky (I/O block) a programovateľné prepojenie týchto blokov. Ďalej môžu byť na čipe prítomné pamäťové bloky a ďalšie podporné obvody [18].



Obr. 3.1: Bloková schéma FPGA z rodiny Spartan-II od firmy Xilinx. Prevzaté z [16].

Bloky *CLB* tvoria základný stavebný kameň celého čipu a obsahujú programovateľnú logiku FPGA [18].



Obr. 3.2: Zjednodušená bloková schéma CLB Xilinx série XC4000. Prevzaté z [15].

Na obrázku 3.2 je znázornená základná štruktúra CLB blokov, ktoré sú tvorené:

lookup tabuľkami (LUT) – využívané ako generátory ľubovoľných n-bitových logických funkcií, prípadne ako distribuovaná pamäť,

klopnými obvodmi (FF) – využívané ako registre uchovávajúce jednobitovú informáciu,

multiplexormi (MX) – umožňujú meniť vzájomné prepojenie LUT a FF v CLB.

Typické FPGA obsahuje rádovo stovky až tisíce takýchto blokov a v niektorých architektúrach sú CLB ďalej delené na identické bloky, nazývané *slice*. Tieto bloky sú prepojené pomocou programovateľných spojov a umožňujú tak vytvárať komplexnejšie logické obvody.

Táto prepojovacia sieť nespája len CLB, ale aj ostatné súčasti FPGA, ako sú vstupno-/výstupné bloky a blokové pamäte typu RAM (BRAM) [16].

Vzájomné rekonfigurovateľné prepojenie BRAM dovoľuje vytvoriť väčšie pamäte a taktiež umožňuje dosiahnuť požadovanú bitovú šírku adresy a dát.

3.2 Jazyk VHDL

Jazyk *VHDL* patrí do triedy programovacích jazykov označovaných ako *jazyky pre popis hardware* (Hardware Description Language, HDL). Vznikol v 80-tych rokoch a hlavným podnetom pre jeho vznik bola narastajúca zložitosť obvodov. Tie bolo čoraz viac problematické popisovať pomocou obyčajných elektronických schém a preto vznika potreba vymyslieť jednoduchší spôsob popisu elektronických obvodov [4].

Jazyk umožňuje opisovať obvody (komponenty) dvomi základnými metódami:

- *Štrukturálny popis* – pri použití tejto metódy je komponenta definovaná ako vzájomné prepojenie logických hradiel, alebo iných, už definovaných komponent.
- *Behaviorálny popis* – komponenta je definovaná pomocou svojho správania, v zmysle závislosti výstupov na vstupoch alebo pomocou sekvencie príkazov.

Kapitola 4

Existujúce architektúry pre vyhľadávanie regulárnych výrazov

Náplňou tejto kapitoly je predstavenie si základných architektúr využívaných pre vyhľadávanie regulárnych výrazov, priblíženie princípu ich fungovania a zhodnotenie ich pozitívnych a negatívnych vlastností.

Hlavnými hodnotiacimi kritériami jednotlivých implementácií bude časová a priestorová (pamäťová) zložitosť. Časová zložitosť je závislosť, medzi dĺžkou slova na vstupe a počtom krokov výpočtu (počet vykonaných prechodov) potrebných pre jeho celé spracovanie a priestorová zložitosť predstavuje závislosť medzi počtom a dĺžkou regulárnych výrazov a počtom stavov a pravidiel automatu, resp. množstvom zdrojov, ktoré daná architektúra na FPGA zaberá.

4.1 Nedeterministické konečné automaty

Nedeterministické konečné automaty (NFA) nie sú, z pohľadu konvenčných procesorov, príliš vhodné na implementáciu. Hlavnou príčinou je, že NFA môže súčasne obsahovať viacero aktívnych stavov, z ktorých je obecne možné vykonať viacero prechodov. Na procesore sa všetky tieto paralelné prechody musia spracovávať sekvenčne a keďže počet paralelných prechodov je s každým prečítaným symbolom iný, je pri takejto implementácii problematické garantovať minimálnu priepustnosť.

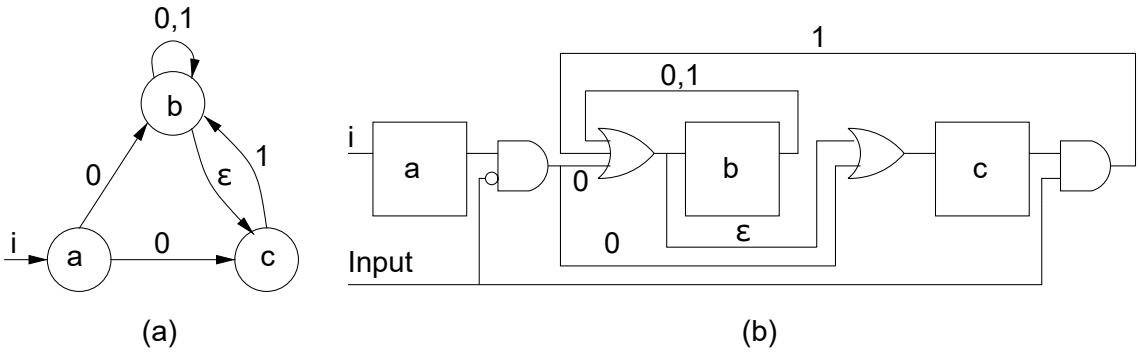
Pri FPGA je však situácia iná, vnútorná štruktúra čipu totiž dovoľuje paralelnú implementáciu všetkých stavov priamo v logike. Tým je zaručené, že spracovanie jedného symbolu na vstupe trvá pevný počet krokov, čo predstavuje odstránenie hlavnej nevýhody NFA. Fakt, že NFA nie je nutné determinizovať a nehrozí exponenciálny nárast počtu stavov (viz sekcia 2.3.3), umožňuje na čipe implementovať rozsiahlejšie a komplexnejšie automaty.

Architektúra od Sindhu a Prasanna, predstavená v ich článku [9], je príkladom architektúry založenej na NFA zostrojenom pomocou Thompsonovej konštrukcie. Automat ďalej nie je nijak upravovaný a je priamo namapovaný do logiky FPGA.

Obrázok 4.1 znázorňuje hardvérovú reprezentáciu jednoduchého NFA. Každý stav je reprezentovaný ako jednobitový register, ktorého hodnota určuje, či je daný stav aktívny, alebo nie. Keďže každý stav je reprezentovaný ako samostatný register, automat môže mať viacero aktívnych stavov súčasne.

Prechodová funkcia je riešená pomocou kombinačnej logiky. Každý stav má komparátor, ktorý porovnáva vstupný symbol so symbolom prechodu. V prípade epsilon prechodov nie

je potrebná žiadna ďalšia logika, postačuje prepojenie výstupu a vstupu registrov prislúchajúcich k stavom, medzi ktorými vedie epsilon prechod.

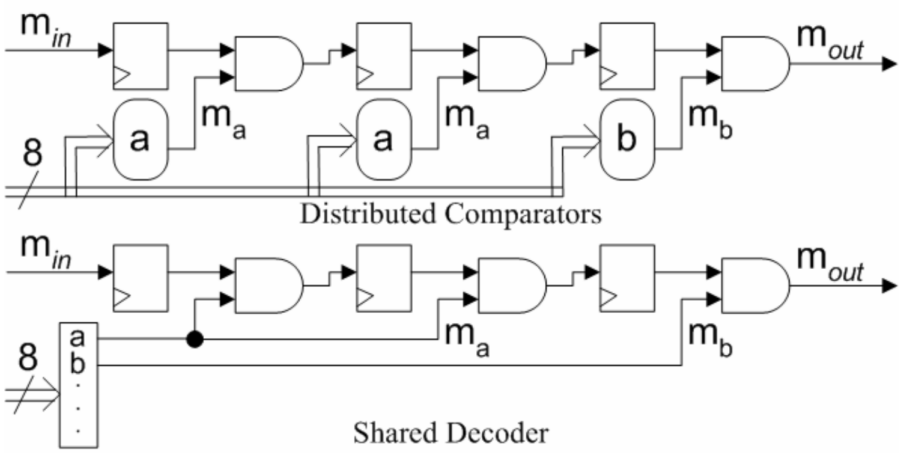


Obr. 4.1: (a) Jednoduchý NFA. (b) Implementácia v logike. Prevzaté z [9].

Bohužiaľ, použitie tejto architektúry pre rozsiahlejšie automaty komplikuje jedna vec. Každému stavu odpovedá jeden komparátor, ktoré pri väčšom počte stavov zaberajú výraznú časť zdrojov FPGA.

4.2 Nedeterministické konečné automaty so zdieľaným dekodérom

Clark vo svojom článku [1] navrhol malé, ale veľmi výrazné vylepšenie predchádzajúcej architektúry. Na miesto separátneho komparátora vstupného symbolu pre každý stav, navrhol použitie zdieľaného dekodéra vstupného symbolu na 1 z n. Následne sú tieto jednobitové signály vedené do kombinačnej logiky jednotlivých stavov, kde nahradia jednobitové výstupy nahradených komparátorov. Príklad takejto reprezentácie NFA je možné vidieť na obrázku 4.2.



Obr. 4.2: NFA so zdieľaným dekodérom. Prevzaté z [1].

4.3 Deterministické konečné automaty

Ako už bolo spomenuté v sekcii 2.3.3, jedna z výhod DFA spočíva v tom, že maximálny počet aktívnych stavov je jedna a teda aj maximálny počet možných prechodov, po prečítaní jedného vstupného symbolu, je jedna. Architektúry DFA túto vlastnosť využívajú spôsobom, že namiesto reprezentovania prechodov pomocou kombinačnej logiky, je prechodová tabuľka uložená v pamäti. Adresa do tejto pamäte je vypočítaná na základe čísla aktívneho stavu uloženého v pomocnom registri a symbolu na vstupe. Záznam na tejto adrese reprezentuje číslo nasledujúceho stavu, ktoré sa po vykonaní prechodu uloží do pomocného registru a nahradí pôvodnú hodnotu.

Architektúra má teda vysoké nároky na pamäťové zdroje FPGA, ale zdroje, ktoré sú potrebné pre logiku automatu, s počtom stavov automatu nenarastajú (za predpokladu, že nie je nutné meniť bitovú šírku čísiel stavov).

