Sem vložte zadání Vaší práce.

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra počítačových systémů

Diplomová práce

Architektura řešiče soustav lineárních rovnic v modulární aritmetice

Bc. Michal Daňhelka

Vedoucí práce: Ing. Jiří Buček

5. května 2015

Poděkování

Na tomto místě bych rád poděkoval Ing. Jiřímu Bučkovi a prof. Ing. Róbertu Lórenczovi, CSc. za trpělivost, pomoc a připomínky k této práci. Dále bych rád poděkoval rodině za pochopení a trpělivost.

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, zejména skutečnost, že České vysoké učení technické v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 5. května 2015 .....

Odkaz na tuto práci

Daňhelka, Michal. *Architektura řešiče soustav lineárních rovnic v modulární aritmetice*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2015.

Abstract

This thesis deals with analysis of architecture used for solving linear equations in modular arithmetic. Works also deals with design and simulation one such architecture. Based on design of this architecure is created software simulation. This simulation is evaluated. This evalution is based on design assumptions. Standalone application is created to show the steps of this simulation. This visualization helps to understand behaviour of the designed architecture.

Keywords linear equation, congruence, modular arithmetics, solving equations, architecture, SystemC, visualization, XML, Gauss-Jordan elimination method, Gauss-Jordan-Rutishauser elimination method

Abstrakt

Tato práce se zabývá analýzou architektur určených k řešení soustav lineárních rovnic v modulární aritmetice. Dále se práce zabývá návrhem a simulací takovéto architektury. Navržené architektura je na základě této simulace zhodnocena z hlediska jejího chování a podrobena měření závislostí plynoucí z vlastností architektury. Dále je vytvořena aplikace zobrazující chování architektury během jednotlivých částí hledání řešení soustavy rovnic.

Klíčová slova lineární rovnice, kongruence, modulární aritmetika, řešení rovnic, architektury, SystemC, vizualizace, XML, Gauss-Jordanova eliminační metoda, Gauss-Jordanova-Rutishauserova eliminační metoda

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Úkolem práce je analýza architektur využitelných k řešení soustav lineárních rovnic v modulární aritmetice. Cílem práce je návrh, analýza a simulace takovéto architektury. Na návrhu bude vytvořena simulační aplikace, která bude dodržovat chování navržené architektury. Na základě této simulace bude podrobena architektura podrobena analýze a popsáno její chování. Dále bude vytvořena aplikace, která bude vizualizovat jednotlivé kroky vedoucí k nalezení řešení soustavy lineárních rovnic.

Práce je složená z několika částí. První část se zabývá analýzou matematických struktur používaných při hledání řešení soustav lineárních rovnic. Další část se zabývá metodami hledání řešení těchto soustav. Další část se již zabývá řešením lineárních kongruencí rovnic.

Následující část práce se zabývá analýzou existujících hardwarových architektur a jejich zhodnocením.

Na základě informací získaných z analýzy a hledisek návrhu je navržena architektura použitelná k hledání řešení soustav lineárních rovnic v modulární aritmetice.

V části práce zabývající se realizací je realizována simulace simulující chování navržené architektury. Tato simulace je využita k pochopení výpočetních závislostí a závislostí spojených s použitím omezené šířky paměťové sběrnice. Samotná realizace simulace navržené architektury je provedena pomocí knihovny SystemC jazyka C++.

Dále je vytvořena vizualizační aplikace zobrazující jednotlivé kroky výpočtu a práci s pamětí.

V závěru jsou zhodnoceny navržená architektura, realizovaná simulace a naměřené výsledky ze simulace chodu architektury. Dále je zde přítomen manuál pro simulaci, vizualizační aplikaci a jsou zde popsány další vytvořené nástroje, které slouží k ověření a zpracování naměřených výsledků.

Kapitola 1

Analýza matematických struktur

Následující sekce se zabývá potřebnými matematickými strukturami a postupy nalezení řešení soustavy rovnic.

Sekce 1.2 a 1.5 se zabývají potřebnými matematickými strukturami. Sekce 1.3, 1.4, 1.6 se zabývají postupy řešení soustav rovnic.

1.1 Motivace

Řešení soustav lineárních rovnic je jedním ze základních matematických problémů. Vyskytuje se v mnoha vědních disciplínách.

Soustavy lineárních rovnic lze využít například pro vyčíslování chemických reakcí [22], pro kryptografii a kryptoanalýzu, pro eliminaci zaokrouhlovacích chyb při výpočtech v architekturách s pohyblivou plovoucí čárkou a mnoho dalších využití.

Soustavy lineárních rovnic lze využít pro eliminaci zaokrouhlovacích problémů při hledání přesného řešení výpočtu. Tohoto se využívá při použití *float* architektury. Nejdříve převedeme číslo z *float* typů do celočíselných datových typů, vyřešíme několik soustav lineárních rovnic v patřičných modulech a poté částečná řešení složíme dohromady pomocí Čínské věty o zbytcích. Více informací lze nalézt například v článku [7].

1.2 Soustavy rovnic

Nyní budeme definovat základní matematické struktury.

Lineární rovnicí nazvěme rovnici ve tvaru $ax = b$, pak pro $a \neq 0$ existuje řešení $x = \frac{b}{a}$.

Soustavou m rovnic o n neznámých nazvěme soustavu lineárních rovnic ve tvaru:

$$\begin{array}{ccccccc} a_{1,1}x_1 & + & a_{1,2}x_2 & + \cdots + & a_{1,n}x_n & = & b_1 \\ a_{2,1}x_1 & + & a_{2,2}x_2 & + \cdots + & a_{2,n}x_n & = & b_2 \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{m,1}x_1 & + & a_{m,2}x_2 & + \cdots + & a_{m,n}x_n & = & b_m \end{array}$$

kde $x_1, \cdots, x_n$ jsou neznámé, $a_{i,j}$ jsou koeficienty soustavy a čísla $b_1, \cdots, b_n$ jsou absolutní členy rovnice.

Dále v textu, nebude-li uvedeno jinak, bude vždy uvažováno o lineárních rovnicích a jejich soustavách.

Takovou soustavu lze zapsat v maticovém zápise složeném z matice soustavy A, vektoru neznámých $\vec{x}$ a vektoru pravých stran $\vec{b}$.

Matice soustavy A má tvar:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} \end{pmatrix}$$

Vektor neznámých $\vec{x}$ má tvar:

$$\vec{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

Vektor pravých stran $\vec{b}$ má tvar:

$$\vec{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$$

Pro celkový sdružení zápis se používá tzv. rozšířený maticový zápisu, ve kterém se zapíše matice soustavy a vektor pravých stran, oddělených čarou. Výsledný zápis pak vypadá takto:

$$\left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array}\right)$$

V práci bude dále využíván výhradně tento zápis.

1.2.1 Řešení soustavy rovnic

Řešení soustavy rovnic je ohodnocení proměnných $x_1, x_2, \cdots, x_n$, takové že každá rovnice soustavy je splněna. Množina těchto ohodnocení se nazývá množina řešení. Triviální složitost nalezení řešení soustavy lineárních rovnic je kubická $O(n^3)$. Této složitosti lze dosáhnout například Gaussovou eliminační metodou popsanou v sekci 1.3.1.

1.2.1.1 Podmínky pro řešení soustavy

V závislosti na soustavě rovnic může množina řešení soustavy obsahovat:

1. nekonečně mnoho řešení
2. jedno jedinečné řešení
3. žádné řešení

Pokud je matice soustavy singulární, $\det A = 0$, má soustava buď nekonečně mnoho nebo žádné řešení.

Pro to aby soustava měla nekonečně mnoho řešení, musí být některá z rovnic k-násobkem ($k \neq 0$) jiné rovnice soustavy.

Jedná-li o soustavu přeurčenou, tak nemá žádné řešení. Přeurčená soustava je taková, která obsahuje více rovnic než neznámých. Determinant matice soustavy je také $\det A = 0$, ale zatímco k-tý řádek matice soustavy je násobkem l-tého řádku matice soustavy ($k \neq l$), tak k-tý člen z vektoru pravých stran není násobkem l-tého člena vektoru pravých stran, jako je tomu u soustav s nekonečně mnoho řešeními.

Pro to aby soustava měla jedno jedinečné řešení, musí být matice soustavy regulární. Musí tedy platit, že $\det A \neq 0$. Rovnice v této soustavě jsou nezávislé a každá rovnice nese novou informaci o proměnné [13].

1.3 Metody řešení soustav lineárních rovnic

Pro nalezení řešení soustav lineárních rovnic existuje mnoho metod. Tyto metody lze rozdělit na metody přímé a iterační metody.

Mezi nejznámější přímé metody patří Hornerova metoda, Gaussova eliminace, LU dekompozice a další. Přímé metody se vyznačují tím, že v konečném čase naleznou přesné řešení dané soustavy, neuvažujeme-li zaokrouhlovací chyby.

