VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PARALELNÍ VÝPOČETNÍ ARCHITEKTURY ZALOŽENÉ NA NUMERICKÉ INTEGRACI

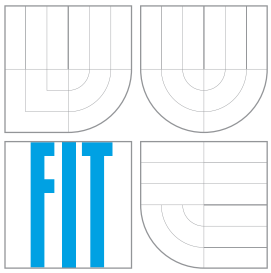
DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE
AUTHOR

Ing. MICHAL KRAUS

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PARALELNÍ VÝPOČETNÍ ARCHITEKTURY ZALOŽENÉ NA NUMERICKÉ INTEGRACI

PARALLEL COMPUTER SYSTEMS BASED ON NUMERICAL INTEGRATIONS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE — Ing. MICHAL KRAUS
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. JIŘÍ KUNOVSKÝ, CSc.
SUPERVISOR

BRNO 2013

Abstrakt

Předkládaná práce se zabývá simulací spojitých systémů popsaných soustavou diferenciálních rovnic nebo blokového diagramu. Zcela běžné je numerické řešení diferenciálních rovnic a používání simulačních programových celků (Matlab, Maple, TKSL). Pro řešení diferenciálních rovnic je použita metoda Taylorovy řady. Bylo dokázáno, že metoda dosahuje velké přesnosti a rychlosti a nabízí možnost paralelního provádění a tím další urychlení výpočtu. Hlavní část práce obsahuje popis návrhu a realizace specializovaného paralelního systému provádějící výpočet numerické integrace v několika variantách a jejich porovnání.

Abstract

This thesis deals with continuous system simulation. The systems can be described by system of differential equations or block diagram. Differential equations are usually solved by numerical methods that are integrated into simulation software such as Matlab, Maple or TKSL. Taylor series method has been used for numerical solutions of differential equations. The presented method has been proved to be both very accurate and fast and also procesed in parallel systems. The aim of the thesis is to design, implement and compare a few versions of the parallel system.

Klíčová slova

numerická integrace, Taylorova řada, paralelní systém, propojovací sítě, integrátor

Keywords

numerical integration, Taylor series, parallel system, interconnection networks, integrator

Citace

Michal Kraus: Paralelní výpočetní architektury založené na numerické integraci, disertační práce, Brno, FIT VUT v Brně, 2013

Paralelní výpočetní architektury založené na numerické integraci

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracoval samostatně pod vedením pana doc. Ing. Jiřího Kunovského, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Kraus
2. června 2013

Poděkování

Chtěl bych poděkovat svému školiteli doc. Ing. Jiřímu Kunovskému, CSc. za jeho podporu a vedení během mého studia, za cenné rady, připomínky a komentáře k mé práci. Můj dík patří také přítelkyni Petře, rodině a všem ostatním, kteří mě po dobu studia jakkoliv podporovali a bez nichž by tato práce nemohla vzniknout. Tato práce byla podporována MŠMT v rámci grantu MSM0021630528 - Výzkum informačních technologií z hlediska bezpečnosti a projektem FR2681/2010/G1 - Modernizace laboratorního přípravku do předmětu Prvky počítačů. Díky mezinárodnímu stipendijnímu programu Aktion jsem byl na roční stáži na TU Wien ve školním roce 2009/2010. Na základě tohoto pobytu vznikla dlouholetá spolupráce, která pokračuje dalšími výzkumnými projekty.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Přestože výpočetní kapacity sekvenčních počítačů neustále rostou, vždy se najdou náročné aplikace, pro které není dostupný výkon dostačující. Myšlenka znásobení stávajícího výkonu paralelizací je prostá a je stará v podstatě jako výpočetní technika sama. Myšlenku lze formulovat takto: Sdružme několik počítačů a rozdělme výpočetní zátěž mezi ně. Budeme moci zpracovat větší úlohy a výpočet proběhne rychleji. Jinými slovy: Paralelní zpracování slibuje zvýšení výkonu, kapacity, poměru výkonu a ceny, spolehlivosti apod.

Základní problémy vědy a inženýrství s širokými ekonomickými a vědeckými dopady na naší civilizaci vyžadují následující vysoce náročné výpočty (High Performance Computing - HPC):

- předpovídání počasí, modelování atmosféry, moří a zemětřesení
- modelování v astronomii, kosmologii, biochemii a genetice
- návrhy a testy složitých inženýrských systémů (letadla, auta, elektrárny)
- počítačové zkoušky jaderných zbraní a skladování jaderného odpadu,
- rozpoznání lidské řeči, modelování lidské inteligence, robotika
- dobývání znalostí a hledání na internetu
- návrh VLSI obvodů
- a mnoho dalších

Modelování a simulace reálných systémů hraje významnou roli ve všech odvětvích dnešní společnosti. Pro simulaci reálných systémů reálného světa se používají dva odlišné přístupy: spojitý a diskrétní. O tom, který přístup je vhodnější, rozhoduje úroveň abstrakce, která se při simulaci použije. Odpovídající výběr popisu modelu samozřejmě závisí jak na účelu modelu, tak na úrovni našich znalostí o modelovaném systému.

Tato disertační práce má kořeny v mé diplomové práci [Kra06], kterou jsem úspěšně obhájil v roce 2006. V této diplomové práci vznikl první návrh základního výukového výpočetního prvku celého specializovaného paralelního systému - integrátoru v provedení pevné řádové čárky a byla nastíněna i verze v pohyblivé řádové čárce.

1.1 Motivace, cíle disertační práce

Předkládaná disertační práce se zabývá simulací spojitých systémů. Spojité systémy se popisují soustavou diferenciálních a algebraických rovnic, velmi přirozenou formou je rovněž popis spojitých systémů pomocí blokového diagramu (zcela typická je tato situace v teorii řízení). Blokový diagram je chápán jako paralelně pracující systém. Při výpočtu se odpovídající blokový diagram převede na soustavu obyčejných diferenciálních rovnic, které se řeší v zásadě analyticky nebo numericky. Analytické řešení soustav diferenciálních rovnic je omezeno jen na určitou omezenou třídu úloh a je velmi komplikované. Zcela běžné je proto numerické řešení diferenciálních rovnic a používání simulačních programových celků (Matlab [Mat12], Maple [Map12], TKSL [Kun94]).

Nejčastěji se používají metody Runge-Kutta, Eulerova metoda, nebo vícekrokové metody typu prediktor-korektor. Zde prezentovaná metoda využívající Taylorovy řady se od běžných metod liší možností výpočtu vyšších derivací a jejich následným využitím při výpočtu. Bližší seznámení s metodou Taylorovy řady je v [Kun94]. Srovnání přesnosti a rychlosti výpočtu za pomocí této metody s ostatními používanými metodami bylo obsahem několik vědeckých článků publikovaných na konferencích a v časopisech, např [1] a [17]. Z rozboru prezentovaným také v této práci je patrné, že metoda nabízí možnost paralelního provádění výpočtu a tím další urychlení.

Hlavní cíle předložené práce shrnu do několika následujících bodů:

- Vymezení typu diferenciálních rovnic, které je možno řešit metodou Taylorovy řady a následně tedy i navrženým paralelním systémem
- Srovnání metody Taylorovy řady s numerickými metodami světových standardů (Matlab, Maple)
- Návrh specializovaného paralelního systému určeného pro numerické řešení diferenciálních rovnic
- Rozbor a srovnání variant integrátorů a celých paralelních systémů

1.2 Struktura práce

Struktura je volena tak, aby odrážela současný stav vědění v tématu disertační práce a obsahovala všechny důležité informace popisující realizovanou výzkumnou činnost a její výsledky.

Důležitou náplní této práce je numerická integrace. Základními rysy a vlastnostmi metod numerické integrace se zabývá kapitola 2. Jsou zde uvedeny základní pojmy numerické integrace i se stručným popisem jednokrokových a vícekrokových numerických integračních metod. Hlavně je však uvedena nově pojatá metoda Taylorovy řady (v současné době implementovaná v simulačním jazyce TKSL) a porovnána s používanými metodami Runge-Kutta. Metoda Taylorovy řady se vyznačuje nejen vysokou přesností výpočtu, ale především vysokým stupněm paralelismu. Výpočetní operace (numerické integrace) jsou při použití moderní metody Taylorovy řady na sobě ve speciálních případech nezávislé a mohou být realizovány v oddělených paralelně pracujících procesorech.

Z těchto důvodů je obsahem následujícího textu v kapitolách 3 a 4 bližší seznámení s paralelními architekturami a propojovacími sítěmi. Jsou zde uvedeny typické paralelní architektury a propojovací sítě, které se v dnešní době u paralelních systémů používají.

Preferovanou variantou paralelní architektury v této práci je SIMD (Single Instruction Multiple Data), preferovaným typem propojovací sítě je dynamická propojovací síť založená na křížových přepínačích.

Kapitola 5 je poslední kapitolou odrážející současný stav a představující prostředky, které jsou použity dále v této práci. Jsou zde uvedeny matematické operace a jejich možné realizace, které se využívají při provádění numerické integrace pomoci metody Taylorovy řady.

Následující kapitoly jsou už zaměřeny na samotný specializovaný výukový paralelní systém. Základní verze paralelní výpočetní architektury založené na numerické integraci obsahuje výpočetní aritmeticko-logické jednotky, propojovací systém a řídící jednotku. Možné varianty výpočetní architektury vycházejí ze tří verzí aritmeticko-logických jednotek (tří verzí numerických integrátorů). Kapitola 6 obsahuje popis návrhu celého výukového systému. Konkrétně návrh tří typů aritmeticko-logických jednotek, řídící jednotky a propojovacího systému. Kapitola 7 obsahuje popis realizace jednotlivých prvků specializovaného paralelního systému a porovnání jednotlivých variant. V kapitole 8 jsou uvedeny oblasti možného využití i s konkrétními příklady.

Kapitola 9 shrnuje dosažené výsledky disertační práce, hodnotí splnění cílů a představuje možnosti dalšího výzkumu v této oblasti.

Kapitola 2

Numerické řešení diferenciálních rovnic

Nalezení analytického řešení rozsáhlých soustav diferenciálních rovnic je složité, mnohdy nemožné. V současné době se s rozmachem výpočetní techniky čím dál častěji využívá numerického řešení. Zaměřme se tedy na problém numerického řešení obyčejných diferenciálních rovnic s počáteční podmínkou.

Počáteční problém pro ODR.
Obyčejná diferenciální rovnice (ODR) se nazývá rovnice *n-tého řádu* (ODRn), jestliže neznámá funkce je funkcí jedné proměnné a její nejvyšší derivace neznámé funkce je *n*-tého řádu. V práci se budeme zabývat především diferenciálními rovnicemi prvního řádu. Popis jednotlivých typů obyčejných diferenciálních rovnic a způsobů jejich řešení obsahuje [Šva99, DB91].

Obecný tvar diferenciální rovnice prvního řádu je

$$g(t, y(t), y'(t)) = 0. \tag{2.1}$$

Předpokládejme, že rovnici (2.1) lze také vyjádřit explicitně ve tvaru

$$y'(t) = f(t, y(t)). \tag{2.2}$$

Obecné řešení diferenciální rovnice obsahuje integrační konstantu, která může nabývat libovolné hodnoty. Proto k jednoznačnému určení $y(t)$ musíme ještě doplnit hodnotu funkce v určitém bodě $t = t_0$, tedy *počáteční podmínku*

$$y(t_0) = y_0. \tag{2.3}$$

Rovnice (2.2) spolu s počáteční podmínkou (2.3) se nazývá *počáteční úloha*, nebo také *Cauchyova úloha*.

Budeme předpokládat, že každá Cauchyova úloha, o jejímž numerickém řešení budeme uvažovat, má jediné řešení. Je dobře známo, že tato podmínka **existence a jednoznačnosti řešení** je splněna, když je funkce $f(t, y(t))$ spojitá, ohraničená a splňuje Lipschitzovu podmínku.

Předcházející část obsahující teoretický popis diferenciálních rovnic byla převzata z [Šát12].

Za numerické řešení obyčejné diferenciální rovnice s počáteční podmínkou (2.4)

$$y' = f(t, y), \qquad y(t_0) = y_0 \tag{2.4}$$

je považována sekvence hodnot (2.5)

$$[y(t_0) = y_0], \quad [y(t_1) = y_1], \quad \dots \quad [y(t_N) = y_N]. \tag{2.5}$$

Tato čísla aproximují hodnoty přesného analytického řešení $y(t_0), y(t_1), \dots, y(t_N)$ v bodech $t_0, t_1, \dots, t_N$. Obvykle volíme ekvidistantní síť, tj. $t_i - t_{i-1} = h$; $i = 0, 1, \dots, N$. Číslo h se nazývá integrační krok. Hodnoty funkce mezi zvolenými body lze určit buď interpolací z okolních vypočtených bodů, nebo opětovnou aplikací numerické metody s použitím menšího integračního kroku.

Výpočet probíhá iteračně. Numerické řešení y_i v bodě t_i se počítá z hodnoty předešlého numerického řešení y_{i-1}. To platí pro *jednokrokové metody*. Existují i *vícekrokové metody*, které k výpočtu aktuálního řešení y_i používají k předešlých výsledků $y_{i-1}, y_{i-2}, \dots, y_{i-k}$. Při využití vícekrokových metod je problematický postup v prvních krocích výpočtu, kdy ještě nemáme k dispozici dostatečný počet předcházejících hodnot. Pro zahájení výpočtu je tedy nutné použít některou z jednokrokových metod.

Diferenciální rovnice vyšších řádů se musí pro numerické řešení převést na soustavu obyčejných diferenciálních rovnic prvního řádu. K realizaci tohoto převodu může být použita:

- metoda snižování řádu derivace - jednodušší, ale vyžaduje, aby na pravé straně rovnice nebyly derivace vstupu
- metoda postupné integrace - zvládne i rovnice s derivacemi vstupu na pravé straně

Obě metody využívají úprav rovnice a zavádění pomocných proměnných. Jsou použitelné i na soustavy rovnic vyššího řádu - stačí postup opakovat pro každou zadanou rovnici.

2.1 Numerické metody pro řešení diferenciálních rovnic

Bylo již publikováno veliké množství numerických metod řešení diferenciálních rovnic navzájem se lišících především ve dvou hlavních kritériích - *přesnosti* a *rychlosti*. Tato kritéria stojí proti sobě. Vždy je jedno kritérium potlačeno na úkor druhého. Popis numerických metod je v [KDJ05]. V této práci uvádím jen popis několika základních metod nutných pro definování principu výpočtu.

2.1.1 Taylorova řada

Taylorova řada je základní jednokroková metoda a vyjadřuje se zápisem:

$$y_{i+1} = y_i + h \cdot f^{[1]}(t_i, y_i) + \frac{h^2}{2!} \cdot f^{[2]}(t_i, y_i) + \dots + \frac{h^p}{p!} \cdot f^{[p]}(t_i, y_i) \tag{2.6}$$

h je integrační krok. Tato metoda umožňuje určit přesnost výpočtu. Výpočet konkrétní hodnoty y_{i+1} je iteračně prováděn až po dosažení zadané přesnosti výpočtu, kdy rozdíl dvou po sobě následujících hodnot členů Taylorovy řady je menší než požadovaná přesnost.

Hlavní problém spojený s použitím Taylorovy řady spočívá v nutnosti použití vyšších derivací. Aplikace Taylorovy řady na numerické řešení obyčejných diferenciálních rovnic jsou popsány např. v [Gib60, BWZ72].

2.1.2 Eulerova metoda

Nejjednodušší jednokroková Eulerova metoda využívá pouze první dva členy Taylorovy řady:

$$y_{i+1} = y_i + h \cdot f^{[1]}(t_i, y_i) \tag{2.7}$$

Z hlediska geometrie vyjadřuje pravá strana rovnici přímky, kde $f^{[1]}(t_i, y_i)$ je směrnice tečny v bodě t_i.

Metoda se používá pro dostatečně malé h.

2.1.3 Metody Runge-Kutta

Vyšší přesnost a rychlost výpočtu ve srovnání s Eulerovou metodou lze získat metodami Runge-Kutta. Výpočet provádíme i mezi uzly y_i a y_{i+1}. Výsledný přírůstek najdeme jako váhový průměr vypočtených hodnot a jejich počet nám udává řád metody.

Obecný tvar jednokrokových metod Runge-Kutta je

$$y_{i+1} = y_i + h \cdot (w_1 k_1 + \cdots + w_s k_s) \tag{2.8}$$

- $k_1 = f(t_i, y_i)$
- $k_i = f(t_i + \alpha_l h, y_i + h \sum_{j=1}^{l-1} \beta_{lj} k_j), \quad i = 2, \ldots, s$
- konstanty w_n, α_n a β_{lj} jsou konstanty volené podle řádu metody tak, aby získané řešení souhlasilo s Taylorovou řadou v bodě t_{i+1}.

Metoda Runge-Kutta 2. řádu je používána ve dvou variantách (označované jako lichoběžníková (2.9) a zlepšený Euler (2.10)). Jejich obecný tvar je:

$$\begin{aligned} y_{i+1} &= y_i + h \cdot (\frac{1}{2} k_1 + \frac{1}{2} k_2) \\ k_1 &= f(t_i, y_i) \\ k_2 &= f(t_{i+1} + h, y_i + h \cdot k_1) \end{aligned} \tag{2.9}$$

$$\begin{aligned} y_{i+1} &= y_i + k_2 \\ k_1 &= f(t_i, y_i) \\ k_2 &= f(t_{i+1} + \frac{1}{2} h, y_i + \frac{1}{2} h \cdot k_1) \end{aligned} \tag{2.10}$$

Nejvíce rozšířenou a používanou metodou je Runge-Kutta 4. řádu. Její obecný tvar je (2.11)

$$\begin{aligned} y_{i+1} &= y_i + \frac{1}{6} h \cdot (k_1 + 2 \cdot k_2 + 2 \cdot k_3 + k_4) \\ k_1 &= f(t_i, y_i) \\ k_2 &= f(t_{i+1} + \frac{1}{2} h, y_i + \frac{1}{2} h \cdot k_1) \\ k_3 &= f(t_{i+1} + \frac{1}{2} h, y_i + \frac{1}{2} h \cdot k_2) \\ k_4 &= f(t_{i+1} + h, y_i + h \cdot k_3) \end{aligned} \tag{2.11}$$

Více informací o metodách Runge-Kutta je v [But08].

2.1.4 Metody Adams-Bashforth

Jedná se o první významnou numerickou metodu pro řešení obyčejných diferenciálních rovnic. Byla vyvinuta již v 19. století [But00] a dodnes tvoří významnou část ve většině moderních softwarových nástrojů. Jedná se o vícekrokovou metodu. Např. metoda Adams-Bashforth počítá následující hodnotu ze čtyř předchozích hodnot podle vzorce

$$y_{i+1} = y_i + \frac{h}{24} \cdot (55y_i - 59y_{i-1} + 37y_{i-2} - 9y_{i-3}) \tag{2.12}$$

Problém nastává při spuštění výpočtu. Metoda není samostartující a je nutné použít jednokrokovou metodu.

2.2 Metoda Taylorovy řady

Analytické počítání členů Taylorovy řady (především odvozování vyšších derivací) bylo považováno v mnoha případech za příliš komplikované. V mnoha moderních knihách o numerické a aplikované matematice je uvedeno, že Taylorova řada je z teoretického a koncepčního hlediska velice zajímavá, avšak může být použita jen v určitých speciálních případech. Obecné použití integračního algoritmu je příliš složité. Předkládaná práce se zabývá běžnými "technickými problémy". Pro jejich řešení lze Taylorovu řadu úspěšně využít [Mik00].

Důležitou součástí výpočtu je automatické nastavení řádu metody. To znamená, že se použije tolik členů Taylorovy řady, kolik je potřeba k dosažení požadované přesnost výpočtu. Metoda Taylorovy řady má velmi příznivé paralelní vlastnosti. Většina výpočetních operací je vzájemně nezávislých, takže mohou být prováděny paralelně v oddělených procesorech paralelního výpočetního systému. Nezbytnou součástí metody je automatická transformace zadání. Původně zadaná soustava nelineárních diferenciálních rovnic se automaticky transformuje na polynomiální tvar, t.j. na tvar, u kterého lze snadno rekurentně vypočítat jednotlivé členy Taylorovy řady. Tato transformace je představena v podkapitole 2.2.3.

I když nebyla metoda Taylorovy řady v literatuře velmi preferovaná kvůli nutnosti použití vyšších derivací, experimentální výpočty ukázaly a teoretické analýzy dokázaly, že přesnost a stabilita metody Taylorovy řady předčí běžně používané explicitní metody pro numerické řešení diferenciálních rovnic. Tato srovnání jsou také obsahem několika vědeckých článků.

V článku [10] je ukázáno řešení homogenní diferenciální rovnice s konstantními koeficienty. Na tomto řešení je předvedena extrémní přesnost a rychlost výpočtu pomocí Taylorovy řady. Jsou zde porovnány různé šířky výpočetní aritmetiky a zobrazeny závislosti na změně použitého integračního kroku a počtu členů Taylorovy řady.

Článek [17] představuje využití metody Taylorovy řady k výpočtu diferenciálních rovnic. Zaměřuje se na speciální případy, které vyžadují vysokou přesnost výpočtu. Porovnává numerické řešení diferenciálních rovnic pomocí TKSL a Matlabu.

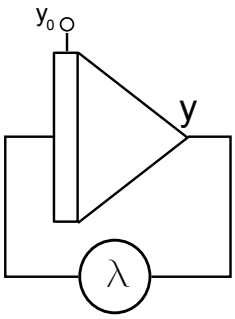
Článek [1] zahrnuje také řešení problémů, které mohou být transformovány na řešení systému diferenciálních rovnic. V článku je Taylorova řada aplikována na konkrétní reálný model torzní hřídele automobilu a diskutována stabilita a konvergence. Dále je Taylorova řada srovnána s metodami, které se využívají v programu Matlab/Simulink. Článek vznikl ve spolupráci s Technickou univerzitou ve Vídni během mého stipendijního pobytu.

2.2.1 Použití Taylorovy řady

Předkládaná nová specializovaná metoda Taylorovy řady [Kun94] pro numerické řešení obyčejných diferenciálních rovnic byla vytvořena na základě analýzy nejjednodušší homogenní lineární diferenciální rovnice 1. řádu s konstantními koeficienty - tzv. *Dahlquistův problém* [HW96].

$$y' = \lambda \cdot y, \qquad y(0) = y_0 \tag{2.13}$$

Odpovídající blokové řešení s využitím známého symbolu integrátoru a násobící konstanty je znázorněno na obrázku 2.1:



Obrázek 2.1: Blokové znázornění - Dahlquistův problém

V následujícím textu se rovnice (2.6)–(2.11) využívající zápis $y' = f(t, y)$ modifikují na $y' = f(y), y(0) = y_0$. Zápis rovnice (2.13) umožňuje jednoduše definovat pro výpočet prvního kroku numerického řešení následující algoritmy výpočtu (pro $\lambda = 1$):

- při aplikaci Eulerovy metody (rovnice 2.7):

$$\begin{aligned} y_1 &= y_0 + DY1_0 \\ DY1_0 &= h \cdot y_0 \end{aligned} \tag{2.14}$$

- při aplikaci metody Runge-Kutta 2. řádu (rovnice 2.10):

$$\begin{aligned} y_1 &= y_0 + k_2 \\ k_1 &= h \cdot y_0 \\ k_2 &= h \cdot (y_0 + \frac{k_1}{2}) \end{aligned} \tag{2.15}$$

- při aplikaci standardní metody Runge-Kutta 4. řádu (rovnice 2.11):

$$\begin{aligned} y_1 &= y_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 \\ k_1 &= h \cdot y_0 \\ k_2 &= h \cdot (y_0 + \frac{k_1}{2}) \\ k_3 &= h \cdot (y_0 + \frac{k_2}{2}) \\ k_4 &= h \cdot (y_0 + \frac{k_3}{2}) \end{aligned} \tag{2.16}$$

Správnost definovaných algoritmů (rovnice 2.14–2.16) (při numerickém řešení rovnice 2.13) lze potvrdit úpravou vztahů pro výpočet hodnoty y_1:

- při aplikaci Eulerovy metody (po dosazení do rovnice 2.14) je

$$y_1 = y_0 \cdot (1 + h) \tag{2.17}$$

- při aplikaci metody Runge-Kutta 2. řádu (po dosazení do rovnice 2.15) je

$$y_1 = y_0 \cdot (1 + h + \frac{h^2}{2}) \tag{2.18}$$

- při aplikaci metody Runge-Kutta 4. řádu (po dosazení do rovnice do 2.16) je

$$y_1 = y_0 \cdot (1 + h + \frac{h^2}{2} + \frac{h^3}{3!} + \frac{h^4}{4!}) \tag{2.19}$$

Ze zápisu rovnic (2.17)–(2.19) je zřejmé, že se jedná o zcela charakteristický rozvoj funkce do exponenciální řady ($e^t = 1 + t/1! + t^2/2! + t^3/3! + \dots$). Tento výsledek bylo možno očekávat, protože známým analytickým řešením rovnice (2.13) je

$$y = e^t \tag{2.20}$$

Stejný výsledek zápisu první hodnoty numerického výpočtu y_1 rovnice (2.13) lze však získat i rozvojem funkce y (rovnice 2.20) do Taylorovy řady (rovnice 2.1.1). Podmínka platnosti Taylorovy řady (funkce $f(t)$ a její derivace musí být jednoznačné, konečné a spojité v intervalu mezi t a $t + h$) je splněna, protože platí

$$y' = y \qquad \text{resp.} \qquad y' = y'' = y''' = \cdots \tag{2.21}$$

- při aplikaci metody Taylorovy řady (po dosazení do rovnice 2.6) je

$$y_1 = y_0 + h \cdot y_0 + \frac{h^2}{2!} y_0 + \frac{h^3}{3!} y_0 + \cdots \tag{2.22}$$

tzn. skutečně je

$$y_1 = y_0 \cdot (1 + h + \frac{h^2}{2} + \frac{h^3}{3!} + \cdots)$$

Provede-li se v rovnici (2.22) označení

$$y_1 = y_0 + DY1_0 + DY2_0 + DY3_0 + \cdots \tag{2.23}$$

potom platí

$$DY1_0 = h \cdot y_0 \tag{2.24}$$

$$DY2_0 = \frac{h}{2} DY1_0 \tag{2.25}$$

$$DY3_0 = \frac{h}{3} DY2_0 \tag{2.26}$$

$$\vdots$$

$$DYp_0 = \frac{h}{p} DY(p-1)_0 \tag{2.27}$$

Lze tedy rovnice 2.23–2.27 považovat za algoritmus numerického výpočtu hodnoty y rovnice (2.13) metodou Taylorovy řady.

Prakticky se výpočet provede tak, že se ze známé počáteční podmínky y_0 stanoví (dle rovnice 2.24) výraz $DY1_0$ (tj. členy Taylorovy řady s první mocninou kroku h). Z této vypočítané hodnoty $DY1_0$ se stanoví (dle rovnice 2.25) výraz $DY2_0$ (tj. člen Taylorovy řady s druhou mocninou kroku h). Tímto postupem se pokračuje až do p-tého řádu.

Nová hodnota řešení y_1 (tj. hodnota prvního kroku výpočtu) se stanoví jako součet počáteční podmínky y_0 a dílčích přírůstků (členů Taylorovy řady) - rovnice (2.23).

Postup výpočtu metodou Taylorovy řady je velmi jednoduše algoritmizovatelný - z hlediska uživatele stačí definovat počáteční podmínku, zvolený integrační krok h a řád metody - nabízí se rovněž varianta výpočtu na "plnou přesnost", tzn. bude se provádět výpočet takovým řádem metody (s tolika členy Taylorovy řady), pokud se hodnota DYp_0 uplatní v součtu (rovnice 2.23).

2.2.2 Srovnání efektivnosti Taylorovy řady a metody Runge-Kutta

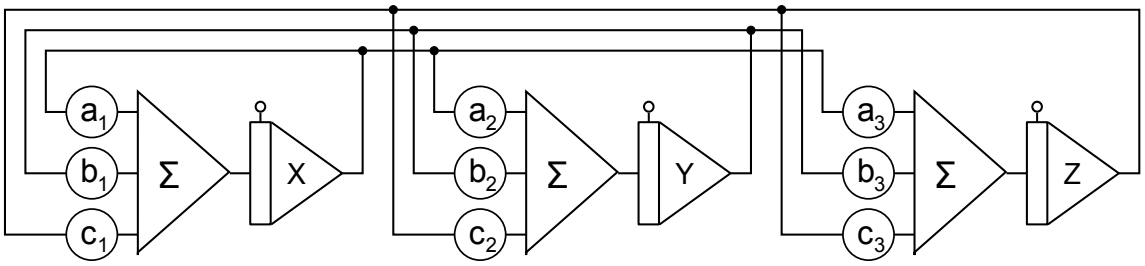
Postup numerického výpočtu rovnice (2.13) lze aplikovat na soustavu homogenních lineárních diferenciálních rovnic s konstantními koeficienty (např. pro soustavu rovnic (2.28)–(2.30))

$$x' = a_1 \cdot x + b_1 \cdot y + c_1 \cdot z \qquad x(0) = x_0 \tag{2.28}$$

$$y' = a_2 \cdot x + b_2 \cdot y + c_2 \cdot z \qquad y(0) = y_0 \tag{2.29}$$

$$z' = a_3 \cdot x + b_3 \cdot y + c_3 \cdot z \qquad z(0) = z_0 \tag{2.30}$$

Ze zápisu numerických řešení této soustavy (přesněji, z výpočtu jednoho kroku numerického řešení) homogenních lineárních diferenciálních rovnic s konstantními koeficienty vyplývají algoritmy paralelní spolupráce nezávislých procesorů (integrátorů). Odpovídající blokové schéma je na obrázku 2.2.



Obrázek 2.2: Blokové schéma pro řešení soustavy diferenciálních rovnic (2.28)–(2.30)

Cílem následující části je provést srovnání efektivnosti Taylorovy řady a metody Runge-Kutta při řešení soustavy homogenních diferenciálních rovnic 1. řádu (2.28)–(2.30). Nejprve se použijí verze druhého řádu obou srovnávaných metod a následně čtvrtého řádu.

Řešení soustavy (2.28)–(2.30) metodou Runge-Kutta 2. řádu

Obecný tvar pro metodu Runge-Kutta 2. řádu (2.9) lze pro první krok výpočtu soustavy (2.28)–(2.30) přepsat do tvaru:

$$x_1 = x_0 + \frac{1}{2}(k_{1,x} + k_{2,x}) \tag{2.31}$$

$$y_1 = y_0 + \frac{1}{2}(k_{1,y} + k_{2,y}) \tag{2.32}$$

$$z_1 = z_0 + \frac{1}{2}(k_{1,z} + k_{2,z}) \tag{2.33}$$

kde pro koeficienty k_1 platí:

$$k_{1,x} = h(a_1 \cdot x_0 + b_1 \cdot y_0 + c_1 \cdot z_0) \tag{2.34}$$

$$k_{1,y} = h(a_2 \cdot x_0 + b_2 \cdot y_0 + c_2 \cdot z_0) \tag{2.35}$$

$$k_{1,z} = h(a_3 \cdot x_0 + b_3 \cdot y_0 + c_3 \cdot z_0) \tag{2.36}$$

a pro koeficienty k_2:

$$k_{2,x} = h(a_1(x_0 + k_{1,x}) + b_1(y_0 + k_{1,y}) + c_1(z_0 + k_{1,z})) \tag{2.37}$$

$$k_{2,y} = h(a_2(x_0 + k_{1,x}) + b_2(y_0 + k_{1,y}) + c_2(z_0 + k_{1,z})) \tag{2.38}$$

$$k_{2,z} = h(a_3(x_0 + k_{1,x}) + b_3(y_0 + k_{1,y}) + c_3(z_0 + k_{1,z})) \tag{2.39}$$

Ze zápisu rovnic (2.31)–(2.39) je zřejmé, že požadované matematické operace v navzájem si odpovídajících rovnicích (tj. (2.31)–(2.33), (2.34)–(2.36) a (2.37)–(2.39)) jsou stejné. Mohou být tedy provedeny paralelně ve třech oddělených mikroprocesorech. Předpokládá se, že procesory provádí základní elementární operace (sčítání a násobení).

Základní posloupnost všech operací při řešení soustavy pro všechny paralelní procesory je názorně uvedena v tabulce 2.1.

Postupnými elementárními operacemi se získá k_1: konkrétně v procesoru X je $k_{1,x} = X_6$, v procesoru Y je $k_{1,y} = Y_6$ a v procesoru Z je $k_{1,z} = Z_6$. Dále následuje obdobným způsobem souběžně ve všech třech procesorech postupný výpočet k_2: v procesoru X je $k_{2,x} = X_{15}$, v procesoru Y je $k_{2,y} = Y_{15}$ a v procesoru Z je $k_{2,z} = Z_{15}$. Na závěr se z vypočtených hodnot k_1, k_2 a počátečních podmínek dopočítá výsledek $x_1 = X_{18}$, $y_1 = Y_{18}$ a $z_1 = Z_{18}$.

Řešení soustavy (2.28)–(2.30) metodou Taylorovy řady 2. řádu

Pokud použijeme označení analogicky s rovnicí (2.23), obdržíme soustavu pro numerický výpočet nových hodnot x_1, y_1 a z_1 ve tvaru (použijí se pouze 2 členy Taylorovy řady):

$$x_1 = x_0 + DX1_0 + DX2_0 \tag{2.40}$$

$$y_1 = y_0 + DY1_0 + DY2_0 \tag{2.41}$$

$$z_1 = z_0 + DZ1_0 + DZ2_0 \tag{2.42}$$

kde význam jednotlivých členů je:

$$DX1_0 = h \cdot (a_1 \cdot x_0 + b_1 \cdot y_0 + c_1 \cdot z_0) \tag{2.43}$$

$$DY1_0 = h \cdot (a_2 \cdot x_0 + b_2 \cdot y_0 + c_2 \cdot z_0) \tag{2.44}$$

$$DZ1_0 = h \cdot (a_3 \cdot x_0 + b_3 \cdot y_0 + c_3 \cdot z_0) \tag{2.45}$$

Tabulka 2.1: Posloupnost operací při řešení soustavy metodou Runge-Kutta 2. řádu

Pořadí	**Procesor X**	**Procesor Y**	**Procesor Z**
1	$X1 = a_1 \cdot x_0$	$Y1 = a_2 \cdot x_0$	$Z1 = a_3 \cdot x_0$
2	$X2 = b_1 \cdot y_0$	$Y2 = b_2 \cdot y_0$	$Z2 = b_3 \cdot y_0$
3	$X3 = c_1 \cdot z_0$	$Y3 = c_2 \cdot z_0$	$Z3 = c_3 \cdot z_0$
4	$X4 = X1 + X2$	$Y4 = Y1 + Y2$	$Z4 = Z1 + Z2$
5	$X5 = X4 + X3$	$Y5 = Y4 + Y3$	$Z5 = Z4 + Z3$
6	$X6 = h \cdot X5$	$Y6 = h \cdot Y5$	$Z6 = h \cdot Z5$
7	$X7 = x_0 + X6$	$Y7 = x_0 + X6$	$Z7 = x_0 + X6$
8	$X8 = y_0 + Y6$	$Y8 = y_0 + Y6$	$Z8 = y_0 + Y6$
9	$X9 = z_0 + Z6$	$Y9 = z_0 + Z6$	$Z9 = z_0 + Z6$
10	$X10 = a_1 \cdot X7$	$Y10 = a_2 \cdot Y7$	$Z10 = a_3 \cdot Z7$
11	$X11 = b_1 \cdot X8$	$Y11 = b_2 \cdot Y8$	$Z11 = b_3 \cdot Z8$
12	$X12 = c_1 \cdot X9$	$Y12 = c_2 \cdot Y9$	$Z12 = c_3 \cdot Z9$
13	$X13 = X10 + X11$	$Y13 = Y10 + Y11$	$Z13 = Z10 + Z11$
14	$X14 = X13 + X12$	$Y14 = Y13 + Y12$	$Z14 = Z13 + Z12$
15	$X15 = h \cdot X14$	$Y15 = h \cdot Y14$	$Z15 = h \cdot Z14$
16	$X16 = X6 + X15$	$Y16 = Y6 + Y15$	$Z16 = Z6 + Z15$
17	$X17 = 1/2 \cdot X16$	$Y17 = 1/2 \cdot Y16$	$Z17 = 1/2 \cdot Z16$
18	$X18 = x_0 + X17$	$Y18 = y_0 + Y17$	$Z18 = z_0 + Z17$

$$DX2_0 = \frac{h}{2} \cdot (a_1 \cdot DX1_0 + b_1 \cdot DY1_0 + c_1 \cdot DZ1_0) \tag{2.46}$$

$$DY2_0 = \frac{h}{2} \cdot (a_2 \cdot DX1_0 + b_2 \cdot DY1_0 + c_2 \cdot DZ1_0) \tag{2.47}$$

$$DZ2_0 = \frac{h}{2} \cdot (a_3 \cdot DX1_0 + b_3 \cdot DY1_0 + c_3 \cdot DZ1_0) \tag{2.48}$$

Ze zápisu rovnic (2.40)–(2.48) je zřejmé, že výpočet lze provést opět paralelně. Základní posloupnost operací při řešení soustavy pro všechny paralelní procesory je uvedena v tabulce 2.2.

Ze srovnání počtu operací v tabulkách 2.1 a 2.2 vyplývá vyšší efektivnost Taylorovy řady. K dosažení kompletního výsledku x_1, y_1 a z_1 bylo nutné provést 18 výpočetních operací při použití metody Runge-Kutta, kdežto metoda Taylorovy řady potřebovala 14 výpočetních operací.

Pro potvrzení předešlých zajímavých výsledků je dále uvedeno srovnání pro metodu Runge-Kutta 4. řádu a metodu Taylorovy řady 4. řádu.

Tabulka 2.2: Posloupnost operací při řešení soustavy metodou Taylorovy řady 2. řádu

Pořadí Procesor X	**Procesor Y**	**Procesor Z**	
1	$X1 = a_1 \cdot x_0$	$Y1 = a_2 \cdot x_0$	$Z1 = a_3 \cdot x_0$
2	$X2 = b_1 \cdot y_0$	$Y2 = b_2 \cdot y_0$	$Z2 = b_3 \cdot y_0$
3	$X3 = c_1 \cdot z_0$	$Y3 = c_2 \cdot z_0$	$Z3 = c_3 \cdot z_0$
4	$X4 = X1 + X2$	$Y4 = Y1 + Y2$	$Z4 = Z1 + Z2$
5	$X5 = X4 + X3$	$Y5 = Y4 + Y3$	$Z5 = Z4 + Z3$
6	$X6 = h \cdot X5$	$Y6 = h \cdot Y5$	$Z6 = h \cdot Z5$
7	$X7 = a_1 \cdot X6$	$Y7 = a_2 \cdot X6$	$Z7 = a_3 \cdot X6$
8	$X8 = b_1 \cdot Z6$	$Y8 = b_2 \cdot Y6$	$Z8 = b_3 \cdot Z6$
9	$X9 = c_1 \cdot Y6$	$Y9 = c_2 \cdot Y6$	$Z9 = c_3 \cdot Z6$
10	$X10 = X7 + X8$	$Y10 = Y7 + Y8$	$Z10 = Z7 + Z8$
11	$X11 = X10 + X9$	$Y11 = Y10 + Y9$	$Z11 = Z10 + Z9$
12	$X12 = h/2 \cdot X11$	$Y12 = h/2 \cdot Y11$	$Z12 = h/2 \cdot Z11$
13	$X13 = X12 + X6$	$Y13 = Y12 + Y6$	$Z13 = Z12 + Z6$
14	$X14 = x_0 + X13$	$Y14 = y_0 + Y13$	$Z14 = z_0 + Z13$

Řešení soustavy (2.28)–(2.30) metodou Runge-Kutta 4. řádu

Metoda Runge-Kutta 4.řádu je velmi rozšířenou a používanou metodou. Její obecný tvar (2.11) lze pro první krok výpočtu soustavy (2.28)–(2.30) přepsat do tvaru:

$$x_1 = x_0 + \frac{1}{6}(k_{1,x} + 2 \cdot k_{2,x} + 2 \cdot k_{3,x} + k_{4,x}) \tag{2.49}$$

$$y_1 = y_0 + \frac{1}{6}(k_{1,y} + 2 \cdot k_{2,y} + 2 \cdot k_{3,y} + k_{4,y}) \tag{2.50}$$

$$z_1 = z_0 + \frac{1}{6}(k_{1,z} + 2 \cdot k_{2,z} + 2 \cdot k_{3,z} + k_{4,z}) \tag{2.51}$$

kde pro koeficienty k_1 platí:

$$k_{1,x} = h(a_1 \cdot x_0 + b_1 \cdot y_0 + c_1 \cdot z_0) \tag{2.52}$$

$$k_{1,y} = h(a_2 \cdot x_0 + b_2 \cdot y_0 + c_2 \cdot z_0) \tag{2.53}$$

$$k_{1,z} = h(a_3 \cdot x_0 + b_3 \cdot y_0 + c_3 \cdot z_0) \tag{2.54}$$

pro koeficienty k_2:

$$k_{2,x} = h(a_1(x_0 + \frac{1}{2}k_{1,x}) + b_1(y_0 + \frac{1}{2}k_{1,y}) + c_1(z_0 + \frac{1}{2}k_{1,z})) \tag{2.55}$$

$$k_{2,y} = h(a_2(x_0 + \frac{1}{2}k_{1,x}) + b_2(y_0 + \frac{1}{2}k_{1,y}) + c_2(z_0 + \frac{1}{2}k_{1,z})) \tag{2.56}$$

$$k_{2,z} = h(a_3(x_0 + \frac{1}{2}k_{1,x}) + b_3(y_0 + \frac{1}{2}k_{1,y}) + c_3(z_0 + \frac{1}{2}k_{1,z})) \tag{2.57}$$

pro koeficienty k_3:

$$k_{3,x} = h(a_1(x_0 + \frac{1}{2}k_{2,x}) + b_1(y_0 + \frac{1}{2}k_{2,y}) + c_1(z_0 + \frac{1}{2}k_{2,z})) \quad (2.58)$$

$$k_{3,y} = h(a_2(x_0 + \frac{1}{2}k_{2,x}) + b_2(y_0 + \frac{1}{2}k_{2,y}) + c_2(z_0 + \frac{1}{2}k_{2,z})) \quad (2.59)$$

$$k_{3,z} = h(a_3(x_0 + \frac{1}{2}k_{2,x}) + b_3(y_0 + \frac{1}{2}k_{2,y}) + c_3(z_0 + \frac{1}{2}k_{2,z})) \quad (2.60)$$

a pro koeficienty k_4:

$$k_{4,x} = h(a_1(x_0 + k_{3,x}) + b_1(y_0 + k_{3,y}) + c_1(z_0 + k_{3,z})) \quad (2.61)$$

$$k_{4,y} = h(a_2(x_0 + k_{3,x}) + b_2(y_0 + k_{3,y}) + c_2(z_0 + k_{3,z})) \quad (2.62)$$

$$k_{4,z} = h(a_3(x_0 + k_{3,x}) + b_3(y_0 + k_{3,y}) + c_3(z_0 + k_{3,z})) \quad (2.63)$$

Řešení soustavy (2.28)–(2.30) metodou Taylorovy řady 4. řádu

Soustava pro numerický výpočet hodnot x_1, y_1 a z_1 má tvar:

$$x_1 = x_0 + DX1_0 + DX2_0 + DX3_0 + DX4_0 \quad (2.64)$$

$$y_1 = y_0 + DY1_0 + DY2_0 + DY3_0 + DY4_0 \quad (2.65)$$

$$z_1 = z_0 + DZ1_0 + DZ2_0 + DZ3_0 + DZ4_0 \quad (2.66)$$

význam jednotlivých členů je:

$$DX1_0 = h{\cdot}(a_1{\cdot}x_0 + b_1{\cdot}y_0 + c_1{\cdot}z_0) \quad (2.67)$$

$$DY1_0 = h{\cdot}(a_2{\cdot}x_0 + b_2{\cdot}y_0 + c_2{\cdot}z_0) \quad (2.68)$$

$$DZ1_0 = h{\cdot}(a_3{\cdot}x_0 + b_3{\cdot}y_0 + c_3{\cdot}z_0) \quad (2.69)$$

$$DX2_0 = \frac{h}{2}{\cdot}(a_1{\cdot}DX1_0 + b_1{\cdot}DY1_0 + c_1{\cdot}DZ1_0) \quad (2.70)$$

$$DY2_0 = \frac{h}{2}{\cdot}(a_2{\cdot}DX1_0 + b_2{\cdot}DY1_0 + c_2{\cdot}DZ1_0) \quad (2.71)$$

$$DZ2_0 = \frac{h}{2}{\cdot}(a_3{\cdot}DX1_0 + b_3{\cdot}DY1_0 + c_3{\cdot}DZ1_0) \quad (2.72)$$

$$DX3_0 = \frac{h}{3}{\cdot}(a_1{\cdot}DX2_0 + b_1{\cdot}DY2_0 + c_1{\cdot}DZ2_0) \quad (2.73)$$

$$DY3_0 = \frac{h}{3}{\cdot}(a_2{\cdot}DX2_0 + b_2{\cdot}DY2_0 + c_2{\cdot}DZ2_0) \quad (2.74)$$

$$DZ3_0 = \frac{h}{3}{\cdot}(a_3{\cdot}DX2_0 + b_3{\cdot}DY2_0 + c_3{\cdot}DZ2_0) \quad (2.75)$$

$$DX4_0 = \frac{h}{4}{\cdot}(a_1{\cdot}DX3_0 + b_1{\cdot}DY3_0 + c_1{\cdot}DZ3_0) \quad (2.76)$$

$$DY4_0 = \frac{h}{4}{\cdot}(a_2{\cdot}DX3_0 + b_2{\cdot}DY3_0 + c_2{\cdot}DZ3_0) \quad (2.77)$$

$$DZ4_0 = \frac{h}{4}{\cdot}(a_3{\cdot}DX3_0 + b_3{\cdot}DY3_0 + c_3{\cdot}DZ3_0) \quad (2.78)$$

Detailně popsanou posloupnost výpočetních operací použitých při řešení pomocí obou metod zde nebudu uvádět kvůli její větší délce. Postup výpočtu je analogický s metodami 2. řádu - rovnice (2.31)–(2.48). Při použití metody Runge-Kutta 4.řádu každý mikroprocesor provádí 46 operací, u metody Taylorovy řady 4. řádu je to 28 operací.

Z provedených rozborů je zřejmé, že při použití metody Runge-Kutta a Taylorovy řady stejných řádů, je počet výpočetních operací při výpočtu Taylorovou řadou menší. S použitím vyššího řádu metody je tento rozdíl ještě výraznější.

To například znamená, že při stejném počtu použitých operací jsme schopni metodou Taylorovy řady dosáhnout vyšší přesnosti (díky dosažení vyššího řádu metody) než metodou Runge-Kutta.

2.2.3 Metodika tvořících diferenciálních rovnic

Získané výsledky numerického řešení soustav homogenních lineárních diferenciálních rovnic s konstantními koeficienty byly použity jako podklady pro numerické řešení další skupiny obyčejných diferenciálních rovnic - pro soustavy nehomogenních lineárních diferenciálních rovnic s konstantními koeficienty.

Při řešení nehomogenní diferenciální rovnice se požadovaná funkce $f(t)$ ("pravá strana" diferenciální rovnice) "generuje" pomocí *tvořících diferenciálních rovnic* - v závislosti na typu funkce $f(t)$ se původní soustava diferenciálních rovnic rozšíří o další tvořící soustavu diferenciálních rovnic 1. řádu. Bližší seznámení s touto transformací je obsahem článku [13].

Např. rovnice (2.79)

$$y' = y + \sin(t) \qquad y(0) = y_0 \tag{2.79}$$

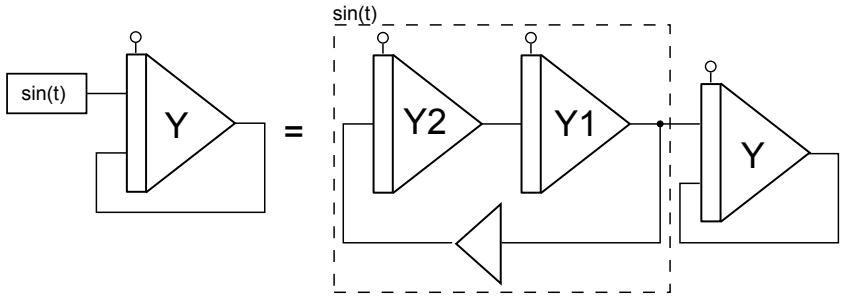
se převede na systém (2.80)–(2.82) obdobný soustavě rovnic (2.28)–(2.30)

$$y' = y + y_1 \qquad y(0) = y_0 \tag{2.80}$$

$$y_1' = y_2 \qquad y_1(0) = 0 \tag{2.81}$$

$$y_2' = -y_1 \qquad y_2(0) = 1 \tag{2.82}$$

Blokově se překreslí na schéma obsahující 3 integrátory (viz obrázek 2.3), které odpovídá nové soustavě diferenciálních rovnic (2.80)–(2.82).



Obrázek 2.3: Blokové schéma rovnice (2.79) před a po transformaci

Princip tvořících diferenciálních rovnic je charakteristický rovněž při řešení soustav diferenciálních rovnic s proměnnými koeficienty (tvořící diferenciální rovnice "vygenerují" požadované proměnné koeficienty).

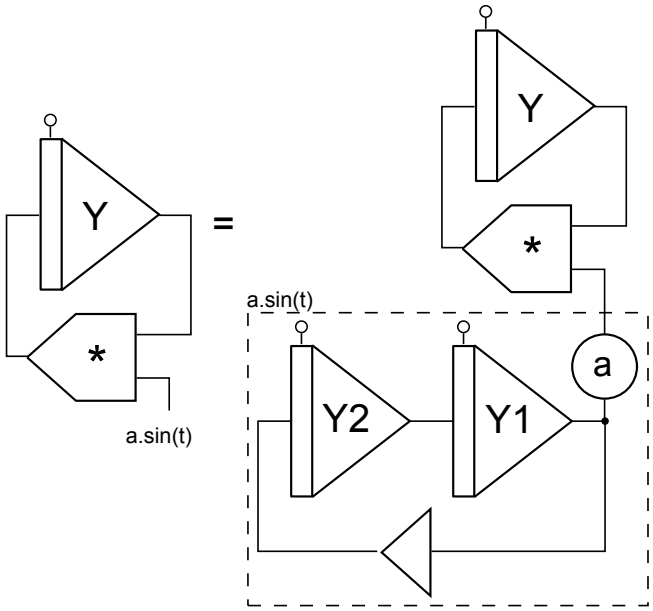
Např. při řešení rovnice (2.83)

$$y' = a{\cdot}y{\cdot}\sin(t) \qquad y(0) = y_0 \tag{2.83}$$

bude mít nově vzniklá soustava diferenciálních rovnic tvar

$$\begin{aligned} y' &= a{\cdot}y{\cdot}y_1 & y(0) &= y_0 & (2.84)\\ y_1' &= y_2 & y_1(0) &= 0 & (2.85)\\ y_2' &= -y_1 & y_2(0) &= 1 & (2.86) \end{aligned}$$

Na obrázku 2.4 je uvedeno blokové schéma odpovídající původní rovnici (2.83) a nově vzniklé soustavě diferenciálních rovnic(2.84)–(2.86). V obrázku je použit známý symbol násobičky.



