České vysoké učení technické v Praze
Fakulta elektrotechnická

BAKALÁŘSKÁ PRÁCE

Automatické generování VHDL kódu pro FPGA

Praha, 2007 Autor: Tomáš Novák

Prohlášení

Prohlašuji, že jsem svou bakalářskou práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č.121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 28.5.2007

Poděkování

Rád bych touto cestou vyjádřil svůj dík vedoucímu práce Ing. Přemyslu Šůchovi za jeho cenné připomínky, trpělivost a ochotu při vedení mé bakalářské práce.

Poděkování patří i všem lidem, kteří mi nebránili v úspěšném dokončení práce, stejně tak jako programu pdfCSLaTeX, který mi usnadnil mnohou práci s formátováním textu.

Abstrakt

Aplikace digitálního zpracování signálu má v dnešní době velký význam. Použití běžných procesorů k provádění výpočtů není vždy vhodné vzhledem k tomu, že pracují pouze sekvenčně. Paralelismus dosažitelný pomocí signálových procesorů je také malý, a tak se jako vynikající prostředek pro zrychlení výpočtů zdají být programovatelná hradlová pole FPGA spolu s optimálním časovým rozvrhováním.

Následující práce se zaměřuje na popis struktury VHDL kódu sloužícího k popisu hardwarové struktury, která zajišťuje vykonávání DSP algoritmu. Dále se zabývá automatickým generováním tohoto VHDL kódu z daného optimálního časového rozvrhu. Automatický generátor je součástí nástroje ACGM (Automatic Code Generator for Matlab), který se zabývá rozvrhováním algoritmů pro FPGA architektury s pipelinovanými aritmetickými jednotkami.

Abstract

Nowadays, Digital Signal Processing (DSP) has a great significance. Usage of common processors is often inefficient due to their only sequential operating. Another posibility is to use signal processors, but the achieveable parallelism isn't great. The best way is to use Field Programmable Arrays (FPGAs) together with optimal scheduling.

This thesis concentrates on automatic VHDL generation from a given optimal schedule of a DSP algorithm. The automatic VHDL generator is a part of the tool ACGM (Automatic Code Generator for Matlab), which deals with scheduling of Matlab−compatible algorithms for FPGA architectures with pipelined arithmetic units. The thesis describes the architecture designed for implementation of DSP algorithms on FPGAs and how the automatic generator was created.

České vysoké učení technické v Praze, fakulta elektrotechnická

Katedra řídicí techniky   Školní rok: 2006/2007

Zadání bakalářské práce

Student: Tomáš Novák

Obor: Kybernetika a měření

Název tématu: Automatické generování VHDL kódu pro FPGA

Zásady pro vypracování:

1. Nastudujte struktury VHDL kódu generovaného nástroji jako je SPARK[1].
2. Navrhněte hardwarovou architekturu, vhodnou pro ACGM[2].
3. Implementujte generátor VHDL v jazyce C#.
4. Otestujte algoritmus na dodaných příkladech.

Seznam odborné literatury:

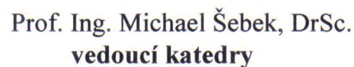
[1] Sumit Gupta, R.K. Gupta, N.D. Dutt, A. Nicolau, *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits*, Kluwer Academic Publishers, 2004

[2] P. Šůcha, M. Kutil, M. Sojka, Z. Hanzálek. TORSCHE Scheduling Toolbox for Matlab. To appear at IEEE International Symposium on Computer-Aided Control Systems Design. Munich, Germany: 2006

Vedoucí bakalářské práce: Ing. Přemysl Šůcha

Datum zadání bakalářské práce: zimní semestr 2006/07

Termín odevzdání bakalářské práce: 15. 8. 2007



Prof. Ing. Michael Šebek, DrSc.
vedoucí katedry

Prof. Ing. Zbyněk Škvor, CSc.
děkan

V Praze, dne 6. 3. 2007

Obsah

Abstrakt iii

Obsah ix

Seznam obrázků xii

Seznam tabulek xiii

1 Úvod 1

1.1 Související práce 1

1.2 Struktura ACGM 2

1.3 Přínos práce 3

1.4 Struktura práce 3

2 Automat pro DSP algoritmy ve VHDL 5

2.1 Reprezentace DSP algoritmu automatem 5

2.2 Realizace časového rozvrhu 5

2.3 Stavový automat ve VHDL 7

2.3.1 Rozhraní automatu 8

2.3.2 Signály 8

2.3.3 Procesy 10

2.4 Registry 13

2.5 Rozběh automatu 14

2.6 Algoritmy s vnořenými smyčkami 16

2.6.1 Reprezentace stavovým automatem 16

2.6.2 Primární automat 16

2.6.3 Sekundární automat 17

2.6.4 Blok MPX 18

2.6.5 RAM 18
2.6.6 Cyklický buffer 19
2.6.7 Vstup a výstup dat 19
2.7 Generátor VHDL 20
2.8 Zpracování vstupního XML 20
2.9 Generování VHDL 21
2.10 Konverze čísel 24
2.11 Ošetření chyb 24
2.12 Výstup VHDL 25

3 Experimenty **27**
3.1 Testovací rozhraní v FPGA 27
3.1.1 Aritmetické jednotky 28
3.1.2 FIFO 29
3.1.3 UART 29
3.1.4 Řadič 29
3.2 Testovací rozhraní v Matlabu 31
3.3 Výsledky experimentů 32
3.3.1 WDF 32
3.3.2 Elliptic 33
3.3.3 IIR7 33
3.3.4 Porovnání obou typů automatů 34
3.4 Porovnání se SPARK 34
3.4.1 Výsledek porovnání 35

4 Závěr **37**

Literatura **40**

A Kódy pro příklad bez vnořených smyček **I**
A.1 Algoritmus v ACGM I
A.2 Plný automat I
A.3 Redukovaný automat III

B Algoritmy bez vnořených smyček **VII**
B.1 WDF VII

B.2 PSD VIII
B.3 DSVF IX
B.4 Elliptic X
B.5 Kód pro testování v Matlabu X

C Algoritmy s vnořenými smyčkami **XIII**
C.1 Benchmark nb1 XIII
C.2 Benchmark nb2 XIV
C.3 Benchmark nb3 XIV

Seznam obrázků

1.1 Struktura ACGM 3

2.1 Rozhraní stavového automatu 6
2.2 Časový rozvrh a přiřazení stavů automatu 7
2.3 Blokové schéma automatu mealy 7
2.4 Možnosti využití více registrů pro uchování hodnot proměnných 14
2.5 Realizace algoritmu s vnoř. smyčkami 17
2.6 Kolize automatů 18
2.7 Propojení tříd *Task, Processor* a *Variable* 21
2.8 Asociace *Task* k jednotlivým hodinovým tikům *Tick* 22

3.1 Blokové schéma zapojení v FPGA 28
3.2 Skutečné schéma zapojení z Xilinx ISE 30
3.3 Porovnání výpočtu v FPGA a v Matlabu 31
3.4 Ganttův diagram rozvrhu ze SPARKu (w=56) 35
3.5 Ganttův diagram rozvrhu z ACGM (w=54) 35

B.1 Ganttův diagram pro WDF filtr VII
B.2 Ganttův diagram pro WDF filtr s jednotkami HSLA VIII
B.3 Ganttův diagram pro PSD filtr VIII
B.4 Ganttův diagram pro PSD filtr s jednotkami HSLA IX
B.5 Ganttův diagram pro DSVF filtr IX
B.6 Ganttův diagram pro DSVF filtr s jednotkami HSLA IX
B.7 Ganttův diagram pro filtr elliptic s jednotkami HSLA X

C.1 Ganttův diagram benchmarku 1 XIII
C.2 Ganttův diagram benchmarku 2 XIV
C.3 Ganttův diagram benchmarku 3 XV
C.4 Zapojení bloků v benchmarku nb3 XVI

C.5 Řídící a výkonný blok - nb3 XVI

Seznam tabulek

3.1 Porovnání automatů pro WDF filtr 33
3.2 Porovnání automatů pro filtr elliptic 33
3.3 Porovnání automatů pro filtr IIR7 34

B.1 Výsledky syntézy WDF VIII
B.2 Výsledky syntézy PSD VIII
B.3 Výsledky syntézy DSVF X
B.4 Výsledky syntézy filtu elliptic X

C.1 Výsledky syntézy NB1 XIV
C.2 Výsledky syntézy NB2 XIV
C.3 Výsledky syntézy NB3 XV

Kapitola 1

Úvod

Zpracování digitálního signálu hraje významnou roli v různých oblastech. S rostoucím počtem dat a požadavky na přesnost a rychlost zpracování se začnou běžné procesory jevit jako nevhodné. Můžeme použít DSP[1] procesory, mnohem větší prostor k urychlování výpočtů však poskytuje optimální časové rozvrhování výpočtů a jejich paralelizace. Pro následnou realizaci časově zoptimalizovaných algoritmů jsou velmi vhodná hradlová pole FPGA, která jsou dnes k provádění paralelních výpočtů běžně používána.

Protože programování hradlových polí ve VHDL [4, 5] je velmi zdlouhavé a náročné, směřuje vývoj ke kompilátorům vyšších jazyků, tedy C/Matlab → VHDL. Tato práce se zabývá způsobem implementace rozvrženého algoritmu v FPGA a dále automatickým generováním syntetizovatelného VHDL kódu.

1.1 Související práce

Syntézou kódu vyšších programovacích jazyků a následující implementací v FPGA se zabývá několik dalších projektů. Existují komerční nástroje, mezi nimiž uveďme *DSP Builder* [16] firmy Altera, *Accel DSP* [17] firmy Xilinx a *True DSP* [15], který poskytuje firma Synplicity. Stejnou problematikou se zabývají i další univerzity, uveďme *WDF toolbox* [7, 8] z TU Delft a *SPARK* [6] vyvinutý na univerzitě San Diegu.

Komerční nástroje vesměs pracují jako knihovna bločků pro Matlab Simulink. Sestavený obvod pro zpracování signálu je posléze převeden buďto do VHDL, nebo je přímo generován netlist pro další zpracování softwarem pro implementaci obvodů na FPGA.

[1] Digital Signal Processing

Vývoj komerčních nástrojů je uzavřený a nelze proto o nich získat mnoho informací. Všechny zmíněné nástroje provádí výpočty v pevné řádové čárce, informace o případné časové optimalizaci paralelních výpočtů nejsou k dispozici.

Accel DSP je nadstavba programu Xilinx System Generator a jako cílové obvody podporuje především vlastní obvody Xilinx.

True DSP podporuje jako cílové obvody FPGA různých výrobců. Výhodou tohoto softwaru je podle výrobce striktní oddělení návrhu obvodu v Matlabu, kde se nenastavují žádné konkrétní parametry výsledné FPGA struktury, od vlastní implementace v hradlovém poli.

DSP Builder se pak zaměřuje na hradlová pole Altera. Na rozdíl od True DSP je potřeba nastavovat ve schématu typ a šířku datových vodičů. Je umožněna simulace obvodu jak v Simulinku, tak v ModelSim určeném pro simulaci jazyka HDL.

Lattice WDF toolbox je využívaný pro potřeby výuky na TU Delft. Je zaměřen pouze na návrh a syntézu WDF[2] filtrů. Ze zadaných parametrů WDF (mezní frekvence, řád filtru) se vygeneruje popis ve speciálním vstupním jazyce rozvrhovacího toolboxu. Operace jsou potom rozvrženy na zadaný počet aritmetických jednotek, optimalizován je i počet použitých registrů. Jednotky mají pevnou latenci 1. Výstupem je VHDL kód, ve kterém je inicializován potřebný počet jednotek a registrů typu D a definováno jejich vzájemné propojení.

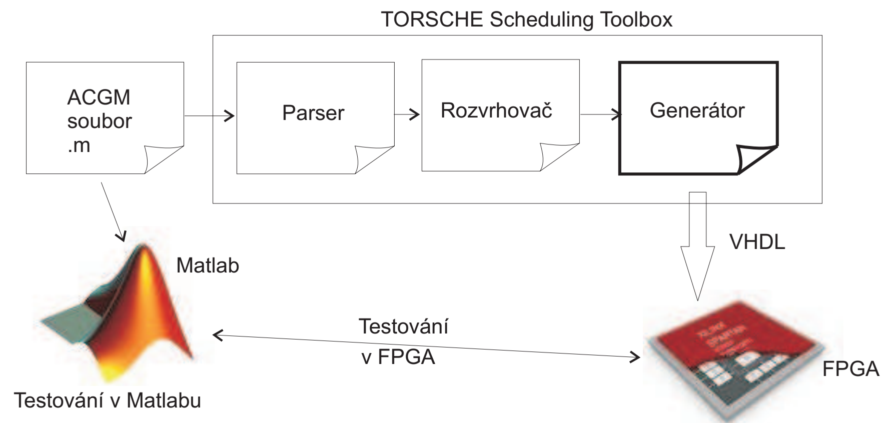
SPARK se zaměřuje především na řízení cyklů, smyček a zpracování podmínek. Dále zajišťuje rozvrhování jednotek, jejichž parametry jsou definovány v externím souboru. Neumožňuje práci s pipelinovanými jednotkami. Vstupním jazykem je redukované C, výstupem VHDL.

Porovnání ACGM právě se SPARKem je uvedeno v kapitole 3.

1.2 Struktura ACGM

Strukturu ACGM (Automatic Code Generator for Matlab) symbolicky znázorňuje obr. 1.1. Nástroj ACGM umožňuje zapsat v Matlabu iterační algoritmus zpracování digitálního signálu. Algoritmus prochází parserem [3]. Na výstupu parseru dostáváme algoritmus převedený do grafů. Nad grafovou reprezentací poté probíhá rozvrhování [2]. Rozvrhovací algoritmus přidá k původním informacím údaje o časovém rozvrhu. Po-

[2]Wave Digital Filter - číslicová aproximace analogových filtrů



Obrázek 1.1: Struktura ACGM

sledním článkem je generátor, který ze všech dat vytváří VHDL kód, který je možné syntetizovat pro FPGA. Způsob implementace algoritmu v hradlovém poli a generátor kódu jsou popsány v této práci.

1.3 Přínos práce

V průběhu práce byla navržena a implementována hardwarová struktura stavového automatu pro realizaci jednosmyčkových iterativních DSP algoritmů v hradlovém poli. Rovněž byla navržena struktura pro algoritmy s vnořenými smyčkami.

Dále byl vytvořen generátor VHDL kódu pro jednoduché algoritmy, který byl posléze upraven pro omezenou podmnožinu algoritmů s vnořenými smyčkami. Rozvržené algoritmy je možno testovat přímo z Matlabu. Pro dosažení takového řešení bylo potřeba vytvořit modul pro testování přes RS232 na straně FPGA a napsat v Matlabu funkce pro sériovou komunikaci s vytvořeným obvodem.

