České vysoké učení technické v Praze
Fakulta elektrotechnická

DIPLOMOVÁ PRÁCE

Optimalizace algoritmů pro FPGA

Praha, 2007 **Autor: David Matějíček**

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne 16 . 5 . 2007

podpis

Poděkování

Zde bych rád poděkovat vedoucímu této práce, ing. Přemyslu Šůchovi, za jeho podněty ke zpracovávanému tématu, ochotu a věnovaný čas při psaní této diplomové práce.

Abstrakt

Tato práce je motivována potřebou optimalizovat dobu výpočtu algoritmů používaných pro číslicové zpracování signálů (DSP), zefektivnit implementaci těchto algoritmů na programovatelné hradlové pole FPGA a hlavně zkrátit celkovou dobu návrhu. DSP algoritmy jsou často realizovány na hradlových polích FPGA. Výhodou je vysoký stupeň paralelizmu výpočtu, kterého lze pomocí těchto obvodů dosáhnout.

Práce se zabývá implementací nástroje ACGM (Automatic Code Generation for Matlab), který je součástí Torsche Scheduling toolboxu pro Matlab. Tento nástroj generuje kód pro simulaci v nástroji TrueTime a VHDL kód pro obvody FPGA.

Těžištěm této práce je návrh způsobu zadávání vstupních dat, implementace syntaktického analyzátoru a implementace generátoru kódu pro nástroj TrueTime.

Abstract

This thesis is motivated by the need to optimize computation time of Digital Signal Processing (DSP) algorithms designed in Matlab, generate an efficient FPGA implementation and especially reduce total design time. DSP algorithms are often implemented on Field Programmable Gate Arrays (FPGAs), because the FPGA design achieves a high level of parallelism.

The thesis deals with implementation of Automatic Code Generation from Matlab tool, which is part of Torsche Scheduling toolbox for Matlab. This tool generates code for simulation in TrueTime and FPGA design in VHDL.

The primary focus of this thesis is to design a format of input data, implement syntaktical analyser and implement code generator for TrueTime.

České vysoké učení technické v Praze - Fakulta elektrotechnická

Katedra řídicí techniky *Školní rok:* 2005/2006

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: David Matějíček

Obor: Technická kybernetika

Název tématu: Optimalizace algoritmů pro FPGA

Zásady pro vypracování:

1. Seznamte se se způsobem implementace algoritmů na FPGA.
2. Navrhněte řešení automatického generování kódu pro FPGA.
3. Implementujte a odzkoušejte navržené řešení.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí diplomové práce: Ing. Přemysl Šůcha

Termín zadání diplomové práce: zimní semestr 2005/2006

Termín odevzdání diplomové práce: leden 2007

prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická

v zast.
prof. Ing. Vladimír Kučera, DrSc.
děkan

V Praze dne 20.01.2006

Obsah

Seznam obrázků

Seznam tabulek

Seznam použitých zkratek

AST	Abstract Syntax Tree
DSP	Digital Signal Processing
DSVF	Digital State Variable Filter
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
HLS	High-Level Synthesis
HTG	Hierarchical Task Graph
LALR	Look-Ahead Left to Right
LUT	Look-up Table
PRDG	Polyhedral Reduced Dependence Graph
RTL	Register Transfer Level
SAP	Single Assignment Program
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WDF	Wave Digital Filter
WDEF	Wave Digital Elliptic Filter
XML	eXtensible Markup Language
XSLT	Extensible Stylesheet Language Transformations

Kapitola 1

Úvod

1.1 Motivace

Složitost dnešních DSP (Digital Signal Processing) systémů prudce narůstá a k implementaci těchto systémů pomalu přestávají stačit univerzální DSP procesory. Alternativou DSP procesorů jsou nové architektury programovatelných hradlových polí FPGA (Field Programmable Gate Array). Obvod FPGA může poskytnout více flexibilní a rychlejší řešení pro DSP aplikace. FPGA dovolují přizpůsobení architektury algoritmům, na rozdíl od DSP procesorů kde je nutné algoritmus přizpůsobit architektuře. V obvodech FPGA lze dosáhnou velkého stupně paralelismu a tím velké výkonnosti systémů.

Návrhy DSP algoritmů pro FPGA jsou tradičně děleny do dvou kroků — vývoj algoritmů a fyzická implementace hardwaru. Algoritmický návrhář vytváří a verifikuje potřebné DSP algoritmy využívající vyšší programovací jazyky pro popis na behaviorální úrovni bez ohledu na detailní implementaci hardwaru. Většina DSP systémových návrhářů algoritmů užívá jazyk MATLAB, který poskytuje výkonné a komfortní DSP vývojové prostředí, protože vestavěné matematické a grafické funkce umožňují snadnou simulaci. Druhým krokem návrhu je navržené specifikace algoritmů zapsané ve vyšším programovacím jazyce převést do popisu na úrovni meziregistrových přenosů (RTL – Register Transfer Level) v jazyce pro hardwarový popis (HDL – Hardware Description Language) jako jsou Verilog nebo VHDL.

Jelikož manuální tvorba RTL modelů je velmi složitá a časově velmi náročná jsou vyvíjeny nástroje pro automatickou transformaci behaviorálního popisu do RTL popisu. Proces automatické transformace behaviorálního popisu do RTL popisu se nazývá HL (High-Level) syntéza [8]. První fází HL syntézy je kompilace do vnitřní reprezentace, ob-

vykle založené na grafu řízení a grafu toku dat. Po kompilaci do vnitřní formy následují optimalizace jako propagace konstant a proměnných, eliminace společných podvýrazů a rozbalení smyček. Další fází HL syntézy je rozvržení. Během rozvržení dochází k přiřazení operací ke skutečným časovým okamžikům běhu algoritmu. Po rozvržení následuje fáze alokace zdrojů, při které jsou operace přiřazeny ke skutečným hardwarovým prostředkům. V poslední fázi je vygenerována datová cesta a na základě ní je vytvořen řadič.

1.2 Související literatura

Nástrojů pro HL syntézu existuje mnoho a můžeme je rozdělit podle několika kritérií, například podle jazyka použitého pro specifikaci algoritmů a nebo podle jazyka, ve kterém jsou implementovány. Pro specifikaci se používají jednak běžné programovací jazyky jako jsou C [10, 24, 23, 15], Matlab [3, 7] a Java [13] a jazyky od nich odvozené nebo speciálně navržené. Jednotlivé nástroje se dále liší podle míry interakce s uživatelem, kromě plně automatických existují nástroje umožňující kombinaci automatického a manuálního návrhu, kdy uživatel může střídat v jednotlivých krocích automatickou a manuální syntézu.

Jedním z mnoha nástrojů pro syntézu na vyšší úrovni je akademický nástroj SPARK [10]. Jako vstupní jazyk pro behavioralní popis algoritmů používá ANSI-C a výstupem je RTL VHDL. Vstupní jazyk má několik omezení, například nepodporuje použití pointerů, rekurzivní volání funkcí a vícerozměrné pole. K vnitřní reprezentaci využívá graf HTG (Hierarchical Task Graph), na který aplikuje řadu optimalizačních technik, jako eliminaci mrtvého kódu (dead code elimination), eliminaci společných podvýrazů (common sub-expression elimination) a nebo rozbalení smyček (loop unrolling), které zvyšuje paralelitu výpočtů. K nalezení nejlepšího rozvrhu používá transformační techniky „speculative code motions“ a „dynamic common sub-expression elimination“.

DEFACTO [24] je akademický nástroj pro HL syntézu. Vstupní specifikaci v jazyce C nebo Fortran převádí do vnitřní reprezentace ve formátu SUIF (Stanford University Intermediate Format). Tento nástroj hledá vhodnou hardwarovou implementaci FPGA ve VHLD splňující požadavky jak na výpočetní dobu, tak na velikost návrhu, množství logiky v hradlech. Pro dosažení nejlepšího návrhu hledá vhodnou kombinaci úrovní rozbalení jednotlivých smyček „loop unrolling factor“.

PACT HDL [15] je kompilátor jazyka C do hardwarového popisu HDL pro obvody FPGA a ASIC (Application Specific Integrated Circuit). Překlad je rozdělen do tří částí.

V první fázi je z kódu v jazyce C vytvořen strom abstraktní syntaxe AST (Abstract Syntax Tree) a jsou aplikovány optimalizace nezávislé na cílové hardwarové architektuře jako rozbalení smyček a propagace konstant. V další části dochází k optimalizacím pro danou hardwarovou architekturu a nakonec je generován výstupní RTL kód.

Matlab pro popis algoritmů používá nástroj Match (MATlab Compiler for Heterogeneous computing systems) [3]. Match ze vstupního programu v Matlabu vytvoří grafy toku řízení a toku dat nad kterým provádí strojově nezávislé optimalizace. Překlad se řídí pomocí direktiv. Například pomocí direktivy TARGET se specifikuje konkrétní hardwarová platforma (DSP, embedded, FPGA) a direktivou SHAPE se určuje velikost maticových proměnných. Některé direktivy slouží i k optimalizaci, například direktivou UNROLL lze zadat úroveň rozbalení smyček a zvýšit tím paralelitu výpočtu.

Compaan (Compilation of Matlab to Process Networks) [7] je soubor nástrojů (MatParser, DgParser, Panda) pro transformaci algoritmu popsaného souborem vnořených smyček v jazyce Matlab do „Process Networks“. První část tvoří nástroj MatParser [16], který hledá datové závislosti mezi proměnnými a převádí vstupní program do speciální formy SAP (Single Assignment Program), ve které je každé proměnné přiřazena jen jedna hodnota během vykonání algoritmu. DgParser pak převádí SAP na PRDG (Polyhedral Reduced Dependence Graph), který nakonec převede systém Panda do „Process Networks“. Pomocí nástroje VHDL Visitor [12] pak lze z „Process Networks“ generovat hardwarový popis VHDL.

Kompilátor SA-C (Single Assignment C) [22] převádí vstupní program v jazyce SA-C, odvozeném z programovacího jazyka C, do VHDL. Jazyk SA-C má několik omezení oproti jazyku C, třeba nepodporuje práci s pointery a každé proměnné můžeme přiřadit hodnotu jen jednou. Na rozdíl od jazyka C umožňuje definovat libovolnou bitovou šířku proměnných a ve smyčkách lze použít jako index pole, pro paralelní operace s celým polem. Vstupní program převádí SA-C na graf toku dat, nad kterým provádí standardní optimalizace jako propagace konstant, rozbalení smyček a eliminaci společných podvýrazů.

Mezi další deriváty jazyka C patří HandelC [5] a systemC [20]. HandelC umožňuje pomocí standardních konstrukcí jazyka C a několika rozšiřujících výrazů popsat požadovanou hardwarovou strukturu. Mezi rozšíření patří například příkaz PAR pro vkládání paralelismu, možnost definovat libovolnou bitovou šířku slov a možnost popsat hardwarové struktury jako jsou paměti a sběrnice. SystemC je implementován jako C++ knihovna, poskytující konstrukce pro softwarový design a hardwarovou specifikaci.

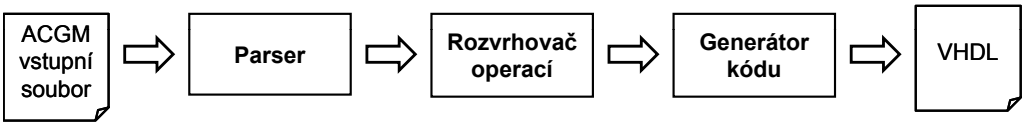
Odlišný postup návrhu poskytují nástroje SYSTEM Generátor [25] od společnosti Xilinx, který propojuje Simulink, prostředí pro návrh a simulaci systémů, s prostředky

pro automatickou implementaci na FPGA. SYSTEM Generátor umožňuje blokový návrh systému v Simulinku, jeho simulaci a automatické generování hardwarového popisu ve VHDL nebo Verilogu.

1.3 Přínos práce

Byl vyvinut nástroj ACGM (Automatic Code Generation for Matlab) pro HL syntézu algoritmů používaných pro číslicové zpracování signálů. Nástroj ACGM by měl optimalizovat dobu výpočtu DSP algoritmů, zefektivnit implementaci těchto algoritmů na programovatelné hradlové pole FPGA a hlavně zkrátit celkovou dobu návrhu. Tento nástroj generuje kód pro obvody FPGA ve VHDL a kód pro simulaci v nástroji TrueTime.

Struktura nástroje ACGM je znázorněna na obr. 1.1. Nástroj tvoří tři samostatné části: parser, rozvrhovač operací a generátor kódu. Pro zadávání vstupních dat byl navržen jazyk, jehož syntaxe vychází z jazyka Matlabu. Tím bylo umožněno ověřovat správnost výsledku navrženého DSP algoritmu jednak výpočtem v Matlabu a nebo následnou simulací s využitím nástroje TrueTime.



Obrázek 1.1: Struktura nástroje ACGM

Parser provádí syntaktickou analýzu a převádí vstupní algoritmus do grafové reprezentace. Po rozvržení s využitím Scheduling toolboxu je vstupní algoritmus uložen do formátu XML, ze kterého se pak generují požadované výstupní formáty. Generátor kódu pro TrueTime je realizován pomocí XSLT stylů. Generátor kódu pro obvody FPGA ve VHDL [18] implementoval Tomáš Novák jako samostatnou aplikaci v C#.

V následující kapitole 2 je uveden motivační příklad, na kterém je vysvětleno cyklické rozvrhování použité pro optimalizaci DSP algoritmů. Vstupní jazyk nástroje ACGM pro popis DSP algoritmů je popsán v kapitole 3. V kapitole 4 jsou uvedeny použité softwarové nástroje a technologie. Způsob implementace parseru a generátoru kódu pro TrueTime jsou popsané v kapitole 5. V kapitole 6 jsou uvedeny výsledky nástroje na několika testovacích algoritmech a v kapitole 7 jsou zhodnoceny dosažené výsledky.

Kapitola 2

Motivace

DSP algoritmy mají charakter opakující se sekvence operací, které jsou prováděny v nekonečné smyčce. Optimalizací takovýchto algoritmů se zabývá cyklické rozvrhování [11]. V této kapitole je na jednoduchém příkladě DSVF filtru [14] jednak popsáno cyklické rozvrhování a také ukázáno použití simulačního nástroje TrueTime pro simulaci DSP algoritmů.