Základním rozdílem přímých a iteračních metod, je že iterační metody hledají přibližná řešení s danou přesností. Myšlenkou iteračních metod je,

že nalezneme vhodné zobrazení takové, které povede k tomu, že ze stávajícího řešení pomocí zobrazení nalezneme jiné řešení, které je blíže přesnému řešení. Získáme tak vztah, při kterém se námi získané přibližné řešení soustavy rovnic limitně blíží přesnému řešení. Mezi nejznámější iterační metody patří Richardsova metoda, Jacobiho metoda, Gauss-Seidlova metoda nebo superrelaxační metoda (SOR). Konkrétní popis těchto metod lze nalézt například v [12],[23].

V práci budou dále uvažovány jen přímé metody řešení soustav lineárních rovnic, konkrétně Gaussova eliminační metoda a její úpravami.

1.3.1 Gaussova eliminační metoda

Gaussova eliminační metoda je přímá metoda založená na LU rozkladu matice soustavy. Soustavu rovnic zapsanou v rozšířeném maticovém tvaru

$$\left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{nn,} & b_n \end{array}\right)$$

převedeme pomocí operací, které nemění bázi řešení na tvar horní trojúhelníkové matice:

$$\left(\begin{array}{cccc|c} a'_{1,1} & a'_{1,2} & \cdots & a'_{1,n} & b'_1 \\ 0 & a'_{2,2} & \cdots & a'_{2,n} & b'_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & a'_{n,n} & b'_n \end{array}\right)$$

Operace, které nemění bázi řešení jsou následující:

- Výměna řádků matice mezi sebou.
- Vynásobení řádku matice nenulovou konstantou.
- Přičtení libovolného násobku libovolného řádku matice k jinému řádku matice.

Z tohoto tvaru, lze již pomocí zpětného dosazení vyjádřit řešení soustavy rovnic.

Gauss-Jordanova eliminační metoda se od Gaussovy eliminační metody liší tím, že převede matici do redukovaného stupňovitého tvaru, který vznikne právě zpětným dosazením do horní trojúhelníkové matice. Samotná výsledná soustava, v rozšířeném maticovém tvaru, je pak složená z jednotkové matice soustavy a vektoru řešení soustavy. Bližší informace o rozdílech těchto metod lze získat z článku *Gauss-Jordan reduction: a brief history*[1].

Existuje několik modifikací této metody, pro práci je důležitá především Gauss-Jordanova eliminace s pivotem a Gauss-Jordan-Rutishauserova eliminace. Tyto metody jsou popsány níže v sekci 1.4.

1.4 Možné úpravy a varianty Gaussovy eliminační metody

Samotná Gaussova eliminační metoda je pro hardwarovou realizaci méně vhodná, protože vyžaduje zpětné dosazení do rovnic. Pro takovouto realizaci je proto vhodnější využit některou z jejích modifikací. Při využití modifikovaných metod odstraníme jak nutnost zpětného dosazení, tak i nutnost udržovat v paměti celou matici soustavy.

1.4.1 Gauss-Jordanova metoda s pivotizací

Přidáme-li možnost záměny dvou rovnic, získáme Gauss-Jordanovu metodu s částečnou pivotizací a přidáme-li ještě možnost přejmenování proměnných, získáme Gauss-Jordanovu metodu s úplnou pivotizací.

Níže vidíme, jak vypadá matice soustavy s vektorem pravých stran po provedení Gauss-Jordanovy metody s pivotizací.

$$\left(\begin{array}{ccc|c} 1 & 0 & 0 & 2 \\ 0 & 1 & 0 & 3 \\ 0 & 0 & 1 & 1 \end{array}\right)$$

1.4.2 Gauss-Jordanova-Rutishauserova metoda

Gauss-Jordanova metoda pracuje s maticí soustavy a vektorem pravých stran. Z původní matice soustavy se postupným výpočtem stává jednotková matice, která je ve výsledném řešení nadbytečná. Tuto jednotkovou matici tedy není nutné dále udržovat v paměti a proto můžeme po každém eliminačním kroku pracovat s maticí o 1 sloupec menším. Tohoto využívá Gauss-Jordanova-Rutishauserova eliminační metoda.

Níže vidíme, jak vypadá matice soustavy po prvním eliminačním kroku složeném z vynásobení prvního řádku a odečtení násobku prvního řádku od ostatních sloupců. Tečky v zápise znázorňují hodnoty z jednotkové matice, která je pro výpočet nepodstatná.

$$\left(\begin{array}{ccc|c} \cdot & 10 & 8 & 1 \\ \cdot & 0 & 6 & 6 \\ \cdot & 11 & 8 & 2 \end{array}\right)$$

1.5 Modulární aritmetika

Neomezíme-li se pouze na tělesa s reálnými čísly, může řešit soustavy lineárních rovnic nad dalšími tělesy. Pro využití v počítačové bezpečnosti je to především těleso $\mathbb{GF}(p^{\mathrm{k}})$. Mezi významné podmnožiny tohoto tělesa patří podtěleso $\mathbb{GF}(p)$, pro $k = 1$, p je prvočíslo a podtěleso $\mathbb{GF}(2^{\mathrm{k}})$ pro $p = 2$. Při práci nad tělesem $\mathbb{GF}(p^{\mathrm{k}})$ pak použijeme operace modulární sčítání, modulární odčítání, modulární násobení a modulární inverze. Bližší informace, jak jsou tyto operace definovány, lze nalézt například v [11], [17], [5], [6].

1.5.1 Lineární kongruence

Lineární kongruencí nazvěme ekvivalenci ve tvaru $ax \equiv b \pmod{m}$. Tato ekvivalence má řešení za podmínky, že $d \mid b$, kde $d = \gcd(a, m)$ (největší společný dělitel). Omezíme-li se na situaci kdy $d = 1$, pak toto řešení je $x \equiv b \cdot a^{-1} \pmod{m}$.

Nyní můžeme definovat soustavy lineárních kongruencí a využít stejné metody pro jejich řešení jako u tělesa reálných čísel, jen s operacemi nad odpovídajícími tělesy.

Pro zápis je opět vhodné využít rozšířenou maticovou formu. Abychom nemuseli u každého koeficientu zapisovat daný modul, přidáme vyznačení modulu k rozšířené maticové formě. Níže uvedený zápis znamená, že všechny koeficienty v této soustavě jsou v modulu m. Tohoto zápisu bude dále využíváno pro zápis soustavy lineárních kongruencí. Takováto soustava zapsaná v této formě je například tato soustava:

$$\left(\begin{array}{cccc|c} a_{1,1} & a_{1,2} & \cdots & a_{1,n} & b_1 \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} & b_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & a_{n,2} & \cdots & a_{n,n} & b_n \end{array}\right), m$$

Všechny koeficienty soustavy lineárních kongruencí $a_{1,1}, \cdots, a_{n,n}, b_1, \cdots, b_n$ jsou pak v daném modulu m.

V práci bude využíván zápis $a = |b|_m$ vyjadřující $a = b \bmod m$, tj. že a je nejmenší nezáporný zbytek po dělení m.

1.6 Postup nalezení řešení soustavy rovnic

Následující kapitoly se budou zabývat pouze hledáním řešení soustavy rovnic v tělese $\mathbb{GF}(p^k)$, nebude-li řečeno jinak.

Pro vyřešení soustavy potřebujeme postupně normalizovat všechny řádky matice a eliminovat jimi ostatní řádky. Normalizace řádku matice se provede vynásobením řádku inverzí pivota. Eliminace se provede odečtením e-násobku normalizovaného řádku. Stejné úpravy při tom musíme aplikovat i na vektor pravých stran.

Během výpočtu lze získat determinant matice. Determinant je roven součinu pivotů nalezených v průběhu řešení soustavy. Pokud byl pivot nalezen mimo pořadí, je nutné zaměnit znaménko determinantu. Znaménko determinantu se řídí znaménkem (paritou) permutace řádků, ve kterých byly nalezeny pivoty.

Pro nalezení řešení soustavy rovnic o n neznámých je potřeba n eliminačních kroků, tyto kroky budeme dále nazývat fáze výpočtu.

1.6.1 Normalizace řádku pivota

Normalizace řádků (vynásobení řádku inverzí pivota) pivota ve fázi (eliminačním kroku) k lze provést algoritmem popsaným níže:

1. $inv = |a_{k,k}^{-1}|_m$
2. **for** $j = k$ **to** $n + 1$
3. $\quad a_{k,j} = |a_{k,j} \cdot inv|_m$
4. **end for**

V prvním kroku se vypočte inverze pivota a všechny prvky se v daném řádku k vynásobí touto inverzí.

1.6.2 Eliminace ostatních řádků

Ostatní řádky je vždy potřeba eliminovat normalizovaným řádkem pivota (odečíst e-násobek). Pro všechny řádky, které nebyly normalizovány, se v dané fázi (eliminačním kroku) k provede následující algoritmus:

1. $e = a_{i,j}$
2. $a_{i,j} = 0$
3. **for** $j = k + 1$ **to** $n + 1$
4. $\quad a_{i,j} = |a_{i,j} - e \cdot a_{k,j}|_m$
5. **end for**

Matematicky se jedná o odečtení e násobku řádku pivota od řádku i, kde e je prvek ve sloupci pivota na daném řádku. Řádek $a_{k,1\ldots n+1}$ je řádek

pivota, v tomto řádku se eliminace neprovádí, protože byl v předchozím kroku normalizován.