Obrázek 2.4: Blokové schéma rovnice (2.83) před a po transformaci

Ke generování funkce $\sin(t)$, jak bylo vidět v předešlých dvou příkladech, je použita soustava dvou tvořících diferenciálních rovnic. Podobné výrazy (tvořící diferenciální rovnice) mohou být vytvořeny pro všechny elementární funkce, jako jsou $\exp, \sin, \cos, \tan, \coth, \ln, \sinh, \ldots$.

Potom může být zadaná diferenciální rovnice (pravá strana rovnice) rozložena na sekvenci jednoduchých operací: sčítání, odčítání, násobení, dělení a jejich kombinaci. Získání vyšších derivací těchto nově vzniklých rovnic (2.80)–(2.82) nebo (2.84)–(2.86) které se využívají v Taylorově řadě již není problém. Kompletní tabulka tvořících diferenciálních rovnic je obsažena v [Hol99].

Nově popsanou metodikou tvořících diferenciálních rovnic lze tedy řešit soustavy homogenních diferenciálních rovnic, nehomogenních diferenciálních rovnic, diferenciálních rovnic s proměnnými koeficienty i soustavy nelineárních diferenciálních rovnic.

Pro metodiku tvořících diferenciálních rovnic je charakteristické, že dokonce již jednu nelineární diferenciální rovnici lze řešit pomocí paralelně spolupracujících procesorů (integrátorů). Obecně je počet paralelně spolupracujících procesorů (integrátorů) roven počtu diferenciálních rovnic 1. řádu, na který byl zadaný problém převeden.

Pro nelineární diferenciální rovnici tvaru

$$y' = \sin(\sqrt{\cos t}) \qquad y(0) = y_0. \tag{2.87}$$

Formální jednoduchá úprava je zřejmě

$$\begin{aligned} y' &= y_1 \qquad y(0) = y_0 && (2.88)\\ y_1 &= \sin y_3 && (2.89)\\ y_3 &= \sqrt{y_4} && (2.90)\\ y_4 &= \cos t && (2.91) \end{aligned}$$

Pro rovnici (2.89) platí

$$\begin{aligned} y_1' &= y_2 \cdot y_3' \qquad y_1(0) = y_{1,0} = \sin y_{3,0} && (2.92)\\ y_2' &= -y_1 \cdot y_3' \qquad y_2(0) = y_{2,0} = \cos y_{3,0} && (2.93) \end{aligned}$$

Pro rovnici (2.90) platí

$$y_3' = \frac{1}{2}\frac{1}{y_3} y_4' \qquad y_3(0) = y_{3,0} = \sqrt{y_{4,0}} \tag{2.94}$$

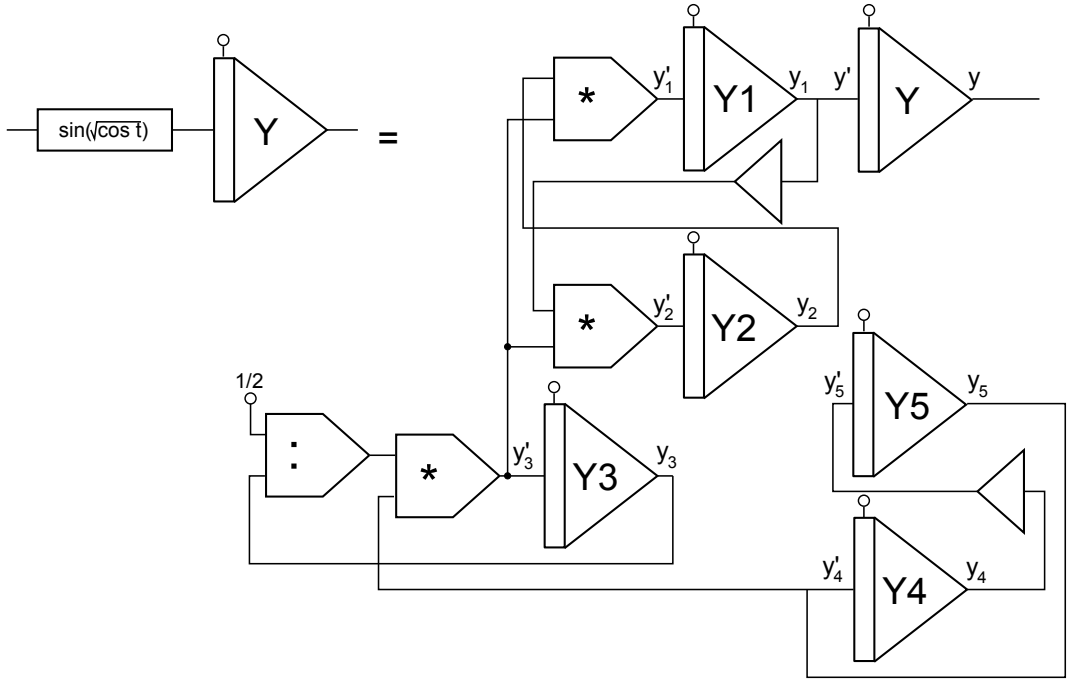
Pro rovnici (2.91) platí

$$\begin{aligned} y_4' &= y_5 \qquad y_4(0) = y_{4,0} = 1 && (2.95)\\ y_5' &= -y_4 \qquad y_5(0) = y_{5,0} = 0 && (2.96) \end{aligned}$$

Jak již bylo uvedeno, počet paralelně spolupracujících procesorů (integrátorů) se rovná počtu diferenciálních rovnic 1. řádu. Celkem bude v tomto příkladě tedy 6 procesorů (integrátorů) - viz obrázek 2.5.

Ze zápisu rovnic (2.88), (2.92)–(2.96) vyplývá rovněž, které procesory (integrátory) budou navzájem propojeny mezi sebou:

- výsledné řešení y získáme (dle rovnice 2.88) na výstupu procesoru Y
- podle zápisu rovnice 2.88 bude vstupem procesoru Y výstup procesoru Y1
- podle zápisu rovnice 2.92 bude vstupem procesoru Y1 výstup procesoru Y2 a výstup procesoru Y3
- podle zápisu rovnice 2.93 bude vstupem procesoru Y2 výstup procesoru Y1 a výstup procesoru Y3
- podle zápisu rovnice 2.94 bude vstupem procesoru Y3 výstup procesoru Y3 a výstup procesoru Y4
- podle zápisu rovnice 2.95 bude vstupem procesoru Y4 výstup procesoru Y5
- podle zápisu rovnice 2.96 bude vstupem procesoru Y5 výstup procesoru Y4



Obrázek 2.5: Odpovídající blokové schéma nelineární diferenciální rovnice (2.87)

2.3 Shrnutí

Jak bylo vidět v této kapitole, každou diferenciální rovnici, resp. soustavu rovnic, lze převést na ekvivalentní blokové schéma. Tento způsob je typický pro analogové počítače, kde se podle blokového (analogového) schématu provedlo zapojení jednotlivých prvků a následně jsme obdrželi požadované řešení. Bližší popis analogových počítačů je v [BHSL82] a [Mac07]. Abychom toto analogové schéma, resp. jemu odpovídající soustavu diferenciálních rovnic, vyčíslili v diskrétním výpočetním systému, je potřeba realizovat všechny elementární prvky v podobě diskrétních výpočetních jednotek.

Metoda Taylorovy řady poskytuje velice přesné řešení obyčejných diferenciálních rovnic. Ze srovnání obsaženého v této kapitole je zřejmé, že při použití metody Runge-Kutta a Taylorovy řady stejných řádů, je počet výpočetních operací při použití Taylorovy řady menší. Tento rozdíl se ještě zvětší, pokud se použije vyšší řád metody.

Při využití tvořících diferenciálních rovnic lze řešit i nelineární diferenciální rovnice. Nově vzniklé soustavy diferenciálních rovnic lze řešit velmi efektivně paralelně.

Kapitola 3

Paralelní architektury a modely

Paralelní systém (počítač) můžeme definovat jako množinu počítačů (procesorů či jader), které kooperují a komunikují, aby rychle řešily velké problémy. Paralelizací usilujeme především o zkrácení doby výpočtu, případně o možnost zpracovat více dat najednou.

Každou prováděnou větev paralelního programu nazýváme proces. Mezi souběžnými procesy většinou probíhá komunikace, jejíž technická realizace je závislá na architektuře paralelního systému. Existují dvě hlavní třídy těchto systémů, se sdílenou nebo distribuovanou pamětí.

- Systémy se **sdílenou pamětí** - obsahují množinu procesorů, které všechny sdílejí stejný adresový prostor. Komunikace probíhá tak, že jeden proces zapíše data a druhý (druzí) je čtou. Je nutná synchronizace při čtení/zápisu z/do této paměti.
- Systémy s **distribuovanou pamětí** - každý procesor má svou paměť, procesory jsou propojeny propojovací sítí a vyměňují si data mezi sebou. Komunikace mezi nimi probíhá pomocí zasílání zpráv.

V současné době je trend umístit na jeden čip více jader (běžně 2, 4, 8, nebo 16). Díky tomu se dosahuje vyšší výkonnosti při stejné či dokonce nižší spotřebě v porovnáním s jedno-jádrovým procesorem. Dnešní běžné osobní počítače můžeme tedy považovat za paralelní systémy.

Jedno-procesorové počítače (s jedním jádrem) se již v dnešní době téměř nevyvíjí, protože neposkytují dostatečný růst výkonu a dochází naopak ke značnému nárůstu příkonu a teploty.

3.1 Úrovně paralelismu

Je užitečné připomenout, že termín "paralelní zpracování" se také používá v dalších významech. Při počítačovém zpracování rozlišujeme minimálně mezi čtyřmi úrovněmi paralelismu:

- uvnitř instrukcí - nejjemnější paralelismus na úrovni bitů, použití 8, 16, 32, 64-bitových procesorů
- mezi instrukcemi - některé instrukce se provádějí paralelně, ale není to vidět na úrovni programovacího jazyka (zřetězené zpracování, zapojením více funkčních jednotek např. pro operace v pohyblivé řádové čárce)

- mezi bloky procesu (vlákna, threads) - proces je rozdělen do několika vláken, u jednoprocesorového systému se běh vláken postupně střídá na tomto procesoru, u více procesorového chipu běží vlákna na všech jádrech současně, vlákna využívají společnou sdílenou paměť
- mezi procesy - běh úlohy (paralelního programu) iniciuje množinu procesů (např. příkazem `fork`), které jsou vykonávány na jednotlivých procesorech víceprocesorového systému a jež spolupracují k jejímu vyřešení, každý proces má celou paměť jen sám pro sebe - nesdílí nic s ostatními procesy

U prvních dvou možností nemusí uživatel/programátor mít žádné větší znalosti o daném systému ani o speciálních konstrukcích používaných při programování. O využití těchto zdrojů se postará kompilátor.

Pokud chce využívat paralelismus na úrovni vláken a procesů, musí již tuto možnost explicitně zadat při tvorbě programů - počítat s režijí spojenou s vytvářením vláken/procesů, myslet na úskalí při práci s pamětí, komunikací a synchronizací.

Bližší popis architektur procesorů a způsobů využití popisovaných metod počítačového zpracování (typy instrukcí, jednotky využívané v procesorech, zřetězené zpracování a principy multivláknového provozu) lze nalézt v [DD99] a [HP06].

3.2 Modely paralelních architektur

Je více hledisek, podle kterých lze (víceprocesorové) počítačové systémy klasifikovat. Tato práce využívá tzv. Flynnovu klasifikaci a dělení podle uspořádání operační paměti.

M. J. Flynn provedl první základní rozdělení podle počtu instrukčních a datových proudů, které lze v jednom okamžiku na dané počítačové architektuře zpracovávat. Z anglických výrazů *Single/Multiple Instruction - Single/Multiple Data* jsou odvozeny názvy jednotlivých kategorií:

1. **SISD** (Single Instruction Single Data)
   Nejedná se vlastně o paralelní architekturu, jeden procesor provádí jednu sekvenční posloupnost instrukcí nad daty uloženými v jedné paměti.

2. **SIMD** (Single Instruction Multiple Data)
   Procesory, které mají společné řízení a pracují paralelně nad různými daty. Mohou mezi sebou komunikovat zasíláním zpráv nebo přes sdílenou paměť. Výhodou je implicitní synchronizace procesorů a menší náročnost na hardware, protože se zde využívá pouze jedna společná řídící jednotka. Nevýhodou tohoto modelu je menší univerzálnost.

3. **MISD** (Multiple Instruction Single Data)
   Stejná data jsou postupně zpracována více jednotkami. Typickým zástupcem jsou systolická pole [Pet92], což jsou velmi speciální architektury. MISD architektury jsou využívány jen velmi zřídka.

4. **MIMD** (Multiple Instruction Multiple Data)
   Paralelně pracující procesory, které jsou řízeny nezávisle, z nichž každý pracuje nad odlišnými daty. Mohou mezi sebou komunikovat zasíláním zpráv nebo přes sdílenou paměť. Předností tohoto modelu je velká univerzálnost.

Téměř všechny významné paralelní architektury současné doby patří do MIMD kategorie. Díky tomu Flynnovy taxonomie ztrácí na svém významu a bylo proto zavedeno další rozdělení paralelních architektur a to podle komunikace:

1. **systémy se sdílenou pamětí**
   Obsahují fyzicky sdílenou paměť, do které mají všechny procesory stejně rychlý přístup - **UMA** (*Uniform Memory Access*) architektury. Stejná adresa na různých procesorech odkazuje na stejnou fyzickou paměťovou buňku. Sdílená paměť je tu prostředkem komunikace. Typickým příkladem je SMP - symetrický multiprocesing. Sdílená paměť se stává úzkým hrdlem celého systému, proto se tyto architektury omezují na maximálně 100 procesorů.

2. **systémy s distribuovanou pamětí**
   Nemají společnou paměť ani virtuální adresový prostor. Komunikují spolu zasíláním zpráv přes komunikační síť. Tento přístup, kdy je odstraněna společná paměť, umožňuje vytvářet systémy obsahující tisíce procesorů.

3. **systémy se sdíleně distribuovanou pamětí**
   Jedná se o architektury s distribuovanou pamětí, které mají podporu pro sdílený virtuální adresový prostor - **NUMA** (*Non-Uniform Memory Access*) architektury. K lokálním datům bývá rychlejší přístup než ke sdíleným. Podpora pro virtuální adresový prostor bývá zabudovaná již na úrovni hardware.

Programování založené na posílání zpráv je náročnější. Distribuovaná paměť vyžaduje rozčlenění a distribuci jak dat tak práce mezi procesory. Posílání zpráv lze snadno a efektivně emulovat na systémech se sdílenou pamětí. Neplatí to však naopak.

3.3 Paralelní zpracování

Při paralelním zpracování je problém rozdělen na jistý počet souběžných procesů / vláken. Paralelní programování musí mít příkazy (explicitní nebo implicitní) pro:

- správu procesů / vláken - tvorba procesů, ukončování, přepínání kontextu
- interakci procesů - komunikace, synchronizace (bariéra, zámky, kritické oblasti)

3.3.1 Režie paralelního zpracování

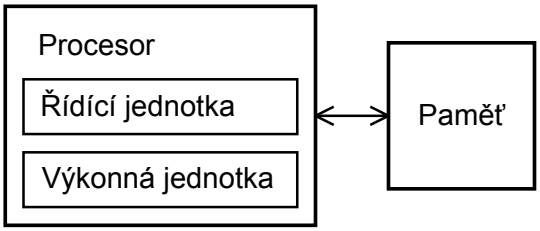
Výpočetní operace jako v sekvenčních programech představují užitečnou práci. Naproti tomu režie paralelního zpracování je tvořena:

- čekáním na jiné procesory až dokončí svoji činnost - nestejné využití procesorů
- správou procesů - přepínáním kontextu
- interakcí procesů - komunikací, synchronizací procesů nebo vláken, které u sekvenčního programu nemáme

3.4 Typy paralelních počítačů

3.4.1 Sekvenční počítač

Pro odvození složitějších paralelních architektur může být použita jednoduchá představa o konvenčním sekvenčním počítači, obsahující jediný procesor, jenž má přístup do operační paměti - tedy klasický von Neumannův model. Procesor obsahuje řídící jednotku a výkonnou jednotku, která provádí aritmeticko-logické operace. Schematicky je tento model zobrazen na obrázku 3.1.

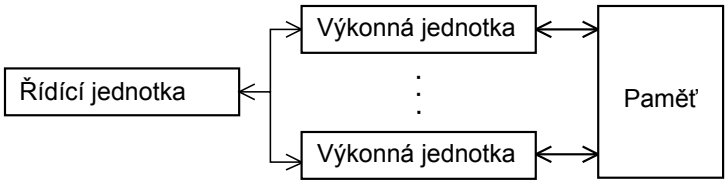


Obrázek 3.1: Architektura sekvenčního počítače

Sekvenční počítač zpracovává instrukci po instrukci program, který je uložen v paměti, jak to odpovídá architektuře SISD Flynnovy klasifikace.

3.4.2 Vektorový počítač

Vektorový počítač může být zařazen do architektury SIMD. Vektorové počítače jsou totiž specializovány na efektivní provádění stejných aritmetických operací nad dlouhými posloupnostmi (vektory) čísel. Lze si je tedy představit jako stroje, kde byly v porovnáním se sekvenčním počítačem znásobeny jeho výkonné jednotky, které jsou řízeny řídící jednotkou a provádějí danou operaci nad více složkami vektorů najednou. Schematicky je tento model zobrazen na obrázku 3.2.



Obrázek 3.2: Architektura vektorového počítače

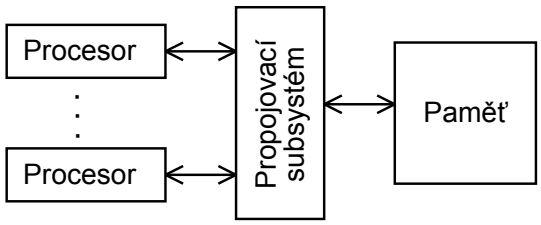
Ve skutečnosti se ve vektorových počítačích uplatňuje především proudové zpracování (pipelining), které dovoluje rozpracovat několik instrukcí zároveň zřetězením na úrovni mikroinstrukcí, takže v každém taktu procesor vydává výsledek jedné operace. Více informací o vektorových procesorech obsahuje kniha [DD99].

3.4.3 Symetrický multiprocesor SMP

Tento typ si lze představit tak, že model sekvenčního počítače rozšíříme o další, totožné procesory. Lze je vyrábět z běžných a tedy i levných procesorů. Přístup více procesorů ke společné sdílené paměti zařizuje propojovací subsystém.

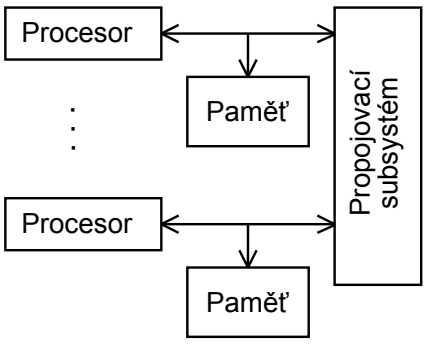
Tento subsystém se se zvyšujícím počtem procesorů stává úzkým hrdlem omezujícím tok dat mezi procesory a pamětí. S rostoucím počtem procesorů vznikají problémy při obsluze všech požadavků. Proto se zavedly také multiprocesorové systémy s distribuovanou pamětí. Multiprocesorové architektury se tedy dělí na dva druhy:

- **systémy se sdílenou pamětí**
  Tyto systémy jsou odvozeny z jednoprocesorového systému pouhým přidáním dalších CPU propojených sběrnicí. Všechny procesory jsou rovnocenné a přístup do sdílené globální paměti je vždy stejně rychlý - UMA architektura. Příkladem symetrických multiprocesorů jsou dnes běžné vícejádrové počítače. Procesory bývají zpravidla rozšířeny lokálními vyrovnávacími pamětmi cache, aby se snížily požadavky na přenos dat přes propojovací subsystém. Schematicky je tento model zobrazen na obrázku 3.3.



Obrázek 3.3: Architektura symetrického multiprocesoru se sdílenou pamětí

- **systémy s distribuovanou pamětí**
  Každý procesor má rychlý přístup do své paměti ale velice pomalý přístup do cizí paměti. Předpokládá se, že většina přístupů do paměti bude probíhat na úrovni lokální paměti daného procesoru, což snižuje zátěž na propojovací systém. To umožňuje vytvářet systémy s mnohem větším počtem procesorů (řádově tisíce). Soubor všech lokálních pamětí jednotlivých procesorů dohromady tvoří jeden logický adresový prostor. Protože se rychlost přístupu k jednotlivým adresám liší, jedná se o NUMA architekturu. Schematicky je tento model zobrazen na obrázku 3.4.

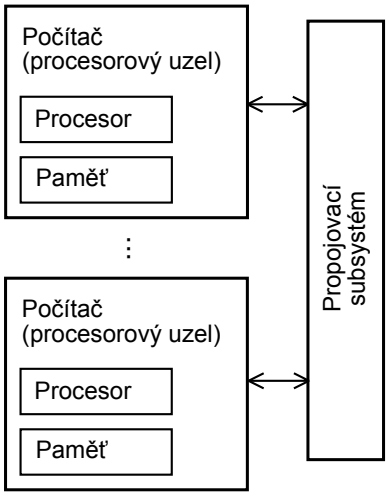


Obrázek 3.4: Architektura symetrického multiprocesoru s distribuovanou pamětí

Procesory pracují nezávisle, tj. mají své samostatné instrukční a datové proudy. Symetrické multiprocesory tedy náleží do architektury MIMD Flynnovy klasifikace.

3.4.4 Multipočítač

Jedná se o masivně paralelní systém. Lze si ho představit jako kolekci sekvenčních počítačů, často označovaných jako procesorové uzly, které jsou svázány propojovacím subsystémem. Jelikož tento subsystém nemá za úkol přenášet data mezi procesory a operační pamětí, nýbrž propojovat relativně samostatné počítače, má spíše charakter sítě. Každý procesor má přístup jen do vlastní paměti. Distribuovaná paměť zde neumožňuje využití systému se sdílenými proměnnými. Takže procesy běžící na různých procesorových uzlech komunikují přes propojovací síť výhradně pomocí zasílání zpráv. Masivně paralelní systémy mají pochopitelně architekturu MIMD Flynnovy klasifikace. Schematicky je tento model zobrazen na obrázku 3.5.



Obrázek 3.5: Architektura multipočítače

Jednou podmnožinou masivně paralelních systémů jsou klastry (cluster). Uzly klastru (jednotlivé procesorové uzly - počítače) nemají připojené zobrazovací zařízení a na všech uzlech by měl běžet identický operační systém. Na kvalitu síťového propojení je kladen velký důraz.

Druhou podmnožinou je paralelní systém označovaný jako grid. Podobný systém jako klastr obsahující tisíce uzlů. Jednotlivé uzly jsou ale většinou běžná PC, která disponují různým výkonem, architekturou i operačním systémem.

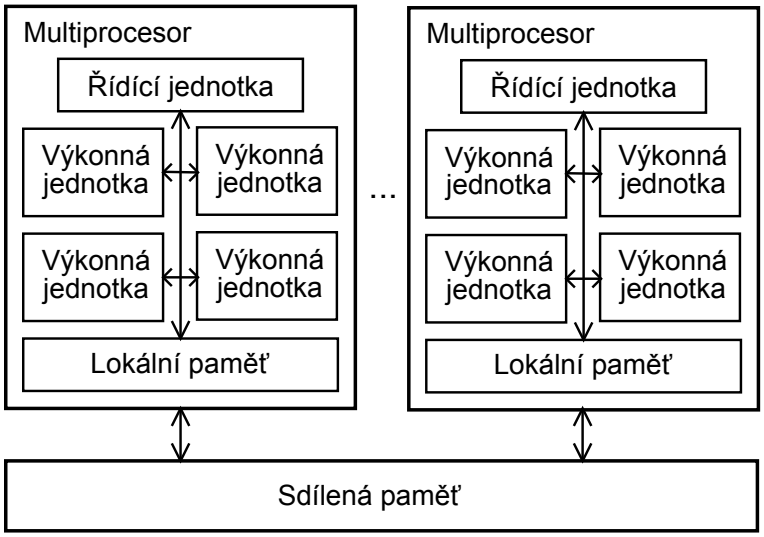
3.4.5 GPU

GPU = graphical processing unit (grafický procesor). Současné grafické karty disponují vyšším výkonem než obyčejné procesory. Proto je snaha provádět složité výpočty na grafické kartě místo na procesoru.

GPU je optimalizován převážně pro jednodušší aritmetické operace v pohyblivé řádové čárce. Na rozdíl od CPU má méně vnitřní logiky - jsou méně univerzální. Hlavní rozdíl je však v počtu jednotek provádějící výpočetní operace. Současné moderní CPU mají 2 až 8 jader, zatímco GPU obsahuje řádově stovky výpočetních jednotek. GPU neposkytuje tolik funkčnosti jako CPU, zato je velice rychlý. U GPU je určitá operace prováděná zároveň nad všemi daty. To odpovídá architektuře SIMD Flynnovy klasifikace.

GPU se skládá z několika částí. Základ tvoří tzv. multiprocesor, který obsahuje řídící jednotku, několik výpočetních jednotek a lokální paměť. Tyto multiprocesory se skládají do

větších celků. Počet multiproceosrů závisí na typu grafické karty. Do lokální paměti mají velmi rychlý přístup výkonné jednotky v daném multiprocesoru. Pro ostatní multiprocesory je tato paměť nedostupná. Všechny multiprocesory mají přístup do společné sdílené paměti, tento přístup je však podstatně pomalejší než do jednotlivých lokálních pamětí. Schematicky je tento model zobrazen na obrázku 3.6.



Obrázek 3.6: Architektura GPU

Existují dva hlavní výrobci grafických čipů, společnost AMD/ATI a nVidia. Každá vyvinula svoje technologie pro programování, nVidia používá technologii *CUDA* (Compute Unified Device Architecture) [nVi10] a AMD/ATI používá technologii *ATI Stream* [ATI10]. V současné době však oba výrobci přistoupili i na standard *OpenCL* (Open Computing Language), který vznikl pod záštitou konsorcia Khronos [Khr10] (tvořená společnostmi jako AMD, Intel, nVidia, IBM, a dalšími). OpenCL je nyní průmyslový standard pro paralelní programování heterogenních počítačových systémů (tedy ne jen GPU).

Bližší popis paralelních architektur lze nalézt v [Dvo04], [Tvr03], [Jak05].

3.5 Shrnutí

Paralelní přístup může přinést podstatné zvýšení výkonnosti. Při výběru vhodné architektury je nutné mít dostatečné znalosti o typu řešených úloh, aby nedošlo k tomu, že paralelní výpočet bude ve výsledku pomalejší než sekvenční.

V našem případě, kdy chceme řešit rozsáhlé soustavy diferenciálních rovnic, je doba výpočtu jedné diferenciální rovnice stejná jako při sekvenčním zpracování. Zrychlení výpočtu je dosaženo tím, že dokážeme najednou řešit všechny rovnice dané soustavy diferenciálních rovnic. Zpracováváme tedy více dat najednou. Z toho vyplývá, že výsledná paralelní architektura bude SIMD nebo MIMD.

Ne vždy jsou výpočetní operace v jednotlivých větvích paralelního zpracování na sobě nezávislé a je nutné řešit vzájemnou komunikaci. Volba optimální propojovací sítě a způsobu komunikace bude mít zásadní vliv na celkové zrychlení paralelního zpracování oproti sekvenčnímu. V následující kapitole jsou představeny nejznámější propojovací sítě a popsány jejich nejdůležitější vlastnosti.

Kapitola 4

Propojovací sítě

Všechny typy paralelních systémů potřebují realizovat komunikaci mezi svými jednotkami. Fyzikální realizací používanou pro komunikaci mezi jednotkami je propojovací síť. Propojovací sítě používané v paralelních systémech můžeme rozdělit z hlediska připojení jednotek na jednostranné (propojují jednotky stejného typu, např. CPU–CPU) a oboustranné (propojují jednotky různého typu např. CPU–paměť).

Ukazuje se, že při datových přenosech mezi větším množstvím uzlů pomocí propojovací sítě hraje rozhodující úlohu topologie sítě a méně již její bitová šířka či rychlost. Proto je pro využití výkonnosti paralelních počítačů návrh vysoce výkonné propojovací sítě velmi důležitý. Prostředky a požadavky používané v různých systémech jsou velmi různorodé a typy používaných sítí se tedy vzájemně velmi liší. Propojovací sítě se dělí do dvou základních částí:

- **přímé** (statické) propojovací sítě:
  Během činnosti systému se přímá propojovací síť nemůže měnit a proto je třeba její strukturu od samého začátku přizpůsobit předpokládané činnosti. Zprávy jsou zasílány přímo mezi koncovými uzly.

- **nepřímé** (dynamické) propojovací sítě:
  Nepřímé propojovací sítě nepropojují uzly systému přímo, ale přes směrovací prvky - přepínače. Zprávy jsou zasílány nepřímo pomocí k tomu určených směrovacích uzlů. Typ a propojení těchto prvků určuje další vlastnosti sítě.

4.1 Přímé propojovací sítě

Přímé propojovací sítě propojují přímo koncové uzly (výpočetní jednotky). Neobsahují žádné čistě směrovací prvky (přepínače), jak je tomu u nepřímých propojovacích sítí.

Přímé propojovací sítě se zpravidla popisují pomocí grafů. V tomto grafu každý vrchol odpovídá jednomu modulu systému (procesoru, paměti) a každá hrana jednomu komunikačnímu spoji. Každý vrchol grafu spojuje začátek případně konec jedné přenosové linky. Tyto linky tedy nejsou přímo propojeny.

Pokud chceme zprostředkovat informaci mezi dvěma moduly, které nejsou bezprostředně propojeny jednou přenosovou linkou, musíme překonat n přenosových kroků prostřednictvím modulů umístěných v požadované cestě přenosu informace. Součástí každého modulu (v našem případě uvažujeme procesor) musí být tedy jakýsi směrovač, který se stará o komunikaci (vytvoření spojovací cesty) mezi ním a s ním spojenými uzly.

V acyklických grafech (v nichž neexistuje kružnice) je volba spojovací cesty jednoznačná. Nevýhodou je, když nastane porucha některého uzlu. Ta znemožní spojení některé cesty, případně celé sítě. Grafy cyklické (obsahující kružnici) naproti tomu umožňují spojení libovolných dvou vrcholů více cestami. Případná porucha některého uzlu zde zcela neznemožní spojení, protože pro toto spojení existuje jiná cesta.

Statická síť se během činnosti systému nemůže měnit, proto je třeba její strukturu od samého začátku přizpůsobit předpokládaným činnostem systému. Z tohoto důvodu musíme pro každý typ systému v závislosti na předpokládaném typu propojení a komunikace použít jinou statickou síť. Z potřeby různých požadavků se vyvinul poměrně velký počet přímých propojovacích sítí. Uvádím zde pouze několik základních typů přímých propojovacích sítí, které se v praxi nejčastěji vyskytují.

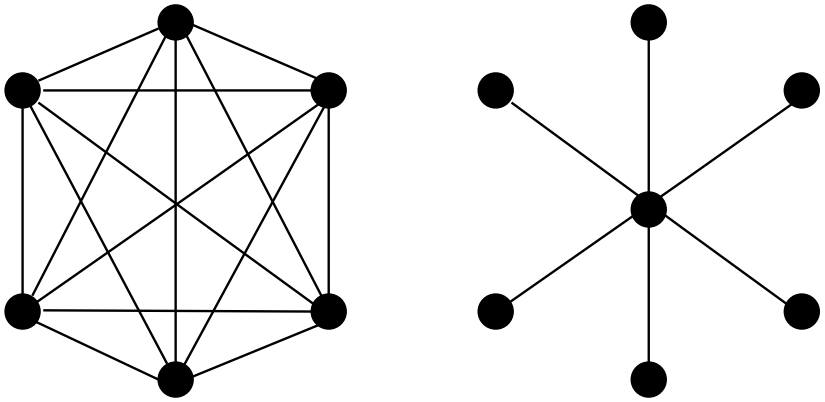
4.1.1 Úplně propojená síť

Ideální propojovací síť je plné propojení každého uzlu systému s každým (libovolné dva uzly jsou spojeny hranou). Příklad sítě s úplným propojením je na obrázku 4.1 vlevo.

V libovolný okamžik může komunikovat libovolný uzel s libovolným uzlem. Úplně propojená síť je velmi špatně rozšiřitelná (škálovatelná), protože velmi roste složitost uzlů (nutnost napojení více linek na každý uzel) a rostoucí počet linek komplikuje realizaci propojení. Tato síť je velmi drahá. Je použitelná do malého počtu uzlů (řádově jednotky). V praxi se pro desítky a stovky procesorů používají ekonomičtější sítě s řidším než úplným propojením.

4.1.2 Hvězdicová síť

Má jeden centrální uzel, který zajišťuje komunikaci mezi ostatními. Centrální uzel, přes který probíhá veškerá komunikace, je ale úzkým hrdlem této sítě. Pokud tento uzel přestane pracovat, propojovací síť se stane nefunkční. Příklad hvězdicové sítě s šesti výpočetními a centrálním uzlem je na obrázku 4.1 vpravo.



Obrázek 4.1: Topologie sítě úplné propojení a hvězdicové

4.1.3 Ortogonální sítě

Významná skupina přímých propojovacích sítí. Jednotlivé uzly jsou umístěny v souřadnicovém prostoru, obecně k uzlů v každé z n dimenzí. Existují dva základní typy ortogonálních sítí: lineární řetězec a kruh. Ostatní sítě se odvozují pomocí kartézských součinů grafů představující tyto dvě základní sítě.

Základní ortogonální topologie jsou:

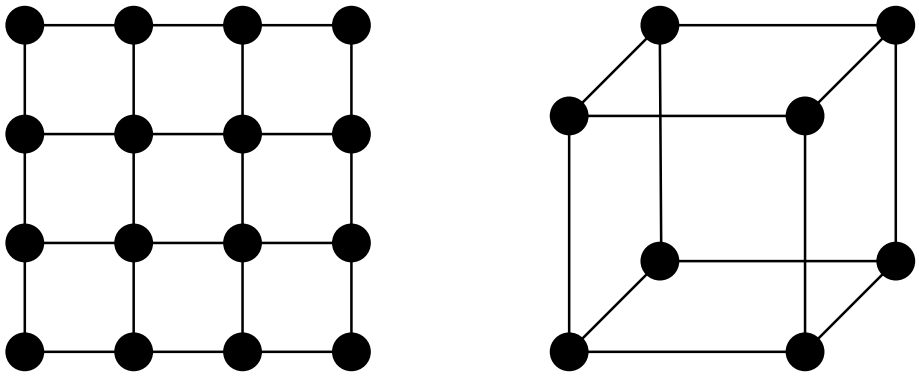
1. **lineární řetězec**
   Ortogonální síť dimenze $n = 1$. Všech k uzlů kromě prvního a posledního mají dva sousedy. Na této topologii může probíhat několik přenosů současně. Vzdálenost dvou uzlů je ale dosti rozdílná. Např. při komunikaci prvního s posledním musí přenos probíhat přes všechny uzly a blokuje tak přenosové prostředky ostatním. Obrázek 4.2 vlevo.

2. **kruh**
   Ortogonální síť dimenze $n = 1$. Vychází z lineárního řetězce, v němž je spojen první a poslední uzel. Pokud je komunikace po kruhu obousměrná, zkrátí se maximální vzdálenost dvou uzlů na polovinu v porovnáním s lineárním řetězcem. Obrázek 4.2 vpravo. Graf této propojovací sítě již obsahuje kružnici. Tzn. existuje více možných cest z jednoho uzlu do druhého. Pokud se jeden uzel stane nefunkční, ostatní uzly spolu mohou stále komunikovat.

Obrázek 4.2: Ortogonální sítě: lineární řetězec a kruh

3. **mřížka**
   Ortogonální topologie dimenze $n = 2$, která vznikne kartézským vynásobením lineárního řetězce. Obrázek 4.3 vlevo zobrazuje 4-ární ($k = 4$) mřížku.

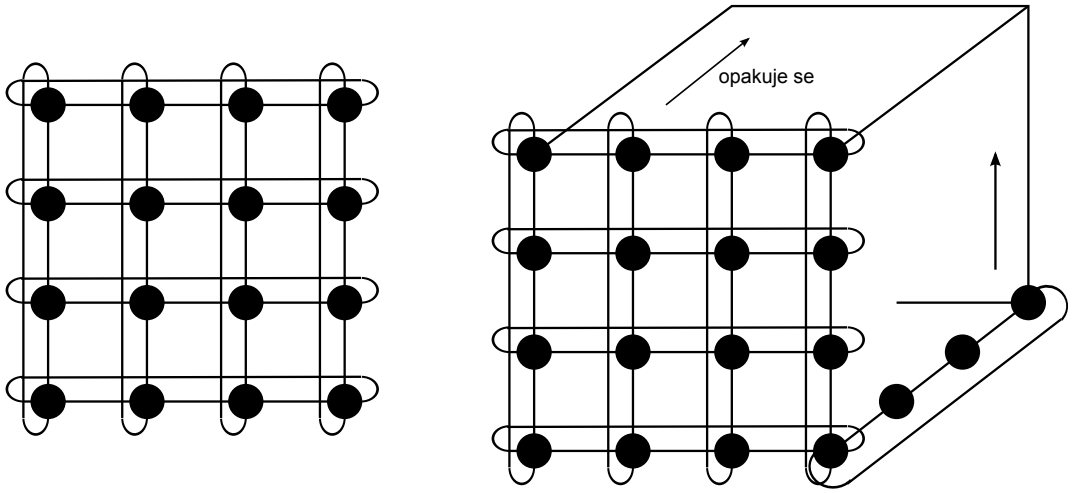
4. **kostka**
   Ortogonální topologie dimenze $n = 3$. Vznikne vynásobením binární mřížky ($k = 2$). Obrázek 4.3 vpravo.



Obrázek 4.3: Ortogonální sítě: mřížka a kostka

5. **2D torus**
   Vznikne z mřížky tak, že první a poslední prvek každé dimenze (řádku a sloupce) dané mřížky se propojí - vytvoří kruhovou síť. Tím naroste počet možných komunikačních kanálů mezi jednotlivými uzly. Obrázek 4.4 vlevo.

6. **3D torus**
   Ortogonální síť, která vynikne ze sítě 2D torus rozšířením o další dimenzi $n = 3$. Obrázek 4.4 vpravo.



Obrázek 4.4: Ortogonální sítě: 2D torus a 3D torus

4.2 Nepřímé propojovací sítě

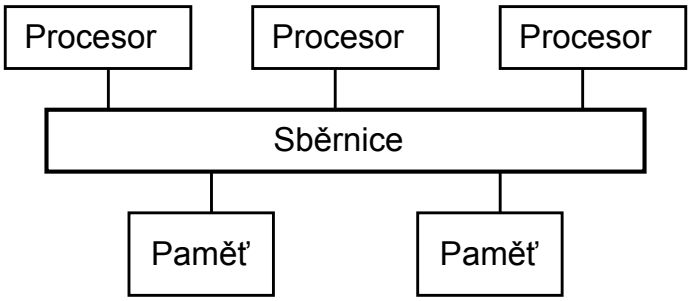
Nepřímé propojovací sítě neobsahují jen koncové uzly (procesor, paměť), ale i samostatné směrovací prvky - přepínače. Uzly systému nejsou propojeny přímo, ale přes tyto směrovací prvky. Pokud chceme zprostředkovat informaci mezi dvěma uzly, musí zpráva vždy putovat mezi přepínači, které jsou umístěny v požadované cestě přenosu informace mezi těmito uzly. Přepínače nejsou zdrojem ani cílem dat.

V porovnáním s přímými propojovacími sítěmi je zde nižší počet spojů (a tedy i cena). Často však bývají blokující, tzn. neumožňují provést současně dvě spojení, která vyžadují jeden přepínač v různých stavech.

4.2.1 Sběrnice

Patří mezi nepřímé propojovací sítě. Jedná se o nejjednodušší a velmi používaný typ propojení. Skládá se z mnoha linek, které jsou připojeny ke všem modulům (představující procesory či paměti) systému. Komunikační spoj je sdílen všemi připojenými zařízeními.

Každé zařízení jen poslouchá probíhající komunikaci na sběrnici a čeká na signál, který určuje, že daná komunikace se týká i jeho. Jakmile tento signál obdrží, připojí se. Současně mohou komunikovat jen dvě zařízení, ostatní mohou jen naslouchat. Možný způsob propojení modulů se sběrnicí je na obrázku 4.5.



Obrázek 4.5: Sběrnice

Na sběrnici se přenáší adresy, data, řízení,... Stále vyšší požadavky na výkonnost (propustnost) sběrnic vyžadují tyto části oddělené (ne multiplexování na jedné sběrnici). Z tohoto důvodu dnešní systémy obsahují oddělené datové, adresové či řídící sběrnice nebo jednu širokou sběrnici, která má ale určité vodiče vyhrazeny jen pro data, adresu nebo řízení.

Omezení, kdy celou sběrnici zablokuje jedna transakce, dovoluje připojit jen velmi omezený počet zařízení. Sběrnice se ale vyvíjely k vyšší výkonnosti a vznikly zřetězené sběrnice a sběrnice s rozdělenými transakcemi, které dovolují propojit i 16 až 32 procesorů. Dalším krokem k vyšší výkonnosti je použití několika paralelních sběrnic, alespoň adresových. Např. se 4 sběrnicemi můžeme přistupovat na 4 adresy současně.

Více informací o těchto vylepšeních sběrnic se nachází v [Dvo04, str. 20].

4.2.2 Křížové přepínače

Jedním z nejstarších prvků využívaným v nepřímých propojovacích sítích je křížový přepínač. Původně byl používaný v telefonních ústřednách. Byl sestaven z mechanických prvků, které umožňovaly obousměrný přenos signálů. V současné době bývá realizován pomocí spínacích prvků, které umožňují spojení pouze jedním směrem. Křížový přepínač $n \times m$ přímo propojuje n vstupů a m výstupů bez jakýchkoliv mezistupňů. Každý výstup musí být připojen nejvýše k jednomu vstupu, jeden vstup ale může být připojen k několika výstupům.

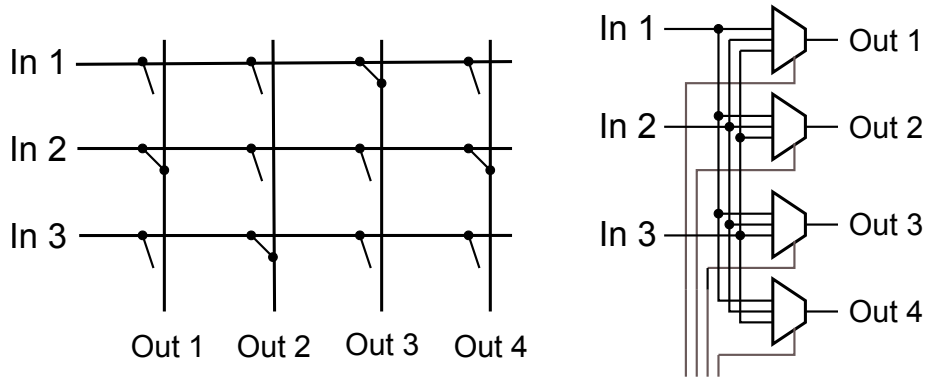
Velká výhoda křížového přepínače je vzájemná nezávislost propojovacích cest. Nevýhodou je jeho vysoká cena, protože počet spínacích prvků je roven součinu počtu modulů, které propojuje $(n \times m)$. Proto se většinou používá v jednodušších systémech s menším počtem modulů.

Na obrázku 4.6 vlevo je zobrazen model křížového přepínače se 3 vstupy (In 1 ... 3) a 4 výstupy (Out 1 ... 4). Vstup 1 je připojen na výstup 3, vstup 2 na výstup 1 a 4 a vstup 3 na výstup 2. Na obrázku 4.6 vpravo je model tohoto křížového přepínače realizován pomocí multiplexorů.

Ideální nepřímá propojovací síť je jediný N-cestný křížový přepínač, který by propojoval všech N uzlů systému, aniž by některé propojení vstupu s výstupem blokovalo jiné propojení. Pro velký počet uzlů je ale tento křížový přepínač velmi drahý.

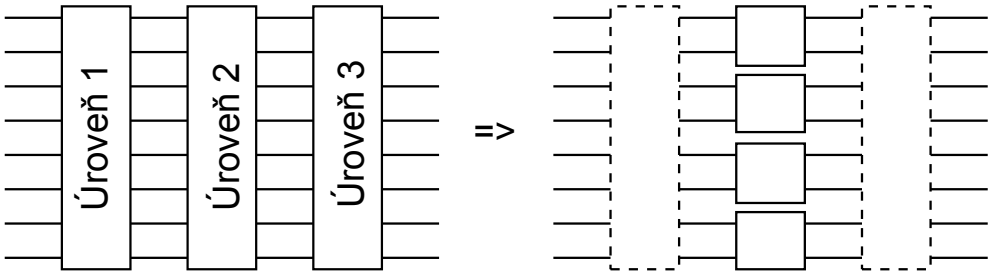
4.2.3 Víceúrovňové sítě

V důsledku vysoké ceny křížových přepínačů pro rozsáhlejší systémy byla navržena řada jiných sítí, kde zprávy musí projít přes několik přepínačů, než dosáhnou svého cíle. V těchto



Obrázek 4.6: Model křížového přepínače

sítích bývají přepínače identické a uspořádávají se do jednotlivých úrovní (stupňů). Proto se tyto sítě nazývají víceúrovňové. Schematicky je víceúrovňová propojovací síť zobrazena na obrázku 4.7.



Obrázek 4.7: Víceúrovňová propojovací síť

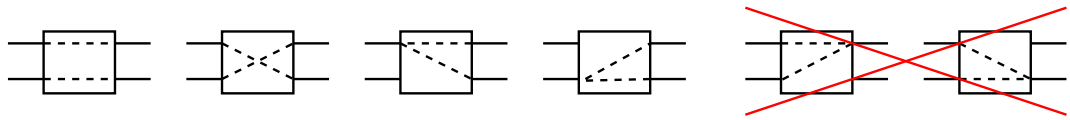
Víceúrovňová propojovací síť nejčastěji obsahuje stejný počet vstupů a výstupů. Mezi vstupy a výstupy jsou jednotlivé úrovně sítě, které jsou tvořeny:

- statickou částí - zajišťuje určité propojení (permutaci) vstupů a výstupů dané úrovně sítě (úroveň 1 a 3 na obrázku 4.7)
- dynamickou částí - tvořena přepínači, které realizují požadované propojení sítě (úroveň 2 na obrázku 4.7)

Víceúrovňové sítě snižují náklady křížových přepínačů ale za cenu toho, že některá spojení jsou blokující. Navíc nemají výkonové omezení při rozšíření sítě jako u sběrnic (lepší škálovatelnost). Dnes se různé varianty těchto sítí používají k propojení stovek procesorů v paralelních systémech.

Důležitým prvkem, který tvoří dynamickou část víceúrovňové propojovací sítě, je křížový přepínač. Stupeň použitých křížových přepínačů určuje i stupeň celé sítě. Nejčastěji se používá přepínač, který má dva vstupy a dva výstupy. Tyto sítě označujeme tedy jako binární.

Dvoucestný křížový přepínač podle obrázku 4.8 umožňuje propojení vstupu s výstupem: přímé, křížené a rozhlášení z jednoho nebo druhého vstupu na oba výstupy. Propojení obou vstupů na jeden výstup není přípustné.



Obrázek 4.8: Možná propojení dvoucestného křížového přepínače

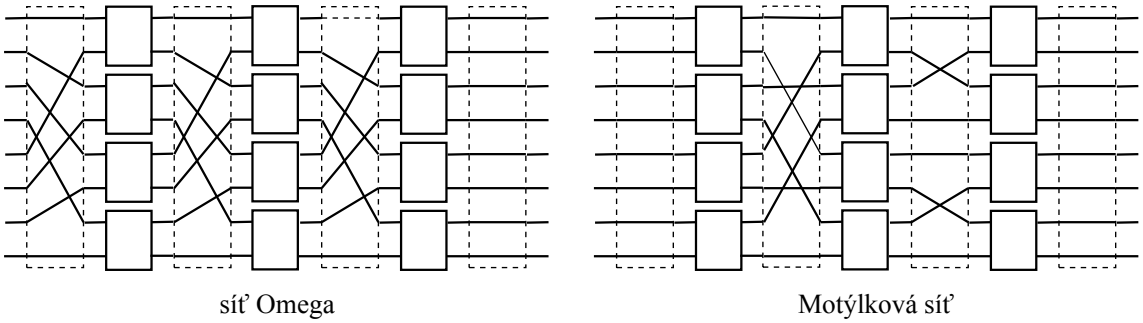
Podle přenosových schopností mohou být víceúrovňové propojovací sítě rozděleny do dvou hlavních skupin:

- blokující sítě
- neblokující sítě

Blokující sítě

Blokující sítě jsou nejjednodušší a nejlevnější propojovací sítě. Mají ale omezené komunikační schopnosti. V těchto sítích existuje alespoň jeden stav, pro který nelze najít cestu propojení. To může nastat v situaci, kdy některá komunikace potřebuje shodný výstup křížového přepínače, který je již ale obsazen jinou probíhající komunikací a nemohou proto proběhnout současně, ale jedna po druhé.

Z nejznámějších blokující sítí jsou to například síť Omega nebo Motýlková (Butterfly) síť. Schéma těchto sítí pro 8 vstupů a výstupů je na obrázku 4.9.



Obrázek 4.9: Blokující víceúrovňové propojovací sítě Omega a Motýlek (Butterfly)

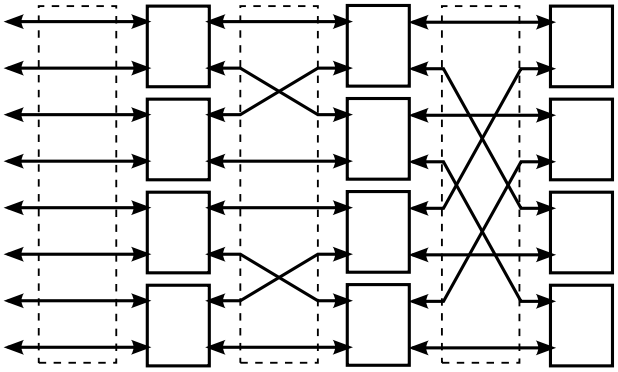
Uvedené vícestupňové blokující sítě se při hustším provozu snadno zahltí, protože z jednoho zdroje do určitého cíle vede právě jedna cesta. Proto byly vyvinuty i neblokující sítě, které toto zahlcení odstraňují.