DSVF (Digital State Variable Filter) filtr lze popsat pomocí sekvence operací sčítání, odčítání a násobení, jak je vidět na obr. 2.1. V příkladě budem uvažovat, že *processing time*, doba potřebná k naplnění jednotky, aritmetické jednotky pro sčítání a odčítání je rovna jedné a jednotky pro násobení je rovna třem a *latency*, doba potřebná na zpracování operace, je stejná jako *processing time* u obou jednotek.

```
for k=2:N
    FB{k} = F1 * B{k-1};
    L{k}  = L{k-1} + FB{k};
    QB{k} = Q1 * B{k-1};
    IL{k} = I{k} - L{k};
    H{k}  = IL{k} - QB{k};
    FH{k} = F1 * H{k};
    B{k}  = FH{k} + B{k-1};
    N{k}  = H{k} + L{k};
end
```

Obrázek 2.1: Algoritmus DSVF filtru

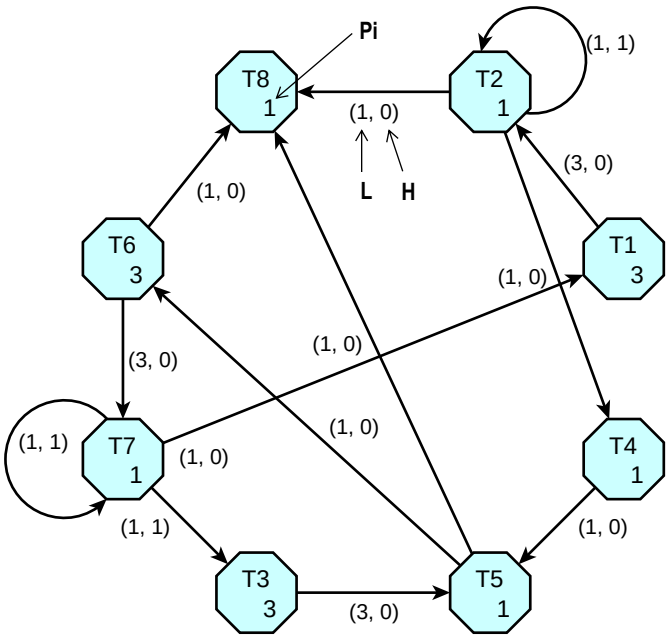
2.1 Cyklické rozvrhování

Obecně lze výpočetní smyčku popsat sekvencí operací f_i, jejichž doba vykonávání je p_i. Na algoritmus se můžeme dívat jako na množinu úloh, které je třeba rozvrhnout. Jednotlivé operace f_i označme jako úlohy T_i. Dále uvažujme, že úlohy T_i budou prováděny periodicky s periodou w. To znamená, že každá operace T_i se bude opakovat N-krát, tj. v N iteracích, které budeme značit indexem k. Potom periodický rozvrh je dán začátkem vykonávání úloh v první iteraci s_i a periodou w, tj. pro začátek vykonávání úloh v k-té iteraci $s_i(k)$ platí

$$\forall i \in T, \forall k \geq 1, s_i(k) = s_i + w \cdot (k - 1). \tag{2.1}$$

Algoritmus můžeme znázornit pomocí orientovaného grafu G. Uzly odpovídají úlohám a pokud úloha T_i předchází úloze T_j jsou propojeny orientovanou hranou e_{ij}. Hranám přiřaďme ohodnocení dvojicí čísel (L, H). Konstanta L (délka) je doba vykonávání úlohy T_i a konstanta H (výška) určuje za kolik iterací potřebuje úloha T_j výsledek úlohy T_i. Potom přípustný rozvrh s periodou w musí (pro všechny hrany grafu G) vyhovět omezení

$$\forall k \geq 1, s_i(k) + L(e_{ij}) \leq s_j(k + H(e_{ij})). \tag{2.2}$$



Obrázek 2.2: Graf algoritmu DSVF

Graf odpovídající algoritmu DSVF je znázorněn na obr. 2.2. Důležitým parametrem celého rozvrhu je perioda w. Spodní mez periody w je dána tzv. *kritickým cyklem* grafu G. Kritický cyklus odpovídá časově nejnáročnější výpočetní smyčce v algoritmu. Jeho délkou je dána minimální přípustná perioda w rozvrhu. Graf na obr. 2.2 má kritický cyklus c = (T1; T2; T4; T5; T6; T7) s kritickou periodou $w(G) = 10$. Cílem cyklického rozvrhování je najít rozvrh s minimální periodou. V případě základního cyklického rozvrhování je každá úloha přiřazena na samostatný procesor a tudíž nemůže docházet ke konfliktům mezi úlohami. Proto optimální rozvrh bude perioda w dánu kritickým cyklem a jediné čemu úlohy musí vyhovět je podmínka (2.2).

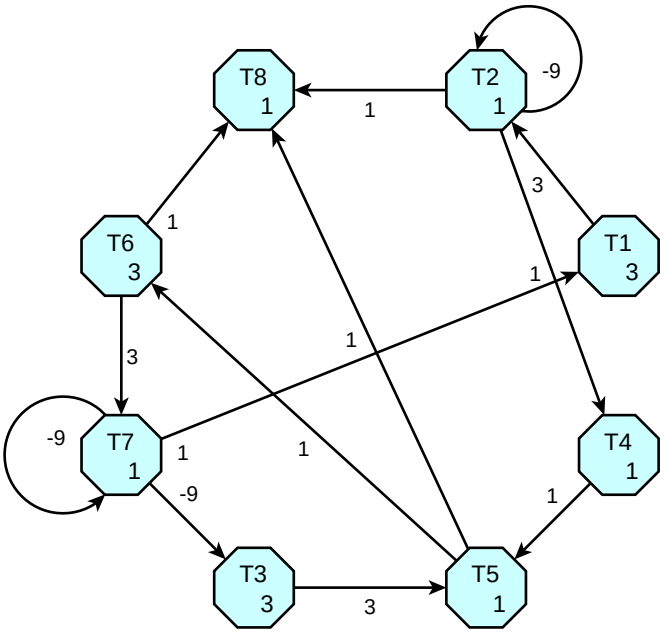
Dosazením vztahu (2.1) do podmínky (2.2) dostaneme:

$$s_i - s_j \geq L(e_{ij}) - w \cdot H(e_{ij})). \tag{2.3}$$

Protože pravá strana je konstantní, označme

$$a(e_{ij}) = L(e_{ij}) - w \cdot H(e_{ij})) \tag{2.4}$$

a vytvoříme nový graf (G,a_w), kde hrany e_{ij} budou ohodnoceny $a(e_{ij})$.



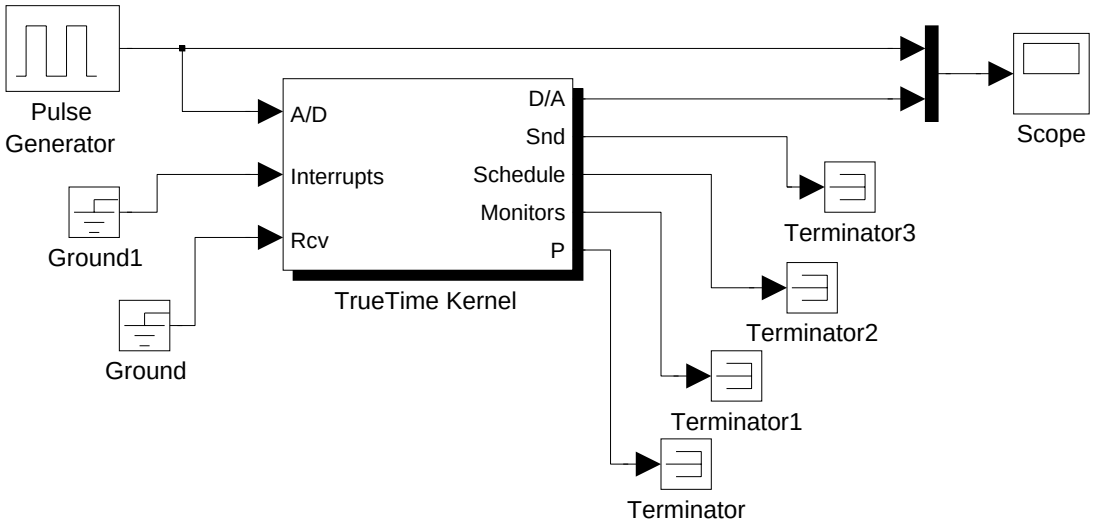
Obrázek 2.3: (G,a_w) Graf algoritmu DSVF

Pokud máme k dispozici tolik procesorů, kolik je úloh, optimální rozvrh získáme

z grafu (G,a_w) například výpočtem nejdelších cest v grafu. To konkrétně tak, že každou úlohu přiřadíme na odpovídající procesor v čase, který je dán výpočtem nejdelší cesty v grafu (G,a_w) z libovolné pevně zvolené referenční úlohy. Tím vyhovíme podmínce (2.3) pro všechny hrany e_{ij} grafu.

2.2 Simulace v nástroji TrueTime

TrueTime [19] je simulační nástroj reálného času pro Matlab/Simulink. Pomocí blokového simulačního schématu na obr. 2.4 lze simulovat chování navrženého DSVF filtru v reálném čase. Blok *TrueTime kernel* vykonává uživatelský kód, který reprezentuje jednotlivé operace filtru. Kromě uživatelského kódu je ještě nutné napsat inicializační kód, kde se nastavují parametry simulace a inicializují jednotlivé proměnné. Uživatelský kód stejně jako inicializační kód může být napsaný buď jako m-file v Matlabu, a nebo v jazyce C++. Uživatelský a inicializační kód pro příklad DSVF filtru jsou uvedeny v příloze B a výsledky simulace jsou znázorněny na obr. 6.1 v kapitole 6.



Obrázek 2.4: Simulační schema

Kapitola 3

Jazyk pro popis DSP algoritmů

Tato kapitola popisuje vstupní jazyk nástroje ACGM pro popis DSP algoritmů. Ten byl navržen tak, aby byl kompatibilní se zápisem v Matlabu, což umožňuje ověřit navržený agoritmus výpočtem v Matlabu. Jazyk umožňuje definovat numerický formát (integer, fixpoint, floatingpoint), typ a parametry aritmetických jednotek a umístění dat (registr nebo paměť). Algoritmy se zapisují pomocí konečné nebo nekonečné *for* smyčky. Uvnitř smyčky mohou být skalární operace nebo operace s maticemi. Maticové operace se specifikují pomocí maker.

Vstupní specifikace algoritmu začíná hlavičkou a skládá ze sedmi částí, mezi nimiž nejsou žádné oddělovací znaky a souvisle na sebe navazují. Pořadí všech částí je pevně stanoveno a nesmí se porušit.

1. Definice numerického formátu.
2. Definice aritmetických jednotek.
3. Definice paměťových jednotek.
4. Definice proměnných.
5. Doplňující informace.
6. Popis iterativního algoritmu.
7. Definice funkcí a maker.

Hlavička slouží k definování vstupních a výstupních proměnných. Tvar hlavičky je následující:

function [výstupní proměnné] = jméno (vstupní proměnné)

Komentáře se mohou vyskytnout v jakékoliv části vstupní specifikace algoritmu, jsou uvozeny znakem '%' a pokračují až do konce řádku.

3.1 Definice numerického formátu

Definice numerického formátu začíná klíčovým slovem *struct*, po něm následuje samotná definice, která je uzavřena do kulatých závorek a ukončena středníkem.

struct ('*datatype*', 'hodnota', '*datawidth*', hodnota);

Zvýrazněná slova jsou slova klíčová, po nich se definuje vlastní hodnota. Datový typ se definuje za klíčovým slovem *datatype*, možné hodnoty jsou: integer, fixpoint a floatingpoint. Po klíčovém slově *datawidth* se definuje datová šířka. U všech slov uvnitř definice jsou povinné uvozovky, číselné hodnoty se píší bez uvozovek.

Příklad definice numerického formátu:

struct ('*datatype*', 'integer', '*datawidth*', 32);

3.2 Definice aritmetických jednotek

V této části jsou definovány použité výpočetní jednotky. Aritmetické jednotky se definují pomocí klíčového slova *struct* a pomocí následujících klíčových slov:

operator	identifikátor operace
number	počet dostupných jednotek daného typu
proctime	doba potřebná k naplnění jednotky
latency	doba potřebná na zpracování operace
feedoper	identifikátor funkce provádějící plnění jednotky, který bude vygenerovaný do výstupního kódu (např. ve VHDL)
getoper	identifikátor funkce provádějící čtení výsledku, který bude vygenerovaný do výstupního kódu (např. ve VHDL)

Příklad definice aritmetické jednotky:

struct ('*operator*', '+', '*number*', 1,'*proctime*', 1,'*latency*', 9,'*feedoper*', 'add',...
'*getoper*', 'add_out');

3.3 Definice paměťových jednotek

Tato část slouží ke specifikaci, zda je proměnná uložena v blokové paměti nebo v registru. Definice paměťových jednotek má stejnou strukturu jako definice aritmetických jednotek, tedy začíná klíčovým slovem *struct*, je uzavřena do kulatých závorek a ukončena středníkem.

struct ('*memory*','*register*','*var*', 'proměnná');

Klíčové slovo *memory* udává, že jde o definici paměťové jednotky, *register* označuje typ paměti a za klíčovým slovem *var* se specifikuje název proměnné uložené v registru. Pokud je proměnných uložených v registrech více, lze je definovat najednou pomocí složených závorek {'proměnná', 'proměnná',...}.

Definice blokové paměti vypadá následovně:

struct ('*memory*','*bram*','*var*', 'proměnná','*ports*', hodnota);

Bloková paměť je označena klíčovým slovem *bram*. Proměnné, které mají být uloženy v blokové paměti, se definují za klíčovým slovem *var* do složených závorek, pokud je jen jedna, nemusí být ohraničena závorkami. Počet portů blokové paměti se specifikuje po klíčovém slově *ports*. U všech slov uvnitř definice jsou povinné uvozovky, číselné hodnoty se píší bez uvozovek.