1.6.3 Pivotizace

Pro vyřešení soustavy, je nutné provést normalizaci všech řádků. Abychom mohli postupně všechny řádky normalizovat, musíme v každé fázi vždy nalézt takový prvek, který bude splňovat následující podmínky:

1. Prvek se nachází na řádku, který ještě nebyl normalizován.
2. Prvek je nenulový.

Takovýto prvek nazvěme pivotem. Samotný proces hledání pivota se nazývá pivotizace. Pivotizace lze provádět několika způsoby. Částečná pivotizace vybere v daném sloupci vždy největší prvek pouze z daného sloupce. Úplná pivotizace hledá absolutní maximum v celé matici a zaměňuje řádky a sloupce pro dosažení co největší přesnosti. Hledání maximálního prvku se provádí pro zmenšení zaokrouhlovacích chyb při výpočtu v pohyblivé řádové čárce. V tělese $\mathbb{GF}(p^k)$ nedochází k zaokrouhlovacím chybám, proto stačí najít nenulový prvek. Částečná pivotizace je nutná a postačující. Dále v práci budeme uvažovat vždy částečnou pivotizaci.

Jestliže při částečné pivotizaci nenalezneme takový prvek, jenž by mohl být pivotem, jedná o matici singulární a soustava nemá řešení. Existuje několik možností úpravy matice po nalezení pivota, každý má určité výhody a nevýhody.

1.6.3.1 Částečná pivotizace s výměnou řádků

Tato metoda, podrobněji popsána dále, předpokládá že vždy budeme normalizovat prvky na hlavní diagonále matice soustavy. Tímto zajistíme, že nebude potřeba vektor pravých stran pře-uspořádávat. Na druhou stranu metoda předpokládá, že můžeme mezi sebou přesunout řádky matice a členy vektoru pravých stran. Samotný matematický algoritmus pro tento postup je převzat z bakalářské práce na téma *Paralelní řešič soustav lineárních rovnic v aritmetice kódů zbytkových tříd* [21].

Algoritmus postupně prochází prvky na hlavní diagonále matice soustavy, jestliže prvek na dané pozici je nulový, projde prvky v daném sloupci, pod prvkem jenž nemůže být pivotem, pokud nalezne nenulový prvek ve zbytku sloupce, prohodí tento řádek s řádkem, který v daném kroku nemůže být pivotem. Pozice tohoto řádku je dána aktuální fází (eliminačním krokem) výpočtu. Algoritmus při nalezení pivota také prohodí koeficienty

mezi sebou ve vektoru pravých stran. Díky tomuto prohození, není potřeba závěrečné přeuspořádání vektoru, takže řešení soustavy je k dispozici hned po provedení poslední fáze výpočtu.

Tato metoda je použita v HW architekturách popsaných v článku *Hardware SLE solvers: Efficient Building Blocks for Cryptographic and Cryptanalytic Applications* [20]. Další výklad k této metodě lze nalézt například v [10].

Samotný postup nalezení pivota pro normalizace prvku na hlavní diagonále lze shrnout algoritmem popsaným níže:

1. $i = k$
2. **while** $a_{i,k}$=0
3. $\quad i++$
4. $\quad$ **if** $i = k + 1$ **Error:** Pivot nenalezen, matice je singulární.
5. **end while**
6. **if** $i \neq k$
7. $\quad$ **SWAP_ROW** $(a_{i,1\ldots n+1}, a_{k,1\ldots n+1})$
8. **end if**

Pokud je prvek $a_{i,k}$, nulový, nelze ho v daném kroku použít jako pivota, a musíme postoupit ve sloupci o jednu pozici níže. Jestliže nenalezneme nenulový prvek v daném sloupci k, v řádcích, které ještě nebyly normalizovány, je matice singulární.

Jestliže jsme nalezli nenulový prvek na pozici i, zaměníme řádek $a_{i,1\ldots n+1}$ s řádkem $a_{k,1\ldots n+1}$, tím jsme dosáhli, že na hlavní diagonále na pozici $a_{i,1\ldots n+1}$ máme nenulový prvek, pivota. V řádku $a_{k,1\ldots n+1}$, se nalézá řádek s prvkem, který nemohl být pivotem.

Jedná o částečnou pivotizaci prováděnou v tělese $\mathbb{GF}(p^k)$.

1.6.3.2 Částečná pivotizace s výměnou ukazatelů

Na rozdíl od předchozí metody tato metoda nevyžaduje přesun řádků, ale je potřeba udržovat informaci o tom, které řádky již byly pivotem a které ještě ne. K tomu nejčastěji slouží pomocný registr.

Algoritmus si udržuje ukazatel na aktuální pozici řádku s pivotem, seznam řádků, které můžou být a nebo již byly pivotem (záleží na konkrétní implementaci). Dále se zaznamenává pořadí řádků, které již byly pivotem. Po normalizaci všech řádků je pak následně přeuspořádán výsledný vektor podle pořadí řádků s pivotem. Algoritmus z řádků, které můžou ještě být pivotem vybere první s nenulový prvkem v daném sloupci a podle pořadí

takto volených řádků, po skončení výpočtu provede závěrečné přeuspořádání.

Tato metoda je použita v HW architektuře popsané v článku *Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver* [4] a v diplomové práci *Simulace řešiče soustav lineárních kongruencí* [9]. Návrh architektury je popsán v článku *A modular system for solving linear equations exactly* [14].

Protože algoritmus není použit pro návrh architektury zde uvedené, není zde uveden, další informace a podrobnější popis jeho implementace lze nalézt ve zdrojích citovaných výše.

1.6.4 Algoritmický popis postupu hledání řešení

Samotný postup nalezení řešení lze rozdělit do několika stejných fází (eliminačních kroků) a každou fázi pak do několika různých částí. Pro nalezení řešení soustavy o n neznámých potřebujeme provést celkem n fází hledání pivota, každá fáze je pak složena ze 3 různých částí.

V následujícím popisu hledání řešení bude použito následující značení:

A	Soustava lineárních kongruencí, zapsaná v rozšířeném maticovém zápisu.
$\det A$	Determinant matice soustavy.
$a_{i,1\ldots n+1}$	i-tý řádek soustavy v roz. mat. zápisu.
$a_{1\ldots n,j}$	j-tý sloupec soustavy v roz. mat. zápisu.
$a_{i,j}$	Prvek soustavy na pozici $[i][j]$
k	Aktuální fáze hledání.
n	Celkový počet fází hledání.

Pro jednoduchost budeme předpokládat, že všechny operace jsou prováděny v odpovídajícím modulu.

Algoritmus 1. Nalezení řešení soustavy rovnic pomocí Gauss-Jordanovy eliminace s částečnou pivotizací výměnou řádků. Parametry: n je velikost matice, A řešená soustava zapsaná v rozšířené maticové formě v matici o rozměrech $(n+1) \times n$, modul m. Všechny operace jsou prováděny v daném modulu. Výstup: d determinant matice soustavy, $a_{n+1,j}$ vektor pravých stran s řešením soustavy.

1. $d = 1$
2. **for** $k = 1$ **to** n
   Část 1 (Pivotizace):
3. **if** $a_{k,k} = 0$
4. **for** $i = k + 1$ **to** n
5. **if** $a_{i,i} \neq 0$
6. **SWAP_ROW** $(a_{i,1\ldots n+1}, a_{k,1\ldots n+1})$
7. **goto** 12
8. **end if**
9. **end for**
10. Error: Pivot nenalezen, matice je singulární.
11. **end if**
    Část 2 (Výpočet determinantu, inverze, normalizace řádku pivota):
12. **if** $i \neq k$
13. $d = |-d|_m$
14. **end if**
15. $d = d \cdot a_{k,k}$
16. $inv = |a_{k,k}^{-1}|_m$
17. **for** $j = k$ **to** $n + 1$
18. $a_{k,j} = |a_{k,j} \cdot inv|_m$
19. **end for**
    Část 3 (Eliminace ostatních řádků):
20. **for** $i = 1$ **to** n
21. **if** $i \neq k$
22. $e = a_{i,k}$
23. **for** $j = k$ **to** $n + 1$
24. $a_{i,j} = |a_{i,j} - a_{k,j} \cdot e|_m$
25. **end for**
26. **end if**
27. **end for**
28. **end for**

Algoritmus stručně popisuje postup nalezení řešení soustavy, tak jak jej lze provádět. Jedná se o jednu z možných variant postupu nalezení řešení, samotných variant je několik.

V první části probíhá hledání pivota. Samotné hledání probíhá, tak že se projdou prvky ve sloupci $a_{1\ldots n,k}$, od řádku $a_{k,1\ldots k+1}$ do řádku $a_{n,1\ldots k+1}$. Jestliže je nalezen nenulový prvek, je řádek s tímto prvek zaměněn s řádkem $a_{k,1\ldots n+1}$. Pokud takový prvek není nalezen, nelze nalézt řešení této soustavy protože byla porušena některá z podmínek definovaná v 1.2.1.1. Dále je na základě toho zda byly v předchozím kroku zaměněny řádky upraveno znaménko determinantu.