V práci je rovněž provedeno porovnání ACGM s nástrojem SPARK [6].

1.4 Struktura práce

V kapitole 2 je uveden způsob realizace automatů pro jednoduché smyčky (sekce 2.1) a pro algoritmy s vnořenými smyčkami (sekce 2.6). Generátor VHDL kódu je popsán

v sekci 2.7 .

Následující kapitola 3 ukazuje způsob testování vytvořeného obvodu a shrnuje část výsledků syntézy. Výsledky zbylých benchmarků s jednoduchými smyčkami jsou uvedeny v příloze B a benchmarky s vnořenými smyčkami shrnuje příloha C.

Nakonec je uvedeno srovnání ACGM s nástrojem SPARK, které se nachází v sekci 3.4.

Kapitola 2

Automat pro DSP algoritmy ve VHDL

2.1 Reprezentace DSP algoritmu automatem

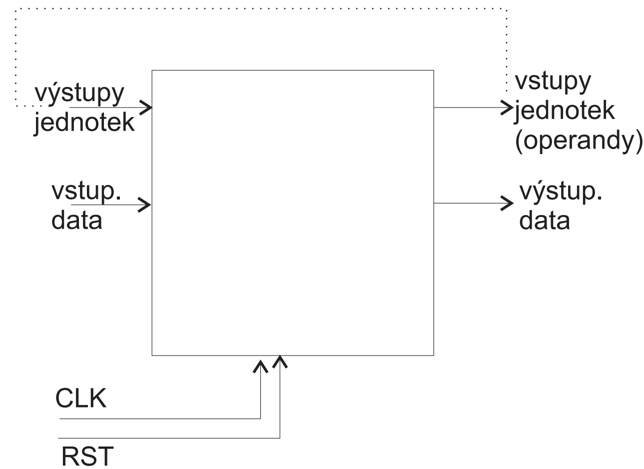
Vstupní DSP algoritmus projde rozvrhovacím mechanismem, který pro předem dané jednotky, zejména pak jejich latence[1], vytvoří optimální časový rozvrh provádění požadovaných operací. A tento rozvrh musíme nějak oživit. K tomu využijeme FPGA a jazyk pro návrh hardwarových struktur VHDL. Rozvrh budeme implementovat jako stavový automat.

Úkolem stavového automatu je řízení toku dat mezi aritmetickými jednotkami (procesory). Jeho koncepce je naznačena na obr. 2.1. Vidíme, že do obvodu vstupují jednak data ke zpracování, a dále výsledky operací z procesorů. Na výstupu jsou naopak operandy pro zpracování v procesorech a výstupní data. Obvod je řízen hodinovým signálem synchronně s jednotkami.

2.2 Realizace časového rozvrhu

Než se budeme věnovat samotnému stavovému automatu, je třeba vysvětlit, co vlastně představují jeho stavy. Stavy mají přímou souvislost s časovým rozvrhem, a jsou navrženy dvě odlišné reprezentace. *Úplný* automat odpovídá svými stavy přesně hodinovým tikům

[1] Latencí jednotky rozumíme počet hodinových cyklů od naplnění jednotky operandy do vystavení výsledku operace na výstupu jednotky.



Obrázek 2.1: Rozhraní stavového automatu

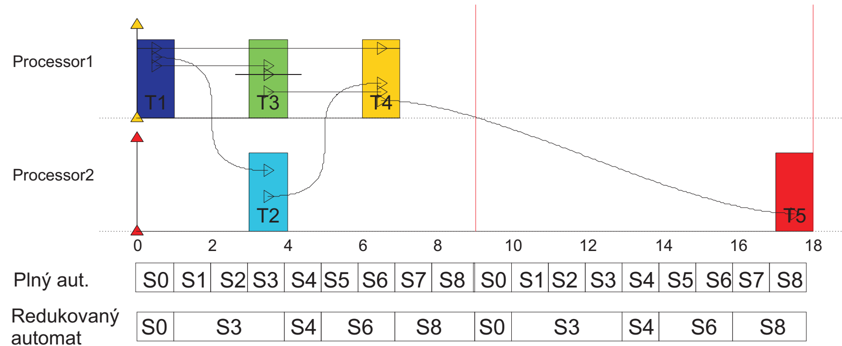
jedné periody těla časového rozvrhu. S každým hodinovým cyklem tedy dochází ke změně stavu a velikost automatu je přímo úměrná délce periody rozvrženého algoritmu.

Nevýhodou prvního automatu je to, že počet jeho stavů narůstá i pro hodinové cykly, ve kterých probíhá pouze výpočet uvnitř pipelinovaných jednotek, ale nejsou k dispozici výsledky na jejich výstupech, ani nové vstupní operandy. Počet takovýchto *prázdných* stavů závisí na latenci jednotlivých jednotek a je pevně určen časovým rozvrhem. Pro eliminaci těchto nadbytečných stavů byl navržen *redukovaný* automat, který prázdné stavy slučuje spolu s prvním následujícím neprázdným dohromady. Tento přístup však vyžaduje rozšíření automatu o blokování hodin. Dále přibyl čítač, který pozastavování automatu řídí. Přiřazení stavů k časovému rozvrhu znázorňuje obr. 2.2. K provedení operace S0 dochází ve skutečnosti v okamžiku přechodu S0→S1.

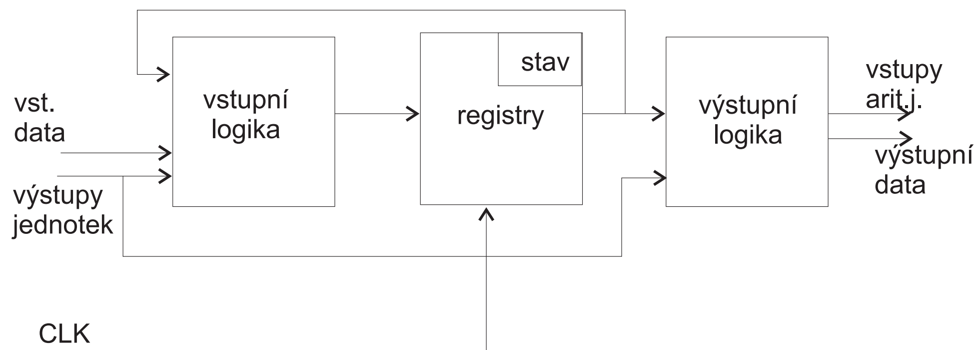
Použitý diagram odpovídá následujícímu algoritmu a nebude-li uvedeno jinak, budou se k tomuto zadání vztahovat i další uvedené příklady.

```
a = -0.375;
b =  0.5;

for k=1:10
 n1{k} = X{k} + n4{k-1};
 n2{k} = a * n1{k};
 n3{k} = n3{k-1} - n1{k};
 n4{k} = n2{k} + n3{k};
 Y{k}  = b * n4{k};
end;
```



Obrázek 2.2: Časový rozvrh a přiřazení stavů automatu



Obrázek 2.3: Blokové schéma automatu mealy

2.3 Stavový automat ve VHDL

Jedná se o automat typu Mealy, jehož blokové schéma je na obr. 2.3. Stavové automaty jsou ve VHDL reprezentovány obvykle třemi paralelními procesy, představujícími jednotlivé bloky schématu. *Vstupní logika* v našem případě představuje kombinační obvod (kterému ve VHDL přísluší odpovídající proces), jenž se stará o generování dalšího stavu. Dále se zde nachází demultiplexory, které přivádí výsledky z procesorů do správných registrů. Jak je uvedeno dále, registry odpovídají části proměnných vstupního algoritmu.

Část automatu s *registry* je sekvenční. Je zde jednak uložen binárně zakódovaný stav automatu a jednak některé mezivýsledky. Konkrétní zakódování stavů je provedeno až při syntéze, kdy syntetizátor volí vhodný kód, například Johansonnův, Grayův nebo tzv. one-hot. Mezivýpočty jsou uloženy v registrech nazvaných podle proměnných algoritmu. Některé proměnné (v našem případě *n1*) jsou vyeliminovány tím, že mezivýsledek je

přímo přesunut mezi procesory a nevyžaduje tedy další ukládání hodnoty.

Konečně *výstupní logika* je opět kombinační. Reprezentuje multiplexory, které připojují operandy na vstupy aritmetických jednotek. V roli operandů zde vystupují jak hodnoty uložené v registrech, tak vstupní data (určená ke zpracování) a výsledky z předešlých právě ukončených operací, které nepotřebují dočasné uložení v registru.

2.3.1 Rozhraní automatu

Definice entity ve VHDL obsahuje seznam portů pro připojení hodinového signálu, resetu, procesorů (operandy + výsledek), vstup zpracovávaných dat, výstup a potvrzovací signály. Šířka datových sběrnic je pro jednoduchost určena parametrem, její změna ale neobnáší pouze změnu parametru. Musely by být předefinovány i hodnoty konstant, které jsou uložené přímo jako 16bitové číslo.

Konkrétní definice entity pro použitý příklad je následující:

```
entity automat is            -- za dvema pomlckami je komentar
generic(dw : integer := 16); -- sirka datovych sbernic
port(clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    add_IN0 : out STD_LOGIC_VECTOR(dw-1 downto 0);
    add_IN1 : out STD_LOGIC_VECTOR(dw-1 downto 0);
    add_OUT : in  STD_LOGIC_VECTOR(dw-1 downto 0);
    mul_IN0 : out STD_LOGIC_VECTOR(dw-1 downto 0);
    mul_IN1 : out STD_LOGIC_VECTOR(dw-1 downto 0);
    mul_OUT : in  STD_LOGIC_VECTOR(dw-1 downto 0);
    X_v     : in  STD_LOGIC_VECTOR(dw-1 downto 0);
    next_X  : out STD_LOGIC;
    Y_out   : out STD_LOGIC_VECTOR(dw-1 downto 0);
    valid_Y : out STD_LOGIC );
end entity automat;
```

2.3.2 Signály

Signálem ve VHDL rozumíme jak jednobitové vodiče, tak sběrnice. Signály slouží ve VHDL ke komunikaci jednotlivých procesů. Signály odpovídající názvům portů jsou dostupné automaticky. Dále je potřeba vytvořit registry odpovídající jednotlivým proměnným DSP algoritmu. Jednou z možností, jak popsat synchronní registry, je definovat každý registr pomocí dvojice signálů. Jeden reprezentuje vstup do registru, druhý výstup

registru. Synchronně s hodinovým signálem potom dochází k přepisu vstupní hodnoty na výstup. Vytváření synchronních registrů je potřeba věnovat náležitou pozornost - při špatném popisu může být vygenerován asynchronní *latch*. Ten je ve většině případů nežádoucí a může způsobovat problémy s hazardy.

Poslední registr je nutno vytvořit pro uchování výsledku výpočtu jedné iterace DSP algoritmu.

Signál *state* určuje aktuální stav automatu. Operace přidružené např. stavu S0 budou provedeny s náběžnou hranou hodin při přechodu S0→S1. Naopak signál *nstate* je výstup kombinační logiky, který určuje, jaký bude následující stav. Přepis hodnot nstate→state je synchronizován opět s náběžnou hranou hodin.

V této deklarační sekci je také výčet stavů automatu. Stavy jsou nazvány symbolicky, k jejich zakódování na příslušnou binární posloupnost dojde až při syntéze. V sekci 3.3 je pak v tabulkách uvedeno, jakou posloupností byly v každém případě stavy zakódovány. Pro malý počet stavů je obvykle použit *Grayův kód*, který je úsporný svou datovou šířkou. Pro rostoucí počet stavů je pak použit *one-hot* kód, který se vyznačuje jednoduchostí dekódování stavu.

Pro redukovanou verzi je dále potřeba signál povolení hodin, signál určující kolik tiků bude automat zastaven a signál, který po syntéze reprezentuje vlastní čítač. Toto sice přísně vzato není součást automatu, nicméně nic nebrání popis této části přidat do stejné entity tak, aby se plný i redukovaný automat z pohledu portů jevily a fungovaly naprosto stejně.

Začátek deklarace *architektury* spolu s definicí signálů v jazyce VHDL vypadá následovně:

```
architecture priklad of automat is
-- type stavy is (sidle, sT0, sT1, sT2, sT3, sT4, sT5, sT6, sT7, sT8);
type stavy is (sidle, sT0, sT3, sT4, sT6, sT8);  -- redukovany
signal state, nstate: stavy;
signal n2, n2_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n2
signal n3, n3_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n3
signal n4, n4_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n4
constant a_v : STD_LOGIC_VECTOR(dw-1 downto 0) := "1111111110100000";
constant b_v : STD_LOGIC_VECTOR(dw-1 downto 0) := "0000000010000000";
signal Y_v,Y : STD_LOGIC_VECTOR(dw-1 downto 0);    -- registr pro výstup

signal waitTicks, cnt : std_logic_vector(1 downto 0);
signal aut_en : std_logic;   -- povoleni hodin pro automat
```

2.3.3 Procesy

O hlavních procesech, které popisují vlastni Mealyho stavový automat, bylo již zmíněno výše. Přechody do následujícího stavu se dějí cyklicky bez jakýchkoli podmínek. Následuje kód, který tuto cyklickou změnu stavu zajišťuje. Použití příkazu *case* je v této situaci zcela standardní.

```
stavy: process(state)
begin
  case(state) is
    when sidle => nstate <= sT0;
    when sT0 => nstate <= sT3;
    when sT3 => nstate <= sT4;
    when sT4 => nstate <= sT6;
    when sT6 => nstate <= sT8;
    when sT8 => nstate <= sT0;
  end case;
end process prech;
```

Pro každý registr je zatím definována dvojice signálů. Aby syntetizátor skutečně rozeznal, že se jedná o registry, je potřeba vytvořit další synchronní proces. Ten s náběžnou hranou hodinového signálu provádí zápis hodnoty vstupního signálu na signál výstupní - tak funguje registr typu „D“. Navíc je zajištěno vynulování všech registrů po resetu.