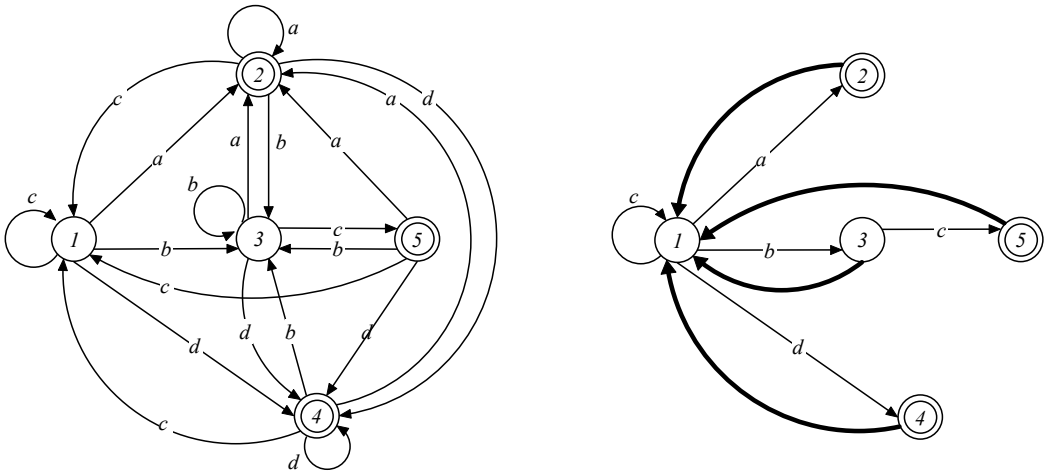
Nevýhody spojené s touto architektúrou vyplývajú hlavne z faktu, že počet stavov automatu môže pri jeho determinizácii narásť exponenciálne, čo predstavuje pri FPGA, ktoré majú obmedzené pamäťové zdroje, výrazný problém. Okrem toho, aj samotný proces determinizácie má exponenciálnu pamäťovú zložitosť a preto je determinizácia rozsiahlejších automatov veľmi problematická.

4.4 Deterministické konečné automaty s implicitnými prechodmi

Nevýhodami, ktoré boli spomenuté v predchádzajúcej kapitole, teda exponenciálnym nárastom počtu stavov, sa vo svojej práci [3] zaoberal aj Kumar. Ten zavádza nový typ deterministického konečného automatu, ktorý je oproti klasickému DFA rozšírený o tzv. *implicitné prechody*. Automat vykoná implicitný prechod iba v prípade, že pre daný aktuálny stav a vstupný symbol neexistuje normálny prechod.

Pri vytváraní tohto modelu vychádzal zo zistenia, že DFA obsahuje viacero prechodov, ktoré vedú do rovnakého stavu a sú podmienené prečítaním rovnakého vstupného symbolu. Navrhol teda, že zo všetkých takýchto prechodov ponechá vždy iba jeden a ostatné nahradí prechodmi implicitnými, ktoré vedú práve do stavu s oným jedným ponechaným prechodom. Pri vykonaní implicitného prechodu automat neprečíta zo vstupného reťazca žiaden symbol a preto je časová zložitosť tohto riešenia horšia, ako pri bežnej DFA architektúre. Z tohto dôvodu je v literatúre tento typ automatu označovaný aj ako *Delayed Input DFA* (D^2FA).

Garantovanie maximálnej doby spracovávania jedného vstupného symbolu je preto riešené tak, že je obmedzený maximálny počet na seba nadväzujúcich implicitných prechodov a teda, že automat môže za sebou vykonať predom daný maximálny počet implicitných prechodov a následne musí vykonať štandardný prechod s prečítaním symbolu (prípadne sa automat zasekne). Tento maximálny počet nadväzujúcich implicitných prechodov nazývame *polomer* D^2FA.



Obr. 4.3: DFA a D^{2}FA odpovedajúce regulárnym výrazom a^+, b^+c a c^*d^+. Prevzaté z [3].

Na obrázku 4.3 možno vidieť, aký výrazný vplyv má D^{2}FA transformácia na počet prechodov automatu. Zo stavov 2, 4 a 5 boli odstránené všetky odchádzajúce prechody, nakoľko všetky viedli, po prečítaní určitých symbolov, do tých istých stavov a boli nahradené prechodmi implicitnými. Tie vedú do stavu číslo 1, pri ktorom odchádzajúce stavy zostali nezmenené. Stavy 3 a 1 sa líšili v prechode na symbol c, preto je tento prechod ponechaný pri oboch stavoch.

Kapitola 5

Navrhovaná architektúra

Architektúra pre vyhľadávanie regulárnych výrazov bola navrhovaná s ohľadom na dva hlavné požiadavky, ktoré by mala spĺňať. Tým prvým je, že výsledná architektúra musí zaistiť multigigabitovú priepustnosť a druhým, že jej rekonfigurácia, teda zmena vyhľadávanej sady regulárnych výrazov, musí byť čo najrýchlejšia.

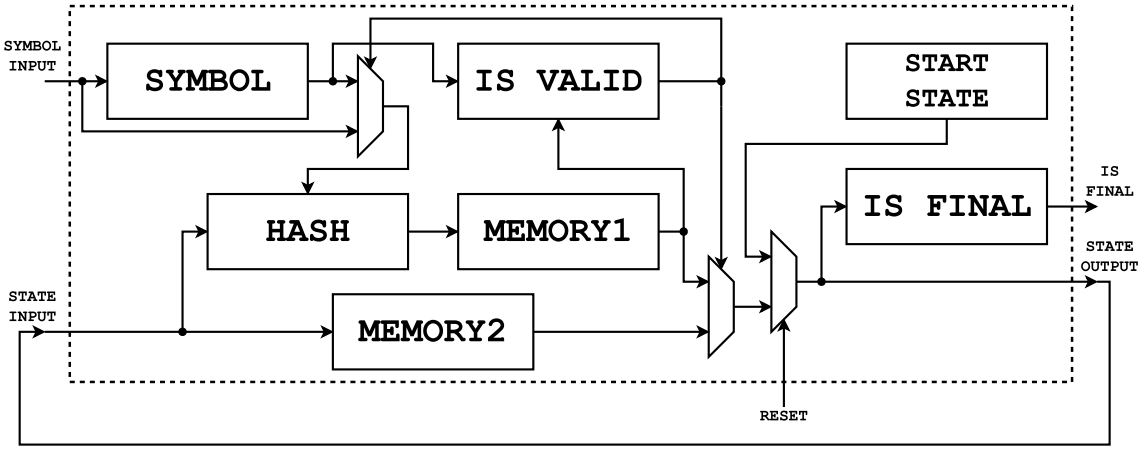
Zaistenie dostatočne vysokej priepustnosti je možné len s prihliadaním na počet zdrojov, ktoré daná architektúra zaberá. Ako bolo spomenuté v úvode, vysoký výkon je pri hardvérovom vyhľadávaní dosahovaný najmä paralelizáciou a to je práve ten dôvod, prečo sa musíme snažiť o minimalizáciu zdrojov. To totiž umožňuje umiestniť na čip väčšie množstvo paralelných jednotiek a tým škálovať celkový výkon a priepustnosť.

Rýchlosť zmeny sady vyhľadávaných regulárnych výrazov je závislá na spôsobe, akým je reprezentovaná prechodová funkcia automatu na čipe. NFA architektúry, ktoré prechodové funkcie mapujú priamo do logiky čipu, majú tú nevýhodu, že akákoľvek zmena je možná len rekonfiguráciou celého čipu FPGA. Naproti tomu, DFA architektúry ukladajú prechodovú tabuľku do pamätí a v takom prípade predstavuje rekonfigurácia len zmenu obsahu týchto pamätí.

Po dôkladnom oboznámení sa s existujúcimi architektúrami a ich vlastnosťami bolo teda rozhodnuté, že navrhovaná architektúra bude založená na *DFA s implicitnými prechodmi*, ktoré svojimi vlastnosťami spĺňajú obe požiadavky. Pre rýchlu zmenu vyhľadávanej sady totiž stačí zmeniť obsah pamäte a s tým spojené pamäťové nároky sú zase redukované vytvorením implicitných prechodov.

5.1 Bloková schéma

Na obrázku 5.1 je znázornená jednoduchá bloková schéma, ktorej vytvorenie predstavovalo prvý krok návrhu výslednej architektúry. Jej jadro tvoria dve pamäte, ktoré reprezentujú prechodové funkcie pre štandardné a implicitné prechody. Prechod automatu z jedného stavu do druhého v navrhnutej architektúre predstavuje prístup do pamäte a vyčítanie čísla nasledujúceho stavu. Úlohou ostatných blokov je dočasné uloženie tohto čísla, jeho validácia, rozhodnutie, či je stavom koncovým a jeho použitie na adresovanie pamäte pri ďalšom prechode. Bližší opis úloh a funkcionality jednotlivých blokov je náplňou sekcie 5.2.



Obr. 5.1: Bloková schéma navrhovanej architektúry.

Popis jednotlivých blokov v schéme na obrázku 5.1:

- `SYMBOL` - register uchovávajúci hodnotu práve prečítaného symbolu,
- `START STATE` - register uchovávajúci číslo počiatočného stavu,
- `HASH` - funkcia, ktorá ráta adresu do pamäte z čísla stavu a symbolu,
- `IS VALID` - funkcia, ktorá validuje záznam v pamäti,
- `MEMORY1` - pamäť obsahujúca štandardné prechody,
- `MEMORY2` - pamäť obsahujúca implicitné prechody,
- `IS FINAL` - funkcia, ktorá rozhoduje, či je daný stav koncový.

5.2 Kľúčové prvky architektúry

5.2.1 Hašovacia funkcia

Úlohou tohto bloku je skombinovať číslo aktuálneho stavu a hodnotu vstupného symbolu a to spôsobom vhodným na adresovanie pamäte. Okrem toho sú na túto funkciu kladené ďalšie požiadavky:

- výsledok tejto funkcie musí byť pre každý prechod jedinečný,
- množina výsledkov pre všetky prechody by mala byť čo najucelenejšia (minimalizácia nevyužitých adries),
- funkcia musí byť čo najjednoduchšia (minimalizácia využitých zdrojov).

Z hore uvedených podmienok, len posledná je jednoducho splniteľná v rovine návrhu architektúry. Dostatočne jednoduchá hašovacia funkcia (aritmetický súčet, exkluzívny súčet, ...) nedokáže garantovať bezkolíznosť pre ľubovoľné očíslovanie stavov automatu a minimalizovanie nevyužitých adries. Pre splnenie aj zvyšných dvoch podmienok je nutné upraviť aj samotnú štruktúru automatu (viz kapitola 6).

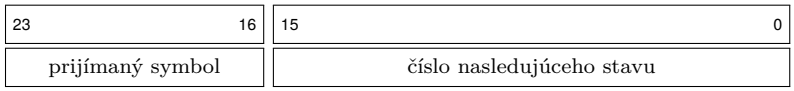
5.2.2 Pamäť

Vzhľadom na fakt, že sa v automate vyskytujú dva druhy prechodov, je aj spôsob uloženia týchto prechodov odlišný a to konkrétne do dvoch separátne adresovaných pamätí s rôznou bitovou šírkou záznamu.

