Simulace řešiče soustav lineárních kongruencí

Prostudujte Gauss-Jordanovu metodu řešení soustav lineárních rovnic.
Prostudujte metody řešení soustav lineárních kongruencí (rovnic modulo m).
Navrhněte a implementujte simulátor hardwarového řešiče soustav lineárních kongruencí.
Simulátor bude umožňovat volbu velikosti architektury (zejm. počet funkčních jednotek).
Použijte vhodnou simulační platformu (např. SystemC).

České vysoké učení technické v Praze

Fakulta informačních technologií

Katedra počítačových systémů

Diplomová práce

Simulace řešiče soustav lineárních kongruencí

Bc. Jiří Jahn

Vedoucí práce: Ing. Jiří Buček

9. května 2012

Poděkování

Na tomto místě bych rád poděkoval Ing. Jiřímu Bučkovi a doc. Ing. Róbertu Lórenczovi, CSc. za pomoc a připomínky k této práci a rodině za podporu při celém průběhu studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 9. května 2012 .....

Odkaz na tuto práci

Jiří Jahn. *Simulace řešiče soustav lineárních kongruencí: Diplomová práce.* Praha: ČVUT v Praze, Fakulta informačních technologií, 2012.

Abstract

The main goal of this master's thesis is to design and implement a simulator of hardware solver of systems of linear congruences. This thesis discusses methods for solving these systems and describes a process of design and realisation of the simulator which is implemented using the SystemC simulation language and utilises the Gauss-Jordan-Rutishauser elimination method.

Keywords

system of linear congruences, residual processor, Gauss-Jordan-Rutishauser elimination method in modular arithmetic, SystemC

Abstrakt

Cílem této práce je navrhnout a implementovat simulátor hardwarového řešiče soustav lineárních kongruencí. Tato práce popisuje metody řešení těchto soustav a postup při návrhu a realizaci implementovaného simulátoru pomocí vhodného modelovacího jazyka SystemC s použitím Gauss-Jordan-Rutishauserovy eliminační metody.

Klíčová slova

soustava lineárních kongruencí, procesor pracující v aritmetice kódů zbytkových tříd, Gauss-Jordan-Rutishauserova eliminační metoda v modulární aritmetice, SystemC

Obsah

Seznam obrázků

Seznam tabulek

Úvod

K nejčastěji prováděným numerickým úlohám patří řešení soustav lineárních rovnic. Tento problém je možné pomocí Čínské věty o zbytcích rozdělit na podproblémy, kdy se řeší soustavy lineárních kongruencí. Jelikož jsou pak jednotlivé soustavy na sobě nezávislé, je možné v takovém případě výpočet paralelizovat, což je podstatná výhoda při řešení.

V rámci fakulty probíhá hardwarová realizace řešiče, kdy je k výpočtu využit vektor aritmetických jednotek a tato výzkumná práce má sloužit k tomu, aby objasnila, jak by vypadalo řešení soustav lineárních kongruencí na struktuře, která sestává z aritmetických jednotek uspořádaných v matici.

Existují jazyky pro popis harwaru, které se používají pro návrh digitálních integrovaných obvodů. Cílem této práce ale není nasazování navrhovaného řešení na skutečný hardware, ale vytvoření simulace, která bude z chování hardwaru vycházet.

Při paralelním řešení soustav lineárních kongruencí lze využít procesory pracující v aritmetice kódů zbytkových tříd. Náplní této práce je řešení soustav na jednom takovém procesoru, který k výpočtu využívá jeden modul a vnitřní struktura tohoto procesoru pracuje paralelně. Řešení všech nezávislých soustav, tedy výpočet na více procesorech, není tématem práce.

V simulační platformě SystemC bude navržena maticová struktura aritmetických jednotek pracujících v modulární aritmetice, kdy budou jednotlivé jednotky pracovat paralelně a celá výpočetní matice bude řešit soustavy lineárních kongruencí pomocí Gauss-Jordan-Rutishauserovy eliminační metody. Počet výpočetních jednotek bude závislý na velikosti zadaných soustav.

Součástí práce bude i vizualizační aplikace, která bude graficky reprezentovat všechny operace, které během řešení soustav jednotlivé funkční jednotky provádějí.

V první části této práce se objevuje teoretický úvod do lineárních kongruencí, metody, které lze k řešení soustav lineárních kongruencí použít, úvod do simulačního jazyka a analýza navrhovaného a implementovaného řešení. V následující části je popsán návrh a realizace simulační aplikace, kde je popsána jak samotná struktura, tak výpočet. V závěru této kapitoly je uveden návrh a realizace aplikace sloužící k vizualizaci běhu simulační aplikace. Třetí část práce se zabývá testováním simulace. Dále se v této části textu objevují výsledky z měření činnosti aritmetických jednotek během výpočtu.

KAPITOLA 1

Analýza

Tato práce se zabývá simulací řešiče soustav lineárních kongruencí. První kapitola slouží k seznámení se s výpočetní metodou, simulačním jazykem a výběrem možností pro navrhované řešení.

V první části této kapitoly (1.1) jsou stručně popsány lineární kongruence a zbytkové třídy v modulární aritmetice. Dále pak soustavy lineárních kongruencí a také způsob, jak se dají tyto soustavy řešit paralelně.

Ve druhé části (1.2) je uveden přehled výpočetních metod, které je možné použít při řešení soustav lineárních kongruencí včetně jejich popisu, způsobu použití a algoritmu, jak lze metody implementovat.

V následující části (1.3) je popsána Gauss-Jordanova eliminační metoda. Je zde uveden základní popis metody, ze které vznikla i popis vylepšené verze této metody, protože právě tato slouží v celé práci jako hlavní výpočetní metoda. Je rozebrána dosti podrobně, včetně příkladu použití následovaného ukázkou implementace.

Ve čtvrté části této kapitoly (1.4) se nachází úvod do simulačního jazyka SystemC. Po dohodě s vedoucím práce byl právě tento jazyk zvolen jako nástroj k simulování řešiče. Dále se objevují jednotlivé prvky SystemC vždy s popisem a ukázkami kódu pro lepší porozumění.

V poslední, páté části (1.5), je uvedena analýza řešeného problému, tedy simulátoru. Přesněji řečeno, jedná se o analýzu procesoru pracujícího v aritmetice kódů zbytkových tříd, což je jednotka, která provádí celý výpočet. Je zde rozebrána struktura tohoto procesoru - volba nejlepších možností, se kterými se dále pracuje v návrhu a realizaci.

1.1 Lineární kongruence, zbytkové třídy modulo m

Informace v této kapitole a podkapitolách vycházejí z [20][12][2][15][16]. Dvě celá čísla a a b jsou kongruentní modulo m (libovolné přirozené číslo větší než

1), pokud mají a i b po dělení číslem m stejný zbytek. Zapisuje se jako:

$$a \equiv b \,(\mathrm{mod}\, m).$$

Dále v textu (v příkladu 1.3.3.1.1) je využit zápis $b = |a|_m$. Tento zápis neznamená totéž, co $a \equiv b \,(\mathrm{mod}\, m)$, ale to, že b je rovno zbytku po dělení čísla a číslem m. Lineární kongruencí o jedné neznámé se pak rozumí:

$$ax \equiv b \,(\mathrm{mod}\, m),$$

kde a (není kongruentní s nulou v modulu m) a b jsou celá čísla, m je přirozené číslo větší než 1 a x je hledané řešení lineární kongruence.

Podmnožina v $\mathbb{Z}$ (množina všech celých čísel) skládající se právě ze všech čísel, která mají při dělení číslem m stejný zbytek, se nazývá *zbytková třída* modulo m. Třída obsahující $i \in \mathbb{Z}$ se označí $[i]_m$, protože do této třídy spadají všechna čísla, která mají po dělení číslem m stejný zbytek jako číslo i. Těchto množin je m a všechny jsou disjunktní - nemají žádné společné prvky. Příkladem jsou všechny zbytkové třídy modulo 3, které vypadají takto:

$$\begin{aligned}
[0]_3 &= \{\ldots, -9, -6, -3, 0, 3, 6, 9, \ldots\},\\
[1]_3 &= \{\ldots, -8, -5, -2, 1, 4, 7, 10, \ldots\},\\
[2]_3 &= \{\ldots, -7, -4, -1, 2, 5, 8, 11, \ldots\}.
\end{aligned}$$

Množina zbytkových tříd podle modulu 3 je tedy $\mathbb{Z}_3 = \{0, 1, 2\}$. Pojmy známé z lineární algebry (jako např. operace s maticemi) se dají analogicky zavést i pro $\mathbb{Z}_p$. Místo reálných čísel a operací se jen používají prvky a operace ze $\mathbb{Z}_p$. Je-li p prvočíslo, zůstává vše v platnosti, rozdíl je jen v tom, že množina nad $\mathbb{Z}_p$ je konečná.

1.1.1 Soustavy lineárních kongruencí

Mějme soustavu lineárních kongruencí

$$\mathbf{A}x \equiv y \,(\mathrm{mod}\, M),$$

kde $\mathbf{A} \in \mathbb{Z}^{n\times n}$ je maticí soustavy n lineárních kongruencí o n neznámých modulo M, $y \in \mathbb{Z}^n$ je vektor pravých stran soustavy modulo M a $x \in \mathbb{Z}^n$ je hledané řešení soustavy lineárních kongruencí modulo M.

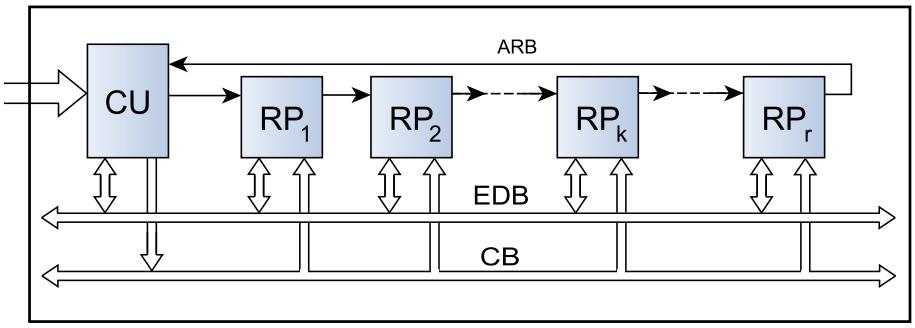
Modul M může být velmi velké číslo, což je pro výpočet nevhodné. Vícemodulární aritmetika kódů zbytkových tříd je ekvivalentní jednomodulární aritmetice kódů zbytkových tříd s modulem rovným součinu všech modulů z vícemodulární aritmetiky. Proto je možné použít moduly $m_1, m_2, \ldots, m_r$. Při rozdělení na soustavy nezávislých lineárních kongruencí musí jednotlivé moduly splňovat následující pravidla (převzato z [10]):

- $m_1 \cdot m_2 \cdot \cdots \cdot m_r \geq \mathrm{M}$,
- každé m_k, $k \in \{1, 2, \ldots, r\}$, je prvočíslo,
- $m_1 < m_2 < \cdots < m_r < (2^e - 1)$, kde e značí počet bitů datové sběrnice mezi jednotlivými procesory pracujícími v aritmetice kódů zbytkových tříd (viz 1.1.1.1).

Soustava může být řešena v aritmetice kódů zbytkových tříd. To je výhodné kvůli tomu, že aritmetika kódů zbytkových tříd poskytuje velmi dobré předpoklady pro to, aby výpočet probíhal paralelně. Předpoklad je takový, že se řeší tolik soustav lineárních kongruencí, kolik je použito modulů. Jelikož jsou jednotlivé soustavy lineárních kongruencí na sobě nezávislé, je na místě využití paralelního výpočtu (viz podkapitola 1.1.1.1).

1.1.1.1 Způsob paralelního řešení soustav lineárních kongruencí

Informace v této části textu vycházejí z [10][11]. Jeden ze způsobů, jak paralelně řešit soustavy lineárních kongruencí, je za použití procesorů pracujících v aritmetice kódů zbytkových tříd. Tyto procesory jsou propojeny datovou sběrnicí a společně tvoří modulární systém, jak je znázorněno na obrázku 1.1. Každý z těchto procesorů pracuje s jinou soustavou lineárních kongruencí a také s jiným modulem. Původní soustava byla rozdělena do r na sobě nezávislých soustav lineárních kongruencí. K jejich vyřešení se budou využívat moduly $m_1, m_2, \ldots, m_r$. Modulární systém se bude skládat z r procesorů pracujících v aritmetice kódů zbytkových tříd. Každý procesor bude řešit jednu soustavu lineárních kongruencí v jednom modulu (výpočet je popsán v 1.3.3.1). Jelikož jsou všechny soustavy na sobě nezávislé, je možné výpočet paralelizovat.



Obrázek 1.1: Modulární systém (převzato z [11])

V obrázku 1.1 je naznačeno, jak vypadá celý modulární systém. Jednotlivá označení (převzato z [11]) v obrázku znamenají:

- RP_1 až RP_r jsou procesory pracující v aritmetice kódů zbytkových tříd používající k výpočtu moduly m_1 až m_r,
- CU je kontrolní jednotka, která řídí celý výpočet,
- EDB je e-bitová datová sběrnice sloužící k toku dat mezi jednotlivými procesory navzájem a mezi kontrolní jednotkou a procesory,
- CB je kontrolní sběrnice a slouží k tomu, aby přenášela kontrolní signály z kontrolní jednotky do jednotlivých procesorů,
- ARB je uzavřený okruh sloužící k adresování jednotlivých procesorů během fáze, kdy se převádí výsledky ze zbytkové reprezentace zpět do racionálních čísel.

1.2 Metody řešení soustav lineárních kongruencí

Pro řešení soustav lineárních kongruencí je možné využívat takové metody řešení soustav lineárních rovnic, které umožňují místo běžných operací použití operací v modulární aritmetice. Existují metody přímé a metody iterační. V metodách iteračních se vždy mezi sebou porovnávají dvě čísla. A to je právě důvod, proč se standardně pro řešení soustav lineárních kongruencí metody iterační nepoužívají. Za účelem zjištění, které ze dvou čísel je větší, se čísla porovnávají, což ale není možné, pokud jsou čísla v různých modulech. Řešením by bylo převést čísla z modulární aritmetiky do racionálních čísel, tam čísla porovnat a převést zpět do modulární aritmetiky. Takto by to muselo být řešeno při každé iteraci. Vzhledem k náročnosti, a v případě použití tohoto způsobu i k vysoké ceně, se tedy iterační metody k řešení soustav lineárních kongruencí nepoužívají. Používají se metody přímé.

1.2.1 Gauss-Jordanova metoda

Metodou, kterou lze použít pro řešení soustavy lineárních kongruencí, je Gauss-Jordanova metoda vycházející z metody Gaussovy. Po konzultaci a dohodě s vedoucím práce byla právě tato metoda, resp. její rozšířená verze - Gauss-Jordan-Rutishauserova metoda - vybrána k tomu, aby sloužila jako hlavní výpočetní metoda k řešení soustav v této práci. Z tohoto důvodu je Gauss-Jordanova eliminační metoda podrobně rozebrána v následující kapitole této práce.

1.2.2 Metoda LU-rozkladu

Obsah této kapitoly a podkapitoly je založen na [7][5][13]. Další metodou, kterou je možné využít při řešení soustav lineárních kongruencí, je takzvaná metoda LU-rozkladu. V Gaussově eliminační metodě (1.3.1) se upravuje původní

čtvercová matice $\mathbf{A}$ s jednotlivými prvky $a_{i,j} \bmod m$, kde $i, j \in \{1, 2, \ldots, n\}$ na tzv. trojúhelníkový tvar, ze kterého se pak dá taková matice jednoduše vyřešit. V metodě LU-rozkladu při řešení soustav lineárních kongruencí se hledá matice $\mathbf{L}$ a matice $\mathbf{U}$ tak, aby platilo

$$\mathbf{A} \equiv \mathbf{L} \cdot \mathbf{U} \pmod{m},$$

a matice $\mathbf{L}$ byla dolní trojúhelníková matice s jedničkami na diagonále

$$\mathbf{L} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n,1} & l_{n,2} & \cdots & 1 \end{pmatrix}$$

a matice $\mathbf{U}$ horní trojúhelníková matice

$$\mathbf{U} = \begin{pmatrix} u_{1,1} & u_{1,2} & \cdots & u_{1,n} \\ 0 & u_{2,2} & \cdots & u_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{n,n} \end{pmatrix}.$$

Řeší se soustava $\mathbf{A} \cdot \mathbf{x} \equiv \mathbf{b} \pmod{m}$, kde $\mathbf{x}$ je hledané řešení soustavy a $\mathbf{b}$ vektor pravých stran. Po rozložení matice $\mathbf{A}$ na součin matic $\mathbf{L}$ a $\mathbf{U}$ tedy

$$\mathbf{L} \cdot \mathbf{U} \cdot \mathbf{x} \equiv \mathbf{b} \pmod{m}.$$

Pokud se $\mathbf{U} \cdot \mathbf{x}$ označí jako $\mathbf{y}$ a dosadí, vznikne trojúhelníková soustava

$$\mathbf{L} \cdot \mathbf{y} \equiv \mathbf{b} \pmod{m},$$

ze které se vypočte vektor $\mathbf{y} \bmod m$. Tento vektor se pak použije jako pravá strana soustavy

$$\mathbf{U} \cdot \mathbf{x} \equiv \mathbf{y} \pmod{m},$$

ze které se konečně určí vektor neznámých $\mathbf{x} \bmod m$. Při výpočtu vektoru $\mathbf{y}$ se využívá dopředné substituce a při výpočtu vektoru $\mathbf{x}$ se používá zpětná substituce.

1.2.2.1 Algoritmus rozkladu

Do matic $\mathbf{L}$ a $\mathbf{U}$ se ukládají příslušné hodnoty postupně po sloupcích tak, jak je uvedeno v popisu na následující straně. V bodě *I.* se nastaví prvky na diagonále v matici $\mathbf{L}$ na jedničku. V bodě *II.* se vždy nejdříve nastaví na správnou hodnotu všechny prvky matice $\mathbf{U}$ (cyklus *1.*) tak, aby byla matice

horní trojúhelníková. Potom se nastaví všechny prvky matice **L** (cyklus *2.*) tak, aby byla dolní trojúhelníková.

Nechť $\mathbf{A} = a_{i,j} \bmod m\ , \quad i,j \in \{1, 2, \ldots, n\}$,
$\mathbf{L} = l_{i,j}, \quad l_{i,j} = 0, \quad i,j \in \{1, 2, \ldots, n\}$,
$\mathbf{U} = u_{i,j}, \quad u_{i,j} = 0, \quad i,j \in \{1, 2, \ldots, n\}$.

I. **For** $s = 1, 2, \ldots, n$ **do**:
$l_{s,s} = 1$
end
II. **for** $j = 1, 2, \ldots, n$ **do**:
1. **for** $i = 1, 2, \ldots, j$ **do**:
$u_{i,j} = [a_{i,j} - \sum_{k=1}^{i-1} l_{i,k} \cdot u_{k,j}] \bmod m$
end
2. **for** $i = j+1, j+2, \ldots, n$ **do**:
$l_{i,j} = [(a_{i,j} - \sum_{k=1}^{j-1} l_{i,k} \cdot u_{k,j}) \cdot (u_{j,j})^{-1}] \bmod m$
end
end

Funkčnost tohoto algoritmu byla v rámci práce ověřena implementací. Předpokladem pro použití této metody je to, že všechny hlavní prvky, tedy prvky $u_{j,j}$, jsou během výpočtu nenulové. Nutnost této podmínky vyplývá i z bodu *2.* v popisu výše, kde se násobí inverzní hodnotou právě tohoto prvku a inverze k nule neexistuje.

Pokud by se na diagonále objevila nula, dalo by se to vyřešit prohozením řádků matice **A**, tak, aby se tento problém odstranil. Pak by se ale musely prohodit také řádky v matici **L**. To by ale matice **L** už nebyla trojúhelníková. Problém se řeší tzv. permutační maticí **P**, která bude na začátku jednotková a budou se v ní prohazovat řádky. Rozklad by pak tedy byl

$$\mathbf{A} \equiv \mathbf{P} \cdot \mathbf{L} \cdot \mathbf{U} \ (\mathrm{mod}\, m).$$

Metoda LU-rozkladu a Gaussova eliminace jsou si hodně podobné. Aritmetické operace jsou v obou metodách ekvivalentní, provádějí se pouze v jiném pořadí. Eliminační část Gaussova algoritmu odpovídá nalezení LU-rozkladu a vypočtení vektoru **y**. Ten se zde počítá po nalezení rozkladu matice **A**, v Gaussově eliminaci se počítá průběžně během eliminace. Vektor **y** se v metodě LU-rozkladu dále použije k vypočtení výsledného vektoru **x**. Velká výhoda této metody oproti Gaussově je to, že v případě stejné soustavy, pouze jiných pravých stran, je možné využít rozklad matice nalezený při řešení první soustavy, což sníží počet prováděných operací. Jednou nalezený rozklad lze využít pro řešení všech soustav se shodnou levou stranou.

1.3 Gauss-Jordanova metoda

Informace uvedené v této části textu a všech podkapitolách jsou z [6][19][16][10]. V této kapitole se nejdříve objevuje jednoduchý popis klasické Gaussovy eliminační metody sloužící k řešení soustavy lineárních algebraických rovnic pomocí úpravy matice. Dále je zde popsána Gauss-Jordanova eliminační metoda, která vychází z Gaussovy metody eliminace. Následuje popis Gauss-Jordan-Rutishauserovy metody, která je rozšířením a vylepšením Gauss-Jordanovy eliminace. Nakonec je detailně rozebrána Gauss-Jordan-Rutishauserova eliminační metoda v modulární aritmetice, která je použita v rámci této práce k řešení soustav lineárních kongruencí.

1.3.1 Gaussova metoda

Gaussova eliminační metoda je metodou řešení soustavy lineárních algebraických rovnic. Tato metoda sestává ze dvou částí:

1. Gaussova eliminace,
2. zpětná substituce.

Gaussova eliminace představuje řešení problému vyjádřeného pomocí matice převedením dané matice na horní trojúhelníkovou (matice, která má všechny prvky pod hlavní diagonálou nulové). Soustavu lineárních algebraických rovnic lze vyjádřit rozšířenou maticí soustavy (v matici je zapsán i vektor pravých stran rovnic). Řádkovými úpravami lze převádět tuto matici do tvaru, kdy se pod hlavní diagonálou nacházejí pouze nuly. Upravená matice pak odpovídá soustavě rovnic, která je ekvivalentní s původní soustavou. Aby se nezměnila hodnost (určuje počet lineárně nezávislých řádků matice) soustavy, jsou povoleny tyto operace (elementární řádkové úpravy):

- násobení či dělení jednotlivých řádků nenulovým číslem,
- prohazování libovolných řádků soustavy,
- přičítání násobků jednotlivých řádků k jinému řádku.

Po získání horní trojúhelníkové matice zbývá jednotlivá řešení dopočítat zpětným dosazováním (substitucí) od spodního řádku matice směrem nahoru, vždy za pomoci již spočtených neznámých.

1.3.2 Gauss-Jordanova metoda

Gauss-Jordanova eliminace je upravená Gaussova eliminace tak, že převádí rozšířenou matici soustavy do tvaru, kde na místě matice soustavy bude jednotková matice (na hlavní diagonále jedničky, pod ní a nad ní všude nuly)

a na místě pravých stran se pak vyskytne řešení soustavy. Převod se provádí stejně jako u Gaussovy eliminace pomocí elementárních řádkových úprav, navíc lze ale využít i sloupcových úprav. V tomto případě je ale potom potřeba brát na zřetel, že mohlo dojít ke změně příslušnosti proměnných k jednotlivým sloupcům. Po provedení Gauss-Jordanovy eliminační metody vypadá výsledná matice takto:

$$\left(\begin{array}{cccc|c} 1 & 0 & \cdots & 0 & x_1 \\ 0 & 1 & \cdots & 0 & x_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & x_n \end{array}\right),$$

kde x_i pro $i \in \{1, 2, \ldots, n\}$ jsou jednotlivá hledaná řešení soustavy lineárních rovnic. Při použití této eliminační metody již není nutné provádět žádnou zpětnou substituci.

1.3.3 Gauss-Jordan-Rutishauserova metoda

Gauss-Jordan-Rutishauserova eliminační metoda je shodná s Gauss-Jordanovou metodou, v každém eliminačním kroku se provádějí shodné operace, jen je vylepšena tak, že se navíc hodnoty posouvají vždy o sloupec doleva v paměťových buňkách matice (převzato z [10]). Při každém jednom eliminačním kroku se vypočtené hodnoty posunou o jeden sloupec vlevo. Tím vpravo vždy při každém dalším kroku eliminace přibude jeden sloupec k těm, v nichž jsou hodnoty nepodstatné. Po skončení všech eliminačních kroků je tedy výsledný vektor uložen v prvním sloupci matice.

1.3.3.1 Gauss-Jordan-Rutishauserova eliminace v modulární aritmetice s pivotizací a architekturou procesoru pracujícího v aritmetice kódů zbytkových tříd

V této části textu následuje popis výpočetního algoritmu (převzat z [10]) na jednom z procesorů RP_1, RP_2, …, RP_r (viz 1.1.1.1) pracujících v aritmetice kódů zbytkových tříd. Tento procesor využívá k výpočtu jeden z modulů m_1, m_2, …, m_r a řeší odpovídající soustavu lineárních kongruencí v tomto modulu.

Mějme matici $\mathbf{M}$, která bude mít rozměry $n \times (n + 1)$. Každá hodnota v této matici se dá označit $a_{i,j}$, kde $i, j \in \{1, 2, \ldots, n\}$, a obsahuje prvek matice $\mathbf{A}$ (viz 1.1.1) modulo m a hodnoty $a_{i,n+1}$ obsahují prvky vektoru y (tedy vektor pravých stran) modulo m:

$$\mathbf{M} = \left(\begin{array}{ccc|c} a_{1,1} & \cdots & a_{1,n} & y_1 \\ \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} & y_n \end{array}\right)_m.$$

Pivotizace, která je využívána v Gauss-Jordanově eliminaci v modulární aritmetice je analogická k výběru pivota v klasické Gauss-Jordanově eliminaci, jen s tím rozdílem, že se zde nehledá prvek s nejvyšší hodnotou, ale stačí najít jakýkoliv nenulový prvek v prvním sloupci matice **M** v dané aritmetice kódů zbytkových tříd. Tento proces se nazývá nenulová modulární pivotizace. Gauss-Jordan-Rutishauserova eliminace v aritmetice kódů zbytkových tříd lze popsat následujícími kroky (převzato z [10]):

Nechť $a_{i,j}^{(0)} = [a_{i,j}] \bmod m; \quad i \in \{1, 2, \ldots, n\}, j \in \{1, 2, \ldots, n+1\}.$

For $k = 1, 2, \ldots, n$ (k značí eliminační krok) **do**:

1. najdi nějaké $i \neq c_f$ $(f < k)$ tak, aby $a_{i,1} \neq 0$, $c_k = i$,
2. $a_{i,j}^{(k)} = [a_{i,j+1}^{(k-1)} \cdot (a_{i,1}^{(k-1)})^{-1}] \bmod m, \quad j \in \{1, 2, \ldots, n+1-k\}$,
3. **for** $l = 1, 2, \ldots, n \wedge l \neq i$ **do**:
   $a_{i,j}^{(k)} = [a_{l,j+1}^{(k-1)} - a_{l,1}^{(k-1)} \cdot a_{i,j}^{(k)}] \bmod m, \quad j \in \{1, 2, \ldots, n+1-k\}.$
   end

end

$z_i = a_{c_i,1}, \quad i \in \{1, 2, \ldots, n\},$

$a_{i,1} = z_i, \quad i \in \{1, 2, \ldots, n\}.$

V tomto popisu eliminace značí i řádek matice a j sloupec. U hodnoty $a_{i,j}^{(k)}$ znamenají spodní indexy pozici v matici (řádek, sloupec) a horní index znamená eliminační krok ($^{(0)}$ značí stav před začátkem eliminace, $^{(k)}$ značí aktuální eliminační krok a $^{(k-1)}$ značí předchozí eliminační krok). Celá eliminace probíhá v cyklu, který se opakuje tolikrát, kolik je řádků v matici.

Nejdříve (1.) se vybere takový řádek, kde hodnota v prvním sloupci v tomto řádku není nulová a zároveň ještě nebyl tento řádek v žádném eliminačním kroku zvolen. Index vybraného řádku splňujícího podmínky je i a tato hodnota se uloží v eliminačním kroku k do odpovídajícího registru c_k. Prvek v této řádce a prvním sloupci, tedy $a_{i,1}$, je pivot pro tento eliminační krok.

V dalším bodě (2.) je provedena multiplikativní inverze pivota (proto byl jako pivot v 1. bodě zvolen nenulový prvek - k nulovému prvku neexistuje inverze) a pro všechny zbylé prvky v řádce i je proveden součin s touto inverzí pivota v příslušném modulu. Jelikož vynásobením prvku s jeho inverzí vzniká jednička, je možné počítat s tím, že vznikla a všechny prvky v řádce i posunout doleva o jedno místo v matici. Tím vlevo vypadne jednička (to není problém) a vpravo zbyde nějaká nepodstatná hodnota.

V posledním bodě (3.) se pro všechny zbylé řádky (kromě řádku, kde se nacházel pivot - tedy řádek i) provede negace hodnoty prvku v prvním sloupci, pak součin odpovídající negace vždy s prvky v příslušném sloupci z řádky

s pivotem (řádka i) a k odpovídajícímu součinu nakonec přičtení hodnoty daného prvku. To vše v příslušném modulu. Negace není problém, protože je definována i pro nulový prvek. Násobení se provádí s tím prvkem, který byl v předchozím bodě (2.) přesunut zprava, tedy multiplikativní inverze pivota vynásobená hodnotou příslušné buňky. Jelikož by byla bez posuvu v prvním sloupci a řádce i hodnota rovna 1, po negaci, násobení a součtu, by ve všech zbylých řádcích v prvním sloupci vznikla 0, je opět možné s tímto počítat a posunout i tyto řádky o jednu pozici v matici doleva (vlevo vypadnou nuly a vpravo budou opět nepodstatné hodnoty).

Po průběhu jednoho eliminačního kroku budou posunuty všechny hodnoty v celé matici o sloupec vlevo. Pak se proces opakuje znovu od začátku, opět se hledá nenulový pivot, atd., tolikrát, kolik obsahuje matice řádků. Po skončení celé Gauss-Jordan-Rutishauserovy eliminace je výsledný vektor uložen v prvním sloupci matice.

Pokud by byl v každém eliminačním kroku vybrán pivot v takové řádce, jaký eliminační krok zrovna probíhá, na pomyslné diagonále (bez posouvání) by se vytvořily samé jedničky, nad ní a pod ní samé nuly a výsledný vektor by mohl být reprezentován v tom pořadí, v jakém je po skončení eliminace. V průběhu eliminace však mohlo dojít k tomu, že ne vždy byl pivot vybrán na takové řádce, v jakém eliminačním kroku se hledal. Pokud nemůže být vybrán jako pivot v určitém eliminačním kroku k takový prvek, který leží na řádce k (jeho hodnota je rovna 0), musí být tato řádka přeskočena a vybrána jiná. V takovém případě jsou po skončení eliminace hodnoty vektoru výsledků v přeházeném pořadí. K tomu, aby se výsledky uspořádaly do správného pořadí, slouží poslední dvě řádky v popisu eliminace výše.