Příklad definice paměťové jednotky:

struct ('*memory*','*bram*','*var*', {'a', 'b' ,'c' , 'd'},'*ports*', 2);

3.4 Definice proměnných

Názvy proměnných a konstant jsou tvořeny identifikátorem, posloupností písmen a číslic, dovoleno je i použití podtržítka, ale první znak musí být písmeno. Jako identifikátor nesmí být použito žádné z klíčovýh slov – viz odstavec 3.8.

V definici proměnných se specifikuje velikost proměnných a konstant a také jejich inicializační hodnota. Vstupní a výstupní proměnné se definují v hlavičce vstupní specifikace algoritmu. Umístění jednotlivých proměnných se specifikuje v části definice paměťových jednotek.

Proměnné a konstanty se definují pomocí operátoru přiřazení '=' následovně:

Identifikátor konstanty = hodnota konstanty;
Identifikátor proměnné{1} = inicializační hodnota proměnné;

Proměnné mají na rozdíl od konstant k identifikátoru připojen operátor {*iterační index*}.

Vícerozměrné proměnné se definují buď pomocí funkcí zeros(M,N), *ones*(M,N), *rand*(M,N) a *eye*(M,N), kde M a N udávají rozměry proměnné, a nebo pomocí inicializace v hranatých závorkách, kde sloupce se oddělují čárkou a řádky se oddělují středníkem.

Příklady definic:

a{1} = *rand*(2);
B{1} = 3**ones*(3,3);
P = [3,3;3,3];

Kromě příkazů *zeros*, *ones*, *rand*, *eye* lze použít příkaz *num2cell*, tento příkaz slouží jen pro inicializaci více hodnot proměnných pro simulaci a je parserem ignorován.

Příklady použití příkazu num2cell:

a = *num2cell*(1,2,3,4,5);
b = *num2cell*((1,2,3),(0,0,0),(1,1,1));

3.5 Doplňující informace

Tato část slouží k definování frekvence simulace pro nástroj TrueTime. Frekvence se definuje pomocí klíčových slov *struct* a *frequency*.

Příklad definice frekvence pro simulaci:

struct ('*frequency*', 1000);

3.6 Popis algoritmu

Iterativní algoritmus je uvozen smyčkou *for* ... *end*. Každý řádek odpovídá jedné elementární operaci, která je zadána buď pomocí binárního operátoru (+,-,*,/) nebo jako funkční volání a operátoru přiřazení '='. Meziiterační vzdálenosti jsou specifikovány pomocí operátoru {iterační index - meziiterační vzdálenost}. Tento operátor se vkládá hned za identifikátor proměnné.

Příklad algoritmu:

```
for k=3:K-1
    a{k} = sub(X{k}, e{k-1});
    b{k} = suba({k}, g{k-1});
    d{k} = gamma1*b{k};
    e{k} = d{k}+e{k-1};
    c{k} = b{k}+e{k};
    f{k} = gamma2*b{k};
    g{k} = f{k}+g{k-1};
    Y{k} = sub(c{k}, {k});
end
```

3.7 Definice funkcí a maker

Funkce se definují pro simulaci v nástroji TrueTime, odpovídají deklaraci aritmetických jednotek. Funkce jsou uvozeny hlavičkou funkce, po ní následuje vlastní tělo funkce ukončené klíčovým slovem *return*. Identifikátor funkce musí být stejný s identifikátorem operace aritmetické jednotky kterou funkce deklaruje.

Příklad definice funkce:

```
function y = add (a, b)
    y=a+b;
return
```

Makra slouží ke specifikaci maticových a vektorových operací. Deklarují se stejně jako funkce, jsou uvozeny hlavičkou a ukončeny klíčovým slovem *return*. V deklaraci maker jsou povoleny jen atomické operace uvedené v deklaraci aritmetických jednotek, operace zadané pomocí binárních aritmetických operátorů (+,-,*,/) a operace nulování pomocí příkazu *zeros*. Uvnitř maker jsou povoleny vnořené smyčky. Jelikož jsou povoleny pouze atomické operace, pracuje se s jednotlivými prvky matic a vektorů, ty jsou určeny následovně:

identifikátor_matice(index + offset,index + offset)

Index je řídící proměnná některé *for* smyčky a offset je číselná hodnota. U všech proměnných uvedených v makrech musí být uveden index nebo offset a nebo jejich kombinace v kulatých závorkách, výjimkou jsou proměnné použité v operaci nulování pomocí příkazu *zeros*. U skalárních proměnných je nutné uvést offset 1 (např. a(1)).

Příklady definic maker:

- Násobení vektorů

```
function y = mulvtv(u,v)
y=zeros;
    for i=1:length(u)
        TEMP(1)=mul(u(i),v(i));
        y(1)=add(y(1),TEMP(1));
    end
return
```

- Násobení matice a vektoru

```
function y = mulmtxv(A,u)
y=zeros(size(A,1),1);
    for i=1:size(A,2)
        for j=1:size(A,1)
            TEMP(1)=mul(A(j,i),u(i));
            y(j)=add(y(j),TEMP(1));
        end
    end
return
```

- Násobení matic

```
function C = mulmtx(A,B)
    C=zeros(size(A,1),size(B,2));
    for i=1:size(A,2)
        for j=1:size(B,2)
            for k=1:size(A,1)
                TEMP(1)=mul(A(k,j),B(j,i));
                C(k,i)=add(C(k,i),TEMP(1));
            end
        end
    end
return
```

3.8 Seznam klíčových slov

Tato klíčová slova nesmí být použita jako identifikátor proměnných a funkcí.

bram	getoper	ports
datatype	input	proctime
datawidth	latency	rand
end	length	register
eye	memory	return
feedoper	number	size
for	ones	var
frequency	operator	zeros
function	output	

Tabulka 3.1: Seznam klíčových slov

Vstupní specifikace algoritmu pro příklad DSVF filtru je uvedena v příloze B.

Kapitola 4

Použité technologie

V této kapitole jsou popsány použité softwarové nástroje a technologie. Větší část této kapitoly se věnuje automatizovaným nástrojům Flex a Bison, které slouží pro konstruování překladačů. Dále je zde stručně popsán obecný značkovací jazyk XML, transformační jazyk XSLT, simulační nástroj TrueTime a TORSCHE Scheduling Toolbox pro Matlab.

4.1 Flex

Flex [2] je generátor lexikálních analyzátorů v jazyce C nebo C++ a spolupracuje s generátory syntaktických analyzátorů typu Bison (Yacc). Jedná se o software vydaný pod Obecnou veřejnou licencí GNU a je tedy dostupný zdarma. Vstupem programu Flex je textový soubor, který obsahuje popis generovaného lexikálního analyzátoru. Výstupem je soubor standardně pojmenovaný „lex.yy.c“, který obsahuje hlavní funkci yylex(). Formát vstupního souboru vypadá následovně:

definice
%%
pravidla
%%
uživatelský kód

Vstupní soubor pro Flex se skládá ze tří částí, které jsou od sebe oddělené řádky obsahujícími pouze %%.

Část definic

Tato část může obsahovat:

- Definice jmen regulárních výrazů, ty pak lze používat místo vypisování daného regulárního výrazu v části pravidel.
- Definice stavů.
- Kód v jazyce C ohraničený % a %, který bude okopírován na začátek výsledného zdrojového kódu.
- Parametry pro program Flex.

Část pravidel

Každé pravidlo se skládá ze vzoru a akce, která se má provést při nalezení daného vzoru ve vstupním textu. Vzor je regulární výraz, přehled základních regulárních výrazů je v následující tabulce.

x	odpovídá znaku 'x'
.	jakýkoliv znak kromě nového řádku
xyz	odpovídá slovu xyz
[xyz]	třída znaků, vzor odpovídá buď 'x' nebo 'y' nebo 'z'
[a-z]	třída znaků, vzor odpovídá malému písmenu abecedy
[ˆA-Z]	negace třídy znaků, vyhovuje cokoliv kromě velkých písmen
r*	libovolný počet opakování regulárního výrazu r
r+	alespoň jedno opakování regulárního výrazu r
r|s	vyhovuje buď regulární výraz r nebo s
r/s	regulární výraz r, právě tehdy když je následován výrazem s
ˆ	začátek řádku
$	konec řádku
<<EOF>>	konec souboru

Tabulka 4.1: Regulární výrazy

V případě, že analyzátor rozpozná na vstupu řetězec, který odpovídá určitému vzoru,

je následně spuštěna akce, která danému vzoru přísluší. Akce je libovolný kód v jazyce C ohraničený složenými závorkami. Akci můžeme také vynechat, pak budou vstupní řetězce odpovídající danému vzoru ignorovány. V akcích lze používat předdefinované makra, funkce a proměnné z nichž nejdůležitější jsou:

- char *yytext - ukazatel na začátek textu odpovídajícího regulárnímu výrazu,
- int yylength - délka textu odpovídajícího regulárnímu výrazu.

Část uživatelského kódu

Tato část je beze změn okopírována do výsledného zdrojového souboru, můžete sem tedy umístit těla libovolných pomocných funkcí. Tuto část lze zcela vynechat.

4.2 Bison

Bison [1] je generátor syntaktických analyzátorů. Generuje část překladače, která má za úkol provádět syntaktickou analýzu vstupního textu. Vstupem je soubor popisující gramatiku a výstupem pak analyzátor v jazyce C. Bison je schopen zpracovat téměř jakoukoliv bezkontextovou gramatiku, je optimalizován pro použití s LALR(1) gramatikami. LALR(1) zjednodušeně znamená, že musí být jednoznačně definováno, jak postupovat dále pohledem pouze o jeden token dopředu. Formát vstupního souboru vypadá následovně:

```
%{
deklarace v jazyce C
%}
deklarace pro Bison
%%
pravidla gramatiky
%%
uživatelský kód v jazyce C
```

Část deklarací v jazyce C

V první části máme možnost definovat makra a deklarovat funkce a proměnné, které poté využijeme v pravidlech. Dále lze na tomto místě připojovat hlavičkové soubory.

Část deklarací pro Bison

Tato část se využívá pro deklarace terminálních a neterminálních symbolů , definování priorit a deklaraci sémantických typů symbolů. Terminální symboly deklarujeme pomocí příkazu:

%token <typ> jméno_terminálního_symbolu.

Pro nastavení asociativity jsou k dispozici místo *%token* příkazy: *%left, %right a %nonassoc.* Pořadí deklarací jednotlivých tokenů určuje jejich vzájemnou prioritu. Tokeny uvedené na jednom řádku mají stejnou prioritu a čím výše je token uveden, tím nižší prioritu má. Neterminální symboly není třeba deklarovat, ale můžeme určit jejich typ pomocí příkazu:

%type <typ> jméno_neterminálního_symbolu.

Sémantické typy symbolů se sedeklarují následovně:

```
% union {
    int celé_číslo;
    char *řetězec;
}
```

Část gramatických pravidel

Tato část popisuje gramatiku. Pravidla gramatiky začínají neterminálním symbolem, po něm následuje dvojtečka a za ní posloupnost terminálních a neterminálních symbolů,

ze kterých první neterminální symbol skládáme. Pak následuje volitelná akce, která se provede, když dojde k použití daného pravidla a vše je ukončeno středníkem. Pokud se je možné neterminální symbol přepsat podle více pravidel, lze tato pravidla sdružit a oddělit znakem '|' následovně:

```
vyraz : CISLO '+' CISLO ';' { kód akce }
      | CISLO '*' CISLO ';' { kód akce }
;
```

Na sémantické hodnoty se odkazujeme pomocí proměnné $n, kde n označuje n-tý symbol na pravé straně pravidla a proměnná $$ uchovává sémantickou hodnotu neterminálního symbolu na levé straně pravidla.

Část uživatelského kódu

Tato část je beze změn okopírována do výsledného zdrojového souboru, můžete sem tedy umístit těla libovolných pomocných funkcí. Tuto část lze zcela vynechat.

4.3 TrueTime

TrueTime [19] je simulační nástroj řídících systémů reálného času pro Matlab. TrueTime poskytuje simulační bloky pro Simulink, kterými lze modelovat jádra reálného času, drátové a bezdrátové lokální sítě a baterie. Nejdůležitější je blok *TrueTime kernel*, ten vykonává uživatelský kód, který reprezentuje jednotlivé úlohy v systému. Tento uživatelský kód stejně jako inicializační kód, kterým se blok inicializuje, může být napsaný buď jako m-file v Matlabu, a nebo v jazyce C++. Blok *TrueTime network* podporuje modelování šesti sítí: Ethernet, CAN, TDMA, FDMA, Round Robin a přepínaný Ethernet. Pro modelování bezdrátových sítí slouží blok *TrueTime wireless network*, který podporuje protokoly: IEEE 802.11b/g a ZigBee.

4.4 TORSCHE Scheduling toolbox

TORSCHE (Time Optimisation of Resources, SCHEduling) Scheduling Toolbox [6] pro Matlab je vyvíjen na Katedře řídicí techniky FEL ČVUT a distribuován pod Obecnou veřejnou licencí GNU. TORSCHE toolbox obsahuje řadu rozvrhovacích algoritmů použitelných pro řadu optimalizačních problémů. Základní objekty používané pro formulaci optimalizačních problémů jsou *Task*, *TaskSet* a *Problem*. Objekt *Task* je datová struktura obsahující všechny parametry úlohy jako startovní čas, doba trvání, termín dokončení atd. Objekt *TaskSet* spojuje několik úloh a obsahuje další informace, například precedenční omezení. Popis samotného rozvrhovacího problému je uložen do objektu *Problem*. Vstupní data jsou typicky reprezentovány orientovaným grafem a výstupem rozvrhování je pak Gantův diagram.

4.5 XML

XML (eXtensible Markup Language) je obecný značkovací jazyk, který byl vyvinut a standardizován konsorciem W3C. Je určen především pro výměnu dat mezi aplikacemi a pro publikování dokumentů. Základními prvky jazyka XML jsou elementy, které jsou do sebe vzájemně vnořovány. Elementy jsou určeny pomocí značek, přičemž většině elementů odpovídají dvě značky – počáteční a koncová. Značky jsou uzavřeny mezi znaky '<' a '>', koncová značka má navíc na začátku znak '/'. Názvy, přípustný obsah a vzájemné vztahy elementů je možné definovat pomocí XML schémat. Dalšími prvky XML dokumentu jsou atributy. Jsou součástí počáteční značky elementu a obsahují informace vztahující se k danému elementu. Každý atribut má dvě části – název a hodnotu uzavřenou do uvozovek. Kromě elementů a atributů se mohou ještě v XML dokumentu vyskytovat komentáře, sekce CDATA a instrukce pro zpracování.