Následuje kód procesu, popisujícího chování registrů.

```
registry: process(clk,rst)
begin
if rst='0' then   -- reset aktivni v '0'
  state <= sidle; -- inicializacni stav
  n2_v <= (others => '0'); -- vynulovani registru
  n3_v <= (others => '0');
  n4_v <= (others => '0');
  Y_v  <= (others => '0');
elsif rising_edge(clk) then -- pri nabezne hrane hodin
 if aut_en = '1' then       -- !! blokovani hodin
  state <= nstate;
  n2_v <= n2;  -- pro vytvoreni registru
  n3_v <= n3;  -- pro vytvoreni registru
  n4_v <= n4;  -- pro vytvoreni registru
  Y_v <= Y;
 end if;
end if;
end process sync;
```

Poslední operací s registry je vyvedení hodnoty výstupního registru na odpovídající výstupní port *entity*. To je zajištěno jedním řádkem VHDL kódu:

```
Y_out <= Y_v;
```

Konečně výstupní logika v každém stavu propojuje patřičné registry s procesory. V jazyce VHDL platí pro přiřazení hodnoty každému signálu poslední hodnota. Toho je zde využito tak, že nejprve nadefinujeme zpětnou vazbu pro každý registr (zapamatování minulé hodnoty), a tuto ZV případně předefinujeme vstupem hodnoty z aritmetické jednotky. V této části kódu je také provedena případná změna znaménka operandu. Je totiž potřeba umožnit změnu znaménka jakéhokoli operandu a není tedy možné spoléhat na to, že funkci obrácení znaménka bude poskytovat aritmetická jednotka. Například jednotka add/sub umožňuje změnu znaménka pouze druhé vstupující hodnoty. Při reprezentaci v plovoucí řádové čárce bude stačit změna nejvyššího bitu pomocí operace *xor*, při reprezentaci v pevné čárce je potřeba vypočítat dvojkový doplněk. Výpočet dvojkového doplňku vyžaduje použití sčítačky. Počet těchto sčítaček je při syntéze pomocí Xilinx XST [19] optimalizován na minimum. To však znamená, že je v FPGA vytvořen další multiplexor, který data ke sčítačce připojuje. Pokud by bylo sčítaček vygenerováno více, ušetřily by se naopak multiplexory, a náročnost výsledného designu by se pravděpodobně příliš nezměnila. Navíc pevná řádová čárka je použita spíše pouze pro potřeby testování a proto není potřeba se problémem změny znaménka v pevné řádové čárce podrobně zabývat.

Následující úsek kódu ilustruje výše uvedený postup:

```
vystup: process(state, n2_v, n3_v, n4_v, X_v, Y_v, add_OUT, mul_OUT)
begin
    add_IN0 <= (others => '0');  -- defaultni hodnota
    add_IN1 <= (others => '0');  -- na vstupu procesoru
    mul_IN0 <= (others => '0');
    mul_IN1 <= (others => '0');
    n2 <= n2_v ;  -- zpetna vazba pro zapamatovani posledni hodnoty
    n3 <= n3_v ;  --
    n4 <= n4_v ;  --
    Y <= Y_v ;    --
case(state) is
  when sT0 =>          -- T0
    n4 <= add_OUT;
    Y <= mul_OUT;
    add_IN0 <= X_v;
    add_IN1 <= add_OUT;
```

```
    waitTicks <= conv_std_logic_vector(2,2); -- zastavit na 2 tiky
  when sT3 =>          -- T3
    mul_IN0 <= a_v;
    mul_IN1 <= add_OUT;
    add_IN0 <= n3_v;
    add_IN1 <= (add_OUT xor x"FFFF") + x"0001" ; -- zmena znamenka operandu
    waitTicks <= conv_std_logic_vector(0,2);
  when sT4 =>          -- T4
    n2 <= mul_OUT;
    waitTicks <= conv_std_logic_vector(1,2);
  when sT6 =>          -- T6
    n3 <= add_OUT;
    add_IN0 <= n2_v;
    add_IN1 <= add_OUT;
    waitTicks <= conv_std_logic_vector(1,2);
  when sT8 =>          -- T8
    mul_IN0 <= b_v;
    mul_IN1 <= n4_v;
    waitTicks <= conv_std_logic_vector(0,2);
   when others => waitTicks <= (others => '0');
end case;
end process output;
```

V případě plného automatu chybí řádky s dosazením do `waitTicks`.

Počítání prázdných cyklů zajišťuje n-bitový dekrementální čítač. Počet bitů lze z maximálního počtu po sobě jdoucích prázdných stavů určit jako $\text{floor}(\log_2(max_cek)) + 1$, kde max_cek je rovno maximálnímu počtu po sobě jdoucích „prázdných" stavů. Dojde-li hodnota čítače k nule, je povolen `clk_en` automatu a zároveň dojde k načtení nové hodnoty do čítače. Je-li nová hodnota nenulová, je automat, který mezitím přešel do dalšího stavu, opět zastaven a probíhá dekrementování hodnoty čítače.

Popis takovéhoto čítače ve VHDL uvádí následující úsek kódu.

```
citac: process (clk, rst)
begin  -- process
  if rst = '0' then    -- asynchronni reset
    cnt <= "00";
  elsif clk'event and clk = '1' then
    if cnt = "00" then
      cnt <= waitTicks; -- nahrat novou hodnotu
    else
      cnt  <= cnt - 1;  -- dekrementace hodnoty, nebyla-li nulova
    end if;
  end if;
```

```
end process;
aut_en <= '1' when cnt="00" else '0'; -- povoleni nebo zakazani clk_en
```

Synchronizaci výměny dat s okolím obstarává poslední proces. Jedná se o sekvenční obvod, který vždy po dobu jednoho hodinového cyklu signalizuje, že byla přečtena hodnota ze vstupu a má být přivedena další, nebo naopak že je na výstupu nový platný výsledek. V uvedeném příkladu náhodou došlo k situaci, kdy je vstup načítán ve stejný hodinový cyklus jako se na výstupu objevuje výsledek, a to v čase T0. U jiných algoritmů může být výsledek k dispozici v jiném cyklu, než je načítán vstup, potom se budou obě podmínky ve VHDL lišit.

Následuje zdrojový kód synchronizačního procesu.

```
hshake: process(clk,rst)
begin
 if rst='0' then
   next_X <= '0';
   valid_Y <= '0';
 elsif rising_edge(clk) then
   next_X <= '0';
   valid_Y <= '0';
   if state = sT0 and aut_en='1' then
     next_X <= '1';
   end if;
   if state = sT0 and aut_en='1' then
     valid_Y <= '1';
   end if;
 end if;
end process hshake;
```

2.4 Registry

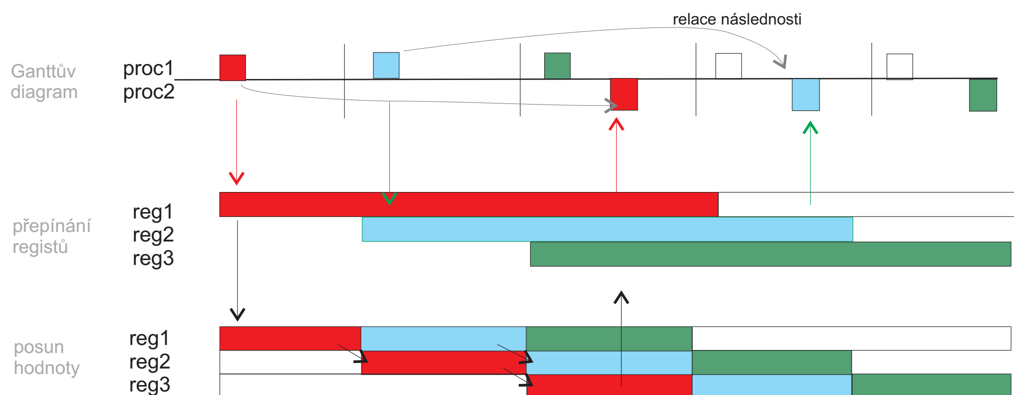
V tak jednoduchém případě, jako je použitý příklad, nedošlo k situaci, kdy je potřeba některý mezivýsledek zachovat po dobu delší než je jedna celá perioda. Nastane-li taková situace, je jasné, že jeden registr nestačí, neboť by jeho hodnota byla přepsána novou ještě dříve, než bychom tu starou stačili použít.

Nabízí se dva způsoby, jak tuto situaci vyřešit. Můžeme použít dva (nebo více) registrů, a používat je střídavě. Takové řešení vypadá jako schůdné, ale každý takový registr navíc vyžaduje jak rozšíření demultiplexoru za procesorem, jehož výsledek se ukládá

(proc1 na obr. 2.4), tak i multiplexoru před procesory (proc2), do kterých operand vstupuje.

Mnohem elegantnější řešení tedy je taková konfigurace registrů, kdy jsou jednotlivé registry seřazeny za sebou a jednou za periodu algoritmu mezi sebou data postupně přesouvají. Takové řešení nepřináší žádné další zesložitění okolní logiky. Soustava takovýchto registrů je použitým syntetizátorem rozpoznána a implementována jako posuvný registr.

Na obr. 2.4 jsou symbolicky znázorněny oba možné způsoby, nicméně vhodný pro implementaci je pouze způsob druhý.



Obrázek 2.4: Možnosti využití více registrů pro uchování hodnot proměnných

2.5 Rozběh automatu

Vzhledem k prolnutí iteračních smyček ve výsledném rozvrhu dojde v několika počátečních periodách algoritmu k situaci, kdy ještě není na výstupu jednotek platný výsledek, ale podle rozvrhu má být s hodnotou pracováno. Jedná se vlastně o výsledky operací v čase $t < 0$. V uvedeném příkladu tato situace nastala také. Jedná se o výsledek násobení (T5), který je poprvé k dispozici až ve stavu *S1* při třetím průchodu smyčkou algoritmu (viz Ganttův diagram na obr. 2.2). Pro přehlednost však nebyl tento problém řešen. Při počáteční inicializaci registrů na hodnotu 0 a použití sčítačky a násobičky totiž nedochází ke špatné funkci automatu. Pokud bychom však použili děličku, mohlo by dojít k dělení nulou a automat by se vůbec správně nerozběhl.

Proto je potřeba několik počátečních singulárních period odlišit, a použití výsledku z jednotky vázat podmínkou validity dat. Nový signál `nextLoop` indikuje po dobu jedné periody hodinového signálu přechod algoritmu do další smyčky. Signály `loop1, loop2` (počet dán počtem singulárních period) slouží jako podmínka pro použití výsledku z procesoru. Signály `loopX` jsou po resetu nastaveny na hodnotu '0'. Po každé iteraci algoritmu je odpovídající signál nastaven na '1', přičemž tato hodnota již zůstává přiřazena neustále. V okamžiku, kdy mají všechny signály `loopX` hodnotu '1', došlo již k úplnému rozběhu automatu. Potom již výpočty probíhají přesně podle časového rozvrhu.

Podmínka uložení hodnoty do registru a použití hodnoty jinou aritmetickou jednotkou pak vypadá následovně:

```
case(state) is
    when sT0 =>                  -- T0
      add_IN0 <= X_v;
      if loop1 = '1' then       -- od 2. periody
        n4 <= add_OUT;
        add_IN1 <= add_OUT;
      else
        add_IN1 <= "0000000000000000";  -- inic.hodnota n4
      end if;
    when sT1 =>                  -- T1
      if loop2 = '1' then
        Y <= mul_OUT;
      end if;
 ...
```

Nastavování hodnot pomocných signálů zajišťuje další proces `lpcnt`. Jeho VHDL kód je následující:

```
lpcnt:process(clk,rst)
begin
 if rst='0' then
    loop1 <= '0';
    loop2 <= '0';
 elsif rising_edge(clk) then
  if nextLoop = '1' then
    loop1 <= '1';
    loop2 <= loop1;
  end if;
 end if;
end process lpcnt;
```

2.6 Algoritmy s vnořenými smyčkami

Předcházející sekce popisovaly způsob, jakým jsou vytvořeny stavové automaty pro jednoduché algoritmy. Dalším krokem bude rozšíření algoritmů o takové, které navíc obsahují vnořené smyčky. Vnořené smyčky (makra, [3]) umožňují například provádět maticové násobení.

U skupiny takto rozšířených algoritmů se nepředpokládá použití pouze skalárních proměnných, nýbrž i vektorů a matic. Nyní následuje návrh hardwarové struktury pro implementaci těchto algoritmů v FPGA. Samotný návrh struktury obvodu v FPGA je univerzální. Podle návrhu byly vytvořeny skutečné obvody pro tři testovací algoritmy. Testovací algoritmy byly omezeny na jedinou úroveň vnoření smyček. Cyklické buffery také nejsou použity.

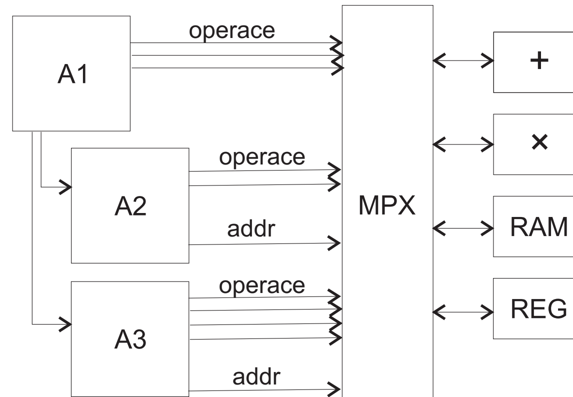
2.6.1 Reprezentace stavovým automatem

Algoritmus s vnořenými smyčkami je realizován skupinou stavových automatů, organizovaných ve stromové struktuře. Primární automat spouští operace hlavní smyčky a dále aktivuje výpočty v automatech sekundárních. Sekundární automat potom na základě spouštěcího impulsu provede žádaný počet iterací a přejde zpět do čekacího režimu. Kdyby každý stavový automat přistupoval k jednotkám a pamětem, musely by být použity třístavové sběrnice spolu s řízením přístupu. Proto jednotlivé automaty tvoří pouze řídící část celého zapojení, generují pouze jednobitové požadavky na provedení operací uložení výsledku nebo naplnění jednotky. Vlastní provádění těchto operací obsluhuje samostatný výkonný blok, jak je naznačeno na obr. 2.5. Každý automat navíc pro přístupy k RAM poskytuje aktuální adresu.

2.6.2 Primární automat

Úkolem primárního automatu je vykonávání operací hlavní smyčky a spouštění výpočtů v podřízených automatech.