Neblokující sítě

Jedna z možností, jak rozšířit počet cest mezi jednotlivými uzly, je použití *obousměrné sítě*. Obousměrná síť je sestavena stejně jako víceúrovňová blokující síť. Komunikační kanály mezi jednotlivými úrovněmi jsou ale obousměrné. Přesněji řečeno, každé propojení jednotlivých úrovní obsahuje oddělený kanál pro komunikaci jedním i druhým směrem. Tyto sítě nemají výstupy (bloky na pravé straně jako u blokujících sítí), ale vstupní bloky jsou

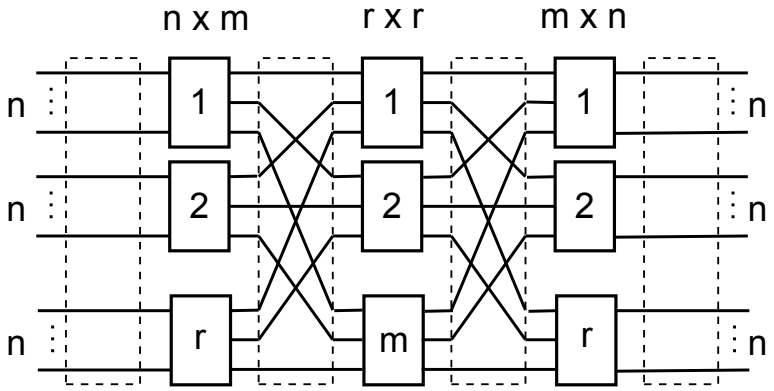
zároveň i cílové díky použití obousměrných komunikačních kanálů. Místo křížových přepínačů na poslední úrovni mají tyto sítě prvky, které mají propojeny oba výstupy.

Obousměrné sítě si lze představit jako dvě víceúrovňové sítě za sebou zrcadlově obrácené a navíc překlopené na sebe. Příklad takové sítě je na obrázku 4.10. Jedná se o motýlkovou síť pro 8 vstupů/výstupů.



Obrázek 4.10: Obousměrná motýlková neblokující síť

Další typ neblokující sítě je *Closova síť*. Jedná se o tříúrovňovou propojovací síť, která je sestavená z tří různých typů křížových přepínačů. Uvedena je na obrázku 4.11.



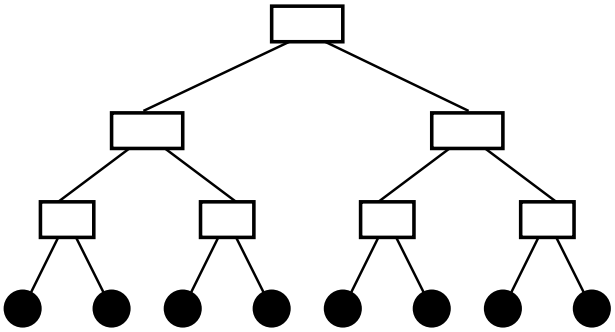
Obrázek 4.11: Neblokující Closova síť

V první úrovni jsou použity přepínače typu $n \times m$. Těchto přepínačů je r a síť má tedy $r{\cdot}n$ vstupů. Ve druhé úrovni jsou použity přepínače $r \times r$, kterých je m. Ve třetí úrovni je r přepínačů typu $m \times n$, takže síť má $n{\cdot}r$ výstupů. Z každého vstupu Closovy sítě existuje n různých cest ke každému výstupu.

Velmi malé použití těchto sítí je zapříčiněno jejich složitostí a vysokou cenou. Využití mají například jako specializované jednotky pro třídění. Více detailních informací o Closově síti lze najít v [DT04, str. 116].

4.2.4 Stromové sítě

Stromové sítě jsou popsány grafem, pro který platí, že mezi libovolným párem uzlů existuje jen jedna cesta. Na obrázku 4.12 je zakreslen binární strom výšky 3. Uzly představují křížové přepínače a listy jsou vstupní a výstupní bloky sítě.



Obrázek 4.12: Propojovací síť úplný binární strom výšky 3

Pokud posíláme data z uzlu v_1 do uzlu v_2, data se posílají směrem nahoru, až dosáhnou ke kořenu nejmenšího podstromu obsahujícího oba uzly v_1 a v_2. Potom putují data směrem dolů k uzlu v_2. Každá zpráva z levé poloviny stromu do uzlu v pravé polovině musí proto projít přes kořen stromu. Pokud je více takových zpráv, kořen celého stromu se stává úzkým hrdlem.

Problém je řešitelný pomocí tzv. tlustého stromu. U tlustých stromů se počet linek zvětšuje směrem ke kořenu stromu tak, aby počet hran spojujících kořen podstromu s jeho rodičem byl roven počtu listů v podstromě. Tlustý strom lze např. sestavit pomocí obousměrné víceúrovňové propojovací sítě.

O využití topologie tlustého stromu (fat tree) jako velmi výkonné propojovací sítě multipočítače, který obsahuje tisíce výpočetních uzlů, je pojednáno v článku [AFLV08].

4.3 Shrnutí

Ideální propojovací síť je statické propojení podle řešené soustavy. V tomto případě není nutné žádné směrování zpráv či inicializace propojení. Je jasně dané, jaké uzly jsou mezi sebou propojeny a může probíhat jen vlastní datová komunikace. Tento způsob propojení ale není univerzální. Umožňuje řešit pouze jednu soustavu rovnic a je tedy použitelný pouze jako specializovaný systém pro jeden konkrétní výpočet. Protože paralelní systém by měl být schopen řešit libovolnou soustavu diferenciálních rovnic, musí být zvolena nějaká univerzální propojovací síť.

V této fázi zatím nelze říci, kterou síť použít. To lze až po provedení dalšího rozboru: jaké komunikace budou vyžadovány a na jaké architektuře bude výsledný paralelní systém implementován.

Kapitola 5

Výpočetní operace

Pro numerické integrační metody probírané v této práci (viz kap. 2) je charakteristické, že stanovení nové hodnoty y_{i+1} je provedeno pomocí přírůstku vyjádřeného ve tvaru rekurentního vztahu (2.27).

Návrh mikroprocesoru jako jednoúčelové aritmeticko-logické jednotky (ALU) provádějící numerickou integraci je závislý na matematických operacích prováděných při vyčíslení tohoto rekurentního vztahu. Nejjednodušší situace z hlediska návrhu nastane při řešení homogenní diferenciální rovnice 1. řádu (viz ukázka řešení rovnice (2.13) resp. (2.22)).

Z příkladu řešení homogenní diferenciální rovnice 1. řádu vyplývají jednotlivé požadované operace mikroprocesoru, které jsou nezbytné pro vypočítání jednotlivých členů Taylorovy řady (2.27):

- sčítání - provádí kombinační nebo sériová sčítačka
- odčítání - převede se na sčítání změnou znaménka menšitele
- násobení - provádí kombinační násobička nebo sérioparalelní násobička (metodou dílčích součtů a posuvů na principu Boothova algoritmu)
- dělení - operace h/p lze předpřipravit

Detailnější rozbor několika obvodových realizací těchto operací bude představen dále v této kapitole.

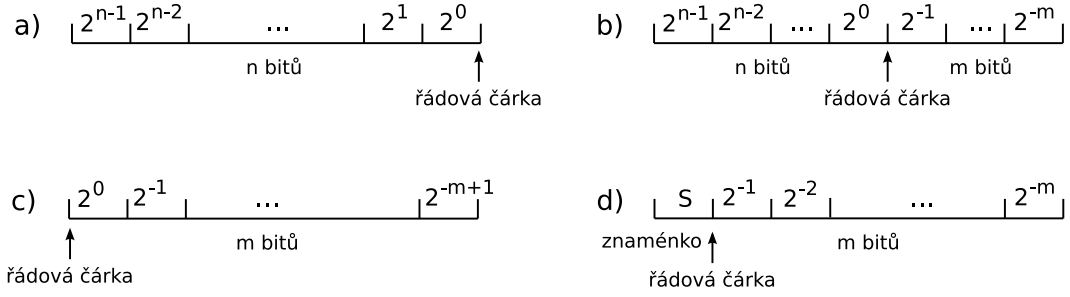
Ze zápisu soustavy rovnic s konstantními koeficienty (2.28)–(2.30), resp. ze zápisu řešení této soustavy (2.40)–(2.48) nebo (2.64)–(2.78) je vidět, že bude také potřeba sčítat několik operandů najednou (v příkladě pouze tři, ale obecně m). Proto bude potřeba i vícevstupá sčítačka, která může být řešena několika způsoby. Rozbor možných řešení je také obsahem této kapitoly.

5.1 Reprezentace dat

Abychom mohli nastínit matematické operace potřebné pro náš typ výpočtu, musíme si nejprve definovat formáty pro vnitřní reprezentaci dat používaných v počítačích. Existují dva základní typy: pevná a pohyblivá řádová čárka.

5.1.1 Čísla s pevnou řádovou čárkou

Používaných formátů čísel s pevnou řádovou čárkou je více. Ty nejrozšířenější jsou uvedeny na obrázku 5.1.



Obrázek 5.1: Formáty čísla s pevnou řádovou čárkou

Mezi používané kódy pro uvedené formáty patří přímý kód se znaménkem, inverzní kód, doplňkový kód a kód s lichým či sudým posunutím.

Pro operace v pevné řádové čárce je výhodné použít doplňkový kód, budeme proto v dalším textu uvažovat právě tento typ kódu. Výhoda použití doplňkového kódu spočívá zejména v tom, že se sčítačka pro tento kód nijak neliší od klasické sčítačky pracující s přímým kódem.

5.1.2 Čísla s pohyblivou řádovou čárkou

Čísla s pohyblivou řádovou čárkou (FP) mají tvar $M * Z^E$. Pro přesnou specifikaci je zapotřebí zadat základ Z, dále počet bitů, kód a tvar mantisy M a počet bitů a kód exponentu E.

Hodnota základu Z určuje rozsah zobrazitelných čísel. Rostoucí Z tento rozsah zvětšuje, ale zobrazení prostoru čísel je řidší, tedy přesnost zobrazení klesá. Zápis jednoho čísla je nejednoznačný, protože $1,0 * 10^{18} = 0,1 * 10^{19} = \cdots$. Proto se definuje normalizovaný tvar mantisy. např. $1/Z \leq |M| < 1$. Převod do tohoto tvaru se nazývá normalizace.

Rovněž vyjádření nuly je problematické. Je-li mantisa $M = 0$, pak exponent E může mít libovolnou hodnotu. Má-li však být číslo umístěno co nejblíže k nule, musí být E veliké záporné číslo. Exponent se vyjadřuje v posunutém kódu, protože velikost exponentů se pak snadno porovnává pomocí obyčejných komparátorů. Posunutí bývá liché i sudé.

Jeden z nejrozšířenějších formátů zobrazení čísel s pohyblivou řádovou čárkou je standard IEEE 754.

Standard IEEE 754

Standard IEEE 754 definuje Z=2 a jednoduchou a dvojitou přesnost. Při jednoduché přesnosti jsou čísla uložena na 32 bitech, při dvojité na 64 bitech. Kromě definice Z, M, E definuje standard další vyjimečné situace. V první řadě je to nečíselný výsledek, označený zkratkou NaN - Not a Number. Dále pamatuje na definici nekonečna, které vznikne podílem $x/0$.

Formát čísla FP podle standartu IEEE 754 v jednoduché přesnosti (32 bitů):

S	E	E	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M	M
31	30						23	22																				0

Nejvyšší významový bit obsahuje znaménko (S=0 - kladné, S=1 - záporné). Následuje 8 bitů exponentu a 23 bitů mantisy.

Uložené číslo potom získáme podle vzorce:

$$X = (-1) * S * 2^{E-BIAS} * (1, M)$$

Mantisa je vyjádřena přímým kódem bez znaménka, exponent kódem s lichým posunutím. Je důležité si uvědomit, že v poli exponentu je uvedeno číslo zvětšené o hodnotu BIAS (127). Zlomková část M udává číslo menší než 1. Mantisu získáme součtem 1 + M, což můžeme zapsat 1,M.

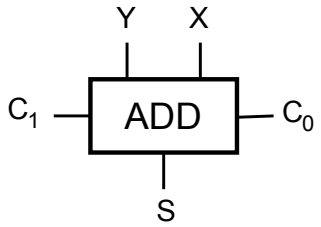
5.2 Sčítání

Operace sčítání je využívána jak složitějšími obvody (například každým čítačem), tak i při provádění operace odečítání, násobení i dělení. N-bitová sčítačka se tak stává klíčovým prvkem a na její rychlosti a dalších vlastnostech závisí parametry celého výpočetního systému.

Sčítačky se dělí na sériové (jednobitové) a paralelní. Paralelní sčítačky bývají většinou složeny z jednobitových. Základní problém při této konstrukci ale nastane s rychlostí šíření vzniklého přenosu mezi jednotlivými sčítačkami. Jak docílit co nejrychlejšího šíření přenosu je vysvětleno dále.

5.2.1 Sériová sčítačka

Základním stavebním prvkem pro operaci sčítání je úplná jednobitová sčítačka, která je zobrazena na obrázku 5.2.

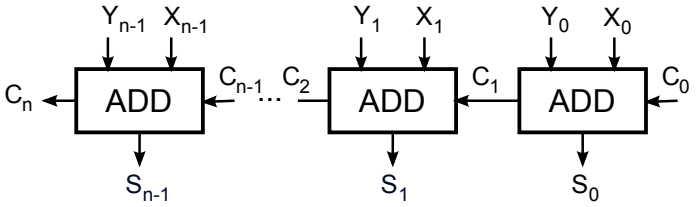


Obrázek 5.2: Úplná jednobitová sčítačka

Pokud chceme provádět sčítání n-bitových čísel, přivádíme postupně na vstupy X a Y oba sčítance od nejnižšího bitu a na výstupu S postupně získáváme výsledek součtu. C_1 je výstup obsahující přenos, který vzniká při sčítání a C_0 je vstup pro přenos z předchozího řádu. Při sériovém sčítání se mezi ně zapojí nějaký paměťový prvek (např. klopný obvod D), který slouží k uchování přenosu.

5.2.2 Paralelní sčítačka s postupným přenosem

Když zapojíme několik jednobitových sčítaček za sebe, dostaneme nejjednodušší paralelní sčítačku - s postupným šířením přenosu. Tato sčítačka je zobrazena na obrázku 5.3.

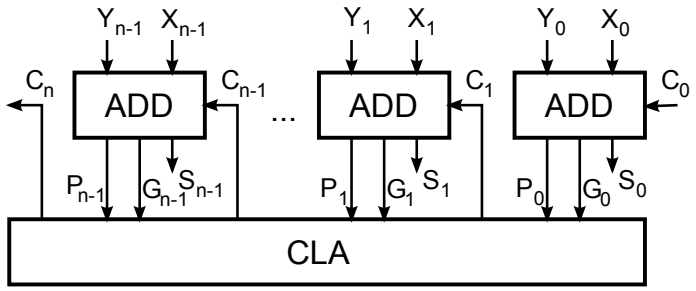


Obrázek 5.3: Paralelní sčítačka s postupným přenosem

Přenos se postupně šíří z nejnižší sčítačky do nejvyšší. U první sčítačky se objeví výstupní přenos C_1 za dobu $2T$. Čas výpočtu celého přenosu C_n je roven $n{\cdot}2T$. To je ale příliš dlouho. Proto byly vyvinuty sčítačky s rychlým šířením přenosu.

5.2.3 Paralelní sčítačka s rychlým přenosem

Ke sčítačkám je zapojen pomocný obvod CLA (Carry Look - Ahead). Ten slouží ke zrychlení šíření přenosu. Zapojení je zobrazeno na obrázku 5.4.



Obrázek 5.4: Paralelní sčítačka s rychlým přenosem

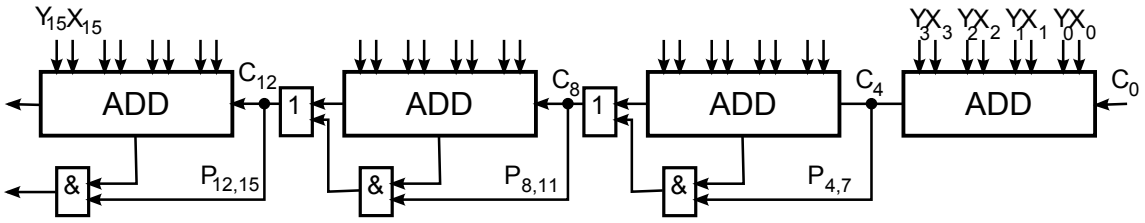
Návrh je založen na definici dvou pomocných funkcí sčítačky, označené P_i a G_i. P_i je funkce popisující kdy přenos C_i projde sčítačkou ze vstupu přenosu na výstup přenosu C_{i+1} (šířený přenos - Propagate). Funkce G_i určuje, kdy vznikne přenos v odpovídající sčítačce (generovaný přenos - Generate). Pravdivostní tabulka a zápis rovnic pro nově upravenou jednobitovou sčítačku a obvod CLA je v [Drá95, str. 51].

Funkce P_i a G_i se tvoří se zpožděním T, v čase $3T$ jsou k dispozici všechny rychlé přenosy, a součet je tedy vytvořen v čase $4T$. Tento typ sčítačky s rychlým šířením přenosu je nejrychlejší možná varianta paralelní sčítačky. Složitost obvodu pro generování přenosu CLA však roste s rostoucí šířkou sčítačky druhou mocninou šířky. Byla proto navržena další řešení, která tento nedostatek odstraňují.

Jedno takové řešení je stromový generátor přenosu. Je založen na několikanásobném použití generátoru CLA menší šířky, např. pro 2 nebo 4 bity, které jsou zapojeny do stromové struktury. Více v [Drá95, str. 52].

5.2.4 Paralelní sčítačka s přeskakováním přenosu

Na obrázku 5.5 je znázorněna paralelní 16-bitová sčítačka s přeskakováním přenosu.



Obrázek 5.5: Paralelní sčítačka s přeskakováním přenosu

Tvoří ji 4-bitové sčítačky s postupným přenosem - ADD (viz podkapitola 5.2.2), které realizují i globální funkci $PropagateP$. Postupný přenos se začne šířit současně ve všech blocích ADD. Pokud některý blok vytvoří přenos, je tato složka přenosu platná, i když vstupní přenos ještě nemá správnou hodnotu. Výstup přenosu z každého bloku tedy můžeme pokládat za přenos G. Jak je vidět na obrázku, výstupní přenos každého bloku se přivádí na vstup přenosu následujícího bloku a na hradlo otevírané signálem P následujícího bloku.

Při použití k-bitových sčítaček do celkové šířky n bitů je nejdelší doba průchodu přenosu úměrná výrazu $n/k - 2 + 2k$. Průchod postupného přenosu prvním a posledním blokem trvá $2k$, vnitřních bloků, které mají zpoždění 1 je tedy $n/k - 2$. Toto uspořádání jde dále optimalizovat s cílem dosáhnout ve všech blocích stejné zpoždění.

5.2.5 Sčítání čísel v pohyblivé řádové čárce

Všechny představené sčítačky provádějí sčítání čísel v pevné řádové čárce. Pokud použijeme formát uložení čísel v pohyblivé čárce, bude použit zásadně odlišný postup. I v praxi jsou oddělené obvody pro realizaci operací v pevné a pohyblivé řádové čárce.

Operace v aritmetice pohyblivé řádové čárky se rozpadají na operace s mantisami, operace s exponenty a normalizace.

Mějme 2 čísla X a Y:

$$X = M_X * Z^{E_X}$$

$$Y = M_Y * Z^{E_Y}$$

Na nich si ukážeme operaci sčítání. Výpočet součtu dvou čísel lze provést takto:

$$X + Y = (M_X * Z^{E_X - E_Y} + M_Y) * Z^{E_Y} = (M_X + M_Y * Z^{E_Y - E_X}) * Z^{E_X}$$

Získáme tedy dva předpisy pro výpočet součtu. V praxi je třeba uvážit, že mantisa je realizována na konečném počtu bitů. Pokud mají čísla různé exponenty, tak mají odpovídající si bity různé váhy. Je jasné, že sečítat se musí odpovídající si bity se stejnou váhou. Proto dojde k posuvu celé mantisy, aby tomu tak bylo. Nově vzniklé číslo bude tedy potřebovat v případě různých exponentů pro zachování všech platných číslic mnohem více bitů. Reprezentace čísla je však pevně daná. Je tedy třeba zaokrouhlovat. Nelze na začátku ořezat nejvyšší platné bity většího čísla, protože ty jsou pro výsledek určující, nýbrž se ořežou nejnižší platné bity menšího čísla.

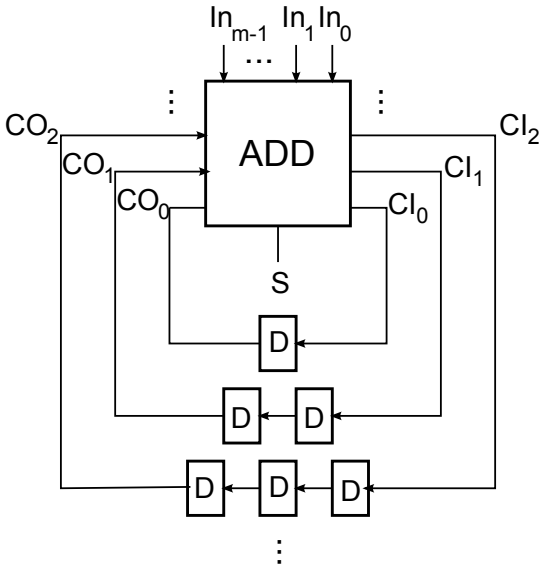
5.2.6 Víceoperandové sčítání

Jak již bylo zmíněno na začátku této kapitoly, bude také potřeba sčítat několik (obecně m) operandů najednou. Proto bude potřeba i vícevstupá sčítačka, která může být řešena několika způsoby. Tyto principy platí pro čísla uložená v pevné řádové čárce, nebo pro mantisy čísel v pohyblivé řádové čárce.

Při sčítání dvou čísel vzniká za určitých podmínek přenos do vyššího řádu. Totéž platí při sčítání více než dvou čísel. Počet řádů l, o které přeteče součet, je dán vztahem $l = \log_2 m$, kde m je počet sčítanců. Z toho plyne, že výsledný součet je zobrazen na $l + n$ bitech, kde n je počet bitů, na kterých jsou uloženy původní sčítance.

Sériová realizace

Popisovaný obvod je zobrazen na obrázku 5.6.



Obrázek 5.6: Princip víceoperandového sériového sčítání

Základ tvoří m-vstupá sériová sčítačka (vstupy In_0–In_{m-1}). Sčítačka musí být rozšířena o l vstupů, které se účastní přenosu z předcházejících výpočtů. Protože se jedná o sériový obvod, tak pouze jediný - nejméně významový bit z obvodu, tvoří výsledek. Ostatní výstupy se účastní přenosu, a to tak, aby byly použity v řádu, kterému odpovídají. To je realizováno zapojením odpovídajícího počtu záchytných registrů (např. klopných obvodů typu D) do smyčky každého přenosu.

Paralelní realizace

Všechny zde představované postupy provádějí postupnou redukci počtu sčítanců z m až na 2, které na konci sečteme obyčejnou paralelní dvouoperandovou sčítačkou a obdržíme konečný výsledek. Tyto přístupy se liší hlavně ve způsobu získání jednotlivých redukcí počtu sčítanců do konečného počtu dvou.

Základní princip provádění víceoperandového sčítání je založen na použití paralelních sčítaček zapojených do kaskády nebo do stromové struktury. V prvním stupni jsou sečteny 2 operandy (existují i varianty se třemi vstupy) a v každém následujícím stupni je k výsledku

předchozího sečtení přičten další operand. Tento postup platí pro sčítačky, které jsou propojeny do kaskády.

Další variantou jsou různé stromové struktury. Nejvyšší úroveň stromu obsahuje dvouoperandové sčítačky. Jejich počet je přímo úměrný počtu sčítaných operandů a tvoří listy stromu. V každé další úrovni je počet sčítaček menší a sčítá výsledky, které produkují sčítačky z vyšší úrovně. Až sčítačka v kořeni stromu dává výsledek celého součtu. Další možností jsou různé kombinace obou předcházejících metod. Bližší popis různých typů realizací víceoperandového sčítání lze nalézt v [Par09, Chapter 8: Multi-Operand Addition].

Sčítačka založená na BCD kódu

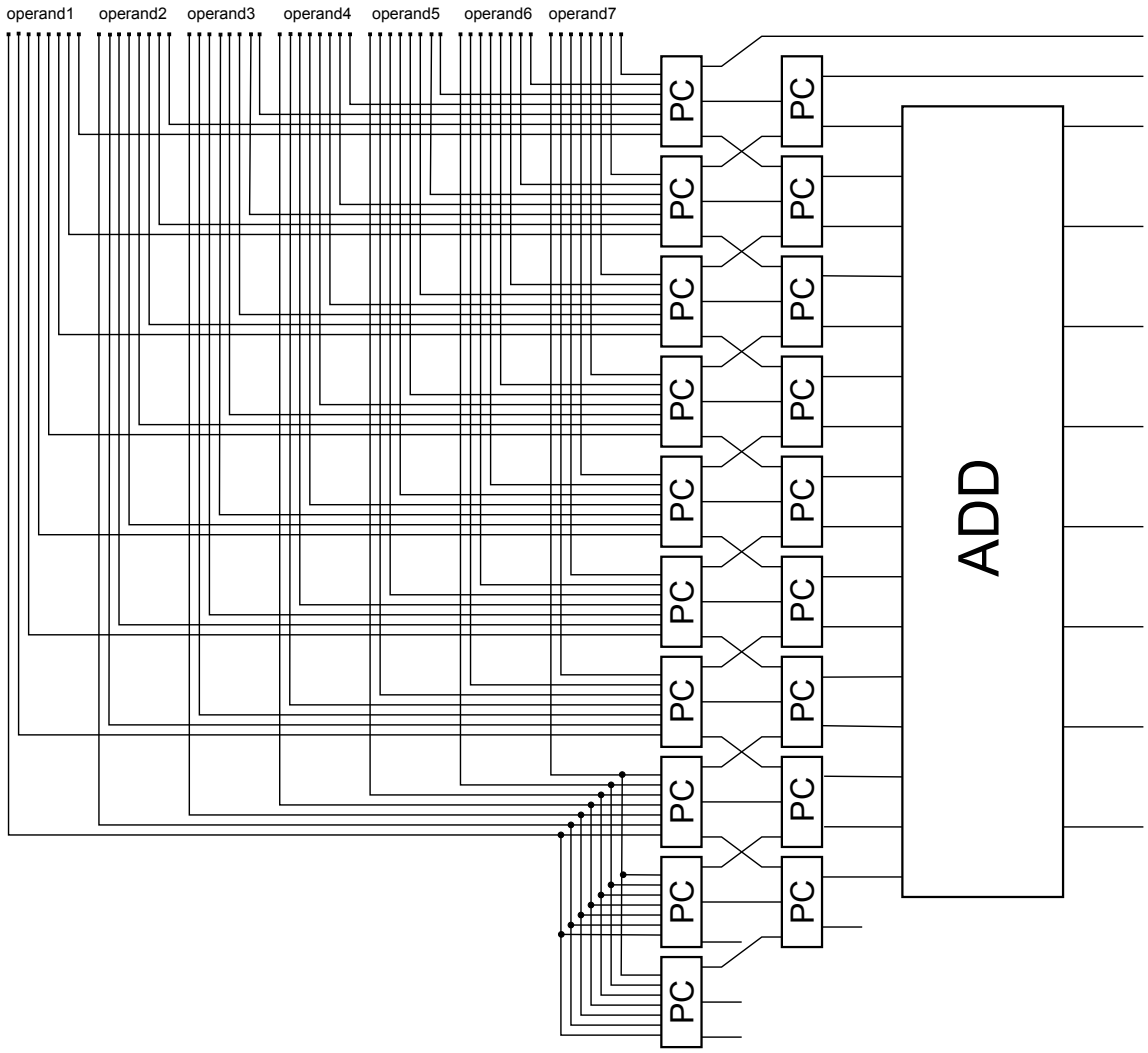
V obvodu na obrázku 5.7 realizující zde popisovaný typ víceoperandového sčítání je použit speciální kombinační obvod PC (angl. parallel counters [Swa73]), který sečte počet jedniček jednotlivých pozic binárních čísel a vyjádří tento součet v kódu BCD. Bity tohoto součtu se přivedou do dalších kombinačních obvodů, které je zpracují stejným způsobem. Až obdržíme na výstupu pouze dvoubitové BCD kódy počtů jedniček, použijeme k jejich sečtení dvouoperandovou sčítačku.

Pro správnou funkci sčítačky musíme rozšířit všechny sčítance o l bitů do nejvyšších řádů. Tyto bity musí mít stejnou logickou hodnotu, jako bit určující znaménko každého ze sčítanců. Sčítáním záporných čísel vznikají i přenosy do vyšších řádů než $n + l$, ale ty se mohou ignorovat, protože jsou pro výsledek bezvýznamné. Princip je ukázán na následujícím příkladě sečtení sedmi 8-bitových čísel.

1 1 1 0 0 0 0 0 0 0	-128	operand 1
0 0 0 1 1 1 1 1 1 1	+127	operand 2
1 1 1 1 1 1 1 1 1 1	-1	operand 3
0 0 0 1 1 1 1 1 1 1	+127	operand 4
0 0 0 1 0 0 0 0 0 0	+64	operand 5
0 0 0 1 0 1 0 1 0 1	+85	operand 6
0 0 0 0 1 0 1 0 1 0	+42	operand 7
1 0 0		
1 0 0		
1 0 0		
1 0 0		
1 0 0		
1 0 0		
1 0 1		
0 1 0		
0 1 0		
0 1 0		
0 0		
0 0		
0 1		
0 1		
0 1		
0 1		
1 0		
0 1		
1 0		
0 1		
0 1		
1 0		
0 1 0 0 1 0 0 1 1 1 1 0 0	+316	výsledek

V příkladě je vodorovnou čárou zobrazeno umístění kombinačních obvodů a dvojitou vodorovnou čárou je zobrazeno umístění sčítačky. Ve výsledku se nejvyšší bity 0,1,0 ignorují, další bit dává znaménko a následuje 9 bitů určující hodnotu celého čísla.

Každý sloupec čísel obsluhuje jeden kombinační obvod a v každé úrovni jsou tyto obvody různé složitosti. Obvodová složitost je dána počtem jeho vstupů. Schématické zapojení celého obvodu, které provádí sčítání 7 operandů je na obrázku 5.7.



Obrázek 5.7: Schéma obvodu víceoperandového sčítání

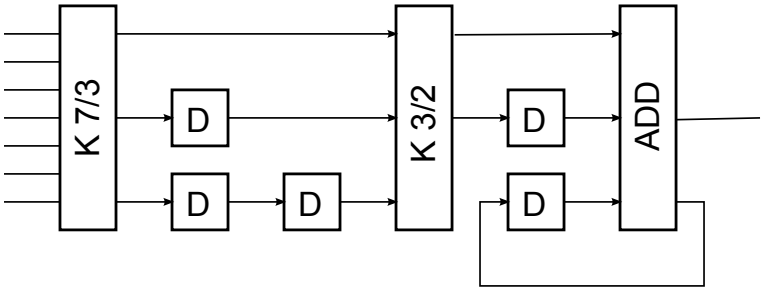
V obrázku je vidět zjednodušení obvodu, kdy nejsou zapojeny bezvýznamné bity v nejvyšších řádech (ve schématu dole) a tím jsou vypuštěny zbytečné kombinační obvody. V pravé části schématu je vidět klasická paralelní dvouoperandová 8-bitová sčítačka, jejímiž vstupy jsou výsledky z posledního sloupce kombinačních obvodů.

Při analýze celého obvodu je vidět, že cesta pro zpracování nejnižších bitů sčítanců je stejná s cestami zpracování všech bitů ostatních. Tato cesta obsahuje několik kombinačních obvodů a část dvouoperandové sčítačky. Konstrukcí jediné této cesty se získá proudově pracující obvod.

Princip proudového zpracování je v rozdělení funkčního bloku daného systému na několik částí. Za jednotlivými částmi jsou zařazeny záchytné registry, které slouží k uchování jejich

výstupních hodnot. Celkový výpočet probíhá tak, že v prvním kroku přivedeme na vstup nejnižší významový bit všech sčítanců, v druhém kroku další bit v pořadí atd. V posledním kroku přivedeme nejvyšší bit rozšířených sčítanců.

Na obrázku 5.8 je vidět proudově pracující obvod pro 7 sčítanců. Jednotlivé kombinační obvody jsou označeny písmenem K s číselnými indexy, kde první hodnota udává počet vstupů a druhá počet výstupů (např. K 7/3). Za každým kombinačním blokem jsou zařazeny záchytné registry D.



Obrázek 5.8: Proudově pracující obvod víceoperandového sčítání pro 7 sčítanců

Různost kombinačních obvodů a tím i jejich hw mohutnost závisí pouze na počtu jejich vstupů. Počet výstupů je dán počtem bitů, na kterých lze vyjádřit maximální počet vstupních jedniček. Např. pro sečtení 16 logických jedniček potřebujeme 5 bitů. Zde je ale rezerva v možnosti zápisu čísla zobrazeného na pěti bitech. Pro 5-bitový výstup tak může být zkonstruován kombinační obvod s 31 vstupy.

Pokud budeme chtít sčítat více než 31 operandů, musíme předřadit další kombinační obvod, který realizuje požadovaných m vstupů do 31 výstupů. Pokud vycházíme z toho, že poslední kombinační obvod před sčítačkou musí mít 2 výstupy, optimální počet vstupů a výstupů jednotlivých kombinačních obvodů je: 5 (3 vstupy/2 výstupy), 10 (7 vstupů/3 výstupy), 134 (127 vstupů/7 výstupů) atd. První (vstupní) obvod pak obsahuje takové množství vstupů, které odpovídá počtu operandů, které mají být sečteny.

Hw realizace kombinačního obvodu závisí na používané technologii. Jednou z možností je použít vyhledávací tabulku, kde by odpovídající kombinace byly uloženy např. v paměti typu ROM. Vstupní kombinace by sloužily jako adresy do paměti, na kterých budou uloženy BCD reprezentace počtu jedniček v daném vektoru. Další možnost je použít minimalizaci pomocí Karnaughových map. Z pravdivostní tabulky pro každý kombinační obvod lze sestavit Karnaughovu mapu a z následných rovnic vytvořit odpovídající síť pomocí hradel. Detailní popis návrhu kombinačních obvodů pomocí pravdivostních tabulek a následnou minimalizací lze nalézt v [PP06].

5.3 Odčítání

Odčítání se nejčastěji realizuje pomocí přičítání čísla s opačným znaménkem. Obvod sčítačky se tedy doplní pouze obvody, které vytvoří doplňkový kód jednoho z operandů a ten se potom přivede na vstup sčítačky. Doplněk se vytvoří negací všech bitů daného operandu a následným přičtením jedničky k nejméně významovému bitu.

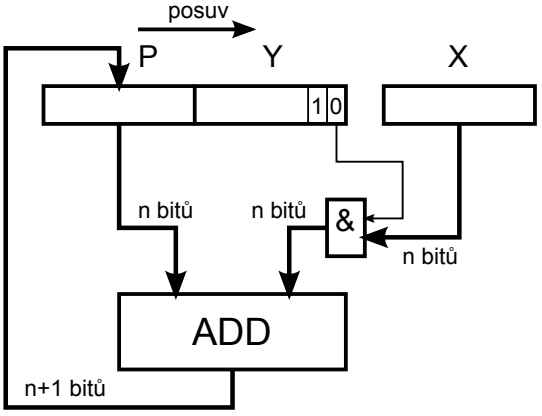
5.4 Násobení

Jak je vidět z rekurentního vztahu (2.27), základní operace prováděná při numerické integraci Taylorovou řadou je operace násobení. Na výběr vhodné násobičky musí být proto kladen největší důraz.

Mohou se násobit absolutní hodnoty čísel a pak doplněno znaménko, nebo lze násobit přímo čísla se znaménkem. Operace násobení může být prováděna sériově nebo paralelně v jednom kroku.

5.4.1 Sériová násobička

Obvod, který provádí sériové násobení je na obrázku 5.9.



Obrázek 5.9: Princip sériového násobení

Do dolní poloviny registru PY se uloží násobitel Y, do horní poloviny nuly, do registru X se uloží násobenec. Nejnižším bitem registru PY se vynásobí násobenec X a tento výsledek se přičte k horní části registru PY. V něm se ukládají průběžné výsledky násobení (součet dílčích součinů). Dále se provede posuv registru PY o jeden bit vpravo. To se provede n-krát až se zpracuje všech n bitů násobitele Y.

Pokud používáme čísla se znaménkem je tento postup násobení nepraktický, protože musíme důsledně provádět práci se znaménkem (šíření znaménka). Viz příklad [Drá95, str. 57]. Proto bylo vytvořeno několik dalších algoritmů, které provádějí násobení čísel se znaménkem. Nejznámější a nejpoužívanější z nich je Boothův algoritmus násobení.

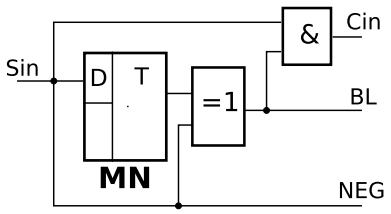
5.4.2 Sériová násobička - Boothův algoritmus

Boothův algoritmus násobení je založen na dílčích součtech (rozdílech) a posuvech. Při realizaci násobení tímto algoritmem je prováděno zpracování násobitele vyhodnocením logických hodnot dvou sousedních bitů (právě zpracovávaného a předchozího) podle tabulky 5.1. Tato varianta se označuje jako *Boothovo překódování s radixem 2.*

Pro řízení násobení pomocí tohoto algoritmu slouží obvod, který je zobrazen na obrázku 5.10. Signál *NEG* společně se signálem *Cin* (log. hodnota "1") slouží k vytvoření záporného násobence (odečtení násobence od mezivýsledku). Signál *BL* (log. hodnota "0") slouží k zablokování přičtení násobence k mezivýsledku (při stejné kombinaci vstupních bitů) - vynuluje násobence (přičtení nul).

Tabulka 5.1: Princip Boothova algoritmu - překódování s radixem 2

log. hodnota bitů násobitele		
b_i bit	b_{i-1} bit	odpovídající akce
0	0	přičtení nul k mezivýsledku
0	1	přičtení násobence k mezivýsledku
1	0	odečtení násobence od mezivýsledku
1	1	přičtení nul k mezivýsledku



Obrázek 5.10: Obvod generující řídící signály pro Boothův algoritmus

Tyto dva bity testujeme pouze za pomocí jednoho vodiče. A to tak, že na vstupu máme aktuální bit *Sin* a předchozí je uložen v paměti (klopném obvodu např. typu D), který je označen MN. Po provedení přičtení (získání mezivýsledku) a dříve než získáme na vstupu další bit v pořadí je nutné uložit aktuální bit, aby ho bylo možné použít jako předchozí v dalším mezivýpočtu.

K výpočtu je potřeba kladný i záporný tvar násobence. Podle aktuálně zpracovávaných bitů násobitele algoritmus rozhodne, který tvar (nebo nulu) přičte k mezivýsledku.

Z jednoduchého Boothova překódování (s radixem 2) je odvozeno překódování "2 bity najednou", neboli *Boothovo překódování s radixem 4*. Lze odvodit i Boothovo překódování s radixem 8, 16. Tyto varianty používají menší počet mezivýsledků (dílčích součinů), takže výpočet probíhá rychleji. Více informací o těchto variantách Boothova algoritmu násobení je v [Drá95].

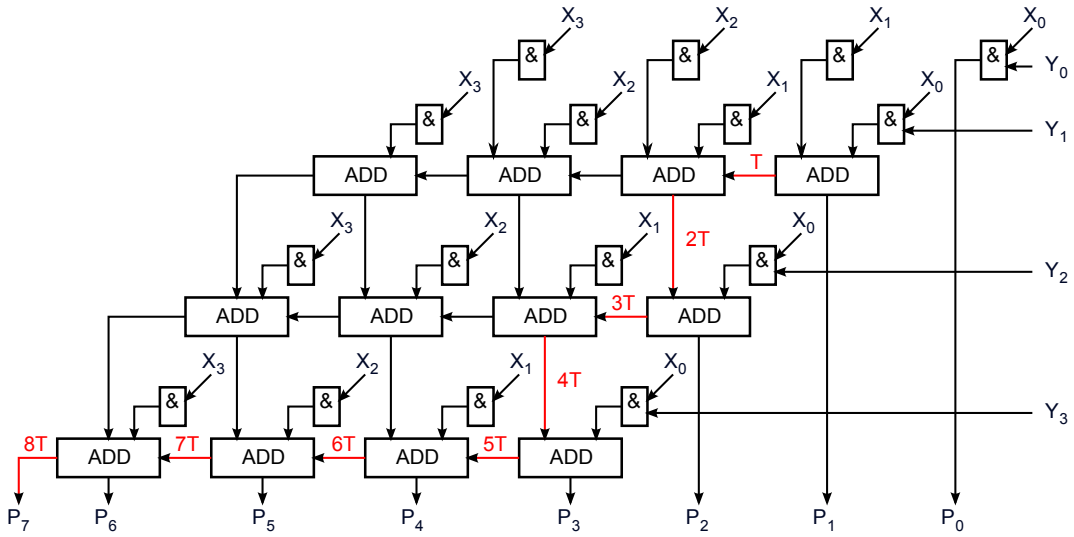
Řešení, které odstraňuje nevýhodu nízké rychlosti výpočtu ale za cenu zvětšení plochy obvodu a zpoždění logiky, představují paralelní násobičky.

5.4.3 Paralelní násobička

Paralelní násobičku si je možno představit v nejjednodušším případě jako sériovou násobičku "rozbalenou do prostoru". 4-bitová násobička je schematicky zobrazena na obrázku 5.11.

Ve schématu je zdůrazněná kritická cesta, která nám zobrazuje šíření přenosu přes celou násobičku a určuje zpoždění celého obvodu. Zpoždění obvodu se zvyšuje s rostoucí délkou operandů.

Rychlost kombinační násobičky můžeme zvýšit přidáním registru pro uchování přenosů. Po této změně mohou být sčítačky s postupným přenosem odstraněny a nahrazeny sčítačkami s uchováním přenosu. Tím dosáhneme zrychlení, protože přenos se už nebude šířit klasicky po řádcích, ale bude přeskakovat z jednoho řádku do následujícího. Navíc se mohou sčítačky pro nejvyšší řád nahradit jen hradlem. Nakonec se ale musí přidat řada sčítaček s postupným přenosem, kde dojde ke kompletaci výsledku. Toto uspořádání je rychlejší,



Obrázek 5.11: Symbolické schéma paralelní násobičky

než paralelní násobička využívající sčítačky s postupným přenosem a nazývá se Wallaceův strom. Ten slouží jako základ pro další možná vylepšení paralelní násobičky. Více informací o těchto vylepšeních je v [HP06, Appendix I: Computer Arithmetic] a [Drá95].

5.4.4 Násobení čísel v pohyblivé řádové čárce

Násobičky představené výše slouží k násobení čísel v pevné řádové čárce. Pokud použijeme formát uložení čísel v pohyblivé čárce, musíme použít odlišný postup.

Máme opět dvě čísla X a Y:

$$X = M_X * Z^{E_X}$$

$$Y = M_Y * Z^{E_Y}$$

Operaci násobení lze zapsat takto:

$$X * Y = (M_X * M_Y) * Z^{E_X + E_Y}$$

Při této operaci se provádí vynásobení mantis a sečtení exponentů. Po provedení násobení musí být provedena normalizace výsledku.

Existuje několik algoritmů násobení, z nichž některé řeší i problém násobení záporných čísel v doplňkovém kódu.

5.5 Dělení

Z rozboru použití Taylorovy řady (rovnice (2.23)–(2.27)) vyplývá, že dělení se používá pouze k vyčíslení podílů integračního kroku h/p při výpočtu jednotlivých členů Taylorovy řady. Tato hodnota je stejná pro všechny řešené rovnice.

Jedná se o dělení dvou konstant, protože v době spuštění výpočtu víme, jaký bude použit integrační krok h. Proto si tyto hodnoty můžeme předpočítat dopředu mimo celý paralelní systém např. na počítači, který bude celý systém ovládat (inicializovat výpočet a očekávat výsledky k dalšímu zpracování). Následně řadič zajistí dodání jednotlivých podílů všem výpočetním jednotkám.

Hardwarová realizace děličky přímo v navrhovaném paralelním systému by byla zbytečně složitá, pomalá a zabírala by zbytečné místo, které může být využito účinněji.

5.6 Porovnání sčítaček a násobiček

Výkonové zhodnocení zde představených typů sčítaček a násobiček pro operace v pevné řádové čárce je obsahem článků [Bt05] a [Ma09].

Článek [Bt05] se zabývá porovnáním sčítaček a násobiček, jejichž rozbor jsem provedl v této části práce. Tyto jednotky byly implementovány do hradlových polí FPGA (Field Programmable Gate Array). Obecně by rychlost výpočtu měla být vysoká a plocha obsazená výpočetní jednotkou na čipu by měla být co nejmenší. To jsou dva nejdůležitější požadavky, které ale stojí proti sobě.

Ze sčítaček uvedených v předchozích částech byla po implemetaci v jazyce VHDL nejrychlejší sčítačka s přeskakováním přenosu a zabírala také nejméně místa na čipu. Podle měření, která jsou popsána v článku, dosahuje ale nejlepších výsledků sčítačka, která byla vygenerována nástrojem pro syntézu obvodů do FPGA po použití operátoru + (použití konstrukce typu `S = X + Y`). Převážná většina dnešních obvodů FPGA tento problém řeší pomocí dedikovaného kanálu pro akceleraci přenosu (tzv. carry chain). Není proto výhodné přesně implementovat konkrétní sčítačku, ale nechat výsledný návrh na automatizovaném prostředí, které naplno využije možností použitého čipu.

Jak již bylo zmíněno výše, násobičky mohou být sériové a paralelní. Obecně je zřejmé, a dokazují to i měření, že nesrovnatelně menší plochu čipu zabírají sériové násobičky. Ty také dosahují menšího zpoždění T_S než paralelní násobičky T_P. Celkový výpočet ale trvá n kroků. Čas, po kterém obdržíme výsledek násobení je tedy $t_S = n \cdot T_S$, kdežto u paralelní násobičky je přímo roven $t_P = T_P$.

Při implementaci paralelních násobiček pracujících s vícebitovými čísly naráží implementace na limity, které jsou dány použitou technologií (FPGA a nástrojů pro syntézu). Podstatně lepších výsledků dosáhneme, když použijeme násobičky, které jsou integrované už v čipu FPGA. Výsledná implementace násobičky velmi závisí na typu použitého obvodu FPGA. Při použití starších či levnějších obvodů FPGA je vygenerovaná struktura násobičky složená z diskrétních logických prvků (např. LUT). Taková násobička bude patrně zabírat nezanedbatelnou část plochy obvodu FPGA. Naproti tomu moderní programovatelná hradlová pole mají integrované hardwarové násobičky a nástroje pro syntézu, rozmístění a propojení je dokáží využít. Výhodami integrovaných násobiček je nižší zpoždění a menší zabraná plocha obvodu FPGA.

Výrazného zrychlení klasických paralelních násobiček uvedených výše dosáhneme použitím pipeline technik. Obsazená plocha je však stále velmi vysoká v porovnání s použitím integrovaných násobiček.

Článek [Ma09] se zaměřuje na srovnání sériově prováděných operací sčítání a násobení s jejich paralelními verzemi. Výsledky ukazují, že sériové verze sčítaček a násobiček jsou stále používané při práci s FPGA architekturami, které jsou orientované na vysoce výkonné paralelní zpracování.

Následující zajímavé vlastnosti má sériová aritmetika:

1. Zabírá výrazně menší plochu čipu.

2. Sériově prováděné výpočetní operace zpracovávají data bit po bitu a komunikační cesty jsou většinou výrazně kratší než u paralelních verzí. Kratší potom může být i minimální perioda systémových hodin (zpoždění členu), což částečně kompenzuje zpomalení způsobené sériovým zpracováním.

3. Sériový aritmetický obvod může dosáhnout vyšší propustnosti než paralelní a zároveň zabírá méně místa na čipu. Potom je možné dosáhnout i příznivější hodnoty součinu *čas zpracování* $\times$ *plocha na čipu*.

Paralelní aritmetika je mnohem efektivnější při malých datových šířkách. S rostoucí datovou šířkou se rozdíly mezi sériovou a paralelní verzí podstatně zmenšují. Pro značně zaplněný FPGA čip bude sériová aritmetika výhodnější než paralelní i pro menší šířky dat, protože se zvětšuje zpoždění na složitějších spojích v obvodu. Výkonnost paralelní aplikace je také omezena rychlostí paralelních datových cest propojujících jednotlivé výkonné jednotky. Sériové propojení dosahuje obecně vyšší propustnosti než paralelní.

5.7 Shrnutí

Z provedeného rozboru je zřejmé, že výpočetní jednotky musí umět provádět hlavně dvě výpočetní operace sčítání a násobení. Ty je možné provádět v pevné a pohyblivé řádové čárce. Zaměříme se na pevnou řádovou čárku, protože výpočty v pohyblivé řádové čárce se rozpadají na operace s mantisami a exponenty, které vlastně představují čísla v pevné řádové čárce.

Přesný způsob, jakým bude provedeno sčítání a hlavně násobení závisí na použité architektuře. Předpokládá se využití hradlových polí FPGA. Nejefektivnější sčítačka je paralelní, která vznikne vygenerováním pomocí syntezátoru. Sériová sčítačka najde uplatnění jen v případě, kdy máme požadavky na minimální obsazenost čipu a rychlost výpočtu není klíčová.

U násobiček už toto rozhodnutí není tak jasné. Paralelní násobička je využitelná pouze v případě, kdy použijeme FPGA čip s integrovanými násobičkami, případně při nutnosti velmi rychlého výpočtu. Sériová násobička najde větší uplatnění hlavně při použití většího počtu výpočetních jednotek. Jednotky mohou být v tomto případě propojeny sériově a tím se podstatně zjednoduší a urychlí propojovací síť.

Kapitola 6

Návrh paralelního systému

Z předchozího rozboru, který je pro názornost proveden pro homogenní soustavu diferenciálních rovnic, vyplývá, že v závislosti na typu řešených úloh musí být při návrhu paralelního systému kladen důraz na výběr vhodné architektury procesoru (aritmeticko-logické jednotky). Dále musí být použit vhodný typ propojovací sítě, případně kombinace více různých sítí. Největší problémy nastávají při návrhu velmi rozsáhlých paralelních systémů, kde musí být efektivně propojeny stovky až tisíce výpočetních uzlů. V neposlední řadě je také důležité řízení celého výpočetního a propojovacího systému.

Použité výpočetní bloky a jejich propojení vychází ze schémat, která byla použita v kapitole 2. Tzn. výpočetní jednotky provádí operace, které odpovídají integrátorům, sčítačkám případně odčítačkám, násobičkám a děličkám. Není problém navrhnout sčítačku případně násobičku. Otázkou ale je návrh integrátoru. Navržené výpočetní, propojovací a řídící podsystémy jsou obsahem této kapitoly. Na závěr je popis propojení těchto podsystémů do jednoho celku - paralelního systému.