Na první z nich je do pomocného vektoru z za pomoci registrů c_i nahrána hodnota výsledku z daného eliminačního kroku tak, aby byly výsledky seřazeny ve správném pořadí. Ve druhé řádce už jsou jen seřazené hodnoty přiřazeny zpět do prvků $a_{i,1}, i \in \{1, 2, \ldots, n\}$. Seřazený výsledný vektor umístěný v prvním sloupci matice je samozřejmě ve zbytkové reprezentaci, tudíž $[x]$ mod m. Takže ještě zbývá převést výsledky soustav lineárních kongruencí ze zbytkové reprezentace do racionálních čísel.

1.3.3.1.1 Příklad Gauss-Jordan-Rutishauserovy eliminace v modulární aritmetice

Pro jasnější demonstraci toho, jak přesně funguje Gauss-Jordan-Rutishauserova eliminace v modulární aritmetice, je zde uveden příklad, který kopíruje popis eliminace uvedený výše. Před začátkem samotné eliminace je sestavena matice, do které jsou zapsány i pravé strany, vše v příslušném modulu:

$$\left(\begin{array}{ccc|c} \boxed{6} & 7 & 12 & 8 \\ 3 & 2 & 9 & 1 \\ 2 & 4 & 7 & 11 \end{array}\right)_{13}$$

Jelikož má matice tři řádky, budou se provádět tři eliminační kroky. Řešení začne výběrem prvního pivota. To může být např. hned 6. Existuje k němu inverze v modulu 13, protože je nenulový:

$$|6^{-1}|_{13} = 11$$

Vynásobením hodnoty s její inverzí je výsledkem 1:

$$|6 \cdot 6^{-1}|_{13} = |6 \cdot 11|_{13} = |66|_{13} = 1$$

Pro zbylé prvky v řádce stejný postup (tedy vynásobení inverzí pivota):

$$|7 \cdot 6^{-1}|_{13} = |7 \cdot 11|_{13} = |77|_{13} = 12$$

$$|12 \cdot 6^{-1}|_{13} = |12 \cdot 11|_{13} = |132|_{13} = 2$$

$$|8 \cdot 6^{-1}|_{13} = |8 \cdot 11|_{13} = |88|_{13} = 10$$

Pro zbylé řádky a jejich hodnoty v prvním sloupci bude vždy po provedení negace, součinu a součtu výsledkem 0:

$$|3 - 3 \cdot 1|_{13} = |3 + 10 \cdot 1|_{13} = |13|_{13} = 0$$

$$|2 - 2 \cdot 1|_{13} = |2 + 11 \cdot 1|_{13} = |13|_{13} = 0$$

Zatím je matice ve tvaru:

$$\left(\begin{array}{ccc|c} 1 & 12 & 2 & 10 \\ 0 & x_{22} & x_{23} & x_{24} \\ 0 & x_{32} & x_{33} & x_{34} \end{array}\right)_{13},$$

kde x_{ij} značí hodnotu, která byla v původní matici a zatím nebyla změněna. Protože v prvním sloupci vnikla jednička a nuly, přichází na řadu vylepšení Gauss-Jordan-Rutishauserovy metody oproti Gauss-Jordanově - posun vlevo. Po posunutí o jednu pozici vlevo bude mít matice následující tvar:

$$\left(\begin{array}{ccc|c} 12 & 2 & 10 & . \\ x_{22} & x_{23} & x_{24} & . \\ x_{32} & x_{33} & x_{34} & . \end{array}\right)_{13},$$

kde . značí prvek, který je nepodstatný pro další výpočet a nezáleží na tom, jaká je jeho hodnota. V prvním eliminačním kroku zbývá tedy ještě dopočíst hodnoty x:

$$|2 - 3 \cdot 12|_{13} = |2 + 10 \cdot 12|_{13} = |122|_{13} = 5$$

$$|9 - 3 \cdot 2|_{13} = |9 + 10 \cdot 2|_{13} = |29|_{13} = 3$$

$$|1 - 3 \cdot 10|_{13} = |1 + 10 \cdot 10|_{13} = |101|_{13} = 10$$

$$|4 - 2 \cdot 12|_{13} = |4 + 11 \cdot 12|_{13} = |136|_{13} = 6$$

$$|7 - 2 \cdot 2|_{13} = |7 + 11 \cdot 2|_{13} = |29|_{13} = 3$$

$$|11 - 2 \cdot 10|_{13} = |11 + 11 \cdot 10|_{13} = |121|_{13} = 4$$

Po provedení prvního eliminačního kroku vypadá matice následovně:

$$\left(\begin{array}{ccc|c} 12 & 2 & 10 & . \\ 5 & 3 & 10 & . \\ 6 & 3 & 4 & . \end{array}\right)_{13}.$$

Teď je na řadě výběr dalšího pivota (nesmí to být 12, protože v prvním řádku už pivot vybrán byl), provedení druhého eliminačního kroku, pak výběr třetího pivota a třetí eliminační krok, po kterém bude matice ve tvaru:

$$\left(\begin{array}{ccc|c} 12 & . & . & . \\ 7 & . & . & . \\ 9 & . & . & . \end{array}\right)_{13},$$

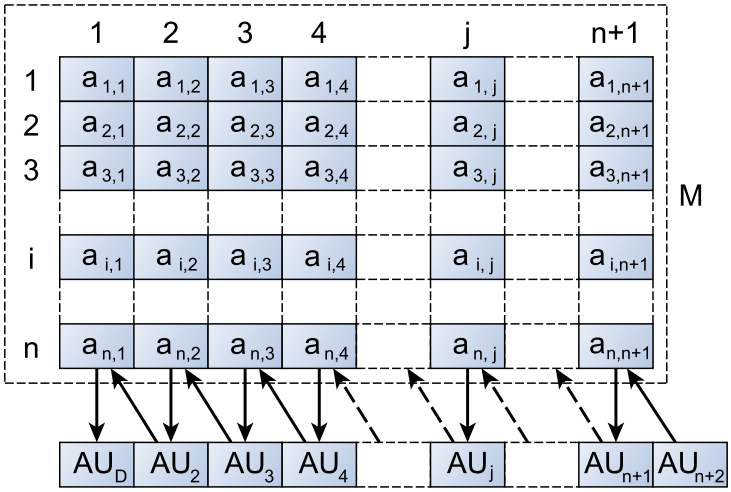
kde první sloupec značí vektor výsledků $[x]$ mod *13*.

1.3.3.1.2 Popis implementace Gauss-Jordan-Rutishauserovy eliminace

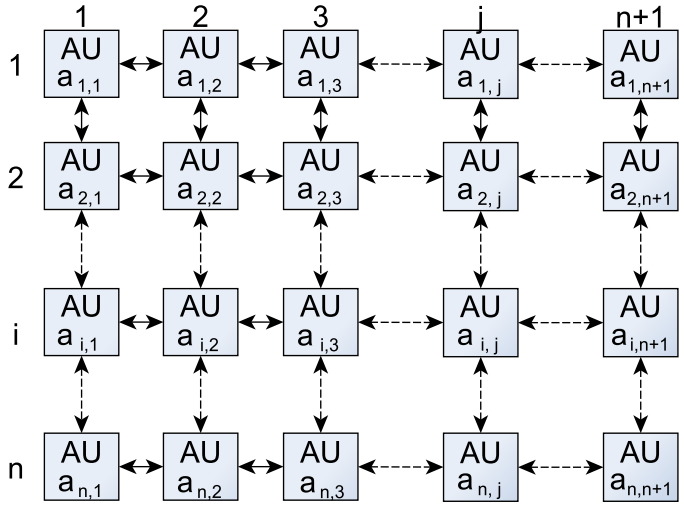
Výše popsaný algoritmus může být jednoduše implementován za pomoci procesoru pracujícího v aritmetice kódů zbytkových tříd. Nabízejí se tři varianty struktury aritmetických jednotek, které provádějí výpočet a paměťových míst na uložení hodnot v takovém procesoru:

- Hodnoty budou uloženy v paměťové matici a ta bude propojena s jednou aritmetickou jednotkou, která bude vykonávat celý výpočet. Jelikož je v tomto případě k dispozici pouze jedna jednotka, je zpracování sekvenční. Postupně se zpracovávají prvky v jednotlivé řádce a po zpracování celé řádky zase další řádka postupně po jednotlivých hodnotách.
- Hodnoty budou uloženy v paměťové matici M a ta bude sběrnicemi propojena s vektorem aritmetických jednotek AU (viz obrázek 1.2). Z tohoto zapojení plyne, že by se zpracovávaly jednotlivé řádky matice vždy postupně, nicméně všechny prvky v celé řádce by se mohly zpracovávat paralelně (převzato z [10]).
- Matice bude sestavena rovnou z aritmetických jednotek AU (viz obrázek 1.3). V tomto případě by mezi sebou jednotlivé výpočetní jednotky musely komunikovat a paměťové místo pro uložení příslušné hodnoty by

bylo v každé jednotce. V tomto případě by se mohly provádět operace paralelně nejen v jedné řádce, ale navíc i mezi řádkami.



Obrázek 1.2: Paměťová matice propojená s vektorem aritmetických jednotek (převzato z [10])



Obrázek 1.3: Aritmetické jednotky uspořádané do struktury mřížky

V pseudokódu 1.1 (převzat z [10]) je ukázáno, jak by bylo možné implementovat celý algoritmus na struktuře vektoru aritmetických jednotek propojených s paměťovou maticí (obrázek 1.2). Pro porozumění je potřeba vysvětlit některá

použitá označení (převzato z [10]):

- WR_j a WR_d jsou pracovní registry aritmetických jednotek AU_j a AU_d,
- IR_j a IR_d jsou interní (pomocné) registry aritmetických jednotek AU_j a AU_d,
- *INV(a)* znamená multiplikativní inverzi k hodnotě a v modulu m,
- *NEG(a)* značí negaci k hodnotě a v modulu m,
- c_1 až c_k jsou registry pro ukládání sekvence eliminačních kroků, podle kterých musí být po skončení eliminace správně přeházeny výsledné hodnoty ve vektoru,
- e je jednobitový registr, který slouží k ukládání znaménka determinantu, které je měněno shodně s eliminační sekvencí.

Ukázka kódu 1.1: Gauss-Jordan-Rutishauserova eliminace

```
1  IR_d := 1; e := 0; c_k := 0,   k ∈ {1, 2, ..., n}.
2
3  for i = 1, 2, ..., n do
4      for k = 1, 2, ..., n do
5          e := [e + sign(c_k)] mod 2,
6          if (a_{k,1} ≠ 0 ∧ c_k = 0) then
7              c_k := i,
8              e := [k + e − 1] mod 2,
9              pokračuj na řádce 14,
10         else if (k = n) then
11             Return("Konec − neexistuje nenulový pivot."),
12         end if
13     end for
14     WR_j := [ a_{k,j} · INV(a_{k,1}) ] mod m,   j ∈ {2, 3, ..., n + 2},
15     WR_d := [ IR_d · a_{k,1} ] mod m;
16     a_{k,j−1} := WR_j, IR_j := WR_j,   j ∈ {2, 3, ..., n + 2},
17     IR_d := WR_d;
18     for l = 1, 2, ..., n where l ≠ k do
19         WR_j := [ a_{l,j} + NEG(a_{l,1}) · IR_j ] mod m,   j ∈ {2, 3, ..., n + 2},
20         a_{l,j−1} := WR_j,   j ∈ {2, 3, ..., n + 2}.
21     end for
22 end for
23
24 WR_2 := 1
25 for l = 1, 2, ..., n do
26     WR_{l+2} := a_{c_l,1}
27 end for
28 WR_j := WR_d · WR_j · (−1)^e,   j ∈ {2, 3, ..., n + 2},
29 IR_j := WR_j,   j ∈ {2, 3, ..., n + 2},
```

Uvedený algoritmus popisuje, jak je možné Gauss-Jordan-Rutishauserovu eliminační metodu implementovat. Pro lepší porozumění pseudokódu je dále uveden popis, který je pro snazší čitelnost rozdělen podle čísel jednotlivých řádků:

1 potřebná počáteční inicializace

3 - 22 cyklus opakující eliminační krok tolikrát, kolik je řádků

4 - 13 cyklus přes všechny řádky hledající pivota

5 určení znaménka determinantu

6 - 10 podmínka, která je splněna pokud je hodnota v řádce a prvním sloupci nenulová a zároveň ještě nebyl na této řádce zvolen pivot

7 do odpovídajícího registru c_k uložit eliminační krok (slouží k přeházení výsledků po skončení všech eliminačních kroků)

8 určení znaménka determinantu

9 byl nalezen pivot splňující podmínky

10 - 12 pokud už je testována poslední řádka a předchozí podmínka nebyla splněna, je jasné, že další pivot splňující podmínky už se v matici nenachází

11 přerušení výpočtu - žádný vhodný pivot

14 provedení multiplikativní inverze pivota, součiny této inverze s hodnotami všech zbylých prvků v řádce a uložení výsledků do pracovních registrů

15 výpočet determinantu - v každém eliminačním kroku se hodnota násobí zvoleným pivotem, po všech eliminačních krocích tak vznikne determinant

16 pro všechny prvky v řádce s pivotem posuny doleva a záloha pracovních registrů do pomocných (interních) registrů

17 záloha dosavadní hodnoty budoucího determinantu do interního registru

18 - 21 cyklus přes všechny řádky kromě té, ve které je pivot

19 pro každou řádku negace hodnoty prvku v prvním sloupci, pak pro všechny zbylé hodnoty v každém řádku vynásobení s hodnotou v odpovídajícím interním registru (hodnota z té řádky, ve které je pivot) a přičtení vždy k hodnotě příslušného prvku

20 pro všechny prvky posun doleva

25 - 27 cyklus opakující se tolikrát, kolik je řádků

26 přeházení výsledků do správného pořadí pomocí registrů c_k a uložení do pracovních registrů

28 do pracovních registrů uložit výsledky přenásobené determinantem s příslušným znaménkem

29 do všech interních registrů uložit správně seřazené hodnoty výsledků z pracovních registrů

1.4 SystemC

Obsah této kapitoly a všech podkapitol je založen na [27][18][14][29][28][30][9][3][31][1][17][8][4]. SystemC není samostatným formálním jazykem, jedná se o knihovnu šablon tříd a maker pro jazyk C++. Třídy knihovny SystemC byly vyvinuty speciálně pro podporu návrhu modelu na úrovni hardwaru, proto poskytují potřebné konstruktory pro hardwarovou simulaci. SystemC poskytuje simulační jádro, se kterým mohou být číslicové obvody simulovány. Při tom používá přístup tzv. událostmi řízené simulace. Tato metoda je obdobná simulačním mechanizmům využívaných ve většině VHDL (Very-high-speed integrated circuit Hardware Description Language) prostředcích. Díky tomu je umožněno simulovat souběžnost procesů, z nichž každý je popsán běžnou syntaxí C++.

C++ jako nosný jazyk pro SystemC poskytuje základní datové typy, datové a řídící struktury (aritmetické a logické operátory, podmíněné příkazy, smyčky, atd.) a podporu objektového programování (třídy, virtuální metody, rozhraní). Samostatná knihovna SystemC pak poskytuje především podporu pro paralelní struktury, které jsou běžné v HDL (Hardware Description Language), ale které v C++ chybí. Jedná se především o koncept signálů a procesů, tak, jak je známe třeba z VHDL. Procesy v SystemC mohou komunikovat v simulovaném prostředí v reálném čase a signály mohou využívat všechny datové typy C++, SystemC a uživatelem vytvořené.

Dříve se C (nebo C++) používalo k psaní softwarové části návrhu. U hardwarové části se zase používal nějaký z existujících HDL. Pak bylo ale velice obtížné sestavit testovací sady společné pro oba jazyky, protože jsou zcela odlišné. Zavedením SystemC je mnoho z těchto problémů vyřešeno. Knihovna je podporována jak operačním systémem Windows, tak Linux a je možné ji volně stáhnout z webu [27].

Popis hardwaru se v HDL skládá z popisu struktury a popisu chování. V SystemC, stejně jako ve VHDL, jsou obě části vždy statické, nemění se během simulace. V některých ohledech SystemC záměrně napodobuje jazyky pro popis hardwaru, jako jsou VHDL a Verilog, ale je více vhodné popsat ho

jako modelovací jazyk na úrovni systému, k čemuž se také používá. Dále je využíván k výkonnostnímu modelování, prozkoumávání architektury, vývoji softwaru, funkčnímu přezkoušení a syntéze na vysoké úrovni.

1.4.1 Historie SystemC

Knihovna tříd SystemC se uvolňuje postupně a pořád se vyvíjí. Byla navržena tak, aby byla vysoce rozšiřitelná, takže všechny třídy, které definuje, mohou být rozšířeny a nové mohou být definovány. Třídy knihovny SystemC byly vyvinuty skupinou společností Open SystemC Initiative (OSCI).

- Verze 1.0 - V první fázi (v současné době ve verzi 1.0.2) byla poskytnuta podpora veškerých potřebných zázemí pro modelování hardwaru podobných těm, které lze popsat pomocí jazyka pro popis hardwaru, např. jazyk VHDL. Tato verze nabízí simulaci jádra, datové typy vhodné pro fixed-point aritmetiku, komunikační kanály, jejichž chování je obdobné s chováním vodičů (signály) a moduly pro rozepsání designu do menších částí.

- Verze 2.0 - Ve druhé fázi (v současné době u verze 2.0.1) byla knihovna tříd značně přepracována, aby odpovídala skutečnému provedení na úrovni hardwaru. Funkce, které byly ve verzi 1.0 "vestavěné", jako jsou např. signály, jsou nyní postaveny na základě struktury kanálů, rozhraní a portů. Události byly poskytnuty jako primitivní prostředky spouštění reakcí, spolu se souborem primitivních kanálů, jako je FIFO a mutex. Verze 2.0 umožňuje mnohem silnější modelování, dosažené modelováním na úrovni transakcí.

- Verze 2.1 - Tato verze přidala řadu funkcí, včetně schopnosti vytvořit procesy až poté, co simulace začala, nebo pak další zpětná volání do provozu simulace jádra.

- Verze 2.2 - V roce 2005 byl jazyk standardizován jako IEEE 1666-2005 [4] a verze 2.2 je jako referenční implementací tříd knihovny v současné době k dispozici a je aktualizována v souladu s IEEE standardem.

- Verze 3.0 - V budoucnosti by měla být tato verze rozšířena o podporu modelování operačních systémů a o podporu vývoje modelů vestavěných (embedded) zařízení.

1.4.2 Prvky jazyka SystemC

SystemC sestává z různých specifických prvků, ze kterých se při modelování sestaví výsledný simulovaný hardware. Každý slouží k určité činnosti tak, aby při použití jednotlivých prvků a správného propojení, byl model simulovaného hardwaru co nejvíce přesný a co nejméně se lišil od skutečnosti. Tyto

prvky slouží k tomu, aby bylo možné s knihovnou tříd SystemC pracovat co nejjednodušeji a zároveň aby se použití opíralo o modelovaný hardware.

1.4.2.1 Moduly

Moduly jsou základní stavební kameny hierarchie návrhu v SystemC. Ekvivalentem ve VHDL je entita. Moduly v SystemC jsou implementovány jako uživatelské třídy jazyka C++ odvozené od základní třídy `sc_module`. Model v SystemC se obvykle skládá z několika modulů, které komunikují přes porty. Modul může obsahovat porty, signály, podmoduly, proměnné a procesy (metody a vlákna).

Jsou dvě možnosti, jak vytvořit novou definici modulu. Buď pomocí makra `SC_MODULE`, nebo odvozením třídy od `class sc_module`. Oba dva způsoby (viz ukázka kódu 1.2) jsou rovnocenné.

Ukázka kódu 1.2: Dva způsoby definice modulů

```
1   //první způsob definice
2   SC_MODULE(NovyModul)
3   {
4       SC_CTOR(NovyModul)  //konstruktor
5       {
6           //kód konstruktoru
7       }
8   };
9
10  //druhý způsob definice
11  class NovyModul : public sc_module
12  {
13      NovyModul(sc_module_name jmeno) : sc_module(jmeno)  //konstruktor
14      {
15          //kód konstruktoru
16      }
17  };
18
19  SC_HAS_PROCESS(NovyModul);    //je třeba zavolat toto makro,
20                                //pokud není použito makro SC_CTOR
```

Makra `SC_MODULE` a `SC_CTOR` zpřehledňují zdrojové soubory v SystemC. Při použití makra `SC_CTOR` jsme nuceni použít jen jeden parametr při vytváření konstruktoru a tím je řetězec jména instance modulu. Pokud bychom potřebovali více než jeden argument konstruktoru, je nutné použít zápis pomocí odvození třídy od `class sc_module`.

1.4.2.2 Porty

Porty (viz příklad 1.3) poskytují komunikaci zevnitř modulu směrem ven (obvykle do dalších modulů), propojují moduly mezi sebou. Mohou přenášet v principu libovolný typ informace (bit, znak, integer, nebo i nějaký jiný datový typ, viz tabulka 1.3). Je tedy vhodné implementovat je pomocí šablon v jazyce C++ (viz tabulka 1.1).

Ukázka kódu 1.3: Porty

```
1  SC_MODULE(NovyModul)
2  {
3     sc_in<bool> vstup;  //vstupní port datového typu bool
4     sc_out<sc_bv<4>> vystup;  //výstupní port – vektor 4 jednobitových prvků
5
6     SC_CTOR(NovyModul)  //konstruktor
7     {
8        //kód konstruktoru
9     }
10 };
```

Šablona C++	Význam
`sc_in<T>`	vstupní port datového typu `T`
`sc_out<T>`	výstupní port datového typu `T`
`sc_inout<T>`	vstupně/výstupní port datového typu `T`
`sc_in_clk`	shodné s `sc_in<bool>`
`sc_out_rv<W>`	shodné s `sc_out<sc_lv<W>>`, tedy výstupní port datového typu vektor `W` prvků čtyřhodnotové logiky

Tabulka 1.1: Šablony některých portů a jejich význam

Porty modulů se propojují zpravidla v konstruktoru nadřízeného modulu. Je možné použít buď tzv. poziční vázání (pokud existuje modul M2 s porty s datovým typem po řadě stejným jako porty v jiném modulu M1, je možné provést instanciaci modulu M2 jako sobmodulu v M1), nebo použít vázání po jménu (viz příklad 1.4). Druhá možnost se zdá být lepší, protože nezávisí na pořadí definic portů v podřízeném modulu.

Ukázka kódu 1.4: Vázání portů po jménu

```
1  SC_MODULE(M2)  //podřízený modul
2  {
3      sc_out<sc_bv<4>> a;  //výstupní port – vektor 4 jednobitových prvků
4      sc_in<bool> b;  //vstupní port datového typu bool
5
6  };
7
8  SC_MODULE(M1)
9  {
10     sc_in<bool> vstup;  //vstupní port datového typu bool
11     sc_out<sc_bv<4>> vystup;  //výstupní port – vektor 4 jednobitových prvků
12
13     M2 m;     //instance submodulu
14
15     SC_CTOR(M1) : m("modul2")  //predání jména
16     {
17         m.a(vystup);   //vázání po jménu
18         m.b(vstup);    //vázání po jménu
19     }
20 };
```

1.4.2.3 Signály

Signál v SystemC je ekvivalent k vodiči. Signály (viz ukázka kódu 1.5) přenášejí informaci, přičemž na rozdíl od obyčejných proměnných jazyka C++ je zajištěna sémantika paralelního zpracování. Stejně tak, jako porty, i signály mohou přenášet libovolný typ informace a je tedy dobré použít k jejich implementaci šablony C++ (viz tabulka 1.2).

Šablona C++	Význam
`sc_signal<T>`	signál datového typu `T`
`sc_fifo<T>`	signál obecné FIFO fronty datového typu `T`
`sc_mutex`	mutex (zámek)
`sc_semaphore`	semafor
`sc_signal_rv<W>`	shodné s `sc_signal<sc_lv<W>>`, tedy signál datového typu vektor `W` prvků čtyřhodnotové logiky

Tabulka 1.2: Šablony některých signálů a jejich význam

Ukázka kódu 1.5: Signály

```
1   SC_MODULE(NovyModul)
2   {
3       sc_in<int> vstup;   //vstupní port datového typu int
4       sc_signal<sc_bit> s;  //jednobitový vnitřní signál modulu
5
6       SC_CTOR(NovyModul)  //konstruktor
7       {
8           //kód konstruktoru
9       }
10  };
11
12  NovyModul nm("NM");   //instance modulu
13  sc_signal<int> sig;   //signál datového typu bool
14  nm.vstup(sig);     //propojení vstupního portu modulu se signálem
15  sig = 7;       //přenos hodnoty 7 do modulu
```

1.4.2.4 Kanály

Kanály jsou komunikační prvky v SystemC. Jedná se o komunikaci mezi dvěma moduly. Kanály jednoduchým způsobem vytvářejí komplexní komunikaci. Každý kanál musí být odvozen od třídy `sc_channel`. V SystemC jsou dva základní typy kanálů. Primitivní a hierarchické.

Hierarchické kanály jsou implementovány jako moduly SystemC, jsou odvozeny od `sc_module`. Jsou založeny na myšlence, že kanál může obsahovat poměrně složité chování. SystemC využívá myšlenky definovat kanál jako věc, která implementuje rozhraní (viz kapitola 1.4.2.5), které je abstraktní. Rozhraní dává k dispozici virtuální metody pro přístup k danému kanálu. Hierarchický kanál je modul (může obsahovat to, co ostatní moduly - porty, procesy), který poskytuje metody pro realizaci funkcí přidružených rozhraní.

Na druhou stranu primitivní kanály nemohou obsahovat vnitřní strukturu a jsou proto jednodušší. Třída `sc_prim_channel` je základní třída pro všechny primitivní komunikační kanály v SystemC. Tato třída poskytuje komunikační kanály spolu s metodami přístupu do kanálu tak, aby byla zajištěna konzistence dat. Tento standard obsahuje množství předdefinovaných primitivních kanálů (viz příklad 1.6) pro běžnou komunikaci v simulovaném modelu. Základní kanály jsou např.:

- FIFO (`sc_fifo`)
- mutex (`sc_mutex`)
- semafor (`sc_semaphore`)

Ukázka kódu 1.6: FIFO z primitivních kanálů

```
1  SC_MODULE(Fronta)
2  {
3      sc_fifo<int> fifo;  //deklarace fronty
4
5      SC_CTOR(Fronta)  //konstruktor
6      {
7          SC_THREAD(zapis);
8          sc_fifo<int> fifo(5); //kapacita fronty je 5 celých čísel
9      }
10
11     void Fronta::zapis(int co)  //proces
12     {
13         while(true){
14             int cislo = co;  //do proměnné 'cislo' vloží hodnotu argumentu
15             wait(5, SC_NS);   //počká 5 ns
16             fifo.write(cislo);  //zapíše do fronty
17         }
18     }
19 };
20
21 Fronta fr("fr");  //instance
```

FIFO fronta (`sc_fifo`) má v SystemC mimo jiných předdefinovanou i metodu `write()`, která zapíše hodnotu argumentu do fronty. Pokud je fronta plná, počká, než se uvolní místo (např. předdefinovanou metodou `read()`). Když je objekt vytvářen, je stanoven počet slotů (defaultně 16 slotů).

1.4.2.5 Rozhraní

Rozhraní se používají v případě, že jsou ke komunikaci mezi moduly využity hierarchické kanály (viz ukázka kódu 1.7). Kanál je vždy vytvořen z jednoho nebo více existujících rozhraní. Rozhraní je v SystemC abstraktní třída odvozená od `sc_interface`. Podle definice rozhraní se pak pouze definují čistě virtuální metody přístupu. Bývají to zpravidla metody `write()` a `read()`. Tyto metody se nakonec ve všech odvozených kanálech implementují. Tím, že rozlišuje deklaraci rozhraní od implementace jeho metod, podporuje SystemC styl psaní kódu takový, že je oddělena komunikace od chování, což je klíčovým prvkem pro podporu lehkého přechodu z jedné úrovně abstrakce do druhé.

Metody pro čtení a zápis jsou neblokující (bez nějakého čekání se provádějí rovnou). To může být použito v případě, že obě metody budou vracet hodnotu typu `bool` (`true` pokud budou úspěšné). Funkce čtení a zápisu se provádějí v kontextu volajícího modulu. Jinými slovy, provádějí se jako části procesu `SC_THREAD` deklarovaného v komunikujících modulech.

Ukázka kódu 1.7: Rozhraní a hierarchické kanály

```
1  class MojeRozhrani: virtual public sc_interface  //rozhraní jako abstraktní třída
2  {
3  public:
4      virtual bool mujWrite(int) = 0;    //virtuální metoda pro zápis
5      virtual bool mujRead(int&) = 0;    //virtualní metoda pro čtení
6  };
7
8  class Kanal : public sc_module, public MojeRozhrani {  //třída odvozená od abstrakt.
9                                                          //třídy MojeRozhrani
10 private:
11     int data;           //proměnná, do které se zapisuje a ze které se čte
12     bool zapsano;       //příznak, zda je zapsaná do proměnné 'data' nějaká hodnota
13
14 public:
15     Kanal(sc_module_name nm) //konstruktor
16     {
17         //kód konstruktoru
18     }
19
20     bool mujWrite(int d)    //implementace metody pro zápis
21     {
22         if (zapsano == false) {
23             this->data = d;
24             zapsano = true;
25             return true;
26         } else {
27             return false;
28         }
29     }
30
31     void mujRead(int &c)    //implementace metody pro čtení
32     {
33         if (zapsano == true) {
34             c = this->data;
35             zapsano = false;
36             return true;
37         } else {
38             return false;
39         }
40     }
41 };
42
43 Kanal k("k");    //instance nového kanálu
44 sc_port<MojeRozhrani> port;        //port (je běžně součástí modulu, který bude
45                                    //komunikovat s jiným modulem) datového typu MojeRozhrani
46 port(k);       //propojení portu s kanálem
47 port.mujWrite(7);    //do kanálu se zapisuje přes port (hodnota 7)
```

Porty používají rozhraní ke komunikaci s kanály. Port slouží jako agent, který volání metody předá do kanálu jménem volajícího modulu. Port povoluje modulu volat metody deklarované v rozhraní. Na koncích komunikačních kanálů jsou rozhraní, na která se napojí porty, které jsou součástí modulů. Pokud tedy někdo (modul) zapisuje do kanálu, projde informace přes výstupní port do zapisovacího rozhraní a dále do kanálu. Projde celým kanálem a na konci čtecím rozhraním projde na vstupní port druhého modulu. Odesílatel pak může pouze volat metodu zápisu do rozhraní a příjemce může volat metodu čtení z rozhraní a vůbec nemusejí vědět, jak pracuje kanál. Jsou si vědomi jen svých rozhraní a implementace kanálu může být před moduly skryta.

1.4.2.6 Procesy

Popisují chování obvodu. Procesy jsou hlavní prvky výpočtu. Běží souběžně. V SystemC je každý proces implementován jako metoda ve třídě modulu. Tělo procesu se definuje jako tělo obyčejné metody v modulu (viz příklad 1.8).