Pamäť so štandardnými prechodmi

Táto pamäť a dáta, ktoré v sebe obsahuje, reprezentuje všetky štandardné prechody v automate. Je adresovaná výsledkom hašovacej funkcie a na jednotlivých adresách má uložené čísla nasledujúcich stavov a hodnoty vstupných (prijímaných) symbolov pre detegovanie kolízií hašovacej funkcie (viz kapitola 5.2.3). Štruktúru záznamu pamäte štandardných prechodov je možno vidieť na ilustrácii 5.1.

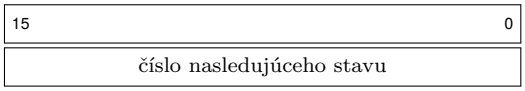
Register 5.1: Štruktúra záznamu v pamäti so štandardnými prechodmi (16-bit)



Pamäť s implicitnými prechodmi

Nakoľko sú implicitné prechody závislé iba na čísle aktuálneho stavu, nie je nutné ukladať do pamäte dodatočné informácie na detekciu kolízií. Pamäť je teda adresovaná číslom aktuálneho stavu a obsahuje iba čísla stavov nasledujúcich. Štruktúru záznamu pamäte implicitných prechodov je možno vidieť na ilustrácii 5.2.

Register 5.2: Štruktúra záznamu v pamäti s implicitnými prechodmi (16-bit)



5.2.3 Detekcia kolízie hašovacej funkcie

Z požiadaviek na hašovaciu funkciu vyplýva, že jej výsledok je jedinečný len pre obor dvojíc stav-symbol vyskytujúcich sa v pravidlách. Nakoľko sa však na vstupe automatu môže objaviť ľubovoľný symbol z abecedy automatu a automat ani zďaleka nemá definované prechody pre každý stav a symbol, môže sa na vstup hašovacej dostať taká dvojica, že výsledok hašovacej funkcie je rovnaký ako výsledok pre nejakú dvojicu, ktorá sa v pravidlách nachádza. Pre detegovanie takéhoto stavu je preto v pamäti štandardných prechodov, okrem čísla nasledujúceho stavu, uložená aj hodnota vstupného symbolu. Pokiaľ je hodnota symbolu na vstupne hašovacej funkcie rovnaká, ako hodnota uložená v pamäti, je možné s istotou rozlíšiť, či pre danú dvojicu na vstupe existuje štandardný prechod a pokiaľ áno, do ktorého stavu vedie.

5.2.4 Detekcia koncových stavov

S využitím toho, že pre účely zaistenia jedinečnosti výsledku hašovacej funkcie a minimalizácie nevyužitých adries je nutné modifikovať aj samotný automat, je k týmto modifikáciám zahrnutá aj taká, ktorá pridelí koncovým stavom čísla v intervale 1 až x, kde x je maximálne číslo koncového stavu. Pre detegovanie koncových stavov teda stačí overiť, či je číslo stavu väčšie ako nula a menšie, alebo rovné x.

5.3 Princíp činnosti

1. Automat je inicializovaný vyvedením čísla počiatočného stavu z registru `START STATE` na výstup `STATE OUTPUT`.

2. Podľa povahy prechádzajúceho prechodu je na vstup hašovacej funkcie `HASH` privedená hodnota aktuálneho symbolu na vstupe (štandardný prechod), alebo hodnota vstupného symbolu z prechádzajúceho prechodu (implicitný prechod), uložená v registri `SYMBOL`

3. Z pamätí sú vyčítané hodnoty. V prípade pamäte `MEMORY1` z adresy danej výsledkom bloku `HASH` a v prípade pamäte `MEMORY2` z adresy zo vstupu `STATE INPUT`.

4. Hodnota z pamäte `MEMORY1` je validovaná – hodnota symbolu uloženého v pamäti je porovnaná s hodnotou registru `SYMBOL`. V prípade zhody a v prípade, že je číslo stavu je rôzne od nuly, je na výstupe `STATE OUTPUT` číslo nasledujúceho stavu z pamäte `MEMORY1`, inak je na výstupe číslo stavu z pamäte `MEMORY2`.

5. Spolu s číslom stavu je na výstup privedený aj jednobitový signál z bloku `IS FINAL`, ktorý signalizuje, či je `NEXT STATE` stavom koncovým, alebo nie.

5.4 VHDL implementácia

Architektúra predstavená v tejto kapitole bola implementovaná v jazyku VHDL. Jej rozhranie je zachytené na výpise 5.1.

Výpis 5.1: Rozhranie implementovanej komponenty.

```
entity DDFA is
  generic(
    STATE_WIDTH   : natural := 16;
    SYMBOL_WIDTH  : natural := 8
  );

  port(
    CLK    : in std_logic;
    RESET  : in std_logic;
    EN     : in std_logic;
    RW     : in std_logic;

    SYMBOL_IN     : in  std_logic_vector(SYMBOL_WIDTH - 1 downto 0);
    SYMBOL_IN_VLD : out std_logic;
    STATE_IN      : in  std_logic_vector(STATE_WIDTH - 1 downto 0);
    STATE_OUT     : out std_logic_vector(STATE_WIDTH - 1 downto 0);
    IS_FINAL      : out std_logic;

    CFG_ADDR : in std_logic_vector(STATE_WIDTH + 1 downto 0);
    CFG_DATA : in std_logic_vector(STATE_WIDTH + SYMBOL_WIDTH - 1 downto 0)
  );
end entity DDFA;
```

Pre jednoduchú úpravu maximálnych bitových šírok, ktoré daná architektúra podporuje, obsahuje toto rozhranie dva generické parametre a to konkrétne:

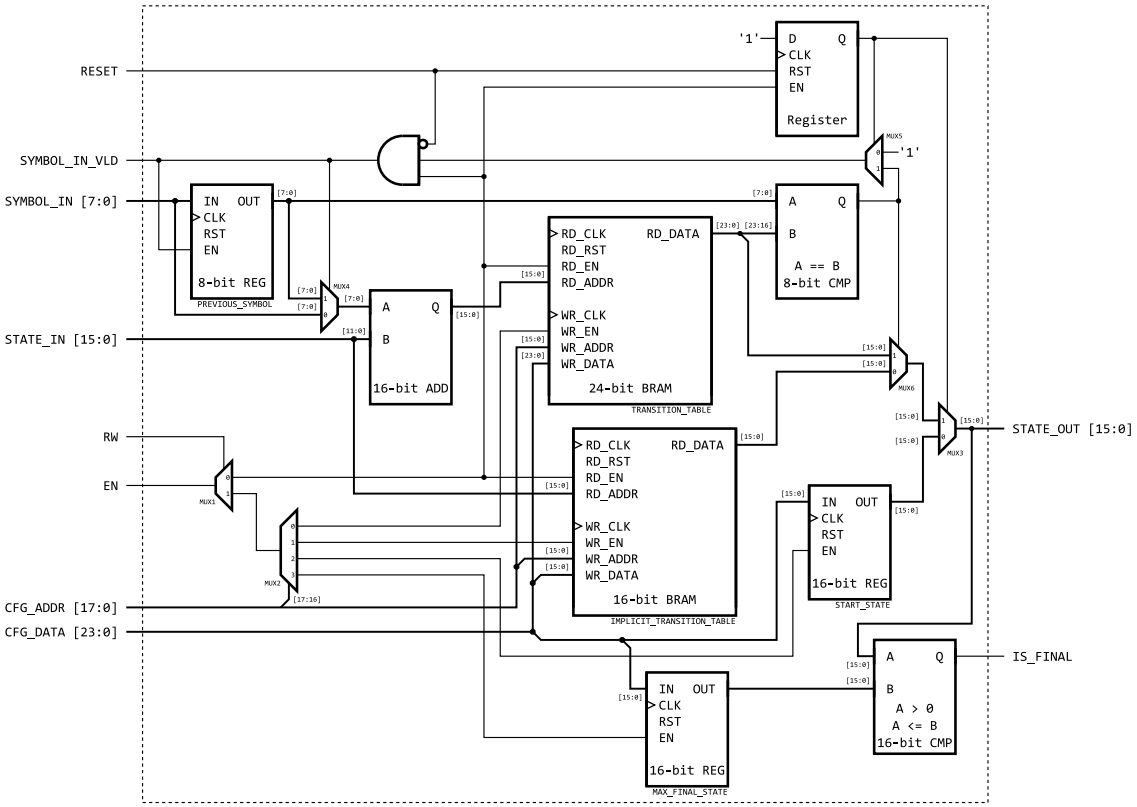
- `STATE_WIDTH`, ktorý upravuje maximálnu bitovú šírku čísla stavu,
- `SYMBOL_WIDTH`, ktorý upravuje maximálnu bitovú šírku symbolu.

Navrhnutá komponenta ďalej obsahuje nasledovné vstupy a výstupy:

- `CLK`, zdroj hodinového signálu. Akákoľvek zmena na výstupoch komponenty sa deje vždy zarovno nábežnej hrany hodinového signálu.
- `RESET`, synchrónny reset komponenty. V prípade logickej `'1'` na tomto vstupe sa s nábežnou hranou `CLK` objaví na výstupe `STATE_OUT` hodnota `START_STATE`.
- `EN`, zdroj povoľovacieho signálu. Funkčnosť celej komponenty je podmienená prítomnosťou logickej `'1'` na tomto vstupe. V prítomnosti logickej `'0'` sa hodnoty všetkých výstupov komponenty s nábežnou hranou `CLK` nemenia.
- `RW`, prepínanie medzi spracovaním symbolov na vstupe a zapisovaním konfigurácie do pamäte. V prípade logickej `'0'` na tomto vstupe, komponenta číta a spracúva symboly zo vstupu, v prípade logickej `'1'` je spracovávanie symbolov pozastavené a do pamätí sú zapisované hodnoty z konfiguračného rozhrania (zmena obsahu prechodovej tabuľky).
- `SYMBOL_IN`, vstupný symbol.
- `SYMBOL_IN_VLD`, jednobitový príznak signalizujúci, či je komponenta pripravená spracovať vstupný symbol, alebo nie (napríklad v prípade vykonávania implicitného prechodu, alebo v prípade zapisovania konfigurácie).
- `STATE_IN`, číslo stavu, z ktorého bude vykonaný ďalší prechod.
- `STATE_OUT`, číslo aktuálneho stavu.
- `IS_FINAL`, jednobitový príznak signalizujúci, či je stav `STATE_OUT` koncový, alebo nie.
- `CFG_ADDR`, konfiguračné rozhranie, adresa, z ktorej vrchné dva bity určujú, do ktorej pamäte/registra, sa bude zapisovať a to:
  - `"00"` - pamäť štandardných prechodov,
  - `"01"` - pamäť implicitných prechodov,
  - `"10"` - register s hodnotou počiatočného stavu,
  - `"11"` - register s hodnotou maximálneho čísla koncového stavu.