Pokud XML dokument syntakticky v pořádku, pak o takovém dokumentu říkáme, že je správně strukturovaný. Základní pravidla správně strukturovaných XML dokumentů jsou: každá počáteční značka musí mít odpovídající koncovou značku, elementy se nesmějí překrývat, musí mít právě jeden kořenový element a element nesmí mít dva atributy se stejným názvem.

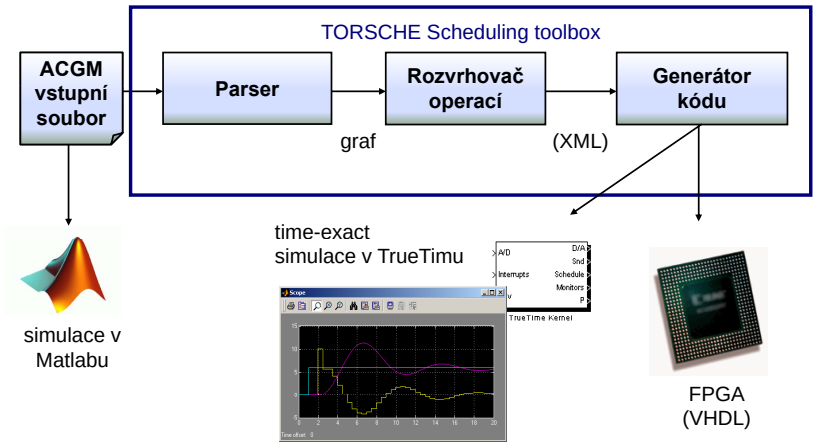
4.6 XSLT

XSLT (Extensible Stylesheet Language Transformations) je součástí specifikace XSL (Extensible Stylesheet Language), kterou vydalo konsorcium W3C. Jazyk XSLT slouží k převodům dat ve formátu XML do libovolného jiného požadovaného formátu, nejčastěji HTML, jiného XML nebo prostého textu. XSLT Transformace se provádí pomocí procesoru XSLT. K dispozici je dnes celá řada softwarových procesorů, mezi volně šířené procesory patří XT, Saxon a Xalan. Soubor s XSLT stylem je sám o sobě XML dokument, používá syntaxi XML. V dokumentu se přitom míchají dva druhy značek — řídící příkazy pro procesor a značky výsledného dokumentu. Samotný styl se skládá především ze šablon, které definují, jak se jednotlivé části XML dokumentu budou převádět. Při výběru částí dokumentu se používá jednoduchý dotazovací jazyk XPath, který je také součástí specifikace XSL.

Kapitola 5

Implementace

Nástroj ACGM (Automatic Code Generation from Matlab) se skládá ze tří částí: parser, rozvrhovač operací a generátor kódu. Syntaktickou analýzu vstupního souboru provádí parser, ten je realizovaný jako mex-file v jazyce C a při jeho implementaci byly využity nástroje Flex [2] a Bison [1]. Po syntaktické analýze je vstupní algoritmus převeden do grafové reprezentace a rozvrhovač operací nalezne optimální rozvrh. Rozvržený algoritmus je následně uložen do formátu XML. Generátor kódu pro TrueTime je realizován pomocí XSLT stylů a generátor kódu pro obvody FPGA ve VHDL je realizován jako samostatná aplikace implementovaná v jazyce C# [18].

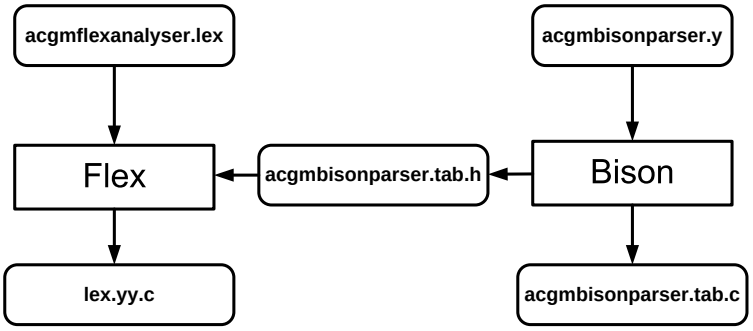


Obrázek 5.1: Struktura nástroje ACGM

Tato kapitola popisuje způsob implementace parseru a generátoru kódu pro TrueTime.

5.1 Parser

Parser je realizovaný jako samostatný modul *acgmparser.dll* pro Matlab (mex-file). Při implementaci parseru byl využit nástroj pro generování lexikálních analyzátorů Flex a nástroj pro generování syntaktických analyzátorů Bison. Na obr. 5.2 je ukázáno propojení obou nástrojů. Pro vygenerování lexikálního analyzátoru nástrojem Flex je potřeba hlavičkový soubor s definicí jednotlivých tokenů. Spolupráce vygenerovaných analyzátorů pak probíhá automaticky, syntaktický analyzátor si volá funkci *ylex()*, která vrací čísla rozpoznaných tokenů lexikálním analyzátorem.



Obrázek 5.2: Propojení nástojů Flex a Bison

Zdrojové soubory modulu *acgmparser.dll* jsou:

- *acgmparser.c* - „gateway interface“, která zajišťuje komunikaci s Matlabem
- *acgmparsersubroutines.c* - funkce pro vytváření vnitřní reprezentace
- *acgmbisonparser.tab.c* - syntaktický analyzátor
- *lex.yy.c* - lexikální analyzátor

Mex-funkce *acgmparser* se volá pomocí příkazu:

```
[H,g,u,p,l,Variables,Functions,Processors,CodeGenerationTaskParam,Frequency]=acgmparser(filename);
```

Vstupním parametrem mex-funkce *acgmparser* je název souboru se specifikací DSP algoritmu, výstupní parametry funkce jsou popsány v části 5.1.3.

5.1.1 Lexikální analyzátor

Lexikální analyzátor čte postupně znaky zdrojového programu a vytváří z nich lexikální symboly (tokeny) programu jako jsou identifikátory, čísla nebo klíčová slova. Pro každý lexikální symbol se uchovává jeho typ případně i atributy. Během analýzy vynechává lexikální analyzátor znaky, které pro program nemají význam (mezery, komentáře).

Lexikální ananalyzátor je realizovaný jako stavový automat s devíti stavy. Kromě počátečního stavu *INITIAL*, ve kterém se lexikální analyzátor nachází ihned po startu, to jsou: *mainloop*, *unitdeclaration*, *comment*, *testunitormacro*, *unitdescription*, *macrodescription*, *initvariables* a *arrayinit*. Všechny stavy jsou typu exclusive, což znamená, že pokud je analyzátor v některém stavu, vybírá pouze mezi pravidly, které jsou podmíněné daným stavem. V pravidlech pro všechny stavy se využívají definované regularní výrazy viz tabulka 5.1.

DIGIT	$[0-9]$
ID	[a-zA-Z]([a-zA-Z0-9]|"_")*
WHITE	[\n\t\r\]

Tabulka 5.1: Definované regulární výrazy použité v pravidlech

Po rozpoznání klíčového slova *for* přechází lexikální analyzátor z počátečního stavu do stavu *mainloop*, ze kterého se vrací po rozpoznání klíčového slova *end* viz tabulka 5.2.

end	přejdi do počátečního stavu
" = "	vrať token ASSGNOP
{DIGIT}+	vrať token NUMVALUE
{ID}	vrať token IDENTIFIER
" * "|"/"	vrať token OPERATORSIGN
" + "|" − "	vrať token PLUSMINUS
{WHITE}+	bíle znaky ignoruj
%	přejdi do stavu *comment*
.	vrať libovolný znak

Tabulka 5.2: Pravidla pro stav mainloop

Do stavu *unitdeclaration* přehází lexikální analyzátor po rozpoznání klíčového slova *struct*. V následující tabulce jsou uvedeny pravidla pro tento stav.

";"	přejdi do počátečního stavu
operator	vrať token OPERATOR
number	vrať token NUMBER
proctime	vrať token PROCTIME
latency	vrať token LATENCY
feedoper	vrať token FEEDOPER
getoper	vrať token GETOPER
memory	vrať token MEMORY
bram	vrať token BRAM
var	vrať token VAR
ports	vrať token PORTS
register	vrať token REGISTER
{DIGIT}+	vrať token NUMVALUE
{ID}	vrať token IDENTIFIER
"+"|"−"|"∗"|"/"	vrať token OPERATORSIGN
{WHITE}+	bíle znaky ignoruj
.	vrať libovolný znak

Tabulka 5.3: Pravidla pro stav unitdeclaration

Po rozpoznání klíčového slova *function* přejde stavový automat do stavu *testunitormacro*, ten slouží k testování, jestli jde o funkci popisující aritmetické jednotky a nebo jde o makro popisující maticové operace. Pokud se jméno dané funkce nalézá v seznamu aritmetických jednotek, stavový automat přejde do stavu *unitdescription*, jinak přejde do stavu *macrodescription*.

(return)(.∗)(\n)	přejdi do počátečního stavu
(.∗).$	vrať token LINE
{WHITE}+	bíle znaky ignoruj

Tabulka 5.4: Pravidla pro stav unitdescription

return	vrať token RETURN
for	vrať token FOR
end	vrať token END
length	vrať token LENGTH
size	vrať token SIZE
zeros(.∗)	vrať token ZEROS
{DIGIT}+	vrať token NUM
{ID}	vrať token ID
” ∗ ”|” / ”	vrať token OPERATORSIGN
” + ”|” − ”	vrať token PLUSMINUS
{WHITE}+	bíle znaky ignoruj
%	přejdi do stavu *comment*
.	vrať libovolný znak

Tabulka 5.5: Pravidla pro stav macrodescription

Z počátečního stavu může ještě stavový automat přejít do stavu *initvariables* a to pokud rozpozná identifikátor s nebo bez složených závorek následovaný znakem rovnítka.

”;”	přejdi do počátečního stavu
”[”	přejdi do stavu arrayinit
num2cell(.∗)(”;”)	ignoruj
ones	vrať token ONES
zeros	vrať token ZEROS
rand	vrať token RAND
eye	vrať token EYE
(” − ”)?[\t] ∗ [0 − 9.]+	vrať token DOUBLEVALUE
{ID}	vrať token IDENTIFIER
””. ∗ ””	ignoruj
{WHITE}+	bíle znaky ignoruj
%	přejdi do stavu *comment*
.	libovolný znak ignoruj

Tabulka 5.6: Pravidla pro stav initvariables

Po rozpoznání znaku ”[” stavový automat přechází ze stavu *initvariables* do stavu

arrayinit.

; ?[\t] ∗ ”]”	vtať token ENDOFARRAY
(” − ”)?[\t] ∗ [0 − 9.]+	vrať token DOUBLEVALUE
”;”	inkrementuj počet řádků matice
{WHITE}+	bíle znaky ignoruj
%	přejdi do stavu *comment*
.	libovolný znak ignoruj

Tabulka 5.7: Pravidla pro stav arrayinit

Ve stavu *comment* jsou ignorovány všechny znaky až do konce řádku, pak se stavový automat vrací do předchozího stavu.

5.1.2 Syntaktický analyzátor

Syntaktický analyzátor vygenerovaný pomocí nástroje Bison provádí samotnou analýzu vstupního jazyka. Lexikální analyzátor posílá syntaktickému analyzátoru terminální symboly (tokeny) a ten je ukládá na zásobník i s jejich sémantickými hodnotami. Operace uložení tokenu na zásobník se nazývá přesun. Pokud několik posledních tokenů či neterminálních symbolů v zásobníku odpovídá pravé straně některého z pravidel gramatiky, jsou následně na zásobníku nahrazeny levou stranou odpovídajícího pravidla. Tato operace se nazývá redukce. Spolu s redukcí se zároveň provádí akce odpovídajícího pravidla. Pomocí operací přesun a redukce se analyzátor snaží zredukovat celý vstup na jediný neterminální symbol, který se označuje jako startovací symbol gramatiky.

Následující gramatika popisuje vstupní jazyk pro nástroj ACGM, startovací symbol je specification.

specification	⟶	mainfunction datatypedefinition aritmeticunits memoryunits variablesdefinition simulationdata mainloop functions
mainfunction	⟶	FUNCTION STRING
	⟶	FUNCTION STRING = STRING (strings)
	⟶	FUNCTION [strings] = STRING (strings)

strings	$\longrightarrow$	
	$\longrightarrow$	STRING
	$\longrightarrow$	strings , STRING
datatypedefinition	$\longrightarrow$	STRUCT (' DATATYPE ' , ' IDENTIFIER ' , 'DATAWIDTH ' , NUMVALUE)
aritmeticunits	$\longrightarrow$	aritmeticunit
	$\longrightarrow$	aritmeticunits aritmeticunit
aritmeticunit	$\longrightarrow$	STRUCT (operator number proctime latency feedoper getoper ')'
operator	$\longrightarrow$	' OPERATOR ' , ' IDENTIFIER ' ,
	$\longrightarrow$	' OPERATOR ' , ' OPERATORSIGN ' ,
number	$\longrightarrow$	' NUMBER ' , NUMVALUE ,
proctime	$\longrightarrow$	' PROCTIME ' , NUMVALUE ,
latency	$\longrightarrow$	' LATENCY ' , NUMVALUE ,
feedoper	$\longrightarrow$	' FEEDOPER ' , ' IDENTIFIER ' ,
getoper	$\longrightarrow$	' GETOPER ' , ' IDENTIFIER '
memoryunits	$\longrightarrow$	memoryunit
	$\longrightarrow$	memoryunits memoryunit
memoryunit	$\longrightarrow$	STRUCT (' MEMORY ' , ' BRAM ' , ' VAR' , variables , ' PORTS ' , NUMVALUE)
	$\longrightarrow$	STRUCT (' MEMORY ' , ' REGISTER ' , ' VAR' , variables)
variables	$\longrightarrow$	variable
	$\longrightarrow$	variables , variable