Ukázka kódu 1.8: Proces

```
1  SC_MODULE(NovyModul)
2  {
3      SC_CTOR(NovyModul)  //konstruktor
4      {
5          //kód konstruktoru
6      }
7
8      void NovyModul::novyProces(void)  //proces
9      {
10         //kód procesu
11     }
12 };
```

V knihovně SystemC se rozlišují tři druhy procesů. V konstruktoru modulu se musí proces deklarovat pomocí jednoho z maker `SC_METHOD`, `SC_CTHREAD` nebo `SC_THREAD`. V příkladu 1.9 je vidět tělo konstruktoru a také použití procesů.

- `SC_METHOD` - jedná se o klasický kombinační proces, který by neměl obsahovat žádnou sekvenční logiku. Má definovaný statický citlivostní seznam.
- `SC_CTHREAD` - jde o proces, který implementuje sekvenční logiku. Je staticky citlivý na jeden hodinový signál. Při volání funkce `wait()` je způsobeno čekání na další hranu hodinového signálu.

- `SC_THREAD` - proces implementující sekvenční logiku, který může používat statický i dynamický citlivostní seznam. Volání funkce `wait()` způsobí čekání podle statického seznamu, ale lze použít další přetížené varianty `wait()` pro vytvoření dynamického seznamu.

Ukázka kódu 1.9: Procesy a jejich citlivost

```
1  SC_MODULE(NovyModul)
2  {
3      sc_in<bool> povolit; //vstup typu bool
4      sc_in<int> in; //vstup typu int
5      sc_in<bool> clk; //hodinový vstup
6      sc_out<int> out; //výstup typu int
7      sc_signal<int> interni; //interní signál
8
9      SC_CTOR(NovyModul) //konstruktor
10     {
11         SC_METHOD(procesJedna);     //deklarace METHOD procesu
12         sensitive << povolit << in;     //jeho statistický citlivostní seznam
13         SC_CTHREAD(procesDva, clk.pos());         //CTHREAD proces citlivý na
14     }                                                    //pozitivní hranu hodin
15
16     void procesJedna() //proces METHOD - zavolá se, když je změněn 'povolit' nebo 'in'
17     {
18         if (povolit.read()) //když je 'povolit' == true
19         {
20             interni.write(in.read()); //do 'interni' uložit to, co je na vstupu
21         }
22         else
23         {
24             interni.write(0); //jinak tam uložit nulu
25         }
26     }
27
28     void procesDva()   //proces CTHREAD
29     {
30         while (true)   //nesmí skončit
31         {
32             wait();    //čeká na hodiny
33             out.write(interni.read()); //na výstup poslat to, co je v 'interni'
34         }
35     }
36 };
```

U procesů `SC_CTHREAD` je hodinový signál uveden jako parametr makra. Navíc lze pro tento typ procesu definovat zvláštní resetovací signál pomocí metody `reset_signal_is(signal, polarita)`. Statický citlivostní seznam je pro kombinační obvody (`SC_METHOD`) definován zpravidla pomocí konstrukce `sensitive << signal1 << signal2 << ....`

Procesy `SC_METHOD` se spustí, když je detekována aktivita signálů, na které jsou citlivé. Tyto procesy musejí vždy bez blokování doběhnout až do konce. Procesy `SC_CTHREAD` by naproti tomu neměly skončit nikdy. Obvykle je toho dosaženo vložením nekonečné smyčky. Pokud zavolá proces funkci `wait()`, dává tím najevo, že veškerou činnost, kterou měl udělat, už udělal a chce být znovu probuzen až při další události na jeho citlivých signálech.

Simulační jádro pracuje podobně jako ve VHDL sekvenčně. V jednom okamžiku vždy běží právě jeden proces. Jedná se o kooperativní multitasking, kdy si jednotlivé úlohy předávají procesor dobrovolně tehdy, kdy to ony samy uznají za vhodné. Simulační jádro udržuje kontexty jednotlivých různých procesů (resp. míst v programu, kde byly pozastaveny) tak, že má každý proces svůj vlastní zásobník s návratovou adresou a s lokálními proměnnými. Vhodným přepínáním zásobníků je dosaženo kooperativního multitaskingu. U procesů `SC_CTHREAD` je možné definovat asynchronní signál pro reset. Pokud přejde tento signál do aktivního stavu, simulační jádro zahodí zásobník procesu, který má být resetován a znovu jej spustí s novým zásobníkem.

1.4.2.7 Události

Události musí být definovány během inicializace. Poskytují synchronizaci mezi procesy. SystemC je soubor C++ tříd a maker, které poskytují událostmi řízené simulační jádro v jazyce C++. To umožňuje návrháři modelu hardwaru simulovat souběžný běh procesů, kdy je každý z nich popsán běžnou syntaxí C++.

Součástí každého událostmi řízeného programu je cyklus, ve kterém se zachytávají vzniklé události a zpracovávají se. Události mohou být generovány operačním systémem, nebo program sám zjišťuje, zda nenastávají určité události. Ty jsou zařazeny do fronty událostí a když se daná událost vyskytne, simulační jádro na ni zareaguje příslušným způsobem a skončí teprve v případě, kdy je to řečeno příslušnou událostí.

1.4.2.8 Datové typy, proměnné

Proměnnými v SystémC jsou myšleny běžné proměnné z C++, které nejsou moduly, signály ani porty. Proměnné se obvykle používají v rámci procesů (lokální proměnné metody třídy), méně často jako členské proměnné C++ třídy, která implementuje modul. V takovém případě je totiž tato proměnná potenciálně přístupná všem procesům modulu, což by mohlo vést až k porušení principu paralelního zpracování procesů a k chybám v souběhu.

V SystemC je možné používat libovolné datové typy známé z C++, jako jsou `bool`, `char`, `int`, atd. Nad tyto standardní datové typy jsou postaveny v SystemC speciální datové typy (viz tabulka 1.3), které modelují hardware věrněji.

Třída nebo šablona třídy	Význam
`sc_bit`	1 bit, dvouhodnotová logika
`sc_logic`	čtyřhodnotová logika
`sc_bv<N>`	vektor `N` prvků typu `sc_bit`
`sc_lv<N>`	vektor `N` prvků typu `sc_logic`
`sc_int<W>`	`W`-bitové celé číslo se znaménkem, $W \leq 64$
`sc_uint<W>`	`W`-bitové celé číslo bez znaménka, $W \leq 64$
`sc_bigint<W>`	`W`-bitové celé číslo se znaménkem
`sc_biguint<W>`	`W`-bitové celé číslo bez znaménka
`sc_signed`	celé číslo se znaménkem
`sc_unsigned`	celé číslo bez znaménka
`sc_fixed<W>`	`W`-bitové fixed-point číslo
`sc_ufixed<W>`	bezznaménkové `W`-bitové fixed-point číslo
`sc_fix<W>`	fixed-point číslo
`sc_ufix<W>`	bezznaménkové fixed-point číslo

Tabulka 1.3: Některé datové typy v SystemC a jejich význam

1.4.3 Elaborace a simulace

Během tzv. elaborační fáze je struktura modelu sestavena programově. Elaborace začíná spuštěním zkompilovaného SystemC programu a končí okamžikem, kdy je programem zavolána funkce `sc_start()`. V tento okamžik začíná simulace. Simulace běží buď dokud není zastavena voláním funkce `sc_stop()`, nebo dokud neuplyne čas předaný funkci `sc_start()` jako parametr. Příkladem je `sc_start(5000, SC_NS)`, což znamená, že simulace bude probíhat 5000ns, pak skončí.

Během elaborační fáze je vytvářena struktura modelu instanciací patřičných modulů a propojováním jejich portů. Všechny vytvořené porty ve všech modulech musí být propojeny, jinak na konci elaborační fáze, před začátkem simulace, dojde k ukončení programu. Běh programu je zastaven s chybou, který port (pokud jich je více, výpis skončí prvním) není nikam napojen.

Hierarchie modulů musí být stromová. Každý modul musí být jednoznačně identifikován jménem `sc_module_name` (jedná se o `string`). Pokud je propojováno více modulů, konečné jméno vznikne spojením jmen instancí, která jsou předávána jako první parametry konstruktorům. Je klidně možné používat při všech instancích shodná jména, protože se případné konflikty vyřeší přidáním číselného indexu na konec jména automaticky.

Vykonávání začne ve funkci `sc_main()`, což je obdoba klasické funkce `main()`. V ukázce kódu 1.10 je vidět celý jednoduchý program v SystemC.

Ukázka kódu 1.10: Celý jednoduchý program v SystemC

```
1  #include <systemc.h>
2
3  SC_MODULE(Modul)   //submodul
4  {
5      sc_in<bool> enable;  //vstupní port
6      sc_in<int> in;    //vstupní port
7      sc_in<bool> clock;  //hodiny
8      sc_out<int> out;   //výstupní port
9
10     SC_CTOR(Modul)    //konstruktor
11     {
12         //kód konstruktoru a citlivostní seznam
13     }
14
15     void udelejNeco(){    //proces
16         //kód procesu
17     }
18 };
19
20 SC_MODULE(NadrazenyModul)  //nadřazený modul
21 {
22         //signály
23     sc_signal<bool> s1;
24     sc_signal<int> s2;
25     sc_signal<int> s3;
26     Modul m;      //instance submodulu
27
28     SC_CTOR(NadrazenyModul) : m("m")  //konstruktor
29     {
30         //propojování portů po jménu
31         m.enable(s1);
32         m.in(s2);
33         m.out(s3);
34     }
35 };
36
37 int sc_main(int argc, char *argv[])
38 {
39         //začátek elaborační fáze
40     sc_clock hodiny();           //generátor hodin
41     NadrazenyModul nm("nm");            //instance top-level modulu
42     nm.m.clock(hodiny);          //propojení portu v submodulu s hodinami
43     nm.s1 = true;             //změna signálu zavolá proces v submodulu
44     nm.s2 = 7;              //změna signálu zavolá proces v submodulu
45         //konec elaborace, začátek simulace
46     sc_start();
47     return(0);
48 }
```

V příkladu 1.10 jsou všechny moduly, porty i signály alokovány staticky na zásobníku. To je jedna možnost, jak alokovat potřebné instance, druhá možnost je dynamicky pomocí C++ operátoru `new`.

1.4.3.1 Dynamická alokace

Pomocí operátoru `new` mohou být všechny objekty SystemC (moduly, porty, signály, atd.) dynamicky alokovány na haldě. Alokace patřící jednomu modulu musí být prováděny v konstruktoru příslušného modulu (nebo ve funkcích z něj volaných). Dynamickou alokaci je vhodné použít zejména v těchto případech:

- Vytvoření pole modulů (portů, signálů, atd.). Pokud by totiž bylo vytvořeno statické pole modulu `Modul` např. o dvou prvcích `Modul pole[2];`, nebylo by možné v C++ zapsat v konstruktoru nadřízeného modulu inicializaci jednotlivých modulů z tohoto pole. Konstruktor by byl volán jako `SC_CTOR(NadrizenyModul) : pole[0]("m1"), pole[1]("m2")`, protože musí být vždy volán konstruktor i se jménem. Tento zápis by vedl na chybu při kompilaci.

- Pokud je potřeba modifikovat strukturu modelu během elaborace podle parametrů zadaných z vnějšku, např. z příkazové řádky.

Oba tyto případy jsou znázorněny v příkladu 1.11.

Ukázka kódu 1.11: Dynamická alokace a modifikující parametr

```
1  SC_MODULE(NadrazenyModul)
2  {
3      Modul **pole;    //pole ukazatelů na moduly
4
5          //konstruktor, kde parametr 'x' umožňuje z vnějšku zvolit počet modulů
6      NadrazenyModul(sc_module_name nm, int x) : sc_module(nm)
7      {
8          pole = new Modul*[x];    //dynamická alokace (pro zvolený počet modulů)
9          for (int i = 0; i < x; i++)
10         {
11             pole[i] = new Modul("m");  //dynamická alokace (stejná jména – automaticky
12                                        //dojde k upravení jména modulu)
13         }
14     }
15     SC_HAS_PROCESS(NadrazenyModul);
16 };
```

1.4.4 Výhody SystemC

1. Přebírá všechny rysy C++, který je stabilním programovacím jazykem, akceptovaným a uznávaným po celém světě. Má rozsáhlé jazykové konstrukce, což umožňuje jednodušeji napsat program a to s menším úsilím.

2. Knihovna je bohatá na datové typy - spolu s datovými typy podporovanými C++. SystemC podporuje použití spoeciálních datových typů, které se často používají při návrhu hardwaru.

3. Přichází se silným simulačním jádrem, aby mohli vývojáři lehce psát dobré testovací sady a simulovat je. To je dobré, protože funkční ověřování na úrovni simulace hardwaru ušetří spoustu času a peněz.

4. K C++ zavádí pojem času k simulování synchronních návrhů hardwaru. To je běžné ve většině HDL.

5. Zatímco většina HDL podporuje RTL (Register Transfer Level) úroveň návrhu, SystemC podporuje návrh na vyšší úrovni abstrakce. To umožňuje modelovat snadno velké systémy bez obav o implementaci. Také podporuje RTL návrh, je to podmožina obvykle označovaná jako SystemC RTL.

6. Souběžnost - pro simulaci souběžného chování digitálního hardwaru je simulační jádro konstruováno tak, že jsou včechny procesy prováděny současně, bez ohledu na pořadí, ve kterém byly volány.

1.5 Procesor pracující v aritmetice kódů zbytkových tříd

Soustava lineárních kongruencí se dá řešit paralelně (1.1.1.1) za pomoci r různých modulů a r procesorů pracujích v aritmetice kódů zbytkových tříd s těmito moduly. Tato práce se zaměřuje na výpočet na jednom takovém procesoru pracujím s jedním modulem. Takový procesor se skládá z aritmetických jednotek, které provádějí výpočet a z paměťových míst, registrů, kam se ukládají potřebné hodnoty při výpočtu.

1.5.1 Uspořádání aritmetických jednotek

Jak je popsáno v 1.3.3.1.2 nabízejí se dvě možnosti uspořádání aritmetických jednotek a paměťových buněk v procesoru tak, aby mohl výpočet probíhat alespoň do jisté míry paralelně. Vektor aritmetických jednotek propojený s paměťovou maticí zajistí sice paralelní výpočet, ale jen v rámci jedné řádky. Musí tedy zpracovávat jednotlivé řádky postupně. Struktura sestavená celá z aritmetických jednotek (obsahujících paměťové buňky) tak, aby tvořily matici, umožňuje paralelní výpočet jak v řádce, tak i mezi řádkami. U této varianty

je tedy možné dosáhnout výsledku rychleji. Na druhé straně je ale fakt, že při použití Gauss-Jordan-Rutishauserovy eliminační metody se posouvají hodnoty doleva a tím vpravo při každém eliminačním kroku přibyde celý sloupec aritmetických jednotek, které se k výpočtu již nepoužívají. Při výpočtu pomocí vektoru aritmetických jednotek přibyde při každém eliminačním kroku navíc pouze jedna neaktivní aritmetická jednotka. Postupné zpracovávání jednotlivých řádek při použití vektoru je ale časově velice nevýhodné.

Po dohodě s vedoucím této práce padla volba na tu strukturu, kde bude celý procesor sestaven z aritmetických jednotek v mřížce, matici (obrázek 1.3). Paralelní zpracování je oproti vektoru aritmetických jednotek nespornou výhodou. Nevýhoda, tedy hromadění se dále k výpočtu nevyužívaných aritmetických jednotek, bude vyřešena tak, že se tyto jednotky postarají o načítání dalších hodnot k výpočtu, tedy následující soustavy lineárních kongruencí. To znamená, že jak se bude výpočet posouvat směrem doleva, vpravo budou aritmetické jednotky načítat další data a distribuovat je směrem doleva vždy do dalších neaktivních jednotek. Tím se nevýhoda setře, protože po celou dobu výpočtu budou pracovat i jednotky, které už nejsou ve výpočtu zapojené a až dojde k vyřešení jedné soustavy, další už bude načtená v příslušných paměťových buňkách a bude tedy možné zahájit řešení i této nebo bude potřeba donačíst jen zbytek soustavy a pak ji začít řešit. Takto je možné řešit paralelně po řádcích i sloupcích celou soustavu a hned po skončení řešit další bez kompletního načítání nových hodnot po výpočtu, atd.

1.5.2 Celkový průběh výpočtu

Celá síť se bude skládat z aritmetických jednotek. V každé z nich budou paměťová místa, registry. Nebude žádné sdílené paměťové úložiště. Všechna úložná místa má aritmetická jednotka pouze lokální. Také nebude žádný nadřazený prvek, který by jednotlivým jednotkám zadával příkazy, co mají dělat. Jednotky budou provádět výpočet nezávisle a řídit se samy. Celý výpočet bude asynchronní z toho důvodu, že je tímto způsobem možné výpočet značně urychlit, protože v případě, že by jednotky musely čekat, než budou všechny hotové s některou operací, se potřebný čas pro dosažení výsledku zvyšuje. Výpočet bude řízen daty, tzn., že pokud bude mít příslušná aritmetická jednotka ve svých registrech všechna data potřebná k některému výpočetnímu kroku, provede jej.

Kvůli asynchronnosti a také tomu, že nebude existovat žádný nadřazený prvek, který by řídil výpočet a komunikaci, bude nutné zajistit velké množství opatření, aby nedocházelo např. k uváznutí procesů, výpočtu s nesprávnými hodnotami, nebo nechtěnému přepisování hodnot.

Výpočetní jednotky budou řízeny pouze hodinovým signálem, který bude jen jeden a podle něj se budou řídit všechny jednotky. Je tedy jasné, že výpočet (a také komunikace) bude probíhat v jednotlivých taktech a každý takt je společný pro celou architekturu. Pokud bude jednotka potřebovat do ně-

kterého směru zapsat data, provede tak, dál už se o data nestará a jednotka sousední (která je adresátem těchto dat) z komunikačního kanálu data přečte a zachová se příslušným způsobem.

1.5.3 Propojení aritmetických jednotek

Jednotky je samozřejmě nutné mezi sebou propojit. Jednak kvůli tomu, že se při použití Gauss-Jordan-Rutishauserovy metody posouvají hodnoty doleva, takže je nutné sdílet data, tedy přesněji posílat data a jednak kvůli tomu, že musí být nějakým způsobem řízen celý výpočet, tedy posílání signálů. Jelikož je z principu zvolené výpočetní metody nutné mít rozmístěné výpočetní jednotky do tvaru matice, je nasnadě propojit každou aritmetickou jednotku se sousedními jednotkami umístěnými ve vertikální a horizontální poloze. Při bližší analýze výpočetního algoritmu lze dospět k ověření tohoto faktu a je tedy potřeba uskutečnit spojení ve všech směrech. Nabízejí se dvě možnosti jaké propojovací kanály použít:

1. jeden kanál propojující dvě aritmetické jednotky, obě do něj mohou zapisovat i z něho číst, tedy vstupně/výstupní,

2. dva kanály mezi dvěma aritmetickými jednotkami, jeden k zápisu, jeden ke čtení, tedy vstupní a výstupní.

Kanály sloužící k veškeré komunikaci budou využívat druhou variantu, tedy propojení jednotek pomocí dvou kanálů, jak je vidět na obrázku 1.4. Bude tomu tak z toho důvodu, že sousední aritmetické jednotky mohou komunikovat zároveň, protože každá z nich zapíše do jiného kanálu, kdežto v případě použití jediného komunikačního spoje by bylo nutné v případě zapisování hodnoty sousedními jednotkami ve stejném čase řešit to, aby byla správná hodnota doručena na správnou stranu kanálu.

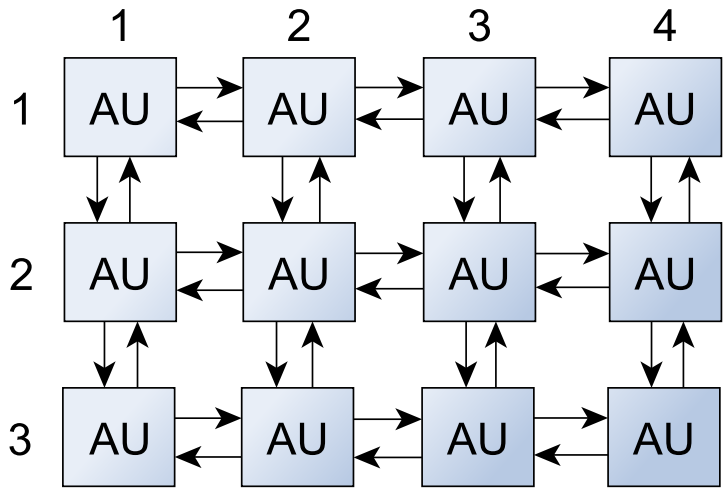
1.5.4 Aritmetická jednotka

Aritmetická jednotka je prvek, který bude provádět celý výpočet. Je tedy potřeba, aby probíhal na aritmetické jednotce proces, který bude provádět Gauss-Jordan-Rutishauserovu eliminační metodu. Dále je nutné, aby měl výpočetní proces k dispozici úložné místo, registry, do kterých bude ukládat hodnoty spočtené, nebo přijaté od některé sousední aritmetické jednotky. Je tedy jasné, že musí výpočetní jednotka umět také číst ze vstupních kanálů a samozřejmě do nich i zapisovat. Nabízejí se varianty, jak vyřešit výpočet a komunikaci v aritmetických jednotkách:

1. v každé aritmetické jednotce bude probíhat pouze jediný proces, který bude vykonávat jak výpočetní operace, tak komunikaci, takže bude umět do komunikačních kanálů zapisovat a umět z nich i číst,

2. výpočetní proces nechat jako výpočetní proces, který se nijak nebude starat o komunikaci a oddělit čtecí proces, ten mít jako samostatný, stejně tak, jako zapisovací proces, tedy na každé aritmetické jednotce budou probíhat tři procesy.

Druhá varianta se jeví jako výhodnější, protože procesy mohou běžet synchronně, tudíž aritmetická jednotka může např. ve stejnou dobu přijmout data, začít s nimi počítat a zároveň je přeposílat dál.



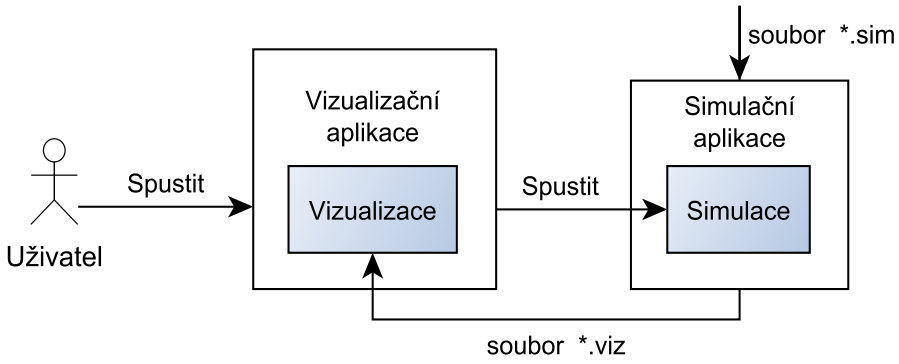
Obrázek 1.4: Matice aritmetických jednotek propojených vstupními a výstupními kanály

KAPITOLA 2

Návrh a realizace

V této kapitole se objevuje nejdříve návrh a realizace simulační aplikace, potom návrh a realizace vizualizační aplikace. Rozdělení do dvou aplikací je z toho důvodu, že každá slouží k jinému účelu. Simulační aplikace k samotnému řešení soustav lineárních kongruencí, vizualizační aplikace ke grafickému zobrazení běhu simulace.

Uživatel spustí vizualizační aplikaci (grafické uživatelské rozhraní). Odtud spustí simulaci (konzolová aplikace). Vstupem simulace je soubor *.sim, který obsahuje soustavy lineárních kongruencí (podle toho, jaké rozměry má zadaná soustava ve vstupním souboru, takový počet funkčních jednotek se vytvoří). Po vyřešení všech soustav je vytvořen soubor *.viz. Tento soubor lze otevřít ve vizualizační aplikaci, která podle jeho obsahu na obrazovku vykresluje průběh simulace. Celé schéma je vidět na obrázku 2.1.



Obrázek 2.1: Schéma použití vizualizační a simulační aplikace

Následujících sedm částí této kapitoly se zabývá vždy nejdříve návrhem a potom realizací určité části simulační aplikace. V poslední, osmé, části kapitoly (2.8) se vyskytuje návrh a realizace vizualizace.

2.1 Aritmetická jednotka

2.1.1 Návrh

Aritmetická jednotka je prvek, který provádí výpočet. K tomu bude potřebovat všechny tyto náležitosti:

- hodinový signál
- proces výpočtu provádějící operace
- proces čtení z komunikačního kanálu
- proces zápisu do komunikačního kanálu
- paměťové buňky (registry) k uložení hodnot vypočtených a mezivýsledků
- paměťové buňky k ukládání hodnot přečtených z komunikačních kanálů
- paměťové buňky k ukládání hodnot k odeslání
- vstupní porty
- výstupní porty

Celý chod aritmetické jednotky se bude synchronizovat hodinovým signálem. Nutnost procesu, který bude provádět operace výpočtu a registrů k ukládání hodnot vypočtených a mezivýsledků, je zřejmá. Procesy umožňující komunikační schopnosti - zápis do komunikačních kanálů (odesílání hodnot - vypočtené výsledky, distribuování potřebných hodnot dál od zdroje) sousedním jednotkám, nebo signálů (určité řízení výpočtu, protože každá jednotka bude provádět výpočet nezávisle na ostatních). Paměťovými buňkami na příchozí hodnoty je myšlena taková paměť, do které se budou hromadit hodnoty přečtené ze všech komunikačních spojů procesem čtení a odkud se budou zpracovávat procesem výpočtu. Do paměti pro odesílání se budou ukládat výpočetním procesem zpracované hodnoty, které bude vkládat do komunikačních kanálů zapisovací proces. Seznam vstupních a výstpních portů je potřebný, protože v případě zápisu se zapisuje do určitého portu (obdobně to je i se čtením).

K výpočtu tedy bude sloužit jeden proces (viz 2.6), který se při zahájení simulace rozběhne. Stejně tak tomu bude s procesem čtení (viz 2.4.2). Jelikož bude výpočet řízen daty, bude se proces výpočtu zastavovat a čekat na hodnoty ze vstupních portů (přečtené procesem čtení), pokud nebude již nic

ke zpracování v paměťových buňkách sloužících k ukládání příchozích hodnot. Vždy s hodinovým signálem přečte proces výpočtu hodnotu ze čtecí paměti a zpracuje ji podle toho, jaký typ hodnoty to je (jelikož je výpočet řízen daty a do výpočetní jednotky mohou vstupovat hodnoty až ze čtyř směrů, musí být jasné, co za hodnotu - druh hodnoty - je na vstupu). Pokud bude čtecí paměť prázdná, proces výpočtu bude čekat na událost vyvolanou procesem čtení. Taková událost bude vyvolána vždy v případě, že proces čtení přečte z některého vstupního portu hodnotu, kterou uloží do čtecí paměti. Proces čtení se nikdy nezastaví, běží neustále a pořád dokola kontroluje všechny vstupní porty.

Jelikož procesy běží zároveň, může proces čtení zapisovat do čtecí paměti hodnoty i když proces výpočtu zrovna provádí nějaké operace. Až potom výpočetní proces doběhne do části, kde kontroluje čtecí paměť, ta, jelikož do ní mezitím proces čtení zapsal hodnotu, nebude prázdná, a tak hodnotu přečte a hned, bez čekání na událost vyvolanou čtecím procesem, ji začne zpracovávat. Událost vyvolaná čtecím procesem aktivuje čekající výpočetní proces pouze v případě, že proces skutečně čeká. Pokud nečeká, nijak se zavolání události neprojeví.

Proces zápisu (viz 2.4.1) se rozeběhne v případě, že proces výpočtu potřebuje nějakou hodnotu odeslat sousední jednotce. Opět je výhoda v tom, že procesy běží zároveň. Proto proces výpočtu (v případě, že chce zapisovat do komunikačního kanálu) jen aktivuje proces zápisu a dál pokračuje ve výpočtu a o zápis se nestará, protože o ten se postará proces zápisu, který po provedení zápisu skončí a je až pak opět aktivován procesem výpočtu.

2.1.2 Realizace

Aritmetická jednotka je objekt třídy `Procesor` (tyto objekty potom provádějí samotný výpočet), která je odvozená od třídy `sc_module`. Jedná se tedy o modul. V parametru konstruktoru se kromě jména (musí být jedinečné) předává ještě informace o tom, na jaké pozici se jednotka v celé matici nachází. To je potřeba kvůli tomu, aby se vytvořil správný počet vstupních a výstupních portů `sc_port`. Ty jsou typu `CteniZKanalu` a `ZapisDoKanalu` (viz 2.2.2) a jsou uloženy ve dvou vectorech (vstupní a výstupní). Mohou existovat 2 - 4 vstupní i výstupní porty v závislosti na poloze jednotky v matici. Způsob uložení portů do vektoru je v pořadí podle směru, ve kterém od jednotky leží sousedé, tedy „doleva, doprava, nahoru, dolů“, což znamená, že pokud má jednotka v konkrétním směru souseda, je ve vectorech na dané pozici uložen vstupní nebo výstupní port, pokud nemá, je ve vectorech na dané pozici `NULL`.

Dále se v konstruktoru vytvářejí procesy a přiřazuje se jim citlivostní seznam. Proces výpočtu a čtení jsou typu `SC_THREAD`, tedy procesy se sekvenční logikou. Jsou citlivé na pozitivní hranu hodinového signálu (s jejím příchodem jsou probuzeny), zapsáno jako `sensitive << Clock.pos();`. Proces zápisu je typu `SC_METHOD`, tedy proces, který skončí po provedení operace. Je citlivý na událost `sc_event` vyvolanou výpočetním procesem.

Všechny registry, které aritmetická jednotka během výpočtu používá, mají stejnou velikost, a to 32 bitů. Je do nich tedy možné uložit číslo o maximální velikosti $2^{31} - 1$, protože jeden bit zabírá znaménko. Je použit datový typ `sc_int<32>`.

Modul aritmetické jednotky ve třídě `Procesor` je podřízený modul. Jeho nadřazeným modulem je třída `CPU`. Ve třídě `Procesor` jsou některé vstupní porty `sc_in`, např. vstup hodinového signálu, nebo vstup pro zahájení celého výpočtu. Na tyto vstupy se v nadřazeném modulu napojí příslušné signály. Do těchto signálů se pak zapisují data a ty přejdou přes nadřazený modul, přes vstupní porty až do třídy `Procesor`. V konstruktoru třídy `CPU` je vždy vytvářen objekt třídy `Procesor` a je použito vázání po jménu mezi signály a vstupními porty. K tomu slouží v SystemC operátor přetížené závorky, zapisuje se jako `vstupni_port(signal);`.

2.2 Komunikační kanál

2.2.1 Návrh

Komunikační kanál musí propojovat dva moduly tak, aby do něj bylo možné na jedné straně uložit data a na druhé straně je přečíst. Je nutné, aby se do kanálu uložila jednak samotná hodnota a jednak také příznak, o jaký typ hodnoty se jedná. Komunikace v SystemC se dá řešit pomocí hierarchických kanálů (viz. 1.4.2.4), kdy bude kanál vytvořen pomocí rozhraní, na která jsou napojené vstupní a výstupní porty. V případě zápisu hodnoty se metoda zápisu (která se volá na výstupní port - ten je typu rozhraní pro zápis) přesvědčí, že je kanál prázdný a zapíše data. V případě čtení se čtecí metoda (ta se volá na vstupní port, který je typu rozhraní pro čtení) přesvědčí, že kanál není prázdný a přečte data. Obě metody musí mít příznak, zda byly úspěšné, či nikoliv. Metoda pro zápis uloží příslušná data z výstupního portu do k tomu určeného paměťového prostoru v kanálu a metoda čtení naopak odtamtud data přesune na vstupní port přijímacího modulu.