Jelikož se jedná o sekvenční obvod, reaguje podřízený automat vždy se zpožděním rovným jednomu hodinovému cyklu. Proto je spouštění automatů potřeba provádět o jeden hodinový cyklus dříve, než provádění skalárních operací, které jsou podle Ganttova diagramu vykonávány ve stejný časový okamžik, jako první operace makra. Protože může být potřeba aktivovat podřízené automaty, které pracují od času $t = 0$, musí mít hlavní



Obrázek 2.5: Realizace algoritmu s vnoř. smyčkami

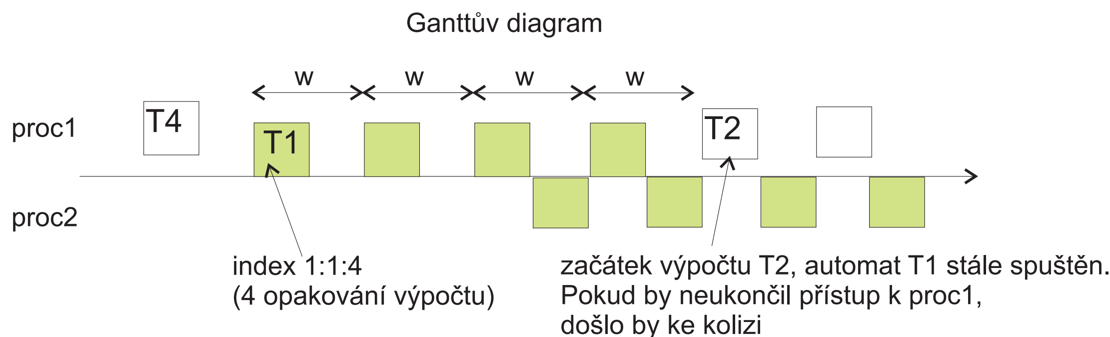
automat navíc stav, do kterého přechází pouze jednou před začátkem výpočtu. V tomto stavu dojde k případnému spuštění výpočtu vnořené smyčky. Dále se již bude tento výpočet aktivovat v posledním stavu hlavní smyčky tak, aby podřízený automat začal pracovat ve správný okamžik.

2.6.3 Sekundární automat

Automat, reprezentující operace vnořené smyčky, je ovládán z nadřazeného automatu pomocí signálu *start*. V době nečinnosti je sekundární automat v inicializovaném stavu s definovanými výstupy; není generován požadavek na žádnou operaci. Po nastavení *start* ='1' dojde k rozběhu automatu. Automat provede daný počet iterací odpovídající počtu opakování vnořené smyčky a opět přejde do inicializačního stavu.

Vzhledem k pravděpodobnému prolnutí iteračních smyček algoritmu je potřeba počítat průchody smyčkami. Jinak by mohlo dojít ke kolizi přístupu k jednotkám, jak ukazuje obr. 2.6. Prolnutí smyček totiž umožňuje vznik situace, kdy „vnořený“ automat stále pracuje, ale část výpočtů na jednotce *proc1* je již ukončena a tato jednotka může být použita k výpočtům, řízeným z jiného místa (*T2*). Pokud by automat neukončil požadavek na přístup k jednotce, musely by se řešit priority jednotlivých automatů. Otázkou je, zda by se vždy daly priority navrhnout tak, aby bylo dosaženo požadovaného výsledku.

Zde se nabízí dvě varianty počítání cyklů: každému pořadí cyklu přiřadit jeden bit, nebo použít čítač. První varianta, jeden bit pro průchod smyčkou, je jednodušší na realizaci (posuvný registr) i na vyhodnocení, stačí vyhodnocovat jediný bit. V případě vyššího počtu požadovaných cyklů by však narůstal počet bitů lineárně na příliš vysokou hod-



Obrázek 2.6: Kolize automatů

notu. Druhá varianta vyžaduje vytvoření hardwarově náročnějšího čítače a komparátoru, nicméně s růstem počtu period narůstá požadovaná bitová šířka pouze logaritmicky.

Pro generování adres pamětí jsou použity čítače. Předpokládá se lineární indexování paměti s přírůstkem 1. Případná nenulová počáteční adresa (offset) je nastavena inicializací hodnoty čítače na žádanou adresu, poté již dále probíhá postupné inkrementování. Má-li být například provedena operace zápisu výsledku výpočtu z aritmetické jednotky do paměti, je vystavena adresa paměti a nastaven příznak žádosti o provedení přesunu a zápisu dat. V příštím hodinovém cyklu dojde k inkrementaci hodnoty čítače a za dobu periody w se toto opakuje.

2.6.4 Blok MPX

Blok ovládající jednotky a paměti obsahuje multiplexory, které propojují jednotlivé datové vodiče připojených bloků. Při přístupu k pamětem navíc propojuje adresu generovanou automatem na adresní vstup paměti.

Požadavek na provedení každé operace přesunu dat je představován samotným bitem. Tento bit je aktivován některým ze stavových automatů. Nastavené bity pak v bloku MPX slouží k přepínání multiplexorů.

2.6.5 RAM

Výsledky rozvrhování pro jednoduchá testovací makra ukazují, že je potřeba přistupovat v jediném hodinovém cyklu k paměti jak v režimu čtení, tak v režimu zápisu. Je

tedy nutné použít dualportové[2] paměti. Omezení na jediný přístup za hodinový cyklus by bylo příliš restriktivní, a zbytečně by zhoršovalo časový rozvrh.

Další požadavek na paměť je její nulování. RAM vestavěné na čipu FPGA neumožňují provést reset (nastavení všech paměťových buněk na hodnotu 0) během jednoho hodinového cyklu, proto byla vytvořena ve VHDL paměť, která rychlé nulování umožňuje.

Aby dále nevznikalo zesložitění s indexací synchronně pracující paměti, kdy je potřeba adresu přivést o hodinový cyklus dříve před vlastním čtením, je výstupní port paměti asynchronní. Zápis do paměti musí z principu zůstat synchronní.

2.6.6 Cyklický buffer

Cyklickým bufferem rozumíme posuvné časové okno, které zachovává určitou historii hodnot, například několik vzorků vstupního signálu. Toto okno lze realizovat pomocí RAM paměti, která bude indexována dvojicí báze + offset. Při každé iteraci výpočtu je inkrementována báze, indexování prvků uvnitř okna je dáno indexem. Nabyde-li výsledná adresa hodnoty větší než je velikost paměti, dojde k přetečení adresy a paměť je adresována cyklicky opět od začátku. Není tedy potřeba provádět žádnou zvláštní kontrolu hodnot báze ani offsetu.

2.6.7 Vstup a výstup dat

Zpracování dat probíhá blokově. Všechna vstupní data jsou před zpracováním uložena v jedné z pamětí. Automat nijak neovlivňuje zápis dat do této paměti, z celé struktury jsou vyvedeny signály *adresa, data* a *WE*[3]. Pomocí těchto signálů lze naplnit vstupní paměť a poté je teprve možno spustit samotný výpočet. Podobně výsledky jsou postupně ukládány do výstupní paměti. Po zpracování všech vzorků je možno výsledek z této paměti přečíst obdobným způsobem, jako byl proveden zápis vstupních dat.

[2]paměť umožňuje v jeden časový okamžik nezávisle na sobě zároveň číst i zapisovat

[3]*write enable* – povolení zápisu do paměti

2.7 Generátor VHDL

Ruční tvorba výše popsaných stavových automatů ve VHDL kódu je velmi zdlouhavá. Jelikož se v kódu různých algoritmů opakují stejné části, nabízí se možnost vytvořit automatický generátor tohoto kódu, o němž pojednává následující část práce. Jedná se o generátor VHDL kódu pro jednoduché jednosmyčkové algoritmy.

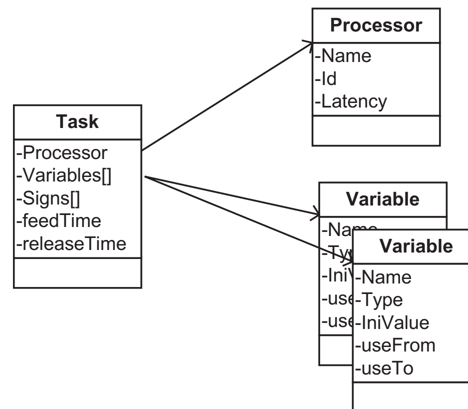
2.8 Zpracování vstupního XML

Vstupem do programu je XML (*Extensible Markup Language*) soubor, který obsahuje obraz struktuty *taskset* [1], se kterou pracuje scheduling toolbox. K analýze XML je využito možností C#, a to balíku `System.Xml`. Tento balík obsahuje i implementaci jazyka XPath [14], pomocí kterého lze jednoduše vytvářet masky specifikující cestu k žádaným datovým strukturám.

Nejprve je načtena délka periody rozvrhu a typ datové reprezentace, poté jsou zpracovány informace o procesorech. Rozvrh může být vytvořen i pro několik identických jednotek, a tak je potřeba určit jejich celkový počet. Třída *Processor* obsahuje informace o jméně jednotky, jejím identifikačním čísle a latenci. V případě vícenásobného použití identické jednotky je jméno modifikováno přidáním čísla na konec. Instance třídy *Variable* jsou pak obdobně naplněny daty o proměnných a konstantách. Později dojde k doplnění správných časů zápisu a čtení z proměnné, a tak *Variable* poskytuje i výpočet, kolik registrů bude při dané délce periody potřeba pro uchování hodnoty proměnné. Další důležitou vlastností *Variable* je typ - `memory, input, output, constant` nebo `unused`. Kvůli načítání inicializačních hodnot proměnných a konstant je navíc v programu nastaveno anglické prostředí tak, aby jako oddělovač desetinného místa byla chápána tečka a nikoli čárka.

Dále přijde na řadu zpracování časových údajů. Instance třídy *Task* (obr. 2.7) jsou naplněny referencemi na daný procesor, vstupní a výstupní proměnné. Ke každé vstupní proměnné je dále uložena informace o znaménku, se kterým do procesoru vstupuje. Podle tabulky s relacemi následnosti dojde k upravení doby počátků operací. Pokud je výška[4] nenulová, tedy parametr $h > 0$, je potřeba k času počátku operace načtenému z XML přičíst h-násobek doby periody. Informace o počátku operace slouží spolu s informací o čase

[4] Například vztahu $a\{k\} = t * b\{k-2\}$ odpovídá $h = 2$



Obrázek 2.7: Propojení tříd *Task, Processor* a *Variable*

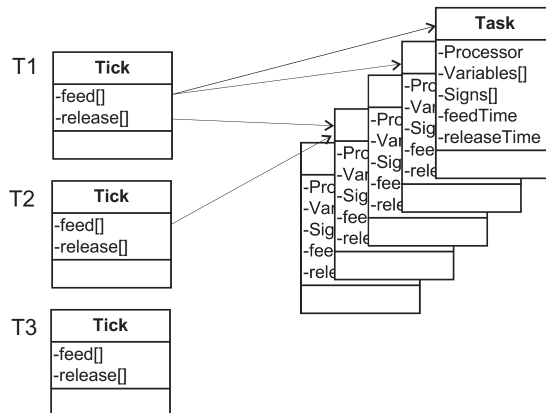
prvního validního výsledku k určení počtu registrů pro zapamatování hodnoty proměnné. Průchodem všech Tasků je nalezen nejzazší čas, při kterém je proměnná přečtena. Zápis do proměnné je prováděn během iterace pouze jednou. Rozdíl mezi okamžikem zápisu do proměnné a nejzazším okamžikem čtení je v instanci třídy *Variable* uložen jako hodnota *useLen*. Pokud by uživatel ve vstupním souboru ACGM specifikoval proměnnou, která by dále nebyla použita, k jejímu deklarování ve VHDL souboru nedojde. Takovýto případ se pozná jednoduše, okamžik čtení totiž zůstane na inicializační hodnotě *int.MinValue*.

Všechna získaná data jsou uložena v instanci třídy *TaskSet*, aby byl umožněn kompaktní přístup a jednoduché předávání všech hodnot.

2.9 Generování VHDL

Vstupním parametrem do třídy generující výstupní kód je právě výše zmíněný *TaskSet*, sdružující všechna potřebná data. Pro účely generování kódu je vytvořena třída *Tick*, jejíž každá instance se vztahuje k jednomu hodinovému cyklu rozvrhu. Dojde tedy k vytvoření pole prvků *Tick* o délce periody algoritmu. Poté program prochází jednotlivé instance třídy *Task* uložené v *TaskSetu* a ukládá reference na ně do příslušného prvku pole *Tick*. Každý *Task* je uložen jednak v čase odpovídající okamžiku, kdy je procesor plněn daty, a jednak v okamžiku čtení výstupu procesoru. Tyto dva případy jsou při ukládání referencí odlišeny konstantou udávající směr přenosu dat.

VHDL kód začíná definicí použitých knihoven IEEE. Následuje definice entity, kde je



Obrázek 2.8: Asociace *Task* k jednotlivým hodinovým tikům *Tick*

definováno rozhraní obvodu. Kromě pevně určeného hodinového a resetovacího vstupu je pro každou aritmetickou jednotku vytvořen potřebný počet portů pro vstupní operandy a jeden port pro výsledek. Další porty generované do VHDL slouží pro vstup dat do obvodu a pro výstup výsledku zpracování. Ke každému vstupu/výstupu dat také patří jednobitový signál - u vstupu se jedné o požadavek na novou vstupní hodnotu, u výstupu je tímto signálem značen nový validní výsledek. Zapsáním všech těchto portů do VHDL kódu je ukončena definice entity.

Vlastní architektura pak začíná vytvořením výčtového typu *stavy*, který je potřeba pro realizaci stavového automatu. Následně vzniknou dva signály state, nstate tohoto typu. Pro každou proměnnou vstupního algoritmu je nadefinován potřebný počet signálů pro registry, někdy není potřeba žádný, jindy jich je naopak několik navazujících na sebe. Registry jsou pojmenovány stejně jako proměnné, je-li potřeba více než jeden, přibírají ty další pořadová čísla.

Výkonný kód začíná synchronizačním procesem. Zde dojde k vytvoření vlastních registrů a synchronizaci automatu při přechodu do dalšího stavu. Vytvoření registrů znamená projít seznam všech proměnných a u všech, které jsou použity, zapsat propojení vstupního signálu na výstupní. Další proces má na starosti přechod mezi stavy automatu. V případě plné verze automatu se postupně přechází přes všechny indexy „for“ smyčky. U redukovaného automatu je potřeba při generování přechodů mezi stavy vynechávat ty stavy, které jsou „prázdné“.

Proces definující datové cesty mezi procesory a registry vyžaduje ve svém sensitivity-listu seznam všech použitých registrů, tedy jsou postupně přidávána jména již dříve vy-

tvořených registrů, a dále také výstupní signály z aritmetických jednotek. Smyčka přes všechny hodinové cykly periody prochází jednotlivé položky pole *Ticks* a je-li množina operací neprázdná, je vytvořena položka *case* a zapsán patřičný datový přesun. Pakliže je potřeba uložit výsledek výpočtu do registru (daná proměnná má nastaven parametr $useLen > 0$, viz sekce 2.8), je vygenerován kód pro zapsání hodnoty do registru. V případě každé hodnoty vstupující do procesoru je naopak potřeba zkontrolovat, zda načítání hodnoty neprobíhá ve stejném čase jako její zápis. Pokud ano, je nutno provést přímý přesun výsledku mezi jednotkami. V případě přímého přesunu je nalezen procesor produkující žádanou hodnotu a jeho výstup je zaveden na vstup dalšího procesoru. Když je hodnota čtena z registru, je potřeba zjistit, jak dlouho už se daná hodnota v registru nachází. Pokud je totiž tato doba delší než délka periody algoritmu, je již žádaná hodnota přesunuta do dalšího registru (viz sekce 2.4) a proto je potřeba číst data ze správného zdroje.