6.1 Koncepce aritmeticko-logické jednotky

Základ celého systému tvoří aritmeticko-logické jednotky (ALU). ALU provádí vlastní výpočet numerické integrace. Aritmeticko-logická jednotka tedy představuje integrátor. Dá se očekávat, že stovky až tisíce aritmeticko-logických jednotek mohou být propojeny a může být řešena velmi rozsáhlá soustava diferenciálních rovnic. ALU je proto koncipována jako specializovaná jednoúčelová jednotka, provádějící základní matematické operace sčítání a násobení.

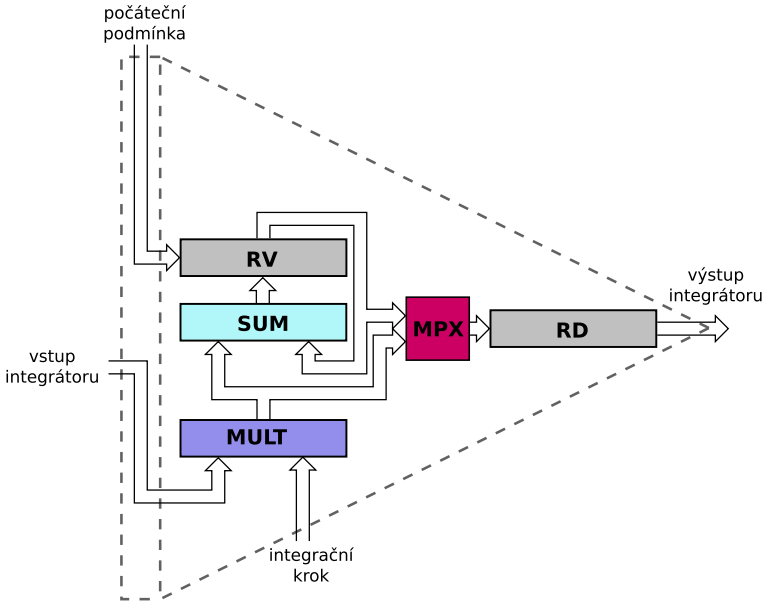
Jak je popsáno v kapitole 2, nebo přímo v části 2.2 popisující metodu Taylorovy řady, zadaný problém je převeden na soustavu homogenních lineárních diferenciálních rovnic s konstantními koeficienty. Při numerickém výpočtu těchto diferenciálních rovnic Taylorovou řadou se používají dvě základní operace: **násobení** (výpočet DYp_i) a **sčítání** (iterativní výpočet y_{i+1}).

Tyto dvě základní operace mohou být prováděny sériově nebo paralelně. Podobně komunikace mezi procesory se může řešit sériově nebo paralelně. Podle toho budeme v následujícím textu rozdělovat integrátory do těchto skupin:

- paralelně-paralelní integrátory (paralelní komunikace a paralelní výpočet)
- sériově-paralelní integrátory (sériová komunikace a paralelní výpočet)
- sériově-sériové integrátory (sériová komunikace a sériový výpočet)

6.1.1 Paralelně-paralelní integrátor

Násobení je realizováno paralelní násobičkou a sčítání paralelní sčítačkou. Blokové schéma je zobrazeno na obrázku 6.1. Čárkovaně je vyznačen symbol integrátoru.



Obrázek 6.1: Blokové schéma paralelního integrátoru

Význam jednotlivých bloků:
RV registr výsledku
RD registr součinu
MPX multiplexor
SUM paralelní sčítačka
MULT paralelní násobička

Funkce integrátoru: cyklus je zahájen tím, že se do registrů RD a RV uloží hodnota y_i. Na vstupu se objeví hodnota $f(y_i)$, což je výstupní hodnota prvku, který je připojen na vstup integrátoru. Integrační krok se nataví na velikost h. Součin (z násobičky MULT) těchto dvou vstupů se přepíše do registru RD a současně se přičte k registru RV. V RD je tedy hodnota $DY1$ a v RV je mezisoučet $y_i + DY1$. V dalším kroku se na vstupu objeví $f(DY1)$ a integrační krok $h/2$. Jejich roznásobením se vypočítá $DY2$, jež se opět uloží do RD sečte s RV. Celý cyklus se opakuje do té doby, dokud není dosaženo požadované přesnosti, nebo maximálního počtu iterací, čímž získáme y_{i+1}.

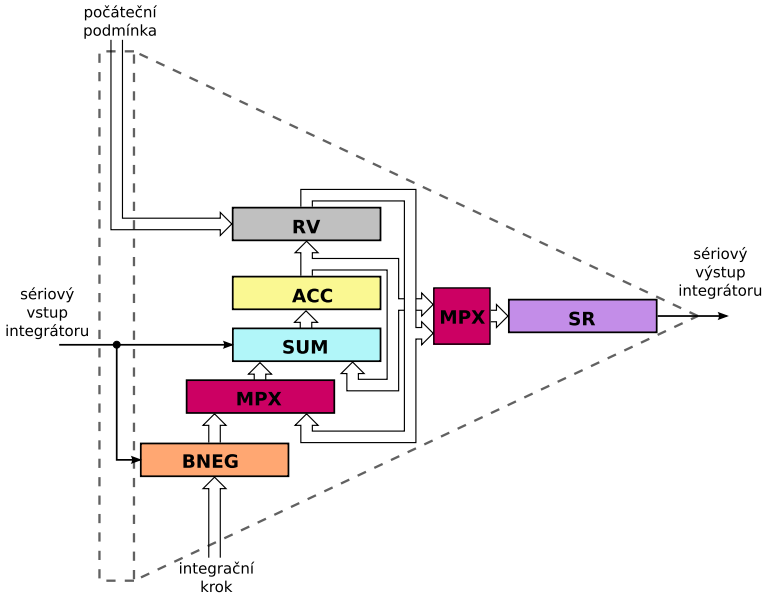
Tento typ integrátoru je nejrychlejší, čas výpočtu jednoho členu Taylorovy řady je:

$$t_{PP} = \tau_{nas} + \tau_{sec} + \tau_{sit} \tag{6.1}$$

tedy je dán součtem času násobení τ_{nas} a sčítání τ_{sec}, případně i zpožděním signálů v propojovací síti τ_{sit}. Ovšem za cenu největší složitosti zapojení, na kterém má největší podíl kombinační násobička. Dalším nepříznivým kritériem je počet vstupů a výstupů, který je přímo úměrný šířce paralelní datové sběrnice.

6.1.2 Sériově-paralelní integrátor

V této variantě se násobení provádí sekvenční metodou na principu Boothova algoritmu násobení. Sčítání se však nadále provádí paralelně v jednom kroku.



Obrázek 6.2: Blokové schéma sériově-paralelního integrátoru

Význam jednotlivých bloků:

RV	registr výsledku
MPX	multiplexor
SUM	paralelní sčítačka
ACC	akumulátor
SR	posuvný registr
BNEG	obvod řízené negace (pro sekvenční násobení)

Princip výpočtu sériově-paralelního integrátoru je následující: V registru RV a SR je uloženo y_i. Vstup integračního kroku má hodnotu h, MPX je přepnut na cestu od bloku řízené negace BNEG. Výpočet y_{i+1} pak probíhá takto: nejprve se akumulátor ACC vynuluje, na vstupu je připraven nejméně významový bit $f(y_i)$. Tento bit určí, zda se bude násobenec z registru RN přičítat do akumulátoru ACC, odečítat (vytvoří se záporná podoba násobence - druhý doplněk), nebo zda se bude ignorovat (vynuluje). Výsledek ze sčítačky se zapíše do ACC a celý akumulátor a posuvný registr SR se potom posune o jedno místo doprava.

Tento postup se opakuje, dokud není přijat poslední bit z $f(y_i)$ a následně zpracován. Tímto se dosáhne vynásobení h a $f(y_i)$. Tento výsledek násobení se uloží do posuvného registru SR. Multiplexor se přepne místo z BNEG na RV a výsledek násobení DYp (uložený v ACC) se sečte s hodnotou uloženou v registru výsledku RV a uloží se do ACC a následně do RV. Celý cyklu se opakuje do té doby, dokud není dosaženo požadované přesnosti, nebo maximálního počtu iterací, čímž získáme y_{i+1}.

Výhodou tohoto přístupu je malý počet potřebných vývodů rozhraní zapojení, protože data vstupují i vystupují sériově. Chceme-li zpřesnit výpočet zvětšením počtu bitů, na nichž jsou čísla zobrazena, pak se, na rozdíl od předchozí varianty, v celkovém zapojení nic nezmění, "pouze" se změní šířka registrů, sčítačky a prodlouží se posloupnost řídících

signálů, ale rozhraní integrátoru, stejně jako propojovací síť, zůstanou zachovány.

Nevýhoda oproti předchozí variantě je zpomalení, protože násobení je zde prováděno v n krocích (n = počet bitů zobrazení čísel). Čas výpočtu jednoho členu Taylorovy řady je:

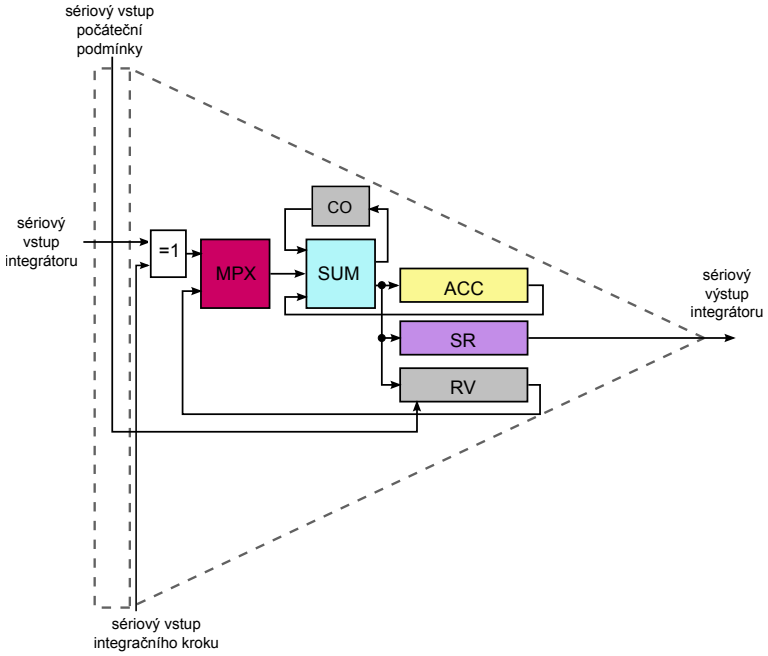
$$\begin{aligned} t_{SP} &= \tau_{nas} + \tau_{sec} + \tau_{sit} \\ t_{SP} &= n{\cdot}\tau_{sec} + \tau_{sec} + \tau_{sit} \end{aligned} \tag{6.2}$$

tedy je dán součtem času násobení (což je vlastně n kroků sčítání $n{\cdot}\tau_{sec}$) a nezbytné přičtení τ_{sec} výsledku násobení k předešlému celkovému výsledku, případně i zpožděním signálů v propojovací síti τ_{sit}.

Jistá možnost optimalizace této varianty integrátoru je v použití překódování více bitů najednou (Boothovo překódování s radixem 4 či 8). K mezivýsledku násobení se pak přičítá násobek integračního kroku, který je určen překódováním vstupní skupiny bitů. Tímto je možno počet potřebných kroků pro násobení zmenšit. Při této variantě nám už ale nestačí pouze vytvářet kladnou a zápornou hodnotu násobence. Musí být použit obvod, který bude vytvářet překódování 2 případně 4 bity najednou. To nám celé zapojení zesložití.

6.1.3 Sériově-sériový integrátor

Tato varianta vychází z principu sériově-paralelního integrátoru. Na rozdíl od předchozí varianty však provádí sekvenčně nejen násobení, ale i operaci sčítání.



Obrázek 6.3: Blokové schéma sériového integrátoru

Význam jednotlivých bloků:
SUM úplná jednobitová sčítačka
CO klopný obvod pro uchování přenosu
MPX multiplexor
ACC akumulátor
RV posuvný registr výsledku
SR výstupní posuvný registr

Funkce je tato: Nejprve se vynuluje ACC, do RV se vloží hodnota y_i. Obvod uchování přenosu CO se vynuluje. Multiplexor MPX se nastaví na cestu z ACC. Na vstupu integrátoru se objeví nejméně významový bit $f(y_i)$ a na sériovém vstupu integračního kroku se postupně objevují jednotlivé bity integračního kroku, počínaje nejméně významovým bitem. Tato posloupnost se v závislosti na hodnotě vstupu integrátoru sériově přičte k akumulátoru. Po dokončení výpočtu mezivýsledku se na vstupu objeví významnější bit $f(y_i)$ a současně se posune výstupní registr SR. Celý postup se opakuje do té doby, až se na vstupu integrátoru objeví poslední (nejvýznamnější) bit $f(y_i)$. Po dokončení operace násobení se hodnota uložená v ACC (DYp) uloží do výstupního registru SR a sériově se přičte k hodnotě uchované v registru výsledku.

Tato varianta má opět menší nároky na zapojení, ale cena, kterou je v tomto případě počet cyklů, jež jsou potřeba na celý výpočet (a tedy i čas), je dána exponenciálním vztahem.

$$\begin{aligned} t_{SS} &= \tau_{nas} + \tau_{sec} + \tau_{sit} \\ t_{SS} &= n{\cdot}n{\cdot}\tau_{sec} + n{\cdot}\tau_{sec} + \tau_{sit} \\ t_{SS} &= n^2{\cdot}\tau_{sec} + n{\cdot}\tau_{sec} + \tau_{sit} \end{aligned} \tag{6.3}$$

τ_{sec} v této verzi představuje čas sčítání úplné jednobitové sčítačky, n je počet bitů, na nichž jsou uloženy hodnoty násobitele i násobence.

6.2 Rozšíření integrátorů

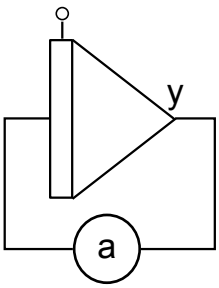
Samotné integrátory, představené výše, slouží k řešení homogenních diferenciálních rovnic typu (2.13). Pro řešení všech typů rovnic vzniklých po provedení transformace pomocí tvořících diferenciálních rovnic je nutné realizovat další operace a to operaci násobení konstantou, při řešení soustav operaci sčítání jednotlivých diferenciálních rovnic (viz např. soustava rovnic a její řešení (2.40)–(2.48)) a násobení jednotlivých diferenciálních rovnic mezi sebou (viz např. rovnice (2.84) nebo (2.92),(2.93)).

6.2.1 Násobení konstantou

Při řešení diferenciální rovnice s konstantními koeficienty:

$$y' = a{\cdot}y \qquad y(0) = y_0 \tag{6.4}$$

Zapojení tohoto typu úlohy je na obrázku 6.4.



Obrázek 6.4: Blokové schéma násobení konstantou

Zápis Taylorovy řady lze obdržet ve známém tvaru:

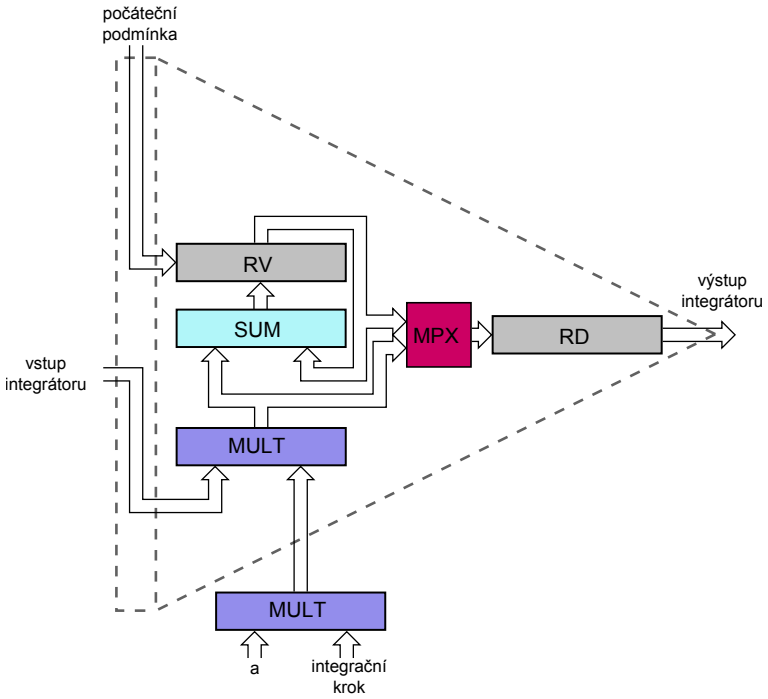
$$y_{(i+1)} = DY0_i + DY1_i + DY2_i + DY3_i + \cdots \tag{6.5}$$

kde význam jednotlivých členů je:

$$\begin{aligned} DY0_i &= a{\cdot}y_i \\ DY1_i &= a{\cdot}h{\cdot}DY0_i \\ DY2_i &= a\frac{h}{2}\,DY1_i \\ DY3_i &= a\frac{h}{3}\,DY2_i \end{aligned}$$

Možné řešení je, že se do přípravných výpočtů zahrne nejen h/p, ale také $a{\cdot}h/p$. Tato možnost bude pro probíhající výpočet v paralelním sytému nejrychlejší a nezabere žádné místo na čipu navíc. Na druhé straně výpočet a distribuce jednotlivých podílů integračního kroku h/p resp. $a{\cdot}h/p$ bude složitější. Už nebude použita v každém integrátoru stejná hodnota, ale musíme zajistit správné nahrání podle řešených diferenciálních rovnic.

Nabízí se i kombinační varianta (relativně velmi nákladná) viz obrázek 6.5, kde je zobrazeno řešení pro paralelně-paralelní verzi integrátoru.



Obrázek 6.5: Technická realizace přednásobení konstantou

Úprava spočívá v tom, že se na vstup pro integrační krok připojí násobička. Na vstupy této násobičky se přivede aktuální podíl integračního kroku h/p a požadovaná konstanta a. Tyto dvě hodnoty se nezávisle na výpočtu v integrátoru vynásobí a výsledek $a{\cdot}h/p$ je připraven na vstupu integrátoru určený pro integrační krok. I v této variantě je důležité před začátkem výpočtu zajistit distribuci konstantních koeficientů ke správným integrátorům resp. násobičkám připojených k těmto integrátorům.

Můžeme také použít k řešení tohoto problému navržených integrátorů. Ty umí operaci násobení provádět. Museli bychom však vyrobit tento integrátor univerzálnější, aby umožňoval při výpočtu jednoho členu Taylorovy řady provádět dvě násobení - $h{\cdot}a$ a následně tento výsledek vynásobit s $DY(p-1)_i$, čímž vypočítáme DYp_i.

Chtěli bychom, aby každý integrátor pracoval stejně, tzn. byl řízen z jedné společné řídící jednotky. Kdyby měl vykonávat tato dvě násobení, museli bychom na vstup pro integrační krok přivádět integrační krok a zadaný koeficient a vše řídit řídící jednotkou. Všechny integrátory by zřejmě nemusely provádět obě násobení - tzn. některé by čekaly, až zase budou moci pokračovat ve společném výpočtu (= náročnější synchronizace).

Jako lepší řešení se jeví přidat do systému násobičky, které by součin $a{\cdot}h$ samostatně vynásobily a potom ho přivedly na vstup pro integrační krok do integrátoru. Integrátory by čekaly pouze při prvním výpočtu součinu, každý další výpočet součinu $a{\cdot}h/p$ by už probíhal současně s výpočtem integrátoru a byl připraven na vstupu integrátoru dříve, než ho bude potřebovat. To nám přináší do našeho systému možnost zřetězeného zpracování - pipeline.

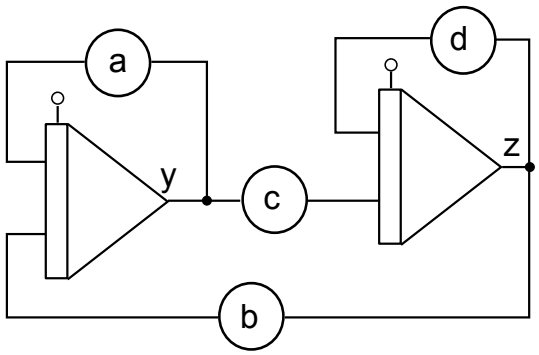
6.2.2 Součtový integrátor

Problém vícevstupého součtového integrátoru se objeví při řešení soustavy diferenciálních rovnic typu:

$$y' = a{\cdot}y + b{\cdot}z \qquad y(0) = y_0 \tag{6.6}$$

$$z' = c{\cdot}y + d{\cdot}z \qquad z(0) = z_0 \tag{6.7}$$

Obecné zapojení tohoto typu úloh je na obrázku 6.6.



Obrázek 6.6: Zapojení s vícevstupým součtovým integrátorem

Pokud na zadanou diferenciální rovnici aplikujeme metodu Taylorovy řady, dostáváme řešení:

$$y_{(i+1)} = y_0 + DY1_i + DY2_i + DY3_i + \cdots$$

$$z_{(i+1)} = y_0 + DZ1_i + DZ2_i + DZ3_i + \cdots$$

kde význam jednotlivých členů je:

$$DY1_i = h(a{\cdot}DY0_i + b{\cdot}DZ0_i) \qquad DZ1_i = h(c{\cdot}DY0_i + d{\cdot}DZ0_i)$$

$$DY2_i = \tfrac{h}{2}(a{\cdot}DY1_i + b{\cdot}DZ1_i) \qquad DZ2_i = \frac{h}{2}(c{\cdot}DY1_i + d{\cdot}DZ1_i)$$

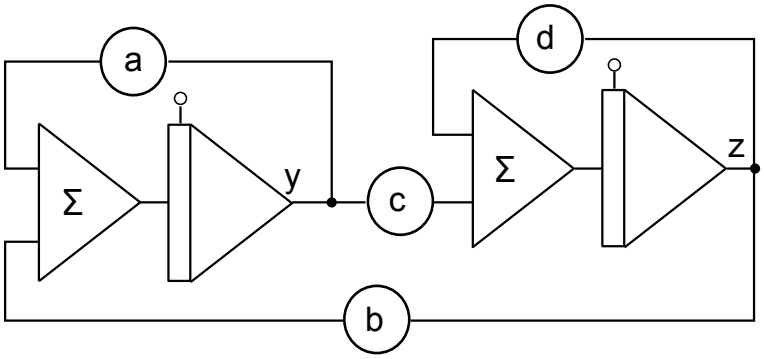
$$DY3_i = \tfrac{h}{3}(a{\cdot}DY2_i + b{\cdot}DZ2_i) \qquad DZ3_i = \frac{h}{3}(c{\cdot}DY2_i + d{\cdot}DZ2_i)$$

Existuje více možností realizace tohoto řešení, které si dále představíme.

Použití bloku vícevstupá sčítačka

Základní a nejjednodušší možností je zařadit do systému víceoperandovou sčítačku. Návrh možností realizace víceoperandového sčítání je obsahem kapitoly 5.2.6.

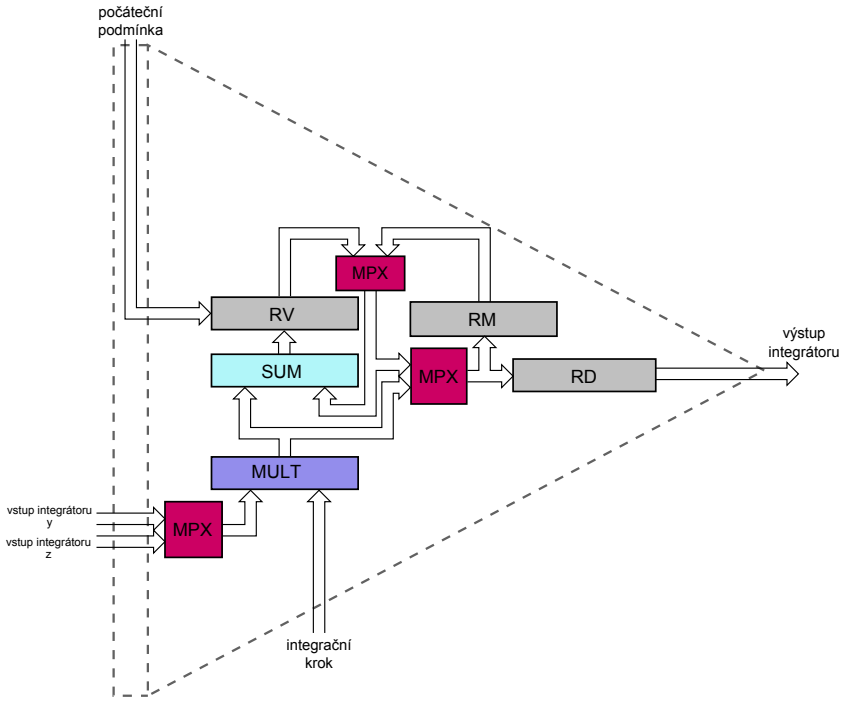
Při této variantě řešení není potřeba nijak upravovat navržené integrátory. Jen se do systému přidají sčítačky, které budou zapojeny jako na obrázku 6.7. Počet operandů sčítačky určuje i počet operandů obsažených v diferenciální rovnici (v našem příkladu dva).



Obrázek 6.7: Součtový integrátor - využití sčítačky a jednovstupého integrátoru

Přepínání vstupů integrátoru (sekvenční integrátor)

Blokové schéma integrátoru, který pracuje na tomto principu je uveden na obrázku 6.8. Nejdříve se provede výpočet odpovídající prvnímu vstupu integrátoru (např. vstup y), potom se dopočítá výpočet odpovídající druhému (vstup z).



Obrázek 6.8: Přepínání vstupů integrátoru

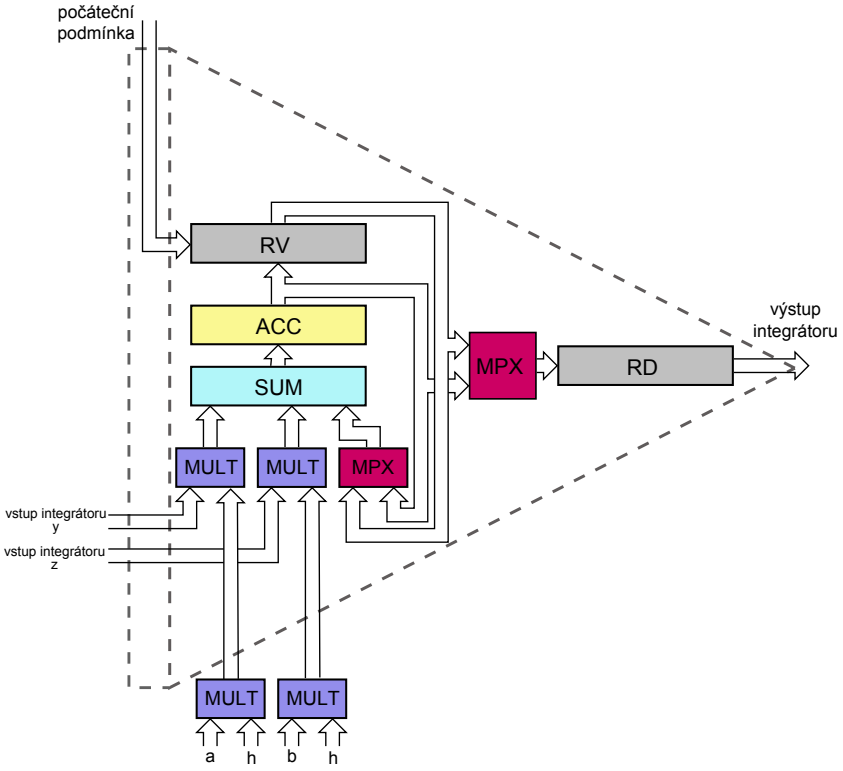
V této variantě použijeme výše navržený jednovstupý integrátor tak, že napřed provedeme jedno vynásobení $(a{\cdot}h{\cdot}DY(p-1)_i)$, které si musíme v integrátoru uložit. Potom přepneme integrátor na druhý vstup a provedeme druhé vynásobení $(b{\cdot}h{\cdot}DZ(p-1)_i)$. Nakonec provedeme sečtení těchto dvou hodnot a získáme výsledek (DYp_i). Tzn. že musíme navržené integrátory rozšířit o registr RM, který budou sloužit k uchování vypočtených hodnot.

Vnitřní zapojení není závislé na počtu operandů obsažených v diferenciální rovnici. Jen se musí odpovídajícím způsobem rozšířit počet vstupů multiplexoru, který přepíná vstupy integrátoru. Jedná se v podstatě o sériový výpočet, kdy jsou postupně zpracovávány jednotlivé vstupy. Z toho vyplývá delší doba výpočtu než v předchozí variantě.

Řízení a synchronizace celého systému bude složitějším a ztratí se univerzálnost navrženého integrátoru. Nastala by situace, že některé integrátory (pouze s jedním vstupem) budou čekat na ty, které počítají s více vstupy (čekají na přepnutí dalšího vstupu).

Vícevstupý kombinační integrátor

Další variantou řešení soustavy rovnic (6.6), (6.7) je návrh vícevstupého integrátoru, který bude oba vstupy zpracovávat současně. Úprava spočívá v použití vícevstupé sčítačky. Princip tohoto rozšíření je demonstrován na obrázku 6.9.



Obrázek 6.9: Vícevstupý integrátor

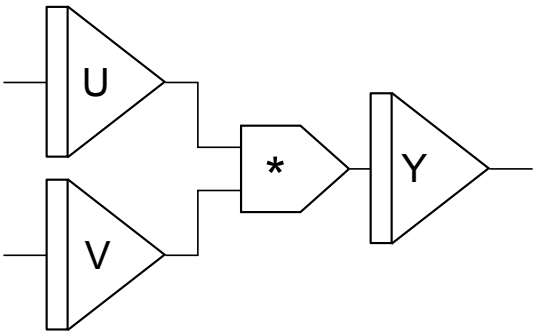
Obvod je samozřejmě nákladný, ale nejrychlejší (vhodný pro jednoúčelové real-time výpočty). Opět toto řešení není zcela univerzální, protože takové integrátory musí mít odlišné řízení od jednovstupého integrátoru. Čas strávený čekáním jednovstupého integrátoru na tento vícevstupý by už ale nebyl tak velký jako u varianty s přepínáním vstupů. Větší počet operandů neprodlužuje zásadně dobu výpočtu jako předchozí varianta, ale celé zapojení se stává podstatně složitější.

6.2.3 Násobící integrátor

Následující integrátor řeší diferenciální rovnici typu:

$$y' = u \cdot v \qquad y(0) = y_0 \tag{6.8}$$

Obecné zapojení tohoto typu úloh je na obrázku 6.10.



Obrázek 6.10: Princip realizace násobícího integrátoru

Zápis Taylorovy řady lze opět obdržet ve známém tvaru:

$$y_{(i+1)} = y_i + DY1_i + DY2_i + DY3_i + DY4_i + \cdots \tag{6.9}$$

Nyní ale výpočet jednotlivých členů bude poněkud obtížnější, protože jde o derivaci součinu. Výpočet jednotlivých derivací je zobrazen dále:

$$\begin{aligned}
y' &= u \cdot v && (6.10)\\
y'' &= u' \cdot v + u \cdot v' && (6.11)\\
y''' &= u'' \cdot v + u' \cdot v' + u' \cdot v' + u \cdot v'' = u'' \cdot v + 2u' \cdot v' + u \cdot v'' && (6.12)\\
y^{IV} &= u''' \cdot v + u'' \cdot v' + 2u'' \cdot v' + 2u' \cdot v'' + u' \cdot v'' + u \cdot v''' = \\
&= u''' \cdot v + 3u'' \cdot v' + 3u' \cdot v'' + u \cdot v''' && (6.13)\\
&\vdots
\end{aligned}$$

Po použití již známého označení bude tvar jednotlivých derivací:

$$\begin{aligned}
DY1 = h \cdot y' \Rightarrow y' = \tfrac{DY1}{h} \qquad & u' = \tfrac{DU1}{h} \qquad & v' = \tfrac{DV1}{h}\\
DY2 = \tfrac{h^2}{2!} y'' \Rightarrow y'' = \tfrac{DY2}{\frac{h^2}{2!}} \qquad & u'' = \tfrac{DU2}{\frac{h^2}{2!}} \qquad & v'' = \tfrac{DV2}{\frac{h^2}{2!}}\\
DY3 = \tfrac{h^3}{3!} y''' \Rightarrow y''' = \tfrac{DY3}{\frac{h^3}{3!}} \qquad & u''' = \tfrac{DU3}{\frac{h^3}{3!}} \qquad & v''' = \tfrac{DV3}{\frac{h^3}{3!}}\\
DY4 = \tfrac{h^4}{4!} y^{IV} \Rightarrow y^{IV} = \tfrac{DY4}{\frac{h^4}{4!}} \qquad & u^{IV} = \tfrac{DU4}{\frac{h^4}{4!}} \qquad & v^{IV} = \tfrac{DV4}{\frac{h^4}{4!}}
\end{aligned}$$

Nyní se vyjádřené derivace z předešlého kroku dosadí do rovnic (6.10)–(6.13) a vyjádří se členy DYp. Následně již obdržíme tvar jednotlivých členů v rovnici (6.9):

$$DY1_i = h{\cdot}(u_i{\cdot}v_i) \tag{6.14}$$

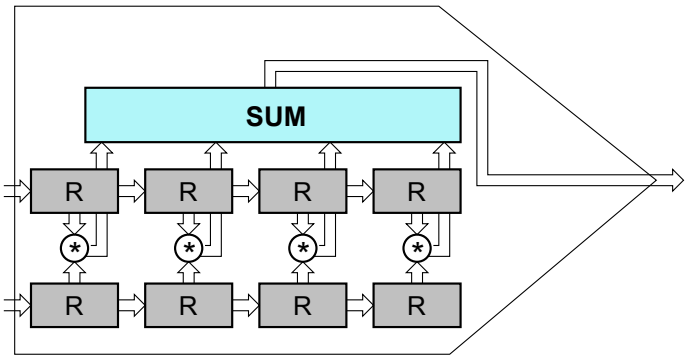
$$DY2_i = \frac{h}{2}(DU1_i{\cdot}v_i + u_i{\cdot}DV1_i) \tag{6.15}$$

$$DY3_i = \frac{h}{3}(DU2_i{\cdot}v_i + DU1_i{\cdot}DV1_i + u_i{\cdot}DV2_i) \tag{6.16}$$

$$DY4_i = \frac{h}{4}(DU3_i{\cdot}v_i + DU2_i{\cdot}DV1_i + DU1_i{\cdot}DV2_i + u_i{\cdot}DV3_i) \tag{6.17}$$

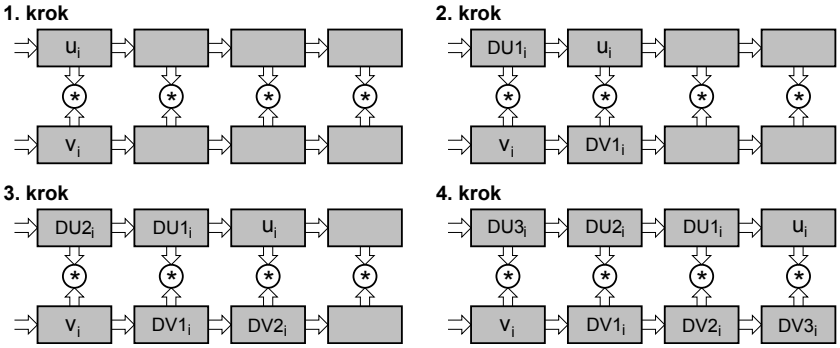
Jak je vidět z obsahu členů DYp_i, v jednotlivých výpočetních krocích (6.14)–(6.17)potřebujeme mít k dispozici všechny předešlé hodnoty DUp_i a DVp_i. Technická realizace "pseudo násobičky" předřazené integrátoru Y na obrázku 6.10 s tím proto musí počítat.

Celý výpočet DYp_i se dělí do dvou kroků. Násobička provádí operaci násobení a následné sečtení výsledků každého násobení - členy v závorce v rovnicích (6.14)–(6.17). Tento výpočet je velmi blízký principu, který využívá operace konvoluce a označuje se multiply-accumulate. Princip této násobičky je zobrazen na obrázku 6.11. Vzniklé číslo přivedeme na vstup integrátoru Y, který je klasické koncepce a provede vynásobení integračním krokem h/p.



Obrázek 6.11: Princip realizace násobičky

Na příkladu jsou k uložení jednotlivých hodnot použity registry. Při tomto způsobu realizace se musí zajistit posouvání hodnot ze vstupu do registrů tak, aby byl výpočet správný. Postup korektního šíření je zobrazen na obrázku 6.12.



Obrázek 6.12: Ukázka šíření operandů registry násobičky

Možností technického řešení je samozřejmě více. Může být například použita paměť.

Ideální by byla paměť s více výstupními branami, které budou připojeny na jednotlivé násobičky. O správné adresování by se musel starat přídavný řídící obvod.

Celková koncepce násobičky je velmi závislá na technologii, která je použita při realizaci paralelního systému. Je zřejmé, že tato komponenta bude mít největší vliv na rychlost výpočtu celého paralelního systému. Všechny integrátory musí čekat na dokončení výpočtu v násobičce. Jak je vidět v rovnicích (6.14)–(6.17), s použitím vyššího řádu metody roste i počet potřebných operací násobení a přičtení. Pokud bude systém provozován v některých kritických situacích, bude rychlejší, když se vyhneme použití vyššího řádu metody Taylorovy řady. Toho můžeme dosáhnout např. použitím menšího integračního kroku.

6.3 Řadič - generátor řídících signálů

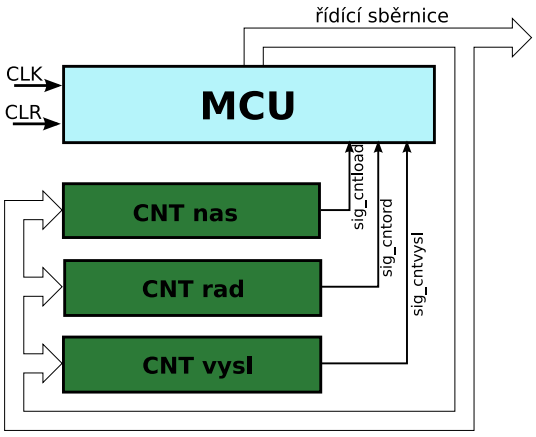
Funkce řadiče spočívá v generování posloupnosti řídících signálů pro procesor (integrátor) tak, aby plnil požadovanou funkci.

Existují dva základní přístupy v návrhu řadičů:

- **obvodový** - základem je dekodér instrukce, který podle typu instrukce, stavových bitů procesoru a aktuálního stavu vygeneruje odpovídající řídící signály pro obvod.
- **mikroprogramový** - základem je paměť mikroprogramu, ze které se podle přiložené adresy vybere požadovaná instrukce - operace, která už přímo generuje řídící signály pro obvod. Každá instrukce obsahuje adresu následující instrukce, která se má provádět.

Podrobný popis řadičů nalezneme v [Drá95] a [Hwa06, Chapter 10: Control Units]. Detailní popis návrhu mikroprogramového řadiče je v [PP06, Kapitola 14: Mikroprogramový automat].

Specializovaný řadič pro náš systém se příliš nepodobá univerzálním řadičům, protože je zde pevně stanoven sled řídících signálů, a není proto potřebný žádný dekodér instrukcí. V našem případě se dá použít řadič vycházející z modifikace mikroprogramového řadiče.



Obrázek 6.13: Generátor řídících signálů

Důležitou částí námi navrhovaného řadiče je počítání smyček. Hlavní smyčka počítá počet výsledků (y_i), další smyčka počítá dosažený řád Taylorovy řady (DYp_i). Další smyčku tvoří počítání délky slova (dílčích součtů při násobení) u sériových variant integrátorů a případně počet součtů úplné jednobitové sčítačky u sério-sériového integrátoru.

Pro realizaci smyček se přímo nabízí použití čítačů, které by přímo ovládal řadič. Na obrázku 6.13 je uvedeno blokové schéma specializovaného řadiče využívajícího čítače.

Význam jednotlivých čítačů je následující:

- **CNT nas** - počítá délku slova (počet posuvů SR a ACC) - dílčích součtů při násobení (u sériově-paralelní varianty integrátoru) případně počet součtů sčítačky (u sériově-sériového integrátoru)
- **CNT rad** - počítá řád Taylorovy řady (DYp_i). Lze spojit s komparátorem, který bude porovnávat poslední dva dosažené výsledky. Pokud budou stejné, nemusíme již zbytečně počítat dále (nedosáhneme již žádného zpřesnění)
- **CNT vysl** - počítá požadovaný počet výsledků (y_i)

Další částí, o kterou může být řadič rozšířen, je komparátor. Ten plní funkci "omezovače" řádu Taylorovy řady. To znamená, že pokud se aktuální výstupní hodnota sledovaného integrátoru shoduje s jeho předchozí hodnotou, generuje komparátor signál, který může přerušit výpočet následujícího řádu. Tím se zamezí výpočtu, který už nezpřesňuje výpočet, ale přispívá pouze k jeho zpomalení.

Protože všechny integrátory (aritmeticko-logické jednotky) provádějí stejný výpočet, mohou být všechny řízeny ze společné řídící jednotky - řadiče. Toto zapojení odpovídá architektuře SIMD (viz kap. 3).

6.4 Propojovací síť

6.4.1 Propojení jednotlivých aritmeticko-logických jednotek

Jedním z úkolů propojovacího systému je realizace propojení vstupů a výstupů jednotlivých mikroprocesorů mezi sebou.

Pro náš systém je jako propojovací síť nejvhodnější obyčejné statické propojení mezi prvky, podle konkrétního výpočetního schématu (řešené soustavy diferenciálních rovnic). Budeme-li však chtít, aby systém byl schopen provést výpočet obecné diferenciální rovnice, musíme použít univerzální statickou či dynamickou propojovací síť. Použití statického propojení podle řešeného zadání je možné pouze v případě, že budeme pro implementaci používat rekonfigurovatelnou technologii. Zde nemusíme mít pevně danou topologii sítě, ale vlastní statické propojení se nastaví při inicializaci celého paralelního systému. Tento přístup je nejrychlejší a nenáročný na další řízení.

Chceme-li použít takovou síť, na níž by bylo možné použít jakékoliv výpočetní schéma, nemůžeme uvažovat o použití nějaké obecné statické topologie uvedené v kapitole 4.1. Hlavní důvod je ten, že nemůžeme zaručit, že libovolné dva integrátory spolu budou moci komunikovat. To můžeme zaručit pouze u úplného propojení, kde je každý prvek propojen se všemi ostatními. Tato verze má však značné omezení týkající se počtu propojených uzlů. Je použitelná jen pro malé systémy. Navíc by každý integrátor musel na vstupu obsahovat logiku, která by zajistila maskování nepotřebných vstupních signálů.

Při použití dynamické (nepřímé) propojovací sítě se provede před spuštěním vlastního výpočtu inicializace sítě (nastavení cest ve směrovacích prvcích). Dále se už propojení sítě nebude měnit. Propojovací cesta tvoří významnou složku zpoždění. Z tohoto důvodu je nevhodné použití víceúrovňových zapojení.

Nejvhodnější propojovací síť (pro nepříliš rozsáhlé systémy) je křížový přepínač. Má ze všech dynamických sítí nejmenší zpoždění, je jednoúrovňový a umožňuje propojení jednoho

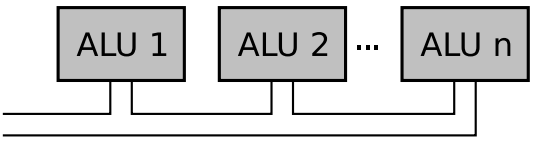
výstupu na více vstupů (větvení v blokovém schématu).

6.4.2 Datové sběrnice pro nahrání počátečních dat a výsledku

Propojovací systém dále slouží k napojení procesorů na řídící obvody, ze kterých se nahrávají hodnoty integračního kroku h a počáteční podmínky y_0 do jednotlivých aritmeticko-logických jednotek a po výpočtu slouží k obdržení hodnoty výsledku.

Jak bylo uvedeno výše, můžeme systémy z hlediska jejich komunikace rozdělit na sériové a paralelní. Z hlediska principiálního však mezi nimi není nijak významný rozdíl. Jedná se totiž "pouze" o počet fyzických cest propojení. Z praktického hlediska je však zřejmé, že mnohem výhodnější je, je-li těchto cest co nejméně.

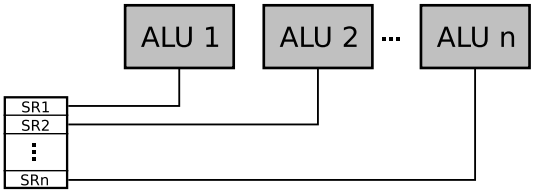
Celý systém může být propojen paralelní sběrnicí (viz obrázek 6.14).



Obrázek 6.14: Propojení paralelní sběrnicí

V každém hodinovém taktu jsou nahrána do jednoho procesoru všechna data. V dalším taktu do dalšího, atd. Tzn. do n procesorů budou data nahrána během n taktů hodin.

Paralelní sběrnici můžeme nahradit n sériovými sběrnicemi (viz obrázek 6.15), kde n udává počet mikroprocesorů.



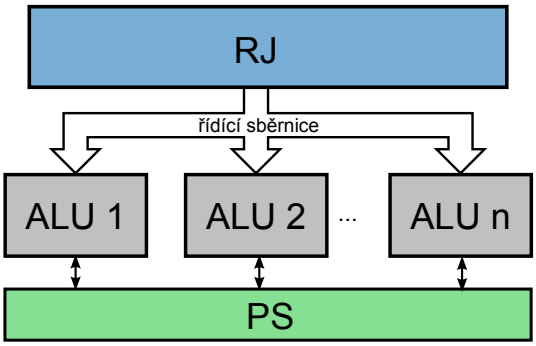
Obrázek 6.15: Propojení sériovými sběrnicemi

Do každého procesoru současně se nahrává s každým taktem hodin bit požadovaných dat (počáteční podmínka h, integrační krok y_0) z bufferů (posuvných registrů SR1 ... SRn), kde jsou požadované hodnoty uloženy. Přenos trvá N taktů, kde N je počet bitů (šířka sběrnice).

Jakmile bude počet procesorů větší než šířka potřebné paralelní sběrnice (což je náš cíl), bude sériový přenos proveden rychleji než paralelní přenos dat.

6.5 Specializovaný paralelní systém

Podoba celého vyvíjeného specializovaného paralelního systému je popsána v této kapitole. Blokové schéma, které ukazuje základní návrh tohoto paralelního systému, je na obrázku 6.16.



Obrázek 6.16: Blokové schéma specializovaného paralelního systému

Celý specializovaný paralelní systém je sestaven z těchto základních bloků:

- aritmeticko-logické jednotky ALU1 ... ALUn
- propojovací systém PS
- řídící jednotky RJ

6.5.1 Aritmeticko-logické jednotky - ALU

Aritmeticko-logické jednotky (integrátory) realizují numerickou integraci. Popis celkové koncepce je uveden výše.

Základním způsobem komunikace mezi jednotlivými elementárními mikroprocesory při výpočtu je komunikace přes sériový vstup a sériový výstup každého procesoru - preferovaný typ integrátoru je sériově-paralelní varianta.

6.5.2 Řídící jednotka - RJ

Řízení celého paralelního systému je realizováno řídící jednotkou RJ.
Základní úkoly jednotky jsou:

- řízení zápisu dat do všech výpočetních jednotek
- řízení výpočtu paralelního systému
- řízení distribuce výsledku

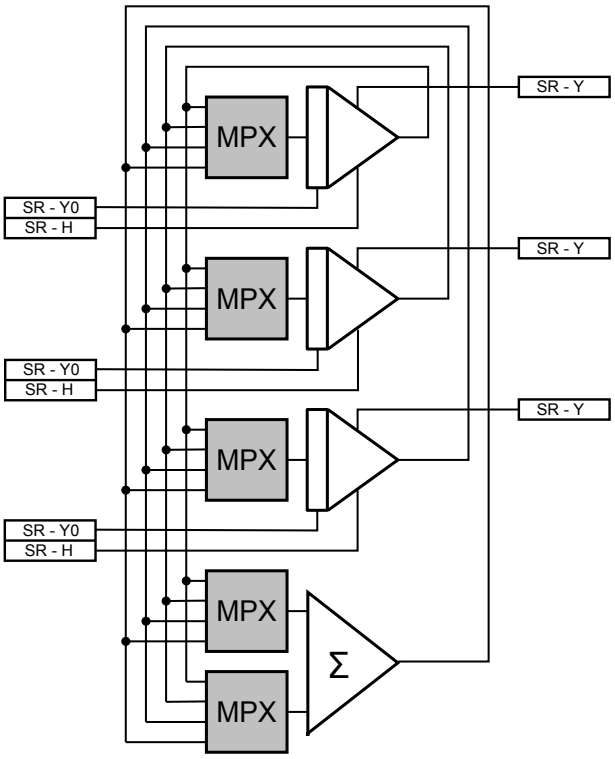
Řídící jednotka přejímá řízení, případně spolupracuje s nadřazenou jednotkou - počítačem. Nadřazená jednotka se stará o transformaci vstupních diferenciálních rovnic s využitím tvořících diferenciálních rovnic. Podle nově vzniklých diferenciálních rovnic provede propojení paralelního systému - nastavení propojovacího systému. Dále zajistí počáteční data (y_0 a h) a předá řízení řídící jednotce. Ta se stará o zápis dat, řídí výpočet a distribuci výsledku.

6.5.3 Propojovací systém - PS

Propojovací systém slouží k propojení vstupů a výstupů jednotlivých výpočetních bloků (integrátorů, sčítaček, násobiček) mezi sebou podle řešené soustavy diferenciálních rovnic. Dále musí sloužit k nahrání počátečních dat (počáteční podmínky a integrační kroky) a

vypočteného výsledku. V neposlední řadě musí být každý integrátor připojen řídící sběrnicí na řídící jednotku.

Na obrázku 6.17 je detailnější blokové schéma paralelního systému zobrazující zejména propojovací systém. Tento paralelní systém obsahuje 3 integrátory a jednu sčítačku.



Obrázek 6.17: Detailnější schéma paralelního systému

K propojení výpočetních jednotek je použit křížový přepínač. Křížový přepínač je realizován multiplexory. Každý multiplexor obsahuje registr adresy, který určuje vstupní kanál multiplexoru, který se bude dostávat na výstup - vstup připojené výpočetní jednotky.

K nahrání počátečních dat slouží sériové sběrnice. Data jsou uložena v posuvných registrech SR - Y0 a SR - H a postupně se sériově nahrají do integrátorů. Tato verze obsahuje zvláštní sběrnici pro počáteční podmínku a integrační krok. Je možná úprava, která spočívá v použití jedné sdílené sběrnice. Napřed by se do integrátoru nahrála počáteční podmínka a následně integrační krok (nutný poloviční počet cest, ale pomalejší nahrávání). A nakonec systém obsahuje sériové sběrnice pro nahrání vypočtených výsledků z integrátorů do posuvných registrů SR - Y.

6.6 Shrnutí

Z rozboru provedeného v této kapitole vyplývá, že specializovaný paralelní systém bude typu SIMD. Všechny výpočetní jednotky budou provádět stejný výpočet nad různými daty a mohou být řízeny z jedné řídící jednotky.