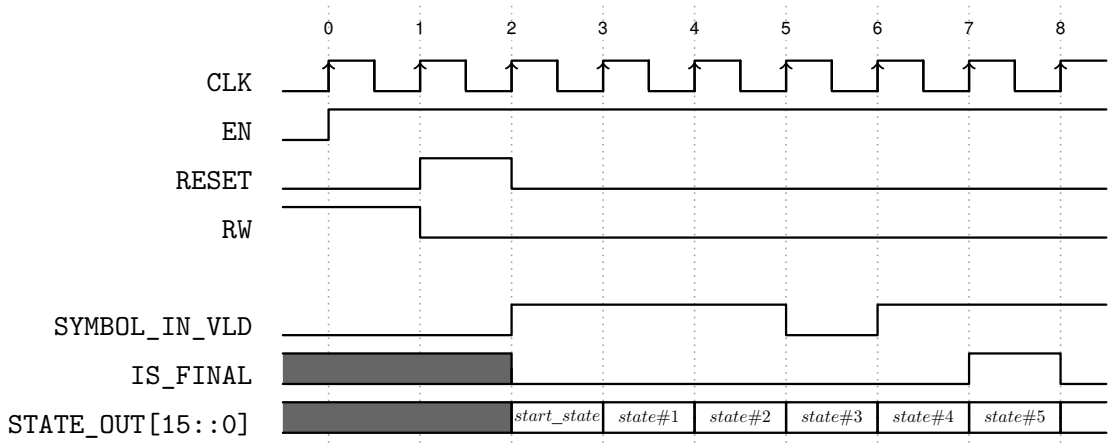
  Ostatné spodné bity sú použité na adresovanie jednotlivých buniek položiek a v prípade zapisovania do registrov je hodnota týchto bitov ignorovaná.
- `CFG_DATA`, konfiguračné rozhranie, hodnota zapisovaná na jednotlivé adresy. V prípade, že bitová šírka pamäte/registra je menšia, ako bitová šírka tohto vstupu, je použitý iba potrebný počet spodných bitov.

Obrázok 5.2 zobrazuje detailnú schému zapojenia navrhnutej komponenty.



Obr. 5.2: Detailná schéma navrhovanej architektúry.

5.4.1 Časové diagramy



Obr. 5.3: Časový diagram spracovávania symbolov na vstupe.

Časový diagram na obrázku 5.3 znázorňuje a demonštruje činnosť komponenty automatu tesne po nahratí konfigurácie do pamäte. Ako je možné vidieť, v čase $t = 0$ je komponenta

aktivovaná (logická 1 na vstupe `EN`) a zároveň je do nej nahrávaná konfigurácia (logická 1 na vstupe `RW`).

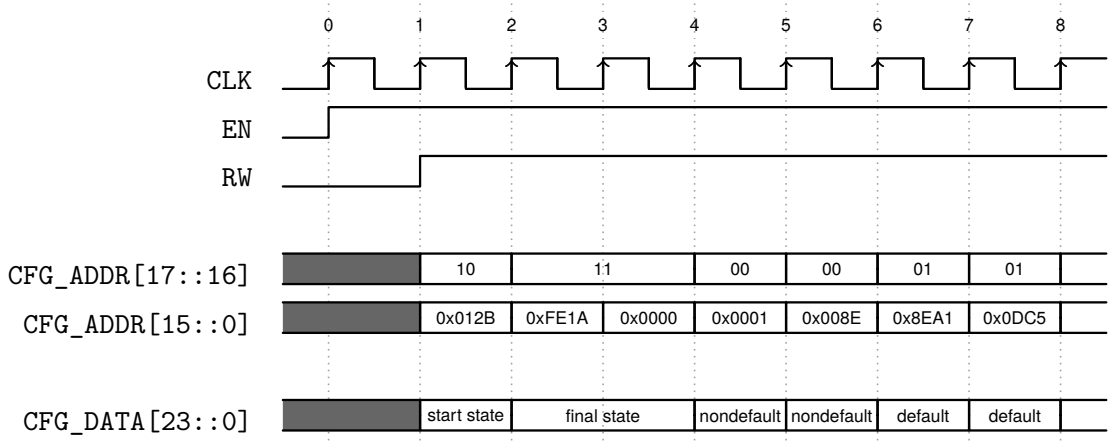
V čase $t = 1$ je nahrávanie konfigurácie dokončené (pre demonštratívne účely trvalo nahrávanie iba jednu periódu hodinového signálu) a automat je resetovaný do počiatočného stavu (logická 1 na vstupe `RESET`).

V čase $t = 2$ je prečítaný prvý vstupný symbol (logická 1 na výstupe `SYMBOL_IN_VLD`) a zároveň je od tohto času definovaná hodnota výstupu `STATE_OUT` – ihneď po resetovaní komponenty je na výstupe `STATE_OUT` číslo počiatočného stavu.

Od času $t = 3$ sa s každou nábežnou hranou `CLK` objaví na výstupe číslo stavu, v ktorom sa automat práve nachádza.

Pri prechode zo stavu *state*#3 do stavu *state*#4 automat vykonáva implicitný prechod – pri tomto prechode nie je prečítaný žiadny vstupný symbol (logická 0 na výstupe `SYMBOL_IN_VLD` v čase $t = 5$).

Stav *state*#5 je stavov koncovým, čo je signalizované logickou 1 na výstupe `IS_FINAL` v čase $t = 7$.



Obr. 5.4: Časový diagram zápisu konfigurácie.

Diagram 5.4 ukazuje správanie sa komponenty pri zapisovaní konfigurácie do pamäte. V čase $t = 1$ vrchné dva bity adresy s hodnotou `"10"` jednoznačne selektujú register obsahujúci číslo počiatočného stavu. V $t = 2$ zase vrchné dva bity adresy s hodnotou `"11"` určujú ako miesto zápisu register s maximálnym číslom koncového stavu. Z toho dôvodu sa po zmene spodných bitov adresy v čase $t = 3$ nezmení, nakoľko sú oba registre adresované iba vrchnými bitmi a spodné bity sú nezapojené.

Situácia je ale iná v prípade adresovania pamätí. Vrchné dva bity určujú, či sa hodnota zapíše do pamäte štandardných prechodov (čas $t = 4$ a $t = 5$), alebo pamäť implicitných prechodov ($t = 6$ a $t = 7$). V oboch prípadoch sú spodné bity využité na adresovanie samotnej pamäte, preto pri ich zmena znamená aj zmenu miesta, kam sa hodnota z `CFG_DATA` zapíše.

Kapitola 6

Programové vybavenie

Súčasťou tejto práce je aj vytvorenie programového vybavenia na prevod regulárnych výrazov na konečné automaty a na optimalizovanie ich vlastností, formy a štruktúry s cieľom dosiahnuť multigigabitovú priepustnosť a s cieľom zaistenia vlastností vyžadovaných navrhnutou architektúrou, ktoré sú potrebné pre jej správnu funkčnosť.

Ako implementačný jazyk bol zvolený jazyk C++ a to z dôvodu, že sa jedná o jazyk prekladaný priamo do strojového kódu a ktorý umožňuje nízkoúrovňové programovanie, vďaka ktorému je možné zaistiť dostatočne nízku časovú a pamäťovú zložitosť implementácie. Štandardná knižnica tohto jazyka zároveň poskytuje veľké množstvo abstraktných dátových typov, ktoré veľkou mierou uľahčujú prácu programátora.

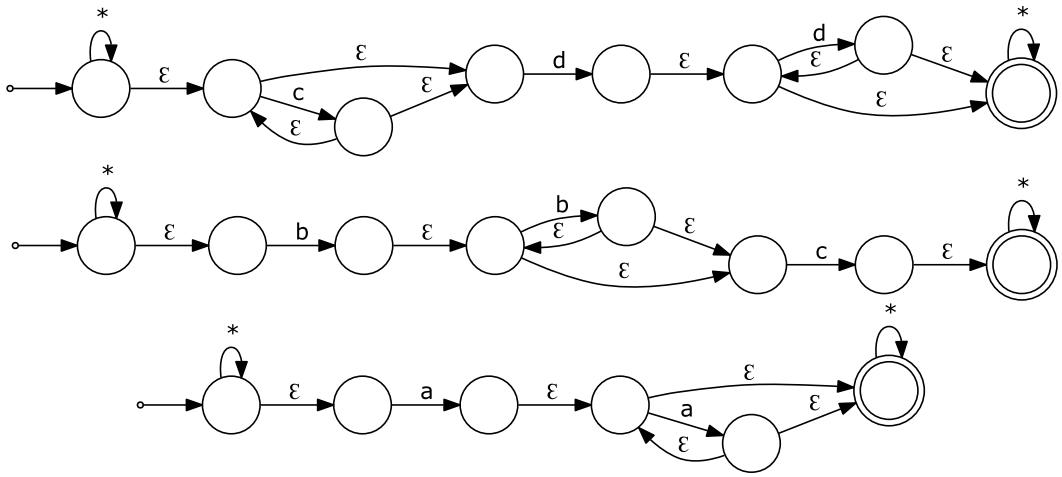
V tejto kapitole bude podrobnejšie objasnená funkcionalita vytvoreného programu a to vo forme nasledovných krokov, z ktorých každý vykonáva určitú transformáciu nad daným automatom:

1. prevod regulárnych výrazov na DFA,
2. redukovanie vstupnej abecedy,
3. transformácia DFA na D^2FA,
4. prečíslovanie abecedy.