V druhé části je aktuální determinant vynásoben pivotem. Dále je vypočtena multiplikativní inverze pivota a tou je následně vynásoben řádek $a_{k,1\ldots n+1}$. Tím jsme tento řádek normalizovali.

Ve třetí části jsou eliminovány ostatní řádky. Pro každý řádek je uložen prvek na pozici $a_{i,k}$. V každém kroku vypočteme prvek na pozici $a_{i,j}$, tak že od prvku na pozici $a_{i,j}$ odečteme součin prvku $a_{k,j}$ a prvku $a_{i,k}$. Tímto eliminujeme prvky řádku i.

Jakmile jsou eliminovány všechny řádky, následuje další fáze (eliminační krok).

1.7 Příklad nalezení řešení soustavy lineárních kongruencí

Pro ilustraci kroků výpočtu a chodu algoritmu je níže uveden příklad pro soustavu lineárních kongruencí o 3 neznámých. Daná soustavu bude počítána v modulární aritmetice v tělese $\mathbb{GF}(13)$.

Soustavu $2x_1+7x_2+3x_3 \equiv 2 \pmod{13}$, $5x_1+11x_2+7x_3 \equiv 11 \pmod{13}$, $1x_1 + 8x_2 + 3x_3 \equiv 3 \pmod{13}$ zapíšeme do rozšířené maticové formy:

$$\left(\begin{array}{ccc|c} 2 & 7 & 3 & 2 \\ 5 & 11 & 7 & 11 \\ 1 & 8 & 3 & 3 \end{array}\right)$$

Aktuálně jsme v 1. fázi, prvek v prvním sloupci matice, na prvním řádku je nenulový bude tedy pivotem.

$$\left(\begin{array}{ccc|c} \boxed{2} & 7 & 3 & 2 \\ 5 & 11 & 7 & 11 \\ 1 & 8 & 3 & 3 \end{array}\right)$$

Vypočítáme multiplikativní inverzi v daném modulu: $2^{-1} \equiv 7 \pmod{13}$

Normalizuje první řádek: $|2 \cdot 7|_{13} = 1$, $|7 \cdot 7|_{13} = 10$, $|3 \cot 7|_{13} = 8$, $|2 \cdot 7|_{13} = 1$

Eliminujeme 2. řádek, tím že od něj odečteme 5-ti násobek normalizovaného prvního řádku, 3. řádek eliminujeme, tak že od něj odečteme normalizovaný první řádek.

Po normalizaci 1. řádku a eliminaci ostatních pak soustava bude vypadat takto:

$$\left(\begin{array}{ccc|c} 1 & 10 & 8 & 1 \\ 0 & 0 & 6 & 6 \\ 0 & 11 & 8 & 2 \end{array}\right)$$

Hodnoty v první sloupci jsou již pro výpočet nepodstatné, proto je není potřeba již dále ukládat, v zápise budou nepodstatné sloupce nahrazeny symbolem $\cdot$.

V druhé fázi budeme hledat pivota na druhém řádku, v druhém sloupci, vidíme, že je zde hodnota $= 0$, musíme tedy projít další řádky, v našem případě pouze poslední, ale pokud by se jednalo o soustavu o více neznámých, prošly by se hodnoty ve druhém sloupci od druhého řádku, dokud bychom nenalezli první nenulovou hodnotu.

$$\left(\begin{array}{ccc|c} \cdot & 10 & 8 & 1 \\ \cdot & 0 & 6 & 6 \\ \cdot & \boxed{11} & 8 & 2 \end{array}\right)$$

Pivot byl nalezen, odpovídající řádky mezi sebou zaměníme a upravíme determinant.

$$\left(\begin{array}{ccc|c} \cdot & 10 & 8 & 1 \\ \cdot & \boxed{11} & 8 & 2 \\ \cdot & 0 & 6 & 6 \end{array}\right)$$

Vypočítáme multiplikativní inverzi v daném modulu: $11^{-1} \equiv 6 \pmod{13}$

Normalizuje druhý řádek: $|11 \cdot 6|_{13} = 1$, $|8 \cdot 6|_{13} = 9$, $|2 \cdot 6|_{13} = 12$

První řádek eliminuje tím, že od něj odečteme desetinásobek normalizovaného druhého řádku.

Po normalizaci 2. řádku a eliminaci ostatních pak soustava bude vypadat takto:

$$\left(\begin{array}{ccc|c} \cdot & 0 & 9 & 11 \\ \cdot & 1 & 9 & 12 \\ \cdot & 0 & 6 & 6 \end{array}\right)$$

Ve třetí fázi budeme hledat pivota na třetím řádku matice soustavy.

$$\left(\begin{array}{ccc|c} \cdot & \cdot & 9 & 11 \\ \cdot & \cdot & 9 & 12 \\ \cdot & \cdot & \boxed{6} & 6 \end{array}\right)$$

Vypočítáme multiplikativní inverzi v daném modulu: $6^{-1} \equiv 11 \pmod{13}$

Normalizuje třetí řádek: $|6 \cdot 11|_{13} = 1$, $|6 \cdot 11|_{13} = 1$

První a druhý řádek eliminuje tím, že od nich odečteme devítinásobek normalizovaného druhého řádku.

Po normalizaci 3. řádku a eliminaci ostatních pak soustava bude vypadat takto:

$$\left(\begin{array}{ccc|c} \cdot & \cdot & 0 & 2 \\ \cdot & \cdot & 0 & 3 \\ \cdot & \cdot & 1 & 1 \end{array}\right)$$

Poslední sloupec matice soustavy by opět nebylo potřeba vypočítávat, je dán jednotkovou maticí.

Výsledné řešení soustavy je pak tedy $x_1 \equiv 2 \pmod{13}$, $x_2 \equiv 3 \pmod{13}$, $x_3 \equiv 1 \pmod{13}$.

Nyní provedeme zkoušku výpočtu. Do původní soustavy

$$\begin{array}{rcrcrcll} 2x_1 & + & 7x_2 & + & 3x_3 & \equiv & 2 & \pmod{13} \\ 5x_1 & + & 11x_2 & + & 7x_3 & \equiv & 11 & \pmod{13} \\ 1x_1 & + & 8x_2 & + & 3x_3 & \equiv & 3 & \pmod{13} \end{array}$$

dosadíme vektor pravých stran:

$$\vec{x} \equiv \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix} \pmod{13}$$

$$\begin{array}{rcrcrcll} 2 \cdot 2 & + & 7 \cdot 3 & + & 3 \cdot 1 & \equiv & 2 & \pmod{13} \\ 5 \cdot 2 & + & 11 \cdot 3 & + & 7 \cdot 1 & \equiv & 11 & \pmod{13} \\ 1 \cdot 2 & + & 8 \cdot 3 & + & 3 \cdot 1 & \equiv & 3 & \pmod{13} \end{array}$$

Získáme tedy

$$\begin{array}{rcrcrcll} 4 & + & 21 & + & 3 & \equiv & 2 & \pmod{13} \\ 10 & + & 33 & + & 7 & \equiv & 11 & \pmod{13} \\ 2 & + & 24 & + & 3 & \equiv & 3 & \pmod{13} \end{array}$$

z čehož je zřejmé, že jsme nalezli správné řešení soustavy lineárních kongruencí.

1.8 Počet operací k nalezení řešení soustavy lineárních kongruencí

Z analýzy hledání řešení soustav lineárních kongruencí vidíme, že pro nalezení řešení soustavy o n neznámých potřebujeme následující operace:

Operace	**Zápis**	**Počet operací**
Modulární inverze	$\lvert a^{-1} \rvert_m$	n
Modulární násobení	$\lvert a \cdot b \rvert_m$	$n \cdot (n \cdot (n+1))$
Modulární sčítání (odčítání)	$\lvert a \pm b \rvert_m$	$n \cdot (n-1) \cdot (n+1)$

Tabulka 1.1: Počet operací potřebných pro nalezení řešení soustavy o n neznámých

V tabulce výše uvedené počty operací nezohledňují vylepšení algoritmu v pozdějších fázích (eliminačních krocích), kdy se počet násobení a sčítání snižuje. Řádově je však počet stejný. Složitost sekvenčního algoritmu je vždy řádově $\mathcal{O}(n^3)$.

Z hlediska rychlosti operací z tabulky vyplývá, že pro rychlost není kritické hledání inverze, ale především násobení a sčítání/odčítání. Proto lze pro výpočet inverze použít i nepříliš optimální metody, jejich vliv na rychlost nebude natolik kritický.

Díky výše uvedenému není nutné využívat žádnou z optimalizovaných metod hledání inverze, plně postačuje hledání inverze pomocí rozšířeného Euklidova algoritmu, tak jak je uveden například v [19].

Naopak pro násobení a sčítání/odčítání je vhodné použít co nejoptimálnější metody, aby bylo dosaženo dostatečné rychlosti nalezení řešení soustavy.

Pro další návrh budeme pracovat v tělese $\mathbb{GF}(p)$, využijeme tedy operace modulární násobení, sčítání, odčítání a vynásobení inverzí. Řešení soustavy lineárních kongruencí budeme hledat pomocí algoritmu využívající normalizaci prvků na hlavní diagonále.