Aritmeticko-logická jednotka provádí pouze dvě základní operace: násobení a sčítání. ALU představuje integrátor a tvoří základ celého paralelního systému. Výpočet těchto dvou základních operací a způsob komunikace může být prováděn sériově nebo paralelně. Z toho

vyplývá rozdělění integrátorů na paralelně-paralelní, sériově-paralelní a sériově-sériové.

Dalšími neméně důležitými částmi paralelního systému jsou řídící jednotka a propojovací síť. K řízení není potřeba použít univerzální řadič, protože je pevně dán sled operací (instrukcí), a je proto použit specializovaný řadič. Je nutno navrhnout dva typy propojovacích sítí. Jedna pro propojení všech výpočetních jednotek mezi sebou a druhá pro nahrání počátečních dat a výsledků. Preferovanou variantou je sériové propojení, které zabírá podstatně menší plochu čipu a dovolí vyšší přenosovou rychlost.

Kapitola 7

Implementace specializovaného paralelního systému

Tato kapitola obsahuje detailnější popis technické realizace celého paralelního systému, jehož návrh je představen v předchozí kapitole. Popisuje se zde podrobněji implementace všech tří typů integrátorů, které jsou realizovány v pevné řádové čárce a představena koncepce integrátoru v pohyblivé řádové čárce. Na závěr je provedeno srovnání implementovaných paralelních systémů.

7.1 Integrátory v pevné řádové čárce

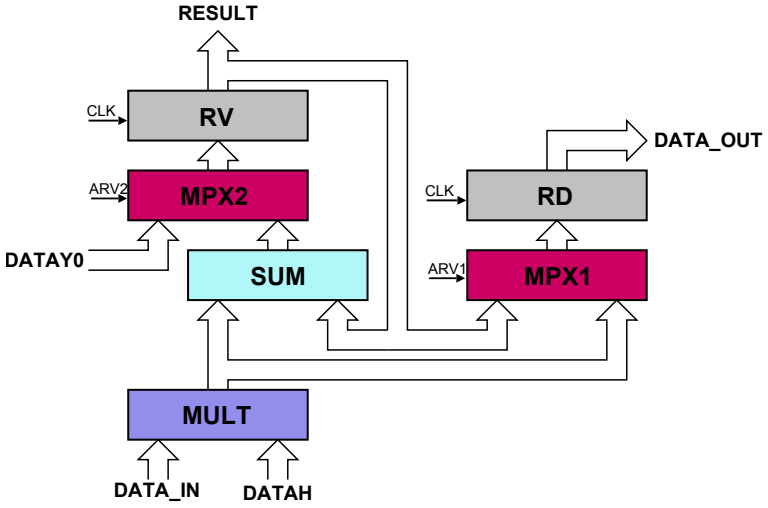
Obecně je koncepce integrátorů jako aritmeticko-logických jednotek popsána v kapitole 6.1. Představené jednotky se využívají pro výpočty v pevné řádové čárce, konkrétně ve formátu d) uvedeném v podkapitole 5.1.1. Jedná se o formát, kde nejvyšší významový bit představuje znaménko, následuje řádová čárka a všechny ostatní bity slouží k uložení desetinné části zobrazeného čísla.

7.1.1 Paralelně-paralelní integrátor

V této paralelně-paralelní verzi ALU je výpočet násobení i sčítání prováděn paralelně a komunikace mezi jednotkami probíhá také paralelně (jednotky jsou propojeny paralelními sběrnicemi). Celé zapojení aritmeticko-logické jednotky je na obrázku 7.1.

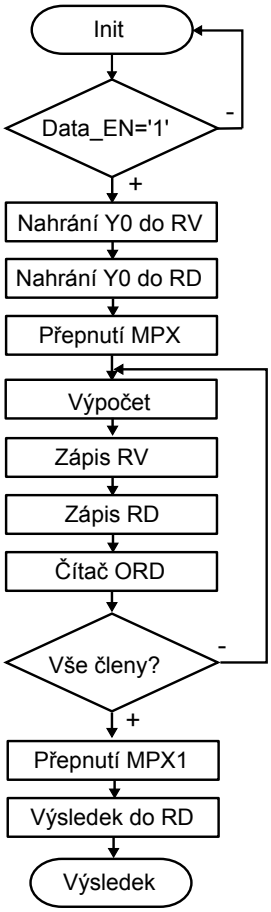
Skládá se z těchto bloků:

- paralelní násobička MULT - provádí násobení dat na vstupu ALU s hodnotou integračního kroku
- paralelní sčítačka SUM - provádí součty jednotlivých členů Taylorovy řady (DYp_i)
- multiplexor MPX1 - slouží k přepnutí hodnoty výsledku násobení DYp_i nebo výsledku z RV do RD
- multiplexor MPX2 - slouží k přepnutí počáteční podmínky na počátku výpočtu, jinak je přepnut na výstup ze sčítačky SUM
- registr výsledku RV - slouží k uchování celkových výsledků (y_{i+1})
- registr násobení RD - slouží k uchování hodnot výsledků násobení DYp_i



Obrázek 7.1: Blokové schéma paralelně-paralelní aritmeticko-logické jednotky

Činnost aritmeticko-logické jednotky je názorně zobrazena vývojovým diagramem (obrázek 7.2), popis jednotlivých stavů je následující:



Obrázek 7.2: Činnost ALU

Init
vynulování čítače čítající výsledky a členy Taylorovy řady, čekání na zadání vstupních dat, MPX2 přepnut na vstup DATAY0

Nahrání Y0 do RV
nahrání počáteční podmínky do registru RV

Nahrání Y0 do RD
nahrání počáteční podmínky do registru RD cestou z registru RV

Přepnutí MPX
oba multiplexory se přepnou signálem ARV=1, MPX1 na vstup z násobičky MULT a MPX2 na vstup ze sčítačky SUM

Výpočet
probíhá výpočet - vynásobení hodnot v násobičce MULT a sečtení ve sčítačce SUM

Zápis RV
zápis mezivýsledku do registru RV

Zápis RD
zápis výsledku násobení DYp_i do registru RD

Čítač ORD
zvýšení čítače řádu metody a test, zda bylo dosaženo požadované přesnosti (byly vypočítány všechny členy DYp_i Taylorovy řady)

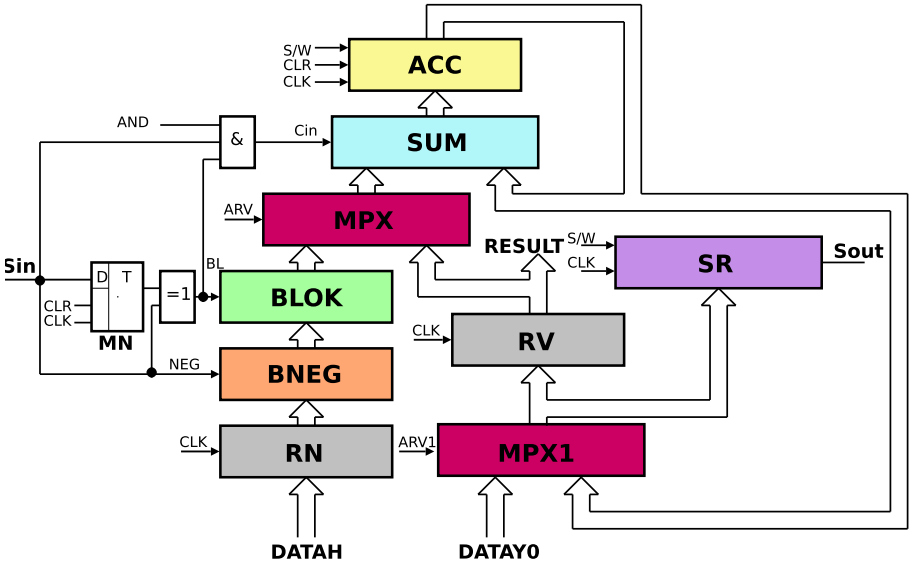
Přepnutí MPX1
přepnutí multiplexoru MPX1 na vstup z registru výsledku RV a vynulování čítače řádu metody

Výsledek do RD
zvýšení čítače počtu výsledků, uložení celkového výsledku y_{i+1} do registru RD - slouží jako počáteční podmínka pro výpočet y_{i+2}

Výsledek
příznak, že je již výsledek vypočítán a je možné ho přečíst

7.1.2 Sériově-paralelní integrátor

V této sériově-paralelní verzi ALU probíhá komunikace mezi jednotkami sériově a vlastní výpočet uvnitř jednotek sério-paralelně. Násobení je realizováno sériově na principu Boothova algoritmu. Jednotlivé mezisoučiny provádí paralelní sčítačka. Celé zapojení aritmeticko-logické jednotky je na obr. 7.3.

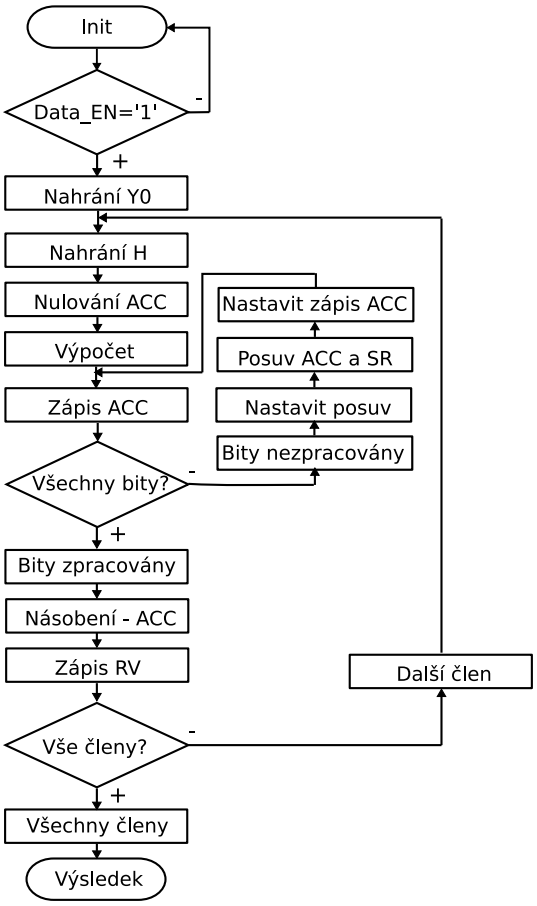


Obrázek 7.3: Blokové schéma sériově-paralelní aritmeticko-logické jednotky

Skládá se z těchto bloků:

- paralelní sčítačka SUM - vytváří dílčí součiny při násobení
- bloky řízení přenosu dat BNEG (negace), BLOK (vynulování) - provádějí řízenou negaci či zablokování (vynulování) dat podle pravidel Boothova algoritmu
- akumulátor ACC - slouží k uložení výsledku (mezivýsledků) ze sčítačky
- obvod malé nuly MN (klopný obvod typu D) - slouží k uchování i-1 bitu (rozšíření registru násobitele SR, při zahájení násobení vynulován)
- pomocné obvody nonekvivalence a logický součin - generují řídící signály pro násobení: NEG (negace dat), BL (zablokování dat) a C_{IN} (přičtení jedničky k nejméně významovému bitu) - bližší popis je v kap 5.4.2
- multiplexor MPX - slouží k přepnutí hodnoty počáteční podmínky do sčítačky - přičtení počáteční podmínky k výsledku násobení
  MPX1 - slouží k nahrání počáteční podmínky před začátkem výpočtu
- registr výsledku RV - slouží k uchování celkových výsledků (y_{i+1})
- registr násobence RN - uložení integračního kroku (h) a jeho podílů
- posuvný registr SR - během výsledku se zde ukládají výsledky násobení (DYp_i) a jeho sériový výstup slouží jako výstup aritmeticko-logické jednotky

Činnost aritmeticko-logické jednotky je názorně zobrazena na vývojovém diagramu (obrázek 7.4), popis jednotlivých stavů je následující:



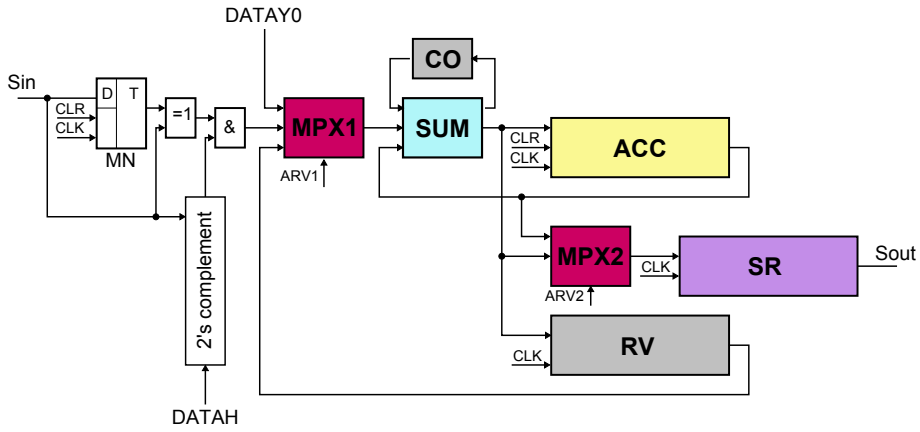
Obrázek 7.4: Činnost ALU

Init
vynulování čítače výsledků a čekání na zadání vstupních dat, MPX1 přepnut na DATAY0
Nahrání Y0
vynulování čítače, který počítá řád metody, nahrání počáteční podmínky do registrů SR a RV
Nahrání H
nahrání integračního kroku do RN a SR, přepnutí multiplexoru MPX1 na cestu z ACC
Nulování ACC
nulování čítače počítající délku slova, nulování akumulátoru ACC a klopného obvodu MN
Výpočet
probíhá výpočet - sečtení ve sčítačce SUM
Zápis ACC
zápis mezivýsledku do ACC, test zda jsou zpracovány všechny bity posuvného registru SR
Bity nezpracovány
uložení hodnoty ze sériového vstupu S_{IN} do klopného obvodu MN
Nastavit posuv
nastavení posuvného registru SR a akumulátoru ACC do režimu posuvu
Posuv
posuv SR a ACC, zvýšení čítače délky slova
Nastavit zápis ACC
nastavení SR a ACC do režimu zápisu
Bity zpracovány
vynulování čítače počítající délku slova, uložení výsledku násobení (DYp_i) z ACC do SR (počáteční podmínka pro výpočet dalšího členu Taylorovy řady), přepnutí multiplexoru MPX na cestu z RV
Násobení ACC
uložení výsledku do ACC (přičteného n-tého členu DYp_i k celkovému výsledku y_{i+1})
Zápis RV
uložení výsledku z ACC do registru výsledku RV, zvýšení čítače řádu metody, přepnutí multiplexoru MPX zpět na cestu z blokovacího obvodu BLOK. Test zda bylo dosaženo požadované přesnosti (vypočítány všechny členy DYp_i Taylorovy řady)
Další člen
nastavení řídících signálů potřebných pro výpočet dalšího upřesňujícího členu
Všechny členy
zvýšení čítače počtu výsledků, vynulování čítače řádu metody, uložení celkového výsledku y_{i+1} do SR - slouží jako počáteční podmínka pro výpočet y_{i+2}
Výsledek
příznak, že je již výsledek vypočítán a je možné ho přečíst

Je také možná verze bez blokovacího obvodu BLOK. Jsou použity jen logické členy, které řídí zápis do ACC. Místo nulování procházejících dat se jen zablokuje ve správný okamžik zápis do ACC (= když jsou testované bity shodné - viz Boothův algoritmus násobení).

7.1.3 Sériově-sériový integrátor

Tato jednotka vychází z předešlé sériově-paralelní verze. Komunikace mezi jednotkami probíhá také sériově a vlastní výpočet uvnitř jednotek tentokrát probíhá celý sériově. Jednotlivé mezisoučty se počítají sériově pomocí jednobitové sčítačky. Celé zapojení aritmeticko-logické jednotky je na obrázku 7.5.



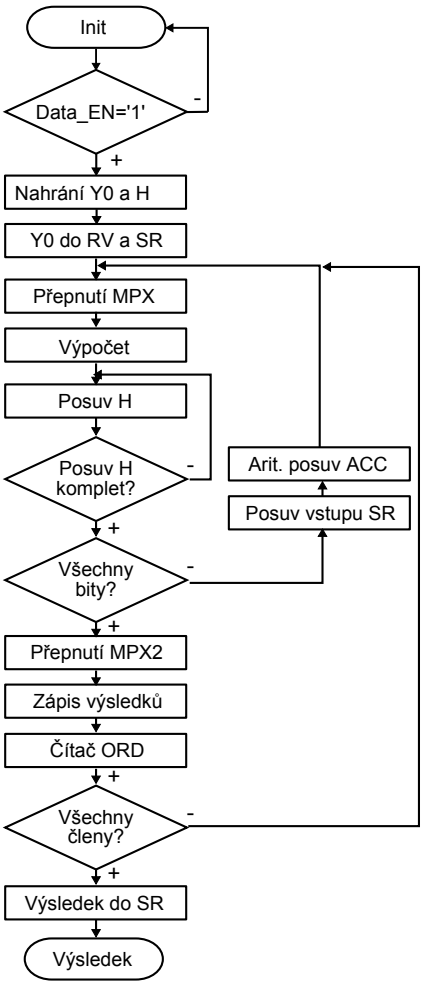
Obrázek 7.5: Blokové schéma sériově-sériové aritmeticko-logické jednotky

Skládá se z těchto bloků:

- úplná jednobitová sčítačka SUM a obvod pro uchování přenosu CO - vytváří dílčí součty a součiny při násobení
- 2's complement - blok vytvářející sériově dvojkový doplněk, podle hodnoty na vstupu se určí, zda použijeme kladnou či zápornou hodnotu vstupu s integračním krokem. Následně hradlo AND zajistí blokování (při kombinaci Sin a MN 00 nebo 11) nebo propuštění hodnot dále do sčítačky
- obvod malé nuly MN (klopný obvod typu D) - slouží k uchování i-1 bitu (rozšíření posuvného registru SR připojeného na vstup, při zahájení násobení vynulován)
- akumulátor ACC - slouží k uložení výsledků (mezivýsledků) ze sčítačky
- multiplexor MPX1 - slouží k přepnutí hodnoty z registru RV do sčítačky - přičtení počáteční podmínky či následně mezivýsledku y_{i+1} k výsledku násobení DYp_i, na začátku je přepnut pro nahrání počáteční podmínky do RV a SR
  MPX2 - slouží k nahrání počáteční podmínky před začátkem výpočtu nebo k nahrání výsledku násobení DYp_i uloženého v ACC
- registr výsledku RV - slouží k uchování celkových výsledků y_{i+1}, na začátku výpočtu je do něho nahrána počáteční podmínka
- posuvný registr SR - během výpočtu se zde ukládají výsledky násobení DYp_i a jeho sériový výstup slouží jako výstup aritmeticko-logické jednotky

Na vstupy DATAY0 a DATAH jsou připojeny výstupy posuvných registrů, které obsahují hodnotu počáteční podmínky y_0 (registr SR - Y0) resp. integračního kroku h či jeho podílů h/p (registr SR - H). To odpovídá propojení sériovými sběrnicemi představenému v kapitole 6.4.2 - "Datové sběrnice pro nahrání počátečních dat a výsledku".

Činnost aritmeticko-logické jednotky je názorně zobrazena na vývojovém diagramu (obrázek 7.6), popis jednotlivých stavů je následující:



Obrázek 7.6: Činnost ALU

Init
vynulování čítače čítající počet výsledků a řád metody, čekání na zadání vstupních dat, MPX1 přepnut na vstup DATAY0 a MPX2 na vstup ze sčítačky SUM

Nahrání Y0 a H
vynulování ACC, CO a MN, nahrání počáteční podmínky do registru SR - Y0 a integračního kroku do SR - H

Y0 do RV a SR
sériové nahrání počáteční podmínky do registru RV a SR, řízeno čítačem počtu posuvů

Přepnutí MPX
nulování čítače počtu posuvů, nulování obvodu CO, MPX1 přepnut na vstup integrátoru, MPX2 přepnut na výstup z ACC. Registr SR - H nastavíme na posuv

Výpočet
probíhá výpočet - sečtení ve sčítačce SUM

Posuv H
zápis mezivýsledku do ACC a posuv registru SR - H, zvýšení čítače posuvů

Posuv H komplet
test zda byl proveden již posuv všech bitů

Všechny bity
posuv registru SR - H proběhl, uložení hodnoty ze vstupu do MN, test zda je kompletní posuv registru SR

Posuv vstupu SR
posuv obsahu registru SR a zvýšení čítače počtu posuvů SR, nastavení ACC na aritmetický posuv

Aritm. posuv ACC
aritmetický posuv akumulátoru ACC

Přepnutí MPX2
posuv celého SR dokončen - násobení kompletní, vynulování čítače počtu posuvů a čítače počtu posuvů SR, přepnutí MPX1 na výstup z RV

Zápis výsledků
postupné ukládání výsledku ze sčítačky do ACC a RV (přičteného p-tého členu DYp_i k celkovému výsledku y_{i+1}), současně se ukládá výsledek násobení z ACC do SR

Čítač ORD
zvýšení čítače řádu metody a nastavení registru SR - H na zápis

Všechny členy
test dosažení požadované přesnosti (vypočítány všechny členy Taylorovy řady DYp_i), nahrání další hodnoty h/p do registru SR - H

Výsledek do SR
zvýšení čítače počtu výsledků, vynulování čítače řádu metody a uložení celkového výsledku y_{i+1} do posuvného registru SR - slouží jako počáteční podmínka pro výpočet y_{i+2}
Výsledek
příznak, že je již výsledek vypočítán a je možné ho přečíst

7.2 Integrátor v pohyblivé řádové čárce

U výpočtů, kde dochází k velkému rozptylu počítaných hodnot, nebude stačit verze v pevné řádové čárce. Docházelo by k velkým chybám, které je sice možno snížit použitím více bitů pro uložení dat, ale to není možné do nekonečna. Je tedy nezbytné použít k uložení dat pohyblivou řádovou čárku a s tím související modifikace prováděných výpočetních operací. Koncepce aritmeticko-logických jednotek popsaných v kapitole 6.1 může být použita pro výpočty také v pohyblivé řádové čárce. Tento návrh však musí být rozšířen o práci s mantisou a exponentem.

Paralelně-paralelní integrátor v pohyblivé řádové čárce se bude nejvíce podobat verzi v pevné řádové čárce. Rozdíl bude jen v použité násobičce a sčítačce. Princip sčítání resp. násobení v pohyblivé řádové čárce je popsán v podkapitole 5.2.5 resp. 5.4.4. Popis možného provedení násobičky je obsahem článku [HKSE07]. V sériových variantách je základním stavebním prvkem sčítačka a násobení se provádí sekvenčně, konkrétně Boothovým algoritmem.

Detailnější návrh sériově-paralelního integrátoru v pohyblivé řádové čárce je obsahem diplomové práce [Čam11]. Základ výpočetní jednotky je shodný s verzí v pevné řádové čárce. Integrátor je rozšířen o obvody, které slouží k posunu mantis a odpovídající zvýšení či snížení hodnoty exponentu, aby operandy měly při sčítání shodné řády. Je také přidán obvod, který zajistí normalizaci výsledku. Dále je použito více registrů sloužící pro uložení exponentů i mantis zvlášť.

Existují dvě verze a to integrátor zpracovávající mantisu i exponent postupně v jedné jednotce a integrátor, který je tvořen dvěma dílčími jednotkami, jednotkou mantisy a jednotkou exponentu. Jednotka exponentu musí zajistit součet i rozdíl exponentů, inkrementaci a řízení aritmetického posuvu mantisy. Jednotka mantisy musí umožňovat násobení, sčítání, odčítání a aritmetické i logické posuvy.

Zde je představen princip, kdy jsou mantisa i exponent zpracovány postupně v jedné jednotce. Pro větší přehlednost jsou registry sloužící pro uložení mantisy a exponetu operandů potřebných při výpočtu nahrazeny pamětí. Celé zapojení je na obrázku 7.7.

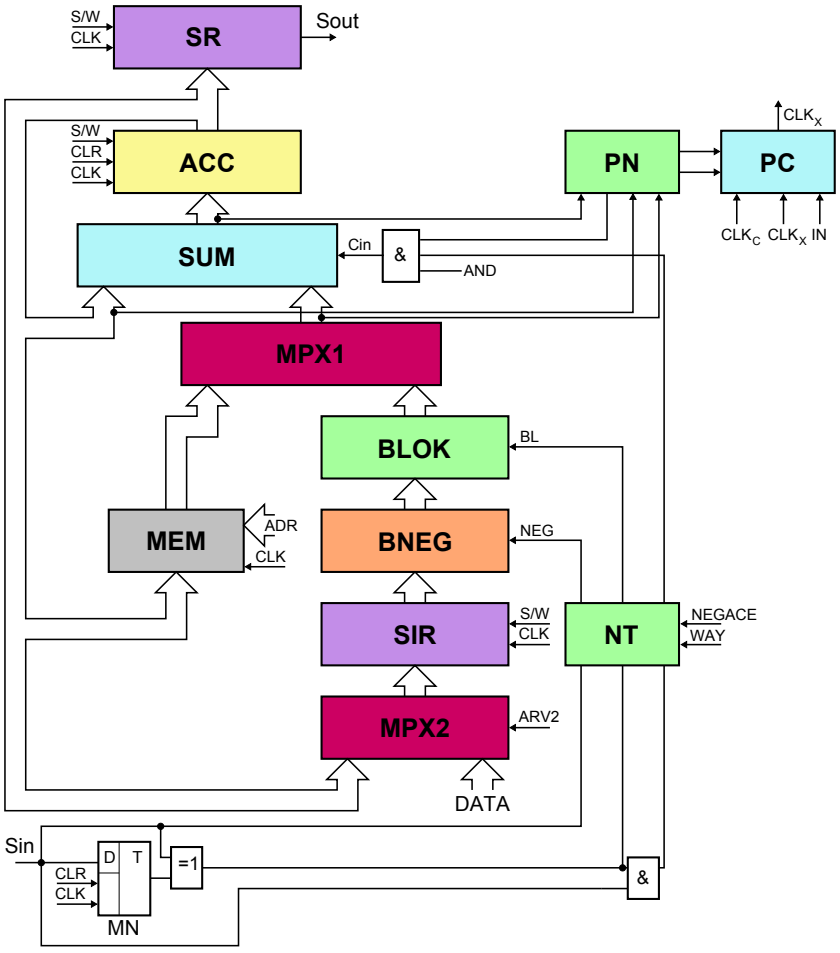
Jsou použity stejné bloky, jako ve verzi pro pevnou řádovou čárku - viz 7.1.2. Navíc jsou použity následující obvody:

- Obvod podmíněného zápisu PC - řídí posunutí mantis, aby měly stejné exponenty a mohly být sečteny. Nejdříve se provede odečtení exponentů, podle znaménka rozdílu se určí mantisa, kterého operandu se bude posouvat. Protože ve všech integrátorech nebude obdržené znaménko stejné, musí se zajistit správná posloupnost operací. K tomu nám slouží vstupní signál IN. Napřed se např. provede posuv v jednotkách se záporným rozdílem a následně s kladným.

  Signály CLK_X slouží k řízení zápisu do jednotlivých registrů (např. CLK_{SR}, CLK_{SIR} atd.). Registry, u kterých uplatňujeme podmíněný zápis, nejsou ovládány přímo řadičem, ale hodinový signál prochází tímto obvodem podmíněného zápisu. Hodinový signál

$CLK_C = 0$ značí, že obvod podmíněného zápisu není v činnosti a hodinové signály CLK_X nejsou nijak blokovány.

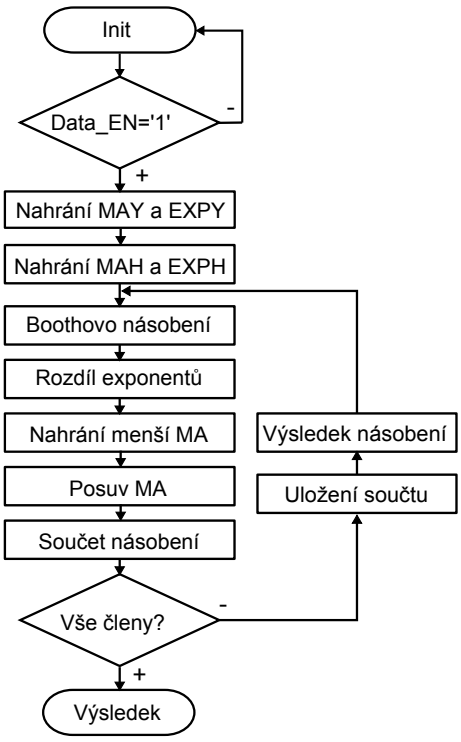
- Obvod blokování hodin registru SIR - posouvá mantisu uloženého operandu o rozdíl exponentů, aby si řády mantis odpovídaly. Je řízen obvodem podmíněného zápisu.
- Obvod ošetření přetečení a normalizace PC - při sčítání operandů může dojít k přetečení. Přetečení je ošetřeno posunem mantisy vpravo a inkrementací exponentu. Dále detekuje nutnost normalizace výsledku obvodovými prostředky.
- Obvod negace a řízení signálů NEG, BL a Cin - během výpočtu je potřeba provádět negaci i mimo násobení Boothovým algoritmem. Pokud budou řídící signály $NEGACE$ a WAY v hodnotě 0, použijí se signály generované ze vstupních hodnot (při provádění Boothova algoritmu). Při $WAY = 1$ zajistíme generování signálů nezávisle na vstupu - vypne se blokování a negace. Pokud navíc bude $NEGACE = 1$, obvod BNEG provede negaci a vygeneruje se signál $C_{IN} = 1$ - tím se zajistí vytvoření druhého doplňku pro odčítání.



Obrázek 7.7: Blokové schéma aritmeticko-logické jednotky v pohyblivé řádové čárce

Činnost aritmeticko-logické jednotky je názorně zobrazena na vývojovém diagramu (obrázek 7.8). Jedná se pouze o princip a ne detailní popis činnosti jak tomu bylo u variant

v pevné řádové čárce. Význam jednotlivých stavů je následující:



Obrázek 7.8: Činnost ALU

Init
čekání na zadání vstupních dat
Nahrání MAY a EXPY
nahrání mantisy počáteční podmínky MAY přes SIR a ACC do SR a paměti a následně exponentu počáteční podmínky EXPY přes SIR a ACC do paměti
Nahrání MAH a EXPH
nahrání exponentu integračního kroku EXPH do SIR a sečtení s EXPY (uloženým v ACC z předešlého kroku), výsledek je exponent výsledného součinu, uložení tohoto exponentu do paměti a nahrání mantisy integračního kroku MAH do SIR
Boothovo násobení
$WAY = 0$, provádění klasického Boothova násobení - totožné se sériově-paralelní variantou integrátoru v pevné řádové čárce
Rozdíl exponentů
$WAY = 1$, do ACC a následně SIR uložíme z paměti exponent výsledného součinu, z paměti dále načteme EXPY a provedeme jejich rozdíl
Nahrání menší MA
dle znaménkového bitu je nahrána mantisa menšího operandu do SIR
Posuv MA
logický posuv mantisy vpravo
Součet násobení
sečtení mantisy výsledku DYp_i s předešlými součtu y_{i+1}, provedení ošetření přetečení a případnou normalizaci
Všechny členy
test dosažení požadované přesnosti (vypočítány všechny členy DYp_i Taylorovy řady)
Uložení součtu
uloží výsledný součet do paměti
Výsledek násobení
uložení výsledku násobení DYp_i do SR (mantisa) a ACC (exponent) a pokračování ve výpočtu $DY(p+1)_i$
Výsledek
příznak, že je již výsledek vypočítán a je možné ho přečíst

Druhou možností řešení integrátoru v pohyblivé řádové čárce je paralelní přístup s dvěma jednotkami, kde jedna provádí výpočty s mantisami a druhá s exponenty. Jednotka exponentu musí zajistit součet i rozdíl exponentů, inkrementaci a řízení aritmetického posuvu mantisy. Jednotka mantisy musí umožňovat násobení, sčítání, odčítání a aritmetické i logické posuvy. Obě jednotky musí být synchronizovány tak, aby bylo dosaženo správného výpočtu a tato synchronizace samozřejmě vyžaduje dodatečnou režii.

V rámci práce [Čam11] vznikl simulátor jehož cílem bylo zhodnocení výhodnosti a efektivity obou realizací. Paralelní přístup poskytuje urychlení ale na úkor vyššího počtu

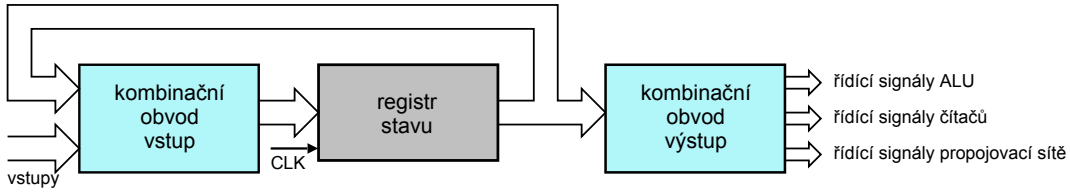
použitých prvků, potřebou dvou řadičů a většího počtu propojovacích sběrnic. Obvod tak má požadavky na dvojnásobnou plochu a na zdvojení jednotlivých datových sběrnic.

Je tedy výhodnější přístup postupného zpracování mantisy a exponentu v jedné společné jednotce. Zapojení více jednotek díky menšímu požadavku na prostor na čipu je méně problematické, nemusí se zde provádět synchronizace a celková cena bude také výrazně nižší.

7.3 Řadič - řídící jednotka

Co by měl řadič umět, je popsáno v kapitole 6.3. Při výpočtu je pevně dána posloupnost prováděných operací, proto se jedná o specializovaný řadič bez nutnosti dekódovat instrukce.

Řídící jednotku tvoří Moorův automat. Blokové schéma tohoto typu automatu je na obrázku 7.9. Výstup automatu a tedy řídící signály pro ovládání celého systému je dán



Obrázek 7.9: Blokové schéma zobrazující princip Moorova automatu

pouze stavem, ve kterém se nachází. V jakém stavu se automat nachází závisí na předcházejícím stavu a vstupních signálech.

K přepnutím z aktuálního do následujícího stavu je zapotřebí určitý časový okamžik daný zpožděním kombinačních obvodů určujících následující stav. Automaty, které pracují tímto způsobem, se označují asynchronní. Tento typ je náchylný na vznik hazardů a parazitních impulsů ve výstupní kombinační části. Po zavedení zpoždění reprezentovaného hodinovým signálem, který řídí zápis do registru uchovávající aktuální stav automatu, se těmto neplatným impulsům vyhneme. Tyto synchronizační impulsy určují okamžik změny vnitřního stavu automatu.

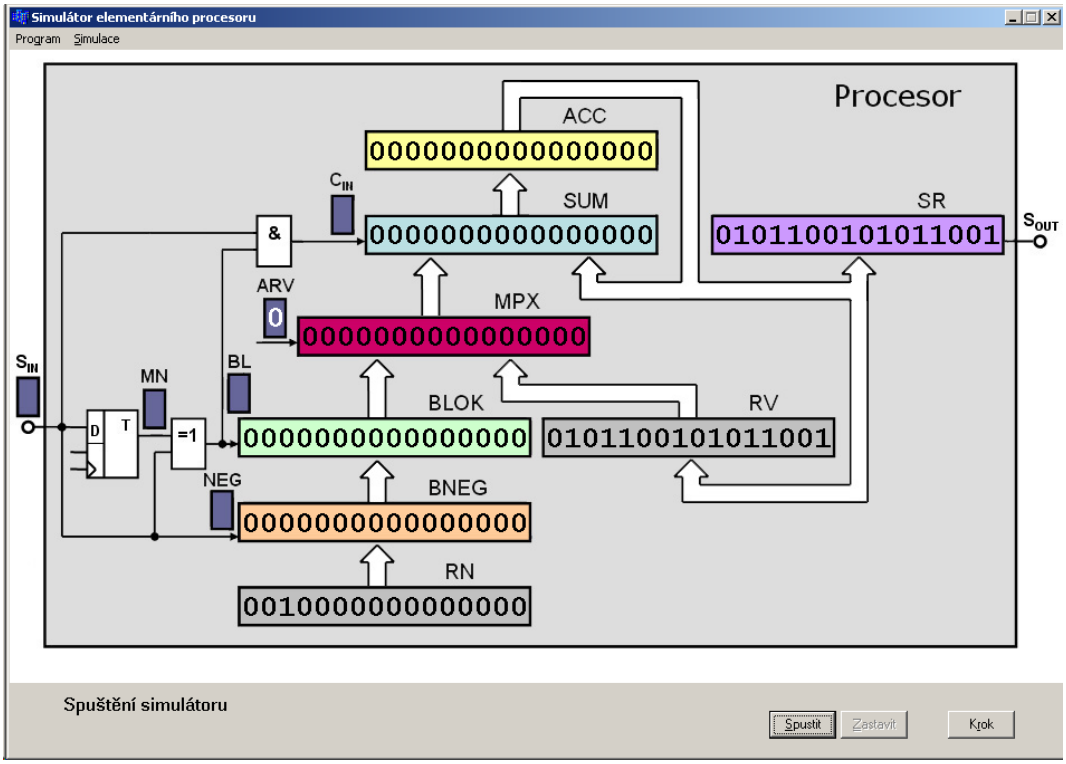
Detailní popis návrhu automatů je v [PP06, Kapitola 10: Sekvenční obvody].

7.4 Propojovací systém

Paralelní systém byl realizován se statickým propojením. Jednotlivé bloky byly propojeny podle řešeného problému. Návrh výkonné propojovací sítě může být námět na další pokračující práci v této problematice.

7.5 Programová realizace simulátoru procesoru

Představuji zde program umožňující věrně simulovat sériově-paralelní verzi integrátoru. Vznikl jako součást diplomové práce [Kra06]. Schopnost programu ukázat v kterémkoliv kroku obsahy všech registrů je velmi užitečná. Simulace probíhá po jednotlivých krocích popsaných v kapitole 7.1.2.



Obrázek 7.10: Zobrazení procesoru

7.5.1 Spuštění simulace

V základním menu, pod položkou *Simulace* se nachází ovládání vlastní simulace. Jsou zde na výběr následující možnosti:

- Procesor: Spustí simulaci jednoprocesorového systému řešící problém pomocí Taylorovy metody. Řeší rovnici 2.13
- Rychlost: Nastavení rychlosti animace simulace
- Spustit: Spustí automatické provádění kroků simulace
- Zastavit: Zastaví automatické provádění kroků simulace
- Krok: Provede jeden krok simulace

7.5.2 Nastavení počátečních hodnot experimentu

Po spuštění Simulace (*Simulace* → *Procesor*) se zobrazí okno seznamující se základy nutnými ke správnému pochopení animace. Může se zatrhnout, aby se po každém spuštění nezobrazovalo. Potom již následuje okno, ve kterém se nastavují počáteční podmínky simulace. Jsou to hodnoty: Počáteční podmínka (y_0), Integrační krok (h) a Přesnost (přesnost výpočtu).

Je nutno znovu uvést, že počáteční podmínka a integrační krok se zadávají v pevné řádové čárce. Z toho vyplývá, v jakém rozmezí mohou být čísla zadána.

7.5.3 Průběh simulace

Po nastavení počátečních podmínek se zobrazí vlastní prostředí simulace (obrázek 7.10). V tomto případě je nastavení $y_0 = 0,69805$ a $h = 0,25$.

Při následné simulaci by ovládání mohlo vypadat tak, že po zmáčknutí některého tlačítka se provede výpočetní krok a znovu se čeká na další stisknutí. Takto ovládá program tlačítko *Krok*. Protože by ale bylo nevýhodné neustále po každé změně znovu a znovu potvrzovat další krok zmáčknutím tlačítka, je simulační program vybaven i tlačítkem *Spustit* a *Zastavit*. Jejich účel však není spustit nějaké přehrávání již vygenerovaných postupů. Pouze zobrazí, v závislosti na nastavené rychlosti simulace, celý simulační proces. Ten je možno kdykoliv tlačítkem *Zastavit* přerušit a znovu pokračovat jak tlačítkem *Krok* tak *Spustit*. Rád bych ještě poukázal na text vlevo pod zobrazeným procesorem. Zde se zobrazuje, ve kterém stavu se procesor zrovna nachází.

7.6 Technická realizace

Byl navržen a realizován celý paralelní systém, společně se všemi verzemi aritmeticko-logických jednotek (integrátorů) v pevné řádové čárce. Implementace proběhla v jazyce VHDL, který slouží pro popis hardware. Detailnější informace o tomto jazyku, ze kterých jsem čerpal při své práci, jsou v [Dou03], [Hwa06]. Vytvořený systém byl odsimulován (simulátor VHDL - *ModelSim* [Gra12]) a následně byla provedena syntéza a zápis do hradlových polích FPGA.

V průběhu prací na projektu se objevil požadavek školitele na implementaci algoritmu Taylorovy řady do školního přípravku FTTkit [FIT11] a zařazení tohoto přípravku do výuky předmětu Prvky počítačů (IPR). Z těchto důvodů je výrazná část implementace věnována využití přípravku FITkit.

Tento stručný popis byl převzat z [FIT11].

> FITkit je samostatný hardware, který obsahuje výkonný mikrokontrolér s nízkým příkonem, hradlové pole FPGA (Field Programmable Gate Array) kategorie Spartan3 a řadu periferií. Důležitým aspektem je využití pokročilého reprogramovatelného hardwaru na bázi hradlových polí FPGA jenž lze, podobně jako software na počítači, neomezeně modifikovat pro různé účely dle potřeby - uživatel tedy nemusí vytvářet nový hardware pro každou aplikaci znovu.

7.6.1 Porovnání jednoprocesorových systémů

Následuje srovnání implementace paralelních systémů obsahující jednotlivé varianty integrátorů. Sledovanými parametry byla obsazenost čipu a dosažitelná rychlost výpočtu v závislosti na použité datové šířce operandů.

Ve všech případech systém řešil homogenní lineární diferenciální rovnice 1. řádu s konstantními koeficienty - rovnice (2.13), blokově je tato rovnice znázorněna na obrázku 2.1.

Implementované systémy obsahují:

- integrátor
- řadič tvořený Moorovým automatem (automat řídí činnost popsanou vývojovými diagramy uvedenými u každé aritmeticko-logické jednotky), počet členů Taylorovy řady nastaven na 8
- vnitřní paměť pro 8 podílů integračního kroku
- statickou propojovací síť
- řadič komunikačního systému FITkitu - SPI

Závislosti obsazení plochy čipu a rychlosti výpočtu na šířce dat pro jednotlivé verze paralelních systémů jsou v tabulkách 7.1, 7.2 a 7.3.

V tabulkách je také zobrazena doba násobení násobičky u paralelně-paralelní verze a u všech verzí je uvedena doba potřebná k operaci sčítání - paralelní sčítačkou u sériově-paralelní verze a sériovou (jednobitovou) sčítačkou u sériově-sériové verze systému. Doba výpočtu ukazuje jak dlouho trvá výpočet jednoho členu Taylorovy řady DYp_i. Doba výpočtu je získána dosazením do rovnic (6.1), (6.2) a (6.3).

První testovaný výpočetní systém obsahuje paralelně-paralelní verzi integrátoru. Základem je paralelní násobička, která vznikla automatickou syntézou, protože čip neobsahuje integrované násobičky. Integrátor je propojen paralelním přímým propojením a data jsou do něho nahrána pomocí paralelní sběrnice. Obdržená data jsou zobrazena v tabulce 7.1.

Tabulka 7.1: Implementace paralelně-paralelního systému

šířka dat [bit]	**16**	**32**	**64**
obsazená plocha [%]	14	25	154
doba násobení [ns]	8	15	28
doba sčítání [ns]	6	7	9
doba výpočtu [ns]	14	22	37

Druhý výpočetní systém je založen na sériově-paralelní verzi integrátoru. Integrátor je propojen sériově a data jsou do něho nahrána také přes paralelní sběrnici jako v předchozí variantě. Obdržená data jsou zobrazena v tabulce 7.2.

Tabulka 7.2: Implementace sériově-paralelního systému

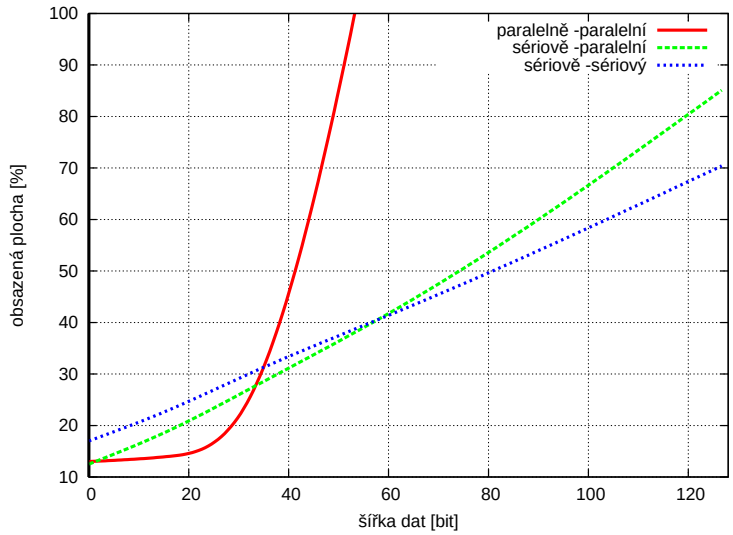
šířka dat [bit]	**16**	**32**	**64**	**128**
obsazená plocha [%]	19	27	44	86
doba sčítání [ns]	6	7	9	13
doba výpočtu [ns]	102	231	585	1677

Jádro posledního výpočetního systému obsahuje sériově-sériovou verzi integrátoru. Integrátor je propojen sériově jako v sériově-paralelní verzi a data jsou do něho nahrána sériovými sběrnicemi. Obdržená data jsou zobrazena v tabulce 7.3.

Tabulka 7.3: Implementace sériově-sériového systému

šířka dat [bit]	**16**	**32**	**64**	**128**
obsazená plocha [%]	23	30	43	71
doba sčítání [ns]	4	4	4	4
doba výpočtu [ns]	1088	4224	16640	66048

Graf závislosti obsazení plochy čipu na šířce dat je na obrázku 7.11. Na obrázku 7.12 je zobrazena doba provedení výpočtu jednoho členu Taylorovy řady jednotlivými systémy.



Obrázek 7.11: Obsazenost plochy implementovaných paralelních systémů

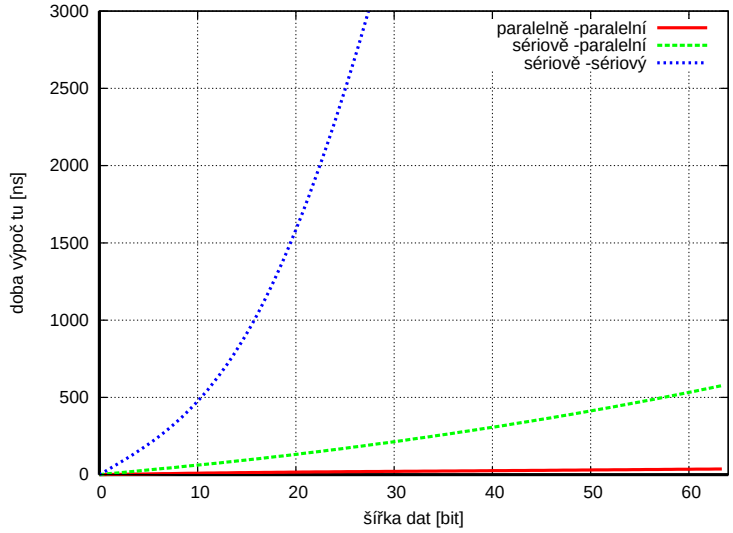
Jak je vidět, sériově-sériová varianta má očekávaný malý nárůst velikosti při nárůstu šířky slova, ale rychlost výpočetního systému je velmi malá. Největší výhodou této varianty je, že zapojení aritmetické jednotky nezávisí na šířce slova. Nárůst obsazení plochy je tedy způsoben pouze rozšířením vnitřních registrů.

V případě sériově-paralelní varianty je nárůst plochy oproti sériově-sériové variantě malý, protože obsahuje navíc jen paralelní sčítačku a proto poměr cena/výkon se zdá být nejlepší.

Paralelně-paralelní verze najde uplatnění jen v případě nutnosti velmi rychlého výpočtu a při použití čipu, který již obsahuje integrované násobičky. Nebude tedy nutné provádět syntézu násobiček a zabírat tak podstatnou část plochy čipu, která půjde využít na zbytek systému. Velká obsazenost plochy čipu je také dána paralelním propojením mezi jednotkami, které při větším počtu jednotek bude velmi náročné.

Dále jsem se zaměřil na zobrazení toho, jaké přesnosti lze dosáhnout v závislosti na datové šířce použité aritmetiky. Byla řešena opět stejná rovnice (2.13) jako v předchozím případě a integrační krok h byl zvolen $0,125$.

Tabulka 7.4 prezentuje, jakého řádu metody lze dosáhnout při použití jednotlivých datových šířek (16, 32, 64 a 128 bitů). Tentokrát je tabulka platná pro všechny typy integrátorů, protože přesnost nezávisí na tom, zda byla použita sériová či paralelní aritmetika. Ve



Obrázek 7.12: Doba výpočtu implementovaných paralelních systémů

všech případech jsou dosaženy stejné výsledky.

Tabulka 7.4: Dosažený řád metody

šířka dat [bit]	**16**	**32**	**64**	**128**
řád metody ORD [-]	3	7	11	21

Podle očekávání je vidět, že s vyšším počtem bitů roste i dosažená přesnost. Např. při použití 16 bitové aritmetiky je možno využít pouze tři členy Taylorovy řady DYp_i, ostatní jsou nulové a na výsledek nemají žádný vliv.

7.6.2 Porovnání víceprocesorových systémů

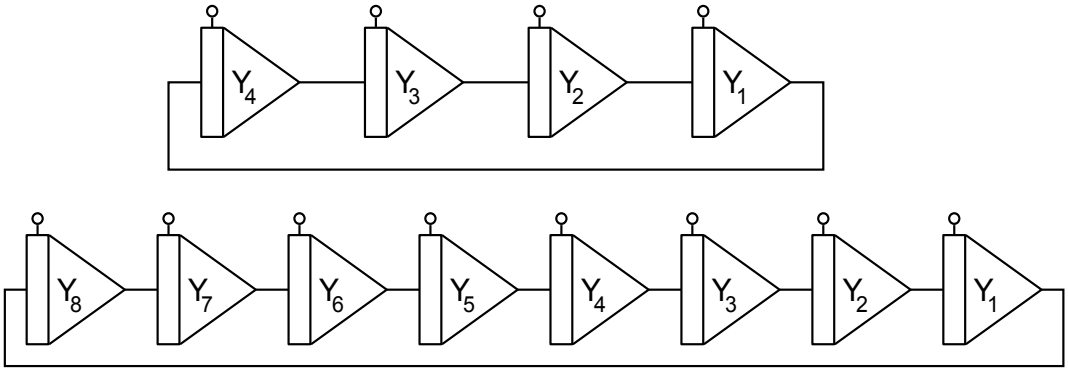
Pokud chceme jednotlivé systémy vzájemně porovnat, musíme tak učinit při řešení stejných soustav diferenciálních rovnic. Uvažujme jednoduché výpočetní schéma, které můžeme dále paralelizovat rozdělením práce mezi dalších N integrátorů, čímž by se čas potřebný pro výpočet zkrátil v ideálním případě Nkrát.