2.2.2 Realizace

Komunikační kanál ve třídě `Kanal` je odvozen od `sc_module` (je to tedy modul) a implementuje rozhraní `sc_interface`, které je abstraktní (třídy `ZapisDoKanalu` a `CteniZKanalu`) a poskytuje metody pro zápis a čtení do/z komunikačních kanálů. V kanálu musí být místo na uložení dat, ze kterého se bude zase číst. Používají se takové kanály, kde je paměťové místo pouze na jednu položku typu `Paket`, ve které je uložena hodnota a také typ hodnoty (o jakou hodnotu se jedná). K zápisu hodnoty se používá 32 bitů `sc_int<32>` a k zápisu typu hodnoty se používají 4 jednobitové hodnoty `sc_bv<4>`, pomocí nichž je možné zakódovat až 16 různých typů hodnot. Ke kanálu je přistupováno pomocí metod `zapsat(Paket)` a `precist()`, které se volají na

příslušné porty (ty jsou typu `ZapisDoKanalu` a `CteniZKanalu`, tedy rozhraní `sc_interface`) v aritmetické jednotce. Obě metody vrací `true`, pokud zápis/čtení proběhlo úspěšně (`false` vrací, pokud zapisuje, ale kanál je obsazený a nebo pokud čte, ale kanál je prázdný). K dispozici jsou také metody `MoznoCist()` a `MoznoZapsat()`, které říkají, jak je na tom kanál s obsazeností.

2.3 Struktura

2.3.1 Návrh

Maticové struktury bude docíleno tak, že se propojí sousední aritmetické jednotky ve svislém i vodorovném umístění. Bude se tedy vytvářet tolik jednotek, jaký rozměr má soustava, která se počítá (i s vektorem pravých stran). Struktura se vytváří podle zadaných parametrů (rozměr soustavy) postupně vždy po řádcích. Vždy se vytvoří aritmetická jednotka, potom další, atd. až do počtu sloupců. Takto se vytvoří celá řádka (vektor) aritmetických jednotek. Tento proces se opakuje tolikrát, kolik má počítaná soustava řádků.

Při konstrukci aritmetické jednotky se vždy uvádějí potřebné parametry, jako např. ukazatel na hodinový signál (udává jednotlivé takty výpočetních jednotek při výpočtu), nebo umístění jednotky v matici. To je potřeba k tomu, aby si příslušná jednotka vytvořila správný počet portů. Porty slouží ke čtení (vstupní porty) a zapisování (výstupní porty) z/do komunikačních kanálů. V závislosti na tom, kde je jednotka v matici umístěna, má vždy dva až čtyři vstupní a výstupní porty. Po vytvoření všech aritmetických jednotek uložených ve struktuře matice je potřeba je správně propojit. Nikdy nesmí existovat vytvořené porty, které nejsou napojené na žádný kanál (to vede k chybě při spuštění simulace).

Vždy se prochází vytvořená struktura aritmetických jednotek po řádcích zleva doprava a vytvořené komunikační kanály se jednotkám přiřazují. To znamená, že na vstupní a výstupní porty se napojí kanály. Jde o to, že sousední dvě jednotky budou propojeny kanálem křížem. To znamená, že např. levá ze dvou jednotek bude mít napojen výstup na vstup pravé jednotky a svůj vstup na výstup pravé jednotky. Tak tomu bude ve všech směrech, ve kterých je jednotka s nějakou jinou jednotkou propojena. Při přiřazování kanálů se vždy levé, nebo horní jednotce přiřadí ještě nepoužitý (nový) kanál, kdežto jednotce vpravo, nebo dole se přiřadí křížem kanál již existující napojen na příslušný port sousedící jednotky. Takto se propojí všechny čtecí i zapisovací porty všech jednotek pomocí kanálů a vznikne struktura, která je znázorněná na obrázku 1.4.

V případě, že by se tato struktura realizovala pomocí skutečného hardwaru, mohly by být jednotky ležící v matici ve druhém až v posledním sloupci jednodušší, operace, které by musely umět, by byly pouze sčítání a násobení v modulární aritmetice. Jednotky v prvním sloupci musí být složitější, protože musejí provádět negaci a multiplikativní inverzi v modulární aritmetice.

2.3.2 Realizace

Při vytváření struktury se nejdříve vytvoří objekt třídy `Matice` (parametrem je rozměr soustavy lineárních kongruencí). Přímo z konstruktoru se potom vytvářejí jednotlivé výpočetní jednotky, tedy moduly třídy `CPU` (v jejím konstruktoru se vytváří vždy modul třídy `Procesor`), které jsou poté uloženy ve dvourozměrném vectoru. Do vectoru se ukládají právě tyto nadřazené moduly, protože se společně s ukazatelem na modul výpočetní jednotky uloží i příslušné signály dané aritmetické jednotky. Třída `Matice` obsahuje metody `vytvorKomunikacniKanaly()`, která vytváří komunikační kanály vždy s unikátním jménem a ukládá je do vectoru vždy v příslušné aritmetické jednotce a metodu `napojKomunikacniKanaly()`, která prochází maticovou strukturu výpočetních jednotek a vytvořené kanály napojuje křížem na příslušné vstupní a výstupní porty jednotek. K napojování kanálů na porty existuje v SystemC přetížený operátor závorek. Konkrétně tedy vypadá zápis takto `port(kanal);`. Obě tyto metody se volají na objekt třídy `Matice`. Tím vznikne celá struktura s napojenými komunikačními kanály.

2.4 Komunikace

2.4.1 Proces zápisu

2.4.1.1 Návrh

Proces zapisování do komunikačního kanálu bude schopen zapsat do více než jen do jednoho směru zároveň. Je to proto, že by se nějaká hodnota například musela distribuovat po celé matici, tedy poslat z aritmetické jednotky všem jednotkám ve sloupci a dané řádce. Tedy z řádky, kde leží jednotka, která je zdrojem této hodnoty, posílat jak nahoru, tak dolů, tak doprava. A to ve stejném taktu.

Funguje to tak, že výpočetní proces aktivuje událostí proces zapisovací s tím, že je potřeba daná hodnota poslat do několika konkrétních směrů. Nicméně pokud už v některém směru nelze hodnotu dále přeposlat (aritmetická jednotka tvoří okraj matice), je nutné poslat hodnotu alespoň ve zbývajících směrech (pokud jich bylo více), nebo neposílat hodnotu dál (pokud byl jen jeden směr).

Je tedy potřeba kontrolovat, zda existují porty výstupu v tom směru, kam by měla být hodnota distribuována. Tato kontrola je vložena mezi výpočetní a zapisovací proces. Za tímto účelem bude vždy změněna hodnota signálu, do kterého se bude pro konkrétní aritmetickou jednotku zapisovat taková hodnota, která je nastavena podle toho, zda existují porty, do kterých se má hodnota přeposlat. Proto se proces zápisu vždy řídí tímto signálem, který říká, do kterých všech výstupních portů se má hodnota zapsat (všechny tyto porty v dané jednotce skutečně existují a jsou propojeny se sousedy).

V jednom taktu tedy postupně projde aktivovaný proces zápisu všechny výstupní porty (přitom proces výpočtu při zapisování hodnot do komunikačních kanálů nezávisle vykonává úplně jiné operace, což je možné díky souběžnosti procesů) a podle hodnoty v signálu u každého rozhodne, zda do něj zapsat, nebo ne. Může tak hodnotu zapsat do všech čtyřech výstupních portů v jednom taktu. Po provedení operace zápisu proces skončí a je opět aktivován od výpočetního procesu s další žádostí o odeslání.

2.4.1.2 Realizace

Proces zápisu `zapisovani()` v modulu třídy `Procesor` je typu `SC_METHOD`, tedy takový proces, který je aktivován příslušnou událostí `sc_event`, vykoná všechny operace a skončí. Pokud je aktivován, vždy postupně prochází přes všechny výstupní porty `sc_port<ZapisDoKanalu>` a kontroluje signál `sc_signal<sc_bv<4>>`, což je signál, jehož datovým typem jsou 4 bity (každý bit znamená jeden směr výstupu z výpočetní jednotky - opět je použito řazení „doleva, doprava, nahoru, dolů“).

Hodnota tohoto signálu je nastavena metodou `odeslat()`, jež má tři parametry. Hodnota, typ hodnoty a všechny směry, kam se hodnota odesílá. Tato metoda je volána výpočetním procesem všude v místech, kde je potřeba zapsat do komunikačních kanálů. Metoda zkontroluje, zda existují ty porty výstupu, do kterých se má zapsat hodnota. Do signálu nastaví hodnotu rovnou jedné každému ze čtyř bitů, do kterých výstupních portů se má hodnota zapsat a skutečně v aritmetické jednotce existují.

Potom událostí `sc_event` (`udalost_zapisovani`) metoda aktivuje proces `zapisovani()`. Ten pro všechny čtyři porty zkontroluje v signálu vždy ten správný bit a pokud je jeho hodnota 1, zavolá na daný výstupní port metodu `zapsat(Paket)`, přičemž v objektu třídy `Paket` je uložena hodnota a její typ. Po provedení jedné iterace přes všechny porty proces zápisu skončí a je opět aktivován událostí `sc_event` v případě dalšího zápisu.

2.4.2 Proces čtení

2.4.2.1 Návrh

Obdobně jako proces zápisu, i proces čtení bude moci číst z více než jen z jednoho vstupního portu ve stejném taktu. Bude moci přečíst hodnoty ze všech čtyř vstupních kanálů najednou, pokud tam budou. Tento proces se nikdy nezastaví. Je to důležité, protože celý výpočet probíhá nezávisle na jednotlivých aritmetických jednotkách, a proto je výpočet řízen daty přečtenými ze vstupních portů. Proces čtení se tedy nesmí nikdy zastavit, protože musí neustále kontrolovat, zda není v některém kanálu hodnota, která by znamenala od aritmetické jednotky nějakou interakci.

Pokud není, kontroluje stále dál ve smyčce vždy jeden port za druhým. Pokud je v některém vstupním kanálu zapsaná hodnota, tuto hodnotu přečte

(samozřejmě i s typem hodnoty) a uloží ji do paměťového prostoru, který je určen k tomuto účelu. Vždy v jednom taktu prověří všechny kanály a všechny přečtené hodnoty zapíše do čtecího paměťového místa. Pokud alespoň z jednoho vstupního portu byla přečtena hodnota, je událostí probuzen výpočetní proces, který právě na tuto událost čeká. Čeká dokud nebudou nějaká data ke zpracování (daty řízený výpočet), tedy dokud nebudou data přečtena ze vstupních kanálů.

I zde se projeví souběžnost procesů. Pokud výpočetní proces právě provádí výpočet, čtecí proces úplně stejně uloží přečtené hodnoty ze vstupních portů do čtecí paměti, stejně zavolá i událost, která by měla výpočetní proces probudit, ale jelikož ten nečekal na probuzení, událost bude bez odezvy. Výpočetní proces vždy před tím, než začne čekat, kontroluje, zda je čtecí paměť prázdná (bude se používat mutex, kdyby chtěly oba procesy přistupovat do paměti ve stejnou dobu). V tomto případě prázdná nebude, takže po dokončení zpracovávání současné operace nebude čekat, ale přečte v dalším taktu ze čtecí paměti hodnotu a začne s jejím zpracováváním.

2.4.2.2 Realizace

Proces čtení `cteni()` v modulu třídy `Procesor` je typu `SC_THREAD`, tedy takový proces, který je řízen hodinovým signálem a nikdy neskončí, opakuje se pořád dokola. V každé iteraci (taktu) vždy postupně prochází přes všechny (které v modulu existují) vstupní porty `sc_port<CteniZKanalu>` a kontroluje, zda je v kanálu zapsaná hodnota. To provádí tak, že volá metodu `precist()` na daný vstupní port. Tato metoda vrací `true`, pokud byla v kanálu uložena hodnota, která byla voláním této metody přečtena.

Pokud je tedy výsledek `true`, je hodnotou a jejím typem naplněn objekt `Paket`. Tento objekt je pak uložen do fronty `prichozi_hodnoty` (před zapisováním zamkne `sc_mutex`, kdyby chtěl ve stejné chvíli do paměti přistupovat i výpočetní proces). Po prozkoumání všech (provede se ve stejném taktu) ve výpočetní jednotce dostupných vstupních kanálů a v případě, že byl `Paket` naplněn, je zavolána událost `sc_event`, na kterou čeká výpočetní proces. Čtecí proces se pak v dalším taktu opět bude snažit číst ze všech vstupních kanálů.

2.5 Mechanizmus sekvenčního řešení více soustav

2.5.1 Rozšíření struktury

2.5.1.1 Návrh

Aby bylo možné načítat a distribuovat data z následující soustavy lineárních kongruencí, která se začne řešit po vyřešení aktuální soustavy, je potřeba ke stávající matici výpočetních jednotek připojit ještě další moduly. Jedná se o moduly, které budou plnit následující funkce:

1. zásobovat aritmetické jednotky dalšími daty k výpočtu,
2. sbírat vypočtené části řešení, pořadí částí řešení, modul a hodnoty pivotů (pro výpočet determinantu).

Přítomnost prvního modulu je zřejmá. Ten bude do aritmetických jednotek posílat další data, konkrétně modul (pokud bude jiný, než v soustavě právě řešené) a samotné hodnoty do jednotlivých aritmetických jednotek. Přítomnost druhého modulu je nutná, protože po vyřešení jedné soustavy budou mít výpočetní jednotky v levém sloupci matice uložené výsledky, pořadí, ve kterém mají být výsledky seřazené, modul, ve kterém probíhal výpočet a pivoty pro určení determinantu. Toto všechno je potřeba distribuovat pryč z matice aritmetických jednotek, protože ta musí počítat další soustavu.

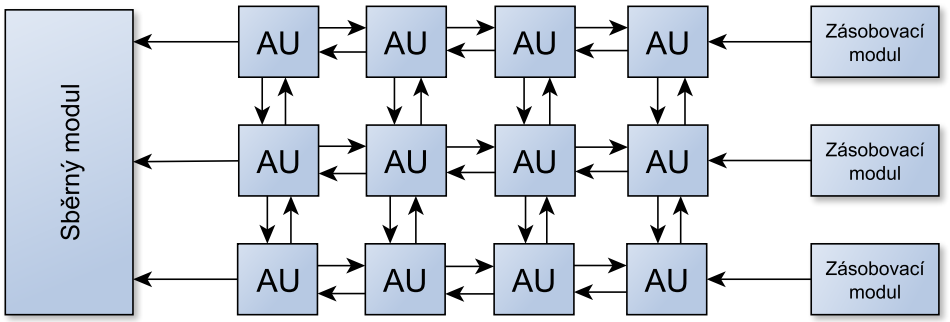
Jelikož se výpočet posouvá v matici směrem doleva, vznikají vpravo volné, neaktivní, výpočetní jednotky. Z tohoto důvodu bude zásobovací modul připojen doprava, za poslední sloupec výpočetních jednotek. Jelikož ale výpočet probíhá nezávisle na jednotlivých jednotkách, nemusí být ve všech řádkách posunut výpočet doleva ve stejném taktu. Každá řádka se může posunout jindy. Z tohoto důvodu nebude existovat jen jeden zásobovací modul, ale bude jich tolik, kolik je řádků. Každá aritmetická jednotka v posledním sloupci matice má proto jeden vstupní port zprava, na který je napojen kanál, jehož vstupem je na druhé straně zásobovací modul (viz obrázek 2.2).

Modul, který bude sbírat výsledky, pořadí výsledků, modul a počítat determinant může být jen jeden, protože nezáleží na pořadí, ve kterém do něj dorazí jednotlivé hodnoty. Důležité je pouze to, aby měl tolik vstupních portů, kolik existuje řádků matice. Každá aritmetická jednotka v levém sloupci matice proto bude mít vlevo výstupní port, na který bude připojen kanál ústící na vstupní port sběrného modulu. Tento modul potom pozná podle toho, z jakého portu data přečetl, z které výpočetní jednotky přišla, ale nezáleží mu na pořadí příchodu. Podle typu hodnoty pozná, o jakou hodnotu se jedná a jak s ní naložit. Tento modul se dále může starat o převod řešení zpět do racionálních čísel, což ale není náplní této práce.

2.5.1.2 Realizace

V konstruktoru třídy `Matice` se vytvářejí jednotlivé výpočetní jednotky. Zároveň s tím se vytvářejí i zásobovací a sběrný modul. Jedná se o objekty tříd `Zasobovac` a `Sberac`. Sběrný modul se vytvoří pro celou strukturu pouze jeden, parametry jeho konstruktoru jsou ukazatel na hodinový sigál a rozměr matice (kvůli tomu, aby si mohl vytvořit správný počet vstupních portů). Zásobovacích modulů se vytváří tolik, kolik je řádků v matici. Parametry konstruktoru jsou hodinový signál a soubor se vstupními daty (odtud bude čerpat při zásobování výpočetní matice novými daty). Po vytvoření celé matice a zbylých potřebných modulů se pomocí metody `vytvorKomunikacniKanaly()` vytvoří komunikační kanály i mezi příslušnými okrajovými aritmetickými jednot-

kami a moduly. Potom se pomocí metody `napojKomunikacniKanaly()` už jen na vstupní a výstupní porty správně napojí vytvořené komunikační kanály. Výsledná struktura je znázorněna na obrázku 2.2.



Obrázek 2.2: Matice aritmetických jednotek s připojeným sběrným a zásobovacími moduly

2.5.2 Načítání a výpočet

2.5.2.1 Návrh

Hodnoty se budou nasouvat zprava a postupně se rozdistribuují (budou se vzájemně vytlačovat) až do prvního sloupce matice aritmetických jednotek. Nejdříve se bude nasouvat modul nové soustavy. Ten se bude posílat pouze v případě, že je jiný, než modul dosavadní. Musí být nasouván před samotnými hodnotami soustavy, aby v případě, že je modul shodný, mohly být poslány rovnou hodnoty soustavy a jednotky v matici po přijetí těchto hodnot poznaly, že když modul nepřišel, použije se k výpočtu další soustavy modul dosavadní.

Nasouvání bude probíhat v průběhu výpočtu vždy postupně. Až když výpočet skončí, nasune se zbytek dat potřebný k řešení další soustavy najednou. To znamená, že jak přibyde neaktivní jednotka, v dalším kroku se zaplní hodnotou z další soustavy (nebo modulem). V rámci výpočtu se směrem doprava posílají dvě hodnoty (multiplikativní inverze pivota nebo negace hodnoty uložené v jednotce). Tyto hodnoty se vždy distribuují až do posledního sloupce matice aritmetických jednotek. Nicméně v těch jednotkách, ve kterých už výpočet neprobíhá, se tyto hodnoty převedou na hodnotu toho typu, který říká, že je nutné nasunout další data. V každém eliminačním kroku tedy dojde k nasunutí další hodnoty.

Zásobovací modul bude udržovat kanál vždy naplněný. Aritmetická jednotka v posledním sloupci matice (v případě, že už je neaktivní) po obdržení

hodnoty s žádostí o nasunutí dat povolí zakázané vstupní porty spojující kanálem tyto jednotky se zásobovacími moduly a tím přečte hodnotu z kanálu. Hned zase porty zakáže. Povolí je opět v době přijetí další hodnoty s žádostí o nová data.

V případě, že už v jednotce jsou načtená nová data a jednotka čte další nová data, uloží tato na místo původních a původní nová data pošle doleva. Takto se posune naplnění matice o sloupec vlevo. Je tedy jasné, že postupně bude jednotka ležící v posledním sploupci matice načítat hodnotu z nové soustavy, která patří do jednotky prvního sloupce matice, potom do jednotky z druhého sloupce, atd., až jako poslední načte hodnotu, která patří do této jednotky, tedy do jednotky v posledním sloupci.

V případě, že už je soustava dořešena, je směrem doprava odeslána žádost o nová data říkající, že už se mají nasunout všechny najednou, ne postupně. Jednotky v posledním sloupci tedy po přijetí této hodnoty nebudou zakazovat porty spojující je se zásobovacími moduly až do té doby, dokud nenačtou správný počet nových hodnot. Průběh je stejný, vždy další nově přijatá hodnota vytlačí tu dříve přijatou hodnotu směrem doleva jen s tím rozdílem, že tato operace bude probíhat hned jedna za druhou, ne s každým krokem eliminace vždy jen jednou.

Po naplnění matice výpočetních jednotek novými daty budou do sběrného modulu z prvního sloupce matice odeslány hodnoty, ze kterých se bude počítat determinant (všechny ostatní hodnoty jsou odeslány hned po skončení všech eliminačních kroků, viz 2.6.1.3.3). Je důležité, aby byly hodnoty na výpočet determinantu odeslány až po přijetí nových hodnot, protože sběrný modul startuje výpočet další soustavy právě po přijetí hodnot sloužících k výpočtu determinantu (viz 2.6.1.3.7). Je tedy jasné, že pokud odstartuje výpočet po přijetí hodnot potřebných na výpočet determinantu, které budou odeslány až po přijetí hodnot z nové soustavy prvním sloupcem výpočetních jednotek, který přijme nová data nejpozději z celé matice, může výpočet bez problému začít, celá matice už bude naplněna novými daty a všechny potřebné výsledné hodnoty předchozí soustavy už budou z matice pryč.

2.5.2.2 Realizace

O správný posun modulu a hodnot následující soustavy se starají aritmetické jednotky, tedy `sc_module` třídy `Procesor`. Zásobovací modul se pouze stará o to, aby byly v kanálu správné hodnoty, a to pořád. Nastavuje se příznak `povolene_zasobovaci_porty`, podle které se řídí čtení ze vstupních portů spojujících poslední sloupec aritmetických jednotek se zásobovacími moduly. V aritmetické jednotce jsou místa pro uložení přijatých nových hodnot a modulu. Jsou to `stary_z`, `novy_z` a `novy_modul`. Po přijetí hodnoty typu *"z"* (viz 2.6.1.3.5) se kontrolují tato místa a podle jejich stavu se odesílá směrem doleva buď modul, nebo příslušná hodnota soustavy.

Nejdříve je přijat modul (v případě, že je nový modul jiný, než dosavadní). Nastaví se `n_m` na hodnotu `true`. Každá fáze (vytlačení hodnoty nebo modulu doleva) je zapříčiněna přijetím hodnoty typu *"z"*. Ta se uloží do `novy_z`. V první fázi je odeslán modul. Ale pouze v případě, že je `n_m` rovno `true`. Pak se `novy_z` uloží do `stary_z`. V každé další fázi (opakuje se tolikrát, dokud nebude přijato tolik hodnot typu *"z"*, aby byla matice naplněna novými hodnotami) pokud je `stary_z` obsazeno hodnotou, je odesláno směrem doleva. Opět je uloženo `novy_z` do `stary_z`. Hodnotou je `stary_z` naplněno vždy kromě druhé fáze (v případě, že se posílá i modul), nebo první fáze (modul je shodný, neposílá se). Jednoduše řečeno, doleva se posílá ta hodnota, která byla přijata v předchozím kroku a je poslána v momentě čtení hodnoty nově příchozí.

Výpočet další soustavy je ve třídě `Sberac` po přijetí všech hodnot (viz 2.5.4.2) zahájen voláním metody `zapsat(Paket)`, kdy se pošle jednotce v horním levém rohu matice hodnota typu *"r"*, která odstartuje výpočet další soustavy (viz 2.6.1.3.7 a realizace výpočetního procesu 2.6.2).

2.5.3 Zásobovací modul

2.5.3.1 Návrh

Zásobovací modul bude řízen hodinovým signálem. Celý chod bude zajišťován procesem, který bude kontrolovat v každém taktu vstupní port do komunikačního kanálu a v případě, že bude kanál volný, zapíše do něj příslušnou hodnotu. Touto hodnotou se rozumí buď modul, nebo přímo hodnota z další soustavy. Proces totiž bude kontrolovat, zda je modul, ve kterém počítá matice výpočetních jednotek aktuální soustavu shodný s tím, ve kterém se má řešit následující soustava. Pokud je modul rozdílný, dříve, než začne do kanálu postupně zapisovat hodnoty z příslušné řádky soustavy lineárních kongruencí, zapíše do kanálu hodnotu modulu, ve kterém se bude soustava řešit. Pokud je modul shodný, zapisují se vždy rovnou hodnoty ze soustavy a při výpočtu se pak použije modul dosavadní.

V případě, že je kanál volný, proces volá příslušnou metodu, která zapíše hodnotu a její typ do výstupního kanálu. Po provedení této operace se pokračuje v procesu. Proces, který kontroluje v každém taktu, zda je možné do kanálu zapsat, rozhoduje, jakou hodnotu tam zapsat a volá zapisovací metodu, neskončí nikdy. Běží ve smyčce a je vždy znovu aktivován hodinami.

2.5.3.2 Realizace

Třída `Zasobovac` je odvozena od `sc_module`. Hlavní proces modulu zásobovače je řízen stejným hodinovým signálem, jako všechny výpočetní jednotky. Proces `zasobovani()` (je typu `SC_THREAD`) nejdříve ze vstupního souboru pomocí metody `nacti()` načte všechna data, která přísluší výpočetním jednotkám

ležícím ve stejné řádce matice, jako zásobovací modul. Ukládají se do vectoru, stejně tak, jako všechny moduly řešených soustav.

Při běhu provádí kontrolu modulu, zda zapsat do komunikačního kanálu modul, nebo ne (podle toho, jaký byl zapsán naposledy, při předchozí soustavě) a popřípadě v jaký okamžik modul zapsat.

V případě, že je ve vectoru k dispozici ještě nějaká hodnota k „nasunutí“ do výpočetní matice a komunikační kanál spojující zásobovací modul s aritmetickou jednotkou je prázdný, zavolá proces `zasobovani()` metodu `zapisovani()`. Ta zapíše do kanálu přes výstupní port `sc_port<ZapisDoKanalu>` příslušnou hodnotu a její typ v objektu třídy `Paket`. Po zapsání paketu do kanálu se pokračuje v procesu `zasobovani()` při další náběžné hraně hodinového signálu. Metoda je opět volána až v případě dalšího zápisu.

2.5.4 Sběrný modul

2.5.4.1 Návrh

Tento modul bude řízen hodinovým signálem. Celý běh bude prováděn jedním procesem, který nikdy neskončí. V každém taktu bude kontrolovat postupně všechny vstupní porty a pokud bude v některém vstupním komunikačním kanálu zapsaná hodnota, přečte ji. Podle toho, o jaký typ hodnoty se jedná, ji uloží do příslušného vektoru. Musí nastřádat všechny hodnoty:

- modul,
- n částí řešení,
- n pořadí řešení,
- n pivotů (slouží k výpočtu determinantu),

kde n je počet řádků matice a tudíž počet eliminačních kroků Gauss-Jordan-Rutishauserovy metody.

Potom, pokud byl nějaký pivot roven 0, vypíše na obrazovku ukončení řešení této soustavy, protože 0 symbolizuje to, že další nenulový pivot nebyl nalezen, a tak matice aritmetických jednotek ukončila výpočet soustavy.

Pokud nebyl pivot nulový, znamená to, že výpočet doběhl až do konce a sběrný modul vykoná tyto činnosti:

- pomocí pořadí částí řešení, hodnot pivotů a modulu spočte hodnotu determinantu (vynásobením pivotů lze zjistit jeho hodnotu, protože hodnota determinantu trojúhelníkové soustavy je rovna součinu prvků na diagonále, tedy hodnot pivotů, jelikož ale nemuselo dojít k jejich postupnému výběru, je potřeba vyřešit znaménko determinantu, které je určeno pomocí pořadí jednotlivých částí řešení - využito vztahů v ukázce popisu implementace Gauss-Jordan-Rutishauserovy metody na řádce *5* a *8* (viz 1.1)),

- pomocí pořadí výsledků a hodnot výsledků seřadí výsledky do správného pořadí,
- pomocí výsledků, modulu a původních dat provede kontrolu řešení dosazením.

Na obrazovku potom vypíše, že bylo řešení nalezeno a toto řešení ve správně seřazené podobě, dále hodnotu determinantu a zda je řešení správné. Nakonec ještě odstartuje výpočet další soustavy (viz 2.6.1.3.7 a poslední odstavec 2.5.2.1).

2.5.4.2 Realizace

V modulu sběrače ve třídě `Sberac` se o běh stará jeden proces `sbirani()` typu `SC_THREAD`. Je to tedy proces, který je řízen hodinovým signálem. Vždy postupně zkontroluje všechny vstupní porty `sc_port<CteniZKanalu>` v každém taktu. Pokud není komunikační kanál prázdný, přečte `Paket`. Podle typu příchozí hodnoty uloží přečtenou hodnotu do příslušného vectoru.

V případě, že hodnota pivota (k výpočtu determinantu) je rovna nule, na obrazovku vypíše informaci o tom, že výpočet nebylo možné dokončit. V případě, že není žádný pivot nulový a všechny potřebné hodnoty už jsou přečtené, volá metodu `determinant()`, ve které se spočte hodnota determinantu a jeho znaménko. Potom se volá metoda `vypis()`, která přehází řešení do správného pořadí. Z této metody je potom ještě volána metoda `kontrolaReseni()`, kde se za pomoci původních hodnot dosadí vypočtené řešení do lineárních kongruencí a porovná se s pravou stranou, zda je vypočtené řešení správné. Nakonec jen vypíše vše na obrazovku a pomocí metody `zapsat(Paket)` volanou na výstupní port napojený kanálem na aritmetickou jednotku ležící v levém horním rohu matice zapíše hodnotu typu *"r"*, která odstartuje výpočet další soustavy (viz 2.6.1.3.7).

2.6 Výpočetní proces

2.6.1 Návrh

Celý výpočet funguje tak, že se najde pivot, udělá se jeho multiplikativní inverze a ta se přeposílá jednotkám v řádce až do posledního sloupce. Jednotky po jejím přijetí vynásobí hodnotu této inverze pivota s hodnotou v dané jednotce. Jednotky v prvním sloupci a ostatních řádcích dělají negaci hodnoty uložené v příslušné jednotce a tu distribuují v rámci řádky také až do posledního sloupce. Jednotky v řádce, kde byl zvolen pivot po vynásobení multiplikativní inverze pivota s hodnotou příslušné jednotky pošlou výsledek součinu ve sloupci nahoru a dolů až se hodnota rozdistribuuje do všech jednotek ve sloupci. Potom tuto hodnotu pošlou i směrem doleva, což je právě ten

posun navíc Gauss-Jordan-Rutishauserovy eliminační metody oproti Gauss-Jordanově metodě. Jednotky v řádcích, kde nebyl zvolen pivot přijmou seshora nebo zezdola tuto hodnotu a vynásobí ji negací hodnoty přijaté zleva a nakonec k ní ještě přičtou hodnotu příslušné jednotky. Potom výsledek také odešlou doleva. Toto je celý jeden eliminační krok, kdy se všechny hodnoty pro další eliminační krok posunuly doleva a vpravo vznikl prázdný sloupec.