Proces starající se o řízení výměny zpracovávaných dat je generován tak, že je pro každý datový kanál zapsána podmínka kontrolující, zda právě v daném hodinovém cyklu byla přečtena ze vstupu nebo zapsána na výstup hodnota. Jestliže ano, je pro daný kanál nastavena log. '1' na signálu značícím validitu výsledku nebo požadavek na nový vstup. Tento asynchronní způsob výměny dat byl zvolen z důvodu různé periody algoritmů. Pro jednu konkrétní aplikaci se známou periodou by bylo možno přesně nastavit časování okolních obvodů a tento VHDL proces vynechat.

Generování redukované verze automatu je implementováno ve třídě *VhdlGenSmall*, odvozené od původní *VhdlGen* pro plný automat. Výčtový typ specifikující množinu stavů automatu je nyní vygenerován pouze pro ty stavy, kde není nulový počet operací. Mezi seznam signálů přibude povolení hodin automatu a signály pro vytvoření a ovládání čítače. Bitová šířka čítače je určena z maximální doby čekání jako $\text{floor}(\log_2 maxWait) + 1$. Proces který řídí změny stavu automatu je modifikován tak, aby bylo přecházeno jen mezi neprázdnými stavy. Struktura s registry je upravena tak, aby byla řízena signálem povolení hodin.

Generování procesu vytvářejícího multiplexory bylo rovněž změněno. Seznam podmínek, odpovídající řídícímu signálu multiplexoru, obsahuje pouze neprázdné stavy. Také je potřeba určit počet hodinových cyklů do dalšího neprázdného stavu, a tuto hodnotu pro každý stav zapsat na signál vstupu blokovacího čítače.

Synchronizační proces je obdobný jako v případě plného automatu, jen zde opět přibyla podmínka na platnost povolení hodin automatu. Je totiž potřeba zajistit, aby log. '1' na daném vodiči byla pouze po dobu jednoho hodinového cyklu. Pokud by '1' byla

vázaná pouze na stav automatu, mohla by se na výstupu objevit po delší dobu a způsobit ztrátu některých dat.

Čítač, jehož úlohou je zastavování chodu automatu, je dalším procesem. Struktura je pevně daná, mění se jen bitová šířka dat, se kterými čítač pracuje.

2.10 Konverze čísel

Jazyk VHDL umí pracovat pouze s binární reprezentací čísel. Použitím knihoven IEEE jsou navíc zpřístupněna i celá čísla ve dvojkovém doplňku. Jelikož je potřeba pracovat s desetinnými čísly v pevné řádové čárce, nebo dokonce s čísly v plovoucí řádové čárce, je potřeba vstupní konstanty převést na odpovídající binární reprezentaci. Teprve tuto binární reprezentaci je možno do VHDL kódu uložit.

Pevná řádová čárka

Pro konverzi čísel na straně PC (v generátoru) do reprezentace v pevné řádové čárce lze s výhodou využít datové reprezentace celých čísel v počítači [13]. Vstupující desetinné číslo je nutno vynásobit hodnotou 2^{n-1} kde n je počet míst za desetinnou čárkou. Tato hodnota je posléze převedena na celé číslo, které je v paměti PC uloženo ve dvojkovém doplňku. Postupným vymaskováním jednotlivých bitů se zrekonstruuje binární reprezentace vstupního čísla. Výstupem je textový řetězec skládající se ze znaků ’1’ a ’0’.

2.11 Ošetření chyb

Většina možných chyb neumožňuje nápravu a pokračování programu, například chyba v XML, nebo špatný rozvrh, kde nejsou v pořádku časové závislosti. V tom případě je vypsána chybová hláška a program je ukončen.

2.12 Výstup VHDL

Název vstupního XML souboru je do programu zadán jako řádkový parametr, nebo na vyžádání po spuštění programu. Generování proběhne pro obě varianty automatu a kód je uložen do dvou souborů `outvhdlF.vhd` pro plný automat a `outvhdlR.vhd` pro redukovanou verzi.

Kapitola 3

Experimenty

Při návrhu algoritmu v Matlabu ihned vidíme, zda navržený algoritmus funguje správně. S týmž algoritmem, optimalizovaným pro FPGA, to již tak jednoduché není. Máme sice simulační nástroje, ovšem bez výrazné podpory desetinných čísel, bez kterých výpočty nemají v podstatě smysl. Navíc v simulaci může leckdy fungovat i to, co ve skutečnosti selže. A tak nezbývá než provést syntézu, nahrát do vytvořený obvod FPGA, propojit s počítačem a otestovat.

Testování je navrženo tak, aby vše probíhalo přímo z okna Matlabu.

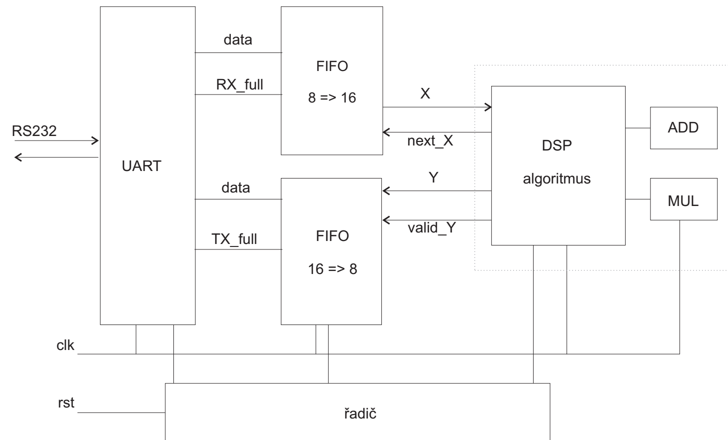
3.1 Testovací rozhraní v FPGA

Na straně FPGA vytvoříme zapojení dle obr. 3.1. Šířka datových sběrnic je kompromisem mezi použitelným rozsahem čísel na jedné straně a velikostí FPGA designu spolu s časem syntézy na straně druhé. Jelikož pro připojení k počítači prozatím slouží sériová linka[1], nejjednodušší by byla šířka 8 bitů a použití celých čísel. Při této konfiguraci však velmi brzy dojde k přetečení hodnot a výsledky nelze porovnávat s běžným výpočtem.

Aby tedy byla zajištěna dostatečná přesnost výpočtu a zároveň jednoduchost (kvůli rychlosti syntézy obvodu), je pro testování použit formát čísel v pevné řádové čárce s datovou šířkou 16 bitů. Jedná se o dvojkový doplněk, řádová čárka je umístěna uprostřed, mezi 7. a 8. bitem.

Celý proces probíhá následovně: Data po sériové lince přichází do FPGA, kde postupně plní vstupní zásobník FIFO. Zásobník má asymetrické datové porty. Po naplnění prvního

[1] RS232, 9600baud/s, 8 bitů



Obrázek 3.1: Blokové schéma zapojení v FPGA

zásobníku spouští řadič stavový automat—testovaný algoritmus. Data jsou postupně odebírána jako 16-ti bitové hodnoty, zpracovávána a výsledky je plněn druhý zásobník. Po zpracování všech dat řadič zastaví automat a aktivuje odesílání zpět po sériové lince.

Dále se budeme zabývat jednotlivými částmi řetězce, který zajišťuje proces testování.

3.1.1 Aritmetické jednotky

Při skutečném využití algoritmu se počítá s využitím jednotek počítajících v plovoucí řádové čárce, například HSLA [9] nebo FP32[2]. Pro účely testování jsou použity pouze jednotky pro pevnou řádovou čárku, u kterých je potřebná latence simulována.

Sčítání a násobení celých čísel je přímo podporováno pomocí VHDL. Jelikož je testování prováděno se 16-ti bitovými daty v pevné řádové čárce (formát 8.8), jedná se také o čísla reprezentovaná ve dvojkovém doplňku a lze použít aritmetické jednotky, které jsou ve VHDL k dispozici pro celá čísla. Sčítání a odčítání funguje naprosto stejně pro celá i desetinná čísla, v případě násobení ($16b * 16b \rightarrow 32b$) je potřeba jako výsledek ukládat bity v rozmezí 23:8 [13]. Desetinná čárka výsledku se totiž po násobení nachází na pozici mezi 16. a 17. bitem.

Při testování s *fixed-point* jednotkami, které mají latenci 1, musíme simulovat latenci složitějších jednotek. Aby nebylo potřeba kvůli každé nové hodnotě latence programovat novou odpovídající testovací jednotku, nabízí se možnost parametrizace popisovaného

[2]Pipelinované jednotky firmy Celoxica

obvodu. Tu dovoluje konstrukt `generic` [4], pomocí nějž se definují potřebné konstanty. Vše pak funguje na stejném principu, jako například podmíněný překlad v běžných programovacích jazycích.

3.1.2 FIFO

Fronta FIFO (*First In First Out*) s různou šířkou vstupních a výstupních dat je dostupný pomocí Xilinx IP generátoru [18]. Avšak nastal zde problém, že vygenerovaný obvod stále signalizoval, že je již naplněný. Jelikož korektní funkci tohoto příznaku vyžaduje navržený řadič, bylo nutno naprogramovat FIFO vlastní. Jako jednoúčelový blok pro potřebu testování postačí i zjednodušená varianta, která umožní jen jedno naplnění a přečtení, po kterém musí být proveden reset. Ten je zajištěn řadičem.

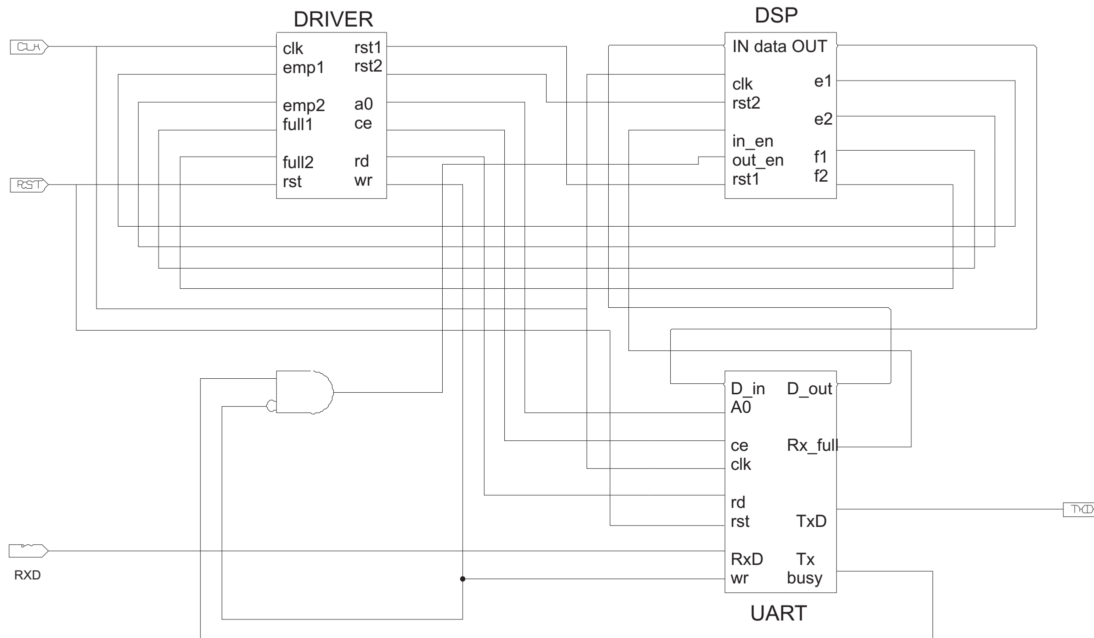
3.1.3 UART

Modul pro sériový port není jako IP core dostupný a jeho návrh by byl zbytečným zdržením. Na Internetu ⟨www.hp-h.com/p/munte⟩ lze nalézt hotový VHDL kód dostupný pod GNU licencí, který je možno použít po drobné úpravě i v této aplikaci.

3.1.4 Řadič

Řadič (na obr. 3.2 blok driver) je malý stavový automat se čtyřmi stavy, reprezentující počáteční reset, přenos dat z počítače, zpracování v FPGA a konečně přenos výsledku zpět do PC. Předávání dat mezi UART a FIFO (zásobníky FIFO jsou ve schématu na obr. 3.2 obsaženy v bloku DSP) lze řídit propojením výstupu `RX_full` na load (`in_en`) fronty FIFO, tedy po přijetí každého bytu je tento zapsán do fronty a zvýšen index ukazující na volnou pozici. Funkci testovaného algoritmu během přijímání vstupních dat blokuje řadič aktivním resetem `rst2`. Naplnění vstupní fronty je signalizováno aktivním bitem `f1` (full1). Řadič přechází do dalšího stavu. Uvolní reset `rst2` testovaného bloku a čeká na naplnění výstupní fronty. Aktivní bit `f2` značí naplnění druhé fronty. Je-li druhá fronta plná, řadič povolí odesílání dat přes sériovou linku nastavením `wr` = '1'.

Během odesílání bytu je aktivní signál `Tx_busy`. Další byte pro odeslání je tedy z výstupní fronty čten v okamžiku, kdy je `Tx_busy`='0', a zároveň je vysílač zapnut signálem `wr`. Tato podmínka je dána logickou operací *AND*, jak ukazuje obr. 3.2. Signál `out_en`



legenda signálů:

clk - hodiny
rst - reset

rst1 - reset front FIFO
rst2 - reset testovaného automatu
e1,e2,f1,f2 - FIFO empty,full
in_en - zapsat byte do FIFO1
out_en - přečíst byte z FIFO2
IN - data do FIFO1
OUT - data z FIFO2

ce - povolení UART
rd - číst data z TxD
wr - odeslat data na TxD
A0 - mód UART
Rx_full - byte přijat
D_out - přijatý byte
Tx.busy - probíhá odesílání
D_in - byte k odeslání

Obrázek 3.2: Skutečné schéma zapojení z Xilinx ISE

slouží k inkrementaci výstupního ukazatele v FIFO, tedy k získání další hodnoty pro odeslání po sériové lince.