6.1 Prevod regulárnych výrazov na DFA

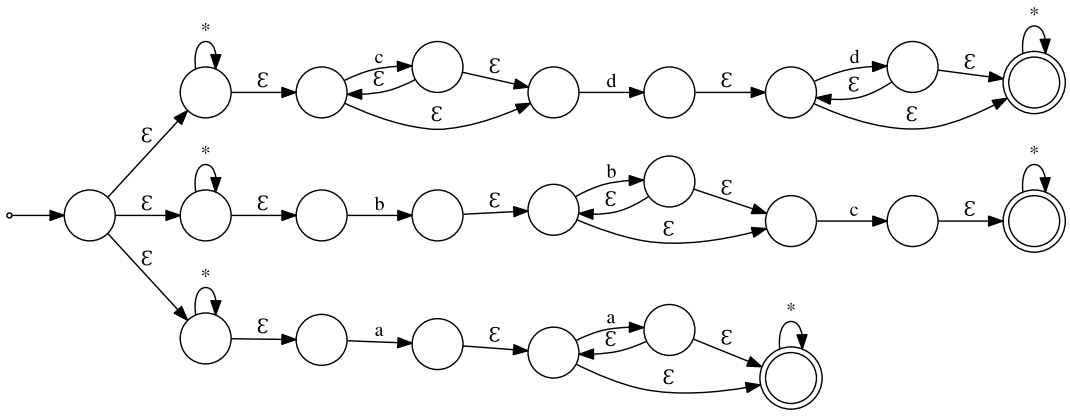
Proces prevodu regulárnych výrazov je realizovaný s využitím knižnice *libfa* [11]. Tá obsahuje implementácie všetkých algoritmov potrebných pre prevod a to konkrétne:

1. **prevod jednotlivých regulárnych výrazov na NFA** – každý regulárny výraz je transformovaný na samostatný automat. Na obrázku 6.1 možno vidieť ukážku takejto transformácie,



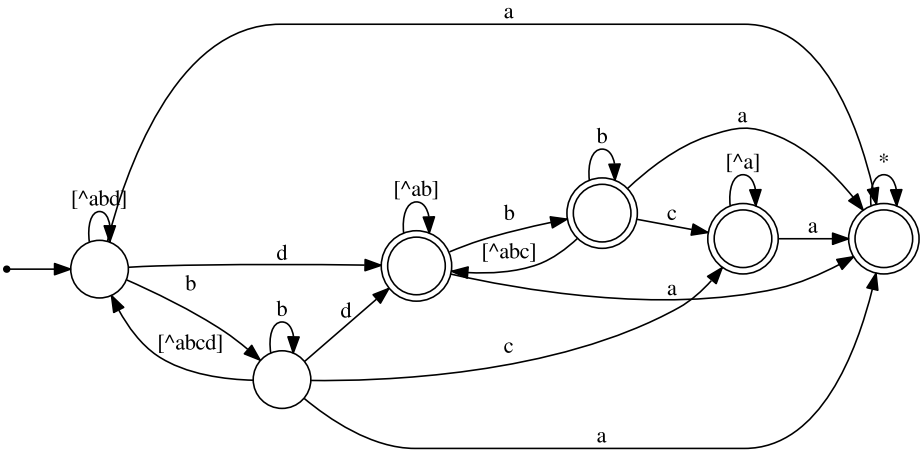
Obr. 6.1: NFA odpovedajúce regulárnym výrazom a^+, b^+c a c^*d^+.

2. **zjednotenie jednotlivých NFA do jedného NFA** – je vytvorený nový počiatočný stav, z ktorého vedú epsilon prechody do počiatočných stavov jednotlivých NFA (viz obrázok 6.2),



Obr. 6.2: Zjednotené NFA pre výrazy a^+, b^+c a c^*d^+.

3. **determinizácia a minimalizácia DFA** – v poslednej fáze je celý automat deterministicky, sú z neho odstránené epsilon prechody a je minimalizovaný (viz obrázok 6.3).



Obr. 6.3: Minimalizované a determinizované DFA pre výrazy a^+, b^+c a c^*d^+.

6.2 Redukovanie vstupnej abecedy

Redukovanie abecedy DFA je prvým priamočiarym spôsobom, ktorým je možné výrazným spôsobom znížiť počet jeho prechodov. Symboly abecedy sú vhodným spôsobom rozdelené na navzájom disjunktné podmnožiny – *triedy*. Každá takáto trieda je potom nahradená jediným symbolom a teda veľkosť novej abecedy je rovná počtu tried vytvorených z pôvodnej abecedy automatu.

Význam tejto optimalizácie spočíva vo výraznom premietnutí sa redukovania abecedy do počtu prechodov samotného automatu. Každú množinu prechodov medzi dvoma stavmi automatu je totižto možné taktiež rozdeliť na podmnožiny – *skupiny*, ktoré sú ekvivalentné k triedam symbolov (všetky prechody z jednej skupiny obsahujú symboly z jedinej triedy). Tieto skupiny sú potom nahradené jediným prechodom obsahujúcim symbol, ktorým bola trieda v abecede nahradená.

Pri rozdeľovaní symbolov abecedy do tried treba zaistiť, aby pôvodný a novo vzniknutý automat boli ekvivalentné – aby prijímali rovnaký jazyk. Toto bude platiť iba v prípade splnenia podmienky, že automat, po prečítaní ľubovoľného symbolu z triedy, vykoná prechod vždy do toho istého stavu. V takom prípade je nepodstatné rozlišovať konkrétny symbol z triedy a takáto trieda môže byť nahradená jediným symbolom.

Pre účely klasifikovania symbolov do tried bol v rámci práce navrhnutý algoritmus 1. Vstupom do tohto algoritmu je množina, kde každý prvok predstavuje množinu symbolov z pravidiel, ktoré majú spoločný stav na ľavej a spoločný stav na pravej strane (stavy na ľavej a pravej strane ale môžu byť rôzne).

Z algoritmu 1 vyplýva, že v najhoršom prípade bude abeceda rozdelená na triedy, z ktorých každá bude obsahovať jediný symbol. V takomto prípade nepredstavuje tento krok žiadnu optimalizáciu, keďže veľkosť abecedy, a tým pádom aj počet stavov automatu zostane nezmenený.

```
Input: outputSymbols
Output: symbolClasses

symbolClasses := ∅
unclassifiedSymbols := {0 − 255}

while |unclassifiedSymbols| > 0 do
    class := ∅

    for symbol : unclassifiedSymbols do
        insert symbol to class
        symbolBelongsToClass := true

        for set : outputSymbols do
            if class ∩ set ≠ ∅ then
                if class ⊈ set then
                    symbolBelongsToClass := false
                    break
                end
            end
        end

        if symbolBelongsToClass then
            erase symbol from unclassifiedSymbols
        else
            erase symbol from class
        end
    end

    insert class to symbolClasses
end
```

Algoritmus 1: Algoritmus pre klasifikovanie symbolov abecedy.

6.3 Transformácia DFA na D^{2}FA

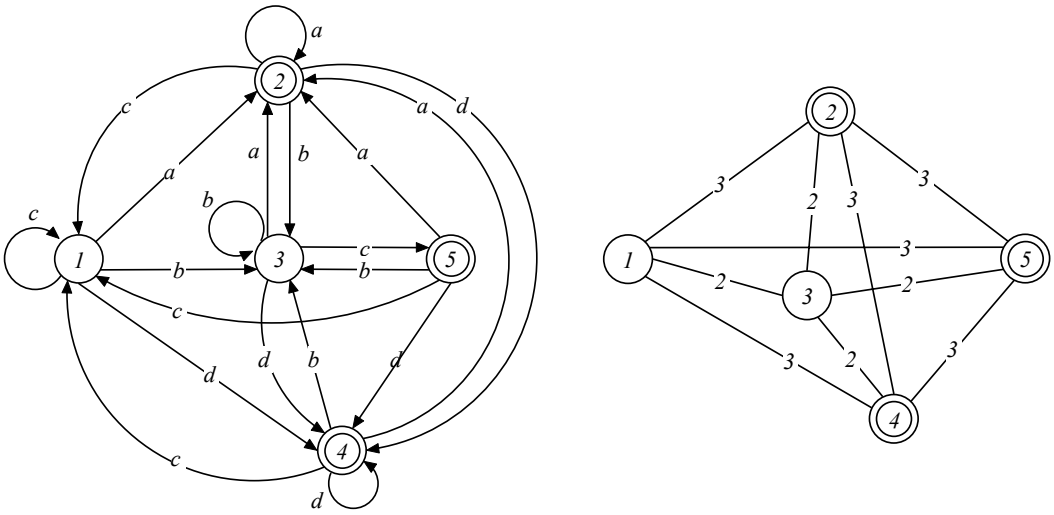
Transformovanie DFA na D^{2}FA a s tým spojené nahradenie štandardných prechodov implicitnými predstavuje optimalizáciu s najvyššou mierou redukcie prechodovej tabuľky.

Táto časť programového vybavenia je založená na referenčnej implementácii, ktorá bola vytvorená v rámci výskumnej skupiny ANT (Accelerated Network Technologies) pôsobiacej na FIT VUT. Táto implementácia a aj obsah tejto podkapitoly vychádza z pôvodného článku [3].

6.3.1 Graf redukovania priestorovej zložitosti

Prvým krokom transformácie je určenie potencionálnej redukcie priestorovej zložitosti. Táto úspora je vyjadrená pre každú možnú dvojicu rôznych stavov $[A, B]$ a to vo forme počtu prechodov vedúcich z A do rovnakého stavu, ako nejaký prechod vedúci z B a pri ktorých je zároveň prečítaný rovnaký symbol. Tento počet je zmenšený o jedna a reprezentuje, o koľko sa zmenší počet prechodov automatu, ak medzi stavmi A a B vytvoríme implicitný prechod.

Všetky tieto informácie je možné reprezentovať formou úplného neorientovaného grafu, ktorého vrcholy odpovedajú jednotlivým stavom automatu a ohodnotenia hrán udávajú potencionálnu redukciu priestorovej zložitosti (viz obrázok 6.4).



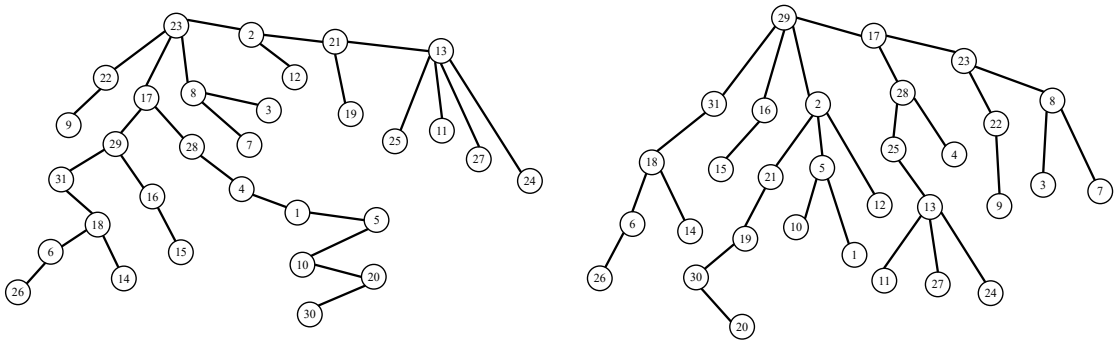
Obr. 6.4: DFA a jemu odpovedajúci graf redukovania priestorovej zložitosti (RPZ). Prevzaté z [3].