Po přijetí hodnoty zprava ihned příslušná jednotka začne testovat, zda může být zvolena pivotem. Nečeká se, až včechny jednotky dokončí jeden eliminační krok. Díky tomu vznikají v matici výpočetních jednotek při výpočtu jakési „vlny“ jednotlivých eliminačních kroků. Díky tomu je výpočet rychlejší než v případě, že by se vždy čekalo na dokončení předchozích eliminačních kroků všemi jednotkami.

Dále v textu je při posílání hodnoty používáno slovo distribuovat nebo přeposílat. Těmito slovy je myšleno, že se hodnota ze zdrojové jednotky posílá vždy dál ve sloupci nebo řádce tak, že „probublává“ postupně až na konec matice. V případě, že jednotka obdrží hodnotu, ihned ji používá k výpočtu, nečeká se, až bude hodnota rozdistribuována např. v celé řádce.

Proces výpočtu se při spuštění simulace rozběhne a poběží celou dobu, dokud nebude výpočet dokončen. Proces je řízen hodinovým signálem shodným pro všechny aritmetické jednotky. Všechny operace se provádějí v jednotlivých taktech. Výpočet není řízen žádným jiným způsobem než tím, že si jednotlivé jednotky posílají komunikačními kanály data. Jelikož je tedy výpočet řízen daty, stěžejní je číst data z komunikačních kanálů a předávat je výpočetnímu procesu (viz 2.4.2), který podle typu jednotlivých dat provede specifické operace. Čtecí proces ukládá příchozí hodnoty i s jejich typem do čtecí paměti, odkud je výpočetní proces vybírá a zpracovává. Vždy v každém taktu vybere jednu hodnotu a podle jejího typu se rozhodne, co dělat. Hodnoty se vybírají vždy postupně, jejich zpracovávání je sekvenční. Teprve v případě, že jsou provedeny všechny příslušné operace patřící k právě vybrané a zpracovávané hodnotě, je ze čtecí paměti vybrána hodnota další. Výpočetní proces v případě, že nejsou žádná data ke zpracování, čeká.

Celý proces výpočtu se dá rozdělit do pomyslných tří částí podle toho, co se v každé části provádí. Jsou to:

- veškerá práce s pivotem,
- výběr hodnoty ke zpracování a nastavení k výpočtu potřebných parametrů,
- provádění samotných operací podle typu vybrané hodnoty.

První část se provádí vždy pouze na začátku nového eliminačního kroku. Druhá a třetí část se provádí v každém taktu pořád znovu a znovu, dokud jsou k dispozici hodnoty ke zpracování. Pokud už nejsou, čeká se, než zase budou, nebo než bude konec simulace.

2.6.1.1 Práce s pivotem

Tato část výpočetního procesu je prováděna pouze aritmetickými jednotkami ležícími v prvním sloupci matice. Hledání pivota se provádí sekvenčně odshora matice směrem dolů. Je tomu tak z toho důvodu, že je potřeba v každém eliminačním kroku vybrat takovou jednotku, která ještě pivotem zvolena nebyla a její hodnota je nenulová. Aby výpočet dospěl k úspěšnému konci, musí být tedy vybrány všechny jednotky ležící v prvním sloupci, každá z nich právě jednou. Nezáleží na pořadí, v jakém budou pivoti vybíráni, protože s řešením se potom ještě zaznamenává, v jakém pořadí byly jednotky jako pivoti zvoleny, aby se mohl vektor výsledků přeházet do správného pořadí. Je pouze nutné, aby se v každém eliminačním kroku Gauss-Jordan-Rutishauserovy metody vybrala právě jedna jednotka. Nicméně v případě, že se budou pivoti vybírat postupně seshora směrem dolů a v každém kroku budou podmínky pro zvolení pivotem splněny, nemusí se vektor řešení přehazovat, je seřazen správně. Proto je výhodné hledat pivota sekvenčně odshora dolů.

Vždy s nově načtenou soustavou lineárních kongruencí se tedy začíná hledat pivot od první jednotky, tedy jednotky v levém horním rohu výpočetní matice. S každým dalším krokem se ale hlednání pivota přesune dál dolů, čímž dojde k značnému urychlení oproti tomu, kdyby se hledání v každém kroku začínalo vždy od první řádky. Při testování jsou dvě možnosti - jednotka splňuje podmínky pro zvolení pivotem, nebo jednotka podmínky nesplňuje.

2.6.1.1.1 Jednotka splňující podmínky pro zvolení pivotem

Pokud jednotka podmínky splňuje, je zvolena pivotem. Jednotkám ležícím nad a pod ní ihned odešle signál (hodnota je rovna jedné) s tím, že pivot byl nalezen (aby mohly po přijetí tohoto signálu začít provádět příslušnou část výpočtu). Začne provádět inverzi pivota. Po provedení inverze odešle výslednou hodnotu směrem doprava, tedy do řádky (aby i tam mohly začít provádět příslušnou část výpočtu).

Potom je směrem dolů poslán signál, aby jednotka, která signál přijme, věděla, že všechny jednotky ve sloupci nad ní už pivotem zvoleny byly a je tedy v příštím kroku eliminace řada na jednotce, co signál přijme, tedy jednotka ležící v další řádce. Tento signál posílá pouze jednotka, která už ale signál seshora obdržela a v aktuálním eliminačním kroku je zvolena pivotem. Může se totiž stát, že by některá jednotka nemohla být v tomto kroku eliminace zvolena pivotem a tak bude zvolena jiná (viz 2.6.1.1.2). V dalším kroku eliminace je ale v tomto případě potřeba testovat zase tu jednotku, která v minulém kroku být zvolena nemohla, protože každá jednotka v prvním sloupci matice musí být zvolena právě jednou. Tento signál je vlastně takový „pešek“, protože může být v držení pouze jedné jednotky a říká jednotce, co jej má v držení, že právě ona testuje sebe, zda splňuje podmínky pro zvolení pivotem.

2.6.1.1.2 Jednotka nesplňující podmínky pro zvolení pivotem

Pokud jednotka podmínky nesplňuje, odešle směrem dolů signál (hodnota je rovna nule) s tím, že pivot nebyl nalezen. Jednotka po přijetí tohoto signálu začne testovat, zda splňuje podmínky. Pokud ano, tak 2.6.1.1.1, pokud ne, tak 2.6.1.1.2. Pokaždé se testuje, zda už nedorazila žádost o pivota do poslední řádky matice. To by totiž znamenalo, že v případě, že by ani tato jednotka nemohla být zvolena pivotem, žádná jednotka splňující podmínky neexistuje, a proto by nebylo možné tuto soustavu dořešit a výpočet by skončil.

Pokud nemůže být jednotka zvolena pivotem a má „peška“, tak 2.6.1.1.2 a „peška“ směrem dolů neposílá. Otestuje pivota v dalším kole a tak pořád dokola, dokud není zvolena pivotem. Potom „peška“ odešle. Jednotka, která „peška“ zezhora přijme, ale už pivotem zvolena byla, jej pouze pošle dál ve sloupci dolů.

2.6.1.2 Výběr hodnoty ke zpracování a nastavení parametrů k výpočtu

Jakmile jednotka v prvním sloupci přijme hodnotu zprava a pokud je to ta jednotka, co má v držení „peška“, ihned zahájí testování sebe, jestli může být pivotem do dalšího kola eliminace. Začne další eliminační krok. Nečeká se na všechny jednotky, než dokončí krok předchozí. Toto značně urychlí výpočet, nicméně je potřeba počítat s tím, že do některé jednotky už dorazí „vlna“ dalšího eliminačního kroku, ale hodnota v jednotce ještě nemusí být ta výsledná z předchozího kroku poslaná zprava, která je ale nutná k dalšímu správnému výpočtu.

Z tohoto důvodu je zde před prováděním samotných operací podle typu konkrétní hodnoty ta část výpočetního procesu, kde se vybírá hodnota ke zpracování a nastavují se potřebné parametry, aby výpočet probíhal se správnými hodnotami, nedošlo k jejich přepisování, uváznutí nebo počítání dalších nadbytečných kroků eliminace.

2.6.1.2.1 Aktuální a poslední eliminační krok

Nejdříve se ze čtecí paměti přečte příchozí hodnota a typ hodnoty (pokud je paměť prázdná, výpočetní proces čeká, dokud není čtecím procesem probuzen). Pak je v případě, že se jedná o náznak nového eliminačního kroku (podle typu hodnoty, směru příchodu a pokud ještě nebylo v tomto kroku navýšeno), navýšeno počítadlo říkající kolikátý krok je právě v této jednotce aktuální. Potom pokud je aktuální eliminační krok o dva větší, než poslední eliminační krok (ten se inkrementuje po přijetí hodnoty zprava), je jasné, že hodnota zprava předchozího eliminačního kroku ještě nebyla přijata, a tak by v případě pokračování nového eliminačního kroku došlo k výpočtu s nesprávnými hodnotami. Proto se tato hodnota uloží stranou, zakážou se všechny příchozí porty kromě toho zprava (aby mohla být přečtena pouze výsledná hodnota

z minulého eliminačního kroku) a proces výpočtu začne čekat na událost vyvolanou čtením.

Je jasné, že tato událost bude vyvolána právě v případě přečtení hodnoty nutné k dalšímu pokračování, tedy výsledné hodnoty přijaté zprava z minulého kroku eliminace. Po probuzení se provedou operace vyvolané touto přijatou hodnotou (uloží hodnotu do jednotky) a v dalším taktu, místo toho, aby výpočetní proces četl ze čtecí paměti, vykoná operace té hodnoty, kterou předtím uložil stranou. To je nyní už možné, může bez problému provádět další krok eliminace, protože má v jednotce uloženou správnou hodnotu. Tímto tedy došlo k takovému přednostnímu vykonávání určitých operací, což je nutné, aby výpočet probíhal se správnými hodnotami.

To, že se uchovává aktuální a poslední eliminační krok, má v celém výpočtu klíčovou vlastnost. Podle toho se vždy jednotky rozhodují co dělat, nebo nedělat. Správné nastavení těchto hodnot je důkladné a potřebné pro celý výpočet. Jednotky podle těchto hodnot poznají, do kterého eliminačního kroku patří příchozí hodnoty, a tak nemůže dojít k přepsání hodnot určitého typu novou hodnotou stejného typu, ale z dalšího kroku eliminace, když ještě bude stará hodnota potřeba. Tyto proměnné také zajišťují, aby jednotky věděly, v jaké fázi je řešení. Všechny jednotky musí totiž vědět, v jaké části výpočtu jsou, protože průběh není nijak synchronizován, nečeká se, a tak každá jednotka může řešit jiný eliminační krok. Jednotky podle těchto parametrů poznají, zda jsou aktivní, nebo ne (zda už hodnoty poslaly doleva a další eliminační krok se jich už netýká), aby nedošlo k výpočtu nesmyslných a nadbytečných eliminačních kroků.

Jelikož tyto parametry říkají, do kterého eliminačního kroku patří příchozí hodnoty, zabrání se uváznutí výpočtu v tom smyslu, že by byla hodnota přijata, přepsala by se pomocí ní hodnota stejného typu, ale minulého eliminačního kroku a až by došlo na použití hodnoty tohoto typu, použila by se ta nově přijatá. Nejen, že by výpočet byl dále chybný, ale v dalším eliminačním kroku by se čekalo na přijetí tohoto typu hodnoty, nicméně ta by už nedorazila, protože už byla příjata dříve.

2.6.1.2.2 Přednostní vykonávání operací

Čtení hodnot ze vstupních kanálů probíhá paralelně, to znamená, že může v jednom taktu přijmout jednotka čtyři hodnoty a uložit je do čtecí paměti. Zpracování těchto hodnot ale probíhá sekvenčně. Zase je zde potřeba upřednostnění těch hodnot, které přicházejí zprava. Pokud totiž přicházejí ve stejném taktu hodnoty zprava (výsledek minulého eliminačního kroku) a např. zleva (další eliminační krok), je lepší upřednostnit zpracování těch hodnot, které přicházejí zprava, protože ty jsou pro správný výpočet klíčové a bez nich není možné pokračovat dál. Aby tedy v případě čtení hodnot náležejících různým eliminačním krokům ve stejném taktu nemuselo dojít k ukládání hodnoty stranou, vyřešení jiné přijaté hodnoty a poté teprve té odložené stranou,

je výhodné přednostně vykonávat hodnoty přijaté zprava jako první. Podle tohoto pravidla se řídí výběr hodnoty výpočetním procesem ze čtecí paměti.

2.6.1.3 Provádění jednotlivých operací

Jednotlivými operacemi se rozumí provádění výpočetních a ostatních operací, nutných ke správnému chodu celé struktury aritmetických jednotek. Tyto operace jsou rozděleny podle hodnot, přesněji podle typu hodnot, které jsou z komunikačních kanálů přečtené a uložené do čtecí paměti, odkud jsou postupně výpočetním procesem zpracovávány. Celý chod je řízen daty, což znamená, že se pro každý typ jednotlivých hodnot provádějí specifické operace. Typ hodnoty je řešen pomocí 4 bitů, ve kterých je zakódováno, o jakou hodnotu se jedná. Pro snazší práci s daným typem hodnoty dojde vždy po přečtení z komunikačního kanálu k dekódování tohoto typu a ve výpočtu se dále používá jako typ každé příchozí hodnoty označení pomocí jednoho znaku. Před odesláním hodnoty se zase její typ zakóduje do 4 bitů. V následující tabulce 2.1 je rozdělení podle typu hodnoty s vysvětlením, co daný typ znamená.

Označení typu hodnoty	**Význam**
"p"	hledání pivota, nalezení pivota, inverze pivota
"n"	negace hodnoty uložené v jednotce
"w"	vypočtená hodnota, poslaná doleva do dalšího kroku eliminace
"i"	výsledná hodnota v řádce s pivotem (součin inverze pivota s hodnotou v jednotce)
"z"	nasouvání hodnot z další soustavy
"d"	žádost o data z další soustavy
"r"	zahájení řešení další soustavy
"e"	konec výpočtu, žádná další soustava není k dispozici
"k"	soustava nelze dořešit
"m"	modul z další soustavy
"s"	signál „pešek“ pro výběr pivota

Tabulka 2.1: Označení typu hodnoty a jeho význam ve výpočetním procesu

Prováděné výpočetní operace a operace potřebné ke správnému chodu jsou pro každý typ hodnoty specifické. Dále je popsáno, jakým způsobem výpočetní proces zpracovává, popř. jak upravuje nastavení parametrů aritmetické jednotky po přijetí hodnoty daného typu.

2.6.1.3.1 Hodnota s označením *"p"*

Rozlišuje se, zda je příchozí hodnota větší, než nula, nebo rovna nule. Pokud je rovna, znamená to, že byl přijat signál říkající, že jednotka umístěná v matici nad nemohla být vybrána pivotem, a tak se o to má pokusit příjemce. V takovém případě jen jednotka aktivuje příslušný signál, což znamená, že v dalším taktu začne testovat, zda může být pivot.

Pokud je nenulová, řeší se směr příchodu hodnoty. Pokud je směr příchodu seshora, nebo zezdola, znamená to, že pivot byl nalezen (příchozí hodnota je rovna jedné), jednotka má hodnotu přeposlat dál ve sloupci a začít provádět negaci své hodnoty. Po provedení negace pošle hodnotu negace směrem doprava. Pokud byl příchod hodnoty *"p"* zleva, znamená to, že je přijatá hodnota multiplikativní inverze pivota (v řádce, ve které byl v tomto eliminačním kroku vybrán pivot). Jednotka tuto hodnotu přepošle dál doprava, vynásobí ji s hodnotou uloženou v jednotce a výsledek pošle nahoru a dolů ve sloupci (označen typem *"i"*) a doleva (jako hodnotu označenou typem *"w"*).

2.6.1.3.2 Hodnota s označením *"n"*

Po přijetí této hodnoty zleva ji jednotka přepošle dál doprava. Potom, pokud už má jednotka seshora nebo zezdola (z řádky s pivotem) přijatou a uloženou hodnotu označenou *"i"* (multiplikativní inverze pivota vynásobená s hodnotou jednotky), tak vynásobí negaci právě s tou hodnotou *"i"* a přičte k výsledku ještě hodnotu uloženou v jednotce. Výsledek odešle doleva s označením *"w"*. Pokud nemá hodnotu *"i"* ještě přijatou, hodnotu *"n"* jen uloží. Výpočetní operace je shodná s tou, která je popsána v hodnotě s označením *"i"*, nicméně tento výpočet musí být na dvou místech, protože není zaručeno, která z hodnot dorazí dříve a která jako druhá. Právě až po přijetí druhé hodnoty se provede onen výpočet. V tomto případě (hodnota typu *"i"* už je uložena) by byla jako druhá přijata hodnota *"n"*, po jejímž příchodu by byla zahájena výpočetní operaci.

2.6.1.3.3 Hodnota s označením *"w"*

Hodnota přijatá zprava se uloží do aritmetické jednotky jako příslušná hodnota pro následující eliminační krok a navýší se počítadlo posledního eliminačního kroku. Je povolena pivotizace, tudíž další eliminační krok. Hned v dalším taktu začne jednotka, která má v držení „peška“ testovat pivota a tím začíná další eliminační krok. Pokud už to byl poslední eliminační krok, je do sběrného modulu odeslána hodnota modulu, ve kterém byla soustava řešena, část řešení náležící příslušné jednotce a pořadí této části řešení v celém vektoru řešení. Doprava je odeslána žádost o nová data (označení *"d"*), resp. o „nasunutí“ zbytku nových dat příslušné řádky do struktury výpočetních jednotek.

2.6.1.3.4 Hodnota s označením *"i"*

Po přijetí multiplikativní inverze pivota vynásobené s hodnotou v příslušné jednotce seshora, nebo zezdola (z řádky s pivotem) dojde k přeposlání dál ve sloupci. Potom, pokud už má jednotka zleva přijatou a uloženou hodnotu označenou *"n"* (negace hodnoty v prvním sloupci matice), vynásobí negaci právě s tou hodnotou *"i"* a přičte k výsledku ještě hodnotu uloženou v jednotce. Výsledek odešle doleva s označením *"w"*. Pokud ještě hodnotu typu *"n"* uloženou nemá, hodnotu typu *"i"* jen uloží. Výpočetní operace je shodná s operací popsanou v hodnotě s označením *"n"*, nicméně v tomto případě (hodnota typu *"n"* už je uložena) by byla zahájena po přijetí *"i"*.

2.6.1.3.5 Hodnota s označením *"z"*

Po přijetí hodnoty označené *"z"*, což je hodnota, která přichází zprava (hodnota, která se „nasouvá“ ze zásobovacích portů směrem doleva až do prvního sloupce vždy postupně), dojde k povolení, nebo zakázání vstupních portů vedoucích ze zásobovacích modulů. Je tomu tak z toho důvodu, že ještě může probíhat výpočet a v takovém případě se hodnoty nasouvají postupně, nebo už výpočet skončil, a tak je potřeba nasunout všechny zbývající hodnoty z další soustavy (viz 2.5.2), aby byla novou hodnotou naplněna každá výpočetní jednotka.

Vždy před posunem hodnoty z nové soustavy v matici aritmetických jednotek o sloupec doleva se posílá nejdříve modul, ve kterém se následující soustava bude řešit. Je tomu tak pouze v případě, že je modul jiný, než modul, ve kterém se řeší soustava aktuální. S každou další přijatou hodnotou *"z"* se posune stará hodnota o sloupec doleva. Nakonec až dorazí *"z"* do prvního sloupce matice výpočetních jednotek (je jasné, že všechny jednotky v ostatních sloupcích už správnou hodnotu mají), pošlou tyto jednotky do sběrného modulu v případě, že bylo možné soustavu dořešit hodnoty pivotů (z pivotů se ve sběrném modulu určuje determinant (viz 2.5.4)), nebo v případě, že nebylo možné soustavu dořešit, odešle jednotka v první řádce hodnotu rovnou nule, aby sběrný modul věděl, že je neúspěšný konec řešení soustavy. Všechny jednotky potom vyresetují svoje nastavení. To znamená, že všechny parametry vrátí do takového stavu, v jakém byly před zahájením řešení soustavy. Je to z toho důvodu, že pak už mohou řešit soustavu další.

2.6.1.3.6 Hodnota s označením *"d"*

Tento typ hodnoty slouží jako žádost o další data z následující soustavy. Znamená to, že v průběhu výpočtu se zleva přijaté hodnoty v jednotkách, které už jsou neaktivní, převede na hodnotu typu *"d"*, která je stěžejní pro jednotky v posledním sloupci výpočetní matice, kam se postupně hodnota typu *"d"* přeposílá. V případě přijetí této hodnoty přečtou tyto jednotky ze vstupních kanálů vedoucích ze zásobovacích modulů hodnotu. Uloží se příznak,

zda ještě probíhá výpočet (hodnota typu *"d"* je větší než nula pokud neprobíhá), aby jednotka v posledním sloupci matice po přijetí hodnoty typu *"z"* věděla, zda nechat povolené, nebo zakázat porty vedoucí ze zásobovacích modulů (viz 2.6.1.3.5). V případě, že již jednotka v posledním sloupci obdržela hodnotu typu *"e"* (značí konec výpočtu, žádná další soustava k řešení není k dispozi), odešle doleva tuto hodnotu, aby jednotky věděly, že výpočet končí (viz 2.6.1.3.8). Pokud jednotka obdrží hodnotu typu *"d"* a má již zprava přijatou ukončovací hodnotu typu *"e"*, hodnotu typu *"d"* již doprava nedistribuuje, ale doleva pošle ukončovací hodnotu typu *"e"*. Tímto se urychlí ukončení simulace, protože se ukončovací hodnota distribuuje postupně doleva už během výpočtu. Pokud by čekala ukončovací hodnota v posledním sloupci matice až do konce výpočtu, musela by se potom přes celou matici distribuovat najednou až do prvního sloupce a všechny jednotky by čekaly, než tam dorazí.

2.6.1.3.7 Hodnota s označením *"r"*

Hodnota označená *"r"* je odesílána sběrným modulem a příjemce je vždy jen výpočetní jednotka v levém horním rohu matice. Tato hodnota po přijetí totiž jednotce říká, že má začít řešit další soustavu. Po tom, co jednotky v levém sloupci matice přijaly hodnoty typu *"z"*, odeslaly do sběrného modulu hodnoty pivotů. Sběrný modul po obdržení všech hodnot provede výpočet determinantu a kontrolu řešení (viz 2.5.4) a do první jednotky v matici odesílá právě hodnotu typu *"r"*. V té chvíli je totiž jasné, že jsou již všechny jednotky připravené (s vyresetovanými parametry) k řešení další soustavy. Je to proto, že jednotky v prvním sloupci obdrží data z nové soustavy nejdéle ze všech jednotek v matici. Teprve po jejich obdržení odesílají hodnoty pivotů a sběrný modul teprve po obdržení všech pivotů (nemusí přijít ve stejný takt, protože jednotlivé řádky nejsou synchronní) pošle hodnotu typu *"r"*. Jednotka v levém sloupci se po obdržení této hodnoty vyresetuje do původního nastavení a povolí pivotizaci, což vede k zahájení výpočtu soustavy.

2.6.1.3.8 Hodnota s označením *"e"*

O konci výpočtu rozhodují zásobovací moduly. Ty totiž v případě, že není již žádná další soustava k řešení k dispozici, uloží do kanálů vedoucích do posledního sloupce výpočetní matice hodnotu typu *"e"*. Tyto jednotky v případě obdržení žádosti o další data pošlou směrem doleva hodnotu tohoto typu, tedy hodnotu říkající, že právě řešená soustava je poslední. Další jednotky po přijetí hodnoty tohoto typu jen přepošlou hodnotu směrem doleva, pokud byl výpočet již dokončen. Pokud ne, distribuce této ukončovací hodnoty se řídí podle hodnoty typu *"d"* (viz 2.6.1.3.6). V případě, že už hodnota dorazila do prvního sloupce matice aritmetických jednotek, je nutné provést to samé, co se provedlo v případě doručení hodnoty typu *"z"* do prvního sloupce. V případě, že bylo možné soustavu vyřešit, odešlou se do sběrného modulu hodnoty

pivotů (k výpočtu determinantu) a v případě, že nebylo možné soustavu vyřešit, odešle jednotka v první řádce matice do sběrného modulu nulu. Místo zahájení výpočtu další soustavy se potom ukončí simulace.

2.6.1.3.9 Hodnota s označením *"k"*

V průběhu výpočtu se může stát, že není možné vybrat žádného dalšího pivota, což znamená, že taková soustava nelze dořešit. Nicméně může být v pořadí za touto soustavou ještě nějaká další, takže s neúspěšnou jednou soustavou nesmí skončit výpočet úplně, ale pouze výpočet oné jedné soustavy. Jednotky po přijetí hodnoty tohoto typu zjistí, že soustavu nebylo možné vyřešit. Hodnota s označením *"k"* vznikne v prvním sloupci, v poslední řádce, pokud k této jednotce došla hodnota *"p"* rovná nule (žádost o hledání pivota na této jednotce) a tato jednotka už pivotem zvolena byla, nebo je její hodnota nulová. V takovém případě neexistuje žádný další pivot a soustavu nelze řešit. Jednotka po přijetí hodnoty typu *"k"* přepošle tuto hodnotu směrem nahoru, aby celý první sloupec věděl, že je konec řešení této soustavy. Doprava odešle vždy jednotka žádost o nová data (označení *"d"*), resp. o „nasunutí" zbytku nových dat příslušné řádky do struktury výpočetních jednotek.

2.6.1.3.10 Hodnota s označením *"m"*

Hodnota nového pivota posílaná zásobovacím modulem a postupně distribuovaná až do prvního sloupce matice výpočetních jednotek před samotnými hodnotami soustavy. Po přijetí této hodnoty se uloží do k tomu určeného místa a pro poslední sloupec se v případě, že ještě probíhá výpočet (postupné nasouvání hodnot) zakážou vstupní porty ze zásobovacích modulů (viz 2.5.2).

2.6.1.3.11 Hodnota s označením *"s"*

Signál „pešek" má v držení vždy jen jedna jednotka a slouží pro zrychlení rozhodování o tom, kdo má být dalším pivotem (viz 2.6.1.1.1). V případě, že jednotka tento signál přijme, ale už byla pivotem, jej pouze přepošle směrem dolů. V případě, že pivotem ještě nebyla, nastaví konkrétní příznak, aby věděla, že po přijetí *"w"* bude testovat sebe, zda splňuje podmínky pro to, být zvolena pivotem.

2.6.2 Realizace

Ve třídě `Procesor` zajišťuje výpočet proces `vypocet()`. Jedná se o proces typu `SC_THREAD`, tedy takový proces, který nikdy neskončí a je aktivován náběžnou hranou hodinového signálu. Tři části výpočetního procesu se provádějí pořád dokola, resp. první část (práce s pivotem) se provádí pouze na jednotkách ležících v prvním sloupci a to pouze na začátku každého eliminačního kroku (řízeno pomocí příznaku `st`). Zbylé dvě části (výběr hodnoty ke zpracování

a nastavení parametrů a provádění operací specifických pro daný typ hodnoty) se provádí na všech aritmetických jednotkách podle hodinového signálu a úplně shodně, bez rozdílu, kde v matici se jednotka nachází.

Signály `najdi_pivota_s` a `najdi_pivota_jinde_s` slouží k řízení nalezení pivota, přičemž první signál je pro zahájení výpočtu, druhý signál si jednotka nastaví vždy sama pro sebe, když je seshora požádána o testování pivota. Při hledání pouze jednotka testuje, zda je hodnota větší než nula a zda je příznak `byl_pivot` roven `false`. Multiplikativní inverze pivota se provádí v metodě `inverze()`.

Pokud potřebuje výpočetní proces odeslat nějakou hodnotu, zavolá metodu `odeslat()`. Jejími parametry jsou signál `sc_signal<sc_bv<4>>`, hodnota a typ hodnoty. Datovým typem tohoto signálu jsou 4 bity. Každý bit znamená jeden směr výstupu z výpočetní jednotky - řazení je „doleva, doprava, nahoru, dolů“ (viz zápis do kanálu 2.4.1.2).

K ukládání hodnot procesem čtení je použita fronta `prichozi_hodnoty` se dvěma konci typu `deque` (na začátek se vkládají hodnoty přijaté zprava a nakonec všechny ostatní - jelikož se vybírají hodnoty ze začátku, dojde k přednostnímu zpracování operací, viz 2.6.1.2.2). Pokud je fronta prázdná, čeká výpočetní proces na událost vyvolanou čtením `wait(udalost_cteni)`. Z této fronty se pak čtou hodnoty, přičemž se vždy manipuluje (zamyká a odemyká) s `sc_mutex`, protože do fronty přistupuje i proces čtení, takže se jedná o kritickou sekci. Potom se u hodnoty vybrané z fronty nastaví `aktualni_el_krok` (viz 2.6.1.2.1) podle směru příchodu, typu hodnoty, samotné hodnoty a podle toho, jestli už byl eliminační krok navýšen. Dále se podle toho rozhodne, zda je možné pokračovat ve vykonávání z fronty vybrané hodnoty, nebo je potřeba ji uložit stranou (`temp`) a čekat na příchod hodnoty zprava `wait(udalost_cteni)`.

V poslední části výpočetního procesu je `switch`, ve kterém se rozhoduje podle `typ_na_vstupu`, což je typ hodnoty (dekódovaný ze čtyř bitů `sc_bv<4>` do typu `char`) a podle něj se budou provádět příslušné operace. Jednotlivé operace jsou rozděleny do metod, kde každá metoda je pro jeden typ hodnoty a jedná se o metody `switchX()`, kde je místo X označení typu hodnoty (viz tabulka 2.1). V těchto metodách se provádí samotný výpočet a nastavování parametrů aritmetické jednotky zajišťující správnou funkčnost. Metoda `negace()` slouží ke zjištění negace hodnoty uložené v jednotce.

K uvedení výpočetní jednotky do původního stavu se využívá metoda `resetujHodnoty()`, která je z každé jednotky volána po načtení dat následující soustavy (před zahájením jejího řešení). Po přijetí hodnoty typu *”r”* (viz 2.6.1.3.7) jednotkou ležící v levém horním rohu je testováno, zda je již přijata hodnota typu *”e”* (viz 2.6.1.3.8) a pokud ano, je zavolána funkce `sc_stop()`, která ukončí běh všech procesů a tím končí simulace.

2.7 Kompletní chování

2.7.1 Návrh

Celá simulace, resp. simulační fáze začne po elaborační fázi. Pro úspěšné spuštění je potřeba zadat vstupní parametry (viz uživatelská příručka B):

- jméno souboru, ve kterém jsou data (soustavy lineárních kongruencí) k vyřešení,
- jméno souboru, do kterého se zaznamená běh celé simulace (tento soubor se použije při vizualizaci, viz 2.8),
- doba trvání jednotlivých operací při výpočtu (multiplikativní inverze, negace, násobení a sčítání).