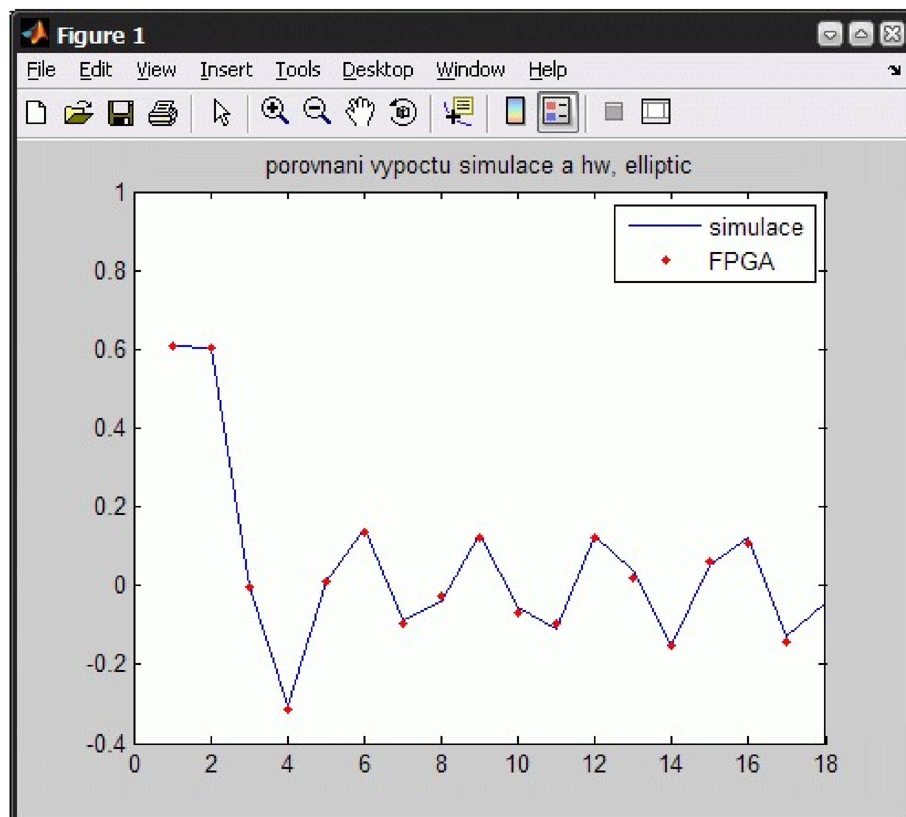
Další signály v bloku DSP na obr. 3.2 jsou `e1`, `e2` signalizující vyprázdnění FIFO. Bitem `rst1` je prováděno vynulování front FIFO, `rst2` je použit pro reset testovaného stavového automatu. Blok UART musí být povolen pomocí *chip enable* `ce`, příjem se zapíná bitem `rd` a vysílání bitem `wr`. Bit `A0` slouží k přepnutí módu UARTu a není využit. `D_out` je osmibitová sběrnice, na které se objevují přijatá data. Naopak na `D_in` jsou přiváděna data pro odeslání do PC.

3.2 Testovací rozhraní v Matlabu

Matlab od verze 7.1 umožňuje jednoduchý přístup na sériový port instalovaný v počítači. Příkazem `port = serial('COM1');` dojde k vytvoření handleru zařízení a dále můžeme s portem pracovat jako s běžným souborem.

Pro převod čísel, která jsou v Matlabu reprezentována v plovoucí řádové čárce, je možno použít *Fixed Point Toolbox*. Stačí nadefinovat formát čísel `quantizer([16 8]);` značící 16bitové číslo s desetinnou čárkou uprostřed. Převedená čísla dostáváme jako string, ze kterého je možné jednoduše postupně odebírat jednotlivé osmice bitů a ty následně odesílat na sériový port.

Pro opačný směr je potřeba přijaté byty převést na binární reprezentaci a z jednotlivých dvojic bytů skládat 16-ti bitové řetězce. Ty lze nakonec převést na desetinné číslo funkcí `bin2num`.



Obrázek 3.3: Porovnání výpočtu v FPGA a v Matlabu

Z důvodu rozdílné přesnosti výpočtů v Matlabu a v FPGA bylo porovnání výsledků znázorněno graficky, příklad výstupu pro filtr Elliptic [10] vidíme na obr. 3.2. Tenká čára představuje spojnice hodnot spočtených v Matlabu, body potom hodnoty, které byly

vypočteny v hradlovém poli.

Kód pro testování je uveden v příloze.

3.3 Výsledky experimentů

Výše uvedeným způsobem byly v FPGA testovány filtry WDF [11], Elliptic [10] a IIR7 [12]. Ve všech případech bylo dosaženo v hradlovém poli správných výsledků. V tabulkách 3.1, 3.2 a 3.3 jsou uvedeny následující parametry: písmeno F respektive R u typu automatu označuje typ automatu - plný nebo redukovaný (viz sekce 2.3). Počet ekvivalentních hradel odpovídá počtu použitých logických hradel v případě, že by byl obvod vytvořen z diskrétních součástek. Počet použitých registrů souvisí s časovým rozvrhem – určitý počet registrů je vyeliminován přímými přesuny mezivýsledků mezi jednotkami. Typ kódování a jeho datová šířka ukazují, jakým způsobem bylo provedeno zakódování stavů daného automatu. Kód je vybírán syntetizátorem automaticky. Maximální hodinová frekvence je kmitočet řídícího hodinového signálu, při kterém je ještě zajištěna správná činnost obvodu. Nakonec je uvedena bitová šířka blokovacího čítače – tento čítač je obsažen pouze v redukovaném automatu.

3.3.1 WDF

Algoritmus WDF je tvořen smyčkou obsahující 8 aritmetických operací. Uvedené výsledky se vztahují k jednodušší verzi pracující s osmibitovými čísly.

V tabulce 3.1 jsou porovnány stavové automaty pro různé realizace WDF filtru: pro latenci sčítačky $l_+ = 3$ a násobičky $l_* = 1$ s periodou rozvrhu $w = 8$, dále pro latence $l_+ = 9$ a $l_* = 2$ (odpovídá jednotkám *HSLA* [9]) s periodou $w = 29$, pro tytéž jednotky s processing time 5 a periodou $w = 31$ a nakonec pro latence $l_+ = 50$ a $l_* = 10$, kde perioda byla $w = 160$.

Tabulka 3.1: Porovnání automatů pro WDF filtr

typ	počet ekviv. hradel	počet 8bit reg.	kódování stavů	počet stavů	f_{clk} [MHz]	čítač [b]
8F	1032	6	gray:4b	9	338	-
8R	1184	6	gray:3b	6	377	2
29F	852	5	johnson:15b	30	324	-
29R	1429	5	gray:4b	11	335	3
31F	900	6	johnson:16b	32	308	-
31R	1249	6	gray:4b	14	342	2
160F	2040	5	johnson:81b	161	249	-
160R	1151	5	gray:4b	11	243	7

3.3.2 Elliptic

Algoritmus je tvořen smyčkou se 34 operacemi, z toho 4 připadají na násobení a 30 na sčítání nebo odčítání.

Implementovaný algoritmus elliptic má 16-ti bitové datové sběrnice a počítá s hodnotami v pevné řádové čárce a formátu 8,8. Perioda je $w = 97$.

Tabulka 3.2: Porovnání automatů pro filtr elliptic

typ	počet ekviv. hradel	počet 8bit reg.	kódování stavů	počet stavů	f_{clk} [MHz]	čítač [b]
F	5555	17	johnson:49b	98	295	-
R	5081	17	one-hot:38b	38	232	4

3.3.3 IIR7

V rozvrhu je jen jeden prázdný stav, proto lze předem říci, že redukovaný automat bude náročnější na velikost designu. Zmenšení automatu o jeden stav nemůže eliminovat zesložitění samotného automatu o blokování hodin a logiku potřebnou na nově vytvořený blokovací čítač. Tento odhad potvrzuje tabulka 3.3.

Tabulka 3.3: Porovnání automatů pro filtr IIR7

typ	počet ekviv. hradel	počet 8bit reg.	kódování stavů	počet stavů	f_{clk} [MHz]	čítač [b]
F	8154	23	seq:5b	21	287	-
R	8370	23	one-hot:20b	20	539	1

3.3.4 Porovnání obou typů automatů

Redukovaná verze automatu snižuje počet stavů automatu, ale na druhou stranu přidává režii spojenou se zavedením signálu `clk_en` pro povolení hodinového signálu v automatu a dále vytvořením čítače, který ovládá tento povolovací signál. Ve výsledku tedy například pro velmi jednoduchý filtr WDF dává redukovaný automat horší výsledek než automat plný. Se zvětšováním počtu stavů automatu a nárůstem počtu prázdných stavů začíná být redukovaná verze efektivnější, nicméně hranici nelze přesně stanovit.

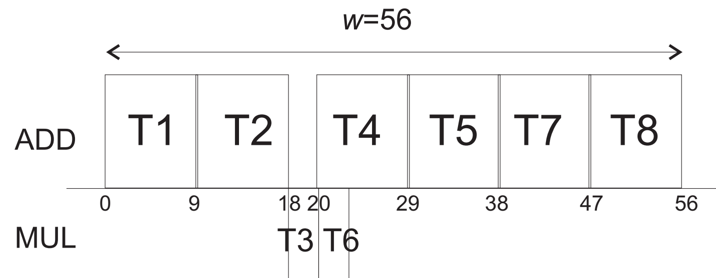
3.4 Porovnání se SPARK

Bylo také provedeno porovnání rozvrhovací schopnosti SPARKu a ACGM. Spark neumožňuje definovat nekonečné smyčky a tedy takové iterativní algoritmy, na které se zaměřuje ACGM. Simulujme například situaci, kdy by bylo potřeba vytvořit rozvrh pro počítání na jednotkách HSLA s latencemi $l_{add} = 9$ a $l_{mul} = 2$ a SPARK by byl použit pro částečné ulehčení práce, s tím, že výstupní kód je nutno ručně upravit. SPARK umí pracovat pouze s jednotkami bez pipeliningu. Algoritmus filtru WDF [11] zapsaný v jazyce C do smyčky *for* byl SPARKem rozvržen na aritmetické jednotky s požadovanou latencí.

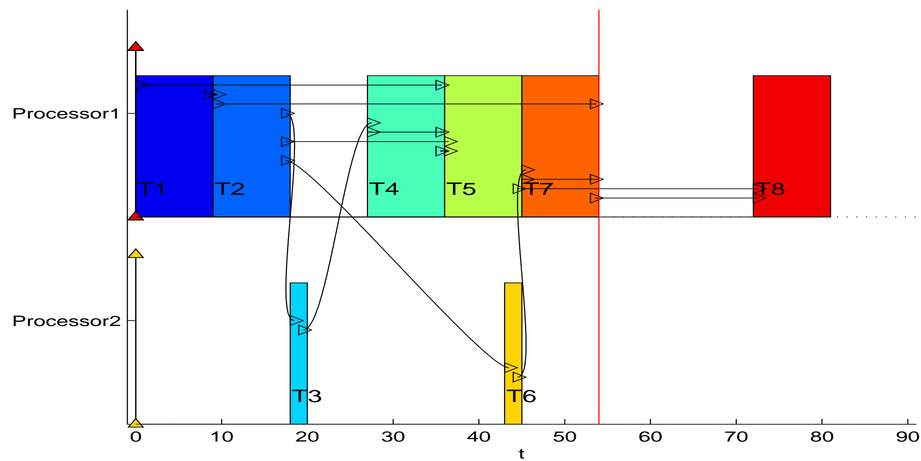
Následnou analýzou výstupního VHDL souboru lze zjistit, že SPARK vygeneroval zbytečně 4 registry do kterých je sice zapisováno, ale už nejsou čteny. Když neuvažujeme počáteční stavy vygenerovaného automatu spojené s režií ohledně řízení smyčky *for*, dostáváme rozvrh s periodou $w = 56$ (obr. 3.4).

Výsledkem rozvrhování téhož filtru pomocí ACGM dostáváme rozvrh s periodou $w = 54$, ve kterém se prolínají dvě iterační smyčky (obr. 3.5).

Pokud navíc připustíme, že jednotky podporují pipelining, zmenší se délka iterační smyčky na $w = 29$.



Obrázek 3.4: Ganttův diagram rozvrhu ze SPARKu (w=56)



Obrázek 3.5: Ganttův diagram rozvrhu z ACGM (w=54)

3.4.1 Výsledek porovnání

Jako nevýhodu SPARKu pro danou oblast použití je nutno považovat absenci podpory pipeliningu a neschopnost prolnutí iteračních smyček, které by v případě složitějších algoritmů vedlo k mnohem větším rozdílům v délce iterační periody, a v důsledku pak i ke snížení rychlosti zpracování dat.

Kapitola 4

Závěr

Práce dokončila řetězec ACGM - automatického generátoru kódu pro Matlab. Byla navržena struktura pro implementaci iteračních algoritmů jak s jedinou smyčkou, tak se smyčkami vnořenými.

Jednoduché smyčky jsou reprezentovány dvěma typy stavových automatů, výběr efektivnějšího z nich závisí na struktuře algoritmu a použitých aritmetických jednotkách, na které byl algoritmus rozvržen. Pro algoritmy s jednoduchými smyčkami se otevírá možnost dále zefektivňovat implementaci snižováním počtu použitých registrů a optimalizací multiplexorů.

Zavedení vnořených smyček se ukázalo jako mnohem složitější problém, než jen pouhé hierarchické propojení několika „jednosmyčkových" automatů. Navržená struktura se zdá být funkční, ale pravděpodobně není vhodná pro složitější a rozsáhlejší algoritmy.

Generátor kódu byl vytvořen pro oba typy algoritmů. Výsledek syntézy generovaného VHDL kódu pro jednoduché smyčky byl pomocí navrženého testovacího řetězce ověřován v hardwaru. Testování probíhalo přímo z Matlabu.

Literatura

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

[2] Z. Pohl, P. Šůcha, J. Kadlec, a Z. Hanzálek. *Performance Tuning of Iterative Algorithms in Signal Processing.* The International Conference on Field-Programmable Logic and Applications, Tampere, Finland, 2005.

[3] David Matějíček. *Optimalizace algoritmů pro FPGA*. Diplomová práce, Katedra řídící techniky ČVUT-FEL, 2007.

[4] Salcic, Z. a Smailagic, A. *Digital Systems design And Prototyping.* Kluwer Adademic Publishers, 2000.

[5] Pong P. Chu. *RTL Hardware Design Using VHDL.* Wiley-Interscience, 2006.

[6] S. Gupta, R.K. Gupta, N.D. Dutt a A. Nicolau. *SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits.* Kluwer Adademic Publishers, 2004.

[7] R. Nouta, H. J. Lincklaen Arriëns. *Design and FPGA-Implementation of Wave Digital Bandpass Filters with Arbitrary Amplitude Transfer Characteristics.* Faculty of EEMCS, Delft University of Technology, 2003.

[8] H. J. Lincklaen Arriëns. *(L)WDF Toolbox for Matlab - User's guide.* Faculty of EEMCS, Delft University of Technology, 2006.