Kapitola 2

Analýza stávajících architektur

Následující kapitola se zabývá analýzou a zhodnocením stávajících architektur pro hledání řešení soustavy lineárních kongruencí.

2.1 Motivace

Jak již bylo popsáno výše, je řešení soustav lineárních rovnic důležitým matematickým problémem. V kryptoanalýze symetrických šifer se nejčastěji vyskytují středně velké husté soustavy rovnic. Výsledky těchto řešení jsou pak v široké škále aplikovatelné na různé blokové nebo proudové šifry. Na druhou stranu rychlé řešiče pro řešení malých až středně velkých soustav rovnic jsou potřebné pro efektivní implementaci třídy MQ kryptosystémů [16].

Pro použití v algebraické kryptoanalýze lze využít linearizace nelineárních rovnic. Tyto rovnice jsou nejdříve zjednodušeny, linearizovány a pak vyřešeny jako soustava lineárních rovnic. Při využití při diferenciální kryptoanalýze šifry PRESENT se ukázalo, že je vhodnější pro snížení počtu rund využít soustavy lineárních rovnic, než hádání hrubou silou [3]. Přidáme-li další informace získané z druhotných zdrojů informací, jako například měření spotřeby nebo elektromagnetického záření při šifrování, lze za pomoci rovnic s takto získanými kolizemi a Hammingovými váhami jednotlivých mezivýsledků šifry dále zjednodušovat, například šifru AES lze takto zjednodušit [18][2].

Také je možné šifru AES podrobit přímé algebraické kryptoanalýze. Při použití 128-bitového klíče je možné šifru AES popsat jako soustavu o 8000 rovnicích s 1600 proměnnými. Další informace lze nalézt v článku [16].

Jak již bylo řečeno hledání řešení soustavy lineárních rovnic je důležitá matematická disciplína použitelná v mnoha oblastech. Pro určité aplikace

se jeví využití speciálních architektur určených k tomuto účelu efektivněji.

2.2 Architektury využívající výměnu řádků

V následující části budu vycházet z informací obsažených v [20], zde lze nalézt bližší informace o těchto architekturách včetně srovnání výkonnostních parametrů. Tento článek dává několik návrhů, jak postupovat při řešení soustavy lineárních rovnic pomocí řešičů založených na různých architekturách. V této sekci se bude jednat vždy o řešení nad tělesem $\mathbb{GF}(2^k)$.

Níže uvedené architektury vycházejí z Gauss-Jordanovy eliminační metody diskutované v sekci 1.6.

2.2.1 GSMITH

Tato architektura, na rozdíl od ostatních níže uvedených, nejdříve načte celou matici do řešiče a pak pomocí lehce modifikovaných a paralelizovaných řádkových operací nalezne řešení dané soustavy.

V architektuře se vyskytují celkem 4 prvky, které jsou mezi sebou propojeny jak globálně tak lokálně.

- *Pivot cell - Buňka pivota:* Provádí inverzi dané hodnoty na tělesem $\mathbb{GF}(2^k)$.
- *Pivot row cell - Buňka v řádku pivota:* Provádí normalizaci řádku pivota.
- *Pivot column cell - Buňka ve sloupci pivota:* Udržuje hodnotu ve sloupci pivota.
- *Basic cell - Základní buňka:* Provádí eliminaci hodnot s pomocí dat buněk v řádku a sloupci pivota.

Výhodou je že díky množství propojení a postupu výpočtu je potřeba jen jeden invertor nad daným tělesem. Nevýhodou je závislost času výpočtu na pravděpodobnostním rozdělení soustavy.

Architektura využívá rotace všech řádků doleva a nahoru pro zajištění přesunu dat k daným buňkám.

2.2.2 TSN, TSA, TSL architektury

Jedná o dvou-rozměrné architektury složené z mřížek výpočetních buněk. Matice obsahuje celkem n buněk a u každého řádku je uveden příznak, zda

již byl pivotem. Řádek je buď normalizován nebo eliminován (dle příznaku), a po provedení dané operace posunut o jednu pozici nahoru a na jeho místo se nasune další řádek. Výpočetní buňky jsou organizovány do spodní trojúhelníkové matice a inverzi provádí buňky na diagonále. Výpočet je dokončen po nasunutí všech řádků a poté je zpětnou substitucí získáno řešení soustavy. V architektuře se vyskytují buď pivotizační buňky nebo základní buňky.

Architektury TSA a TSL jsou modifikací architektury TSN. Jedná se o modifikace buď vložením registru k uložení mezilehlého výsledku u sousedních buněk (TSA) nebo vložení zpožďovacích prvků ve vertikálním směru (TSL).

2.2.3 LSL, LSA architektury

Jedná se jednorozměrné systolické architektury. Místo dvourozměrné mřížky aritmetických buněk zachováme pouze jeden aritmetický řádek buněk a n dalších buněk. S pomocí multiplexerů a posuvných registrů se hodnoty potřebné k výpočtu posouvají k aritmetickému řádku. Buňky v tomto řádku pak podle přivedených kontrolních signálů provádí potřebnou aritmetickou operaci.

Rozdíl mezi architekturou LSL a LSA je takový, že u architektury LSA jsou do horizontálních signálů vloženy paměti pro ukládání těchto signálů. Díky tomu se snižuje počet zpožďovacích registrů, je však potřeba přivádět data do architektury v jiné posloupnosti.

2.3 Architektury využívající výměnu ukazatelů

Další z možných přístupů k návrhu architektur, je že místo výměny řádků, ve kterých není prvek jenž může být pivotem, budeme tyto řádky přeskakovat a budeme pivotizovat řádky nezávisle na hlavní diagonále. Po nalezení řešení, je však potřeba vektor řešení dané soustavy přeuspořádat, podle pořadí, jak byly řádky zpracovávány. Toho lze dosáhnout například vektorem indexů pivota.

Architektura prezentovaná v [4] je složená z několika procesorů, který každý pracuje v jiné zbytkové třídě, aby tak získaly řešení většího problému. Jednotlivá nalezená řešení se buď zpětně složí pomocí Čínské věty o zbytcích nebo pomocí konverze do číselné soustavy s různými základy (tzv. Mixed Radix System).

Jednotlivé procesory v této architektuře využívají pro pivotizaci zápisu adresy řádku pivota do vektoru indexů pivota. Sloupce se postupně v každém eliminačním kroku posouvají o jednu pozici vlevo a tímto, společně s přeuspořádáním vektoru řešení, je zabezpečeno správné pořadí výpočetních kroků a správné pořadí proměnných ve vektoru řešení.

2.4 Zhodnocení stávajících architektur

Vzhledem k tomu, že řešení soustav lineárních rovnic, potažmo lineárních kongurencí, patří k základním matematickým disciplínám, bylo již navrženo mnoho různých architektur. Každá má určité přednosti, zápory a je potřeba se s těmito vlastnostmi vždy vypořádat.

Z těchto důvodů je potřeba danou architekturu vždy blíže prozkoumat, porozumět jejímu chování a zjistit jaké další vlastnosti s jejím použitím souvisí. Abychom mohli architekturu zkoumat i pomocí softwaru lze její chování simulovat.

Vzhledem k tomu, že architektury, které pracují s ukazatelem na řádek s pivotem, byly již dříve popsány a jejich chování včetně porovnání vlastností pro různé procesory již byly popsány, pro další zkoumání jsme zvolili architekturu využívající pivotizaci s výměnou řádků.

Kapitola 3

Návrh architektury řešiče

Následující kapitola se zabývá jednotlivými aspekty návrhu řešiče soustav lineárních kongruencí. Zkoumá jaké aspekty návrhu plynou z matematických struktur a výpočtů potřebných k nalezení řešení soustav.

Mezi posuzované aspekty návrhu patří návrh z hlediska jednotlivých aritmetických jednotek, organizace výpočtu mezi těmito jednotkami, komunikace s pamětí a připojení paměti.

V závěru kapitoly jsou na základě zhodnocení jednotlivých aspektů návrhu zvoleny vlastnosti, které budou určující pro námi navrhovaná architekturu. Chování této architektury bude v následující kapitole realizováno pomocí simulace.

3.1 Návrh řešiče soustav lineárních kongruencí

Pro to abychom mohli řešit soustavy lineárních kongruencí, můžeme buď využít specializovaný hardware, nebo využít již existují hardware a pomocí vhodného softwaru nalézt řešení.

Přístup pomocí softwaru není tématem této práce, proto nebude dále rozvíjen. Pro tento přístup lze například využít algoritmus popsaný v sekci 1.6.4.

Dalším přístupem je využít dedikovaný hardware pro nalezení řešení. Tento hardware budeme nazývat řešičem soustav lineárních kongruencí.

Takovýto řešič je pak složen z několika hardwarových jednotek. Samotný řešič je připojen k paměti, ve které jsou uloženy data potřebná k výpočtu. Připojení řešiče je realizováno pomocí řadiče paměti, který komunikuje s pamětí.

Jednotky řešiče se liší svojí funkcí a v závislosti v jakém kroku se výpočet nachází se tyto jednotky mohou chovat různě.