V elaborační fázi se nejdříve zkontrolují vstupní parametry. Potom se zkontroluje soubor, který obsahuje soustavy k řešení, přesněji zkontroluje se celý obsah tohoto souboru. Při kontrole se určí parametry struktury aritmetických jednotek, která se musí vytvořit a počet soustav ve vstupním souboru. Podle těchto zjištěných parametrů (rozměr soustav lineárních kongruencí) vytvoří aritmetické jednotky (viz 2.1) strukturu ve tvaru matice (viz 2.3) odpovídající velikosti řešené soustavy s pravými stranami. V každé aritmetické jednotce je při vytváření uložena jedna hodnota z řešené soustavy (tato hodnota odpovídá umístění jednotky ve výpočetní matici). První soustava se tedy nemusí nijak nasouvat, je načtena rovnou při vytvoření struktury. Společně s vytvářením matice aritmetických jednotek se vytváří i sběrný (viz 2.5.4) a zásobovací moduly (viz 2.5.3). Dále se jednotky propojí komunikačními kanály (viz 2.2 a poslední odstavec 2.3.1). Na vstup jednotky ležící v prvním sloupci a první řádce je napojen signál, který odstartuje célý výpočet, pokud do něj bude přiřazena hodnota. Tím je celá struktura vytvořena, propojena, všechny moduly čekají na spuštění simulace.

Potom je konec elaborační fáze a začíná fáze simulační. Při simulační fázi se rozeběhnou všechny procesy, které jsou řízeny hodinovým signálem. Jsou to tedy výpočetní procesy všech aritmetických jednotek, čtecí procesy všech aritmetických jednotek, proces ve sběrném modulu a procesy v zásobovacích modulech. Všechny běží simultánně a vždy ve stejný čas (náběžná hrana hodinového signálu - existují pouze jedny hodiny pro všechny moduly) jsou opět reaktivovány. Podle toho, jaký typ dat jednotky zpracovávají, provádějí operace. Jakmile je zahájena simulace, není nic, co by jednotky řídilo. Celý chod je asynchronní a naprosto závislý na datech a na hodinovém signálu. Jednotky se chovají tak, jak je popsáno ve výpočetním procesu (viz 2.6), komunikují spolu čtením (viz 2.4.2) a zapisováním (viz 2.4.1) do komunikačních kanálů (viz 2.2). Při výpočtu využívají jednotky jednotlivých typů hodnot (viz 2.6.1.3), aby probíhal výpočet správně a se správnými daty. Pro věrné simulování hardwaru je využito proměnné trvání jednotlivých operací během výpočtu. Trvání

jednnotlivých operací (multiplikativní inverze, negace, násobení, sčítání) se řídí parametry, které byly na začátku zadány. V průběhu výpočtu je také nasouvána ze zásobovacích modulů směrem doleva do maticové struktury ze vstupního souboru následující soustava i s modulem (viz 2.5.2).

Pokud je matice vyřešena jsou odeslány příslušné hodnoty do sběrného modulu (viz 2.6.1.3.3 a 2.5.4), který tyto hodnoty zpracuje. Mezitím přejdou jednotky do počátečního nastavení, jen s jinými uloženými hodnotami a modulem (náležejícím další soustavě). Sběrný modul poté odstartuje výpočet další soustavy (viz 2.6.1.3.7 a poslední odstavec 2.5.2.1). Tento postup probíhá tak dlouho, dokud zásobovací moduly posílají do matice aritmetických jednotek hodnoty, tedy tak dlouho, dokud nebudou spočteny všechny soustavy ze vstupního souboru. Po zaslání posledních hodnot zásobovacími moduly do matice je ještě poslána ukončovací hodnota (viz 2.6.1.3.8). Ta projde do prvního sloupce matice a až sběrný modul odstartuje řešení další soustavy, aritmetická jednotka v levém horním rohu matice v závislosti na ukončovací hodnotě zastaví simulaci (viz poslední odstavec 2.6.2).

2.7.2 Realizace

Elaborační fáze začíná ve funkci `sc_main()`. Argumenty `argv[]` jsou zkontrolovány pomocí funkce `kontrolaArgumentu()`. V případě chyby je na obrazovku vypsáno, z jakého důvodu došlo k chybě (který parametr je zadán nesprávně) a ukázka správného použití vstupních parametrů. V případě, že se chyba neobjeví, je vstupní soubor a jeho obsah zkontrolován metodou `zkontrolujAUrciRozmeryVstupu()`. Pomocí této metody je zjištěn rozměr soustav, které se budou řešit a je také zkontrolováno, zda je celý obsah korektní. Pomocí zjištěných parametrů se potom vytvoří objekt třídy `Matice` (jedním z parametrů je i objekt třídy `Takty`, jehož položky udávají trvání jednotlivých výpočetních operací). V konstruktoru se vytvoří celá struktura - moduly výpočetních jednotek - třída `Procesor`, sběrný modul - třída `Sberac` a zásobovací moduly - třída `Zasobovac` (viz 2.3.2 a 2.5.1.2).

Na objekt třídy `Matice` jsou postupně volány dvě metody pro vytvoření komunikačními kanály a propojení všech komunikačních portů ve všech modulech. Jsou to `vytvorKomunikacniKanaly()` a `napojKomunikacniKanaly()`. Na vstup jednotky ležící v levém horním rohu je připojen signál metodou `napojSignal()`, který odstartuje výpočet v případě zapsání hodnoty. To se stane po zavolání metody `najdiPivota()`.

Končí elaborační fáze, začíná simulační. Je odstartována voláním metody `sc_start()`. Všechny procesy typu `SC_THREAD` se s pozitivní hranou hodinového signálu `Clock.pos()` rozběhnou a začnou provádět příslušné operace. Nejdříve všechny výpočetní jednotky čekají. Kromě první jednotky, která čte signál `najdiPivota()` startující výpočet. Zahajuje hledání pivota (viz druhý odstavec 2.6.2). Mezitím zásobovací moduly zapisují do kanálů ústících do posledního sloupce výpočetní matice hodnoty z další soustavy ve vstupním sou-

boru. Celý výpočet probíhá po jednotlivých eliminačních krocích, proces výpočtu (viz 2.6.2) je řízen typem do jednotek příchozích hodnot. Postupně se nasouvají hodnoty z nové matice (viz 2.5.2.2). Po skončení řešení je vše odesláno do sběrného modulu, ten po zpracování hodnot startuje řešení další soustavy (viz 2.5.4.2). Takto se řeší opakovaně každá další soustava. Jednotka v levém horním rohu po vyřešení poslední soustavy zavolá metodu `sc_stop()`, čímž se ukončí simulační fáze a běh programu se vrátí zpět do funkce `sc_main()`.

2.7.2.1 Použitý software a knihovny

- Zdrojové kódy jsou napsané v C++, jako vývojové prostředí bylo použito Microsoft Visual Studio 2008 Professional Edition [32] - verze šířená v rámci MSDNAA [22] (Microsoft Developer Network Academic Alliance). Přeloženo překladačem ve vývojovém prostředí.

- Knihovna SystemC ve verzi 2.2.0 dostupná na webu [27] jen s jednou úpravou - byla zakomentována jedna řádka ve zdrojovém kódu, která vypisovala při ukončení simulace informaci o tom, že simulace byla ukončena. Licence je SystemC Open Source License [23].

- Propojení vývojového prostředí se SystemC a nastavení projektu (viz [3]).

2.8 Vizualizace

Simulační aplikace vytvoří soubor, ve kterém bude uložen běh celé simulace (jednotlivých modulů), který potom vizualizační aplikace otevře a bude zobrazovat příslušným způsobem operace z tohoto souboru. Všechna logika bude přitom na simulační aplikaci, vizualizační aplikace bude sloužit jako nástroj pro grafickou reprezentaci, takže, co nebude uloženo v souboru, to nebude graficky zobrazeno.

Je potřeba zaznamenat běh všech aritmetických jednotek, respektive všech operací, které jednotky provádějí. Zobrazení ve vizualizační aplikaci bude rozděleno podle jednotlivých taktů simulace, to znamená, že musí být data v souboru seřazena podle času, kdy jednotky operace prováděly, aby se při jednom konkrétním taktu zobrazily všechny operace, které v simulaci při tomto taktu všechny jednotky provedly. U každého taktu tedy bude uvedeno, která jednotka operaci prováděla a o jakou operaci se jednalo. Musí být stanoveno, kdy operace začala a kdy skončila. S tím souvisí ještě další informace, jako např. jakou hodnotu má jednotka uloženou, jakou vypočítala, jaká hodnota je přečtena a zapisována, jakým směrem z jednotky je hodnota posílána a čtena a o jaký typ hodnoty se jedná.

Do souboru, který vizualizační aplikace bude zpracovávat, se ukládají pouze změněné stavy aritmetických jednotek, které se pak překreslují (je zbytečné

zaznamenávat vždy stav celé matice aritmetických jednotek, protože by se velikost souboru značně zvětšila, jeho parsování by bylo časově náročnější a navíc by spousta informací v souboru říkala, že se s danou jednotkou nic neděje).

Vizualizační aplikace zobrazí strukturu aritmetických jednotek propojených komunikačními kanály a pomocí čísel bude reprezentovat hodnoty, které jsou během výpočtu stěžejní a pomocí barev bude reprezentovat typ těchto hodnot. Barevně také bude odlišen stav jednotlivých aritmetických jednotek a výpočetní operace, které jednotky provádějí.

2.8.1 Vytvoření dat pro vizualizaci

2.8.1.1 Návrh

Je potřeba nasbírat všechny prováděné operace všech aritmetických jednotek a uložit do souboru tak, aby byly seřazeny podle taktu, ve kterém se prováděly. Během řešení soustavy není možné soubor pro vizualizační aplikaci tvořit, protože neexistuje žádné sdílené místo, kam by všechny jednotky mohly přistupovat. Dalším důvodem je fakt, že procesy běží synchronně a tak, i kdyby existovalo sdílené úložné místo, byla by to kritická sekce. Musela by tedy být příslušným způsobem spravována. To by ale znamenalo, že by pouze jedna jednotka mohla do tohoto místa přistupovat a ostatní by čekaly, až sekci uvolní. Takže by musely do sekce zapisovat sekvenčně a kvůli čekání na tuto sekci by celý paralelní výpočet postrádal smysl - čekání na zápis do sdílené paměti by zdržovalo celý výpočet natolik, že by se výpočet řídil pouze přístupem do kritické sekce.

Každá jednotka si z tohoto důvodu bude během výpočtu ukládat všechny prováděné operace i s příslušnými náležitostmi pouze do lokální paměti. Po skončení simulace budou paměťi zaplněné informacemi o běhu jednotek sloučeny ze všech jednotek dohromady. To znamená, že se vizualizační soubor sestaví až po proběhnutí celé simulační fáze (vyřešení všech soustav, které byly k dispozici ve vstupním souboru). Při vytváření souboru se bude paměť jednotlivých jednotek zpracovávat tak, aby byly operace a jejich náležitosti uloženy seřazené podle jednotlivých taktů hodinového signálu.

Při vytváření souboru se nejdříve ze všech aritmetických jednotek získá záznam o jejich prováděných operacích. V souboru je potřeba také uchovat všechny moduly, ve kterých byly jednotlivé soustavy řešeny, velikost maticové struktury a hodnoty načtené v jednotlivých aritmetických jednotkách před zahájením výpočtu (tedy hodnoty náležící první soustavě lineárních kongruencí - další hodnoty se nasouvají vždy postupně, což je zaznamenáno jako příslušné operace v běhu jednotlivých jednotek). Všechny záznamy se cyklicky prochází a sestavuje se soubor. Postupuje se tak, že se ke každému taktu vždy najdou v záznamu z každé aritmetické jednotky všechny operace náležící příslušnému taktu a zapíší se do souboru. Ten se naplněný strukturou už jen uloží.

Struktura souboru, který vizualizační aplikace otevře a postupně jej bude

zpracovávat a vykreslovat podle něj příslušným způsobem na obrazovku barevné a číselné informace, má strukturu XML (Extensible Markup Language - viz 2.8.1.2.1).

2.8.1.2 Realizace

Při vytváření maticové struktury v konstruktoru třídy `Matice` v elaborační fázi se vytvoří objekt třídy `Vizualizace`. Tento objekt bude po skončení simulační fáze shromažďovat běh všech aritmetických jednotek a vytvářet správnou strukturu souboru pro vizualizační aplikaci. Běh, resp. jednotlivé operace, se ukládají v každé aritmetické jdnotce do mapy `map<int, vector<Operace>>`, kde celočíselný klíč značí takt, kdy se operace prováděla a ve vectoru typu `Operace` jsou zaznamenány všechny operace prováděné v tomto taktu.

Po skončení simulace se volá metoda `ziskejBehSimulace()`, která v objektu třídy `Vizualizace` vytvořeného v elaborační fázi uloží do jednoho vectoru všechny mapy s operacemi všech aritmetických jednotek. Po zavolání metody `behSimulaceToXml()` se na objekt třídy `Vizualizace` zavolá metoda `toXml()`, která vytvoří příslušnou strukturu a uloží ji do souboru. Postupuje při tom tak, že cyklicky prochází vector map a v každé mapě hledá vždy jeden konkrétní takt (podle tohoto taktu jsou položky v mapě seřazeny). Pokud tento takt existuje (jednotka v tomto taktu provedla nějaké operace, které se budou graficky zobrazovat), je z vectoru u příslušného taktu z mapy postupně každý objekt typu `Operace` ukládán do výstupního souboru (přetížený operátor `<<`). Až se tímto způsobem zpracuje poslední takt, je v souboru vytvořena struktura, kterou bude vizualizační aplikace zpracovávat.

2.8.1.2.1 Ukázka struktury XML souboru

```
<vizualizace>
  <info>
    <modul>7</modul>
    <modul>13</modul>
    <radku>4</radku>
    <hodn>
      <au c="1">3</au>
      <au c="2">6</au>
      <au c="3">2</au>
      ...
      <au c="20">4</au>
    </hodn>
  </info>
  <simulace>
    <takt t="0">
      <au c="1">
```

```
        <akce>i</akce>
      </au>
      <au c="1">
        <akce>z</akce><smer>3</smer><hodn>1</hodn><druh>p</druh>
      </au>
    </takt>
    <takt t="20">
      <au c="6">
        <akce>c</akce><smer>2</smer><hodn>1</hodn><druh>p</druh>
      </au>
      <au c="6">
        <akce>n</akce>
      </au>
      <au c="6">
        <akce>z</akce><smer>3</smer><hodn>1</hodn><druh>p</druh>
      </au>
      ...
    </takt>
    <takt t="40">
      ...
    </takt>
    ...
    <takt t="240">
      ...
      <au c="7">
        <akce>c</akce><smer>2</smer><hodn>4</hodn><druh>n</druh>
      </au>
      ...
      <au c="19">
        <akce>z</akce><smer>0</smer><hodn>2</hodn><druh>w</druh>
      </au>
    </takt>
    ...
    <takt t="1380">
      ...
    </takt>
  </simulace>
</vizualizace>
```

Na začátku souboru jsou vždy uvedeny potřebné informace `<info>`. Jedná se o výpis všech modulů `<modul>`, které byly použity při řešení (jeden modul pro jednu soustavu). Počet řádků `<radku>` soustavy (sloupců je o jeden více než řádků). A hodnoty `<hodn>` všech jednotek při vytvoření struktury (je to do jednotek načtená první soustava ze vstupního souboru). Jednotky jsou

značeny jako `<au c="X">`, kde v atributu `c` je číslo jednotky (`X`). Jednotky jsou číslovány od jedničky směrem zleva doprava po řádcích.

Potom je v souboru uložen záznam samotné simulace `<simulace>`. Je rozdělen podle jednotlivých taktů `<takt t="X">`, kde v atributu `t` je doba od spuštění simulace (`X`). V jednotlivých taktech jsou vždy všechny jednotky, které v tomto taktu provádějí nějaké operace, které je nutné graficky reprezentovat. Označeny jsou opět `<au c="X">`. Operace, které se zobrazují jsou označené jako `<akce>`. Parametrem je jeden znak, který určuje, jaká operace se provádí (např. v příkladu výše značí `i` začátek provádění multiplikativní inverze, `n` značí začátek provádění negace, `z` znamená zápis hodnoty do komunikačního kanálu a `c` značí čtení z kanálu). U zápisu a čtení je potřeba ještě uvést další informace. Je to směr `<smer>`, kterým jednotka zapisuje (odkud jednotka čte), hodnota `<hodn>` a typ hodnoty `<druh>`, kterou jednotka zapisuje (čte). Směry mohou být čtyři, jsou označeny čísly a znamenají: `0` - doleva/zleva, `1` - doprava/zprava, `2` - nahoru/seshora a `3` - dolů/zezdola. Typ hodnoty je shodný s typem, podle kterého se řídí výpočetní proces (viz 2.6.1.3).

2.8.2 Vizualizační aplikace

Aplikace, která bude otevírat soubor s XML strukturou a zpracovávat ji, bude mít tuto funkčnost:

- zobrazit celou strukturu aritmetických jednotek a propojovacích komunikačních kanálů podle rozměru řešené soustavy,
- ve struktuře zobrazovat čísly hodnoty, barevně pak jednotlivé výpočetní operace, stav aritmetických jednotek a typ hodnoty,
- zobrazit legendu, co jaká barva znamená, modul, ve kterém se počítá a dobu od zahájení výpočtu,
- spustit a pozastavit běh vizualizace,
- krokovat vpřed a krokovat vzad po jednotlivých taktech běh vizualizace,
- nastavit dobu trvání jednoho taktu pro vizualizaci (jak rychle se mají při běhu vizualizace střídat takty),
- spustit simulaci s příslušnými parametry,
- nastavit parametry pro simulační aplikaci a trvání jednotlivých výpočetních operací pro simulaci,
- vytvořit nový soubor pro simulační aplikaci a podle parametrů jej naplnit daty.

2.8.2.1 Návrh

2.8.2.1.1 Rozmístění prvků

Aplikace bude mít v hlavním okně po pravé straně zobrazenou legendu. Ta se vytvoří při spuštění aplikace. V legendě je zobrazeno, co znamená jaká barva pro všechny prováděné výpočetní operace, jaký typ hodnot se zapisuje a čte do/z komunikačních kanálů a jakých stavů mohou nabývat aritmetické jednotky. V horejší části okna bude pruh tlačítek, která budou ovládat celý běh vizualizace a veškeré nastavení. V tomto místě je také zobrazen modul, ve kterém se soustava řeší a čas běhu. Největší část okna zabere samotná vykreslená struktura aritmetických jednotek, která bude při spuštění aplikace prázdná.

2.8.2.1.2 Zobrazení struktury

Po otevření souboru vytvořeného simulační aplikací bude zkontrolována struktura souboru. V případě, že je korektní, bude zpracována první část souboru (ta, ve které jsou veškeré potřebné informace). Struktura se zpracovává podle jednotlivých tagů XML souboru a vždy se čtou hodnoty parametrů a atributů u těchto tagů. Podle informací z této části souboru bude vytvořena struktura aritmetických jednotek. Tato struktura je vykreslována do prázdného místa hlavního okna aplikace. Struktura musí být vytvořena tak, že se zobrazí jednotlivé aritmetické jednotky a u každé jednotky se bude v každém směru zobrazovat a ukládat výstupní a vstupní kanál, protože jsou v souboru jednotlivé operace zapsány podle aritmetických jednotek, které je provádí. V případě vykreslování nějaké operace bude tato operace potom vykreslována buď přímo do aritmetické jednotky, nebo do kanálu náležejícímu této jednotce. Po otevření souboru se tedy v hlavním okně zobrazí černobílá matice aritmetických jednotek s uloženými hodnotami propojených komunikačními kanály. Aritmetické jednotky jsou reprezentovány čtvercem, komunikační kanály šipkami (šipka je ve smyslu vstupního nebo výstupního kanálu) a obdélníkem, který potom zobrazuje do kanálu zapsanou hodnotu.

2.8.2.1.3 Spuštění vizualizace

Při spuštění vizualizace se budou do zobrazené struktury aritmetických jednotek pomocí barev a čísel vykreslovat jednotlivé operace, které každá jednotka při simulaci prováděla. K tomu slouží parsování vstupního XML souboru podle jednotlivých tagů (viz 2.8.2.1.5) a příslušné grafické reprezentování jednotlivých operací (viz 2.8.2.1.6). Vizualizace začne zobrazovat operace od prvního taktu bez přerušení až do posledního. Při každém zobrazovaném taktu dojde k volání naparsování jednoho tagu (udávajícího čas) ze vstupního souboru. V případě, že vizualizace doběhne do posledního taktu, zastaví se. Zastavit se

může ještě v případě, že tak bude učiněno příslušným tlačítkem. Při spuštění bude každý takt trvat tak dlouho, jak bude nastaveno.

K nastavení trvání jednoho ve vizualizaci zobrazeného taktu bude sloužit v horejší liště umístěný ovládací prvek, ve kterém se s krokem jedné desetiny vteřiny dá nastavit, jak dlouho má být v případě spuštění vizualizace zobrazen každý takt. Například v případě nastavení této hodnoty na jednu vteřinu, se každou vteřinu změní číslo udávající zobrazenou dobu od začátku výpočtu a každou vteřinu se zároveň s tím provede zobrazení všech operací příslušejících tomuto zobrazenému času. V tomto případě tedy bude vizualizován běh simulace, kdy jeden hodinový signál trvá vteřinu.

2.8.2.1.4 Krokování vizualizace

Při krokování vpřed se do zobrazené struktury aritmetických jednotek vykreslují pomocí čísel a barev jednotlivé operace, které každá jednotka při simulaci prováděla, úplně stejně, jako když se vizualizace spustí. V tomto případě ale s tím rozdílem, že se při každém kliknutí na příslušné tlačítko provede pouze jeden takt, resp. vykreslí se všechny operace, které se provedly v čase, který je zobrazen a na zobrazení dalšího taktu je potřeba stisknout tlačítko znovu. Tímto způsobem se dá celá vizualizace projít stejně, jako když dojde ke spuštění vizualizace, nicméně zde není stanoveno, jak dlouho bude každý takt trvat. Přechod z jednoho taktu na další se provede při stisku tlačítka.

Při krokování dopředu navíc dochází k ukládání historie. Historií je míněno to, že se zaznamenávají stavy všech vykreslených objektů, aby bylo možné krokovat proti směru běhu simulace. Ukládání historie je nutné, protože samotný vstupní XML soubor nenese všechny nutné informace, které by stačily ke krokování zpět. Jelikož se do souboru ukládají pouze změněné stavy aritmetických jednotek, které se pak překreslují (je zbytečné zaznamenávat vždy stav celé matice aritmetických jednotek, protože by se velikost souboru značně zvětšila, jeho parsování by bylo časově náročnější a navíc by spousta informací v souboru říkala, že se s danou jednotkou nic neděje), nevede vracení se v souboru k zobrazování předchozích kroků. Např. v případě, kdy jednotka nic nedělá, nebo je naopak v polovině provádění nějaké operace (provádění začalo dříve než v konkrétně zobrazovaném taktu a končí až v některém dalším taktu) není grafické zobrazení takové jednotky nijak změněno. Pokud by v nějakém taktu skončilo provádění operace, příslušně by se graficky tato akce zobrazila. V případě, že by ale poté bylo stisknuto krokování zpět, této aritmetické jednotce by nebyla barva operace, kterou v předchozím taktu prováděla, přidělena, protože tato informace v XML souboru není. Tam je pouze informace o tom, kdy barvu jednotce přidělit a kdy barvu zase odstranit. Aby byla barva jednotce přidělena, muselo by být zpět odkrokováno až tam, kde daná operace začala.

Z tohoto důvodu je tedy každý krok dopředu zaznamenán a je možné stejný počet kroků zobrazovat i nazpátek. V případě, že bylo krokováno zpět, se informace pro grafické zobrazení čerpají z uložených stavů v historii. Potom,

pokud dojde opět ke krokování (nebo spuštění) vizualizace ve směru běhu simulace, jsou nejdříve brány všechny stavy postupně z historie a podle nich je zobrazováno vše na obrazovku. Až když nejsou v historii již žádné položky (běh vizualizace dorazil na místo, odkud se krokovalo zpět), zobrazují se operace opět ze vstupního XML z toho místa, kde se skočilo (odkud došlo ke krokování zpět).

2.8.2.1.5 Zpracovávání vstupního souboru

Při spuštění nebo krokování vizualizace směrem vpřed se struktura vstupního XML souboru prochází postupně. Při každém zavolání parsování (buď stisknutím tlačítka krokování vpřed nebo uplynutím intervalu nastaveného mezi jednotlivými zobrazovanými takty) je parametrem předáno, jaký takt byl naposledy zpracován. Vstupní soubor se zpracovává podle jednotlivých tagů. Struktura souboru je členěna podle jednotlivých taktů, protože při parsování je pak možné vzít rovnou sousední tag naposledy zobrazovaného taktu.

Vždy se kontroluje, zda je soused skutečně ten tag, jehož atributem je čas shodný s časem zobrazeným v hlavním okně vizualizační aplikace. Pokud takový čas v atributu tagu je, postupně se příslušný tag prochází. Vnořené tagy říkají co jednotlivé aritmetické jednotky provádějí za operace. Zjistí se všechny parametry a potom je zavoláno zobrazování příslušných operací (viz 2.8.2.1.6). Při jednom zavolání parsování jsou pro všechny vnořené tagy jednoho taktu postupně graficky znázorněny příslušné operace aritmetických jednotek. Parsování se volá postupně tolikrát, kolik je ve vstupním souboru jednotlivých tagů udávajích čas.

2.8.2.1.6 Grafické zobrazování jenotlivých operací

Podle parametrů, které byly přečteny při parsování jednotlivých tagů se vykreslují do zobrazené struktury barevně a textem příslušné operace. Podle parametru, který říká, jaký typ operace se má zobrazit je rozhodováno, jakým způsobem bude operace vykreslena. Hodnoty přijaté (přečtené z komunikačních kanálů), odeslané (zapsané do kanálů), uložené v aritmetické jednotce nebo vypočtené, jsou zobrazovány pomocí textu v příslušné jednotce nebo v příslušném kanálu. Jednotka je příslušnou barvou orámována v případě, že je aktivní (stále na ní probíhá výpočet), neaktivní (připravena načítat hodnoty z další soustavy lineárních kongruencí), má vypočtený výsledek, je zvolena pivotem a nebo byla pivotem zvolena už dříve. Výplň aritmetických jednotek se mění podle toho, jakou výpočetní operaci jednotka provádí (multiplikativní inverzi, negaci, násobení inverze s hodnotou v jednotce nebo s tímto součinem násobenou negaci a přičtenou hodnotou v jednotce). Barevnost šipek (komunikačních kanálů) je zobrazována podle toho, jaký typ hodnoty je do kanálu zapisován, nebo z kanálu čten.

Aritmetická jednotka je objekt, který má uloženy všechny vstupní i výstupní komunikační kanály, má stanoveno, kam se umísťuje text odpovídajícího typu hodnoty a je stanovena škála barev, které jsou pojmenovány podle jednotlivých operací. Aritmetická jednotka má také definované metody, které mění obrys, barvu, text jednotky a ty samé metody pro připojené komunikační kanály. V případě vykreslení příslušné operace se na konkrétní aritmetickou jednotku jen zavolá správná metoda jejíž argument je v případě vykreslování hodnoty text (hodnota je jeden z parametrů, které jsou při parsování zjištěny z příslušného tagu) nebo v případě vybarvení název operace, která se znázorňuje a metoda správně vykreslí text nebo vybere správnou barvu a vybarví objekt.

2.8.2.1.7 Nastavení

Ve formuláři nastavení se dají nastavit parametry, které jsou nutné ke správnému spuštění simulace a dá se vytvořit soubor se soustavami lineárních kongruencí, jehož obsah bude vygenerován podle zadaných parametrů.

Simulace se dá spustit přímo z hlavního okna vizualizační aplikace. Pro úspěšné spuštění simulace je potřeba zadat parametry, které jsou potom použity při běhu simulace. Je to vstupní soubor, ve kterém jsou uloženy soustavy k vyřešení a výstupní soubor, přesněji jeho jméno, do kterého se bude ukládat běh simulace ve struktuře XML. Právě tento soubor bude potom vizualizační aplikací využíván k zobrazení operací vykonávaných všemi aritmetickými jednotkami. Posledním, přesněji posledními čtyřmi parametry, je doba trvání multiplikativní inverze, negace, násobení a sčítání během výpočtu. Všechny tyto parametry je možné nastavit v nastavovacím formuláři a ty se předají simulaci při jejím zahájení příslušným tlačítkem.

Při generování nového souboru se soustavami lineárních kongruencí jsou v nastavovacím formuláři příslušná pole, jejichž vyplnění je nutné, protože hodnoty z těchto polí slouží jako parametry pro generátor. Při vygenerování souboru je tento použit jako vstupní soubor pro simulaci.

Při vybírání souboru pro simulaci je použito procházení adresářů a výběr pouze existujících souborů. Soubor pro vizualizaci je zadán pouze jeho jménem, ale simulace může být spuštěna i bez ukládání běhu aritmetických jednotek do XML souboru. Při zadávání doby trvání jednotlivých operací používaných při výpočtu je kontrolováno, zda zadávané hodnoty dávají smysl. V případě chyby je zobrazeno chybové oznámení s detailním popisem chyby.

Před generováním nového souboru musí být všechny parametry pro generátor vyplněny. Navíc je kontrolováno, zda jsou vyplněny správně. Opět je v případě chyby zobrazeno oznámení s popisem, proč jsou vyplněné hodnoty špatně.

Vyplněné hodnoty z nastavovacího formuláře jsou ukládány a v případě opětovného otevření nastavení nebo spuštění aplikace následované otevřením nastavení, budou hodnoty vyplněny tak, jak byl formulář naposledy potvrzen.

2.8.2.2 Realizace

Při startu aplikace se vytvoří objekt třídy `Vizualizace`. Rovnou z konstruktoru se volá metoda `vytvoritLegendu()`, která do pravého okraje hlavního okna aplikace zobrazí všechny příslušné barvy k popisům v legendě.

Při otevírání seoboru metodou `otevrit()` je zavolána na objekt třídy `Parser` metoda `nacti()`, která zkontroluje strukturu vstupního souboru a voláním metody `naparsujInfo()` zjistí potřebné informace (jako velikost soustavy), podle kterých bude vytvářet strukturu aritmetických jednotek. Voláním metody `vytvoritStrukturu()` (parametrem jsou zjištěné rozměry) se ve třídě `Vizualizace` vytváří všechny aritmetické jednotky i kanály. Všechny objekty, u kterých se bude měnit vzhled, jsou ze třídy `renderArea`. Tyto objekty se vytváří se zadanými parametry (velikost, pozice v ploše) a po vytvoření se vždy vloží do plochy a grafická knihovna je v příslušných místech vykreslí. K vytvořeným aritmetickým jednotkám (třída `vypocetniJednotka`) uloženým ve vectoru se přiřadí komunikační kanály a šipky metodami `priraditKanaly()` a `priraditSipky()` (aby se metody měnící vzhled mohly potom volat na příslušnou jednotku, resp. na objekty reprezentující kanály náležící příslušné jednotce). Poté se voláním metody `pocatecniVzhled()` nastaví původní vzhled celé zobrazené struktuře.

Vizualizace se spouští příslušným tlačítkem. V metodě `spustit()` je napojena metoda `hod()` na hodinový signál, jehož parametrem je interval, po jehož uplynutí bude vždy automaticky znovu zavolána metoda `hod()`. Parametr lze nastavit s krokem jedné desetiny vteřiny a udává, jak dlouho bude ve vizulizaci trvat každý takt. Metoda `hod()` volá metodu `krok()`. Tato metoda je volána i po stisku tlačítka krokování vpřed. Při spuštění vizualizace je pomocí interního timeru volána automaticky vždy po uplynutí nastaveného časového intervalu.