Ke srovnání jednotlivých verzí paralelních systémů mezi sebou použijeme řešení soustavy diferenciálních rovnic (7.2)–(7.5), která vznikne aplikací tvořících diferenciálních rovnic na rovnici 7.1.

$$y = \sin(t) \tag{7.1}$$

$$\begin{aligned} y_1' &= y_2 \qquad y_1(0) = 0 && (7.2)\\ y_2' &= y_3 \qquad y_2(0) = 1 && (7.3)\\ y_3' &= y_4 \qquad y_3(0) = 0 && (7.4)\\ y_4' &= y_1 \qquad y_4(0) = -1 && (7.5) \end{aligned}$$

Tato soustava se dá dále rozšiřovat a vznikne tak soustava 8, 16, 32,... rovnic. Odpovídající blokové schéma je na obrázku 7.13. Je zde také zobrazen princip, jak vznikne soustava osmi diferenciálních rovnic.



Obrázek 7.13: Blokové schéma řešené soustavy

Tabulka 7.5 ukazuje obsazenost čipu paralelními systémy, které obsahují 1–32 jednovstupých aritmeticko-logických jednotek různé datové šířky. Jsou zde potvrzeny výsledky z předchozího srovnání. Navíc je vidět, že s vyšším počtem integrátorů se více zvyšuje rozdíl mezi obsazeností čipu sériově-paralelní a sériově-sériovou verzí. To je způsobeno použitím paralelních sběrnic sloužícím k nahrání počátečních podmínek a integračních kroků do integrátorů.

Tabulka 7.5: Obsazenost čipu paralelním systémem s více integrátory

Počet	**šířka dat [bit]**										
ALU	**16**	**32**	**64**	**16**	**32**	**64**	**128**	**16**	**32**	**64**	**128**
1	14%	25%	154%	19%	27%	44%	86%	23%	30%	43%	71%
2	17%	99%	-	23%	37%	64%	117%	25%	33%	48%	82%
4	25%	-	-	34%	56%	101%	-	28%	38%	59%	99%
8	-	-	-	57%	95%	-	-	35%	50%	79%	-
16	-	-	-	99%	-	-	-	50%	72%	115%	-
32	-	-	-	-	-	-	-	78%	110%	-	-
	paralelně-paralelní			**sériově-paralelní**				**sériově-sériový**			

Pro srovnání výkonnosti specializovaného paralelního výpočetního systému s klasickým výpočtem na univerzálním procesoru, předpokládejme, že integrace jednoho integrátoru spotřebuje 10 taktů procesoru. V těchto zmíněných deseti taktech je obsaženo načtení vstupní hodnoty integrátoru, její zpracování, vynásobení integračním krokem, načtení předcházející hodnoty integrátoru a její sečtení s výsledkem násobení a následné uložení zpět do paměti.

Aby specializovaný systém, ve kterém se provádí numerická integrace paralelně ve všech integrátorech, byl stejně rychlý jako výpočet na univerzálním procesoru, musí být v systému

tolik integrátorů, aby platil následující vztah

$$N = \frac{t_S}{10 \cdot t_P}$$

Kde t_S je perioda výpočtu specializovaného systému a t_P perioda hodin univerzálního procesoru. Např. při použití sériově-paralelního 32bitového systému a procesoru s taktovacím kmitočtem 1 GHz bude výpočet u obou stejně rychlý při použití 23 integrátorů.

7.7 Představení systémů řešící obdobnou problematiku

Nedávné výzkumy [Mac07] se zabývaly způsoby, jak znovu použít výpočetní metody analogových počítačů, zvláště v případech, kdy je požadována velká rychlost výpočtu. Stejný přístup k řešení diferenciálních rovnic zvolili autoři článku [FTCK09]. Rovnice se řeší v systému obdobným jako analogový počítač. Jednotlivé prvky analogového počítače (sčítačky, násobičky, integrátory) jsou nahrazeny digitálními diskrétními ekvivalenty. Díky flexibilitě a možnosti rekonfigurace jsou použity pro implementaci čipy FPGA. Můj přístup jsem prezentoval v článku [18].

V literatuře [WM06] je popsán návrh a implementace numerického integrátoru na FPGA čipu. Řešení obyčejných diferenciálních rovnic se provádí metodou Runge-Kutta 4. řádu. K výpočtu se také využívají dvě základní operace: sčítání a násobení. Výpočet ale probíhá v pohyblivé řádové čárce. Čísla jsou konkrétně uložena ve standardu IEEE 754, jednoduché přesnosti - tedy na 32 bitech. Systém byl implementován na FPGA čipu XC2V6000, což je čip kategorie Virtex2. Výpočet jedné integrace trval asi 3800 ns.

Zcela jiný přístup představuje spolupráce s programovým celkem Matlab/Simulink. Matlab dovoluje prováděné výpočty akcelerovat pomocí FPGA čipů a to tak, že provede vygenerování VHDL kódu ze simulovaného modelu a tento kód lze následně nahrát na připojený FPGA čip. Tímto přístupem se více zabývá např. článek [BHN+04]. Je ale otázkou, zde bude tato možnost efektivnější než návrh vlastního specializovaného systému.

Článek [iDZD10] se zabývá vyhodnocením výkonnosti frameworku OpenCL pro výpočet koncentrace v ovzduší, jehož model je popsaný advekčně-difúzní rovnicí (parciálně diferenciální rovnice). Daná parciální diferenciální rovnice je převedena na soustavu obyčejných diferenciálních rovnic prvního řádu, které jsou následně řešeny numerickou metodou Runge-Kutta. 4. řádu. V článku jsou prezentovány výsledky testů na grafických kartách ATI a nVidia a porovnány s řešením na klasickém procesoru. Použití metody vyšího řádu brání přesnost, kterou jsme schopni na GPU architektuře dosáhnout. Z tohoto důvodu není tato architektura vhodná pro výpočty metodou Taylorovy řady.

7.8 Shrnutí

V této kapitole jsem se zaměřil na detailnější popis implementace paralelního systému ve variantách paralelně-paralelní, sériově-paralelní a sériově-sériové. Implementace byla provedena v jazyce VHDL a následně byla provedena syntéza do hradlových polí FPGA na přípravku FITkit.

Na základě implementace bylo provedeno srovnání paralelních systémů obsahující jednotlivé varianty aritmeticko-logických jednotek. Sledovanými parametry byla obsazenost čipu a rychlost výpočtu v závislosti na datové šířce. Jako nejlepší se jeví použít sériově-paralelní variantu.

Kapitola 8

Využití specializovaného paralelního systému

Na závěr je v této kapitole demonstrováno využití navrženého paralelního systému. Abychom mohli plně využít možnosti zobrazení čísel v pevné řádové čárce, musíme provést transformaci proměnných - normalizaci. Tím dosáhneme, že se nám počítané hodnoty nedostanou mimo rozsah zobrazení čísel. V neposlední řadě jsou součástí této kapitoly ukázky využití paralelního systémů.

8.1 Zobrazování a transformace proměnných

Pro řešení konkrétních úloh se zadanými konkrétními číselnými hodnotami proměnných, konstant a parametrů musíme obvod doplnit řadou dalších úprav. Závisle proměnné veličiny mají nejrůznější fyzikální jednotky a rozměr a v řešeném intervalu nabývají různých velikostí. Abychom mohli tyto závisle proměnné veličiny zobrazit v podporovaném rozsahu, musíme provést jejich transformaci - normalizaci.

8.1.1 Normalizace

Normu N_y závisle proměnné můžeme definovat jako hodnotu větší nebo rovnu maximální velikosti z absolutní hodnoty proměnné y v uvažovaném intervalu. Normalizovaná proměnná je definována jako proměnná dělená normou. Normalizovanou proměnnou píšeme do závorek ve tvaru $(\frac{y}{N_y})$. Normalizované proměnné jsou veličinami relativními, jsou bezrozměrné a v absolutní hodnotě jsou vždy menší, nejvýše rovny jedné.

Skutečné provedení transformace závisle proměnných metodou normalizace si ukážeme na příkladu transformace rovnice (8.1).

$$y'' + 6{\cdot}y' + 4{\cdot}y = 0 \qquad y(0) = 5, \quad y'(0) = 0 \tag{8.1}$$

Rovnici (8.1) upravíme metodou snižování řádu derivace a získáme diferenciální rovnice prvního řádu:

$$y' = \int (-6{\cdot}y' - 4{\cdot}y)\,\mathrm{d}t \tag{8.2}$$

$$y = \int y\,\mathrm{d}t \tag{8.3}$$

Stanovíme normy:

$$N_y \geq |y|_{MAX} = 5 \tag{8.4}$$

$$N_{y'} \geq |y'|_{MAX} = 10 \tag{8.5}$$

Normy z rovnic (8.4) a (8.5) dosadíme do rovnic (8.2), (8.3) a obdržíme (8.6), (8.7).

$$\begin{aligned} \left(\frac{y'}{N_{y'}}\right) N_{y'} &= \int \left(-6\left(\frac{y'}{N_{y'}}\right) N_{y'} - 4\left(\frac{y}{N_y}\right) N_y\right) \mathrm{d}t \\ \left(\frac{y'}{10}\right) 10 &= \int \left(-6\left(\frac{y'}{10}\right) 10 - 4\left(\frac{y}{5}\right) 5\right) \mathrm{d}t \end{aligned} \tag{8.6}$$

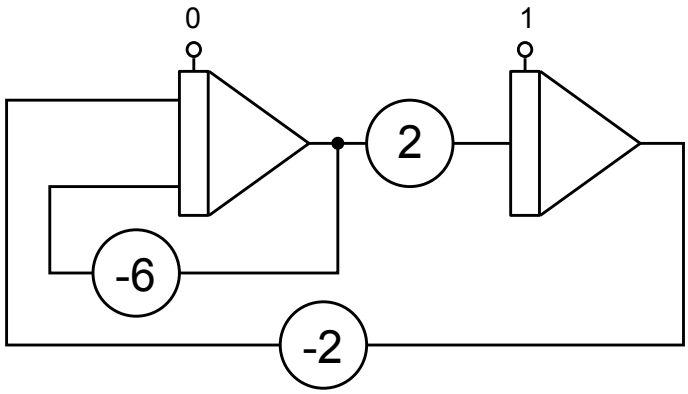
$$\begin{aligned} \left(\frac{y}{N_y}\right) N_y &= \int \left(\frac{y'}{N_{y'}}\right) N_{y'} \,\mathrm{d}t \\ \left(\frac{y}{5}\right) 5 &= \int \left(\frac{y'}{10}\right) 10 \,\mathrm{d}t \end{aligned} \tag{8.7}$$

Rovnice nyní upravíme tak, že celou rovnici (8.6) podělíme číslem 10 a rovnici (8.7) podělíme číslem 5.

$$\left(\frac{y'}{10}\right) = \int \left(-6\left(\frac{y'}{10}\right) - 2\left(\frac{y}{5}\right)\right) \mathrm{d}t \tag{8.8}$$

$$\left(\frac{y}{5}\right) = \int \left(\frac{y'}{10}\right) 2 \,\mathrm{d}t \tag{8.9}$$

Z rovnic (8.8) a (8.9) můžeme nakreslit programové schéma uvedené na obrázku 8.1. Chceme-li převést normalizované veličiny zpět, musíme je vynásobit normou.



Obrázek 8.1: Obvodové schéma rovnice po provedení normalizace

8.1.2 Odhad maximálních hodnot

Pro stanovení norem je zapotřebí znát nebo umět odhadnout maximální hodnoty. K optimální velikosti norem dojdeme zpravidla experimentálně. Účelem je, aby byl využit celý rozsah zobrazení čísel. Byla-li některá norma příliš malá, dojde k překročení rozsahu. Naopak příliš velká norma způsobí nevyužití celého rozsahu zobrazení a díky tomu bude větší chyba řešení. Pro informativní odhad maximálních hodnot, můžeme použít některého ze způsobů uvedených dále.

Odhad ze znalosti řešeného problému

Zabýváme-li se určitým technickým problémem, můžeme z jeho znalosti snadno odhadnout maximální hodnoty celé řady veličin. Odhady závisí na schopnostech a zkušenostech uživatele.

Metoda rovných koeficientů

Odhad maximálních hodnot u systémů popsaných diferenciální rovnicí vyššího řádu můžeme provést z odezvy na jednotkový skok při nulových počátečních podmínkách. Zvolíme-li vstupní funkci $z = 1$ (jednotkový skok), můžeme provést odhad norem následujícím postupem.

$$a_3 \cdot y''' + a_2 \cdot y'' + a_1 \cdot y' + a_0 \cdot y = b_0 \cdot z \quad | = b_0 \left(\frac{z}{1}\right)$$

Odhadnuté normy budou rovny

$$N_y = \frac{b_0}{a_0}, \quad N_{y'} = \frac{b_0}{a_1}, \quad N_{y''} = \frac{b_0}{a_2}, \quad N_{y'''} = \frac{b_0}{a_3}$$

Bude-li mít řešení kmitový charakter, zvolíme

$$N_y = (1,5 - 2)\frac{b_0}{a_0}$$

Tento způsob je jednoduchý a vede rychle k prvnímu orientačnímu odhadu.

Odhady pro kmitové soustavy

Obsahuje-li řešení netlumené nebo málo tlumené kmity, můžeme poměr amplitud jednotlivých derivací odhadnout ze vztahů pro kmitavý pohyb

$$\begin{array}{rcllcl} y & = & A \cdot \sin \omega t & \Rightarrow & |y|_{MAX} & \doteq & A \\ y' & = & A \cdot \omega \cdot \cos \omega t & \Rightarrow & |y'|_{MAX} & \doteq & A \cdot \omega \\ y'' & = & -A \cdot \omega^2 \cdot \sin \omega t & \Rightarrow & |y''|_{MAX} & \doteq & A \cdot \omega^2 \end{array}$$

8.1.3 Transformace času

Čas, ve kterém probíhá řešení úlohy nazýváme strojový čas a označujeme ho T. Řešení úlohy v paralelním systému probíhá jen v určitém časovém intervalu a jen s určitou rychlostí. Avšak úlohy, které máme řešit mají velmi rozdílné rychlosti i délky řešení (např. poločasy rozpadu radioaktivních částic jsou v rozsahu zlomků vteřiny až do desítky roků). Aby bylo možné řešit různé úlohy, musíme provést tzv. transformaci času [BHSL82].

Děje probíhají v reálném čase, který označujeme t. Řešení na modelu probíhá ve strojovém čase T. Vztah mezi strojovým časem a reálným je dán rovnicí 8.10

$$M_t = \frac{T}{t} \tag{8.10}$$

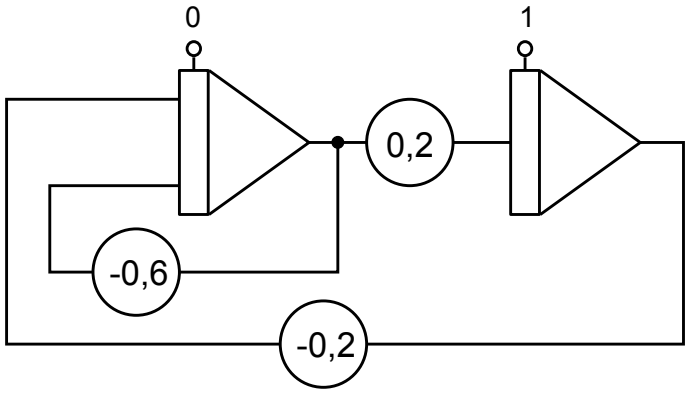
kde M_t je měřítko času.

Nedochází ke změně počátečních podmínek. Vlastní úprava spočívá ve změně koeficientů jednotlivých integrátorů dané měřítkem M_t. Rovnici není nutno přepracovávat.

Nyní provedeme transformaci času na rovnicích (8.8) a (8.9), které nám vznikly po provedení metody normalizace. Zvolíme měřítko času $M_t = 10$, a upravíme koeficienty rovnic.

$$\begin{aligned}
\left(\frac{y'}{10}\right) &= \frac{1}{M_t}\int\left(-6\left(\frac{y'}{10}\right) - 2\left(\frac{y}{5}\right)\right)\mathrm{d}T \\
\left(\frac{y'}{10}\right) &= \int\left(-0,6\left(\frac{y'}{10}\right) - 0,2\left(\frac{y}{5}\right)\right)\mathrm{d}T \qquad (8.11) \\
\left(\frac{y}{5}\right) &= \frac{1}{M_t}\int\left(\frac{y'}{10}\right)2\,\mathrm{d}T \\
\left(\frac{y}{5}\right) &= \int\left(\frac{y'}{10}\right)2\,\mathrm{d}T \qquad (8.12)
\end{aligned}$$

Z rovnic (8.11) a (8.12) můžeme nakreslit výsledné programové schéma, které je uvedené na obrázku 8.2. Dosáhli jsme toho, že hodnoty koeficientů integrátorů nejsou mimo rozsah zobrazení.



Obrázek 8.2: Obvodové schéma rovnice po provedení normalizace a transformace času

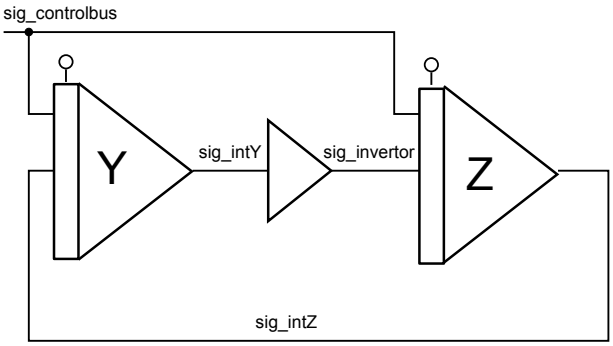
8.2 Využití přípravku ve výuce

Paralelní systém, jehož návrh je představen v této práci, je možné využít také ve výuce. Toto využití je demonstrováno jako součást projektu FRVŠ FR2681/2010/G1. Projekt si klade za cíl zkvalitnit výuku a rozšířit možnosti praktického ověřování znalostí získávaných na přednáškách v předmětech Prvky počítačů (IPR), Vysoce náročné výpočty (VNV) a Teorie obvodů (ITO). Hlavním rozšířením je návrh a vytvoření nového moderního prototypu laboratorního přípravku, který se bude skládat ze základních analogových obvodů a ekvivalentních číslicových obvodů pro jejich vzájemné porovnání.

V této podkapitole je uveden konkrétní příklad, který je součástí zmíněného projektu. Jedná se o příklad generování funkcí $\sin t$ a $\cos t$. Rovnice pro zadaný problém mají tvar:

$$\begin{aligned}
y' &= z \qquad y(0) = 0 \\
z' &= -y \qquad z(0) = 1
\end{aligned}$$

Odpovídající výpočetní schéma je na obrázku 8.3.



Obrázek 8.3: Výpočetní schéma generování funkcí $\sin t$ a $\cos t$

Definice a propojení jednotlivých komponent je naznačeno níže uvedeným VHDL kódem.

Komponenta integrátor - provádí numerickou integraci metodou Taylorovy řady

```
component alu is
   generic (bits_number: integer);
   port (
      CONTROL: in std_logic_vector(9 downto 0);              -- řídící sběrnice
      INPUT:   in std_logic;                                 -- vstup integrátoru
      DATAY0:  in std_logic_vector(bits_number-1 downto 0);  -- počáteční podmínka
      VE:      out std_logic;                                -- výsledek je aktivní
      RESULT:  out std_logic_vector(bits_number-1 downto 0)  -- hodnota výsledku
   );
end component;
```

Komponenta invertor - vytváří dvojkový doplněk vstupního čísla

```
component complement is
   port (
      CLK:      in std_logic;   -- hodinový signál
      CLR:      in std_logic;   -- nulovací signál
      DATA_IN:  in std_logic;   -- vstup
      DATA_OUT: out std_logic   -- výstup
   );
end component;
```

Propojení komponent

Propojení je zobrazeno na obrázku 8.3. Toto schéma neobsahuje všechny propojovací cesty, jen ty nejdůležitější. Jednotlivé komponenty jsou mezi sebou propojeny pomocí signálů:

- `sig_intY` - výstup z integrátoru Y sloužící jako vstup invertoru
- `sig_intZ` - výstup z integrátoru Z sloužící jako vstup integrátoru Y
- `sig_invertor` - výstup invertoru sloužící jako vstup integrátoru Z
- `sig_controlbus` - řídící sběrnice jejíž signály jsou generovány řadičem a řídí výpočet integrátorů

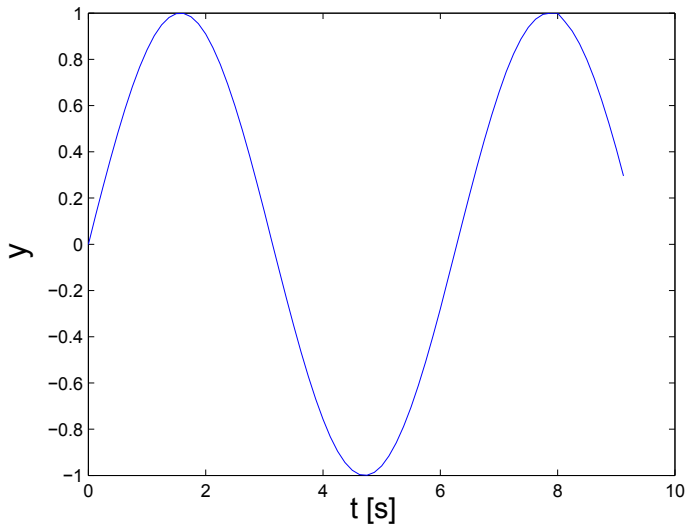
```
Integrator_Y: alu
   generic map (bits_number => 32)
   port map (
```

```
      CONTROL => sig_controlbus,     -- napojení na řídící sběrnici (řadič)
      INPUT   => sig_invertor,       -- napojení na výstup invertoru
      DATAY0  => X"00000000",        -- počáteční podmínka v hexadecimální soustavě -
          0
      VE      => sig_veY,            -- signál označující vypočítaný výsledek
      RESULT  => sig_intY            -- výstup integrátoru - hodnota sin(t)
   );

Integrator_Z: alu
   generic map (bits_number => 32)
   port map (
      CONTROL => sig_controlbus,     -- napojení na řídící sběrnici (řadič)
      INPUT   => sig_intY,           -- napojení na výstup integrátoru Y
      DATAY0  => X"7FFFFFFF",        -- počáteční podmínka v hexadecimální soustavě -
          1
      VE      => sig_veZ,            -- signál označující vypočítaný výsledek
      RESULT  => sig_intZ            -- výstup integrátoru - hodnota cos(t)
   );

Invertor: complement
   port (
      CLK        => sig_clk,         -- hodinový signál
      CLR        => sig_clr,         -- nulovací signál
      DATA_IN    => sig_intZ,        -- napojení na výstup integrátoru Z
      DATA_OUT   => sig_invertor     -- výstup invertoru
   );
```

Výstup řešení je na obrázku 8.4.



Obrázek 8.4: Průběh funkce $\sin t$ získaný z FITkitu

8.3 Regulace a řízení reálných soustav

Zcela nově se otevírá oblast implementace Taylorovy řady v teorii regulace a konkrétně v řízení v reálném čase (real-time control and simulation). Informace z oblasti teorie řízení lze čerpat z [DB08].

Implementace Taylorovy řady v teorii řízení vznikla na základě konzultací a aktivních účastí na konferencích ADIUS společnosti Applied Dynamics International - jako výsledek

spolupráce vznikl společný příspěvek na konferenci Eurosim 2007 Ljubljana [3]. Podobná problematika byla také prezentována na konferenci *10th International Conference on Computer Modeling and Simulation* [4].

8.4 Shrnutí

Výpočty v pevné řádové čárce mají největší výhodu oproti operacím v pohyblivé řádové čárce v tom, že jsou rychlejší a šetří místo na čipu. Abychom ale mohli naplno využít možnosti pevné řádové čárky, musíme zajistit aby byly hodnoty zobrazeny v podporovaném rozsahu - musí se provést transformace proměnných (normalizace) a případně transformace času.

Hlavní využití paralelního systému je demonstrováno na skutečné výukové aplikaci. Jedná se o náhradu analogového laboratorního přípravku využívaným ve výuce. Další možnou aplikací je teorie řízení. Reálný systém je řízen modelem, jehož chování by mělo být shodné s modelovaným reálným systémem. Model je založen na popisu pomocí Taylorovy řady.

Kapitola 9

Závěr

Předložená disertační práce se zabývá návrhem specializovaného paralelního systému, který provádí numerický výpočet diferenciálních rovnic pomocí metody Taylorovy řady. Celá práce bude nyní shrnuta v několika krocích. Nejprve bude představen použitý přístup k řešení, dále dosažené výsledky a konečně naznačení možností dalšího výzkumu.

9.1 Zvolený přístup k řešení a dosažené výsledky

Při simulaci spojitých systémů se využívá popis pomocí diferenciálních rovnic. Vzniklé diferenciální rovnice mohou být převedeny na blokový diagram, který je chápán jako paralelně pracující systém. Jednotlivé prvky blokového diagramu (sčítačky, násobičky, integrátory) jsou implementovány digitálními diskrétními ekvivalenty. Díky flexibilitě a možnosti rekonfigurace jsou pro implementaci použity čipy FPGA. Základní stavební prvek celého systému je integrátor, který provádí numerickou integraci pomocí metody Taylorovy řady. Důležitou náplní této disertační práce je proto i numerická integrace. Bylo dokázáno, že metoda Taylorovy řady se vyznačuje vysokou přesností výpočtu a také vysokým stupněm paralelismu.

Díky metodice tvořících diferenciálních rovnic, která je popsána v této práci, lze metodou Taylorovy řady řešit soustavy homogenních diferenciálních rovnic, nehomogenních diferenciálních rovnic, diferenciálních rovnic s proměnnými koeficienty a soustavy nelineárních diferenciálních rovnic.

Bylo provedeno srovnání efektivnosti metody Taylorovy řady s obecně velmi používanými metodami Runge-Kutta druhého a čtvrtého řádu. Ze srovnání vyplývá, že při výpočtu pomocí Taylorovy řady je počet výpočetních operací menší než metodou Runge-Kutta odpovídajícího řádu. S použitím vyššího řádu je tento rozdíl ještě výraznější.

Všechny výpočetní jednotky provádí stejný výpočet a jsou řízeny z jedné řídící jednotky. Proto byla zvolena paralelní architektura typu SIMD. Podle způsobu provádění výpočetních operací v aritmeticko-logických jednotkách a způsobu komunikace mezi nimi, se rozdělují na paralelně-paralelní, sériově-paralelní a sériově-sériové. Je použito uložení čísel v pevné řádové čárce, protože výpočty v této aritmetice jsou rychlejší a mají menší nároky na hardwarové zdroje.

Specializovaný paralelní systém byl implementován na přípravku FITkit, který obsahuje FPGA čip typu Spartan 3 od společnosti Xilinx. Bylo provedeno srovnání implementovaných paralelních systémů využívající jednotlivé varianty integrátorů. Sledovanými parametry byla obsazenost čipu a rychlost výpočtu v závislosti na datové šířce. Jako nejlepší se jeví použití sériově-paralelní varianty.

Největší výhoda této varianty je sériové propojení. Tento typ propojení umožňuje rychlejší přenos dat a několikanásobně šetří použité prvky na FPGA čipu oproti paralelní variantě. Propojovací síť je příliš náročná. Nevýhoda paralelní varianty je i nutnost realizovat násobičku, která je nejsložitější prvek celé výpočetní jednotky. Tento nedostatek ale odpadá při použití výkonnějšího FPGA čipu, který již obsahuje integrované násobičky.

Využití paralelního systému je ukázáno na skutečné výukové aplikaci, která vznikla jako podpora laboratorní výuky předmětu prvky počítačů. Další možnou aplikací je teorie řízení. Reálný systém je řízen modelem, jehož chování by mělo být shodné s modelovaným reálným systémem. Model je založen na popisu pomocí Taylorovy řady.

9.2 Shrnutí vlastního vědeckého přínosu

- Taylorova řada není v oblasti numerických výpočtů příliš využívána, protože získání vyšších derivací bývá obtížné. Zde představená metodika tvořících diferenciálních rovnic však transformuje zadaný systém na novou soustavu rovnic, ve kterých se vyskytují pouze základní matematické operace: sčítání, odčítání, násobení a dělení. Tento postup je jednoduše algoritmizovatelný a výpočet vyšších derivací v tomto novém systému již není problém. Tyto výsledky byly potvrzeny a prezentovány v několika vědeckých článcích. Taylorova řada byla implementována v simulačním jazyku TKSL.

- Taylorova řada také umožňuje paralelní zpracování a proto byl navržen paralelní systém pro urychlení výpočtu. Celý systém je obdobou analogového počítače, kde jednotlivé prvky jsou implementovány číslicově. Výpočetní jednotka (integrátor) je velmi jednoduchá, protože provádí pouze základní operace. Může tedy být velmi malá. Paralelní systém jich může obsahovat velké množství a řešit rozsáhlé soustavy diferenciálních rovnic.

- Několik obdobných implementací ve světě využívá metodu Runge-Kutta. Z rozboru provedeného v této práci vyplývá, že metodou Taylorovy řady jsme schopni dosáhnout vyšší rychlosti a přesnosti.

- Do budoucna nabízí velké možnosti využití grafických procesorů GPU. Efektivní paralelní systém, který rychle počítá základní operace, zde je již navržen. Velkou nevýhodou je zatím nižší přesnost.

9.3 Možnosti dalšího výzkumu

Tato disertační práce si kladla za cíl podrobně popsat možnosti, jak urychlit úspěšnou metody Taylorovy řady pomocí paralelního hardwarového zpracování. Žádná podobná práce tohoto druhu zatím nevznikla a proto by měla sloužit jako podklad pro další výzkum a vylepšení v této oblasti. V této chvíli se nabízí několik možností, na co se dále zaměřit.

Implementovat rozsáhlejší paralelní systém na výkonnější architektuře než FITkit - např FPGA čip typu Virtex a provést detailní srovnání se systémy, které řeší podobné problémy. Tímto se mohou potvrdit dosavadní příznivé výsledky a může následně pokračovat další výzkum.

Jak bylo v této práci několikrát zmíněno, velmi důležitou částí každého paralelního systému je propojovací síť. Další výzkum může být tedy proveden v oblasti propojovacích sítí - navrhnout několik variant, důkladně analyzovat a provést podrobné srovnání.

Při řešení tuhých systémů dochází k velkému rozptylu vstupních i počítaných hodnot. Zde představená koncepce v pevné řádové čárce by proto nebyla dostačující pro řešení takovýchto problémů. Nabízí se proto dále rozšířit koncepce integrátoru v provedení pohyblivé řádové čárky. Další možností je navrhnout a implementovat víceslovní aritmetiku, kde by se pro uložení čísel využívaly stovky bitů.

V neposlední řadě je možné navrhnout programové prostředí, které bude schopno zadaný problém transformovat pro řešení v našem paralelním systému. Tzn. aplikuje tvořící diferenciální rovnice, podle toho provede celkové propojení systému a nastartuje výpočet. Následně zpracuje výsledná data a uloží je nebo zobrazí v nějaké uživatelsky přívětivé formě (např. do grafu).

Literatura

[AFLV08] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.*, 38(4):63–74, 2008. ISSN 0146-4833.

[Čam11] M. Čambor. Paralelní řešení parciálních diferenciálních rovnic. Master's thesis, FIT VUT v Brně, 2011.

[ATI10] ATI. Ati stream technology [online], [cit. 10-8-2010]. `<www.amd.com/stream>`.

[BHN+04] P. Banerjee, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, S. Bagchi, D.and Pal, N. Tripathi, D. Zaretsky, R. Anderson, and J. R. Uribe. Overview of a compiler for synthesizing MATLAB programs onto FPGAs. *IEEE Trans. VLSI Syst.*, 12(3):312–324, 2004.

[BHSL82] M. Bobek, J. Haška, I. Serba, and M Lukeš. *Analogové počítače*. SNTL Praha, 1982. ISBN 0451082.

[Bt05] M. Bečvář and P. Štukjunger. Fixed-point arithmetic in FPGA. *Acta Polytechnica*, 45(2):67–72, 2005. ISSN 1210-2709.

[But00] J. C. Butcher. Numerical methods for ordinary differential equations in the 20th century. *Journal of Computational and Applied Mathematics*, 125(1-2):1–29, 2000. ISSN 0377-0427.

[But08] J. C. Butcher. *Numerical methods for ordinary differential equations*. Wiley, 2 edition, 2008. ISBN 978-0-470-72335-7.

[BWZ72] D. Barton, I. M. Willers, and R. V. M. Zahar. Taylor series methods for ordinary differential equations - an evaluation. In *Mathematical Software Symposium*, pages 369–390. Ed. Academic Press, New York, 1972.

[DB91] J. Diblík and J. Baštinec. *Matematika III*. VUT v Brně, 1 edition, 1991. ISBN 80-214-0315-2.

[DB08] R. C. Dorf and R. H. Bishop. *Modern control systems*. Pearson Prentice Hall, 2008. ISBN 9780132270281.

[DD99] V. Dvořák and V. Drábek. *Architektura procesorů*. Nakladatelství VUT v Brně - VUTIUM, 1 edition, 1999. ISBN 80-214-1458-8.

[Dou03] J. Douša. *Jazyk VHDL*. České vysoké učení technické v Praze, vydavatelství ČVUT, 2003. ISBN 80-01-02670-1.

[Drá95] V. Drábek. *Výstavba počítačů*. Vysoké učení technické v Brně, nakladatelství PC-DIR, 1995. ISBN 80-214-0691-7.

[DT04] W. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004. ISBN 0-12-200751-4.

[Dvo04] V. Dvořák. *Architektura a programování parelelních systémů*. Vysoké učení technické v Brně, nakladatelství VUTIUM, 2004. ISBN 80-214-2608-X.

[FIT11] FIT. Domovská web stránka fit-kitu [online], [cit. 2-5-2011]. `<http://merlin.fit.vutbr.cz/FITkit/>`.

[FTCK09] A. Fasih, Tuan Do Trong, J. C. Chedjou, and K. Kyamakya. New computational modeling for solving higher order ODE based on FPGA. In *Proceedings of 2nd International Workshop on Nonlinear Dynamics and Synchronization, INDS 2009*, pages 49–53, 2009. ISBN 978-1-4244-3844-0.

[Gib60] A. Gibbons. A program for the automatic integration of differential equations using the method of Taylor series. *Computer Journal*, 3(5):108–111, 1960.

[Gra12] Mentor Graphics. High performance and capacity mixed HDL simulation - ModelSim [online], [cit. 2-3-2012]. `<http://www.mentor.com/products/fv/modelsim/>`.

[HKSE07] B. J. Hickmann, A. Krioukov, M. J. Schulte, and M. A. Erle. A parallel IEEE P754 decimal floating-point multiplier. In *25th International Conference on Computer Design Proceedings*, pages 296–303. IEEE, 2007. ISBN 1-4244-1258-7.

[Hol99] O. Holub. Numerické řešení rozsáhlých soustav diferenciálních a algebraických rovnic. Master's thesis, FEI, VUT, 1999.

[HP06] J. L. Hennessy and D. A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann Publisher, 4 edition, 2006. ISBN 0-12-370490-1.

[HW96] E. Hairer and G. Wanner. *Solving Ordinary Differential Equations II*. Springer series in computational mathematics. Springer-Verlag, 1996. ISBN 3-540-60452-9.

[Hwa06] Enoch O. Hwang. *Digital Logic and Microprocessor Design with VHDL*. Thomson, 1 edition, 2006. ISBN 0-534-46593-5.

[iDZD10] V Šimek, R Dvořák, F. Zbořil, and V. Drábek. Performance evaluation of OpenCL framework for numerical solver of advection diffusion equation. In *Proceedings of CSE 2010 International Scientific Conference on Computer Science and Engineering*, pages 279–286. The University of Technology Košice, 2010.

[Jak05] O. Jakl. Paralelní systémy. Technical report, VŠB-TU Ostrava, 2005.

[KDJ05] M. Kubíček, M. Dubcová, and D. Janovská. *Numerické metody a algoritmy*. Vydavatelství VŠCHT Praha, 2 edition, 2005. ISBN 80-7080-558-7.

[Khr10] Khronos. Standard OpenCL [online], [cit. 10-8-2010]. <http://www.khronos.org/opencl/>.

[Kra06] M. Kraus. Elementární procesor specializovaného paralelního systému. Master's thesis, FIT, VUT, 2006.

[Kun94] J. Kunovský. *Modern Taylor Series Method.* FEI-VUT Brno, 1994. Habilitation work.

[Ma09] Dřínovským M. and J. Šťastný. Implementace sériových aritmetických operátorů na moderních programovatelných hradlových polích. *Elektrorevue*, (32):7, 2009. ISSN 1213-1539.

[Mac07] B. J. MacLennan. A review of analog computing. Technical report, Department of Electrical Engineering & Computer Science University of Tennessee, Knoxville, 2007.

[Map12] MapleSoft. Maplesoft - technical computing software [online], [cit. 10-4-2012]. <www.maplesoft.com/products/maple/>.

[Mat12] MathWorks. Matlab - the language of technical computing [online], [cit. 10-4-2012]. <www.mathworks.com/products/matlab/>.

[Mik00] K. Mikulášek. *Polynomial Transformations of Systems of Differential Equations and Their Applications.* PhD thesis, FEI-VUT Brno, 2000.

[nVi10] nVidia. nVidia CUDA zone [online], [cit. 10-8-2010]. <http://www.nvidia.com/object/cuda_home_new.html>.

[Par09] B. Parhami. *Computer arithmetic: algorithms and hardware designs.* The Oxford Series in Electrical and Computer Engineering Series. Oxford University Press, 2009. ISBN 978-0-19-532848-6.

[Pet92] N. Petkov. *Systolic parallel processing.* Advances in parallel computing. North-Holland, 1992. ISBN 9780444887696.

[PP06] J. Pinker and M. Poupa. *Číslicové systémy a jazyk* VHDL. Ben - technická literatura, 1 edition, 2006. ISBN 80-7300-198-5.

[Swa73] E. E. Swartzlander. Parallel counters. *IEEE Trans. Comput.*, 22(11):1021–1024, 1973. ISSN 0018-9340.

[Šát12] V. Šátek. *Analýza stiff soustav diferenciálních rovnic.* PhD thesis, Department of Intelligent Systems FIT BUT, 2012.

[Tvr03] P. Tvrdík. *Parallel algorithm and computing.* České vysoké učení technické, vydavatelství ČVUT, 2003. ISBN 80-01-02824-0.

[Šva99] S. Švarc. *Matematická analýza I.* VUT v Brně, 4 edition, 1999. ISBN 80-214-1411-1.

[WM06] M. Woulfe and M. Manzke. Towards a Field-Programmable Physics Processor (FP^3). In Catherine Noonan, editor, *Eurographics Ireland Chapter Workshop Proceedings 2006*, volume 5 of *Eurographics Ireland Workshop Series*, pages 44–50, Dún Laoghaire, Co Dublin, Ireland, 31 October 2006. IADT Dún Laoghaire.

Přehled publikací autora

[1] G. Fuchs, J. Kopřiva, M. Kraus, and M. Kozek. Application of the modern taylor series method to a multi-torsion chain. In *Proceedings of the 7th EUROSIM Congress on Modelling and Simulation*, page 7. Czech Technical University Publishing House, 2010.

[2] V. Kaluža, M. Kraus, J. Kunovský, and V. Šátek. Initial problems with polynomials on right-hand sides. In *Proceedings of Third Asia International Conference on Modelling and Simulation*, pages 182–187. IEEE Computer Society, 2009.

[3] M. Kraus and J. Kunovský. Adapting power-series integration to real-time simulation. In *Proceedings of the 6th EUROSIM Congress on Modelling and Simulation*, page 6, 2007.

[4] M. Kraus, J. Kunovský, M. Pindryč, and V. Šátek. Taylor series in control theory. In *Proceedings UKSim 10th International Conference EUROSIM/UKSim2008*, pages 378–379. IEEE Computer Society, 2008.

[5] M. Kraus, J. Kunovský, and V. Šátek. Fourier analysis and modern taylor series method. In *Proceedings of the 7th International Conference on Applied Mathematics*, page 6, 2007.

[6] M. Kraus, J. Kunovský, and V. Šátek. Taylor series numerical integrator. In *Second UKSIM European Symposium on Computer Modeling and Simulation*, pages 177–180. IEEE Computer Society, 2008.

[7] M. Kraus, J. Kunovský, and V. Šátek. Taylorian initial problem. In *Proceedings MATHMOD 09 Vienna - Full Papers CD Volume*, pages 1181–1186. ARGE Simulation News, 2009.

[8] J. Kunovský, J. Konvalina, and M. Kraus. Implementace TKSL v pevné a pohyblivé řádové čárce. In *Proceedings of 40th Spring International Conference Mosis'06, Modelling and Simulation of Systems*, pages 259–264, 2006.

[9] J. Kunovský, M. Kraus, V. Šimek, and J. Petřek. GPU based accelleration of telegraph equation. In *Proceedings UKSim 10th International Conference EUROSIM/UKSim2008*, pages 629–630. IEEE Computer Society, 2008.

[10] J. Kunovský, M. Kraus, V Šátek, and V. Kaluža. Accuracy and word width in TKSL. In *Second UKSIM European Symposium on Computer Modeling and Simulation*, pages 153–158. IEEE Computer Society, 2008.

[11] J. Kunovský, M. Kraus, V. Šátek, and A. Szöllös. Parallel computations based on modified numerical integration methods. In *Proceedings of the 10th International Conference of Numerical Analysis and Applied Mathematics*, number 1472, page 4. American Institute of Physics, 2012.

[12] J. Kunovský, M. Kraus, and V. Vopěnka. Runge-kutta based parallel computations. In *Proceedings of the MATHMOD VIENNA 2012 - 7th Vienna Conference on Mathematical Modelling*, page 6. ARGE Simulation News, 2012.

[13] J. Kunovský, V. Šátek, and M. Kraus. 25th anniversary of TKSL. In *Numerical Analysis and Applied Mathematics*, pages 343–346. American Institute of Physics, 2008.

[14] J. Kunovský, V. Šátek, and M. Kraus. New trends in taylor series based computations. In *Numerical Analysis and Applied Mathematics*, pages 282–285. American Institute of Physics, 2009.

[15] J. Kunovský, V Šátek, M. Kraus, and J. Kopřiva. Semi-analytical computations based on TKSL. In *Second UKSIM European Symposium on Computer Modeling and Simulation*, pages 159–164. IEEE Computer Society, 2008.

[16] J. Kunovský, V. Šátek, M. Kraus, and J. Kopřiva. Automatic method order settings. In *Proceedings of Eleventh International Conference on Computer Modelling and Simulation*, pages 117–122. IEEE Computer Society, 2009.

[17] J. Kunovský, V. Šátek, M. Kraus, and M. Pindryč. Comparison of TKSL to world standards. *International Journal of Autonomic Computing*, 1(2):182–191, 2009.

[18] V. Kunovský, M. Kraus, V. Šátek, and V. Kaluža. Parallel computations based on analogue principles. In *Proceedings of Eleventh International Conference on Computer Modelling and Simulation*, pages 111–116. IEEE Computer Society, 2009.

[19] V. Šátek, J. Kunovský, M. Kraus, and M. Pindryč. TKSL to world standards comparison. In *MOSIS '08*, pages 21–27, 2008.

Příloha A

Popis implementace paralelního systému v jazyce VHDL

Základní návrhová jednotka v jazyce VHDL je tzv. entita. Popisuje rozhraní - vstupy a výstupy daného systému, který může představovat pouze logické hradlo, nebo velký systém. Entita lze přirovnat ke schematické značce, která zobrazuje vstupy a výstupy.

Entita představující celý navržený paralelní systém v jazyce VHDL je uvedena níže a odpovídající schematická značka je na obrázku A.1.

Obrázek A.1: Základní entita paralelního systému

```
entity alu is
   generic (pocet_bitu: integer);
   port (
      CLK: in std_logic;       -- vstup pro hodinovy signal
      CLR: in std_logic;       -- vstup pro nulovaci signal
      DATA_EN: in std_logic;  -- priznak, zda jsou platna vstupni data (Y0)
      DATAY0: in std_logic_vector(pocet_bitu-1 downto 0);   -- vstup pro pocatecni
          podminku Y0
      VE: out std_logic;       -- priznak, ze jsou platna vystupni data (vysledek)
      RESULT: out std_logic_vector(pocet_bitu-1 downto 0)   -- vystup celeho systemu
           - vysledek vypoctu
   );
end alu;
```

Tato nejvyšší entita obsahuje pouze vstupy pro hodinový a nulovací signál, vstup pro počáteční podmínku a příznak, zda jsou vstupní data validní a může se začít provádět výpočet. Dále obsahuje výstup výpočtu a opět příznak, že výsledek je validní a může se přečíst připojeným systémem a následně uložit a zpracovat.

Sekce generic definuje tzv. generické konstanty entity. Je využita pro parametrizaci entity - nastavení bitové šířky dat. Tuto hodnotu je možné měnit při každém použití entity jako komponenty, např. v tzv. test benchi nebo v nadřazeném systému tvořící rozhraní pro navržený paralelní systém s hardwarovou platformou, na kterém běží.

Obvod se může popsat dvěma způsoby: strukturně nebo behaviorálně. Při strukturním způsobu se popíší jednotlivé komponenty obvodu a jejich propojení pomocí vodičů - signálů.

Komponenty mohou být dále dekomponovány na subkomponenty a samostatně popsány - vytvořena hierarchie. Tento způsob je blízký finální obvodové realizaci.

Behaviorální způsob využívá toho, že chování obvodu lze popsat algoritmem. Při tomto popisu se neuvažují obvodové detaily. Tvorba výsledného zapojení obvodu se nechá na procesu syntézy. Tímto způsobem se popisují elementární komponenty, které se následně využívají ve strukturálním popisu. Na určité úrovni je vždy třeba ukončit strukturní popis a definovat elementární prvky.

Na přiloženém CD jsou zdrojové kódy celého paralelního systému umístěny v adresáři `VHDL`. V podadresářích je uloženo několik variant, které řeší různé diferenciální rovnice a využívají různé verze navržených integrátorů. Soubor `description-VHDL.txt` obsahuje přesný popis obsahu jednotlivých podadresářů.

Každý podadresář obsahuje 3 základní soubory:

- `alu.vhd` - implementace hlavní entity paralelního systému. Obsahuje řídící jednotku, čítače pro počítání smyček, aritmeticko-logické jednotky (integrátory) a statickou propojovací síť.
- `alu_tb.vhd` - soubor sloužící pro testování navrženého systému. Představuje testovací prostředí, generuje testovací signály a případně zpracovává odezvy. Spuštěním přiloženého testbenche např. v simulačním programu ModelSim [Gra12] lze získat časové průběhy ze simulované komponenty `alu` (paralelního systému).
- `alu_kernel.vhd` - implementace aritmeticko-logické jednotky - integrátoru. Existují tři verze: paralelně-paralelní, sériově-paralelní a sériově-sériová.

Ostatní soubory obsahují behaviorální popis implementace elementárních prvků, které se využívají ve strukturním popisu nadřazených komponent.

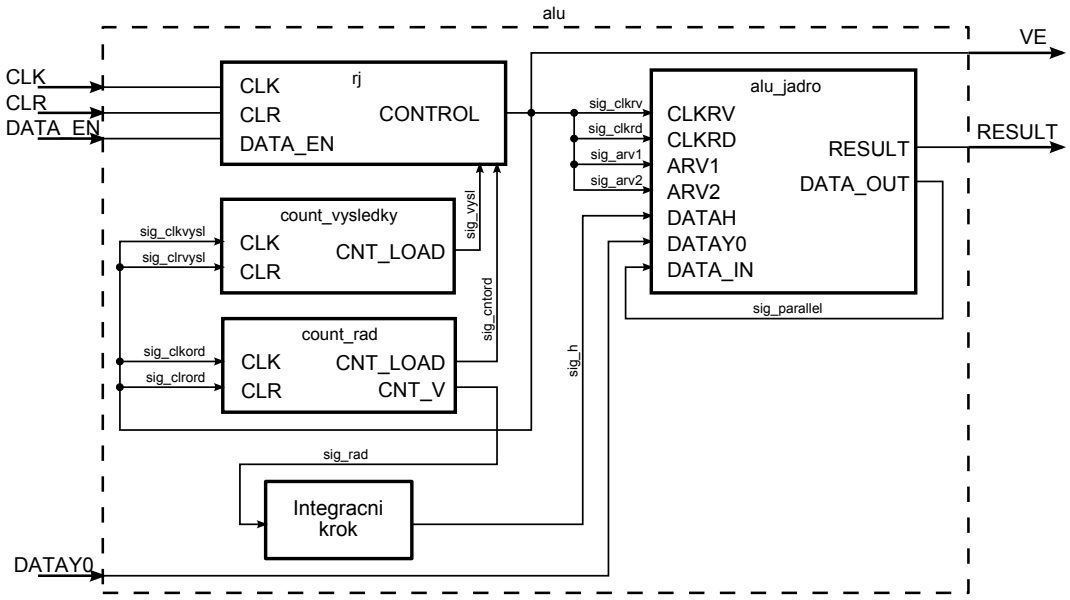
Na CD je dále adresář `FITkit`, který obsahuje kromě souborů představených výše i další soubory sloužící k přeložení a nahrání aplikace do přípravku FITkit [FIT11]. Bližší popis implementace bude uveden dále, konkrétně v podkapitole - A.4.

V následující části budou představeny 3 různé implementace specializovaného paralelního systému. Každý bude obsahovat jiný typ aritmeticko-logické jednotky: paralelně-paralelní, sériově-paralelní a sériově-sériovou. Každý systém bude řešit stejnou diferenciální rovnici (2.13), která byla také použita k analýze numerického řešení pomocí Taylorovy řady v kapitole 2.2. Bude představen soubor `alu.vhd` obsahující strukturní popis celého paralelního systému v jazyce VHDL a odpovídající blokové schéma, které zobrazuje jednotlivé komponenty a signály sloužící k jejich propojení.