Chování hardwarového řešiče lze simulovat pomocí simulačního modelu vytvořeném v některém ze simulačních jazyků. Na tomto simulačním modelu pak můžeme sledovat a analyzovat chování řešiče.

3.2 Návrh aritmetických jednotek

Pro návrh aritmetických jednotek využijeme informace z tabulky 1.1. Vidíme, že se zde vyskytuje výpočet modulární inverze, vynásobení inverzí a odečet s násobením. Z návrhu algoritmu 1.6.4 vidíme, že pro výpočet jedné fáze (eliminačního kroku) řešení soustavy potřebujeme výpočet inverze, vynásobení inverzí a odečet vynásobeného člena.

Vzhledem k výše uvedeného můžeme předpokládat, že řešič bude obsahovat buňky specializované na danou operaci, tak aby bylo možné dosáhnout co nejmenší prostorové a časové náročnosti.

Dále můžeme předpokládat, že jednotlivé jednotky mají kromě samotné výpočetní logiky i malou paměť potřebnou k uložení informací potřebných k výpočtu.

Z toho co bylo uvedeno výše, vidíme že pro návrh aritmetických jednotek je podstatné v jaké fázi (eliminačním kroku) a kroku (části eliminační kroku) výpočtu se nacházíme, tuto informaci bude samotný řešič využívat ke správnému určení posloupnosti přístupů do paměti a zásobení aritmetických jednotek daty.

Dále vidíme, že v prvním kroku výpočtu se vždy provádí jiná operace, než v následujících krocích. Abychom nemuseli pro první krok výpočtu mít speciální aritmetické buňky, přesuneme tyto funkce na jiné buňky a v závislosti na kroku (části eliminačního kroku) výpočtu budeme měnit funkce těchto buněk.

Další důležitou vlastností je, že abychom mohli eliminovat člena, musí již být na dané úrovni dokončena normalizace.

Nyní můžeme přistoupit k samotnému návrhu aritmetických buněk. Navrhneme dva různé typy buněk. První buňku provádějící výpočet inverze a vynásobení inverzí, tato buňka bude zajišťovat normalizaci řádku. Druhá buňka bude na základě dat z první buňky provádět odečtení součinu normalizovaného řádku s eliminovaným řádkem, tzv. eliminaci.

První buňku nazývejme buňkou v řádku pivota, protože provádí operace s řádkem s pivotem. Druhou buňku nazývejme eliminační buňkou.

3.2.1 Buňka v řádku pivota

Funkcí buňky v řádku pivota je výpočet inverze a násobení inverzí. To, která operace se bude provádět, určuje to v jakém kroku výpočtu se nacházíme.

V prvním kroku dané fáze výpočtu se vypočte inverze a v dalších krocích se vynásobí touto inverzí ostatní členy daného řádku. Tímto se provede normalizace daného řádku.

3.2.2 Eliminační buňka

Funkcí eliminační buňky je odečtení násobku výsledku z buňky v řádku pivota. Výše násobku je určena prvním prvkem daného řádku. Tato buňka nemůže zahájit výpočet dříve než buňka v řádku pivota na dané pozici dokončí výpočet.

V prvním kroku se uloží první prvek daného řádku. V dalších krocích dané fáze výpočtu se od členа odečte násobek výsledku z buňky v řádku pivota a uloženého prvku. Tímto se provede eliminace daného řádku.

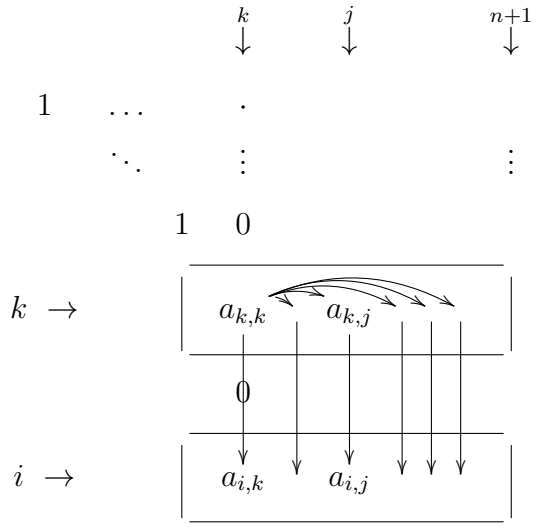
3.3 Organizace výpočtu

Pro nalezení řešení soustavy můžeme výpočet organizovat buď po sloupcích nebo po řádcích.

3.3.1 Výpočet po řádcích

Pokud bude v každém kroku výpočtu vyřešen vždy jeden řádek matice, musí být řádek pivota vyřešen před zahájením výpočtu v řádcích. V každém kroku je vždy celý řádek vynásoben řádkem pivota a odečten.

Následující diagram znázorňuje přeposílání dat mezi buňkami při výpočtu po řádcích.



Postup výpočtu jednoho řádku je pak následující:

1. Dříve vypočtenou inverzí se paralelně vynásobí řádek k.
   $a_{k,k\ldots n+1} = |a_{k,k\ldots n+1} \cdot a_{k,k}^{-1}|_m$
2. Vynásobený řádek $a_{k,k\ldots n+1}$ se pošle do řádku i, kde jednotlivé buňky paralelně pomocí odečtení a násobení eliminují ostatní členy.
   $a_{i,k\ldots n+1} = |a_{i,k\ldots n+1} - a_{i,k} \cdot a_{k,k\ldots n+1}|_m$

Po výpočtu řádku i se přejde na řádek $i + 1$.

Tento přístup k výpočtu není v námi navrhované architektuře použit proto nebude dále zamýšlen.

3.3.2 Výpočet po sloupcích

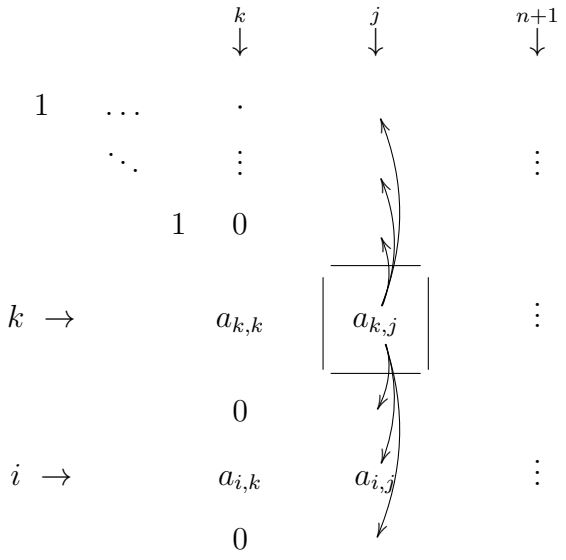
Jestliže bude námi navrhovaná architektura organizovat výpočet po sloupcích, tzv. v jednom kroku bude vyřešen vždy jeden sloupec matice, musí mít každá eliminační buňka před zahájením výpočtu následující data:

- Vstupní data
- Výsledek operace z buňky pivota

Z tohoto plyne nutnost optimalizace výpočtu, tak aby obě informace byly k dispozici nejlépe ve stejný okamžik a nemuselo se čekat na výsledek z buňky pivota.

Samotná buňka v řádku pivota k výpočtu potřebuje kromě vstupních dat ještě vypočtenou inverzi.

Následující diagram znázorňuje přeposílání dat mezi buňkami při výpočtu po sloupcích.



Postup výpočtu jednoho sloupce je pak následující:

1. Dříve vypočtenou inverzí se vynásobí člen $a_{k,j}$.

   $a_{k,j} = |a_{k,k}^{-1} \cdot a_{k,j}|_m$

2. Vynásobený člen $a_{k,j}$ se pošle do sloupce j, kde jednotlivé buňky paralelně pomocí odčítání a násobení eliminují jednotlivé členy.

   $a_{i,j} = |a_{i,j} - a_{i,k} \cdot a_{k,j}|_m$

Po výpočtu sloupce j se přejde na sloupec $j + 1$.

Při tomto postupu je důležité aby člen $a_{k,j}$, kterým se jednotlivé členy násobí, byl k dispozici co nejdříve, v ideálním případě stejně jako je k dispozici sloupec $a_{1\ldots n,j}$.

Pokud není k dispozici dostatek takovýchto buněk, je možné využít každou buňku pro eliminaci jinou hodnotou $a_{i,k}$. Je však nutné aby, zároveň se členem $a_{k,j}$, buňky měli k dispozici správnou hodnotu $a_{i,k}$.

3.3.3 Propojení a počet buněk

Z organizace výpočtu vidíme, že se nabízí několik možností propojení buněk, stejně tak několik možností pro počet aritmetických buněk.

Pro počet aritmetických buněk může zhlediska velikosti matice mohou nastat tři situace vzhledem k počtu aritmetických jednotek:

1. Menší velikost matice než je počet aritmetických jednotek - matice se celá vejde do řešiče s rozdílem, že některé jednotky nejsou využívány.

2. Stejná velikost matice jako počet jednotek - tato situace je podrobněji rozpracována například zde [20] nebo zde [9].

3. Větší velikost matice než je počet jednotek - je nutné zpracovávat matici postupně a do řešiče přivádět jen potřebná data.