Při zavolání metody `krok()` dojde k zavolání metody `proved()` na objekt třídy `Parser` a parametrem se metodě předává aktuálně zobrazený čas od začátku výpočtu a ukazatel na tag, který byl zpracován naposledy. Při krokování se ukládá historie. Vždy se vytvoří objekt třídy `Historie`. Ten je naplněn stavy všech zobrazených aritmetických jednotek a k nim náležejících komunikačních kanálů. Stav jednoho zobrazeného objektu je uložen v objektu třídy `Stav`. Při vytváření jednotho objektu `Historie` se tedy projdou všechny zobrazené objekty třídy `vypocetniJednotka` a všechny potřebné parametry na vykreslení jsou uloženy do objektu `Stav`. Stejně je tomu i v případě komunikačních kanálů. Stavy všech objektů tvoří jeden objekt `Historie`. Jeden objekt `Historie` značí jeden zobrazený čas během vizualizace. Každý stisk tlačítka pro krokování vpřed přidá do vectoru jeden objekt třídy `Historie`. Při krokování zpět se v metodě `krokZpet()` vybírají postupně z vectoru objekty `Historie` a pomocí metody `zobraz()` v této třídě se zobrazí ke správnému času struktura s konkrétním obarvením aritmetických jednotek a kanálů a textem reprezentujícím hodnoty.

Při volání metody `proved()` na objekt typu `Parser` je k naposledy provedenému tagu předanému parametrem nalezen sousední tag (struktura XML souboru je členěna podle tagů udávajících čas od počátku výpočtu, kdy jeden tag zahrnuje všechny operace všech aritmetických jednotek v daném čase). Potom je zavolána metoda `naparsuj()`, která z nalezeného sousedního tagu přečte postupně pro všechny vnořené tagy všechny parametry a vždy pro každý vnořený tag volá metodu `namaluj()` s těmito parametry.

V metodě `namaluj()` se podle parametru `druh_hodnoty` rozhoduje, jak se operace vykreslí. Na daný procesor (jeden z parametrů) se volají metody definované ve třídě `vypocetniJednotka` (např. `setBarvaPozadi()`), které volají metody třídy `renderArea`, které pomocí grafické knihovny provedou změnu zobrazení (např. barvy pozadí). Barvy pro jednotlivé operace jsou definovány pomocí výčtu `enum`.

Formulář nastavení (třída `Nastaveni`) se otevře po stisku příslušného tlačítka. Při otevření formuláře se pomocí metody `zobrazeni()` vyplní do příslušných polí hodnoty, se kterými byl formulář nastavení naposledy potvrzen. Před potvrzením nastavení (a zavřením formuláře) je kontrolováno metodou `kontrola()`, zda jsou pole vyplněná korektní hodnotou. V případě chyby je oznámeno, co bylo zadáno chybně (metoda `chyba()`). Při potvrzení nastavení (metoda `potvrzeniNastaveni()`) jsou všechny vyplněné položky uložené pomocí metody `uloz()`. Při stisku tlačítka pro generování nového souboru pomocí vyplněných parametrů je volána metoda `generuj()`, které se předávají vyplněné parametry.

2.8.2.2.1 Použitý software a knihovny

- Zdrojové kódy jsou napsané v C++, jako vývojové prostředí bylo použito Microsoft Visual Studio 2008 Professional Edition [32] - verze šířená v rámci MSDNAA [22] (Microsoft Developer Network Academic Alliance). Přeloženo překladačem ve vývojovém prostředí.

- Grafické prvky zajišťuje knihovna Qt ve verzi 4.7.1 [26]. Knihovna je použita pod licencí GNU LGPL v2.1 [21] (GNU Lesser General Public License).

- Qt doplněk do Microsoft Visual Studia ve verzi 1.1.7 [25] pro editaci kódu jednotlivých formulářů přímo ve vývojovém prostředí.

KAPITOLA 3

Testování

Testování, které bylo prováděno jednak během realizace, jednak na hotové aplikaci, by se dalo rozdělit do tří částí.

Správná funkčnost jednotlivých realizovaných částí byla testována vždy již při jejich implementaci. Každá dílčí část, která byla navžena, byla po realizaci i testována. Testování probíhalo ručně. Dosavadní funkčnost byla vždy spuštěna ve vývojovém prostředí a pomocí jednoduchých metod k tomu implementovaných bylo vyzkoušeno, že je funkcionalita chování podle očekávání. Z těchto metod se vždy postupem času vytvořily zdokonalováním a přibývající funkcionalitou ty metody, které se používají ve výsledném řešení. Při propojování nových funkčních částí ke stávajícím, již odladěným, vždy docházelo k volání jak metod, které testovaly dosavadní celek, tak metod nových, které otestovaly, zda je vzájemná interakce funkční a výsledné operace jsou správné.

Během vývoje bylo využíváno krokování ve vývojovém prostředí, které umožňuje jak výpis všech proměnných programovacího jazyka, tak společně s knihovnou SystemC i stav jednotlivých signálů a všech používaných částí knihovny. Z tohoto důvodu bylo možné sledovat podrobně správnou funkčnost jednotlivých implementovaných dílčích částí výsledného simulátoru.

V případě testování funkčnosti nezávislého výpočtu na všech aritmetických jednotkách byl za každou operaci, kterou procesy v aritmetické jednotce provádějí, přidán výpis obsahující název jednotky, operaci, data a čas, ve kterém k operaci dochází. SystemC totiž umožňuje výpis do konzole řadit podle času události a i když běží procesy synchronně, výpisy do konzole z jednotlivých procesů se nemíchají dohromady, ale je vždy nejprve vypsán jeden kompletní výpis jednoho procesu, pak dalšího, atd. Pořadí jednotlivých vypsaných operací může být různé, pokud jsou prováděny ve stejném čase. Po spuštění simulace vznikl jakýsi log celého běhu simulace. Jelikož byly výpisy operací seřazeny podle času, kdy byly prováděny, bylo možné celý běh rekonstruovat

jednak kvůli tomu, zda probíhá správně a jednak kvůli tomu, zda probíhá se správnými hodnotami.

Po implemetaci celé struktury aritmetických jednotek a jejich souběhu bylo nutné vyzkoušet všechny krizové situace, které bylo potřeba odstanit. K tomuto účelu sloužil generátor náhodných soustav lineárních kongruencí, který vytvářel náhodně velké soustavy, s náhodnými moduly a náhodnými hodnotami v soustavách. Později ještě s náhodným počtem soustav v jednotlivých generovaných souborech. Ručně potom ještě byly vytvářeny takové soustavy, kde se vyskytuje hodně nulových hodnot, aby soustavu nebylo možné vyřešit a zadání bylo upravováno postupně tak, aby se otestovalo, že je možné skončit výpočet v jakémkoliv kroku eliminace. Dále byly vytvářeny takové soustavy, aby se otestovalo, že je možné vybírat pivota v jakémkoliv pořadí, že nezáleží na tom, zda jsou pivoti vybíráni seshora, nebo ne. Také byly sestaveny takové sady soustav, aby se otestovalo, že po řešení jedné soustavy, může být konec simulace nezávisle na tom, zda ji bylo možné vyřešit, nebo ne. Stejně tak, že je možné řešit další soustavu nezávisle na tom, jak dopadlo řešení předchozí.

K tomu, aby bylo otestováno, že je vypsané výsledné řešení skutečně správné, nebo naopak k tomu, aby bylo možné zjistit, že není skutečně možné soustavu dořešit, sloužil software Wolfram Mathematica [34]. V tomto programu je možné na vstup zadat stejnou soustavu, nebo soustavy, jako do implementovaného simulátoru a vypsat si matici po provedení příslušných operací. Jelikož je tento software dost dobrým adeptem na bezchybné matematické výpočty, stačilo porovnat výstup ze simulátoru s výsledky programu Wolfram Mathematica.

Testováno také bylo řešení soustav s různým nastavením trvání jednotlivých operací. Bylo otestováno i v hardwaru nesmyslné nastavení parametrů, kdy všechny operace trvají jeden takt, nebo kdy např. sčítání trvá mnohonásobně delší dobu, než násobení. Jelikož jsou si všechny aritmetické jednotky rovné, to znamená, že počítají (pokud mají správná data) stejné operace stejnou rychlostí a stejnou dobu, není možné, aby např. polovina jednotek nepočítala, když má, nebo počítala nějakým způsobem rychleji. Nicméně je testováno i umělé zpomalování některých jednotek, protože jinak nebylo možné vyzkoušet některé funkce zabraňující kolizi hodnot nebo uváznutí. V reálném běhu však k těmto ochranným funkcím nikdy nedojde právě z toho důvodu, že jsou všechny jednotky stejné a počítají, když mají počítat.

Jako konečný způsob kontroly bylo prováděno pro všechny generované soustavy lineárních kongruencí dosazování. To znamená, že stačí vektor řešení přeházet do správného pořadí, potom dosadit do levé strany matice a vypočítat ji. Výsledkem je vektor hodnot, který musí být naprosto shodný s vektorem pravých stran zadaných ve zdrojovém souboru.

Tímto způsobem byly laděny jednotlivé dílčí části, jejich spojování v celek, souběh celé struktury a výsledné řešení soustav lineárních kongruencí. Všechny tyto způsoby odhalily chyby, které byly v průběhu vývoje odstraněny a byla zavedena příslušná opatření, aby k nim nedocházelo. Vzhledem k tomu je

simulační aplikace hotovým funkčním celkem pro řešení soustav lineárních kongruencí.

3.1 Měření hodnot

V rámci testování proběhlo měření činnosti aritmetických jednotek během výpočtu. Jednotka je aktivní, pokud provádí výpočetní, nebo komunikační operace. Měření probíhalo na maticích aritmetických jednotek o 4, 14 a 24 řádcích. Vždy se řešila jedna soustava od začátku do konce. U každé velikosti matice probíhala dvě měření. Nejdříve samotného výpočtu bez nasouvání nových hodnot z další matice a potom měření výpočtu, při kterém se zároveň zprava nasouvaly hodnoty z další matice do jednotek, které se již na výpočtu nepodílejí.

- Výpočet probíhal pro každou velikost matice vždy na stejných datech (stejně zadaná soustava lineárních kongruencí), ať se jednalo o výpočet s nasouváním, nebo bez.
- V průběhu měření bylo využito nastavení různého trvání jednotlivých operací během výpočtu:
  - multiplikativní inverze trvala *1,4n* taktů,
  - negace *1* takt,
  - násobení n taktů,
  - sčítání *1* takt.
- Při tomto nastavení se jako n volí počet bitů modulu, což byla v případě měření hodnota rovná 7, tedy tři bity.

Měření probíhalo na jedné soustavě od začátku výpočtu, tedy hned od začátku simulační fáze až do doby, kdy byl přijat signál na zahájení řešení další soustavy. Při příchodu tohoto signálu je buď zahájeno řešení další soustavy, nebo v případě, že žádná další soustava v jednotkách načtená není, simulace končí. Aby bylo možné zjistit, zda se má simulace ukončit, je potřeba, aby zásobovací moduly v případě, že neexistuje další soustava k vyřešení, poslaly do maticové struktury výpočetních jednotek příslušný druh hodnoty. Ten se posílá zprava doleva a až doputuje do prvního sloupce matice, je po příchodu signálu na zahájení řešení další soustavy díky této ukončovací hodnotě upřednostněno zastavení simulace před řešením další soustavy. Z tohoto důvodu není měření bez nasouvání nových hodnot do matice omezeno jen na samotný výpočet, ale na výpočet společně s distribuováním ukončovací hodnotou.

Na druhou stranu v případě měření s nasouváním hodnot, naopak tato ukončovací hodnota distibuována není, protože je potřeba nasunout novou

soustavu do matice výpočetních jednotek a teprve po jejím vyřešení by v případě, že byla poslední, následovalo distibuování ukončovací hodnoty. V měření bylo důležité zahrnout nasouvání nové soustavy, ale ne již její výpočet. Z tohoto důvodu bylo uměle řešení ukončeno při běhu simulace s nasouváním po přijetí signálu startujícího výpočet nové soustavy. Ukončení měření tedy proběhlo na stejném místě, jen s tím rozdílem, že do měření bez nasouvání hodnot se zahrnulo i distribuování ukončovací hodnoty, které v případě měření s nasouváním hodnot zahrnuto není. Z principu není možné jen kvůli měření celý chod struktury předělat tak, aby bylo distribuování ukončovací hodnoty zahrnuto buď v obou typech měření, nebo v žádném.

V následujících tabulkách je vidět, jaké výsledky měření ukázalo pro všechny tři velikosti zvolené struktury.

Soustava 4x5 (20 aritmetických jednotek)	**Nasouvání**	
	Ne	**Ano**
celkový počet taktů výpočtu	72	79
minimum taktů, kdy je AU aktivní	12	21
maximum taktů, kdy je AU aktivní	41	43
průměrně taktů na 1 AU, kdy je aktivní	28,15	32,90
minimální aktivita z celkové doby běhu	16,67 %	26,58 %
maximální aktivita z celkové doby běhu	56,94 %	54,43 %
průměr aktivity na 1 AU z celkové doby běhu	39,10 %	41,65 %
maximum aktivních AU v jednom taktu	16	16
průměr aktivních AU na jeden takt	7,82	8,33

Tabulka 3.1: Měřené hodnoty na struktuře 20 aritmetických jednotek

Soustava 14x15 (210 aritmetických jednotek)	**Nasouvání**	
	Ne	**Ano**
celkový počet taktů výpočtu	247	274
minimum taktů, kdy je AU aktivní	12	51
maximum taktů, kdy je AU aktivní	141	143
průměrně taktů na 1 AU, kdy je aktivní	78,53	101,62
minimální aktivita z celkové doby běhu	4,86 %	18,61 %
maximální aktivita z celkové doby běhu	57,09 %	52,19 %
průměr aktivity na 1 AU z celkové doby běhu	31,79 %	37,09 %
maximum aktivních AU v jednom taktu	132	133
průměr aktivních AU na jeden takt	66,77	77,89

Tabulka 3.2: Měřené hodnoty na struktuře 210 aritmetických jednotek

Soustava 24x25 (600 aritmetických jednotek)	**Nasouvání**	
	Ne	**Ano**
celkový počet taktů výpočtu	426	474
minimum taktů, kdy je AU aktivní	12	81
maximum taktů, kdy je AU aktivní	241	243
průměrně taktů na 1 AU, kdy je aktivní	128,54	170,55
minimální aktivita z celkové doby běhu	2,82 %	17,09 %
maximální aktivita z celkové doby běhu	56,57 %	51,27 %
průměr aktivity na 1 AU z celkové doby běhu	30,17 %	35,98 %
maximum aktivních AU v jednom taktu	385	385
průměr aktivních AU na jeden takt	181,04	215,89

Tabulka 3.3: Měřené hodnoty na struktuře 600 aritmetických jednotek

3.1.1 Naměřená data

U všech tří zvolených velikostí struktury je patrné, že se s nasouváním zvyšuje i počet taktů, ve kterých se simulace provádí. Je tomu tak z toho důvodu, že se soustava do matice nasouvá sice již při výpočtu, ale nikdy se nenasune celá, až po skončení všech eliminačních kroků je potřeba zbytek soustavy nasunout úplně doleva v matici výpočetních jednotek. Proto vznikají takty navíc. Nicméně pokud by byla matice plněna celá až po skončení výpočtu, čas by vzrostl mnohem více, protože v případě nasouvání soustavy zároveň během výpočtu schází po skončení nasunout méně než třetina hodnot.

U všech tří měřených velikostí roste s nasouváním hodnot také průměrný počet taktů na jednu aritmetickou jednotku, kdy je jednotka aktivní, tedy provádí nějaké operace. To je z toho důvodu, že se podstatným způsobem zvětšilo minimum taktů pro všechny jednotky, kdy jsou aktivní. Čím více je jednotka v matici vpravo, tím více se zvětšila její aktivita, protože postupně načítá všechny hodnoty z další soustavy, kdežto jednotka ve sloupci o jeden vlevo celkově distribuuje o hodnotu méně, atd. To je zase kompenzováno tím, že sloupec vpravo je hned po prvním kroku eliminace neaktivní, kdežto druhý zprava až po druhém kroku, atd.

Maximální počet taktů je také s nasouváním vyšší, ale ne do takové míry, protože je v měření bez nasouvání hodnot zahrnuto i distribuování ukončovací hodnoty. Kvůli zahrnutí distribuce ukončovací hodnoty do měření bez nasouvání je totiž v měření bez nasouvání zvýšen počet taktů, kdy jsou jednotky aktivní. Podle toho, jak distribuce ukončovací hodnoty funguje, je jasné, že ovlivňuje maximální aktivitu, protože tu vykazují jednotky v levé části matice. Bez zahrnutí distribuce by byly maximální hodnoty u měření bez nasouvání nižší, než je uvedeno v tabulkách, a tím by byl i u maximální aktivity rozdíl

vyšší.

Poslední tři řádky v tabulkách jsou jen procentuálním vyjádřením hodnot zapsaných v řádcích nad. Jedná se tedy o hodnoty, které říkají, jak dlouho je jednotka z celkové doby běhu aktivní. U maximální aktivity je s nasouváním hodnota nižší, než bez nasouvání. To je dáno tím, že se maximum zvýší oproti měření bez nasouvání jen nepatrně (vzhledem k zahrnutí distribuce ukončovací hodnoty do měření bez nasouvání, která zvedne maximální aktivitu při výpočtu bez nasouvání) a celkový počet taktů se zvýší. Poměr je tedy proto menší.

Při porovnání naměřených údajů z jednotlivých velikostí struktury mezi sebou vyplývá, že klesá průměrná aktivita jednotky s rostoucím počtem řádků matice. Je to zapříčiněno tím, že je minimum aktivity z celkového běhu pořád menší, protože počet taktů, které vykoná jednotka v pravém sloupci je nezávislý na velikosti struktury, ale celkový počet taktů se zvyšuje. Poměr je tedy menší. Naopak je vidět, že se s rostoucím počtem řádků stále více zvyšuje zapojením nasouvání soustavy do výpočtu rozdíl mezi minimem aktivity s a bez nasouvání. Maximální aktivita se pohybuje na téměř stejných hodnotách a je tedy na velikosti soustavy závislá jen minimálně.

Při zvětšování hodnoty modulu, ve kterém se počítá, se snižuje průměr aktivity na aritmetickou jednotku z celkové doby běhu. To je zapčíčiněno tím, že provádění inverze zabere poměrně velké množství taktů, při kterém ostatní jednotky nemají co na práci, protože je pro ně hodnota multiplikativní inverze stěžejní, a proto čekají, a proto se snižuje aktivita vzhledem k celkové době běhu.

Z hodnot v tabulkách je patrné, že při jednom taktu zůstává maximální počet aktivních jednotek nezměněn nezávisle na měření s nebo bez nasouvání nových hodnot. V maximu není zahrnut začátek simulace, kdy jsou všechny jednotky po spuštění aktivní, než začnou čekat na příchozí data. V tomto případě by byl maximální počet aktivních jednotek v jednom taktu vždy stejný, jako celkový počet všech jednotek. Proto není pro lepší představu toho, jak jsou skutečně jednotky aktivní během výpočtu, v tabulce tento počáteční takt zahrnut. Se zapojením nasouvání hodnot z další soustavy do výpočtu se ve všech měřených velikostech zvýšil průměr aktivních jednotek na jeden takt. To je díky tomu, že se jednotky, které už ve výpočtu zapojené nejsou, starají o distribuci nových hodnot, a tím pádem jsou aktivní.

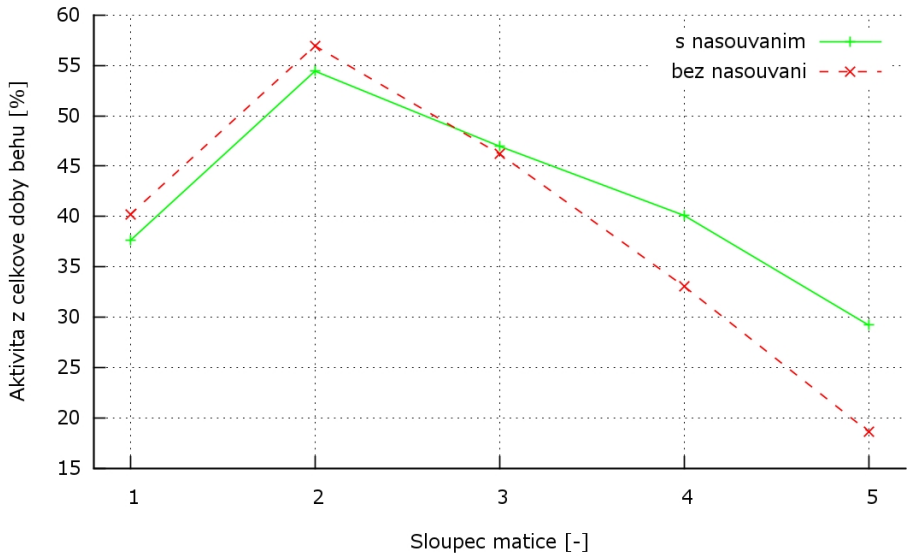
V případě, že je k vyřešení tak zadaná soustava, že se musí vybírat pivoti od poslední řádky vždy směrem nahoru, je to nejhorší možný případ, který lze řešit. Výpočet se prodlouží na dvojnásobný počet taktů, protože pivotizace probíhá seshora a v každém kroku se musí projít až k řádce, kde je nenulový pivot. Při těchto operacích je aktivní vždy jen jedna jednotka, ostatní čekají. Jelikož počet taktů, kdy je jednotka aktivní, zůstává stejný (minimální a maximální) nebo téměř stejný (průměrný - nepodstatným způsobem se počet taktů zvýšil oproti výběru pivota seshora), aktivita jednotek (minimální, maximální i průměrná) spadla vzhledem k celkové době výpočtu na polovinu. Průměrně

je také v jednom taktu aktivní pouze polovina jednotek oproti případu, kdy se pivot vybírá zezhora.

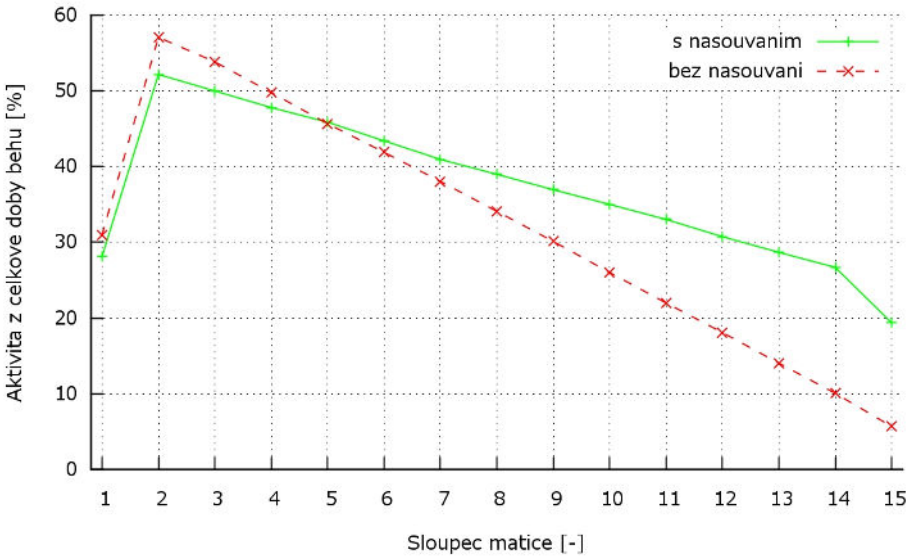
V případě, že je řešeno několik soustav za sebou, je využito načítání další soustavy vždy v průběhu výpočtu soustavy předešlé. V takovém případě se aktivita jednotek vzhledem k celkové době běhu sníží o 2 % oproti hodnotám uvedeným v pravém sloupci tabulek (s nasouváním). Je to proto, že do výpočtu zasahoval také fakt, že byl zvolen malý modul, tudíž je hodnota rovná nule v soustavách poměrně častá a díky ní není možné soustavy dořešit a tím pádem nejsou jednotky aktivní, i když by v případě nenulových hodnot (a pokračování ve výpočtu) mohly být.

3.1.1.1 Činnost aritmetických jednotek v závislosti na umístění v matici

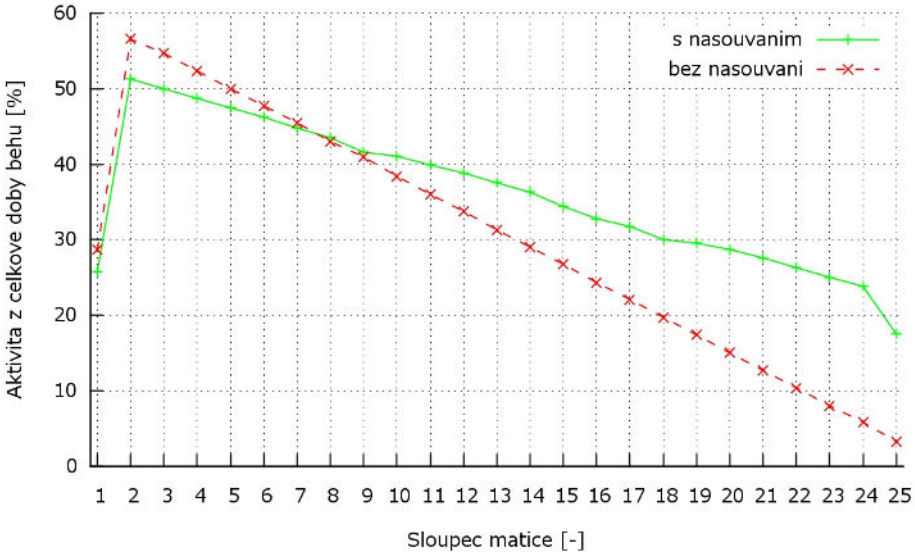
Následující grafy ukazují, jak jsou na tom s činností aritmetické jednotky v závislosti na tom, kde v matici se nacházejí. Při výpočtu se posouvají hodnoty vždy o jeden sloupec matice směrem doleva, díky čemuž je aktivita jednotek ležících ve stejném sloupci vždy téměř přesně shodná. Na vodorovné ose jsou hodnoty udávající ve kterém sloupci matice se jednotky nacházejí, na svislé pak procentuální vyjádření aktivity jednotky z celkové doby běhu.



Obrázek 3.1: Činnost aritmetických jednotek v závislosti na umístění v matici s 5 sloupci



Obrázek 3.2: Činnost aritmetických jednotek v závislosti na umístění v matici s 15 sloupci



Obrázek 3.3: Činnost aritmetických jednotek v závislosti na umístění v matici s 25 sloupci

3.1.2 Maximální velikost soustavy

Maximální velikost simulované soustavy je závislá na hardwarové konfiguraci počítače, na kterém je aplikace spuštěna. Je to zejména z toho důvodu, že všechny procesy (vlákna) běží simultánně a při určitém počtu již není možné, aby hardware dokázal simulovat pomocí simulačního jádra SystemC všechny procesy. Na následující konfiguraci:

- **Procesor:** Intel(R) Core(TM) 2 Duo T5600 @ 1.83GHz,
- **Operační paměť:** 2GB,
- **Operační systém:** Microsoft Windows 7 (32 bit),

bylo možné simulovat maximálně soustavu o velikosti 26 řádků (27 sloupců). Tento počet odpovídá 1431 simultánně běžícím vláknům. V případě větší soustavy (27 řádků) již simulační jádro řídí 1540 procesů zároveň, protože:

- 27 x 28 jednotek = **756** výpočetních procesů,
- **756** čtecích procesů,
- **27** procesů zásobovacích modulů,
- **1** proces sběrného modulu.

Tento počet výše uvedená konfigurace nebyla již schopna simulovat. V případě zápisu aritmetickými jednotkami do komunikačních kanálů se samozřejmě ještě navíc přidávají další zároveň běžící procesy (procesy zápisu).

Doba, za kterou se na výše uvedené hardwarové konfiguraci vyřeší soustava o 26 řádcích odpovídá 3,5 vteřinám.

3.1.3 Složitost výpočtu

Při výpočtu záleží na některých faktorech, ovlivňujících dobu běhu a složitost celého výpočtu:

- rozměr matice n,
- počet bitů modulu m,
- trvání jednotlivých operací,
- zadaná soustava (kvůli výběru pivota).

Složitost jednotlivých operací je v průměrném případě:

- multiplikativní inverze $O(m)$, protože trvá $1{,}4m$ taktů
- negace $O(1)$, protože trvá 1 takt,

- násobení *O(m)*, protože trvá m taktů,
- sčítání *O(1)*, protože trvá *1* takt.

Operace dosahují tedy maximálně lineární složitosti.

V nejlepším případě, kdyby všechny operace trvaly konstantní čas (počet bitů modulu ani složitost operace by se neprojevily), pivot by byl v každém kroku k zvolen na řádce k, by byla doba běhu nejmenší. V případě nekonečné matice a nekonečného běhu jednoho kroku za druhým, by se složitost blížila k *O(n)*, protože by se distribuovaly „vlny“ jednotlivých eliminačních kroků jedna za druhou, a tak by výpočetní jednotky téměř neustále počítaly.

V případě, že by byla soustava zadaná tak, že by se muselo pro nalezení pivota prohledat v prvním kroku n řádků, ve druhém n - *1*, atd., by doba potřebná pro výpočet trvala dvojnásobek.

V případě, že se doba trvání operací považuje za fixní, stejně jako počet bitů modulu, by byla složitost ovlivněna pouze rozměrem matice n. Pokud by byl výpočet uskutečněn jednou aritmetickou jednotkou (sekvenční zpracovávání), byla by složitost $O(n^3)$. V případě použití vektoru aritmetických jednotek, by byl výpočet se složitostí $O(n^2)$. V případě použití matice aritmetických jednotek by teoreticky měla být složitost *O(n)*, nicméně pouze v případě, že by počítaly po celou dobu výpočtu všechny jednotky. To ale není možné, protože v některých případech jednotky čekají na příchozí hodnotu, takže nejsou aktivní. Složitost je tedy $\Omega(n) \wedge O(n^2)$.

V reálném případě se se zvětšováním modulu zvyšuje i doba potřebná pro výpočet, protože čím je vyšší počet bitů modulu, tím déle trvá multiplikativní inverze a jednotky neprovádějící tuto operaci čekají na hodnotu. V takovém případě se složitost se zvyšujícím se počtem bitů modulu posouvá dál od $\Omega(n)$ blíž směrem k $O(n^2)$, nicméně v maticích o velkých rozměrech bude čekání na hodnotu trvat menší dobu než distribuce hodnoty na konec matice a tedy i výpočet všech jednotek v řádcích, takže je velká část jednotek stále aktivních, protože bude v matici za sebou více „vln“ jednotlivých eliminačních kroků. Složitost sice nebude *O(n)*, ale bude rozhodně lepší než $O(n^2)$.