	$\longrightarrow$	{ variables }
variable	$\longrightarrow$	' IDENTIFIER '
variablesdefinition	$\longrightarrow$	VARIABLE
	$\longrightarrow$	variablesdefinition VARIABLE
	$\longrightarrow$	CONSTANT
	$\longrightarrow$	variablesdefinition CONSTANT
	$\longrightarrow$	CONSTANT DOUBLEVALUE
	$\longrightarrow$	variablesdefinition CONSTANT DOUBLEVALUE
	$\longrightarrow$	VARIABLE DOUBLEVALUE
	$\longrightarrow$	variablesdefinition VARIABLE DOUBLEVALUE
	$\longrightarrow$	VARIABLE array
	$\longrightarrow$	variablesdefinition VARIABLE array
	$\longrightarrow$	CONSTANT array
	$\longrightarrow$	variablesdefinition CONSTANT array
array	$\longrightarrow$	arrayvalues ENDOFARRAY
	$\longrightarrow$	ONES arraysizedefinition
	$\longrightarrow$	ZEROS arraysizedefinition
	$\longrightarrow$	EYE arraysizedefinition
	$\longrightarrow$	RAND arraysizedefinition
	$\longrightarrow$	DOUBLEVALUE ONES arraysizedefinition
	$\longrightarrow$	DOUBLEVALUE ZEROS arraysizedefinition
	$\longrightarrow$	DOUBLEVALUE EYE arraysizedefinition
	$\longrightarrow$	DOUBLEVALUE RAND arraysizedefinition
arraysizedefinition	$\longrightarrow$	IDENTIFIER
	$\longrightarrow$	IDENTIFIER IDENTIFIER
	$\longrightarrow$	DOUBLEVALUE
	$\longrightarrow$	DOUBLEVALUE DOUBLEVALUE
	$\longrightarrow$	IDENTIFIER DOUBLEVALUE
	$\longrightarrow$	DOUBLEVALUE IDENTIFIER
arrayvalues	$\longrightarrow$	STARTOFARRAY DOUBLEVALUE

	$\longrightarrow$	arrayvalues DOUBLEVALUE
simulationdata	$\longrightarrow$	
	$\longrightarrow$	DEFINITION ' FREQUENCY ' , NUMVALUE)
mainloop	$\longrightarrow$	IDENTIFIER ASSGNOP NUMVALUE : IDENTIFIER tasks
	$\longrightarrow$	IDENTIFIER ASSGNOP NUMVALUE : IDENTIFIER PLUSMINUS NUMVALUE tasks
tasks	$\longrightarrow$	task
	$\longrightarrow$	tasks task
task	$\longrightarrow$	operand ASSGNOP IDENTIFIER (listofoperands) ;
	$\longrightarrow$	operand ASSGNOP operand ;
	$\longrightarrow$	operand ASSGNOP operand OPERATORSIGN operand ;
	$\longrightarrow$	operand ASSGNOP operand operand ;
listofoperands	$\longrightarrow$	operand
	$\longrightarrow$	listofoperands , operand
operand	$\longrightarrow$	IDENTIFIER
	$\longrightarrow$	IDENTIFIER brackets
	$\longrightarrow$	PLUSMINUS IDENTIFIER
	$\longrightarrow$	PLUSMINUS IDENTIFIER brackets
brackets	$\longrightarrow$	{ IDENTIFIER }
	$\longrightarrow$	{ IDENTIFIER PLUSMINUS NUMVALUE }
functions	$\longrightarrow$	
	$\longrightarrow$	function
	$\longrightarrow$	functions function
function	$\longrightarrow$	functionheader functionbody RETURN
functionheader	$\longrightarrow$	FUNCTION STRING = STRING (strings)

functionbody	$\longrightarrow$	LINE
	$\longrightarrow$	functionbody LINE
	$\longrightarrow$	macrodescription
macrodescription	$\longrightarrow$	macrofractions
macrofractions	$\longrightarrow$	macrofraction
	$\longrightarrow$	macrofractions macrofraction
macrofraction	$\longrightarrow$	forloop
	$\longrightarrow$	macrotask
forloop	$\longrightarrow$	fordescription macrofractions END
fordescription	$\longrightarrow$	FOR ID = NUM : NUM
	$\longrightarrow$	FOR ID = NUM : NUM ;
	$\longrightarrow$	FOR ID = NUM : LENGTH (ID)
	$\longrightarrow$	FOR ID = NUM : LENGTH (ID) ;
	$\longrightarrow$	FOR ID = NUM : SIZE (ID , NUM)
	$\longrightarrow$	FOR ID = NUM : SIZE (ID , NUM) ;
macrotask	$\longrightarrow$	macrooperand = macrooperand ;
	$\longrightarrow$	macrooperand = macrooperand OPERATORSIGN macrooperand ;
	$\longrightarrow$	macrooperand = macrooperand macrooperand ;
	$\longrightarrow$	macrooperand = ID (macrooperands) ;
	$\longrightarrow$	macrooperand = ZEROS
	$\longrightarrow$	ID = ZEROS
macrooperands	$\longrightarrow$	macrooperand
	$\longrightarrow$	macrooperands , macrooperand
macrooperand	$\longrightarrow$	ID memoryindex
	$\longrightarrow$	PLUSMINUS ID memoryindex

memoryindex	$\longrightarrow$	(ID)
	$\longrightarrow$	(ID PLUSMINUS NUM)
	$\longrightarrow$	(NUM)
	$\longrightarrow$	(ID , ID)
	$\longrightarrow$	(ID PLUSMINUS NUM , ID)
	$\longrightarrow$	(NUM , ID)
	$\longrightarrow$	(ID , ID PLUSMINUS NUM)
	$\longrightarrow$	(ID , NUM)
	$\longrightarrow$	(ID PLUSMINUS NUM , ID PLUSMINUS NUM)
	$\longrightarrow$	(NUM , NUM)

Syntaktický analyzátor tokeny neukladá na zásobník ihned po jejich přečtení. Token se nejdříve stane sledovaným tokenem (look-ahead token) a prozatím se na zásobník neumístí. Možné redukce symbolů se provádějí podle typu tohoto sledovaného tokenu. Pokud nelze splnit žádné pravidlo gramatiky analyzátor nahlásí syntaktickou chybu. Na obr. 5.3 je ukázáno chybové hlášení.

```
Syntax error on line     45
    L{k}  = L{k-1 + FB{k};
..................^...........................
syntax error, unexpected "'+' or '-'", expecting '}'
```

Obrázek 5.3: Ukázka chybového hlášení

Chybové hlášení udává typ chybného tokenu a možné správné typy tokenů, které syntaktický analyzátor očekává. Chybové hlášení dále udává místo výskytu dané chyby ve zdrojovém souboru. Při lexikální analýze se uchovává pozice posledního tokenu a ta je v případě syntaktické chyby zobrazena.

Po nalezení syntaktické chyby parser ukončuje svou činnost, tu ukončuje i po nalezení některé sémantické chyby ve vstupní specifikaci algoritmu, jako je např. použití nedefinované aritmetické jednotky.

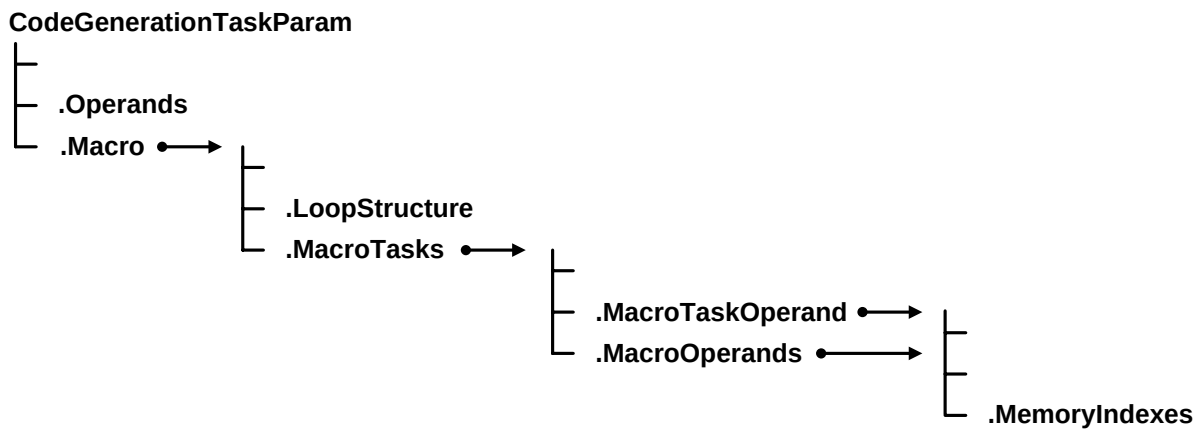
5.1.3 Rozhranní do Matlabu

Výstupem parseru je matice sousednosti H popisující algoritmus, graf g popisující algoritmus, vektor u přiřazení operací na aritmetické jednotky, vektor p dob vykonávání jednotlivých aritmetických jednotek, vektor l zpoždění jednotlivých aritmetických jednotek, pole struktur $Variables$ popisujících proměnné, pole struktur $Functions$ popisujících funkce, pole struktur $Processors$ popisujících aritmetické jednotky, pole struktur $CodeGenerationTaskParam$ popisujících jednotlivé operace a frekvence pro simulaci v nástroji TrueTime je v proměnné $SimulationFrequency$.

Struktura $CodeGenerationTaskParam$ má následující položky:

- TaskOperator - char - identifikátor operace
- TaskOutputIdentifier - char - identifikátor výstupního operandu
- IsMacro - logical - udává jestli jde o skalární operaci (false), nebo o operaci popsanou makrem (true)
- Operands - struct - pole struktur operandů
- Macro - struct - struktura popisující makro

Struktura $CodeGenerationTaskParam$ obsahuje několik vnořených struktur, na obr. 5.4 je znázorněná jejich hierarchie.



Obrázek 5.4: Hierarchie vnořených struktur v CodeGenarationTaskParam

Struktura *.Operands* má položky:

- Name - char - identifikátor operandu
- Sign - char - znaménko operandu

Struktura *.Macro* má položky:

- nLoops - double - počet smyček v makru
- nTasks - double - počet operací v makru
- LoopStructure - struct - pole struktur popisující jednotlivé smyčky v makru
- MakroTasks - sruct - pole struktur popisující jednotlivé operace v makru

Struktura *.LoopStructure* má položky:

- From - double - spodní mez řídící proměnné smyčky
- To - double - horní mez řídící proměnné smyčky
- Index - char - řídící proměnná smyčky
- Separator - logical - informace pro rozvrhovač
- NastedLevel - double - stupeň zanoření smyčky - od 0
- LoopID - double - identifikátor smyčky - pořadí smyčky v makru

Struktura *.MacroTasks* popisující operace v makru má položky:

- MacroTaskName - char - jméno operace (např. T1.1)
- MacroOperator - char - identifikátor operace
- NestedLevel - double - stupeň zanoření
- LoopID - char - identifikátor smyčky ve které se operace nachází je
- nMacroOperands - double - počet operandů
- MacroTaskOperand - struct - struktura popisující levý(výstupní operand)
- MacroOperands - struct - pole struktur popisující operandy

Strukury *.MacroTaskOperand* a *.MacroOperands* popisující operandy mají položky:

- Name - char - identifikátor operandu
- Sign - char - znaménko operandu
- nMemoryIndexes - double - počet paměťových indexů
- MemoryIndexes - struct - pole struktur popisující paměťové indexy

Struktura *.MemoryIndexes* popisující paměťové indexy má položky:

- Name - char - identifikátor indexu
- Offset - double - offset indexu

Struktura popisujících proměnné *Variables* má položky:

- Name - char - jméno proměnné
- Type - double - typ proměnné
  možné hodnoty: memory, input, output, constant
- Location - char - umístění proměnné
  možné hodnoty: BRAM, register
- InitValue - double - počáteční hodnota
- Rows - double - počet řádků
- Columns - double - počet sloupců

Struktura popisujících aritmetické jednotky *Processors* má položky:

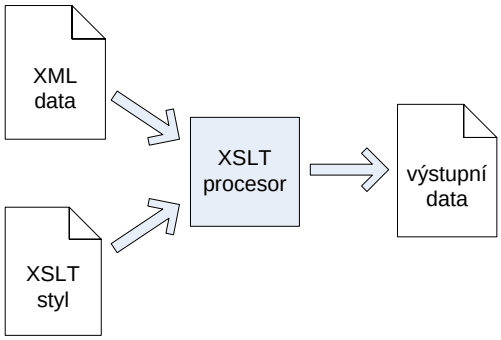
- Operator - char - identifikátor operace
- Number - double - počet dostupných jednotek daného typu
- FeedOper - char - funkce provádějící plnění jednotky
- GetOper - char - funkce provádějící čtení výsledku
- ProcTime - double - doba potřebná k naplnění jednotky
- Latency - double - doba potřebná na zpracování operace

Struktura popisující funkce *Functions* má položky:

- Name - char - identifikátor funkce
- OutputIdentifier - char - identifikátor výstupního operandu funkce
- Operands - struct - pole struktur s jednou položkou Name (jméno operandu)
- Lines - struct - pole struktur s jednou položkou Line (řádek funkce)

5.2 Generátor kódu pro TrueTime

Generátor kódu pro nástroj TrueTime je implementován pouze pro jednodušší DSP algoritmy bez maticových operací. Samotné generování kódu pro nástroj TrueTime je realizováno pomocí transformačního jazyka XSLT. Princip transformace je znázorněn na obr. 5.5. Transformaci provádí XSLT procesor, jehož vstupem je XSLT styl společně se zdrojovými XML daty a výstupem jsou data v předepsaném formátu.



Obrázek 5.5: Princip XSLT transformace

XSLT styl se skládá především ze šablon, které definují, jak se jednotlivé části XML dokumentu budou převádět. Tvar šablon je následující:

```
<xsl:template match = "výraz">
    tělo šablony
</xsl:template>
```

Jak transformovat definuje tělo šablony a pomocí výrazu v dotazovacím jazyce XPath se specifikuje část dokumentu, která se má transformovat.

XSLT procesor pak pracuje tak, že si načte zdrojový XML dokument a vytvoří si jeho stromovou reprezentaci. Tento strom pak postupně prochází od kořene v pořadí v jakém jsou elementy obsaženy v dokumentu. V okamžiku, kdy je nalezena šablona odpovídající uzlu ve stromu, začne se její obsah zpracovávat.