6.3.2 Maximálna kostra grafu redukovania priestorovej zložitosti

Proces transformácie pokračuje selekciou dvojíc stavov, medzi ktorými budú vytvorené implicitné prechody. Keďže zmysel celej transformácie spočíva v redukovaní priestorovej zložitosti, sú tieto dvojice vyberané s cieľom eliminovať čo najviac štandardných prechodov.

Najpriamočiarejšie riešenie tohto problému predstavuje konštrukcia *maximálnej kostry* grafu RPZ, ktorá spoľahlivo vyselektuje hrany s najväčším ohodnotením. Z definície kostry grafu zároveň vyplýva fakt, že kostra neobsahuje kružnice a teda splňuje vlastnosti stromového grafu (koreňom stromu môže byť ľubovoľný vrchol grafu – stav automatu).

S využitím týchto dvoch vlastností je už jednoduché určiť implicitné prechody, a to tak, že implicitné prechody budú viesť medzi uzlami spojenými hranou a ich orientácia bude smerom z potomka do rodičovského uzlu.



Obr. 6.5: Maximálna kostra vytvorená pôvodným (vľavo) a upraveným algoritmom (vpravo). Prevzaté z [3].

Bohužiaľ, hore uvedené riešenie má jednu zásadnú nevýhodu a tou je nemožnosť limitovania výšky stromu, čo predstavuje problém pri definovaní polomeru D^{2}FA. Preto Kumar vo svojom článku [3] predstavil aj modifikovaný algoritmus pre nájdenie maximálnej kostry grafu, ktorá reflektuje maximálny polomer D^{2}FA.

Obrázok 6.5 demonštruje rozdiel výstupu algoritmov pre nájdenie maximálnej kostry grafu. Ako vidieť, výstup modifikovaného algoritmu sa vyznačuje menšími rozdielmi medzi hĺbkami jednotlivých listov a menšou výškou celého stromu.

6.4 Prečíslovanie stavov

Pre čísla jednotlivých stavov platí jediná podmienka a to, že dva rôzne stavy nesmú mať rovnaké číslo a tento fakt využíva aj nasledujúca optimalizácia.

V navrhovanej architektúre je prechodová tabuľka štandardných prechodov reprezentovaná pamäťou, ktorá je adresovaná výsledkom hašovacej funkcie, ktorá má ako vstup číslo aktuálneho stavu a symbol na vstupe. Čísla stavov sú preto volené so zreteľom na:

- unikátnosť čísiel stavov
- najefektívnejšie využívanie adresného priestoru, so snahou o minimalizáciu nevyužitých miest v pamäti (fragmentácia)
- využívanie pamäťových miest ležiacich čo najbližšie k začiatku adresného priestoru

Na základe týchto požiadavkov bol preto v rámci práce navrhnutý algoritmus 2. Jeho vstupom sú jednotlivé stavy automatu, ktoré sú zostupne zoradené podľa počtu prechodov, ktoré z nich vedú. Dôvodom tohto zoradenia je, že pri stavoch s väčším počtom prechodov je väčšia šanca kolízie hašovacej funkcie a preto je obtiažnejšie takéto stavy prečíslovať. Tým, že sú prečíslované prednostne, je pravdepodobnosť kolízie minimalizovaná.

```
Input: automatonStates
for state : automatonStates do
    gap := first min from unusedAddresses
    if |state.outgoingSymbols| > 0 then
        minimalSymbol := min from state.outgoingSymbols
        do
            do
                state.number := inverseHash(gap, minimalSymbol)
                gap := next min from unusedAddresses
            while state.number ∉ usedStateNumbers
        while checkAddressConflicts(state) ≠ true
    else
        state.number := gap
    end
    insert state.number to usedStateNumbers
end
```

Algoritmus 2: Algoritmus pre prečíslovanie stavov.

Algoritmus na začiatku vygeneruje nové číslo stavu a to také, že adresa daná týmto číslom a výstupným symbolom (t.j. symboly, pre ktoré z daného stavu existuje prechod) s najnižšou ordinálnou hodnotou je rovná prvej nevyužitej adrese od začiatku adresného priestoru. Následne sa overí, či sú všetky ostatné adresy, dané zvyšnými výstupnými symbolmi, jedinečné.

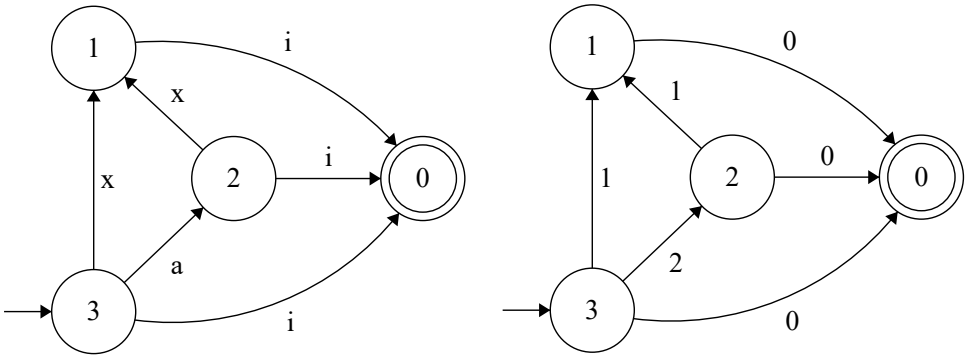
V prípade, že nie sú jedinečné, alebo v prípade, že novo vygenerované číslo stavu je už použité, sa celý proces opakuje a novo vygenerované číslo stavu je dané ďalšou nasledujúcou nevyužitou adresou. Ak sú všetky adresy a aj nové číslo stavu unikátne, označia sa ako použité a celý proces sa opakuje na ďalšom stave.

Pre rozlíšenie koncových stavov od nekoncových, je táto metóda mierne upravená. Koncové stavy sú prečíslované prednostne a ich nové číslo je vždy čo najnižšie nepoužité číslo stavu, rôzne od nuly (nula je vyhradená na špeciálne účely). Eliminácia konfliktov je riešená obdobne a to použitím ďalšieho dostupného čísla stavu a opakovaním procesu. Po prečíslovaní koncových stavov je celý interval od jedna (prvé dostupné číslo koncového stavu je jedna a žiadne konflikty nehrozia) po najvyššie použité číslo koncového stavu označené ako použité (vrátane nevyužitých čísiel stavov). Následne stačí overiť, či číslo stavu leží v tomto intervale a pokiaľ áno, je možné ho s istotou označiť ako stav koncový.

6.5 Prečíslovanie abecedy

Fragmentácia nastáva aj v prípade, že výstupné symboly stavu netvoria ucelený interval. Tento jav sa dá výraznejším spôsobom eliminovať prostým prečíslovaním abecedy.

Každému symbolu z abecedy je pridelená početnosť jeho výskytu – číslo, ktoré udáva, v koľkých pravidlách sa daný symbol nachádza. Symboly sú prečíslované spôsobom, že čím častejší má daný symbol výskyt, tým je jeho nová hodnota nižšia – najčastejšie sa vyskytujúci symbol je prečíslovaný na hodnotu nula. Tento spôsob prečíslovania zaisťuje, že hodnoty najčastejších symbolov ležia blízko pri sebe.



Obr. 6.6: Konečný automat s pôvodnými hodnotami symbolov (vľavo) a po prečíslovaní (vpravo).

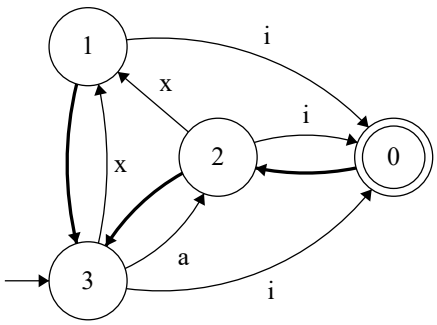
Na obrázku 6.6 možno vidieť konečný automat (vľavo) so stavom (stav číslo 3), ktorého výstupné symboly netvoria ucelený interval a ten istý automat (vpravo) po prečíslovaní symbolov horeuvedeným spôsobom.

6.6 Formát textovej reprezentácie D^{2}FA

Pre účely testovania a budúceho ďalšieho využitia transformovaného automatu, bolo nutné zvoliť vhodnú formu reprezentácie automatu. Tabuľka 6.1 popisuje štruktúru a formát textového súboru, ktorý je používaný pre reprezentovanie výsledného D^{2}FA. Formát bol navrhnutý v rámci výskumnej skupiny ANT@FIT a bol zvolený z dôvodu zachovania kompatibility s už existujúcimi nástrojmi a programovým vybavením.

Číslo riadku	**Formát**	**Typ**
	Popis	
1	S	číslo
	Počet stavov	
2	C	číslo
	Počet symbolov redukovanej abecedy (počet tried)	
3	X_0->Y_0| ... X_{C-1}->Y_{C-1}|	zoznam<číslo, číslo>
	Intervaly indexov zoznamu Z prislúchajúce jednotlivým triedam.	
4	A->Z_0| ... Z_{A-1}|	číslo; zoznam<číslo>
	Veľkosť vstupnej abecedy (zvyčajne 256); Ordinálne hodnoty symbolov patriacich do jednotlivých tried.	
5	B	číslo
	Číslo počiatočného stavu.	
6	D_0| ... D_{S-1}|	zoznam<číslo>
	Počty prechodov vedúcich z jednotlivých stavov (vrátane implicitných)	
7	E_0->F_0| ... $E_{\Sigma D-1}$->$F_{\Sigma D-1}$|	zoznam<číslo, číslo>
	Prechody automatu.	
8	G	číslo
	Počet koncových stavov.	
9	H_0| ... H_{G-1}|	zoznam<číslo>
	Čísla koncových stavov.	
10	I_0| ... I_{S-1}|	zoznam<číslo>
	Čísla cieľových stavov implicitných prechodov vedúcich z jednotlivých stavov, alebo -1, pokiaľ zo stavu nevedie implicitný prechod.	

Tabuľka 6.1: Formát textovej reprezentácie automatu D^{2}FA.



Obr. 6.7: Jednoduchý D^{2}FA.

Na obrázku 6.7 možno vidieť príklad jednoduchého D^{2}FA a vo výpise 6.1 textovú reprezentáciu tohoto automatu vo formáte opísanom v tabuľke 6.1.

```
4
3
0->0|1->1|2->2|
3->97|105|120
3
1|2|3|3
3->2|1->0|3->3|1->0|2->1|3->3|0->2|1->0|2->1|
1
0
2|3|3|-1|
```

Výpis 6.1: Textová reprezentácia jednoduchého D^{2}FA z obrázku 6.7.