[9] J. Kadlec, J. Schier. *HSLA DSP Package.* (Research report no. 1924). UTIA AV ČR, 1998.

[10] 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*, 1997.

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

[12] J. M. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. Fast prototyping of datapath-intensive architectures. *IEEE Des. Test*, 8(2):40–51, 1991.

[13] Tišnovský P. *Fixed point arithmetic.* [online] ⟨www.root.cz⟩.

[14] W3C. *Xpath tutorial.* [online] ⟨www.w3schools.com/xpath⟩.

[15] Synplicity. *True DSP Synthesis for Fast, Efficient, High-Performance FPGA Implementations*, 2005.

[16] Altera. *DSP Builder 6.0 - User Guide*, 2006. [online].

[17] Xilinx. *AccelDSP Synthesis Tool Floating-Point to Fixed-Point Conversion of MATLAB Algorithms.* WP239, 2006.

[18] Xilinx. *FIFO Generator - Product Specification*, 2006.

[19] Xilinx. *XST User Guide*, 2006.

Příloha A

Kódy pro příklad bez vnořených smyček

A.1 Algoritmus v ACGM

```
function Y = priklad(X)
%Arithmetic Units Declaration
struct('operator','+','number',1,'proctime',1,'latency',3,
       'feedoper','add','getoper','add\_out');
struct('operator','*','number',1,'proctime',1,'latency',1,
       'feedoper','mul','getoper','mul\_out');
%Memmory Units Declaration
struct('memory','bram','var',{'n1','n2','n3','n4'},'ports',2);
struct('memory','bram','var',{'a','b'},'ports',2);
struct('memory','register','var',{'X','Y','K'});
%Variables Declaration
K=25;
a=-0.375;
b=0.5;
n4{1}=zeros;
n3{1}=zeros;
%Iterative Algorithm
for k=2:K-1
    n1{k} = X{k} + n4{k-1};
    n2{k} = a * n1{k};
    n3{k} = n3{k-1} - n1{k};
    n4{k} = n2{k} + n3{k};
    Y{k}  = b * n4{k};
end
```

A.2 Plný automat

```
entity automat is            -- za dvema pomlckami je komentar
generic(dw : integer := 16); -- sirka datovych sbernic
port(clk : in STD_LOGIC;
  rst : in STD_LOGIC;
```

```
  add_IN0 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  add_IN1 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  add_OUT : in  STD_LOGIC_VECTOR(dw-1 downto 0);
  mul_IN0 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  mul_IN1 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  mul_OUT : in  STD_LOGIC_VECTOR(dw-1 downto 0);
  X_v     : in  STD_LOGIC_VECTOR(dw-1 downto 0);
  next_X  : out STD_LOGIC;
  Y_out   : out STD_LOGIC_VECTOR(dw-1 downto 0);
  valid_Y : out STD_LOGIC );
end entity automat;

architecture priklad_full of automat is
type stavy is (sidle, sT0, sT1, sT2, sT3, sT4, sT5, sT6, sT7, sT8);
signal state, nstate: stavy;
  signal n2, n2_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n2
  signal n3, n3_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n3
  signal n4, n4_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n4
  constant a_v : STD_LOGIC_VECTOR(dw-1 downto 0) := "1111111110100000";
  constant b_v : STD_LOGIC_VECTOR(dw-1 downto 0) := "0000000010000000";
  signal Y_v,Y : STD_LOGIC_VECTOR(dw-1 downto 0);    -- registr pro výstup
begin

stavy: process(state)
begin
  case(state) is
    when sidle => nstate <= sT0;
    when sT0 => nstate <= sT1;
    when sT1 => nstate <= sT2;
    when sT2 => nstate <= sT3;
    when sT3 => nstate <= sT4;
    when sT4 => nstate <= sT5;
    when sT5 => nstate <= sT6;
    when sT6 => nstate <= sT7;
    when sT7 => nstate <= sT8;
    when sT8 => nstate <= sT0;
  end case;
end process prech;

registry: process(clk,rst)
begin
if rst='0' then   -- reset aktivni v '0'
  state <= sidle; -- inicializacni stav
  n2_v <= (others => '0'); -- vynulovani registru
  n3_v <= (others => '0');
  n4_v <= (others => '0');
  Y_v  <= (others => '0');
elsif rising_edge(clk) then -- pri nabezne hrane hodin
  state <= nstate;
  n2_v  <= n2;  -- popisuje flip-flop typu D
  n3_v  <= n3;  --
  n4_v  <= n4;  --
  Y_v   <= Y;
end if;
end process sync;

 Y_out <= Y_v;

vystup: process(state, n2_v, n3_v, n4_v, X_v, Y_v, add_OUT, mul_OUT)
begin
    add_IN0 <= (others => '0');  -- defaultni hodnota
    add_IN1 <= (others => '0');  -- na vstupu procesoru
    mul_IN0 <= (others => '0');
    mul_IN1 <= (others => '0');
    n2 <= n2_v ;  -- zpetna vazba pro zapamatovani posledni hodnoty
    n3 <= n3_v ;  --
    n4 <= n4_v ;  --
    Y <= Y_v ;    --
case(state) is
  when sT0 =>          -- T0
    n4 <= add_OUT;
    Y <= mul_OUT;
    add_IN0 <= X_v;
    add_IN1 <= add_OUT;
```

```
  -- T1 is empty       -- T1
  -- T2 is empty       -- T2
  when sT3 =>          -- T3
    mul_IN0 <= a_v;
    mul_IN1 <= add_OUT;
    add_IN0 <= n3_v;
    add_IN1 <= (add_OUT xor x"FFFF") + x"0001"; -- dvojkový doplněk
  when sT4 =>          -- T4
    n2 <= mul_OUT;
  -- T5 is empty       -- T5
  when sT6 =>          -- T6
    n3 <= add_OUT;
    add_IN0 <= n2_v;
    add_IN1 <= add_OUT;
  -- T7 is empty       -- T7
  when sT8 =>          -- T8
    mul_IN0 <= b_v;
    mul_IN1 <= n4_v;
   when others => null;
end case;
end process output;

hshake: process(clk,rst)
begin
 if rst='0' then
   next_X <= '0';
   valid_Y <= '0';
 elsif rising_edge(clk) then
   next_X <= '0';
   valid_Y <= '0';
   if state = sT0 and aut_en='1' then
     next_X <= '1';
   end if;
   if state = sT0 and aut_en='1' then
     valid_Y <= '1';
   end if;
 end if;
end process hshake;

end architecture priklad_full
```

A.3 Redukovaný automat

```
entity automat is           -- za dvema pomlckami je komentar
generic(dw : integer := 16); -- sirka datovych sbernic
port(clk : in STD_LOGIC;
  rst : in STD_LOGIC;
  add_IN0 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  add_IN1 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  add_OUT : in  STD_LOGIC_VECTOR(dw-1 downto 0);
  mul_IN0 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  mul_IN1 : out STD_LOGIC_VECTOR(dw-1 downto 0);
  mul_OUT : in  STD_LOGIC_VECTOR(dw-1 downto 0);
  X_v     : in  STD_LOGIC_VECTOR(dw-1 downto 0);
  next_X  : out STD_LOGIC;
  Y_out   : out STD_LOGIC_VECTOR(dw-1 downto 0);
  valid_Y : out STD_LOGIC );
end entity automat;

architecture priklad_redukovany of automat is
type stavy is (sidle, sT0, sT3, sT4, sT6, sT8);  -- redukovany
signal state, nstate: stavy;
  signal n2, n2_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n2
  signal n3, n3_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n3
  signal n4, n4_v : STD_LOGIC_VECTOR(dw-1 downto 0); -- registr n4
  constant a_v : STD_LOGIC_VECTOR(dw-1 downto 0) := "1111111110100000";
  constant b_v : STD_LOGIC_VECTOR(dw-1 downto 0) := "0000000010000000";
  signal Y_v,Y : STD_LOGIC_VECTOR(dw-1 downto 0);    -- registr pro výstup
```

```
  signal waitTicks, cnt : std_logic_vector(1 downto 0);
  signal aut_en : std_logic;   -- povoleni hodin pro automat
begin
stavy: process(state)
begin
  case(state) is
    when sidle => nstate <= sT0;
    when sT0 => nstate <= sT3;
    when sT3 => nstate <= sT4;
    when sT4 => nstate <= sT6;
    when sT6 => nstate <= sT8;
    when sT8 => nstate <= sT0;
  end case;
end process prech;
registry: process(clk,rst)
begin
if rst='0' then   -- reset aktivni v '0'
  state <= sidle; -- inicializacni stav
  n2_v <= (others => '0'); -- vynulovani registru
  n3_v <= (others => '0');
  n4_v <= (others => '0');
  Y_v  <= (others => '0');
elsif rising_edge(clk) then -- pri nabezne hrane hodin
 if aut_en = '1' then       -- blokovani hodin
  state <= nstate;
  n2_v <= n2;  -- pro vytvoreni registru
  n3_v <= n3;  -- pro vytvoreni registru
  n4_v <= n4;  -- pro vytvoreni registru
  Y_v <= Y;
 end if;
end if;
end process sync;
 Y_out <= Y_v;
vystup: process(state, n2_v, n3_v, n4_v, X_v, Y_v, add_OUT, mul_OUT)
begin
    add_IN0 <= (others => '0');  -- defaultni hodnota
    add_IN1 <= (others => '0');  -- na vstupu procesoru
    mul_IN0 <= (others => '0');
    mul_IN1 <= (others => '0');
    n2 <= n2_v ;  -- zpetna vazba pro zapamatovani posledni hodnoty
    n3 <= n3_v ;  --
    n4 <= n4_v ;  --
    Y <= Y_v ;    --
case(state) is
  when sT0 =>         -- T0
    n4 <= add_OUT;
    Y <= mul_OUT;
    add_IN0 <= X_v;
    add_IN1 <= add_OUT;
    waitTicks <= conv_std_logic_vector(2,2); -- zastavit na 2 tiky
  when sT3 =>         -- T3
    mul_IN0 <= a_v;
    mul_IN1 <= add_OUT;
    add_IN0 <= n3_v;
    add_IN1 <= (add_OUT xor x"FFFF") + x"0001" ; -- negace operandu
    waitTicks <= conv_std_logic_vector(0,2);
  when sT4 =>         -- T4
    n2 <= mul_OUT;
    waitTicks <= conv_std_logic_vector(1,2);
  when sT6 =>         -- T6
    n3 <= add_OUT;
    add_IN0 <= n2_v;
    add_IN1 <= add_OUT;
    waitTicks <= conv_std_logic_vector(1,2);
  when sT8 =>         -- T8
    mul_IN0 <= b_v;
    mul_IN1 <= n4_v;
    waitTicks <= conv_std_logic_vector(0,2);
   when others => waitTicks <= (others => '0');
end case;
```

```
end process output;

citac: process (clk, rst)
begin  -- process
  if rst = '0' then    -- asynchronni reset
    cnt <= "00";
  elsif clk'event and clk = '1' then
    if cnt = "00" then
      cnt <= waitTicks; -- nahrat novou hodnotu
    else
      cnt  <= cnt - 1;  -- dekrementace hodnoty, nebyla-li nulova
    end if;
  end if;
end process;
aut_en <= '1' when cnt="00" else '0'; -- povoleni nebo zakazani clk_en

hshake: process(clk,rst)
begin
 if rst='0' then
   next_X <= '0';
   valid_Y <= '0';
 elsif rising_edge(clk) then
   next_X <= '0';
   valid_Y <= '0';
   if state = sT0 and aut_en='1' then
     next_X <= '1';
   end if;
   if state = sT0 and aut_en='1' then
     valid_Y <= '1';
   end if;
 end if;
end process hshake;

end architecture priklad_redukovany
```

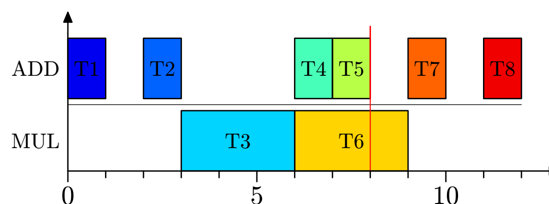
Příloha B

Algoritmy bez vnořených smyček

V této části jsou shrnuty výsledky rozvrhování a syntézy filtrů s iteračním algoritmem o jediné smyčce. Vždy je uveden Ganttův diagram a dále tabulka údajů ze syntézy obvodu: typ automatu (plný/redukovaný) podle sekce 2.2, maximální použitelná frekvence hodinového signálu pro řízení stavového automatu, počet ekvivalentních hradel (odpovídá realizaci diskrétními součástkami) a počet skutečně využitých bloků LUT[1]. Algoritmy jsou rozvrženy pro dva typy jednotek, z nichž v jednom případě vždy odpovídá latence jednotkám HSLA [9] ($l_+ = 9$, $l_* = 2$).

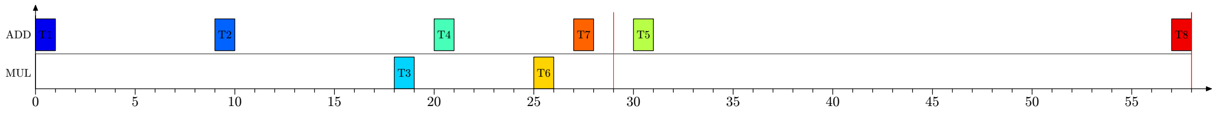
Veškeré dosažené výsledky se vztahují ke konfiguraci se 16-ti bitovými daty v pevné řádové čárce. Výsledek syntézy zahrnuje pouze vlastní automat, registry a multiplexory. Aritmetické jednotky je potřeba k tomuto zapojení připojit externě.