Nástoj TrueTime poskytuje simulační blok *TrueTime kernel*, pomocí kterého lze simulovat chování DSP algoritmů v reálném čase. Pro tento blok je nutné vygenerovat jednak inicializační kód, kde se nastavují parametry simulace a inicializují proměnné, a také uživatelský kód, který bude vykonáván. Inicializační kód je generován ze zdrojového XML pomocí stylu acgmtruetimeinit a uživatelský kód se transformuje podle stylu acgmtruetime. Zdrojové XML je pro oba styly totožné a je detailně popsáno v příloze A.

5.2.1 XSLT styl acgmtruetimeinit

Simulační blok *TrueTime kernel* se inicializuje pomocí funkce *ttInitKernel*, její vstupní argumenty jsou: počet vstupních kanálů, počet výstupních kanálů a rozvrhovací strategie. Tato funkce provádí nezbytné inicializace a musí stát na začátku inicializačního skriptu. Dále je nutné vytvořit periodickou úlohu pomocí funkce *ttCreatePeriodicTask*, její vstupní argumenty jsou: jméno úlohy, offset, perioda, priorita, jméno skriptu s uživatelským kódem a jméno struktury pro uložení dat. Struktura pro uložení dat reprezentuje lokální paměť úlohy, všechny proměnné se definují jako položky této struktury. Zbytek inicializačního skriptu tvoří inicializace proměnných. Ukázka inicializačního kódu je na obr. 5.6.

Jelikož Inicializační kód pro TrueTime nemá příliš složitou a variabilní strukturu, styl obsahuje jen jednu šablonu pro kořenový element zdrojového dokumentu *matlabdata*.

```
<xsl:template match = "/matlabdata">
    tělo šablony
</xsl:template>
```

Kód je generován pomocí instrukce <xsl:text>, její obsah je text, který se beze změn zkopíruje do výstupního dokumentu. Text ze zdrojového dokumentu se vkládá pomocí instrukce <xsl:value-of>. Tato instrukce vybere obsah textových uzlů, které jsou potomky

elementu určeného výrazem zapsáného pomocí syntaxe XPath. Pomocí této instrukce se zapisují ze zdrojového dokumentu jména proměnných a argumenty funkce *ttInitKernel*, počet vstupní a výstupních kanálů. Poslední instrukce použitá v těle šablony je instrukce pro iterativní zpracování <xsl:for-each>, pomocí které se procházejí všechny proměnné.

```
function simple_init
ttInitKernel(1,1,'prioFP');% nbrOfInputs, nbrOfOutputs, fixed priority
data.frequency=1000;
data.reg1=0;
data.reg2=0;
data.reg3=0;
data.units.unit1= [0,0];
data.const1=0.01;
data.const2=0.5;
w=11;
period = w/data.frequency;
deadline = period;
offset = 0;
prio = 1;
ttCreatePeriodictask('task1', offset, period, prio, 'code', data);
```

Obrázek 5.6: Ukázka inicializačního kódu pro TrueTime

5.2.2 XSLT styl acgmtruetime

Vykonávání jednotlivých úloh definuje uživatelský kód, ten je rozdělen do několika segmentů. Jednotlivé segmenty jsou vykonávány postupně za sebou, přičemž doba vykonávání je pro všechny segmenty stejná. Ukázka uživatelského kódu pro příklad DSVF filtru je uvedena v příloze B. Ke čtení proměnných ze vstupních kanálů slouží funkce *ttAnalogIn*, jejíž argument je číslo kanálu, a pro zápis na výstupní kanál slouží funkce *ttAnalogOut*, jejíž argumenty jsou číslo výstupního kanálu a hodnota. Aritmetické jednotky s pipeliningem jsou simulovány pomocí pole. Velikost pole je dána dobou potřebnou na zpracování operace odpovídající jednotky. V průběhu každého segmentu pak dochází pomocí funkce *pipeunit* k posunu prvků v poli směrem ke konci.

Uživatelský kód pro TrueTime má složitější strukturu, styl obsahuje několik pojmeno-

vaných šablon. Tyto šablony mají navíc atribut *name* a k volání šablony slouží instrukce <xsl:call-template>. Šablonám lze předávat parametry instrukcí <xsl:with-param>.

Styl XSLT pro generování uživatelského kódu pro simulační nástroj TrueTime obsahuje následující šablony:

- *GenCase*(*End*, *Counter*) - šablona pro generování operací pro jednotlivé segmenty, parametr *End* udává celkový počet segmentů a parametr *Counter* udává pořadí aktuálního segmentu
- *WriteTasks(*Counter*)* - zapíše operace pro daný segment, parametr *Counter* udává pořadí daného segmentu
- *WritePipeUnitFunction* - generuje funkci *pipeunit*
- *FindVariable*(*MatlabName*) - generuje jméno proměnné, parametr *MatlabName* udává jméno použité ve specifikaci algoritmu
- *OutputChanel*(*MatlabName*) - zapíše číslo výstupního kanálu, parametr *MatlabName* udává jméno použité ve specifikaci algoritmu
- *TaskRHS* - zapíše jména operandů pravé stranu úlohy
- *WriteFunctions* - kopíruje funkce popisující aritmetické jednotky

Kapitola 6

Experimentální výsledky

V této kapitole jsou ukázány výsledky použití nástroje ACGM. Jako jednoduché benchmarky bez maker byly použity DSVF (Digital State Variable Filter) filtr [14], WDF (Wave Digital Filter) filtr druhého řádu [9], WDEF (Wave Digital Elliptic Filter) filtr pátého řádu [4] a diskrétní PSD regulátor [21], dále byly použity tři benchmarky s makry s jednou vnořenou smyčkou. Výsledné rozvrhy pro všechny benchmarky získané pomocí nástroje ACGM jsou uvedeny v příloze C.

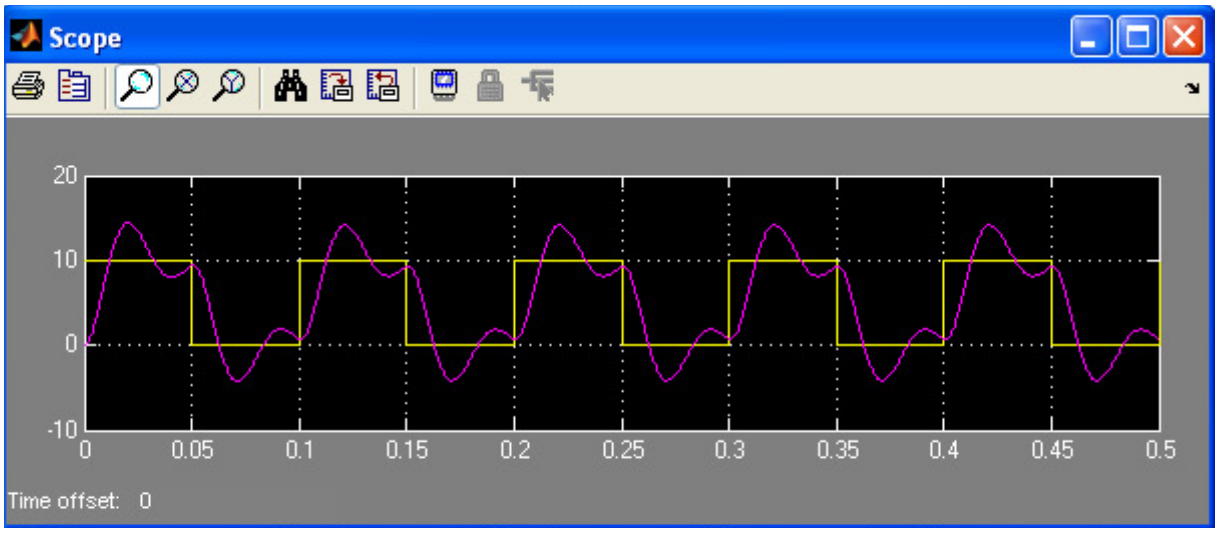
V následující části jsou uvedeny výsledky syntézy benchmarků do FPGA, pro všechny benchmarky je uvedena mezní frekvence f_{clk} stavového automatu, počet ekvivalentních hradel a hodnota LUTs (Look-up Table), která odpovídá počtu logických bloků v FPGA. Výsledky simulace pomocí nástroje TrueTime jsou ukázány na benchmarcích DSVF a PSD.

Algoritmus DSVF se skládá z 8 operací, 3 operace násobení a 5 operací sčítání nebo odčítání. V následující tabulce jsou uvedeny výsledky syntézy pro dvě realizace DSVF filtru, první s hodnotami *latency* a *processing time* sčítačky rovným jedné ($l_+ = 1, p_+ = 1$) a násobičky třem ($l_* = 3, p_* = 3$) a dále pro hodnoty odpovídající jednotkám HSLA [17] ($l_+ = 9, p_+ = 1, l_* = 2, p_* = 1$). Perioda výsledného rozvrhu pro první realizaci je $w = 11$ a pro druhou odpovídající jednotkám HSLA $w = 40$.

	eq. hradel	LUTs	f_{clk}
DSVF	2380	242	316.842MHz
DSVF HSLA	2430	214	545.926MHz

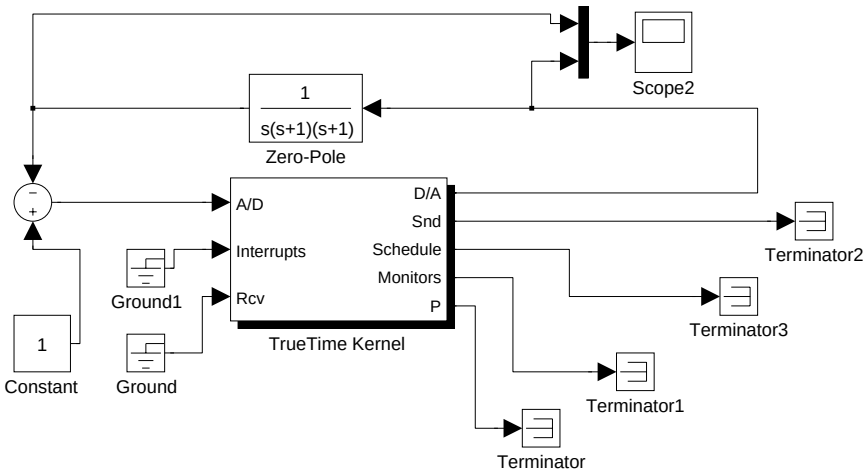
Tabulka 6.1: Výsledky syntézy DSVF filtru

Výsledky simulace DSVF filtru pomocí nástroje TrueTime jsou na obr. 6.1.



Obrázek 6.1: Simulace DSVF filtru v nástroji TrueTime

PSD regulátor obsahuje 12 operací. Parametry regulátoru jsou navrženy pro soustavu na obr. 6.2. Pro syntézu byly použity opět dvě realizace, stejné jako u DSVF filtru, výsledky syntézy viz tabulka 6.2. Perioda výsledného rozvrhu pro první realizaci je $w = 10$ a pro druhou odpovídající jednotkám HSLA $w = 24$.

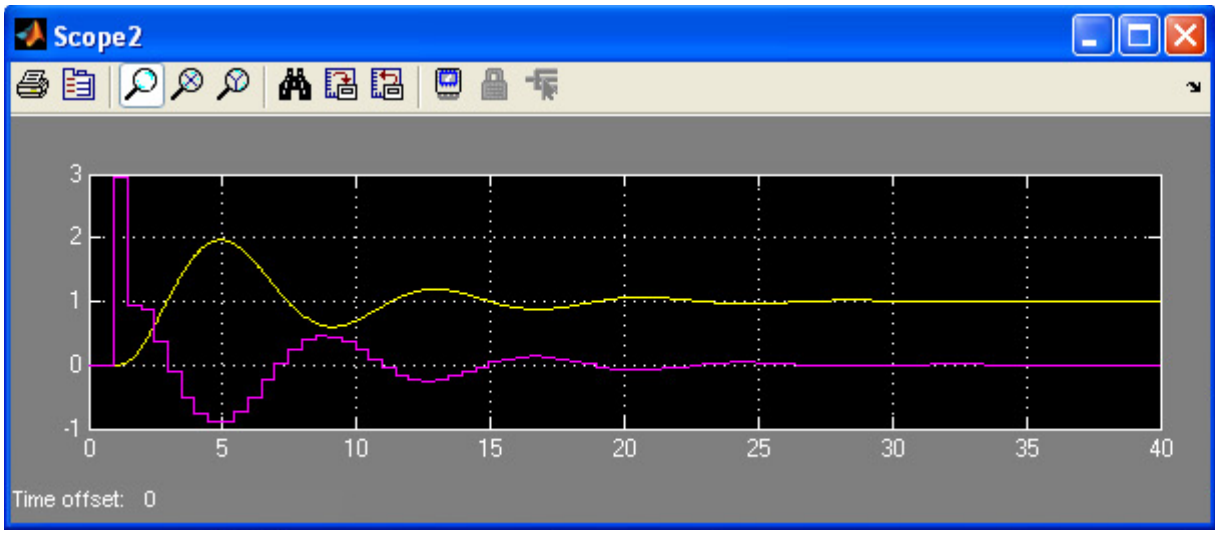


Obrázek 6.2: Simulační schema

	eq. hradel	LUTs	f_{clk}
PSD	2643	272	350.748MHz
PSD HSLA	3059	303	271.481MHz

Tabulka 6.2: Výsledky syntézy PSD regulátoru

Na obr. 6.3 je ukázáno chování navrženého regulačního obvodu pomocí nástroje TrueTime.