A.1 Paralelně-paralelní varianta

Koncepce paralelně-paralelního integrátoru byla představena v kapitole 6.1.1 a detailnější popis realizace v kapitole 7.1.1. Na obrázku A.2 jsou zobrazeny použité komponenty:

- **rj** - řídící jednotka (řadič) která řídí činnost celého systému. Generuje řídící signály pro čítače a aritmeticko-logickou jednotku.
- **count_vysledky** - čítač počítající celkový počet výsledků y_i
- **count_rad** - čítač počítající počet členů Taylorovy řady (řád metody)



Obrázek A.2: Specializovaný paralelní systém s paralelně-paralelní ALU

- **integracni_krok** - proces generující jednotlivé podíly integračního kroku podle aktuálně počítaného řádu metody
- **alu_jadro** - aritmeticko-logická jednotka (integrátor) provádějící vlastní výpočet

```
1  ----------------------------------------------------------------------------
2  -- Paralelni integrator resici rovnici y'=y
3  ----------------------------------------------------------------------------
4  -- Autor: Michal Kraus
5  -- Datum: 18.4.2010
6  -- aritmeticko logicka jednotka paralelniho integratoru
7  ----------------------------------------------------------------------------
8
9  library IEEE;
10 use IEEE.std_logic_1164.all;
11 use ieee.std_logic_arith.all;
12 use ieee.std_logic_unsigned.all;  -- aritmeticke operace (scitani, odcitani,
       nasobeni)
13
14 -- ============================ paralelni system ============================
15 -- CLK - hodiny alu
16 -- CLR - nulovani alu
17 -- DATAY0 - data pocatecni podminky
18
19 entity alu_paralelni is
20    generic (pocet_bitu: integer);
21    port (
22       CLK: in std_logic;     -- original signaly ALU
23       CLR: in std_logic;
24       DATA_EN: in std_logic;
25       DATAY0: in std_logic_vector(pocet_bitu-1 downto 0);
26       VE: out std_logic;
27       RESULT: out std_logic_vector(pocet_bitu-1 downto 0)
28    );
29 end alu_paralelni;
30
31 architecture struct of alu_paralelni is
32
33 -- ============================ komponenty ============================
34 component alu_kernel is
```

```
35      generic ( pocet_bitu: integer );
36      port (
37         CLKRV:    in std_logic;
38         CLKRD:    in std_logic;
39         ARV1:     in std_logic;
40         ARV2:     in std_logic;
41         DATAY0:   in std_logic_vector(pocet_bitu-1 downto 0);
42         DATA_IN:  in std_logic_vector(pocet_bitu-1 downto 0);
43         DATAH:    in std_logic_vector(pocet_bitu-1 downto 0);
44         DATA_OUT: out std_logic_vector(pocet_bitu-1 downto 0);
45         RESULT:   out std_logic_vector(pocet_bitu-1 downto 0)
46      );
47   end component;
48
49   component counter is
50      generic (
51         range_c: integer
52      );
53      port (
54         CLK: in std_logic;
55         CLR: in std_logic;
56         CNT_LOAD: out std_logic
57      );
58   end component;
59
60   component counter2 is
61      port (
62         CLK: in std_logic;
63         CLR: in std_logic;
64         CNT_LOAD: out std_logic;
65         CNT_V: out std_logic_vector(3 downto 0)
66      );
67   end component;
68
69   -- ======================== vlastni popis alu  ============================
70   type TStav is (Init,PPdoRV,PPdoRD,PrepnutiMPX,Vypocet,ZapisRV,ZapisRD,CitacRad,
         TestORD,PrepnutiMPX1,VysledekdoRD,Vysledek);
71   signal sig_h, sig_parallel:  std_logic_vector(pocet_bitu-1 downto 0);
72   signal sig_clkrv,sig_clkrd,sig_arv1,sig_arv2,sig_clkvysl,sig_clrvysl,sig_vysl,
         sig_clkord,sig_clrord,sig_cntord: std_logic;
73   signal AktualniStav, DalsiStav: TStav;
74   signal sig_rad: std_logic_vector(3 downto 0);
75
76   begin
77   -- propojeni alu_kernel s ridicimi signaly
78   alu_jadro: alu_kernel
79      generic map ( pocet_bitu => pocet_bitu )
80      port map (
81         CLKRV    => sig_clkrv,
82         CLKRD    => sig_clkrd,
83         ARV1     => sig_arv1,
84         ARV2     => sig_arv2,
85         DATAY0   => DATAY0,
86         DATA_IN  => sig_parallel,
87         DATAH    => sig_h,
88         DATA_OUT => sig_parallel,
89         RESULT   => RESULT
90      );
91
92   count_vysledky: counter
93      generic map (
94         -- 2 = pouze jeden vysledek (x = i-1; x => pocet pozadovanych vysledku)
95         range_c => 6              -- => i
96      )
97      port map (
98         CLK       => sig_clkvysl,
99         CLR       => sig_clrvysl,
100        CNT_LOAD  => sig_vysl
```

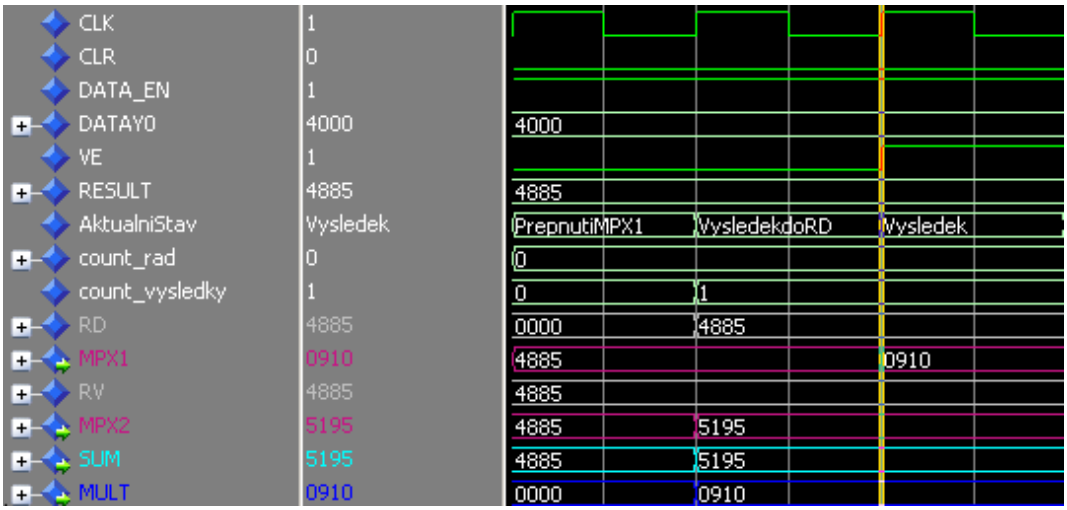
```
101     );
102
103  count_rad: counter2
104     port map (
105        CLK       => sig_clkord,
106        CLR       => sig_clrord,
107        CNT_LOAD  => sig_cntord,
108        CNT_V     => sig_rad
109     );
110
111  --------------------------------------------------------------------------------
112  -- Konecny automat:
113  -- stavovy registr - prepnuti na nasledujici stav
114  Clock: process(CLR,CLK)
115  begin
116    if CLR = '1' then
117      AktualniStav <= Init;
118    elsif CLK'event and CLK='1' then
119      AktualniStav <= DalsiStav;
120    end if;
121  end process;
122  -- zvoleni nasledujiciho stavu
123  NaslStav: process(AktualniStav,DATA_EN)
124  begin
125    case AktualniStav is
126      when Init =>          -- ARV1=0 ARV2=0
127        if DATA_EN = '1' then
128          DalsiStav <= PPdoRV;
129        else
130          DalsiStav <= Init;
131        end if;
132      when PPdoRV =>          -- pocatecni podminka do RV => CLKRV=1 ARV1=0 ARV2=0
133         DalsiStav <= PPdoRD;
134      when PPdoRD =>          -- pocatecni podminka do RD => CLKRD=1 ARV1=0 ARV2=0
135         DalsiStav <= PrepnutiMPX;
136      when PrepnutiMPX =>     -- prepnuti MPX 1 a 2 => ARV1=1 ARV2=1
137         DalsiStav <= Vypocet;
138      when Vypocet =>         -- probiha vypocet
139         DalsiStav <= ZapisRV;
140      when ZapisRV =>         -- zapis do RV
141         DalsiStav <= ZapisRD;
142      when ZapisRD =>         -- zapis do RD
143         DalsiStav <= CitacRad;
144      when CitacRad =>        -- zvyseni citace radu metody
145         DalsiStav <= TestORD;
146      when TestORD =>         -- test, jestli uz jsme dosahli pozadovane presnosti
147        if sig_cntord = '1' then
148          DalsiStav <= PrepnutiMPX1;
149        else
150          DalsiStav <= Vypocet;
151        end if;
152      when PrepnutiMPX1 =>    -- Prepnuti MPX1, abychom mohli nahrat vysledek z RV do
           RD
153        DalsiStav <= VysledekdoRD;
154      when VysledekdoRD =>    -- nahrani vysledku do RD - pocatecni podminka pro dalsi
           vypocet y(i+1)
155        DalsiStav <= Vysledek;
156      when Vysledek =>        -- vysledek je aktivni
157        if sig_vysl = '1' then
158          DalsiStav <= Init;
159        else
160          DalsiStav <= Vypocet;
161        end if;
162      when others =>
163        null;
164    end case;
165  end process;
166  -- podle stavu generuje vystup - ridici signaly mikroprogramu
```

```
167  Vystup: process(AktualniStav)
168  begin
169    VE <= '0';      -- default: vysledek neni aktivni
170    case AktualniStav is
171      when Init =>
172        sig_clrvysl <= '1';
173        sig_clkvysl <= '0';
174        sig_clkord  <= '0';
175        sig_clrord  <= '1';
176        sig_clkrv   <= '0';
177        sig_clkrd   <= '0';
178        sig_arv1    <= '0';
179        sig_arv2    <= '0';
180        VE <= '0';
181      when PPdoRV =>
182        sig_clrvysl <= '0';
183        sig_clkvysl <= '0';
184        sig_clkord  <= '0';
185        sig_clrord  <= '0';
186        sig_clkrv   <= '1';
187        sig_clkrd   <= '0';
188        sig_arv1    <= '0';
189        sig_arv2    <= '0';
190        VE <= '0';
191      when PPdoRD =>
192        sig_clrvysl <= '0';
193        sig_clkvysl <= '0';
194        sig_clkord  <= '0';
195        sig_clrord  <= '0';
196        sig_clkrv   <= '0';
197        sig_clkrd   <= '1';
198        sig_arv1    <= '0';
199        sig_arv2    <= '0';
200        VE <= '0';
201      when PrepnutiMPX =>
202        sig_clrvysl <= '0';
203        sig_clkvysl <= '0';
204        sig_clkord  <= '0';
205        sig_clrord  <= '0';
206        sig_clkrv   <= '0';
207        sig_clkrd   <= '0';
208        sig_arv1    <= '1';
209        sig_arv2    <= '1';
210        VE <= '0';
211      when Vypocet =>
212        sig_clrvysl <= '0';
213        sig_clkvysl <= '0';
214        sig_clkord  <= '0';
215        sig_clrord  <= '0';
216        sig_clkrv   <= '0';
217        sig_clkrd   <= '0';
218        sig_arv1    <= '1';
219        sig_arv2    <= '1';
220        VE <= '0';
221      when ZapisRV =>
222        sig_clrvysl <= '0';
223        sig_clkvysl <= '0';
224        sig_clkord  <= '0';
225        sig_clrord  <= '0';
226        sig_clkrv   <= '1';
227        sig_clkrd   <= '0';
228        sig_arv1    <= '1';
229        sig_arv2    <= '1';
230        VE <= '0';
231      when ZapisRD =>
232        sig_clrvysl <= '0';
233        sig_clkvysl <= '0';
234        sig_clkord  <= '0';
```

```
235         sig_clrord  <= '0';
236         sig_clkrv   <= '0';
237         sig_clkrd   <= '1';
238         sig_arv1    <= '1';
239         sig_arv2    <= '1';
240         VE <= '0';
241       when CitacRad =>
242         sig_clrvysl <= '0';
243         sig_clkvysl <= '0';
244         sig_clkord  <= '1';
245         sig_clrord  <= '0';
246         sig_clkrv   <= '0';
247         sig_clkrd   <= '0';
248         sig_arv1    <= '1';
249         sig_arv2    <= '1';
250         VE <= '0';
251       when TestORD =>
252         sig_clrvysl <= '0';
253         sig_clkvysl <= '0';
254         sig_clkord  <= '0';
255         sig_clrord  <= '0';
256         sig_clkrv   <= '0';
257         sig_clkrd   <= '0';
258         sig_arv1    <= '1';
259         sig_arv2    <= '1';
260         VE <= '0';
261       when PrepnutiMPX1 =>
262         sig_clrvysl <= '0';
263         sig_clkvysl <= '0';
264         sig_clkord  <= '0';
265         sig_clrord  <= '1';
266         sig_clkrv   <= '0';
267         sig_clkrd   <= '0';
268         sig_arv1    <= '0';
269         sig_arv2    <= '1';
270         VE <= '0';
271       when VysledekdoRD =>
272         sig_clrvysl <= '0';
273         sig_clkvysl <= '1';
274         sig_clkord  <= '0';
275         sig_clrord  <= '0';
276         sig_clkrv   <= '0';
277         sig_clkrd   <= '1';
278         sig_arv1    <= '0';
279         sig_arv2    <= '1';
280         VE <= '0';
281       when Vysledek =>
282         sig_clrvysl <= '0';
283         sig_clkvysl <= '0';
284         sig_clkord  <= '0';
285         sig_clrord  <= '0';
286         sig_clkrv   <= '0';
287         sig_clkrd   <= '0';
288         sig_arv1    <= '1';
289         sig_arv2    <= '1';
290         VE <= '1';
291       when others =>
292         null;
293     end case;
294   end process;
295
296   integracni_krok: process(sig_rad)
297   begin
298      case sig_rad is
299         -- 16 bitova verze
300         when "0000" => sig_h <= X"1000";  -- h = 0.125
301         when "0001" => sig_h <= X"0800";  -- h/2 = 0.0625
302         when "0010" => sig_h <= X"0555";  -- h/3 = 0.041666666666667
```

```
303         when "0011" => sig_h <= X"0400";  -- h/4 = 0.03125
304         when "0100" => sig_h <= X"0333";  -- h/5 = 0.025
305         when "0101" => sig_h <= X"02AA";  -- h/6 = 0.0208333333333
306         when "0110" => sig_h <= X"0249";  -- h/7 = 0.0178571428571428571428571428571428571143
307         when "0111" => sig_h <= X"0200";  -- h/8 = 0.015625
308         when others => sig_h <= X"1000";  -- h = 0.125
309     end case;
310 end process;
311
312 end struct;
```

Výsledky simulace z programu ModelSim jsou na obrázku A.3. Je zde zobrazen okamžik, kdy je vypočtena výsledná hodnota y_1 pomocí 8 členů Taylorovy řady.



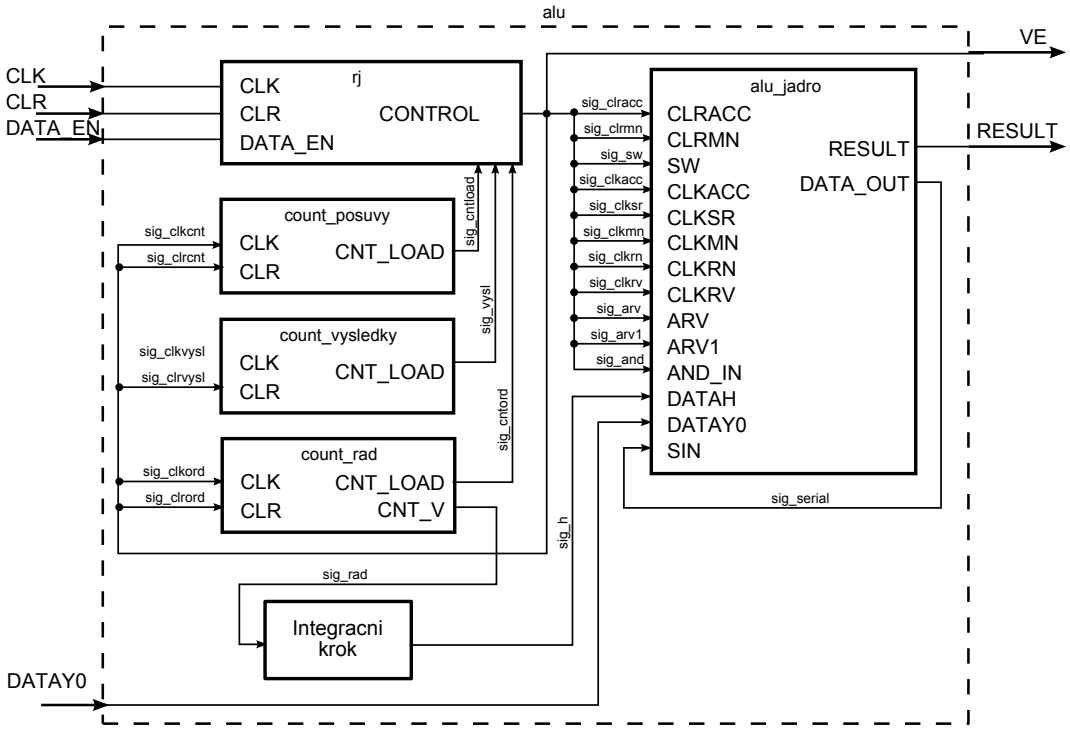
Obrázek A.3: Simulace VHDL kódu paralelně-paralelního systému

A.2 Sériově-paralelní varianta

Koncepce sériově-paralelního integrátoru byla představena v kapitole 6.1.2 a detailnější popis realizace v kapitole 7.1.2. Implementovaná verze obsahuje paralelní vstup pro počáteční podmínku a integrační krok a dále paralelní výstup pro výsledek. Je také možná verze se sériovou sběrnicí, která byla představena v kapitole 6.4.2, resp. 6.5.3. Na obrázku A.4 jsou zobrazeny použité komponenty:

- **rj** - řídící jednotka (řadič) která řídí činnost celého systému. Generuje řídící signály pro čítače a aritmeticko-logickou jednotku.
- **count_posuvy** - čítač počítající posuvy prováděné v SR a ACC
- **count_vysledky** - čítač počítající celkový počet výsledků y_i
- **count_rad** - čítač počítající počet členů Taylorovy řady (řád metody)
- **integracni_krok** - proces generující jednotlivé podíly integračního kroku podle aktuálně počítaného řádu metody
- **alu_jadro** - aritmeticko-logická jednotka (integrátor) provádějící vlastní výpočet

```
1 --------------------------------------------------------------------------------
2 -- Specializovany procesor provadejici numerickou integraci
```



Obrázek A.4: Specializovaný paralelní systém se sériově-paralelní ALU

```
3  ----------------------------------------------------------------------------
4  -- Autor: Michal Kraus
5  -- Datum: 15.4.2010
6  -- Soubor: aritmeticko logicka jednotka + vypocet pouze jednoho vysledku
7  -- predelany ridici a signaly (okomentovany a nechany v kazdem stavu jen ty, ktere
       se meni)
8  ----------------------------------------------------------------------------
9
10 library IEEE;
11 use IEEE.std_logic_1164.all;
12 use ieee.std_logic_arith.all;
13 use ieee.std_logic_unsigned.all;
14
15 -- ============================ paralelni system ============================
16 -- CLK - hodiny alu
17 -- CLR - nulovani alu
18 -- DATAY0 - data pocatecni podminky
19
20 entity alu is
21    generic (pocet_bitu: integer);
22    port (
23       CLK: in std_logic;     -- original signaly ALU
24       CLR: in std_logic;
25       DATA_EN: in std_logic;
26       DATAY0: in std_logic_vector(pocet_bitu-1 downto 0);
27       VE: out std_logic;
28       RESULT: out std_logic_vector(pocet_bitu-1 downto 0)
29    );
30 end alu;
31
32 architecture struct of alu is
33
34 -- ============================ komponenty ============================
35 component alu_kernel is
36    generic ( pocet_bitu: integer );
```

```
37      port (
38         CLRACC:  in std_logic;
39         CLRMN:   in std_logic;
40         SW:      in std_logic;
41         CLKACC:  in std_logic;
42         CLKSR:   in std_logic;
43         CLKMN:   in std_logic;
44         CLKRN:   in std_logic;
45         CLKRV:   in std_logic;
46         ARV:     in std_logic;
47         ARV1:    in std_logic;
48         AND_IN:  in std_logic;
49         SIN:     in std_logic;
50         DATAY0:  in std_logic_vector(pocet_bitu-1 downto 0);
51         DATAH:   in std_logic_vector(pocet_bitu-1 downto 0);
52         SOUT:    out std_logic;
53         RESULT:  out std_logic_vector(pocet_bitu-1 downto 0)
54      );
55   end component;
56
57   component counter is
58      generic (
59         range_c: integer
60      );
61      port (
62         CLK: in std_logic;
63         CLR: in std_logic;
64         CNT_LOAD: out std_logic
65      );
66   end component;
67
68   component counter2 is
69      port (
70         CLK: in std_logic;
71         CLR: in std_logic;
72         CNT_LOAD: out std_logic;
73         CNT_V: out std_logic_vector(3 downto 0)
74      );
75   end component;
76   -- ======================== vlastni popis alu  ============================
77   -- definice stavu automatu, ktery ridi vypocet ALU
78   type TStav is (Init,NahrejY0,NahrejH,NulovaniACC,Vypocet,ZapisACC,BityZpracovany,
         BityNezpracovany,NastavitPosuv,Posuv,NastavitZapisAcc,VysledekNasAcc,ZapisRV,
         DalsiClen,VsechnyCleny,PredVysledek,Vysledek);
79
80   -- paralelni datovy signal pro integracni krok
81   signal sig_h:  std_logic_vector(pocet_bitu-1 downto 0);
82   -- seriovy signal na propojeni vystupu integratoru se vstupem
83   signal sig_serial: std_logic;
84
85   -- ridici signaly ALU (ridici sbernice):
86   signal sig_clkacc, sig_clracc, sig_sw, sig_clkrn, sig_clkrv, sig_arv, sig_mn, sig_bl
         , sig_cin, sig_and, sig_clkmn, sig_clrmn, sig_clksr, sig_arv1: std_logic;
87   -- ridici signaly citacu:
88   signal sig_clkcnt, sig_clrcnt, sig_cntload, sig_clkord, sig_clrord, sig_cntord,
         sig_clkvysl, sig_clrvysl, sig_vysl: std_logic;
89
90   -- uchovani stavu automatu, ktery ridi vypocet ALU
91   signal AktualniStav, DalsiStav: TStav;
92   -- hodnota aktualniho radu metody - vystup z citace
93   signal sig_rad: std_logic_vector(3 downto 0);
94   --signal pocet_vysledku: std_logic_vector(3 downto 0);
95
96   begin
97   -- propojeni alu_kernel s ridicimi signaly
98   alu_jadro: alu_kernel
99      generic map ( pocet_bitu => pocet_bitu )
100     port map (
```

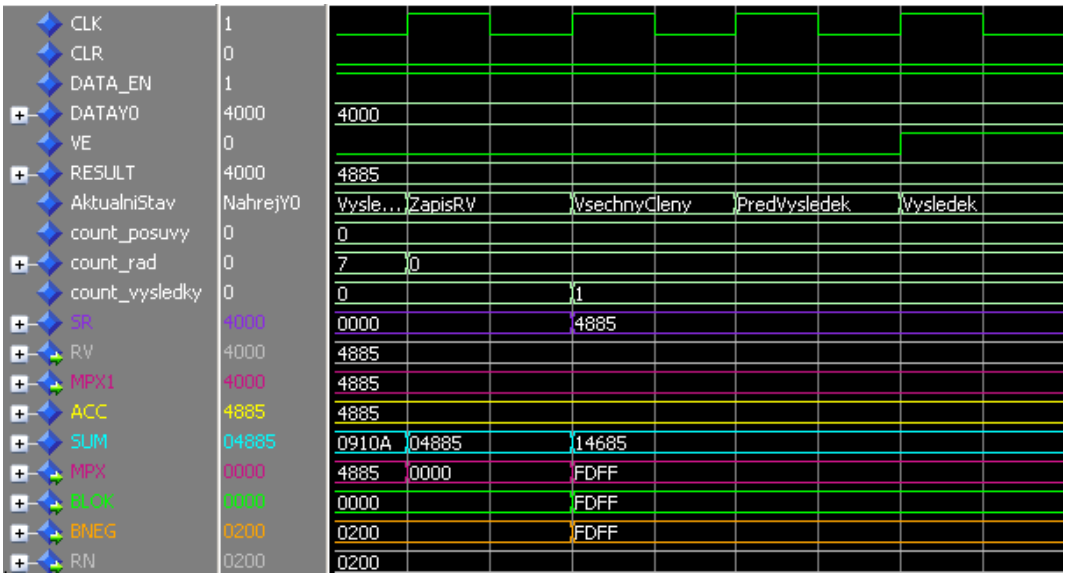
```
101         CLRACC    => sig_clracc,
102         CLRMN     => sig_clrmn,
103         SW        => sig_sw,
104         CLKACC    => sig_clkacc,
105         CLKSR     => sig_clksr,
106         CLKMN     => sig_clkmn,
107         CLKRN     => sig_clkrn,
108         CLKRV     => sig_clkrv,
109         ARV       => sig_arv,
110         ARV1      => sig_arv1,
111         AND_IN    => sig_and,
112         SIN       => sig_serial,
113         DATAY0    => DATAY0,
114         DATAH     => sig_h,
115         SOUT      => sig_serial,
116         RESULT    => RESULT
117     );
118
119 count_posuvy: counter
120     generic map (
121         range_c => pocet_bitu
122     )
123     port map (
124         CLK        => sig_clkcnt,
125         CLR        => sig_clrcnt,
126         CNT_LOAD   => sig_cntload
127     );
128
129 count_vysledky: counter
130     generic map (
131         -- 2 = pouze jeden vysledek (x = i-1; x => pocet pozadovanych vysledku)
132         range_c => 2              -- => i
133     )
134     port map (
135         CLK        => sig_clkvysl,
136         CLR        => sig_clrvysl,
137         CNT_LOAD   => sig_vysl
138     );
139
140 count_rad: counter2
141     port map (
142         CLK        => sig_clkord,
143         CLR        => sig_clrord,
144         CNT_LOAD   => sig_cntord,
145         CNT_V      => sig_rad
146     );
147
148 ---------------------------------------------------------------------------------
149 -- Konecny automat:
150 -- stavovy registr - prepnuti na nasledujici stav
151 Clock: process(CLR,CLK)
152 begin
153   if CLR = '1' then
154     AktualniStav <= Init;
155   elsif CLK'event and CLK='1' then
156     AktualniStav <= DalsiStav;
157   end if;
158 end process;
159 -- zvoleni nasledujiciho stavu
160 NaslStav: process(AktualniStav,DATA_EN)
161 begin
162   --DalsiStav <= Init;
163   case AktualniStav is
164     when Init =>
165       if DATA_EN = '1' then
166         DalsiStav <= NahrejY0;
167       else
168         DalsiStav <= Init;
```

```
169         end if;
170       when NahrejY0 =>
171         DalsiStav <= NahrejH;
172       when NahrejH =>
173         DalsiStav <= NulovaniACC;
174       when NulovaniACC =>
175         DalsiStav <= Vypocet;
176       when Vypocet =>
177         DalsiStav <= ZapisACC;
178       when ZapisACC =>
179         if sig_cntload = '1' then
180           DalsiStav <= BityZpracovany;
181         else
182           DalsiStav <= BityNezpracovany;
183         end if;
184       when BityNezpracovany =>
185         DalsiStav <= NastavitPosuv;
186       when BityZpracovany =>
187         DalsiStav <= VysledekNasACC;
188       when NastavitPosuv =>
189         DalsiStav <= Posuv;
190       when Posuv =>
191         DalsiStav <= NastavitZapisACC;
192       when NastavitZapisACC =>
193         DalsiStav <= ZapisACC;
194       when VysledekNasACC =>
195         DalsiStav <= ZapisRV;
196       when ZapisRV =>
197         if sig_cntord = '1' then
198           DalsiStav <= VsechnyCleny;
199         else
200           DalsiStav <= DalsiClen;
201         end if;
202       when DalsiClen =>
203         DalsiStav <= NahrejH;
204       when VsechnyCleny =>
205         DalsiStav <= PredVysledek;
206       when PredVysledek =>
207         DalsiStav <= Vysledek;
208       when Vysledek =>
209         if sig_vysl = '1' then
210           DalsiStav <= Init;
211         else
212           DalsiStav <= NahrejH;
213         end if;
214       when others =>
215         null;
216     end case;
217   end process;
218   -- podle stavu generuje vystup - ridici signaly mikroprogramu
219   Vystup: process(AktualniStav)
220   begin
221     VE <= '0';      -- default: vysledek neni aktivni
222     case AktualniStav is
223       when Init =>
224         -- ridici signaly citacu:
225         sig_clrvysl  <= '1';   -- vynulovani poctu vysledku
226         sig_clkvysl  <= '0';   -- pricteni vysledku
227         sig_clkcnt   <= '0';   -- pricteni dalsiho posuvu registru
228         sig_clrcnt   <= '0';   -- vynulovani poctu posuvu
229         sig_clkord   <= '0';   -- dalsi clen Taylorovy rady
230         sig_clrord   <= '0';   -- vynulovani poctu clenu Taylorovy rady - kvuli ORD =
              "0000"
231         -- ridici signaly ALU:
232         sig_clracc   <= '0';
233         sig_clrmn    <= '0';
234         sig_sw       <= '0';    -- SW a ACC nastaveny na zapis
235         sig_clkacc   <= '0';
```

```
236         sig_clksr   <= '0';
237         sig_clkmn   <= '0';
238         sig_clkrn   <= '0';
239         sig_clkrv   <= '0';
240         sig_arv     <= '0';
241         sig_arv1    <= '0';   -- vstup MPX1 prepnut na x - pocatecni podminku
242         sig_and     <= '1';
243         -- ridici signal, ze vysledek je hotov
244         VE <= '0';
245       when NahrejY0 =>
246         -- ridici signaly citacu:
247         sig_clrvysl <= '0';   -- zruseni nulovani poctu vysledku
248         sig_clrcnt  <= '1';   -- nulovani citace posuvu
249         sig_clrord  <= '1';   -- nulovani citace radu metody
250         -- ridici signaly ALU:
251         sig_clksr   <= '1';   -- nahrani poc. podminky do SR
252         sig_clkrv   <= '1';   -- nahrani poc. podminky do RV
253       when NahrejH =>
254         -- ridici signaly citacu:
255         sig_clrcnt  <= '0';   -- zruseni nulovani citace posuvu
256         sig_clrord  <= '0';   -- zruseni nulovani citace radu metody
257         -- ridici signaly ALU:
258         sig_clksr   <= '0';   -- zruseni nahravani poc. podminky do SR
259         sig_clkrn   <= '1';   -- nahrani int. kroku do RN
260         sig_clkrv   <= '0';   -- zruseni nahravani poc. podminky do RV
261         sig_arv1    <= '1';   -- MPX1 prepnut na vstup y - vystup z ACC
262       when NulovaniACC =>
263         -- ridici signaly citacu:
264         sig_clrcnt  <= '1';   -- nulovani citace posuvu
265         -- ridici signaly ALU:
266         sig_clracc  <= '1';   -- nulovani ACC
267         sig_clrmn   <= '1';   -- nulovani MN
268         sig_clkrn   <= '0';   -- zruseni nahravani int. kroku du RN.
269       when Vypocet =>
270         -- ridici signaly citacu:
271         sig_clrcnt  <= '0';   -- zruseni nulovani citace posuvu
272         -- ridici signaly ALU:
273         sig_clracc  <= '0';   -- zruseni nulovani ACC
274         sig_clrmn   <= '0';   -- zruseni nulovani MN
275       when ZapisACC =>
276         -- ridici signaly ALU:
277         sig_clkacc  <= '1';   -- zapis vysledku do ACC
278       when BityNezpracovany =>
279         -- ridici signaly ALU:
280         sig_clkacc  <= '0';   -- zruseni zapisu vysledku do ACC
281         sig_clkmn   <= '1';   -- zapis aktualniho vstupu (bude pouzit jako predesla
                hodnota) do MN
282       when NastavitPosuv =>
283         -- ridici signaly ALU:
284         sig_sw      <= '1';   -- nastavit ACC a SR na posuv
285         sig_clkmn   <= '0';   -- zruseni nahravani vstupu do MN
286       when Posuv =>
287         -- ridici signaly citacu:
288         sig_clkcnt  <= '1';   -- zvysit pocet posuvu
289         -- ridici signaly ALU:
290         sig_clkacc  <= '1';   -- posuv ACC
291         sig_clksr   <= '1';   -- posuv SR
292       when NastavitZapisAcc =>
293         -- ridici signaly citacu:
294         sig_clkcnt  <= '0';   -- zruseni zvyseni citace posuvu
295         -- ridici signaly ALU:
296         sig_sw      <= '0';   -- nastavit ACC a SR zpet na zapis
297         sig_clkacc  <= '0';   -- zruseni posuvu ACC
298         sig_clksr   <= '0';   -- zruseni posuvu SR
299       when BityZpracovany =>
300         -- ridici signaly citacu:
301         sig_clrcnt  <= '1';   -- vynovani poctu posuvu
302         -- ridici signaly ALU:
```

```
303         sig_clkacc  <= '0';   -- zruseni zapisu vysledku do ACC
304         sig_clksr   <= '1';   -- nahrani vysledku nasobeni (DYi) do SR
305         sig_arv     <= '1';   -- prepnuti MPX na vstup y - z RV
306         sig_and     <= '0';   -- nastavit AND na '0' => zajisteni pricteni Cin = 0
307       when VysledekNasAcc =>
308         -- ridici signaly citacu:
309         sig_clrcnt  <= '0';   -- zruseni nulovani poctu posuvu
310         -- ridici signaly ALU:
311         sig_clkacc  <= '1';   -- zapis vysledku (yi) do ACC
312         sig_clksr   <= '0';   -- zruseni nahravani vysledku nasobeni do SR
313       when ZapisRV =>
314         -- ridici signaly citacu:
315         sig_clkord  <= '1';   -- zvyseni citace radu metody
316         -- ridici signaly ALU:
317         sig_clkacc  <= '0';   -- zruseni zapisu do ACC
318         sig_clkrv   <= '1';   -- zapsani vysledku do RV
319         sig_arv     <= '0';   -- prepnuti MPX zpet na vstup x - z BLOK
320         sig_and     <= '1';   -- vstup ANDu nastavime opet na '1'
321       when DalsiClen =>
322         -- ridici signaly citacu:
323         sig_clkord  <= '0';   -- zruseni zvyseni citace radu metody
324         -- ridici signaly ALU:
325         sig_clkrv   <= '0';   -- zruseni zapisu do RV
326       when VsechnyCleny =>
327         -- ridici signaly citacu:
328         sig_clkvysl <= '1';   -- zvyseni poctu vysledku
329         sig_clkord  <= '0';   -- zruseni zvyseni citace radu metody
330         sig_clrord  <= '1';   -- nulovani citace radu metody
331         -- ridici signaly ALU:
332         sig_clksr   <= '1';   -- ulozeni vysledku do SR
333         sig_clkrv   <= '0';   -- zruseni zapisu vysledku do RV
334       when Vysledek =>
335         -- ridici signaly citacu:
336         sig_clkvysl <= '0';   -- zruseni zvyseni citace poctu vysledku
337         sig_clrord  <= '0';   -- zruseni nulovani citace radu metody
338         -- ridici signaly ALU:
339         sig_clksr   <= '0';   -- zruseni ukladani vysledku do SR
340         VE <= '1';            -- vysledek je aktivni
341       when others =>
342         null;
343     end case;
344   end process;
345
346   integracni_krok: process(sig_rad)
347   begin
348      case sig_rad is
349         -- 16 bitova verze
350         when "0000" => sig_h <= X"1000";  -- h = 0.125
351         when "0001" => sig_h <= X"0800";  -- h/2 = 0.0625
352         when "0010" => sig_h <= X"0555";  -- h/3 = 0.041666666666667
353         when "0011" => sig_h <= X"0400";  -- h/4 = 0.03125
354         when "0100" => sig_h <= X"0333";  -- h/5 = 0.025
355         when "0101" => sig_h <= X"02AA";  -- h/6 = 0.0208333333333
356         when "0110" => sig_h <= X"0249";  -- h/7 = 0.017857142857142857142857142857143
357         when "0111" => sig_h <= X"0200";  -- h/8 = 0.015625
358         when others => sig_h <= X"1000";  -- h = 0.125
359      end case;
360   end process;
361
362   end struct;
```

Výsledky simulace z programu ModelSim jsou na obrázku A.5. Je zde zobrazen okamžik, kdy je vypočtena výsledná hodnota y_1 pomocí 8 členů Taylorovy řady.

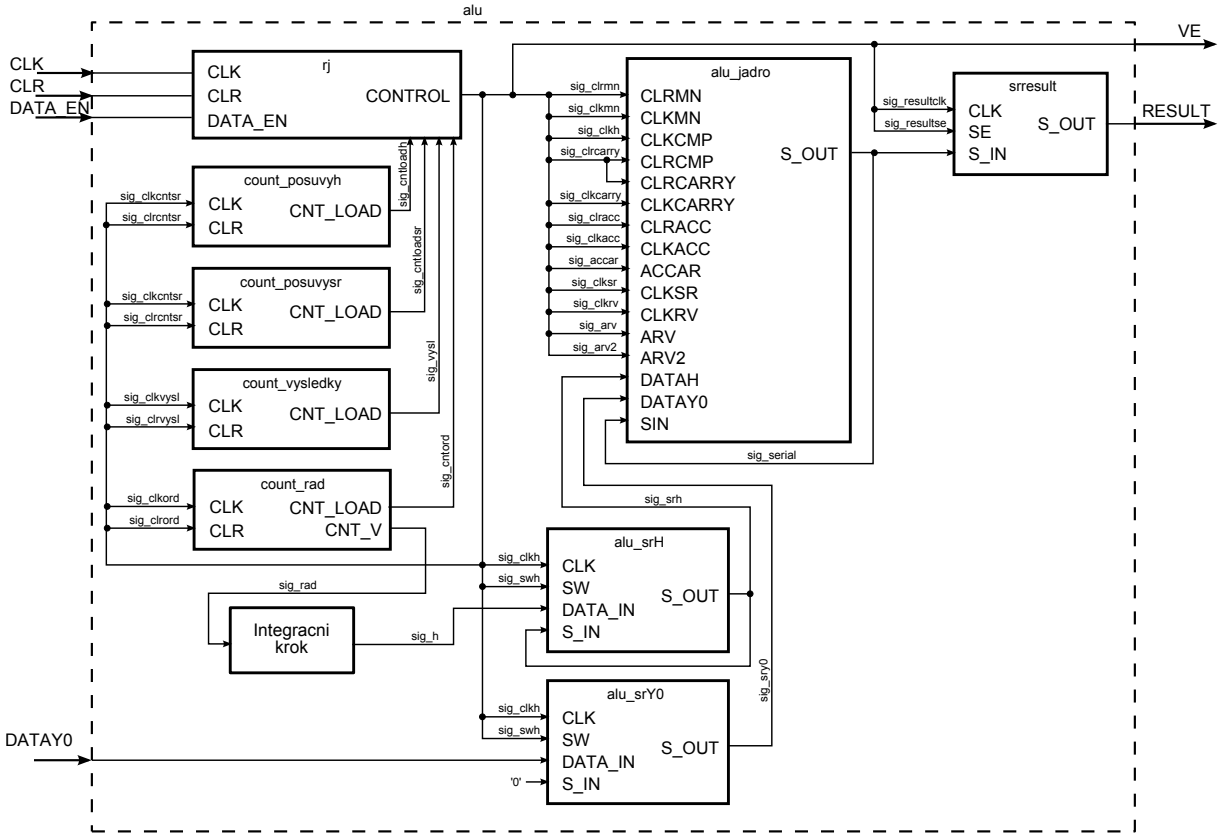


Obrázek A.5: Simulace VHDL kódu sériově-paralelního systému

A.3 Sériově-sériová varianta

Koncepce sériově-sériového integrátoru byla představena v kapitole 6.1.3 a detailnější popis realizace v kapitole 7.1.3. K nahrání počáteční podmínky a integračního kroku a k následnému uložení výsledku jsou použity posuvné registry. Tato koncepce byla představena v kapitole 6.4.2, resp. 6.5.3. Na obrázku A.6 jsou zobrazeny použité komponenty:

- **rj** - řídící jednotka (řadič) která řídí činnost celého systému. Generuje řídící signály pro čítače, aritmeticko-logickou jednotku a pomocné posuvné registry.
- **count_posuvyh** - čítač počítající posuvy integračního kroku
- **count_posuvysr** - čítač počítající posuvy prováděné v SR a ACC
- **count_vysledky** - čítač počítající celkový počet výsledků y_i
- **count_rad** - čítač počítající počet členů Taylorovy řady (řád metody)
- **integracni_krok** - proces generující jednotlivé podíly integračního kroku podle aktuálně počítaného řádu metody
- **alu_jadro** - aritmeticko-logická jednotka (integrátor) provádějící vlastní výpočet
- **alu_srH** - posuvný registr s paralelním vstupem pro nahrání integračního kroku
- **alu_srY0** - posuvný registr s paralelním vstupem pro nahrání počáteční podmínky
- **srresult** - posuvný registr s paralelním výstupem pro uložení výsledku