B.1 WDF



Obrázek B.1: Ganttův diagram pro WDF filtr

[1]Look-Up Table – základní blok FPGA

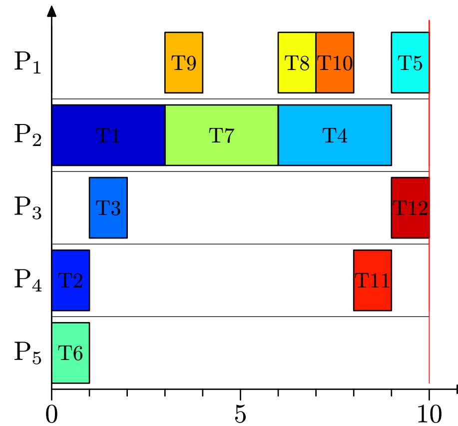


Obrázek B.2: Ganttův diagram pro WDF filtr s jednotkami HSLA

typ	f_{clk} [MHz]	ekv. hradel	LUTs
WDF F	267	2301	224
WDF R	356	2528	257
WDF HSLA F	548	2290	204
WDF HSLA R	256	2158	223

Tabulka B.1: Výsledky syntézy WDF

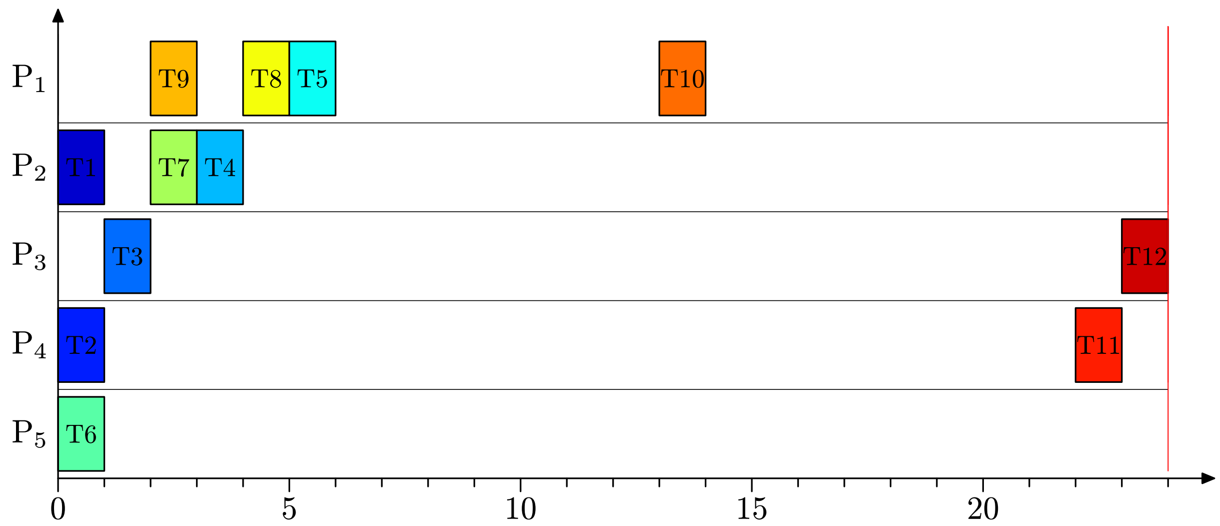
B.2 PSD



Obrázek B.3: Ganttův diagram pro PSD filtr

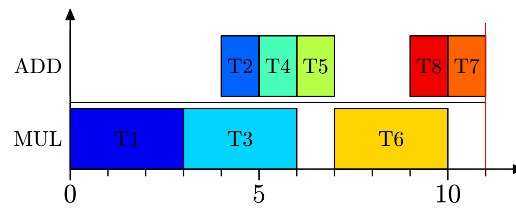
typ	f_{clk} [MHz]	ekv. hradel	LUTs
WDF F	350	2643	272
WDF R	276	2657	268
WDF HSLA F	271	3059	303
WDF HSLA R	314	3008	302

Tabulka B.2: Výsledky syntézy PSD

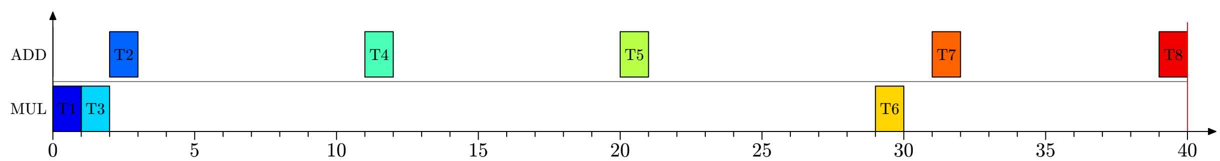


Obrázek B.4: Ganttův diagram pro PSD filtr s jednotkami HSLA

B.3 DSVF



Obrázek B.5: Ganttův diagram pro DSVF filtr

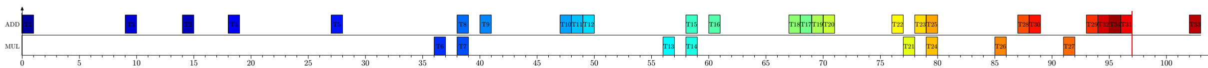


Obrázek B.6: Ganttův diagram pro DSVF filtr s jednotkami HSLA

typ	f_{clk} [MHz]	ekv. hradel	LUTs
DSVF F	316	2380	242
DSVF R	262	2409	242
DSVF HSLA F	545	2430	214
DSVF HSLA R	224	2271	240

Tabulka B.3: Výsledky syntézy DSVF

B.4 Elliptic



Obrázek B.7: Ganttův diagram pro filtr elliptic s jednotkami HSLA

typ	f_{clk} [MHz]	ekv. hradel	LUTs
Elliptic HSLA F	516	7214	658
Elliptic HSLA R	230	6780	627

Tabulka B.4: Výsledky syntézy filtu elliptic

B.5 Kód pro testování v Matlabu

```
clear;clc;                              %%% m-file pro testovani
data = [1 -1 1 -1 1 -1 1 -1 1 -1];       % vektor zpracovavanych dat

odeslat = conv16to8(data);               % data do FPGA ke zpracovani
prijato = transmitdata(odeslat);         % 8-mi bitova data z FPGA
vysledek = conv8to16(prijato);           % prevest na des. cisla

fprintf('ze simulace: ');
sim = wdf(data)                          % vypocist pomoci Matlabu

fprintf('Z FPGA: ');
fprintf('%6i',vysledek')
 % ------------------------------------------------------------

function y = conv16to8(x)                  % prevod cisel pro prenos po RS232
  q = quantizer([16 8]);                   % definice formatu FixPoint 8.8
  bin = num2bin(q,x);                      % vytvori matici bitu
  j = 1;
  for i = 1:size(bin,1)                    % pres vsechny 16bit hodnoty
```

```
      y(j)  = bin2dec(bin(i,1:8));     % dolnich 8 bitu
      y(j+1)= bin2dec(bin(i,9:16));    % hornich 8 bitu
      j     = j + 2;
  end
end

function y = conv8to16(data)
  j = 1;
  for i = 1:length(data)/2
      bin(i,1:8) = dec2bin(data(j),8);
      bin(i,9:16)= dec2bin(data(j+1),8);
      j          = j + 2;
  end
  q = quantizer([16 8]);
  y = bin2num(q,bin)';
end

function zpet = transmitdata(data)
    com = serial('COM1');            % handler portu
    fopen(com);                      % otevrit port
    fwrite(com,data');               % odeslat data
    zpet = fread(com,length(data))'; % prijmout vysledek
    fclose(com);                     % uzavrit port
end
```

Příloha C

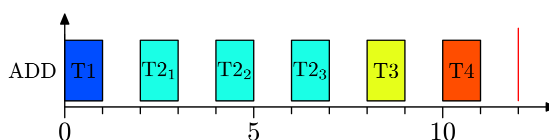
Algoritmy s vnořenými smyčkami

V této části jsou popsány výsledky rozvrhování a syntézy třech benchmarků s vnořenými smyčkami. Vždy je uveden Ganttův diagram a dále tabulka údajů ze syntézy obvodu: odhadovaná maximální dosažitelná hodnota hodinové frekvence, počet ekvivalentních hradel (odpovídá realizaci diskrétními součástkami) a počet skutečně využitých bloků LUT. Pro každý algoritmus jsou udány hodnoty dosažené syntézou pouze řídícího a pouze výkonného bloku a nakonec syntézou celého zapojení.

Veškeré dosažené výsledky se vztahují ke konfiguraci se 16-ti bitovými daty v pevné řádové čárce. Výsledek syntézy celého zapojení zahrnuje paměti, registry i testovací jednotky.

C.1 Benchmark nb1

První benchmark pracuje s pěti proměnnými, provádí tři skalární operace a volá jedno makro. Ganttův diagram rozvrhu je na obr. C.1. Parametry dosažené syntézou shrnuje tabulka C.1.



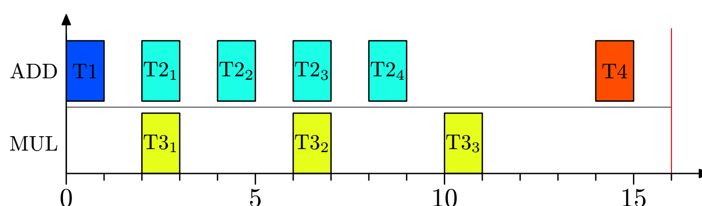
Obrázek C.1: Ganttův diagram benchmarku 1

Tabulka C.1: Výsledky syntézy NB1

blok	hodiny [MHz]	ekv.hradel	LUTs
řídící	199	308	29
výkonný	215	1559	96
celkem	165	1951	139

C.2 Benchmark nb2

V tomto případě se pracuje opět s pěti proměnnými, hlavní smyčka provádí dvě skalární operace a spouští dvě různá makra. Rozvrh je na obr. C.2, výsledky syntézy shrnuje tabulka C.2



Obrázek C.2: Ganttův diagram benchmarku 2

Tabulka C.2: Výsledky syntézy NB2

blok	hodiny [MHz]	ekv.hradel	LUTs
řídící	309	424	39
výkonný	160	1939	81
celkem	160	2427	118

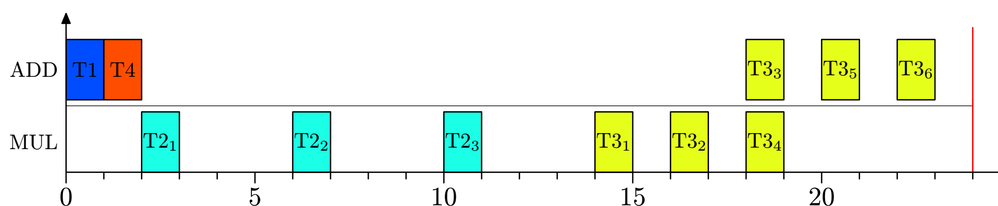
C.3 Benchmark nb3

Algoritmus obsahuje v hlavní smyčce dvě skalární operace a dále se vykonávají dvě makra. Časový diagram je zobrazen na obr. C.3. Na obr. C.5 pak vidíme propojení dvou bloků, řídícího a výkonného, jak je zobrazen po syntéze generovaného VHDL kódu v Xilinx XST. Vnitřní struktura řídícího bloku je zobrazena na obr. C.4. Můžeme zde vidět

hlavní automat a dva automaty podřazené. Z hlavního bloku (nahoře) vedou do zbývajících dvou bloků signály pro jejich spuštění. Rozvod hodin a reset jsou společné.

Výstupní signály z celého bloku jsou jednak jednobitové požadavky na naplnění jednotky daty (`runopXi`) a uložení výsledku z jednotky (`runopXo`), jednak adresy čtecí a zápisové adresy pamětí, s nimiž se při dané aritmetické operaci pracuje.

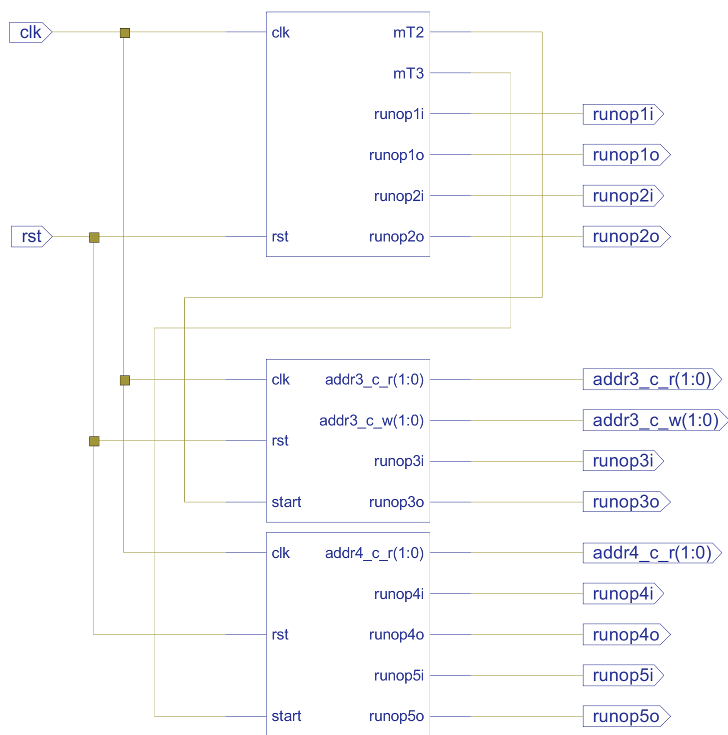
Parametry dosažené po syntéze VHDL jsou shrnuty v tabulka C.3



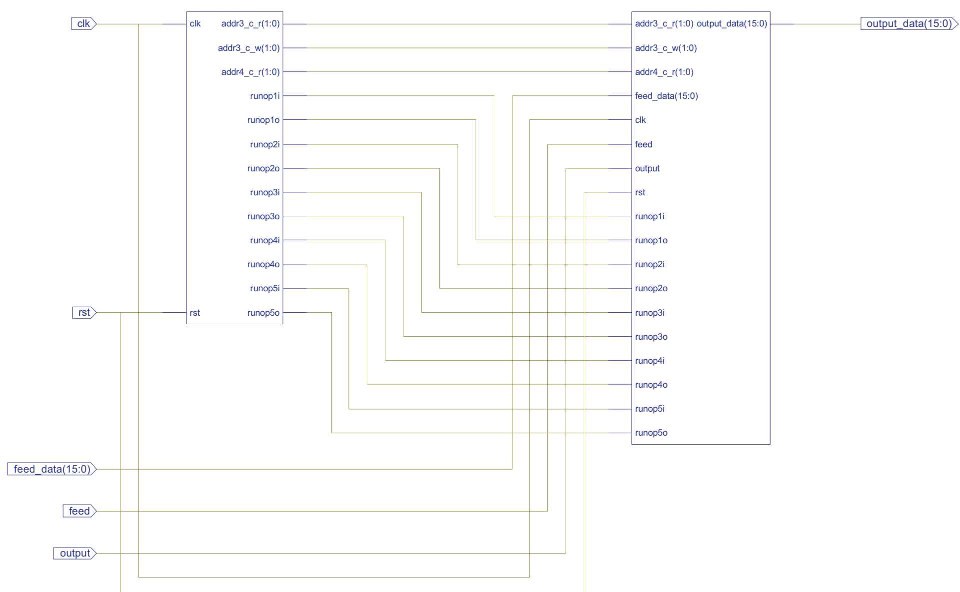
Obrázek C.3: Ganttův diagram benchmarku 3

Tabulka C.3: Výsledky syntézy NB3

blok	hodiny [MHz]	ekv.hradel	LUTs
řídící	272	615	61
výkonný	119	2673	103
celkem	99	3229	154



Obrázek C.4: Zapojení bloků v benchmarku nb3



Obrázek C.5: Řídící a výkonný blok - nb3