Obrázek 6.3: Simulace PSD regulátoru v nástroji TrueTime

Algoritmus WDF se skládá z 8 operací, 6 operací sčítání nebo odčítání a 2 operace násobení. Výsledky syntézy pro dvě stejné realizace jako v předchozích případech ukazuje tabulka 6.3. Perioda výsledného rozvrhu pro první realizaci je $w = 8$ a pro druhou odpovídající jednotkám HSLA $w = 29$.

	eq. hradel	LUTs	f_{clk}
WDF	2301	224	267.562MHz
WFD HSLA	2290	204	548.682MHz

Tabulka 6.3: Výsledky syntézy WDF filtru

Filtr WEDF pátého řádu obsahuje 34 operací, z toho 4 operace násobení a 30 operací sčítání nebo odčítání. Výsledky syntézy pro realizaci s hodnotami odpovídajícími

jednotkám HSLA ukazuje tabulka 6.4. Perioda výsledného rozvrhu je $w = 97$.

	eq. hradel	LUTs	f_{clk}
WEDF HSLA	7214	658	516.863MHz

Tabulka 6.4: Výsledky syntézy WEDF filtru

Výsledky syntézy do FPGA pro benchmarky s makry s jednou vnořenou smyčkou ukazuje tabulka 6.5. Pro tyto benchmarky byly použity hodnoty: *latency* sčítačky $l_+ = 2$, *processing time* sčítačky $p_+ = 1$, *latency* násobičky $l_* = 4$ a *processing time* násobičky $p_* = 1$.

	eq. hradel	LUTs	f_{clk}
nestedloops_benchmak1	1951	139	165.666MHz
nestedloops_benchmak2	2427	118	160.115MHz
nestedloops_benchmak3	3229	154	99.107

Tabulka 6.5: Výsledky syntézy benchmarků s makry

Kapitola 7

Závěr

Tato práce se zabývá optimalizací algoritmů používaných pro číslicové pracování signálů. Konkrétně se zabývá implementací nástroje pro automatické generování kódu pro obvody FPGA, což bylo hlavním cílem práce.

Byl vyvinut nástroj ACGM(Automatic Code Generation from Matlab), který je součástí TORSCHE Scheduling Toolboxu pro Matlab. Nástroj se skládá ze tří částí: parser, rozvrhovač operací a generátor kódu.

Těžištěm práce byl návrh způsobu zadávání vstupních dat, implementace syntaktického analyzátoru a implementace generátoru kódu pro nástroj TrueTime.

Pro zadávání vstupních dat byl navržen jazyk, jehož hlavní výhodou je kompatibilita se zápisem v Matlabu, to umožňuje návrháři ověřit správnost výsledku navrženého DSP algoritmu výpočtem v Matlabu. Navržený jazyk možňuje zadávání maticových operací pomocí maker, které jsou deklarovány jako funkce.

Parser byl realizovaný jako mex-file v jazyce C a při jeho implementaci byly využity automatizované nástroje Flex a Bison, které slouží pro konstruování lexikálních a syntaktických analyzátorů.

Generátor kódu pro TrueTime byl realizován jen pro algoritmy bez maticových operací. Generování probíhá pomocí XSLT stylů, pomocí kterých se transformuje XML, obsahující data o rozvrženém algoritmu, na výstupní kód.

Celý nástroj i s generátorem kódu pro obvody FPGA ve VHDL [18] byl testován na sedmi benchmarcích viz. kapitola 6. Nástroj dosahoval uspokojivých výsledků.

Literatura

[1] *GNU Bison user manual.* `http://www.gnu.org/software/bison/manual`.

[2] *GNU flex user manual.* `http://www.gnu.org/software/flex/manual`.

[3] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, N. Pal, S.and Tripathi, D. Zaretsky, R. Anderson, and J.R. Uribe. Overview of a compiler for synthesizing MATLAB programs onto FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 12:312–24, 2004.

[4] E. Bonsma and S. Gerez. A genetic approach to the overlapped scheduling of iterative data–flow graphs for target architectures with communication delays. In *ProRISC Workshop on Circuits, Systems and Signal Processing*, pages 75–84, November 1997.

[5] Celoxica, `http://www.celoxica.com`. *Handel-C Language Reference Manual.*

[6] P. Šůcha, M. Kutil, M. Sojka, and Z. Hanzálek. TORSCHE Scheduling Toolbox for Matlab. In *IEEE International Symposium on Computer-Aided Control Systems Design*, Munich, Germany, 2006.

[7] E.F. Deprettere, E. Rijpkema, .P Lieverse, and B. Kienhuis. High level modeling for parallel executions of nested loop algorithms. In *Proceedings IEEE International Conference on Application-specific Systems, Architectures and Processors*, page 79, Washington, DC, 2000. IEEE Computer Society.

[8] Goran Doncev, Miriam Leeser, and Shantanu Tarafdar. High level synthesis for designing custom computing hardware. In *FCCM '98: Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, page 326, Washington, DC, USA, 1998. IEEE Computer Society.

[9] A. Fettweis. Wave digital filters: theory and practice. *Proceedings of the IEEE*, 74:270–327, February 1986.

[10] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings 16th International Conference on VLSI Design concurrently with the 2nd International Conference on Embedded Systems Design*, pages 461–6, Los Alamitos, CA, 2003. IEEE Computer Society.

[11] C. Hanen and A. Munier. A study of the cyclic scheduling problem on parallel processors. *Discrete Applied Mathematics*, 57:167–192, February 1995.

[12] T. Harriss, R. Walke, B. Kienhuis, and E. Deprettere. Compilation from matlab to process networks realized in FPGA. *Design Automation for Embedded Systems*, 7:385–403, 2002.

[13] R. Helaihel and K. Olukotun. Java as a specification language for hardware-software systems. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 690 – 697, Washington, DC, 1997. IEEE Computer Society.

[14] National Instruments. Implementation of digital filters as part of custom synthesizer with ni speedy 33, `http://zone.ni.com/devzone/cda/tut/p/id/3476`.

[15] A. Jones, D. Bagchi, S. Pal, P. Banerjee, and A Choudhary. PACT HDL: a compiler targeting ASICs and FPGAs with power and performance optimizations. *The Journal of VLSI Signal Processing*, 31:127–142, 2004.

[16] B. Kienhuis. MatParser: An array dataflow analysis comp. Technical Report UCB/ERL M00/9, EECS Department, University of California, Berkeley, 2000.

[17] R. Matoušek, M. Tichý, A. Z. Pohl, J. Kadlec, and C. Softley. Logarithmic number system and floating-point arithmetics on FPGA. Field-Programable Logic and Applications: Reconfigurable computing Is Going Mainstream. Lecture notes in Computer Science A 2438, Springer, Berlin, 2002.

[18] Tomáš Novák. *Automatické generování VHDL kódu pro FPGA*. Bakalářská práce, Katedra řídicí techniky ČVUT-FEL, 2007.

[19] Martin Ohlin, Dan Henriksson, and Anton Cervin. *TrueTime 1.5—Reference Manual*, jan.

[20] The Open SystemC Initiative, `http://www.systemc.org`. *SystemC 2.1 Language Reference Manual*, 2005.

[21] Petr Pivoňka. *Vyšší formy řízení.* 2003.

[22] Robert Rinker, Margaret Carter, Amitkumar Patel, Monica Chawathe, Charlie Ross, Jeffrey Hammes, Walid A. Najjar, and Wim Bohm. An automated process for compiling dataflow graphs into reconfigurable hardware. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):130–139, 2001.

[23] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B.R. Rau, D. Cronquist, and M. Sivaraman. PICO-NPA: high-level synthesis of nonprogrammable hardware accelerators. *The Journal of VLSI Signal Processing*, 31:127–142, 2004.

[24] Byoungro So, Mary W. Hall, and Pedro C. Diniz. A compiler approach to fast hardware design space exploration in FPGA-based systems. *SIGPLAN Not.*, 37(5):165–176, 2002.

[25] Xilinx, `http://www.xilinx.com`. *System Generator for DSP v9.1 Users Guide*, 2007.

Příloha A

Strukrura XML

Tato kapitola popisuje strukturu XML rozhraní mezi rozvrhovačem operací a generátorem kódu. Popisuje názvy, význam a hodnoty jednotlivých elementů a atributů. Formát vstupního XML z Scheduling Toolboxu je navržen tak, aby umožňoval obecný popis rozvrhovacích problémů, proto obsahuje mnoho nepotřebných elementů a atributů pro generátor kódu, ty budou pro stručnost vynechány. Znak + za koncovou značkou elementů označuje možný počet opakování větší než jedna.

A.1 Element matlabdata

Element *matlabdata* je kořenovým elementem, celý XML dokument je uzavřen do tohoto elementu. Element *matlabdata* musí obsahovat alespoň jeden element *taskset*.

Syntaxe:

$< matlabdata >$
$\quad < taskset > ... < /taskset > +$
$< /matlabdata >$

A.2 Element taskset

Element *taskset* obsahuje data popisující buď množinu úloh hlavní smyčky, a nebo makro. Pokud se jedná o hlavní smyčku má atribut *id* elementu *taskset* hodnotu "sch_taskset",

jinak je jeho hodnota shodná s identifikátorem úlohy popsané makrem. V elementu *schedule* jsou obsaženy informace o rozvrhovacím algoritmu a parametry rozvrhu, z nichž je pro generování důležitá perioda hlavní smyčky, její hodnota je ve vnořeném elementu *period*. Pokud se jená o taskset popisující makro je element *schedule* vynechán. Element *task* obsahuje všechny informace o jednotlivých úlohách. Poslední důležitý element je *tsuserparam*.

Syntaxe:

```
< taskset id = "id" >
    < schedule >
        < period > ... < /period >
    < /schedule >
    < task > ... < /task > +
    < tsuserparam > ... < /tsuserparam >
< /taskset >
```

A.3 Element tsuserparam

Pokud rodičovský element *taskset* elementu *tsuserparam* popisuje úlohy v hlavní smyčce, obsahuje element *tsuserparam* data popisující použité aritmetické jednotky, všechny proměnné, funkce popisující aritmetické jednotky a frekvenci pro simulaci. Pokud rodičovský element *taskset* popisuje makro, obsahuje element *tsuserparam* popis vnořených smyček v makru.

Syntaxe:

```
< tsuserparam >
    < struct >
        < struct name = "CodeGenerationData" >
            < struct name = "Processors" > ... < /struct >
            < struct name = "Variables" > ... < /struct >
            < struct name = "Functions" > ... < /struct >
            < double name = "SimulationFrequency" >
                < item > ... < /item >
            < /double >
        < /struct >
    < /struct >
< /tsuserparam >
```

Aritmetické jednotky popisuje následující element *struct*. Obsahuje elementy *char* popisující identifikátor operace, název funkce provádějící plnění jednotky a název funkce pro čtení výsledku, dále obsahuje vnořené elementy *double* udávající dobu potřebnou k naplnění jednotky, dobu potřebnou na zpracování operace a počet dostupných jednotek daného typu.

Syntaxe:

```
< struct name = "Processors" >
    < struct >
        < char name = "Operator" > ... < /char >
        < char name = "FeedOper" > ... < /char >
        < char name = "GetOper" > ... < /char >
        < double name = "Number" >
            < item > ... < /item >
        < /double >
        < double name = "Proctime" >
            < item > ... < /item >
        < /double >
        < double name = "Latency" >
            < item > ... < /item >
        < /double >
    < /struct > +
< /struct >
```

Popis proměnných obsahuje následující element *struct*. Jeho vnořené elementy *char* obsahují název , typ a umístění proměnné. Vnořené elementy *double* obsahují údaje o velikosti proměnné a její inicializační hodnotu.

Syntaxe:

```
< struct name = "Variables" >
    < struct >
        < char name = "Name" > ... < /char >
        < char name = "Type" > ... < /char >
        < char name = "Location" > ... < /char >
        < double name = "InitValue" >
            < item > ... < /item > +
        < /double >
        < double name = "Rows" >
            < item > ... < /item >
        < /double >
        < double name = "Columns" >
```

```
                < item > ... < /item >
            < /double >
        < /struct > +
    < /struct >
```

Data popisující funkce obsahuje následující element *struct*. Vnořené elementy *char* obsahují identifikátor funkce a identifikátor výstupního operandu funkce, ve vnořených elementech *struct* jsou vstupní operandy funkce a jednotlivé řádky funkce.

Syntaxe:

```
< struct name = "Functions" >
    < struct >
        < char name = "Name" > ... < /char >
        < char name = "OutputIdentifier" > ... < /char >
        < struct name = "Operands" >
            < struct" >
                < char name = "Name" > ... < /char >
            < /struct > +
        < /struct >
        < struct name = "Lines" >
            < struct" >
                < char name = "Line" > ... < /char >
            < /struct > +
        < /struct >
    < /struct > +
< /struct >
```

Pro případ kdy element *taskset* popisuje makro, obsahuje element *tsuserparam* pouze data popisující vnořené smyčky v makru. V popisu smyčky je uveden číselný identifikátor smyčky, indexační proměnná, meze smyčky a perioda se kterou se provádějí jednotlivé úlohy uvnitř smyčky.

Syntaxe:

```
< tsuserparam >
    < struct >
        < struct name = "CodeGenerationData" >
            < struct name = "Loop" >
                < double name = "LoopID" >
                    < item > ... < /item >
                < /double >
                < char name = "Index" > ... < /char >
```

```
                    < double name = "From" >
                        < item > ... < /item >
                    < /double >
                    < double name = "To" >
                        < item > ... < /item >
                    < /double >
                    < double name = "Period" >
                        < item > ... < /item >
                    < /double >
                < /struct >
            < /struct >
        < /struct >
    < /tsuserparam >
```

A.4 Element task

Element *task* obsahuje všechny informace o dané úloze. Každá úloha má informace o rozvrhu ve vnořeném elementu *schedule*. Jméno príslušné úlohy udává element *name*. Data potřebná pro generování výstupního kódu jsou obsaženy v elementu *userparam*.

Syntaxe:

```
< task >
    < name > ... < /name >
    < schedule > ... < /schedule >
    < userparam > ... < /userparam >
< /task >
```

A.5 Element schedule

Element *schedule* obsahuje informace o rozvrhu. Pokud se jedná o úlohu v makru je obsahuje jen jeden element *item* obsahující informace o začátku vykonávání úlohy, délce vykonávání úlohy a číselný identifikátor aritmetické jednotky, kterou je úloha zpracovávána. Element *period* je vynechán, úloha je vykonávána s periodou odpovídající smyčky.

Pokud nejde o úlohu v makru element *period* obsahuje periodu hlavní smyčky a ele-

mentů *item* může být víc.

Syntaxe:

```
< schedule >
    < period > ... < /period >
    < item >
        < start > ... < /start >
        < length > ... < /length >
        < processor > ... < /processor >
    < /item > +
< /schedule >
```

A.6 Element userparam

Element *userparam* obsahuje data popisující jednotlivé úlohy, jako jsou identifikátor operace, jméno výstupního operandu a jména a znaménka pravých operandů a logickou proměnnou, udávající jestli jde o maticovou operaci.