Z tejto reprezentácie možno zistiť, že automat má 4 stavy (1. riadok) a že abeceda je redukovaná na 3 triedy (2. riadok). Každá trieda obsahuje jeden symbol (3. riadok), veľkosť vstupnej abecedy je 3 a tvoria ju symboly s ordinálnymi hodnotami 97, 105 a 120, čo sú symboly a, i, a x (4. riadok). Číslo počiatočného stavu je 3 (5. riadok). Zo stavu číslo 0 vedie jeden prechod, zo stavu 1 vedú dva prechody a zo stavu 2 a 3 vedú tri prechody (6. riadok).

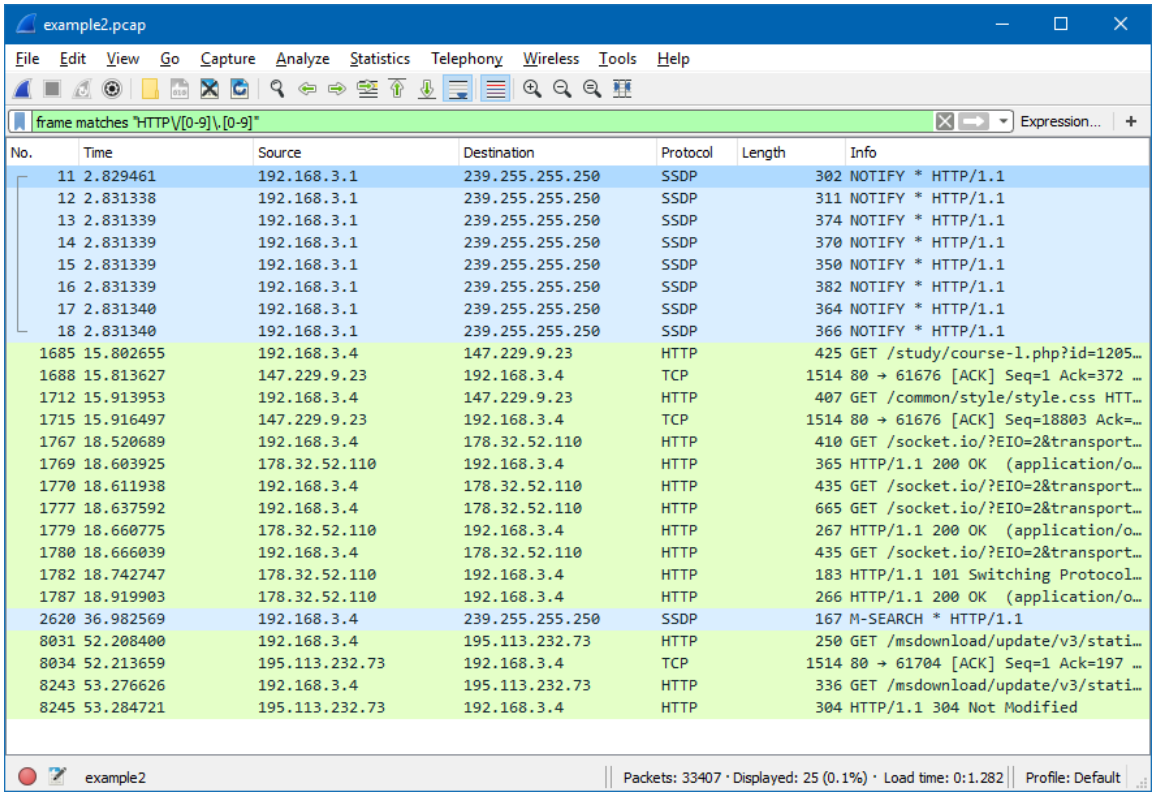
Zo stavu 0 vedie implicitný prechod do stavu 2, zo stavu 1 vedie prechod do stavu 0 a prečíta pri tom ľubovoľný symbol z triedy 1 a implicitný prechod do stavu 3. Zo stavu 2 vedú prechody do stavu 0 s triedou 1, do stavu 1 s triedou 2 a implicitný prechod do stavu 3. Zo stavu 3 vedú prechody do stavu 2 s triedou 0, do stavu 0 s triedou 1 a do stavu 1 s triedou 2 (7. riadok).

Počet koncových stavov je 1 (8. riadok) a číslo koncového stavu je 0 (9. riadok). Implicitné prechody vedú zo stavu 0 do stavu 2, zo stavu 1 do stavu 3, zo stavu 2 do stavu 3. Zo stavu 3 nevie žiadny implicitný prechod (10. riadok).

6.7 Overenie implementácie

Overenie správnej funkčnosti implementovaného nástroja bolo realizované porovnávaním s referenčnými výsledkami získanými pomocou nástroja Wireshark. Ten dokáže spracovávať súbory so zachytenou sieťovou komunikáciou (vo formáte pcap[1]) a jednou z jeho možností je aj filtrovanie paketov pomocou regulárnych výrazov. Obrázok 6.8 zobrazuje grafické užívateľské rozhranie tohto programu a v ňom pakety, ktoré obsahujú reťazec vyhovujúci zadanému regulárnemu výrazu.

[1] https://wiki.wireshark.org/Development/LibpcapFileFormat



Obr. 6.8: Program Wireshark s paketmi odfiltrovanými podľa výrazu `HTTP\/[0-9]\.[0-9]`.

Tieto výsledky boli následne porovnané s výsledkami vyhľadávania (filtrovania) získanými pomocou vytvoreného automatu. Bol pri tom použitý program pre simulovanie konečných automatov (vyvinutý v rámci ANT@FIT), ktorý je kompatibilný s formátom opísanom v sekcii 6.6.

Program na vstupe dostane textovú reprezentáciu simulovaného automatu a súbor obsahujúci zachytenú sieťovú komunikáciu. Jeho implementácia bola mierne upravená a to spôsobom, aby výstupom boli čísla paketov obsahujúce reťazce prijaté konečným automatom.

```
> ./test/runner test/test.ddfa test/example2.pcap | tr '\n' ' '
11 12 13 14 15 16 17 18 1685 1688 1712 1715 1767 1769 1770 1777 1779 1780
1782 1787 2620 8031 8034 8243 8245
```

Výpis 6.2: Výstup programu pre simulovanie D^{2}FA.

Ako je možné vidieť na výpise 6.2, čísla paketov, ktoré boli prijaté konečným automatom sú rovnaké, ako čísla paketov odfiltrované pomocou regulárneho výrazu na obrázku 6.8, čo potvrdzuje správnosť implementácie.

Kapitola 7

Zhodnotenie výsledkov

Nakoľko výsledok tejto práce predstavuje dva ucelené celky, aj zhodnotenie je nutné rozdeliť do týchto dvoch rovín:

- zhodnotenie architektúry hardvérovej reprezentácie konečných automatov
- zhodnotenie softvérového vybavenia pre transformáciu regulárnych výrazov do podoby optimalizovaných konečných automatov

7.1 Vlastnosti výslednej architektúry

Prvá rovina predstavuje vlastnosti navrhnutej architektúry a to konkrétne nároky na zdroje FPGA a priepustnosť výsledného riešenia. Hodnoty budú prezentované na troch rôznych konfiguráciách líšiacich sa bitovou šírkou čísla stavu:

8-bit – hodnoty generických parametrov `SYMBOL_WIDTH` = 8, `STATE_WIDTH` = 8

12-bit – hodnoty generických parametrov `SYMBOL_WIDTH` = 8, `STATE_WIDTH` = 12

16-bit – hodnoty generických parametrov `SYMBOL_WIDTH` = 8, `STATE_WIDTH` = 16

Prezentované hodnoty sú výsledkom syntézy VHDL implementácie a to konkrétne pre čip `xc7vh580thcg1931-2` z rodiny Virtex-7 od firmy Xilinx. Syntéza bola vykonaná pomocou nástroja *Vivado* verzie `2018.1`.

7.1.1 Využitie zdrojov FPGA

	8-bit	**12-bit**	**16-bit**
LUT	42	61	91
FF	25	33	41
BRAM	2	4	80

Tabuľka 7.1: Využitie zdrojov FPGA v závislosti na rôznych dátových šírkach čísiel stavov.

Tabuľka 7.1 zobrazuje využitie jednotlivých zdrojov FPGA v závislosti na rôznych dátových šírkach čísiel stavov. Za pozornosť predovšetkým stojí počet využitých BRAM a závislosť tohto počtu na šírke čísla stavu, ktorá má exponenciálny charakter.

7.1.2 Teoretická priepustnosť architektúry

V tabuľke 7.2 možno vidieť oneskorenie maximálnej trasy po syntéze v závislosti na bitovej šírke čísla stavu a k ním prislúchajúce hodnoty maximálnych pracovných frekvencií.

	8-bit	**12-bit**	**16-bit**
Oneskorenie najdlhšej trasy	4,47 ns	4,568 ns	5,275 ns
Maximálna frekvencia	223,713 MHz	218,914 MHz	189,573 MHz

Tabuľka 7.2: Oneskorenie najdlhšej trasy a maximálna pracovná frekvencia architektúry.

Tabuľka 7.3 znázorňuje teoretickú maximálnu priepustnosť navrhnutej hardvérovej reprezentácie D^2FA, ktorá je odvodená od maximálnych možných pracovných frekvencií uvedených v tabuľke 7.2.

	8-bit	**12-bit**	**16-bit**
Priepustnosť pre r = 1	894,8546 Mbit/s	875,6567 Mbit/s	758,2938 Mbit/s
Priepustnosť pre r = 2	596,5697 Mbit/s	583,7712 Mbit/s	505,5292 Mbit/s
Priepustnosť pre r = 3	447,4273 Mbit/s	437,8284 Mbit/s	379,1469 Mbit/s

Tabuľka 7.3: Teoretická minimálna priepustnosť v závislosti na polomere D^2FA.

Minimálna garantovaná priepustnosť je daná súčinom pracovnej frekvencie a bitovej šírky symbolu (v našom prípade 8 bitov), vydeleným maximálnou dobou spracovávania jedného vstupného symbolu (počet taktov), ktorá je rovná polomeru D^2FA zväčšenému o jedna.

Hodnoty priepustností v tabuľke 7.3 sú tzv. „*worst-case*“ hodnoty, teda hodnoty v najhoršom prípade. Takýto prípad predstavuje 100% pravdepodobnosť vykonania implicitného prechodu. Priepustnosť presahujúca 1 Gbit/s je napríklad v najlepšom prípade (8-bit, r = 1) dosiahnutá, ak táto pravdepodobnosť nie je väčšia ako 78 %.