Závěr

V rámci této práce se podařilo navrhnout a realizovat funkční simulaci řešiče soustav lineárních kongruencí. Tento, v simulačním jazyce SystemC naimplementovaný simulátor, využívá k řešení soustav Gauss-Jordan-Rutishauserovu eliminační metodu. O celý výpočet se starají aritmetické jednotky pracující v modulární aritmetice, přičemž se jejich počet odvíjí vždy od velikosti zadané soustavy. Ty umožňují díky maticové struktuře, do které jsou uspořádány, provádět výpočet paralelně.

Druhou funkční aplikací, která byla v této práci navrhnuta a poté realizována, je vizualizační aplikace, která slouží ke grafické reprezentaci paralelního běhu aritmetických jednotek simulátoru.

V textové podobě práce je čtenáři v první kapitole poskytnut základní přehled všech nutných náležitostí, které bylo k úspěšnému dosažení cílů práce potřeba nastudovat a je potřeba je znát ke správnému porozumění problematice, ze které pak vycházel návrh a realizace.

Všechny stanovené požadavky se podařilo v rámci práce splnit a výsledkem je funkční simulace hardwarového řešiče soustav lineárních kongruencí, kterou je možné k řešení ihned využít. Funkční celek této práce je navíc rozšířen i o možnost zobrazit prováděné operace všech aritmetických jednotek během simulace pomocí vizualizace.

Výsledek této výzkumné práce může posloužit k porovnání s řešičem, který je hardwarově realizován na fakultě, kdy se k výpočtu používá vektor funkčních jednotek. Z tohoto důvodu byla v této práci zvolena maticová struktura jednotek. Nedá se však zcela jednoznačně doporučit maticová struktura výpočetních jednotek oproti vektoru. Sice je v maticové struktuře více využito paralelního výpočtu, aktivita všech jednotek ale není tak vysoká. V případě skutečné realizace by byly za větší počet aritmetických jednotek při použití maticové struktury vyšší výrobní náklady, nicméně díky lepšímu využití paralelního výpočtu by bylo dosaženo jakési kompenzace v podobě kratší doby výpočtu.

Pokračováním této práce by mohlo být kompletní srovnání maticové struktury aritmetických jednotek se strukturou vektoru aritmetických jednotek. Dalším možným pokračováním práce by bylo řešení na více procesorech pracujících v aritmetice kódů zbytkových tříd a převod řešení ze zbytkové reprezentace do racionálních čísel.

Literatura

[1] Black D. C., Donovan J.: *System C: From The Ground Up*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.scribd.com/doc/53124102/Programming-SystemC-From-the-Ground-Up>

[2] Demlová, M.: *Zbytkové třídy modulo n*. [cit. 2012-02-26]. Dostupné z WWW: <http://math.feld.cvut.cz/demlova/teaching/dml/pred11.pdf>

[3] Fonoage, M.: *SystemC with Microsoft Visual Studio 2008*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.ee.ryerson.ca/~courses//coe718/labs/SystemC-Installation-for-Windows.pdf>.

[4] IEEE: *IEEE Standard for Standard SystemC Language Reference Manual*. [cit. 2012-02-26]. Dostupné z WWW: <http://standards.ieee.org/getieee/1666/download/1666-2011.pdf>

[5] Krejsa, M.: *Algoritmizace inženýrských výpočtů - Soustavy lineárních rovnic - přednáška 5*. [cit. 2012-01-12]. Dostupné z WWW: <http://fast10.vsb.cz/krejsa/studium/algoritmy_05.pdf>

[6] Litschmannová, M.: *Gauss-Jordanova metoda*. [cit. 2012-02-26]. Dostupné z WWW: <http://homel.vsb.cz/~lit40/LA/PDF/GJM_p.pdf>.

[7] Lórencz, R.: *Aplikovaná numerická matematika - Řešení soustav lineárních rovnic - přednáška 3*. [cit. 2012-01-12]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/PI-ANM/_media/lectures/03/prednask3.pdf>

[8] Maněna, M.: *Simulace logických obvodů v SystemC*. [cit. 2012-02-26]. Dostupné z WWW: <https://service.felk.cvut.cz/vlsi/dip/Manena08/dp.pdf>

[9] Moondanos, J.: *SystemC Tutorial*. [cit. 2012-02-26]. Dostupné z WWW: <http://embedded.eecs.berkeley.edu/research/hsc/class/ee249/lectures/l10-SystemC.pdf>

[10] Morháč M., Lórencz R.: *Modular system for solving linear equations exactly - Architecture and numerical algorithms*. [cit. 2012-02-26]. Dostupné z WWW: <http://users.fit.cvut.cz/lorencz/clanky/7_Modular_System_2_1992.PDF>

[11] Morháč M., Lórencz R.: *Modular system for solving linear equations exactly - Hardware realization*. [cit. 2012-02-26]. Dostupné z WWW: <http://users.fit.cvut.cz/lorencz/clanky/8_Modular_System_1_1992.PDF>

[12] Šolcová, A.: *Matematika pro informatiku - přednáška 2-3*. [cit. 2011-03-07]. Dostupné z WWW: <https://edux.fit.cvut.cz/courses/MI-MPI/_media/lectures/02/mpi_2-3-as.pdf>

[13] Olšák, P.: *LU tozklad*. [cit. 2012-04-27]. Dostupné z WWW: <http://petr.olsak.net/bilin/lurozklad.pdf>

[14] Prasad, M.: *SystemC: An Introduction for beginners*. [cit. 2012-02-26]. Dostupné z WWW: <http://electrosofts.com/systemC/index.html>

[15] Růžička, J.: *TEORIE ČÍSEL - Sbírka příkladů*. [cit. 2012-02-26]. Dostupné z WWW: <http://is.muni.cz/th/42653/fi_m/DIPLOMKA.pdf>

[16] Sadílek, M.: *Paralelní řešič soustav lineárních rovnic v aritmetice kódů zbytkových tříd*. [cit. 2012-02-26]. Dostupné z WWW: <http://cyber.felk.cvut.cz/research/theses/papers/187.pdf>

[17] Sýkora, J.: *Metody extrakce modelu z jazyka SystemC*. [cit. 2012-02-26]. Dostupné z WWW: <http://necago.ic.cz/prj/sc2vhdl/dip-20090512-rev3.pdf>

[18] Tala, D. K.: *SystemC Tutorial*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.asic-world.com/systemc/tutorial.html>

[19] *Gaussova eliminace*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.karlin.mff.cuni.cz/~tuma/2002/NLinalg2.pdf>.

[20] *Pojem kongruence a její základní vlastnosti*. [cit. 2012-02-26]. Dostupné z WWW: <www.kmt.zcu.cz/subjects/ela/kap3.doc>

[21] *GNU LGPL*. [cit. 2012-05-08]. Dostupné z WWW: <http://www.gnu.org/licenses/lgpl-2.1.html>

[22] *MSDN Academic Alliance*. [cit. 2012-05-08]. Dostupné z WWW: <http://msdn63.e-academy.com/cvut_fit>

[23] *SystemC Open Source License*. [cit. 2012-05-08]. Dostupné z WWW: <http://www.accellera.org/about/policies/SystemC_Open_Source_License_v3.1-f.pdf>

[24] *Download Qt*. [cit. 2012-04-27]. Dostupné z WWW: <http://qt.nokia.com/downloads>

[25] *Qt Visual Studio Add-in Archive*. [cit. 2012-04-27]. Dostupné z WWW: <ftp://ftp.qt.nokia.com/vsaddin/>

[26] *Qt Libraries Archive*. [cit. 2012-04-27]. Dostupné z WWW: <ftp://ftp.qt.nokia.com/qt/source/>

[27] *SystemC*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.accellera.org/downloads/standards/systemc>

[28] *SystemC - Hierarchical Channels*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.doulos.com/knowhow/systemc/tutorial/hierarchical_channels/>

[29] *A Brief Introduction to SystemC*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.doulos.com/knowhow/systemc/tutorial/introduction/>

[30] *SystemC - Primitive Channels*. [cit. 2012-02-26]. Dostupné z WWW: <http://www.doulos.com/knowhow/systemc/tutorial/primitive_channels/>

[31] *SytemC Tutorials*. [cit. 2012-02-26]. Dostupné z WWW: <http://sclive.wordpress.com/category/sytemc-tutorials/>

[32] *Download Microsoft Visual Studio 2008 Professional Edition*. [cit. 2012-04-27]. Dostupné z WWW: <http://www.microsoft.com/en-us/download/details.aspx?id=3713>

[33] *Microsoft Visual Studio 2010*. [cit. 2012-04-27]. Dostupné z WWW: <http://www.microsoft.com/cze/msdn/vstudio/2010/>

[34] *Wolfram Mathematica*. [cit. 2012-04-27]. Dostupné z WWW: <http://www.wolfram.com/>

PŘÍLOHA A

Instalační příručka

Jak simulační, tak vizualizační aplikace jsou spustitelné soubory (`*.exe`), není je před použitím potřeba nijak kompilovat. Pro úspěšné spuštění vizualizační aplikace je jen potřeba mít v adresáři s aplikací soubory `QtCore4.dll`, `QtGui4.dll` a `QtXml4.dll`.

Po případné úpravě zdrojových souborů je ale potřeba danou aplikaci zkompilovat, aby se změny ve spustitelném souboru projevily. Následuje popis všech náležitostí, které jsou potřeba k otevření a zkompilování obou projektů.

A.1 Simulace

- Zdrojové kódy jsou napsané v C++, jako vývojové prostředí bylo použito Microsoft Visual Studio 2008 Professional Edition [32] - verze šířená v rámci MSDNAA [22] (Microsoft Developer Network Academic Alliance). Přeloženo překladačem ve vývojovém prostředí. Novější verze Visual Studia je dostupná na webu [33].
- Knihovna SystemC ve verzi 2.2.0 dostupná na webu [27]. Licence je SystemC Open Source License [23].
- Propojení vývojového prostředí se SystemC a nastavení projektu.

Nejprve je potřeba nainstalovat vývojové prostředí. Potom stáhnout knihovnu SystemC. Ve Visual Studiu je možné knihovnu SystemC otevřít jako projekt (knihovna obsahuje zdrojové soubory i příslušný soubor pro otevření ve Visual Studiu). Je potřeba SystemC zkompilovat, než jej bude možné v projektech používat. Ve vývojovém prostředí je dále potřeba přidat cestu ke zdrojovým souborům SystemC a ke zkompilovanému projektu SystemC. Kompletní postup je detailně popsán v [3]. Pak stačí vytvořit prázdný nový projekt, nebo otevřít existující projekt. Pro projekt, který používá funkce, šablony, atd. ze

SystemC, je potřeba nastavit další náležitosti. Vše je opět detailně popsáno v [3]. Po provedení veškerého nastavení je již možné ze zdrojových souborů sestavit zkompilováním projektu spustitelnou aplikaci.

A.2 Vizualizace

- Zdrojové kódy jsou napsané v C++, jako vývojové prostředí bylo použito Microsoft Visual Studio 2008 Professional Edition [32] - verze šířená v rámci MSDNAA [22] (Microsoft Developer Network Academic Alliance). Přeloženo překladačem ve vývojovém prostředí. Novější verze Visual Studia je dostupná na webu [33].
- Knihovna Qt ve verzi 4.7.1 [26]. Knihovna je použita pod licencí GNU LGPL v2.1 [21] (GNU Lesser General Public License). Poslední verze 4.8.1 je ke stažení na webu [24].
- Qt doplněk do Microsoft Visual Studia ve verzi 1.1.7 [25]. Nejnovější verze 1.1.10 je ke stažení na webu [24].

Nejdříve je potřeba nainstalovat vývojové prostředí. Poté nejprve samotnou knihovnu Qt a až potom doplněk do Visual Studia. Pak je ve vývojovém prostředí možné otevřít zdrojové soubory i se všemi náležitostmi (např. grafickou reprezentací formulářů) a zkompilovat je do výsledné spustitelné aplikace.

PŘÍLOHA B

Uživatelská příručka

Po spuštění (viz instalační příručka v příloze A) aplikace vizualizace (aplikace s grafickým uživatelským rozhraním) se otevře hlavní okno (viz obrázek B.2), v jehož pravé straně je umístěna legenda. V té jsou ke konkrétním operacím přiřazeny příslušné barvy. Dále jsou zde k jednotlivým barevným šipkám (symbolizují zápis a čtení do/z kanálu) přiřazeny typy hodnot, které prochází komunikačními kanály. Nakonec jsou barevně zobrazené jednotlivé stavy, jakých mohou aritmetické jednotky nabývat.

V horejší části okna je umístěna lišta s tlačítky, která slouží k ovládání a nastavení. Jednotlivá tlačítka mají následující funkce:

- „**Otevřít...**“ - při kliknutí na tlačítko se otevře okno pro výběr souboru se strukturou XML, ve kterém je uložen běh simulace. Po vybrání souboru tlačítkem „**Otevřít**“ se ve volné části okna aplikace zobrazí struktura aritmetických jednotek propojených komunikčními kanály, jejíž velikost odpovídá té soustavě, která byla řešena při vytváření příslušného XML souboru.

- „**Nastavení...**“ - kliknutí na toto tlačítko vyvolá otevření formuláře s nastavením (viz obrázek B.1) pro simulační aplikaci. V horejší části okna je místo pro soubor, ve kterém jsou vstupní data (soustavy lineárních kongruencí) pro simulaci. Tento soubor je simulační aplikaci předáván jako povinný parametr a je potřeba vybrat soubor v nastavovacím formuláři, jinak se tlačítko „SIMULACE“ nezaktivní. Je na výběr mezi existujícím souborem a nebo vygenerováním nového. K tomu slouží pole pro vyplnění hodnot použitých při generování nového souboru se soustavami. Ve střední části okna je možnost zapnout funkci vytváření souboru pro vizualizaci při běhu simulace. Ve spodní části okna lze pak nastavit, jak dlouho bude trvat každá jednotlivá operace při řešení soustav. Ve formuláři je možné kliknout na některé z následujících tlačítek:

  - „**Procházet...**“ - při stisku tohoto tlačítka se otevře okno pro výběr existujícího souboru s daty pro simulaci. Po vybrání souboru tlačítkem „**Otevřít**“ se vybraný soubor zobrazí v horní řádce okna.
  - „**Generovat**“ - podle vyplněných parametrů se vygeneruje soubor a naplní příslušnými daty. Ten je pak zobrazen v horní řádce okna.
  - „**OK**“ - při kliknutí dojde k potvrzení hodnot vyplněných v nastavovacím formuláři a k jeho zavření.
  - „**Storno**“ - pouze zavře nastavovací formulář, aniž by se změny uložily.

- „**SIMULACE**“ - toto tlačítko slouží ke spuštění simulace. Je nutné, aby byla simulační aplikace umístěna ve stejném adresáři, jako vizualizační aplikace. Při stisku tohoto tlačítka se otevře konzolové okno, ve kterém se bude zobrazovat průběh simulace. Té jsou automaticky předány potřebné parametry, které jsou nastaveny ve formuláři pod tlačítkem „Nastavení“. Simulace v okně informuje, jakou řeší soustavu, jaký řeší krok eliminace, zda bylo možné soustavu vyřešit, jaké je řešení, zda je řešení správné a jaká je hodnota determinantu. V případě, že byla zapnuta funkce vytváření souboru pro vizualizaci, informuje také o vytváření tohoto souboru. Po dokončení řešení všech soustav ze vstupního souboru (a případného vytvoření vizualizačního souboru) čeká okno na zavření stisknutím libovolné klávesy.

- „**Spustit**“ - kliknutí na tlačítko způsobí zahájení grafického zobrazování běhu simulace. Dojde k vykreslování operací a komunikace mezi jednotkami pomocí barev a textu v zobrazené struktuře. Budou se vykreslovat všechny příslušné operace vždy pro daný zobrazený čas. K přechodu z jednoho zobrazeného času do dalšího (a tím pádem překreslování operací) dojde po uplynutí takového intervalu, jaký je nastaven (viz „Trvání jednoho taktu“). Po kliknutí se tlačítko změní na „Pozastavit“.

- „**Pozastavit**“ - na toto tpačítko lze kliknout v případě, že byla vizualizace spuštěna. Při stisknutí tlačítka dojde k pozastavení běhu vizualizace, neukončí se celá vizualizace. Vizualizace až do stisku některého dalšího tlačítka stále zobrazuje stejný časový okamžik.

- „**Krok »** “ - kliknutí na toto tlačítko způsobí přechod z aktuálně zobrazeného času na čas následující a vykreslení k němu příslušejících operací. S každým kliknutím se zobrazí všechny operace z jednoho následujícího taktu v simulaci. Jde o krokování vpřed. Tímto způsobem lze projít celou vizualizaci, ale je potřeba kliknout postupně tolikrát, kolik taktů trvala simulace. Dokud není kliknuto na jiné tlačítko, je stále zobrazen stejný čas.

- „ **« Krok**“ - tlačítko slouží pro krokování směrem vzad, tedy proti běhu simulace. Při krokování vpřed se s každým krokem ukládá historie a tedy tolikrát, kolikrát bylo krokováno vpřed, je možné krokovat i nazpět. Opět zůstává zobrazen stejný čas, dokud není stisknuto jiné tlačítko.

- „**Reset**“ - na toto tlačítko lze kliknout v jakékoli chvíli během vizualizace. Celá vizualizace se zastaví, zobrazená struktura se vrátí do původního zobrazení (těsně po otevření souboru) a bude možné spustit vizualizaci od úplného začátku znovu. K předtím rozeběhlé vizualizaci se už není možné žádným způsobem vrátit.

- „**Trvání jednoho taktu**“ - šipkami nahoru a dolů je vybrán časový interval v sekundách, který slouží při spuštění vizualizace tlačítkem „Spustit“. Tento interval slouží jako časový signál pro spuštěnou vizualizaci - interval, který je nastaven, slouží jako doba, po kterou bude zobrazen jeden čas při vizualizovaném běhu. Vždy po uplynutí tohoto intervalu se vykreslí všechny operace z dalšího taktu v simulaci.

V horejší liště jsou umístěny mezi tlačítky ještě popisy udávající:

- **ns** - udává dobu v nanosekundách, která uplynula od začátku zahájení simulace. Jeden takt trvá 20ns.

- **mod** - ukazuje, v jakém modulu je právě vizualizovaná soustava řešena.

V průběhu vizualizace je možné stisknout tlačítko „SIMULACE“ a nechat simulaci vyřešit soustavy (např. pro porovnání s výsledky zobrazenými ve vizualizované struktuře). Nicméně pokud je zapnuta funkce ukládání běhu simulace do souboru pro vizualizační aplikaci a tento nastavený soubor je shodný se souborem, ze kterého probíhá vizualizace, je zobrazena chyba s tím, že není možné přepisovat otevřený soubor a simulace se nespustí. Stačí v nastavení vypnout funkci na vytváření vizualizačního souboru a simulaci bude možné spustit.

Pokud je potvrzováno nastavení a nebo pokud je generován nový soubor pro simulaci a některé z polí není správně vyplněno, je zobrazena chyba s detailním popisem, co bylo její příčinou. Je nutné chybnou hodnotu opravit, aby bylo možné soubor vygenerovat nebo nastavení potvrdit.

V případě, že je v nastavovacím formuláři vybráno generování nového souboru a je kliknuto na potvrzení formuláře, aniž by předtím bylo kliknuto na tlačítko pro generování nového souboru, je zobrazen dotaz, zda má být skutečně potvrzeno nastavení bez generování souboru. Pokud bude nastavení v tomto případě potvrzeno bez generování nového souboru, soubor pro simulaci bude vybrán ten, který byl použit naposledy.

Formulář s nastavením si pamatuje hodnoty, se kterými byl naposledy potvrzen. To znamená, že v případě dalšího spuštění a otevření nastavovacího

formuláře budou jednotlivá pole vyplněna hodnotami, která byla ve formuláři při jeho posledním bezchybném zavření.

Soubor s XML strukturou se ukládá během simulace do stejného místa (adresáře) na disku, kde se nachází soubor se vstupními daty (soustavami lineárních kongruencí) pro simulaci. Jeho název je takový, jaký byl v nastavení vyplněn.

Nový soubor, do kterého se generují soustavy lineárních kongruencí podle zadaných parametrů je uložen ve stejném adresáři, v jakém je vizualizační aplikace. Jeho název je takový, jaký je vyplněn před generováním.

B.1 Formát zdrojových dat

Soubor, ve kterém jsou soustavy, které má simulační aplikace řešit, má jen jednu podporovanou podobu. Simulace kontroluje tento soubor a v případě neshody s touto podobou zobrazí chybu a dál nepokračuje. Před každou soustavou musí být zadán modul, ve kterém se má soustava řešit. Modul je uveden klíčovým slovem `mod` a za mezerou následuje prvočíslo, maximálně $2^{31} - 1$. Hodnoty ze soustavy jsou zapsané v matici, hodnota může být maximálně $2^{31} - 2$. Vždy je v řádce za sebou tolik hodnot oddělených mezerami, kolik má soustava sloupců. Řádků je tolik, kolik je řádků v soustavě, tedy n a sloupců je $n + 1$. Struktura souboru může obsahovat více soustav, všechny musí mít stejný rozměr. Každá soustava je vždy zapsána shodným způsobem. Kdekoliv v souboru může být vložena prázdná řádka, nebo i více prázdných řádků např. kvůli lepší přehlednosti (prázdná řádka znamená řádka bez jediného znaku, řádka s mezerou už není prázdná řádka). Řádky, kde je první znak `/` se berou jako komentář a pokud bude před každou řádkou soustavy (i v řádce s modulem) znak `/`, tato soustava se řešit nebude. Soubor může vypadat následovně:

```
/prvni soustava
mod 7
1  0  5
6  2  1

mod 13
2  11  7
0  4  12

/mod 5
/3  4  2
/1  3  0
```

Tento konkrétní soubor obsahuje tři soustavy lineárních kongruencí o dvou řádcích a třech sloupcích. Při spuštění simulace s tímto souborem by byly řešeny první dvě soustavy, každá v příslušném modulu.

B.2 Parametry simulační aplikace

Simulační aplikace může být spuštěna buď z vizualizační aplikace, nebo samostatně. Pokud je spuštěna z vizualizační aplikace, parametry jsou nastaveny v nastavovacím formuláři, který kontroluje chybná zadání. V případě, že jsou parametry korektní, jsou automaticky předány simulační aplikaci, která s nimi pracuje.

V případě spuštění simulační aplikace přímo, bez vizualizační aplikace, musí být parametry zadány jako argumenty příkazové řádky. Je povinný pouze jeden parametr, ostatní jsou volitelné. Povinný parametr je soubor se soustavami, které se mají řešit. Za všemi parametry následuje mezera a konkrétní hodnota parametru. Jsou následující:

- `-in` : povinný; jméno souboru, ve kterém jsou soustavy k řešení,
- `-viz` : volitelný; jméno souboru k ukládání dat pro vizualizační aplikaci, pokud není zadáno, soubor pro vizualizaci není vytvořen,
- `-ti` : volitelný; doba trvání (počet taktů) multiplikativní inverze během výpočtu, pokud není zadáno, hodota je `1.4n`, tedy 1,4 x počet bitů modulu,
- `-tn` : volitelný; doba trvání (počet taktů) negace během výpočtu, pokud není zadáno, hodota je `1`,
- `-tm` : volitelný; doba trvání (počet taktů) násobení během výpočtu, pokud není zadáno, hodota je `1n`, tedy 1 x počet bitů modulu,
- `-ta` : volitelný; doba trvání (počet taktů) sčítání během výpočtu, pokud není zadáno, hodota je `1`.

Simulační aplikace může být spuštěna s parametry například následujícím způsobem, přičemž nezáleží na pořadí jednotlivých zadaných parametrů:

```
simulace.exe -in s.sim -viz v.viz -ti 1.4n -tn 1 -tm 1n -ta 1
```

B.3 Legenda vizualizace

Hodnoty jsou ve vizualizaci vždy reprezentovány číslem buď zobrazeným v příslušné aritmetické jednotce (hodnota přečtená z komunikačního kanálu) nebo v komunikačním kanálu (hodnota odeslaná sousední jednotce). Barevně je potom rozlišeno, jaký typ hodnoty je odesílán nebo přijímán.

Pod označením `IR` je myšlen výsledek součinu multiplikativní inverze pivota a hodnoty uložené v příslušné aritmetické jednotce. Tato hodnota se distribuje vždy v konkrétním sloupci nahoru a/nebo dolů. Tato hodnota je vždy zobrazena v horní části aritmetické jednotky.

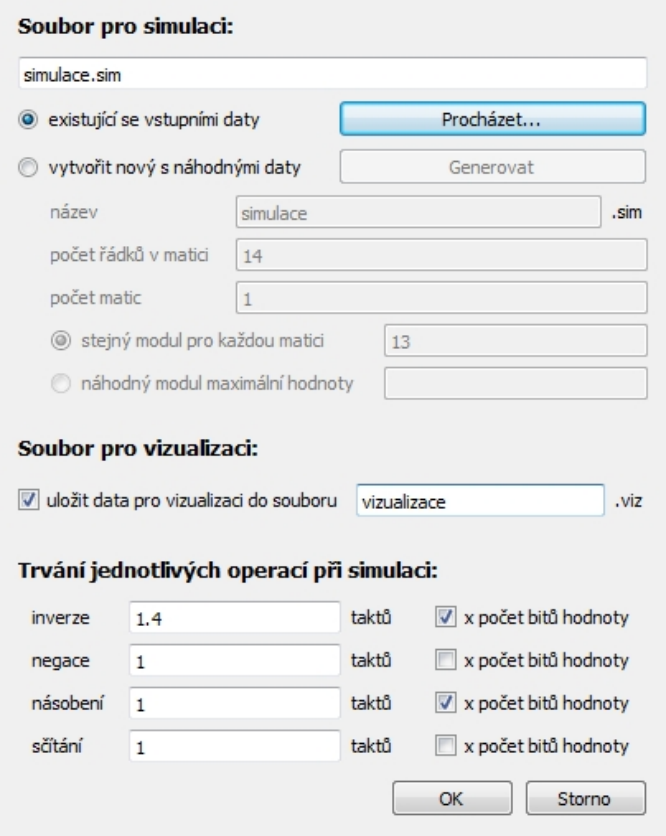
Hodnota označená `'a'` je hodnota příslušející dané aritmetické jednotce. Na začátku jsou to původní hodnoty zadané ve vstupním souboru. Tato hodnota je vždy zobrazena ve středu aritmetické jednotky.

Pod označením `WR` je myšlena hodnota vypočtená v aritmetické jednotce v příslušném eliminačním kroku a je poslána směrem doleva. S touto hodnotou se pak v dalším eliminačním kroku počítá jako s hodnotou uloženou v jednotce, tedy jako s hodnotou `'a'`.

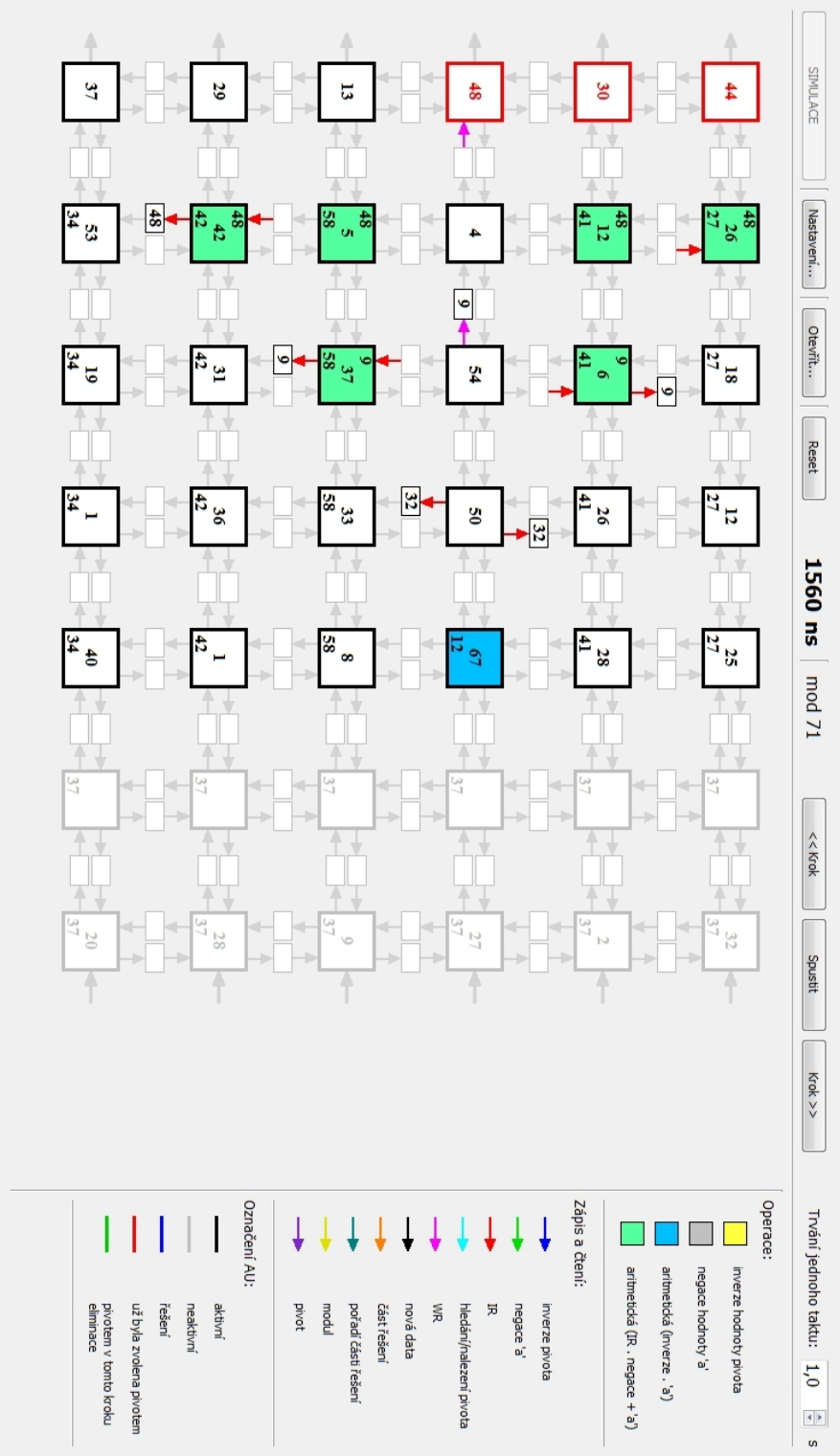
Hodnota přijatá zleva (`negace 'a'` nebo `inverze pivota`) je vždy zobrazena ve spodní části aritmetické jednotky.

V prvním sloupci se posílá hodnota typu `hledání/nalezení pivota`. Pokud je hodnota rovna nule, jedná se o hledání pivota, pokud je rovna jedné, jedná se o oznámení, že byl pivot nalezen.

Hodnota označena `nová data` znamená načítání další soustavy zprava. Pokud je tato hodnota zobrazena ve spodní části jednotky, jedná se o modul, ve kterém se bude následující soustava řešit, pokud je hodnota zobrazena uprostřed, jedná se o hodnotu přímo ze soustavy zadané ve vstupním souboru.



Obrázek B.1: Nastavovací okno vizualizační aplikace



Obrázek B.2: Hlavní okno vizualizační aplikace

PŘÍLOHA C

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnými soubory
  - `simulace` ..... spustitelná aplikace
  - `vizualizace` ..... spustitelná aplikace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `měření` ..... adresář s naměřenými hodnotami
- `text` ..... text práce