Pro další návrh je zajímavá situace číslo 3, kdy je nutné řešič aktivně zásobovat částmi soustavy tak, aby byly co nejefektivněji vytíženy aritmetické jednotky a byla zachována posloupnost výpočtu.

Jako ideální propojení se dá považovat takové, kde eliminační buňky mohou rovnou načítat výsledek z buňky pivota a nemusely čekat než řešič předá data.

Budeme-li chtít zmenšovat počet buněk, případně nám paměťové rozhraní neumožní efektivně zásobovat jednotky daty, musíme počítat s dalším zpožděním, buď za cenu většího počtu paměťových operací nebo za cenu větších nákladů pro řízení výpočtu.

Na druhou stranu, zmenšením počtu výpočetních jednotek lze docílit menší velikosti obvodu. Je však nutné výpočet optimalizovat.

3.4 Návrh paměti a paměťového rozhraní

Protože pro nalezení řešení je nezbytné aktivně zásobovat aritmetické jednotky řešiče daty (vycházíme z předpokladu, že se celá soustava nevejde do řešiče), je nutná optimalizace přístupů do paměti v závislosti na paměťovém rozhraní.

Paměťové rozhraní je připojeno k řešiči pomocí adresové a datové sběrnice. Můžeme předpokládat, že paměť má dostatečnou šířku adresové sběrnice, tak aby mohla adresovat všechny buňky paměti a nedocházelo ke zpoždění.

Samotná datová sběrnice však nemusí poskytovat dostatečnou šířku, takže nebudeme schopní provést datovou operaci v celém sloupci najednou. V ideálním případě bychom byli schopní zpracovat celý sloupec v jednom kroku výpočtu. Námi navrhovaná architektura musí být schopna pracovat i s užší šířkou paměťové sběrnice.

Můžeme však předpokládat, že jsme schopní přenést alespoň 2 buňky paměti zároveň, jinak by docházelo k dalším zpožděním.

Vzhledem k tomu, že většina hardwaru, je dnes vybavena i lokálními paměťmi, můžeme nadále předpokládat, že každá eliminační buňka si dokáže uložit do lokální paměti hodnotu násobku, kterým bude řádek eliminovat. Dále můžeme předpokládat, že buňka pivota si uloží vypočtenou inverzi.

Abychom mohli pracovat s užšími paměťovými sběrnicemi a nemuseli v každém kroku znovu načítat vypočtenou hodnotu v řádku s pivotem, můžeme také předpokládat, že řešič má lokální paměť určenou k tomuto.

Námi navrhovaná architektura, respektive její paměťové rozhraní, však neumožňuje současné čtení a zápis do paměti, je vždy potřeba počkat na dokončení jedné paměťové operace.

Paměť a její paměťové rozhraní však umožňuje blokový přenos dat. Když tedy načítáme/zapisujeme buňky v paměti uložené v sousedních buňkách, doba načtení je řádově menší než kdybychom načítali/zapisovali buňky nesousedních buněk.

V závislosti na tom jak je organizován výpočet je vhodné stejně organizovat data v paměti. Data lze ukládat v paměti buď po sloupcích nebo po řádcích. Při uložení po sloupcích jsou v po sobě jdoucích buňkách paměti uloženy hodnoty ze sloupce matice soustavy.

3.5 Návrh řídící buňky

Hlavní funkcí řídící buňky je komunikace s pamětí skrz paměťové rozhraní, předávání dat jednotlivým buňkám a předávání pokynů těmto buňkám. Řídící buňka se také stará o zahájení a ukončení výpočtu. Samotné výpočty však probíhají na úrovni aritmetických jednotek. Řídící buňka také bude udržovat lokální mezivýsledky výpočtu.

Protože předpokládáme, že nebudeme mít vždy dostatek eliminačních buněk, bude řídící buňka udržovat hodnoty používané k eliminaci v lokální paměti.

3.6 Navržená architektura

Pro realizaci modelu architektury jsme zvolili následující parametry:

- Sloupcová organizace výpočtu architektury
- Pivotizace s normalizací prvků na hlavní diagonále
- Oddělená řídící, výpočetní a paměťová část

- Paměťová část umožňující blokový přenos slov
- Paměť organizována po sloupcích
- Paměť neumožňující současný zápis a čtení

Kapitola 4

Návrh vizualizace výpočtu

Aby bylo možné lépe zachytit průběh výpočtu, bude realizována jednoduchá vizualizační aplikace. Aplikace bude zobrazovat činnost jednotlivých buněk a posloupnost práce s pamětí vedoucí k vyřešení soustavy lineárních rovnic. Samotná vizualizace bude probíhat pomocí průchodu XML souboru vytvořeným simulační aplikací.

Kapitola 5

Realizace modelu architektury

Následující kapitola se zabývá samotnou realizací simulačního modelu architektury.

Pro realizaci simulace chování architektury byl realizován model architektury simulující chování architektury navržené v sekci 3.6. Tento model byl realizován v knihovně SystemC v jazyce C++. Informace o použité verzi knihovny lze nalézt v příloze v sekci A.3. Bližší informace o tom, jak používat simulační model nalézt v sekci A.1.

5.1 Parametrizace modelu

Níže uvedená tabulka 5.1 shrnuje jak je možné parametrizovat model architektury. Samotná parametrizace modelu se provádí pomocí konfiguračního souboru. Ukázka konfiguračního souboru je uvedena v příloze v sekci A.1.4.

Doba provedení jednotlivých operací je zadána počtem hodinových cyklů, za které je daná operace dokončena. Reálný čas provedení operace pak získáme vynásobením počtu hodinových cyklů periodou hodinového signálu. Každá operace je parametrizovatelná samostatně.

Doba přístupu do paměti je parametrizovatelná obdobně, opět se jedná o počet hodinových cyklů, potřebných k dokončení čtení/zápisu z/do paměti.

5.2 Složení simulačního modelu architektury

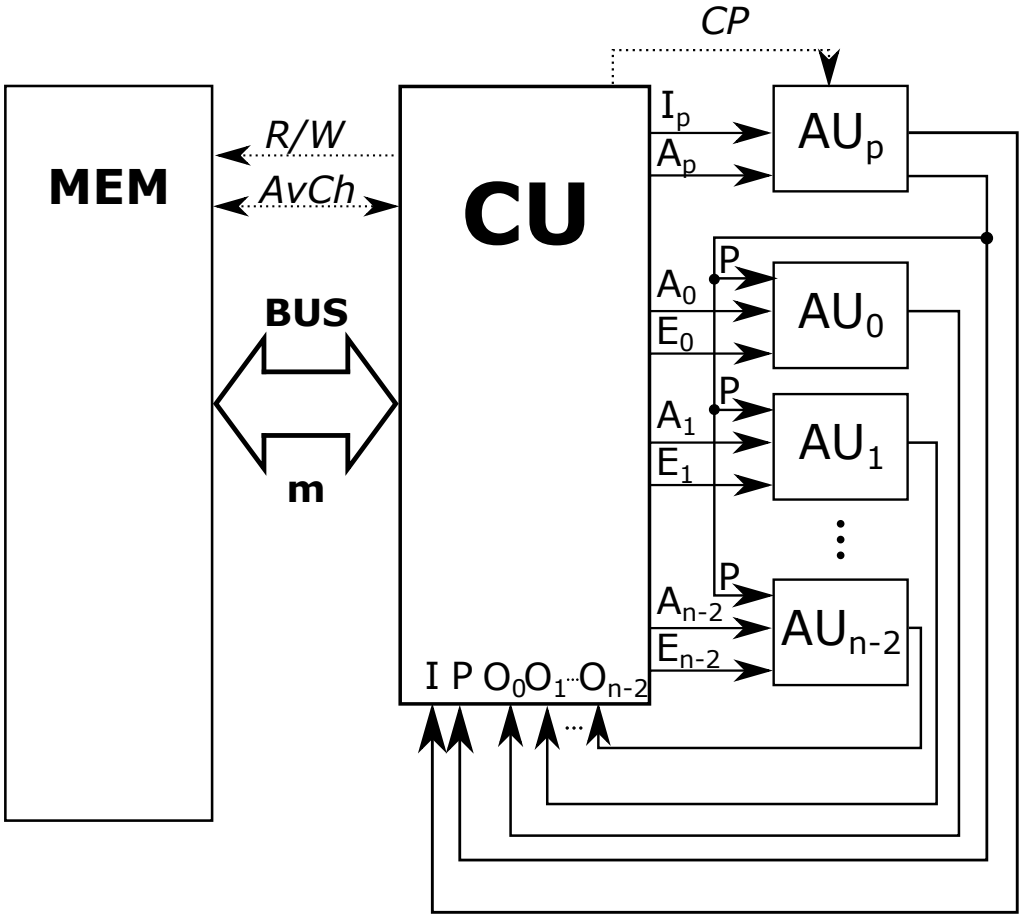
Realizovaný simulační model je složen ze tří částí. Řídící buňky, paměti a ALU buněk. Funkce jednotlivých částí jsou popsány v dalších sekcích.