Obrázek A.6: Specializovaný paralelní systém se sériově-sériovou ALU

```
1  ----------------------------------------------------------------------------
2  -- Specializovany procesor provadejici numerickou integraci
3  ----------------------------------------------------------------------------
4  -- Autor: Michal Kraus
5  -- Datum: 23.3.2011
6  -- Soubor: aritmeticko logicka jednotka - celkove zapojeni
7  -- seriove seriovy integrator, nahrani y0 a h po seriove sbernici
8  ----------------------------------------------------------------------------
9
10 library IEEE;
11 use IEEE.std_logic_1164.all;
12 use ieee.std_logic_arith.all;
13 use ieee.std_logic_unsigned.all;
14
15 entity alu is
16    generic (pocet_bitu: integer);
17    port (
18       CLK: in std_logic;    -- original signaly ALU
19       CLR: in std_logic;
20       DATA_EN: in std_logic;
21       DATAY0: in std_logic_vector(pocet_bitu-1 downto 0);
22       VE: out std_logic;
23       RESULT: out std_logic_vector(pocet_bitu-1 downto 0)
24    );
25 end alu;
26
27 architecture struct of alu is
28
29 -- ============================ komponenty ============================
30 component alu_kernel is
31    generic ( pocet_bitu: integer );
32    port (
33       CLRMN:    in std_logic;
34       CLKMN:    in std_logic;
35       CLRCMP:   in std_logic;
36       CLKCMP:   in std_logic;
37       CLRCARRY: in std_logic;
38       CLKCARRY: in std_logic;
39       CLRACC:   in std_logic;
40       CLKACC:   in std_logic;
41       ACCAR:    in std_logic;
42       CLKRV:    in std_logic;
43       CLKSR:    in std_logic;
44       ARV:      in std_logic_vector(1 downto 0);
45       ARV2:     in std_logic;
46       SIN:      in std_logic;
47       DATAY0:   in std_logic;
48       DATAH:    in std_logic;
49       SOUT:     out std_logic
50    );
51 end component;
52
53 component counter is
54    generic (
55       range_c: integer
56    );
57    port (
58       CLK: in std_logic;
59       CLR: in std_logic;
60       CNT_LOAD: out std_logic
61    );
62 end component;
63
64 component counter2 is
65    port (
66       CLK: in std_logic;
67       CLR: in std_logic;
68       CNT_LOAD: out std_logic;
```

```
69          CNT_V: out std_logic_vector(3 downto 0)
70      );
71  end component;
72
73  component shreg is
74      generic ( n: integer );
75      port (
76          DATA_IN: in std_logic_vector(n-1 downto 0);
77          S_IN: in std_logic;
78          SW: in std_logic;
79          CLK: in std_logic;
80          S_OUT: out std_logic
81      );
82  end component;
83
84  component shreg_parout is
85      generic ( n: integer );
86      port (
87          DATA_OUT: out std_logic_vector(n-1 downto 0);
88          S_IN:     in std_logic;
89          SE:       in std_logic;
90          CLK:      in std_logic
91      );
92  end component;
93
94  -- ======================= vlastni popis alu  ============================
95  -- definice stavu automatu, ktery ridi vypocet ALU
96  -- stavy pro nahrani Y0 do integratoru
97  type TStav is (Init,LoadY0andH,WriteY0Disable,ShiftY0Enable,ShiftY0,ShiftY0Complete,
        MpxSwitch,
98  -- stavy pro vypocet numericke integrace
99  Computing, ShiftH, ShiftHComplete, ShiftACC, ShiftACCComplete, ShiftInput,
        ShiftInputComplete,
100 -- stavy pro ulozeni vysledku nasobeni DY do SR a secteni tohoto vysledku s obsahem
        RV (yi)
101 MpxSwitch2, ShiftLastBit, ShiftLastBitComplete, ShiftRegisters,
        ShiftRegistersComplete,
102 -- stavy pro zvyseni radu metody a poctu vysledku
103 CounterOrd, StoreHP, OrdComplete, ShiftResult, ShiftResultComplete,
        ShiftLastBitResult, ShiftLastBitResultComplete,
104 -- stavy pro seriove nahrani posledniho vysledku do registru SRRESULT
105 ResultsComplete, ShiftResultValue, ShiftResultValueComplete, ShiftResultValueLastBit
        );
106
107 -- paralelni datovy signal pro integracni krok
108 signal sig_h:  std_logic_vector(pocet_bitu-1 downto 0);   -- integracni krok h/p
109 -- seriove datove signaly vstupu/vystupu - seriove propojeni integratoru
110 signal sig_serial: std_logic;
111
112 -- ridici signaly ALU (ridici sbernice):
113 signal sig_clrmn, sig_clkmn, sig_clrcarry, sig_clkcarry, sig_clracc, sig_clkacc,
        sig_accar, sig_clksr, sig_clkrv, sig_arv2 : std_logic;
114 signal sig_arv: std_logic_vector(1 downto 0);
115 -- ridici signaly citacu:
116 -- citace posuvu integracniho kroku (zaroven akumulatoru) a SR (= vstupu integratoru
        )  - serio-seriove nasobeni
117 signal sig_clkcnth, sig_clrcnth, sig_cntloadh, sig_clkcntsr, sig_clrcntsr,
        sig_cntloadsr,
118 -- citace radu metody a poctu vysledku
119 sig_clkord, sig_clrord, sig_cntord, sig_clkvysl, sig_clrvysl, sig_vysl : std_logic;
120
121 -- datove signaly slouzici pro postupne nahrani y0 a h
122 signal sig_sry0, sig_srh: std_logic;
123
124 -- ridici signaly pro postupne nahrani y0 a h a vysledku
125 signal sig_swy0, sig_clky0, sig_swh, sig_clkh, sig_resultse, sig_resultclk:
        std_logic;
126
```

```
127 -- uchovani stavu automatu, ktery ridi vypocet ALU
128 signal CurrentState, NextState: TStav;
129 -- hodnota aktualniho radu metody - vystup z citace
130 signal sig_rad: std_logic_vector(3 downto 0);
131
132
133 begin
134 -- propojeni alu_kernel s ridicimi signaly
135 alu_jadro: alu_kernel
136    generic map ( pocet_bitu => pocet_bitu )
137    port map (
138       CLRMN    => sig_clrmn,
139       CLKMN    => sig_clkmn,
140       CLRCMP   => sig_clrcarry,
141       CLKCMP   => sig_clkh,
142       CLRCARRY => sig_clrcarry,
143       CLKCARRY => sig_clkcarry,
144       CLRACC   => sig_clracc,
145       CLKACC   => sig_clkacc,
146       ACCAR    => sig_accar,
147       CLKSR    => sig_clksr,
148       CLKRV    => sig_clkrv,
149       ARV      => sig_arv,
150       ARV2     => sig_arv2,
151       SIN      => sig_serial,
152       DATAY0   => sig_sry0,
153       DATAH    => sig_srh,
154       SOUT     => sig_serial
155    );
156
157 count_posuvyh: counter
158    generic map (
159       range_c => pocet_bitu
160    )
161    port map (
162       CLK       => sig_clkcnth,
163       CLR       => sig_clrcnth,
164       CNT_LOAD  => sig_cntloadh
165    );
166
167 count_posuvysr: counter
168    generic map (
169       range_c => pocet_bitu
170    )
171    port map (
172       CLK       => sig_clkcntsr,
173       CLR       => sig_clrcntsr,
174       CNT_LOAD  => sig_cntloadsr
175    );
176
177 count_vysledky: counter
178   generic map (
179       -- 2 = pouze jeden vysledek (x = i-1; x => pocet pozadovanych vysledku)
180       range_c => 6              -- => i
181    )
182    port map (
183       CLK       => sig_clkvysl,
184       CLR       => sig_clrvysl,
185       CNT_LOAD  => sig_vysl
186    );
187
188 count_rad: counter2
189    port map (
190       CLK       => sig_clkord,
191       CLR       => sig_clrord,
192       CNT_LOAD  => sig_cntord,
193       CNT_V     => sig_rad
194    );
```

```
195
196 -- posuvny registr na vystupu integratoru (buffer pro vysledek)
197 srresult: shreg_parout
198    generic map ( n => pocet_bitu )
199    port map (
200       DATA_OUT  => RESULT,
201       S_IN      => sig_serial,
202       SE        => sig_resultse,
203       CLK       => sig_resultclk
204    );
205
206 -- posuvny registr pro nahravani pocatecni podminky
207 alu_srY0: shreg
208    generic map ( n => pocet_bitu )
209    port map (
210       DATA_IN  => DATAY0,
211       S_IN     => '0',
212       SW       => sig_swy0,
213       CLK      => sig_clky0,
214       S_OUT    => sig_sry0
215    );
216
217 -- posuvny registr pro nahravani integracniho kroku
218 alu_srH: shreg
219    generic map ( n => pocet_bitu )
220    port map (
221       DATA_IN  => sig_h,
222       S_IN     => sig_srh,    -- funguje jako kruhovy registr
223       SW       => sig_swh,
224       CLK      => sig_clkh,
225       S_OUT    => sig_srh
226    );
227
228 -------------------------------------------------------------------------------
229 -- Konecny automat:
230 -- stavovy registr - prepnuti na nasledujici stav
231 Clock: process(CLR,CLK)
232 begin
233   if CLR = '1' then
234     CurrentState <= Init;
235   elsif CLK'event and CLK='1' then
236     CurrentState <= NextState;
237   end if;
238 end process;
239
240 -- zvoleni nasledujiciho stavu
241 NaslStav: process(CurrentState,DATA_EN)
242 begin
243   -- nahrani pocatecni podminky do SR a RV
244   case CurrentState is
245     when Init =>
246         if DATA_EN = '1' then
247             NextState <= LoadY0andH;
248         else
249             NextState <= Init;
250         end if;
251     when LoadY0andH =>
252         NextState <= WriteY0Disable;
253     when WriteY0Disable =>
254         NextState <= ShiftY0Enable;
255     when ShiftY0Enable =>
256         NextState <= ShiftY0;
257     when ShiftY0 =>
258         NextState <= ShiftY0Complete;
259     when ShiftY0Complete =>
260         if sig_cntloadh = '1' then
261             NextState <= MpxSwitch;
262         else
```

```
263             NextState <= ShiftY0;
264         end if;
265
266     --vlastni vypocet numericke integrace
267     when MpxSwitch =>
268         NextState <= Computing;
269     when Computing =>
270         NextState <= ShiftH;
271     when ShiftH =>
272         NextState <= ShiftHComplete;
273     when ShiftHComplete =>
274         if sig_cntloadh = '1' then
275             NextState <= ShiftACC;
276         else
277             NextState <= ShiftH;
278         end if;
279     -- musime ulozit posledni bit mezivysledku do ACC po poslednim shiftovani SRH
280     when ShiftACC =>
281         NextState <= ShiftACCComplete;
282     -- a testujeme zda tento vypocet uz byl posledni - jsou zpracovany vsechny bity
        SR
283     when ShiftACCComplete =>
284         if sig_cntloadsr = '1' then
285             NextState <= MpxSwitch2;
286         else
287             NextState <= ShiftInput;
288         end if;
289     -- je kompletni secteni vsech bitu integracniho kroku h/p aktualnim bitem na
        vstupu
290     -- nyni nasleduje posuv vstupu (neboli SR) a odpovidajici posuv ACC
291     -- a musime posunout i registr SRH abychom dostali int. krok do puvodniho tvaru
292     when ShiftInput =>
293         NextState <= ShiftInputComplete;
294     when ShiftInputComplete =>
295         NextState <= MpxSwitch;
296     -- kompletni vynasobeni celym vstupem => nasobeni
297     -- musime ulozit vysledek nasobeni z ACC do SR
298     -- a zaroven provest secteni obsahu ACC a RV => DY (ACC) + yi (RV)
299     when MpxSwitch2 =>
300         NextState <= ShiftRegisters;
301     when ShiftRegisters =>
302         NextState <= ShiftRegistersComplete;
303     when ShiftRegistersComplete =>
304         if sig_cntloadsr = '1' then
305             NextState <= ShiftLastBit;
306         else
307             NextState <= ShiftRegisters;
308         end if;
309     -- jeste nam zbyva ulozit do registru posledni bit vysledku
310     when ShiftLastBit =>
311         NextState <= ShiftLastBitComplete;
312     when ShiftLastBitComplete =>
313         NextState <= CounterOrd;
314     -- ulozeni mame kompletni
315     -- nyni musime zvysit rad metody - citac cntord a prepnout SRH do rezimu zapis
316     -- nahrat novou hodnotu integracniho kroku do SRH a vynulovat citac cntsr
317     when CounterOrd =>
318         NextState <= StoreHP;
319     when StoreHP =>
320         if sig_cntord = '1' then
321             NextState <= OrdComplete;
322         else
323             NextState <= MpxSwitch;  -- znovu se spusti cely proces nasobeni
324         end if;
325     when OrdComplete =>
326         NextState <= ShiftResult;
327     -- vypocet je kompletni - dosahli jsme pozadovane presnosti
328     -- nahrajeme vysledek y(i+1) do SR - pocatecni podminka pro vypocet y(i+2)
```

```
329      when ShiftResult =>
330          NextState <= ShiftResultComplete;
331      when ShiftResultComplete =>
332          if sig_cntloadsr = '1' then
333              NextState <= ShiftLastBitResult;
334          else
335              NextState <= ShiftResult;
336          end if;
337      when ShiftLastBitResult =>
338          NextState <= ShiftLastBitResultComplete;
339      -- otestujeme, zda jsme dosahli pozadovaneho poctu vysledku
340      when ShiftLastBitResultComplete =>
341          if sig_vysl = '1' then
342              NextState <= ResultsComplete;
343          else
344              NextState <= MpxSwitch;
345          end if;
346      when ResultsComplete =>
347          NextState <= ShiftResultValue;
348      when ShiftResultValue =>
349          NextState <= ShiftResultValueComplete;
350      when ShiftResultValueComplete =>
351          if sig_cntloadsr = '1' then
352              NextState <= ShiftResultValueLastBit;
353          else
354              NextState <= ShiftResultValue;
355          end if;
356      when ShiftResultValueLastBit =>
357          NextState <= Init;
358      when others =>
359        null;
360    end case;
361  end process;
362
363  -- podle stavu generuje vystup - ridici signaly mikroprogramu
364  Vystup: process(CurrentState)
365  begin
366    VE <= '0';       -- default: vysledek neni aktivni
367    case CurrentState is
368      when Init =>
369        -- ridici signaly citacu:
370        sig_clkcnth  <= '0';   -- pricteni dalsiho posuvu registru
371        sig_clrcnth  <= '0';   -- vynulovani poctu posuvu
372        sig_clkcntsr <= '0';   -- citac poctu posuvu sr
373        sig_clrcntsr <= '0';   -- nulovani poctu posuvu
374        sig_clkord   <= '0';   -- dalsi clen Taylorovy rady
375        sig_clrord   <= '1';   -- vynulovani poctu clenu Taylorovy rady - kvuli ORD =
               "0000"
376        sig_clrvysl  <= '1';   -- vynulovani poctu vysledku
377        sig_clkvysl  <= '0';   -- pricteni vysledku
378        -- ridici signaly ALU:
379        sig_clrcarry <= '0';
380        sig_clkcarry <= '0';
381        sig_clracc   <= '0';
382        sig_clkacc   <= '0';
383        sig_accar    <= '0';
384        sig_clrmn    <= '0';
385        sig_clkmn    <= '0';
386        sig_clksr    <= '0';
387        sig_clkrv    <= '0';
388        sig_arv      <= "00";  -- vstup mpx s pocatecni podminkou
389        sig_arv2     <= '1';   -- vstup mpx ze scitacky a ne z ACC
390        -- ridici signaly serioveho prenosu dat:
391        sig_swy0     <= '0';   -- write do posuvneho registru SRY0
392        sig_clky0    <= '0';   -- nic do registru SRY0 nenahravame
393        sig_swh      <= '0';   -- write do posuvneho registru SRH
394        sig_clkh     <= '0';   -- nic do registru SRH nenahravame
395        sig_resultse <= '0';   -- vypneme zapis do SRRESULT
```

```
396         sig_resultclk<= '0';    -- nic nezapisujeme
397         -- ridici signal, ze vysledek je hotov
398         VE <= '0';
399
400       when LoadY0andH =>
401         -- ridici signaly citacu:
402         sig_clrcnth  <= '1';    -- vynulovani poctu posuvu
403         sig_clrord   <= '0';    -- zruseni nulovani citace radu metody
404         sig_clrvysl  <= '0';    -- zruseni nulovani citace poctu vysledku
405         -- ridici signaly ALU:
406         sig_clrmn    <= '1';    -- nulovani obvodu mala nula MN
407         sig_clracc   <= '1';    -- vynulujeme ACC
408         sig_clrcarry <= '1';    -- vynulujeme prenos scitacky
409         -- ridici signaly serioveho prenosu dat:
410         sig_swy0     <= '0';    -- write do posuvneho registru SRY0
411         sig_clky0    <= '1';    -- nahrajeme do registru SRY0 poc. podminku ze vstupu
412         sig_swh      <= '0';    -- write do posuvneho registru SRH
413         sig_clkh     <= '1';    -- nahrajeme do registru SRH hodnotu int. kroku
414
415       when WriteY0Disable =>
416         -- ridici signaly citacu:
417         sig_clrcnth  <= '0';    -- zrusime nulovani poctu posuvu
418         -- ridici signaly ALU:
419         sig_clrmn    <= '0';    -- zruseni nulovani obvodu mala nula MN
420         sig_clracc   <= '0';    -- zrusime nuluvani ACC
421         sig_clrcarry <= '0';    -- zrusime nulovani prenosu scitacky
422         -- ridici signaly serioveho prenosu dat:
423         sig_clky0    <= '0';    -- zrusime hodiny nahravani do SRY0
424         sig_clkh     <= '0';    -- zrusime hodiny nahravani do SRH
425
426       when ShiftY0Enable =>
427         -- ridici signaly ALU:
428         sig_clksr    <= '1';    -- posuneme SR - nahravame 1. bit pocatecni podminky
429         sig_clkrv    <= '1';    -- posuneme RV - nahravame 1. bit pocatecni podminky
430         -- ridici signaly serioveho prenosu dat:
431         sig_swy0     <= '1';    -- registr SRY0 nastavime na posuv
432         sig_clky0    <= '0';    -- neposouvame registr SRY0
433
434       when ShiftY0 =>
435         -- ridici signaly citacu:
436         sig_clkcnth  <= '1';    -- pricteni dalsiho posuvu registru
437         sig_clrcntsr <= '0';    -- zruseni vynulovani poctu posuvu SR (vystupu/vstupu)
438         -- ridici signaly ALU:
439         sig_clksr    <= '0';    -- zrusime posuv SR
440         sig_clkrv    <= '0';    -- zrusime posuv RV
441         -- ridici signaly serioveho prenosu dat:
442         sig_swy0     <= '1';    -- registr SRY0 nechame nastaveny na posuv
443         sig_clky0    <= '1';    -- posuneme SRY0
444
445       when ShiftY0Complete =>
446         -- ridici signaly citacu:
447         sig_clkcnth  <= '0';    -- zruseni pricteni dalsiho posuvu registru
448         sig_clrcntsr <= '1';    -- vynulovani poctu posuvu SR (vystupu/vstupu)
449         -- ridici signaly ALU:
450         sig_clksr    <= '1';    -- posuneme SR - nahravame dalsi bity pocatecni
                podminky
451         sig_clkrv    <= '1';    -- posuneme RV - nahravame dalsi bity pocatecni
                podminky
452         -- ridici signaly serioveho prenosu dat:
453         sig_swy0     <= '1';    -- registr SRY0 nechame nastaveny na posuv
454         sig_clky0    <= '0';    -- zrusime hodiny posuvu registru SRY0
455
456       when MpxSwitch =>
457         -- ridici signaly citacu:
458         sig_clrcnth  <= '1';    -- vynulovani poctu posuvu
459         sig_clrcntsr <= '0';    -- zruseni vynulovani poctu posuvu SR (vystupu/vstupu)
460         -- ridici signaly ALU:
```

```
461         sig_clrcarry <= '1';   -- nulovani prenosu scitacky a bloku tvorici seriove
                doplnek
462         sig_clksr    <= '0';   -- zrusime posledni posuv SR
463         sig_clkrv    <= '0';   -- zrusime posledni posuv RV
464         sig_arv      <= "01";  -- vstup mpx pripojen na vstup integratoru
465         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
466         -- ridici signaly serioveho prenosu dat:
467         sig_swh      <= '1';   -- nastavime registr SRH na posuv
468         sig_clkh     <= '0';   -- ukoncime zapis do registru SRH hodnoty int. kroku
469
470       when Computing =>
471         -- ridici signaly citacu:
472         sig_clrcnth  <= '0';   -- zruseni vynulovani poctu posuvu
473         sig_clrcntsr <= '0';   -- zruseni vynulovani poctu posuvu SR (vystupu/vstupu)
474         -- ridici signaly ALU:
475         sig_clrcarry <= '0';   -- zruseni nulovani prenosu scitacky
476         sig_arv      <= "01";  -- vstup mpx pripojen na vstup integratoru
477         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
478         -- ridici signaly serioveho prenosu dat:
479         sig_swh      <= '1';   -- nechame nastaveny SRH na posuv
480
481       when ShiftH =>
482         -- ridici signaly citacu:
483         sig_clkcnth  <= '1';   -- citac posuvu integracniho kroku SRH
484         -- ridici signaly ALU:
485         sig_clkcarry <= '1';   -- zapiseme prenos ze scitacky do carry
486         sig_clkacc   <= '1';   -- zapiseme mezivysledek do ACC
487         sig_arv      <= "01";  -- vstup mpx pripojen na vstup integratoru
488         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
489         -- ridici signaly serioveho prenosu dat:
490         sig_swh      <= '1';   -- nechame nastaveny SRH na posuv
491         sig_clkh     <= '1';   -- posuneme integracni krok v SRH
492
493       when ShiftHComplete =>
494         -- ridici signaly citacu:
495         sig_clkcnth  <= '0';   -- zruseni zvyseni citace posuvu
496         -- ridici signaly ALU:
497         sig_clkcarry <= '0';   -- zruseni zapisu prenosu ze scitacky do carry
498         sig_clkacc   <= '0';   -- zruseni zapisu mezivysledeku do ACC
499         sig_arv      <= "01";  -- vstup mpx pripojen na vstup integratoru
500         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
501         -- ridici signaly serioveho prenosu dat:
502         sig_swh      <= '1';   -- nechame nastaveny SRH na posuv
503         sig_clkh     <= '0';   -- zrusime posun integracniho kroku v SRH
504
505       when ShiftACC =>
506         -- ridici signaly ALU:
507         sig_clkmn    <= '1';   -- ulozime do KO D (male nuly) hodnotu vstupu
508         sig_clkacc   <= '1';   -- zapiseme posledni bit mezivysledku do ACC
509
510       when ShiftACCComplete =>
511         -- ridici signaly ALU:
512         sig_clkmn    <= '0';   -- zrusime ulozeni do male nuly
513         sig_clkacc   <= '0';   -- zrusime zapis posledniho bitu mezivysledku do ACC
514         sig_accar    <= '1';   -- nastavime ACC na aritmeticky posuv
515         -- ridici signaly serioveho prenosu dat:
516         sig_resultclk<= '1';   -- zapiseme bit vysledku do SRRESULT
517
518       when ShiftInput =>
519         -- ridici signaly citacu:
520         sig_clkcntsr <= '1';   -- citac posuvu registru SR - vstupu/vystupu
                integratoru
521         -- ridici signaly ALU:
522         sig_clkacc   <= '1';   -- posuv ACC soucasne s
523         sig_clksr    <= '1';   -- posuvem SR
524         sig_arv      <= "01";  -- vstup mpx pripojen na vstup integratoru
525         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
526         -- ridici signaly serioveho prenosu dat:
```

```
527         sig_swh      <= '1';   -- nechame nastaveny SRH na posuv
528         sig_clkh     <= '1';   -- posuneme integracni krok v SRH => dostaneme hodnotu
                v registru SRH do stavu pred posuvy
529         sig_resultclk<= '0';   -- zrusime zapis bitu vysledku do SRRESULT
530
531       when ShiftInputComplete =>
532         -- ridici signaly citacu:
533         sig_clkcntsr <= '0';   -- zrusime hodiny posuvu registru SR
534         -- ridici signaly ALU:
535         sig_clkacc   <= '0';   -- zrusime zapis posledniho bitu mezivysledku do ACC
536         sig_accar    <= '0';   -- zrusime aritmeticky posuv ACC
537         sig_clksr    <= '0';   -- zrusime posuv SR
538         sig_arv      <= "01";  -- vstup mpx pripojen na vstup integratoru
539         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
540         -- ridici signaly serioveho prenosu dat:
541         sig_swh      <= '1';   -- nechame nastaveny SRH na posuv
542         sig_clkh     <= '0';   -- zrusime posun integracniho kroku v SRH
543
544       when MpxSwitch2 =>
545         -- ridici signaly citacu:
546         sig_clrcnth  <= '1';   -- vynulovani poctu posuvu
547         sig_clrcntsr <= '1';   -- vynulovani poctu posuvu SR (vystupu/vstupu)
548         -- ridici signaly ALU:
549         sig_clrcarry <= '1';   -- nulovani prenosu scitacky a bloku tvorici seriove
                doplnek
550         sig_accar    <= '0';   -- zrusime aritmeticky posuv ACC
551         sig_arv      <= "10";  -- vstup mpx pripojen na vystup z RV
552         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
553         -- ridici signaly serioveho prenosu dat:
554         sig_swh      <= '1';   -- nastavime registr SRH na posuv
555         sig_resultse <= '0';   -- zakazeme uz zapis do SRRESULT - vysledek je ulozen
556         sig_resultclk<= '0';   -- zrusime zapis bitu vysledku do SRRESULT
557         VE           <= '1';   -- priznak ze je vysledek aktivni
558
559       when ShiftRegisters =>
560         -- ridici signaly citacu:
561         sig_clrcnth  <= '0';   -- zruseni vynulovani poctu posuvu
562         sig_clrcntsr <= '0';   -- zruseni vynulovani poctu posuvu SR (vystupu/vstupu)
563         sig_clkcntsr <= '1';   -- zvyseni citace posuvu SR,RV a ACC
564         -- ridici signaly ALU:
565         sig_clrcarry <= '0';   -- zruseni nulovani prenosu scitacky
566         sig_clkcarry <= '1';   -- ulozeni carry
567         sig_clkacc   <= '1';   -- ulozeni mezivysledku y(i+1) do ACC
568         sig_clkrv    <= '1';   -- ulozeni toho stejneho mezivysledku y(i+1) do RV
569         sig_clksr    <= '1';   -- ulozeni vysledku nasobeni DY z ACC do SR
570         sig_arv      <= "10";  -- vstup mpx pripojen na vystup z RV
571         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
572         VE           <= '0';   -- zrusime priznak ze je vysledek aktivni
573
574       when ShiftRegistersComplete =>
575         -- ridici signaly citacu:
576         sig_clkcntsr <= '0';   -- zruseni zvyseni citace posuvu SR,RV a ACC
577         -- ridici signaly ALU:
578         sig_clkcarry <= '0';   -- zruseni ulozeni carry
579         sig_clkacc   <= '0';   -- zruseni ulozeni mezivysledku y(i+1) do ACC
580         sig_clkrv    <= '0';   -- zruseni ulozeni toho stejneho mezivysledku y(i+1) do
                RV
581         sig_clksr    <= '0';   -- zruseni ulozeni vysledku nasobeni DY z ACC do SR
582         sig_arv      <= "10";  -- vstup mpx pripojen na vystup z RV
583         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
584
585       when ShiftLastBit =>
586         -- ridici signaly ALU:
587         sig_clkacc   <= '1';   -- ulozeni posledniho bitu mezivysledku y(i+1) do ACC
588         sig_clkrv    <= '1';   -- ulozeni toho stejneho posledniho bitu mezivysledku y
                (i+1) do RV
589         sig_clksr    <= '1';   -- ulozeni posledniho bitu vysledku nasobeni DY z ACC
                do SR
```

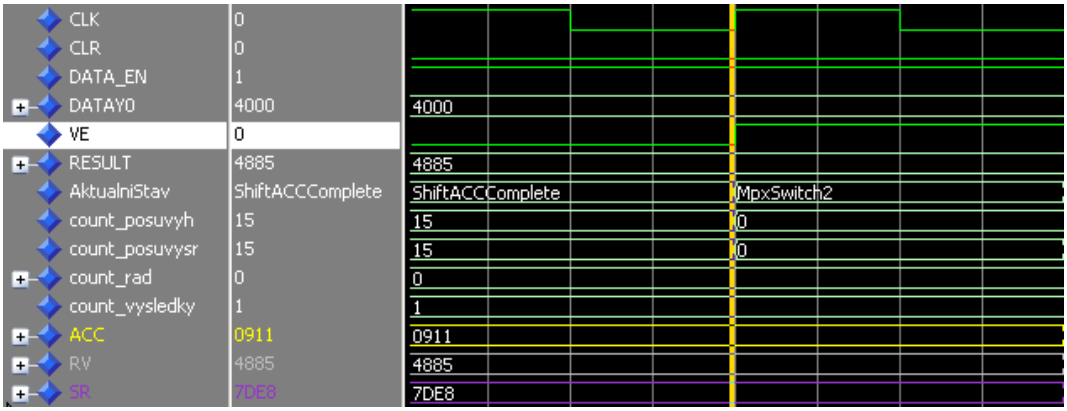
```
590         sig_arv      <= "10";  -- vstup mpx pripojen na vystup z RV
591         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
592
593       when ShiftLastBitComplete =>
594         -- ridici signaly ALU:
595         sig_clkacc   <= '0';   -- zruseni ulozeni posledniho bitu mezivysledku y(i+1)
              do ACC
596         sig_clkrv    <= '0';   -- zruseni ulozeni toho stejneho posledniho bitu
              mezivysledku y(i+1) do RV
597         sig_clksr    <= '0';   -- zruseni ulozeni posledniho bitu vysledku nasobeni DY
               z ACC do SR
598         sig_arv      <= "10";  -- vstup mpx pripojen na vystup z RV
599         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
600
601       when CounterOrd =>
602         -- ridici signaly citacu:
603         sig_clkord   <= '1';   -- dalsi clen Taylorovy rady
604         -- ridici signaly serioveho prenosu dat:
605         sig_swh      <= '0';   -- nastavime write do posuvneho registru SRH
606
607       -- nahrat novou hodnotu integracniho kroku do SRH a vynulovat citac cntsr
608       -- v tomto stavu se testuje ORD
609       when StoreHP =>
610         -- ridici signaly citacu:
611         sig_clkord   <= '0';   -- zruseni zvyseni citace radu metody
612         sig_clrcntsr <= '1';   -- vynulovani citace posuvu
613         -- ridici signaly serioveho prenosu dat:
614         sig_swh      <= '0';   -- nechame nastaveny write do posuvneho registru SRH
615         sig_clkh     <= '1';   -- nahrajeme do registru SRH hodnotu int. kroku
616
617     when OrdComplete =>
618         -- ridici signaly citacu:
619         sig_clrcntsr <= '0';   -- zruseni nulovani citace posuvu
620         sig_clrord   <= '1';   -- nulovani citace radu metody
621         sig_clkvysl  <= '1';   -- zvyseni poctu vysledku
622         -- ridici signaly ALU:
623         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
624         -- ridici signaly serioveho prenosu dat:
625         sig_swh      <= '0';   -- nechame nastaveny write do posuvneho registru SRH
626         sig_clkh     <= '0';   -- zrusime zapis hodnoty int. kroku do registru SRH
627
628       -- musime zajistit ulozeni vysledku taky do SR
629       when ShiftResult =>
630         -- ridici signaly citacu:
631         sig_clkcntsr <= '1';   -- zvyseni citace posuvu SR
632         sig_clrord   <= '0';   -- zruseni nulovani citace radu metody
633         sig_clkvysl  <= '0';   -- zruseni zvyseni poctu vysledku
634         -- ridici signaly ALU:
635         sig_clkacc   <= '1';   -- posuv acc
636         sig_clksr    <= '1';   -- ulozeni vysledku z ACC do SR
637         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
638
639       when ShiftResultComplete =>
640         -- ridici signaly citacu:
641         sig_clkcntsr <= '0';   -- zruseni zvyseni citace posuvu SR,RV a ACC
642         -- ridici signaly ALU:
643         sig_clkacc   <= '0';   -- zruseni posuvu ACC
644         sig_clksr    <= '0';   -- zruseni ukladani vysledku do SR
645         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
646
647       when ShiftLastBitResult =>
648         -- ridici signaly ALU:
649         sig_clksr    <= '1';   -- ulozeni posledniho bitu vysledku z ACC do SR
650         sig_arv2     <= '0';   -- vstup mpx pripojen na vystup z ACC
651
652       when ShiftLastBitResultComplete =>
653         -- ridici signaly citacu:
654         sig_clrcntsr <= '1';   -- vynulovani citace posuvu SR
```

```
655         -- ridici signaly ALU:
656         sig_clksr    <= '0';   -- zruseni ulozeni posledniho bitu vysledku z ACC do SR
657         -- ridici signaly serioveho prenosu dat:
658         sig_resultse <= '1';   -- povolime zapis do SRRESULT
659
660       when ResultsComplete =>
661         -- ridici signaly citacu:
662         sig_clrcntsr <= '0';   -- zruseni vynulovani citace posuvu SR
663         -- ridici signaly serioveho prenosu dat:
664         sig_resultse <= '1';   -- nechame povoleny zapis do SRRESULT
665
666       when ShiftResultValue =>
667         -- ridici signaly citacu:
668         sig_clkcntsr <= '1';   -- hodiny citace poctu posuvu
669         -- ridici signaly ALU:
670         sig_clksr    <= '1';   -- posuv registru SR
671         -- ridici signaly serioveho prenosu dat:
672         sig_resultse <= '1';   -- nechame povoleny zapis do SRRESULT
673         sig_resultclk<= '1';   -- zapisujeme postupne vysledek
674
675       when ShiftResultValueComplete =>
676         -- ridici signaly citacu:
677         sig_clkcntsr <= '0';   -- zrusime hodiny citace poctu posuvu
678         -- ridici signaly ALU:
679         sig_clksr    <= '0';   -- zrusime posuv registru SR
680         -- ridici signaly serioveho prenosu dat:
681         sig_resultse <= '1';   -- nechame povoleny zapis do SRRESULT
682         sig_resultclk<= '0';   -- zrusime zapis
683
684       when ShiftResultValueLastBit =>
685         -- ridici signaly citacu:
686         sig_clrcntsr <= '1';   -- vynulovani citace posuvu SR
687         -- ridici signaly serioveho prenosu dat:
688         sig_resultse <= '1';   -- nechame povoleny zapis do SRRESULT
689         sig_resultclk<= '1';   -- zapiseme posledni bit vysledku
690         VE <= '1';             -- vysledek je aktivni
691
692       when others =>
693         null;
694     end case;
695   end process;
696
697   integracni_krok: process(sig_rad)
698   begin
699      case sig_rad is
700         -- 16 bitova verze
701         when "0000" => sig_h <= X"1000";  -- h = 0.125
702         when "0001" => sig_h <= X"0800";  -- h/2 = 0.0625
703         when "0010" => sig_h <= X"0555";  -- h/3 = 0.041666666666667
704         when "0011" => sig_h <= X"0400";  -- h/4 = 0.03125
705         when "0100" => sig_h <= X"0333";  -- h/5 = 0.025
706         when "0101" => sig_h <= X"02AA";  -- h/6 = 0.0208333333333
707         when "0110" => sig_h <= X"0249";  -- h/7 = 0.017857142857142857142857142857143
708         when "0111" => sig_h <= X"0200";  -- h/8 = 0.015625
709         when others => sig_h <= X"1000";  -- h = 0.125
710      end case;
711   end process;
712
713   end struct;
```

Výsledky simulace z programu ModelSim jsou na obrázku A.7. Je zde zobrazen okamžik, kdy je vypočtena výsledná hodnota y_1 pomocí 8 členů Taylorovy řady.

A.4 VHDL implementace pro přípravek FITkit

Pro vytvoření funkční aplikace jsou potřebné následující soubory:



Obrázek A.7: Simulace VHDL kódu sériově-sériového systému

- `main.c` - soubor obsahující kód pro mikrokontroler
- `alu.vhd` - vlastní paralelní systém (viz. popis v předcházející části)
- `top_level.vhd` - top level kód pro FPGA, tzn. propojení paralelního systému s komunikačním systémem. Obdoba test bench `alu_tb.vhd` souboru sloužícího pro simulaci.

V top-level architektuře se musí instancovat paralelní systém a dále komunikační systém pro distribuci počátečních dat a výsledků. Jsou zvoleny dva možné přístupy: pomocí SPI řadiče s využitím přerušení, které se zpracovává mikrokontrolerem nebo druhá možnost s využitím komponenty `serial_transceiver`.

A.4.1 Využití přerušovacího systému

Výpočet se ovládá přes terminál. Paralelní systém (komponenta ALU) je připojena na řadič přerušení. Ten po výpočtu výsledku - signál VE je aktivní, dá vědět pomocí přerušení mikrokontroleru (MCU) a ten zobrazí výsledná data na terminál.

Propojení usnadňuje navržený propojovací systém popsaný na webových stránkách [FIT11] v části Komunikační systém. Naším úkolem je pouze instancovat pro každý prvek komunikující s MCU tzv. adresový dekodér SPI_adc, na jehož výstupu máme k dispozici jednoduché paralelní rozhraní, které bude obsahovat požadovaná data.

Na následující ukázce je zobrazen pouze kód týkající se připojení paralelního systému a řadiče přerušení na komunikační rozhraní SPI. Kompletní kód lze nalézt na CD.

```
1   -- SPI -> ALU
2   SPI_adc_alu: SPI_adc
3      generic map(
4         ADDR_WIDTH => 8,        -- sirka adresy 8 bitu
5         DATA_WIDTH => 32,       -- sirka dat 32 bitu
6         ADDR_OUT_WIDTH => 1,    -- sirka adresy na vystupu min. 1 bit
7         BASE_ADDR  => 16#02#,   -- adresovy prostor od 0x02
8         DELAY => 0
9      )
10     port map(
11        CLK       => CLK,
12
13        CS        => SPI_CS,
14        DO        => SPI_DO,
15        DO_VLD    => SPI_DO_VLD,
16        DI        => SPI_DI,
17        DI_REQ    => SPI_DI_REQ,
```

```
18
19          ADDR     => open,
20          DATA_OUT => alu_data_in,
21          DATA_IN  => alu_data_out,
22          WRITE_EN => alu_data_en,
23          READ_EN  => open
24       );
25
26    -- ALU
27    ALU_T: alu
28       generic map (pocet_bitu => n)
29       port map (
30          CLK     => CLK,
31          CLR     => reset,
32          DATA_EN => alu_data_en,
33          DATAY0  => alu_data_in,
34          VE      => interrupt(2),
35          RESULT  => alu_data_out
36       );
37
38    -- SPI -> IRQ
39    SPI_adc_INT: SPI_adc
40       generic map(
41          ADDR_WIDTH => 8,        -- sirka adresy 8 bitu
42          DATA_WIDTH => 8,        -- sirka dat 8 bitu
43          ADDR_OUT_WIDTH => 1,    -- sirka adresy na vystupu min. 1 bit
44          BASE_ADDR  => 16#80#,   -- adresovy prostor 0x80
45          DELAY => 0
46       )
47       port map(
48          CLK      => CLK,
49
50          CS       => SPI_CS,
51          DO       => SPI_DO,
52          DO_VLD   => SPI_DO_VLD,
53          DI       => SPI_DI,
54          DI_REQ   => SPI_DI_REQ,
55
56          ADDR     => open,
57          DATA_OUT => int_data_out,
58          DATA_IN  => int_data_in,
59          WRITE_EN => int_write_en,
60          READ_EN  => int_read_en
61       );
62
63    -- IRQ radic
64    IRQradic: interrupt_controller
65       port map(
66          CLK      => CLK,
67          RST    =>  reset,
68
69          IRQ_IN  => interrupt,
70          IRQ_OUT => interrupt_out,   -- celkove vystupni preruseni
71
72          -- vystupni vektor preruseni
73          DATA_OUT    => int_data_in,
74          READ_EN     => int_read_en,
75
76          -- nastaveni masky preruseni
77          DATA_IN     => int_data_out,
78          WRITE_EN    => int_write_en
79       );
```

V první části se instancuje adresový dekodér a náš paralelní systém. Data budou na tomto příkladě přenášena po 32 bitech, k adresování zařízení připojených na propojovací systém se bude používat 8 bitů. Pomocí SPI budeme přivádět počáteční podmínku (DATA_OUT) a následně z paralelního systému číst výsledek (DATA_IN). Po zadání počáteční

podmínky se pomocí aktivního signálu DATA_EN spustí samotný výpočet. Naše zařízení bude namapováno na adresu 0x02, ze které bude mikrokontroler číst jednotlivé výsledky.

Následuje adresový dekodér pro řadič přerušení. Slouží k propojení signálu VE na řadič přerušení (IRQ_IN). Tímto se indikuje stav, kdy je výsledek aktivní a může se přečíst. Přerušení je zpracováno v mikrokontroleru.

A.4.2 Využití komponenty serial_transceiver

Výpočet se řídí sám a paralelní systémem v FPGA čipu je přímo propojen s počítačem, který získává výsledky, není využit mikrokontroler. Komunikace probíhá přes komponentu serial_transceiver, která slouží k sériovému přenosu dat. V tomto případě není využito SPI rozhraní navrženém pro FITkit. K načtení a zobrazení výsledků na straně počítače je zapotřebí další aplikace - `FITKitRead`, která byla vytvořena pro tyto účely.

```
   alu_serpar: alu
     generic map (
        pocet_bitu => n
     )
     port map (
        CLK => SMCLK,
        CLR => reset,
        DATA_EN => signal_start,
        DATAY0 => X"40000000",    -- napevno zadana pocatecni podminka
        VE => signal_vysledek,
        RESULT => signal_buffer
     );

   serial_tr: serial_transceiver
      generic map (
   --   jednotlive prenosove rychlosti:
   --   s1200Bd, s2400Bd, s4800Bd, s9600Bd, s19200Bd, s38400Bd,
   --   s57600Bd, s115200Bd, s230400Bd, s460800Bd, s921600Bd

         SPEED     => s460800Bd,
         DATAWIDTH => 8,
         STOPBITS  => 1,
         PARITY    => sParityOdd
      )
      port map(
         RESET    => reset,
         CLK      => SMCLK,

         -- cteni dat prijatych po seriove lince
         DATA_OUT => serial_data_out,
         DATA_VLD => serial_vld,

         -- pro vysilani dat na RS232
         DATA_IN  => serial_data_in,
         WRITE_EN => serial_write_en,

         BUSY     => serial_busy,
         ERR      => open,
         ERR_RST  => '0',

         RXD      => AFBUS(0),
         TXD      => AFBUS(1),
         RTS      => AFBUS(2),
         CTS      => open
      );


-- Stavovy registr automatu
   pstatereg: process(reset, smclk)
   begin
```

```
51        if (reset='1') then
52           pstate <= SIdle;
53        elsif (smclk'event) and (smclk='1') then
54           pstate <= nstate;
55        end if;
56     end process;
57
58  -- Vlastni automat - urceni pristiho stavu a vystupu
59     nstate_logic: process(pstate, serial_vld, serial_busy, signal_vysledek)
60     begin
61        nstate <= SIdle;
62        serial_write_en <= '0';
63
64        case pstate is
65              -- Idle
66           when SIdle =>
67                  -- priznaky na pocatecni hodnoty
68                  signal_start <= '0';      -- nastartovani vypoctu
69              if (serial_vld = '1') then
70                 nstate <= SReceiving;
71              end if;
72
73        -- Receiving
74           when SReceiving =>
75                 signal_start <= '1';
76                           nstate <= SComputation;
77
78        -- Computation
79           when  SComputation =>
80              nstate <= SComputation;
81              signal_start <= '0';
82              if (signal_vysledek = '1') then
83                 nstate <= SStorage;
84              end if;
85
86        -- Storage
87           when SStorage =>
88              buffer_nb <= signal_buffer;
89              nstate <= STransfer1;
90
91        -- Transfer1
92           when STransfer1 =>
93              nstate <= STransfer1;
94
95              if (serial_busy= '0') then
96                 serial_write_en <= '1';
97                 serial_data_in <= buffer_nb(31 downto 24);
98                 nstate <= STransfer2;
99              else
100                serial_write_en <= '0';
101             end if;
102
103       -- Transfer2
104          when STransfer2 =>
105             nstate <= STransfer2;
106
107             if (serial_busy= '0') then
108                serial_write_en <= '1';
109                serial_data_in <= buffer_nb(23 downto 16);
110                nstate <= STransfer3;
111             else
112                serial_write_en <= '0';
113             end if;
114
115       -- Transfer3
116          when STransfer3 =>
117             nstate <= STransfer3;
118
```

```
119             if (serial_busy= '0') then
120                serial_write_en <= '1';
121                serial_data_in <= buffer_nb(15 downto 8);
122                nstate <= STransfer4;
123             else
124                serial_write_en <= '0';
125             end if;
126
127       -- Transfer4
128          when STransfer4 =>
129             nstate <= STransfer4;
130
131             if (serial_busy= '0') then
132                serial_write_en <= '1';
133                serial_data_in <= buffer_nb(7 downto 0);
134                nstate <= SComputation;
135             else
136                serial_write_en <= '0';
137             end if;
138
139          when others => null;
140
141       end case;
142    end process;
```

V první části se instancuje náš paralelní systém a řadič sériového přenosu serial_transceiver. Následuje konečný automat, který řídí celý sériový přenos. Řadič je schopen autonomně přenést data po 8 bitech, je tedy nutné 32 bitový výsledek rozdělit na 4 části. Kvůli tomu automat obsahuje 4 stavy STransfer.

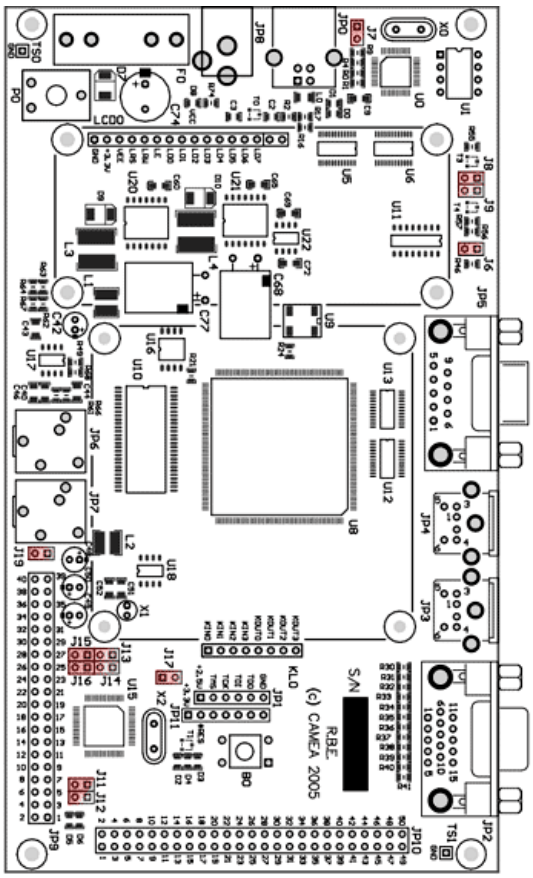
Tento přístup byl zvolen kvůli tomu, že podporuje vyšší rychlosti přenosu dat než první varianta s využitím přerušení a mikrokontroleru. Mikrokontroler nebyl schopen zpracovat všechny výsledky vypočtené paralelním systémem na FPGA čipu a docházelo k jejich ztrátám. V ukázkové aplikaci je proto po zadání počáteční podmínky y_0 vypočten pouze jeden výsledek y_1. Pokud chceme počítat dále, musíme tento výsledek y_1 zadat jako další počáteční podmínku a obdržíme následující výsledek y_2. Toto odpadá při využití druhé varianty, kdy spuštěná aplikace `FITKitRead` je schopná načíst postupně všechny výsledky produkované paralelním systémem na FPGA čipu.

Příloha B

Popis práce s přípravkem FITkit a přiloženými programy

Na webových stránkách projektu FITkit [FIT11] jsou detailní návody, jak si FITkit zprovoznit na svém počítači a jak napsat jednoduchou funkční aplikaci. Nám bude stačit pouze nahrát již připravené a přeložené aplikace do FITkitu a následně je spustit. K tomuto účelu vznikla tato část. Musí být nainstalováno minimum, které slouží ke spuštění připravené aplikace - USB ovladače pro FTDI FT2232, knihovna libkitclient a skriptovatelný terminál QDevKit.

B.1 Popis FITkitu



Obrázek B.1: FITkit

Schematicky je FITkit zobrazen na obrázku B.1.

První základní předpoklad, aby byl FITkit funkční, je propojení USB kabelem mezi konektorem JP0 a počítačem.

Pro naši práci budou důležité dvojice propojek J8, J9 a J11, J12.

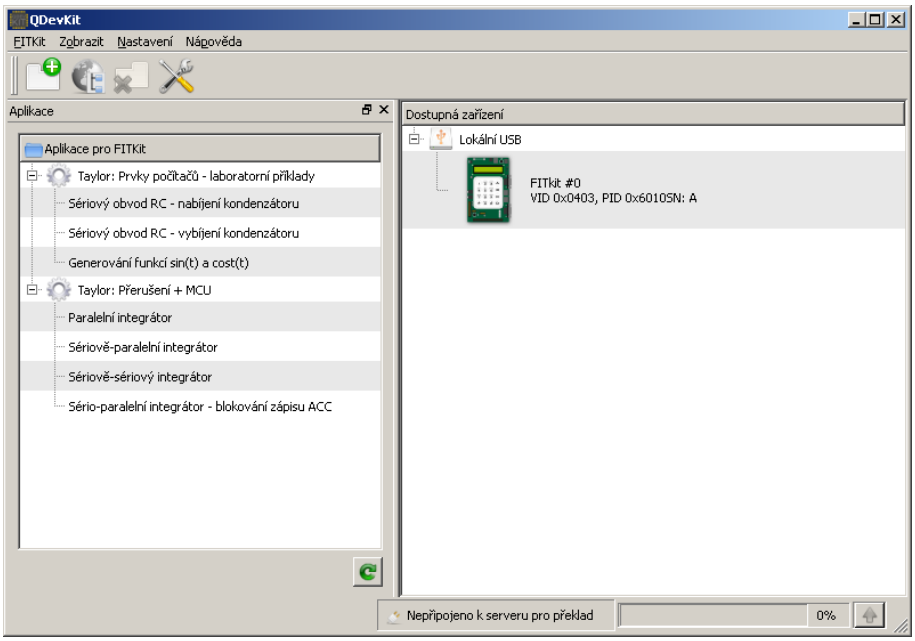
- Pokud budeme FITkit programovat, musí být tyto propojky uzavřené.
- Jestliže chceme aplikaci spustit a komunikovat s ní připraveným externím programem FITKitRead pro načítání dat, musíme je rozpojit.

Pokud používáme externí aplikaci FITKitRead a chceme FITkit restartovat, použijeme tlačítko B0.

B.2 Práce s FITkitem

B.2.1 Spuštění programu pro práci s FITkitem

Základní program, který se používá k ovládání FITkitu, je **QDevKit** - obrázek B.2.



Obrázek B.2: Program QDevKit

V levé části je seznam aplikací, které jsou k dispozici. Z adresáře `FITkit svn-apps` na přiloženém CD je nutné tyto aplikace nahrát do adresáře `apps`, který obsahuje všechny programy pro FITkit. Tento adresář se vytvořil při instalaci aplikace QDevKit.

Připravené aplikace jsou rozdělené do dvou částí:

- **Taylor: Prvky počítačů - laboratorní příklady**
  Zde jsou ukázky, na co může být paralelní systém využit. Vznikly pro podporu laboratorní výuky předmětu Prvky počítačů. Řeší nabíjení či vybíjení modelu RC článku, který je popsaný diferenciálními rovnicemi a třetí příklad slouží ke generování funkce sin(t). Výpočty probíhá s integračním krokem 0.125 po dobu 10 sekund. K přečtení výsledků slouží externí program FITkitRead. V jednotlivých podadresářích jsou i pdf dokumenty popisující dané příklady.

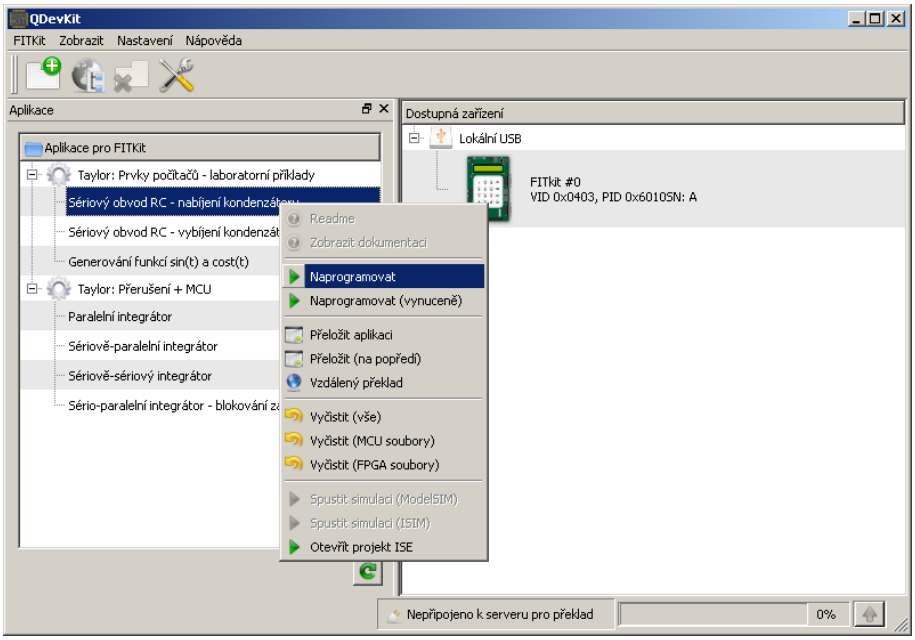
- **Taylor: Přerušení + MCU**
  Původní varianta, kdy paralelní systém v FPGA čipu byl propojen s mikrokontrolerem (MCU). Přes MCU probíhala komunikace - nahrání počáteční podmínky a následné přečtení výsledku. Je zde ukázka implementace všech tří typů integrátorů. Protože MCU je pomalejší než FPGA a nestíhá zpracovávat produkované výsledky, je výpočet omezen pouze na jeden krok - po zadání počáteční podmínky y_0 se vypočte jen jedna následující hodnota y_1 řešené diferenciální rovnice $y' = y$.

V pravé části aplikace QDevKit je indikován připojený FITkit. Ten aplikace nalezne sama. Není nutné žádné další nastavování.

B.2.2 Nahrání aplikace do FITkitu

Postup nahrávání aplikací je následující:

- Propojky J8, J9 a J11, J12 musí být uzavřeny.
- V levém seznamu aplikací si zvolíme tu, kterou chceme nahrát do FITkitu.
- Klikneme na ni pravým tlačítkem myši a zvolíme možnost Naprogramovat, jak je ukázáno na obrázku B.3.



Obrázek B.3: Programování FITkitu pomocí aplikace QDevKit

- Následně se zobrazí okno, které informuje o průběhu nahrávání aplikace na FPGA čip a do MCU. Po úspěšném nahrání okno zmizí.

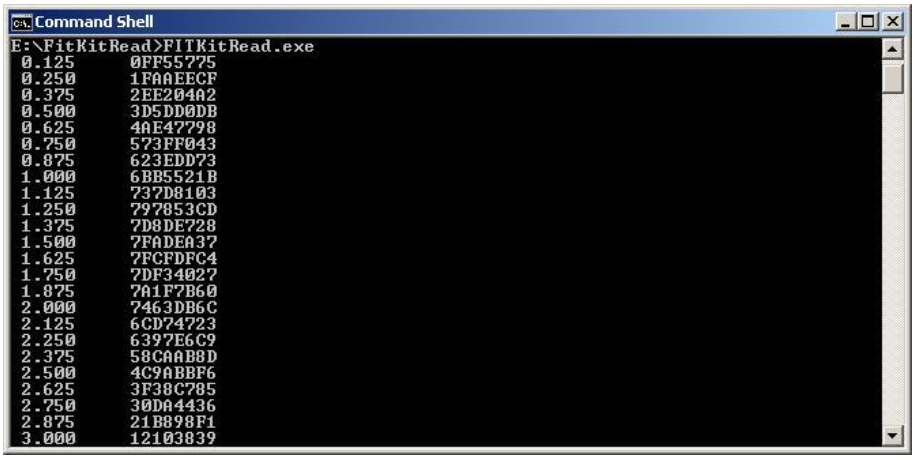
  Nyní je požadovaný obsah nahrán do přípravku FITkit a může být spuštěn.

B.2.3 Spuštění nahrané aplikace

K získání vypočtených výsledků z FITkitu použijeme programu FITkitRead. To platí pro aplikace v první části, která obsahuje laboratorní příklady.

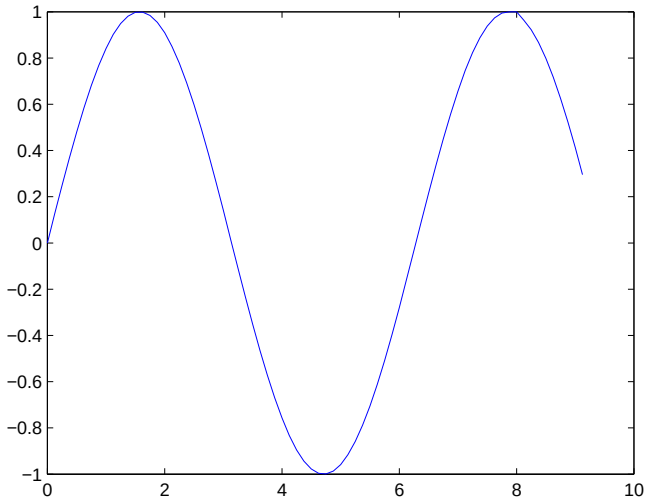
Postup připojení k aplikaci je následující:

- Propojky J8, J9 a J11, J12 musí být rozpojeny.
- V příkazové řádce spustíme program FITkitRead. Ten se připojí automaticky k FITkitu, přečte z něho vypočítané výsledky a zobrazí je v textové podobě ve spuštěném terminálu. Ukázka výstupu je na obrázku. B.4.
- Výsledky si uložíme pro další zpracování.



Obrázek B.4: Výstup aplikace FITKitRead

- Pokud program FITkitRead s FITkitem nekomunikuje, nebo jsme obdrželi výsledky, které neodpovídají očekávanému řešení, provedeme reset FITkitu - tlačítkem B0 a poté spustíme program znovu.
- Uložené výsledky můžeme např. zobrazit v grafu - obrázek B.5.

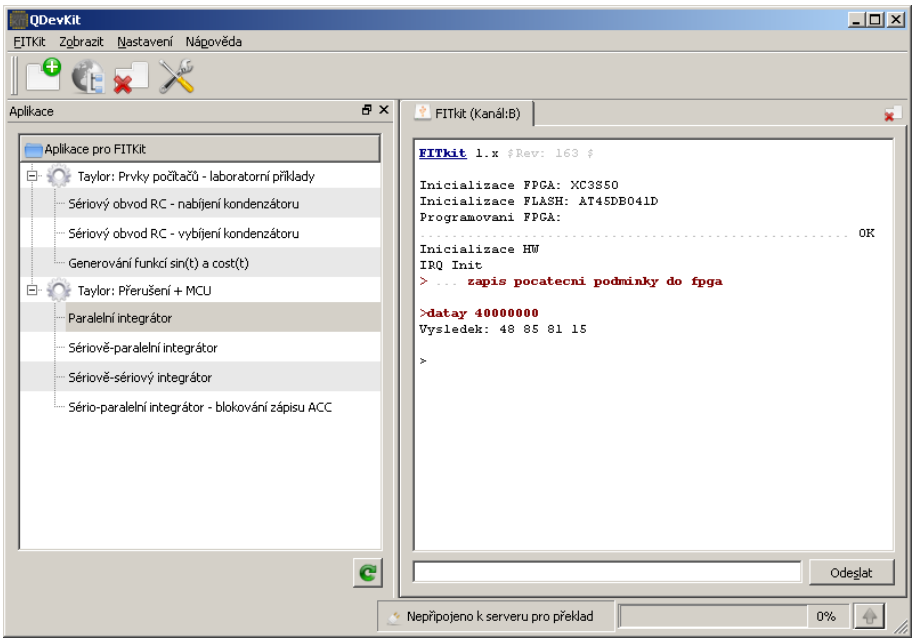


Obrázek B.5: Zobrazení výsledků získaných z FITkitu

Druhou variantou, jak ovládat aplikaci ve FITkitu, je připojit se přímo terminálem přes QDevKit na rozhraní mikrokontroleru.

- Propojky J8, J9 a J11, J12 musí být uzavřeny jako při nahrávání aplikace.
- Klikneme na symbol FITkitu v pravé části programu QDevKit a tím se na něho přípojíme.
- Nyní aplikace čeká na zadání příkazů. Např. příkaz HELP zobrazí všechny podporované příkazy aplikace nahrané na FITkitu.

- Pro naši aplikaci je důležitý příkaz DATAY, kterým zadáme počáteční podmínku. Na obrázku B.6 je vidět ukázka zadání počáteční podmínky 0.5 (do terminálu se zadává v hexadecimální formě) a následný výsledek (zobrazený opět hexadecimálně).



Obrázek B.6: Výstup aplikace FITKitRead

- Jak bylo zmíněno výše, tato varianta počítá pouze jeden výsledek, protože MCU nedokáže zpracovat větší množství produkovaných výsledků z FPGA. Pro výpočet další hodnoty je tedy nutné zadat vypočtený výsledek jako počáteční podmínku dalšího výpočtu.