7.2 Využitie pamäte

Obsahom nasledujúcej časti je demonštrácia jednotlivých transformačných a optimalizačných krokov, konkrétne ich vplyv na priestorovú zložitosť konečného automatu, čo do počtu stavov a prechodov, tak aj do množstva pamäte vyžadovanej na jeho hardvérovú reprezentáciu. Pre tieto účely boli zvolené demonštratívne sady regulárnych výrazov, ktoré sú reálne používané v sieťových aplikáciách.

```
/^.?.?....\\x00.\\x00.\\x00.\\x00.([\x01-\x3F][\x20-\x7f]+)*\\x00\\x00.\\
    x00[\x01\x03\x04\xFE\xFF]/
/^(OPTIONS|GET|HEAD|POST|PUT|DELETE|TRACE|CONNECT|PATCH) [^\r\n]+ HTTP
    \/[0-9]\.[0-9]\r\n/
/^HTTP\/[0-9]\.[0-9] [0-9]{3} ?[^\r\n]*\r\n/
/^(REGISTER|INVITE|ACK|CANCEL|BYE|OPTIONS|PRACK|SUBSCRIBE|NOTIFY|PUBLISH|
    INFO|REFER|MESSAGE|UPDATE) [^\r\n]+ SIP\/[0-9]\.[0-9]\r\n/
/^SIP\/[0-9]\.[0-9] [0-9]{3} ?[^\r\n]*\r\n/
/^([2-5][0-5][0-5]|EHLO)[- ~][\x20-\x7f]+\r\n/
```

Výpis 7.1: Sada regulárnych výrazov 1 pre detekciu základných aplikačných protokolov.

	Automat		Pamäť [pppt[1]]	
	Počet stavov	**Poč. prech./ impl. prech.**	**Bez prečísl. abecedy**	**S prečísl. abecedy**
DFA	488	97404/0	–	–
DFA s RA[2]	488	16354/0	–	–
D^{2}FA (neobm.)	488	2618/394	–	–
D^{2}FA (r = 1)	**488**	**9312/202**	**9361**	**9324**
D^{2}FA (r = 2)	**488**	**4540/337**	**4571**	**4571**
D^{2}FA (r = 3)	**488**	**3257/371**	**3302**	**3289**

Tabuľka 7.4: Pamäťové nároky D^{2}FA pre sadu 1.

```
/^[a-zA-Z]+\s+\x2Ferror-serverdown\x2Ejsp.*\x2E\x2E\x2F/mi
/^[a-zA-Z]+\s+\x2Findex\x2Ejsp\x3Flogout\x3Dtrue.*\x2E\x2E\x2F/mi
/^[a-zA-Z]+\s+\x2Flogin\x2Ejsp.*\x2E\x2E\x2F/mi
/^[a-zA-Z]+\s+\x2Fsetup\x2F\index\x2Ejsp.*\x2E\x2E\x2F/mi
/^[a-zA-Z]+\s+\x2Fsetup\x2Fsetup-.*\x2E\x2E\x2F/mi
/^[a-zA-Z]+\s+\x2F\x2Egif.*\x2E\x2E\x2F/mi
/^[a-zA-Z]+\s+\x2F\x2Epng.*\x2E\x2E\x2F/mi
/http\x3A\x2F\x2F[^\s]/
/^\s*JOIN/smi
/^\s*NICK/smi
/^\s*NOTICE/smi
/^\s*PRIVMSG/smi
/^\s*USERHOST/smi
/^\x3c(REQIMG|RVWCFG)\x3e/ism
```

Výpis 7.2: Sada regulárnych výrazov 2 z modulu programu *Snort*.

	Automat		Pamäť [pppt]	
	Počet stavov	**Poč. prech./ impl. prech.**	**Bez prečísl. abecedy**	**S prečísl. abecedy**
DFA	342	87552/0	–	–
DFA s RA	342	17100/0	–	–
D^{2}FA (neobm.)	342	1796/334	–	–
D^{2}FA (r = 1)	**342**	**9278/161**	**9301**	**9299**
D^{2}FA (r = 2)	**342**	**4168/269**	**4214**	**4224**
D^{2}FA (r = 3)	**342**	**2411/302**	**2464**	**2487**

Tabuľka 7.5: Pamäťové nároky D^{2}FA pre sadu 2.

Tabuľky 7.4 a 7.5 znázorňujú priestorovú zložitosť automatov odpovedajúcich regulárnym výrazom zobrazených na výpisoch 7.1 a 7.2 a pamäťové nároky hardvérovej reprezentácie týchto automatov naprieč rôznymi optimalizačnými technikami.

[1] počet položiek prechodovej tabuľky
[2] redukovaná abeceda

V prípade sady 1 je počet prechodov oproti pôvodnému DFA redukovaný na 9,56% (r = 1), resp. na 4,66% (r = 2) a 3,34% (r = 3) pôvodného počtu a potrebný počet položiek prechodovej tabuľky je 9324, resp. 4571 a 3289.

V prípade sady 2 je počet prechodov oproti pôvodnému DFA redukovaný na 10,59% (r = 1), resp. na 4,76% (r = 2) a 2,75% (r = 3) pôvodného počtu a potrebný počet položiek prechodovej tabuľky je 9299, resp. 4214 a 2464.

Rozdiel medzi počtom prechodov a potrebným počtom položiek tabuľky predstavuje počet nevyužitých položiek tabuľky. Najväčší podiel nevyužitých položiek nastal v prípade sady 2 s polomerom 3 – počet potrebných položiek bol o 53 položiek väčší ako ideálny počet (t.j. počet prechodov automatu), čo predstavovalo 97,84% využitie prechodovej tabuľky.

Prínos prečíslovania abecedy sa ukázal byť otázny, nakoľko sa počet potrebných položiek tabuľky v dvoch prípadoch znížil, v jednom prípade zostal nezmenený a v jednom prípade sa naopak zvýšil.

Kapitola 8

Záver

Cieľom práce bolo vytvoriť programové vybavenie na prevod regulárnych výrazov do podoby konečných automatov. Okrem toho malo toto vybavenie zaistiť aj ich optimalizáciu, s cieľom dosiahnutia multigigabitovej priepustnosti, so zreteľom na minimalizovanie pamäťových nárokov ich hardvérovej reprezentácie.

Na základe týchto požiadavkov bola navrhnutá hardvérová architektúra založená na *deterministických konečných automatoch s implicitnými prechodmi*. Tie výrazne redukujú veľkosť prechodovej tabuľky automatu a umožňujú reprezentovať väčšie množstvo regulárnych výrazov. Architektúra je synchrónna a využíva zreťazené spracovanie, čo umožňuje dosiahnutie vysokej pracovnej frekvencie a zaručuje škálovanie s ľubovoľnou veľkosťou prechodovej tabuľky.

Architektúra taktiež splňuje kritérium minimalizácie doby potrebnej na zmenu vyhľadávanej sady regulárnych výrazov, nakoľko táto zmena predstavuje iba zmenu uloženého obsahu blokových pamätí a nie je teda nutné vykonávať ani čiastočnú rekonfiguráciu čipu.

Pre správnu funkčnosť navrhnutej architektúry je nevyhnutné transformovať regulárne výrazy na vstupe do podoby DFA s implicitnými prechodmi a vhodným spôsobom prečíslovať stavy. To je úlohou navrhnutého a implementovaného programového vybavenia, ktoré taktiež zaisťuje ďalšie optimalizácie automatu, s cieľom minimalizovať pamäťové nároky.

Praktické využitie vytvoreného riešenia je možné v celej palete sieťových aplikácií – monitorovanie sieťovej prevádzky, detekcia bezpečnostných incidentov, vyrovnávanie záťaže, filtrovanie sieťovej prevádzky a iné.

Literatúra

[1] Clark, C. R.; Schimmel, D. E.: Efficient reconfigurable logic circuits for matching complex network intrusion detection patterns. In *Field Programmable Logic and Application*, editácia P. Y. K. Cheung; G. A. Constantinides, Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, ISBN 978-3-540-45234-8, s. 956–959.

[2] Hopcroft, J. E.; Motwani, R.; Ullman, J. D.: Automata theory, languages, and computation. *International Edition*, ročník 24, 2006.

[3] Kumar, S.; Dharmapurikar, S.; Yu, F.; aj.: Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. *SIGCOMM Comput. Commun. Rev.*, ročník 36, č. 4, August 2006: s. 339–350, ISSN 0146-4833.

[4] Limaye, S.: *VHDL: A Design Oriented Approach*. McGraw-Hill Publ., 2008, ISBN 978-0-07-014462-0.

[5] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, 2000, ISBN 81-812-8333-3.

[6] O Rabin, M.; Scott, D.: Finite automata and their decision problems. *IBM Journal of Research and Development*, ročník 3, 04 1959: s. 114–125.

[7] Paxson, V.; Asanović, K.; Dharmapurikar, S.; aj.: Rethinking Hardware Support for Network Analysis and Intrusion Prevention. In *Proceedings of the 1st USENIX Workshop on Hot Topics in Security*, HOTSEC'06, Berkeley, CA, USA: USENIX Association, 2006, s. 11–11.

[8] Sheng Yu: *Handbook of Formal Languages*, ročník 1. Springer, 1997, ISBN 35-406-0420-0.

[9] Sidhu, R.; Prasanna, V. K.: Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM '01. The 9th Annual IEEE Symposium on*, March 2001, s. 227–238.

[10] Sipser, M.: *Introduction to the Theory of Computation*, ročník 2. Thomson Course Technology Boston, 2006, ISBN 978-1133187790.

[11] Wadman, M.: libfa: C automata library to build, determinize, minimize, translate regexp etc. 2016, [Online; cit. 26. 03. 2018].
URL https://github.com/wader/libfa

[12] Watson, B.: A Taxonomy of Finite Automata Construction Algorithms. Technická správa, Computing Science, 1993.

[13] Wikipedia: Field-programmability — Wikipedia, The Free Encyclopedia. 2018, [Online; cit. 11. 5. 2018].
URL http://en.wikipedia.org/w/index.php?title=Field-programmability&oldid=759439313

[14] Wikipedia: Gate array — Wikipedia, The Free Encyclopedia. 2018, [Online; cit. 11. 5. 2018].
URL http://en.wikipedia.org/w/index.php?title=Gate%20array&oldid=837450650

[15] Xilinx: SXC4000E and XC4000X Series Field Programmable Gate Arrays. 1999, [Online; cit. 13. 12. 2017].
URL https://www.xilinx.com/support/documentation/data_sheets/4000.pdf

[16] Xilinx: Spartan-II FPGA Family Data Sheet. 2008, [Online; cit. 13. 12. 2017].
URL https://www.xilinx.com/support/documentation/data_sheets/ds001.pdf

[17] Yu, F.; Chen, Z.; Diao, Y.; aj.: Fast and memory-efficient regular expression matching for deep packet inspection. In *2006 Symposium on Architecture For Networking And Communications Systems*, Dec 2006, s. 93–102.

[18] Zeidman, B.: *Designing with FPGAS and CPLDS*. CRC Press, 2002, ISBN 15-782-0112-8.