Syntaxe:

```
< userparam >
    < struct >
        < struct name = "CodeGenerationTaskParam" >
            < char name = "TaskOperator" > ... < /char >
            < char name = "TaskOutputIdentifier" > ... < /char >
            < struct name = "Operands" >
                < struct" >
                    < char name = "Name" > ... < /char >
                    < char name = "Sign" > ... < /char >
                < /struct > +
            < /struct >
            < logical name = "IsMacro" > ... < /logical >
        < /struct >
    < /struct >
< /userparam >
```

Pokud rodičovský element *task* popisuje makro je syntaxe podobná, navíc je je uveden identifikátor smyčky, do které úloha náleží, počet pravých operandů a u všech operandů je uveden identifikátor paměťového indexu a jeho offset.

```
< userparam >
    < struct >
        < struct name = "CodeGenerationTaskParam" >
            < char name = "MacroOperator" > ... < /char >
            < double name = "LoopID" >
                < item > ... < /item >
            < /double >
            < double name = "nMacroOperands" >
                < item > ... < /item >
            < /double >
            < struct name = "MacroTaskOperand" >
                < char name = "Name" > ... < /char >
                < char name = "Sign" > ... < /char >
                < double name = "nMemoryIndexes" >
                    < item > ... < /item >
                < /double >
                < struct name = "MemoryIndexes" >
                    < char name = "Name" > ... < /char >
                    < double name = "Offset" >
                        < item > ... < /item >
                    < /double >
                < /struct > +
            < /struct >
            < struct name = "MacroOperands" >
                < struct >
                    < char name = "Name" > ... < /char >
                    < char name = "Sign" > ... < /char >
                    < double name = "nMemoryIndexes" >
                        < item > ... < /item >
                    < /double >
                    < struct name = "MemoryIndexes" >
                        < char name = "Name" > ... < /char >
                        < double name = "Offset" >
                            < item > ... < /item >
                        < /double >
                    < /struct > +
                < /struct > +
            < /struct >
        < /struct >
    < /struct >
< /userparam >
```

Příloha B

Výpisy kódu

B.1 Algoritmus DSVF

B.1.1 Specifikace DSVF algoritmu

```
function N=dsvf(I)

%Data Type definition
struct('datatype','integer','datawidth',32);

%Arithmetic Units Declaration
%Operace napr. na HSLA
struct('operator','+','number',1,'proctime',1,'latency',1,'feedoper','add','getoper','add_out');
struct('operator','*','number',1,'proctime',3,'latency',3,'feedoper','mul','getoper','mul_out');

%Memmory Units Declaration
struct('memory','bram','var',{'I','L','B','H','N','F1','Q1'},'ports',2);
struct('memory','bram','var',{'K','f','fs','Q'},'ports',2);
struct('memory','bram','var',{'FB','QB','IL','FH'},'ports',2);

%Variables Declaration
f = 50;
fs = 40000;
Q = 2;
K = 1000;
F1 = 0.0079;
Q1 = 0.5;
I{1} = ones(1,K);
L{1} = zeros(1,K);
B{1} = zeros(1,K);
H{1} = zeros(1,K);
N{1} = zeros(1,K);
FB{1} = zeros(1,K);
QB{1} = zeros(1,K);
IL{1} = zeros(1,K);
FH{1} = zeros(1,K);
```

```
    struct('frequency',220000);

    %Iterative Algorithm

    for k=2:K
        FB{k} = F1 * B{k-1};
        L{k}  = L{k-1} + FB{k};
        QB{k} = Q1 * B{k-1};
        IL{k} = I{k} - L{k};
        H{k}  = IL{k} - QB{k};
        FH{k} = F1 * H{k};
        B{k}  = FH{k} + B{k-1};
        N{k}  = H{k} + L{k};
    end
```

B.1.2 Iniciallizační kód pro TrueTime

```
function simple_init

ttInitKernel(1,1,'prioFP');% nbrOfInputs, nbrOfOutputs, fixed priority

data.frequency=220000;    %simulation frequency
data.reg1=0;    % initialization of variable L
data.reg2=0;    % initialization of variable B
data.reg3=0;    % initialization of variable H
data.reg4=0;    % initialization of variable FB
data.reg5=0;    % initialization of variable QB
data.reg6=0;    % initialization of variable IL
data.reg7=0;    % initialization of variable FH
data.reg8=0;    % initialization of variable N
data.units.unit1= [0,0];    % initialization of unit +
data.units.unit2= [0,0,0,0];    % initialization of unit *
data.const1=0.007900000000000001;    % initialization of constant F1
data.const2=0.5;    % initialization of constant Q1
data.const3=1000;    % initialization of constant K
data.const4=50;    % initialization of constant f
data.const5=40000;    % initialization of constant fs
data.const6=2;    % initialization of constant Q
w=11;
period = w/data.frequency;
deadline = period;
offset = 0;
prio = 1;
ttCreatePeriodictask('task1', offset, period, prio, 'code', data);
```

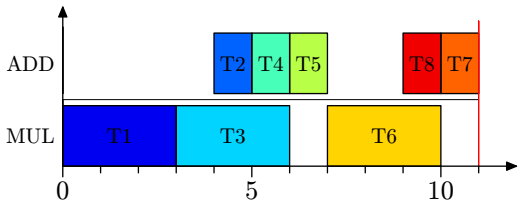
B.1.3 Uživatelský kód pro TrueTime

```
function [exectime,data] = code(seg,data)

 i=floor(ttCurrentTime/ttGetPeriod);

switch(seg)
case 1
       data.reg2 = data.units.unit1(2); %T7 Out
       data.units.unit2(1) = data.const1*data.reg2; %T1
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 2
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 3
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 4
       data.reg4 = data.units.unit2(4); %T1 Out
       data.units.unit2(1) = data.const2*data.reg2; %T3
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 5
       data.units.unit1(1) = data.reg1+data.reg4; %T2
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 6
       data.reg1 = data.units.unit1(2); %T2 Out
       data.units.unit1(1) = ttAnalogIn(1)-data.reg1;  %T4
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 7
       data.reg5 = data.units.unit2(4); %T3 Out
       data.reg6 = data.units.unit1(2); %T4 Out
       data.units.unit1(1) = data.reg6-data.reg5; %T5
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 8
       data.reg3 = data.units.unit1(2) ; %T5 Out
       data.units.unit2(1) = data.const1*data.reg3; %T6
       data.units.unit1(1) = data.reg3+data.reg1; %T8
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 9
       data.reg8 = data.units.unit1(2); %T8 Out
       ttAnalogOut(1,data.units.unit1(2));
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 10
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
```

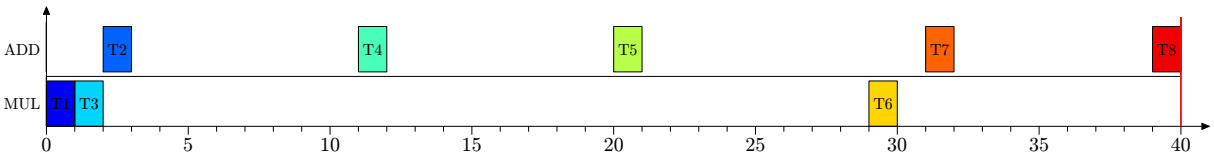
```
case 11
       data.reg7 = data.units.unit2(4); %T6 Out
       data.units.unit1(1) = data.reg7+data.reg2; %T7
       data.units=pipeunit(data.units);
       exectime = 1/data.frequency;
case 12
       exectime = -1;
end

function u=pipeunit(u)
        %  unit +
       for i=1:1
              u.unit1(3 - i) =  u.unit1(2 - i);
       end
       u.unit1(1)=0;
        %  unit *
       for i=1:3
              u.unit2(5 - i) =  u.unit2(4 - i);
       end
       u.unit2(1)=0;
return
```

Příloha C

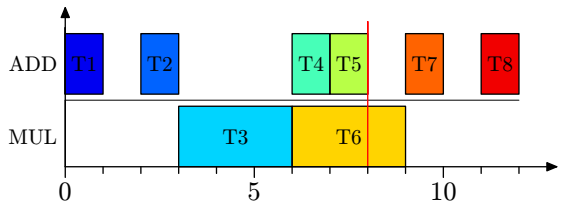
Výsledné rozvrhy testovacích benchmarků



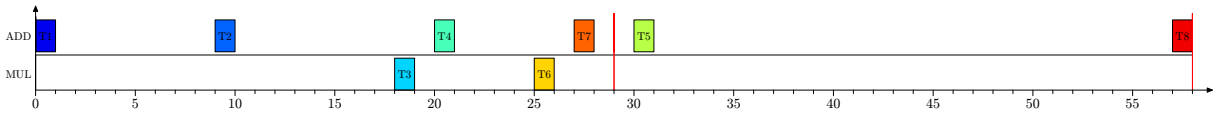
Obrázek C.1: Výsledný rozvrh DSVF filtru



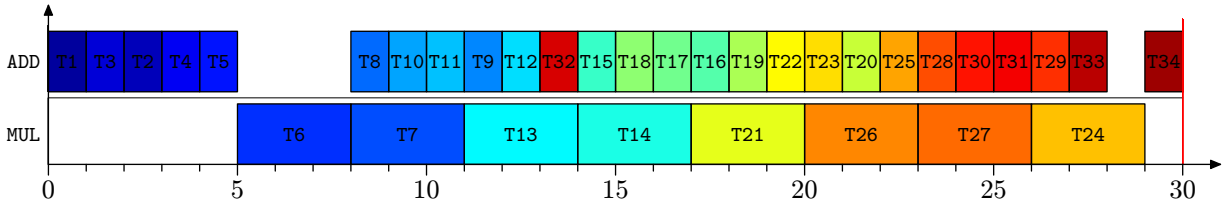
Obrázek C.2: Výsledný rozvrh DSVF filtru (hsla)



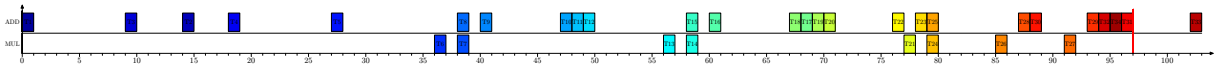
Obrázek C.3: Výsledný rozvrh WDF filtru



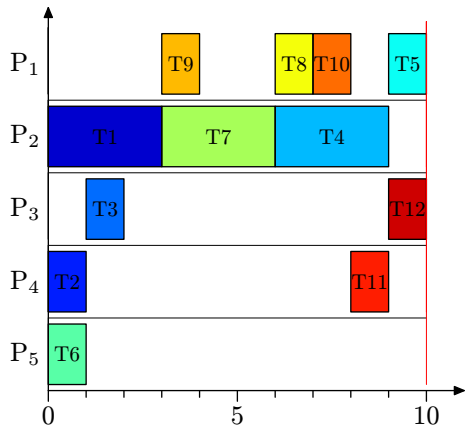
Obrázek C.4: Výsledný rozvrh WDF filtru (hsla)



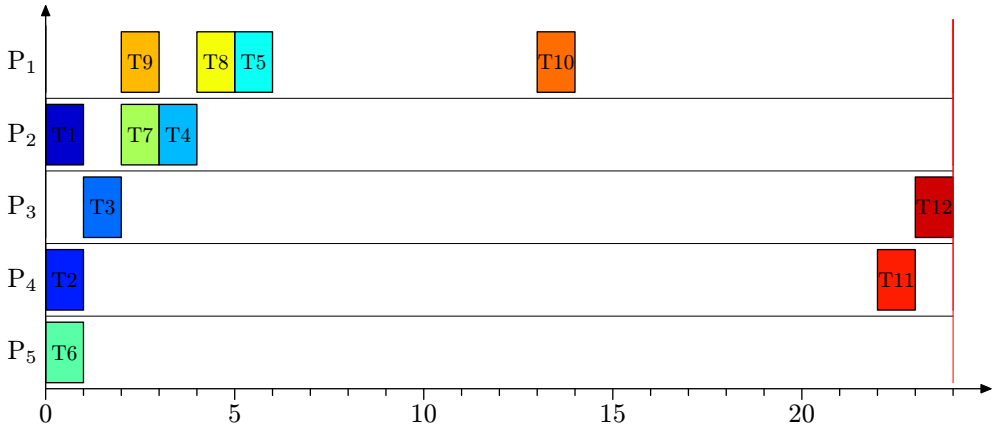
Obrázek C.5: Výsledný rozvrh filtru elliptic



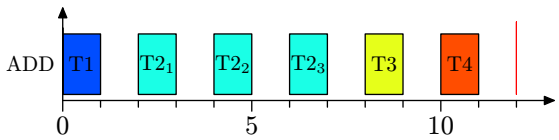
Obrázek C.6: Výsledný rozvrh filtru elliptic (hsla)



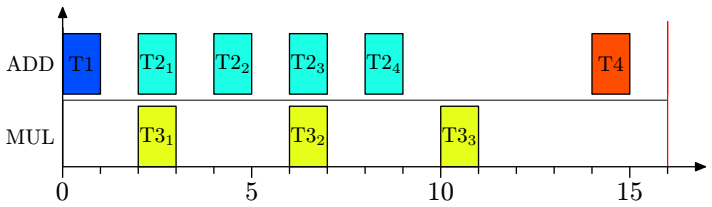
Obrázek C.7: Výsledný rozvrh PSD regulátoru



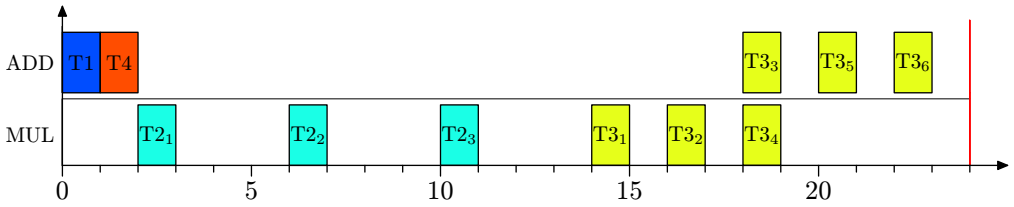
Obrázek C.8: Výsledný rozvrh PSD regulátoru (hsla)



Obrázek C.9: Výsledný rozvrh benchmarku nestedloops_benchmark1.m



Obrázek C.10: Výsledný rozvrh benchmarku nestedloops_benchmark2.m



Obrázek C.11: Výsledný rozvrh benchmarku nestedloops_benchmark3.m