Název	**Symbol**	**Matematický výraz**	**Význam**
Velikost matice	n		Počet řádků soustavy lineárních kongruencí.
Modul	m		Velikost modulu soustavy.
Perioda hodinové signálu	T_{clk}		Perioda hodinového signálu.
Velikost operandu	k	$k \geq$ počet bitů pro uložení modulu	Závisí na počtu bitů potřebných k uložení modulu. Pokud matematické operace zahrnují i další skryté konstanty je možné je zohlednit v dané konstantě k.
Počet cyklů pro násobení	$T_{\cdot}$	$T_{\cdot} = (a \cdot k^2 + b \cdot k + c) \cdot T_{clk}$	Čas potřebný k provedení operace modulárního násobení.
Počet cyklů pro inverzi	$T_{^{-1}}$	$T_{^{-1}} = (a \cdot k^2 + b \cdot k + c) \cdot T_{clk}$	Čas potřebný k provedení operace modulární inverze.
Počet cyklů pro odčítání	T_{-}	$T_{-} = (a \cdot k^2 + b \cdot k + c) \cdot T_{clk}$	Čas potřebný k provedení operace modulární odčítání.
Šířka paměťové sběrnice	$\#_{Bus}$	$2 \leq \#_{Bus} \leq n$	Počet po sobě jdoucích hodnot, které lze z paměti přečíst/zapsat během jednoho sekvenčního čtecího/zapisovacího cyklu.
Počet ALU jednotek	$\#_{ALU}$	$2 \leq \#_{Bus} \leq \#_{ALU} \leq n$	Počet aritmetických jednotek použitých pro výpočet.
Počet cyklů pro sekvenční zápis/čtení do paměti	$T_{R/W}$	$T_{R/W} = c \cdot T_{clk}$	Čas potřebný k provedení sekvenčního zápisu nebo čtení z buňky paměti.
Počet cyklů pro náhodný zápis/čtení do paměti	$T_{nR/nW}$	$T_{nR/nW} = c \cdot T_{clk}$	Čas potřebný k provedení náhodného zápisu nebo čtení z buňky paměti.

Tabulka 5.1: Možnosti parametrizace modelu architektury

Pro výpočet předpokládáme, že data jsou uložena v paměti a nalezené řešení je pak také uložené v paměti. Samotný výpočet je realizován podle navržené architektury.

Schéma 5.1 znázorňuje navržené uspořádání modelu architektury.



Obrázek 5.1: Schématické znázornění modelu architektury

Paměť (na schématu vyznačena jako *MEM*) je přes sběrnici (*BUS*) o šířce m připojena k řídící buňce (*CU*). Řídící buňka komunikuje s pamětí pomocí událostí *readEvent*, *writeEvent* (ve schématu *R/W*), *testReadWriteAvailableCheck* (*AvCh*).

Zatímco mezi výpočetními buňkami a řídící buňkou se přenáší pouze datové hodnoty, mezi pamětí se po sběrnici přenášejí tzv. zprávy. Tato komunikace pomocí zpráv simuluje přenos po oddělené datové a adresové sběrnici. Každá zpráva navíc obsahuje údaje o směru, zda je od řídící buňky nebo z paměti.

Výpočetní buňky jsou dvojího typu, buňka v řádku pivota (AU_p) a eliminační buňka (AU_n).

Buňka v řádku pivota je k řídící buňce připojena vstupními porty *in_vstup* (A_p) a *in_k_inverzi* (I_p). Výstupními porty *out_vystup* (P) a *out_vystup_inverze* (I) k řídící buňce.

Každá eliminační buňka je k řídící buňce připojena vstupními porty *in_vstup* (A_n) a *in_eliminacni_clen* (E_n). Dále jsou všechny eliminační buňky připojeny na výstupní port buňky v řádku pivota *out_vystup* (P) vstupním portem *in_od_pivota* (P).

Připojení eliminačních buněk je v řídící buňce realizováno jako vektor. Samotné eliminační buňky jsou realizovány jako vektor těchto buněk. Propojení mezi eliminačními buňkami a řídící buňkou je realizováno pomocí struktury *sc_buffer*, která se od struktury *sc_signal* simulující chování propojovacích signálů liší tím, že zapíšemi–li stejnou hodnotu do struktury *sc_signal* znovu, není vyvolána událost, že se hodnota změnila. Kdežto u struktury *sc_buffer*, zapíšeme-li stejnou hodnotu znovu, je vyvolána událost označující změnu hodnoty.

5.3 Řídící buňka

Řídící buňka zajišťuje koordinaci výpočtu mezi jednotlivými ALU buňkami. Zajišťuje komunikaci s pamětí a udržuje informace o aktuální fázi. Dále je odpovědná za výpočet determinantu, hledání pivota a udržuje hodnoty řídícího sloupce pro eliminaci v ALU buňkách. Samotná řídící buňka je vybuzena k činnosti signálem *clk*.

5.4 Realizace paměti

Protože námi navrhovaná architektura využívá uložení dat po sloupcích jsou data interně uloženy v jednorozměrném vektoru. Tento vektor je na soustavu namapován po sloupcích. V paměti nejsou striktně odděleny jednotlivé sloupce, takže po poslední buňce sloupce i následuje první buňka sloupce $i + 1$. Proto čtení následující buňky po poslední buňce sloupce i je chápáno jako čtení první buňky sloupce $i + 1$.

Paměťové rozhraní je zde simulováno pomocí vstupně výstupní sběrnice. V této sběrnici je sdružená jak datová, tak adresová sběrnice. Šířka této sběrnice je dána šířkou paměťové sběrnice. Realizace proměnné šířky sběrnice je provedena pomocí přenosu vektoru více hodnot po této sběrnici.

Samotné čtení a zápis je synchronizován pomocí událostí. Stejně tak kontrola zda v paměti již neprobíhá paměťová operace je realizována pomocí událostí. Doba potřebná pro čtení/zápis může být rozdílná pro sekvenční a náhodné čtení. Pro sekvenční paměťovou operaci vždy předpokládáme, že další buňka paměti je vzdálená právě o šířku paměťové sběrnice. Simulace v sekvenční paměťové operaci nerozlišuje zda předchozí operace bylo čtení nebo zápis.

Pro každou paměťovou operaci je zde vlastní vlákno, které podle toho zda se jedná o čtení/zápis po sobě jdoucích buněk nebo ne přečte/zapíše data z/do paměti za určitý časový úsek.

Dále je zde vlákno, které slouží k otestování zda je již dokončená předchozí paměťová operace.

5.5 Realizace komunikace s pamětí během výpočtu

Aby námi simulovaná architektura mohla paralelně řídit výpočet a komunikovat s pamětí, je nutné aby se čtení/zápis realizovalo nezávisle na událostech souvisejících s pamětí. Řídící buňka nemůže zároveň čekat na dokončení paměťové operace a změnu na výstupních portech ALU buněk.

Z výše popsaných důvodů je nutné oddělit čtení/zápis z paměti od hlavní logiky řídící buňky. Toto je realizováno vlastními vlákny pro čtení/zápis, samotná řídící buňka vždy jen zapíše do dané fronty a případně počká na vyprázdnění fronty, jestliže je to nutné. Také je potřeba zajistit aby čtení mělo větší prioritu než zápis do paměti.

5.5.1 Realizace čtecí fronty

Pro čtení je realizováno vlastní vlákno řídící buňky, které pro svojí činnost využívá 2 čtecí fronty, jednu frontu se samotnými povely pro paměť a druhou s adresami ALU jednotek do kterých se mají načtené hodnoty přivést.

Vlákno se probudí na událost, kterou řídící buňka oznámí, že vložila do fronty vektor ke čtení. Řídící buňka také musí zajistit vložení adresy ALU jednotky do vlastní fronty. Vlákno z fronty ke čtení přečte první prvek a přečte jej z paměti. Vlákno vyčká na dokončení čtení z paměti, přečtené hodnoty přivede na vstupy daných buněk. Dále odebere právě přečtený prvek z fronty a stejně tak z fronty odebere adresy ALU buněk, které použilo. Vlákno je využíváno i v prvních krocích každé fáze (eliminačního kroku) a to pro načítání řídícího sloupce, kterým se pak eliminují ostatní sloupce.

V tomto prvním kroku, jakmile je načten pivot, předá ho buňce v řádku pivota, která se postará o výpočet inverze.

Jestliže fronta ke čtení ještě není prázdná, znamená to že řídící buňka vložila do čtecí fronty další prvek ke čtení během právě probíhajícího čtení z paměti. Pokud tedy čtecí fronta není prázdná, pokračuje vlákno v činnosti se čtením dalších prvků. Jakmile je fronta vyprázdněná, vlákno vyvolá událost určenou k indikaci toho, že již byla fronta vyprázdněná a že byly přečteny všechny hodnoty z paměti. Na tuto událost může čekat řídící buňka, která pak může pokračovat v činnosti.

5.5.2 Realizace zapisovací fronty

Řídící buňka má pro zápis z paměti zapisovací frontu a zvláštní vlákno. Do této zapisovací fronty řídící buňka průběžně vkládá vektory složené z adres, na které se má zapisovat, a dat, které budou zapisovány.

Vlákno se probudí na událost, kterou řídící buňka oznámí, že vložila do fronty vektor k zapsání. Vlákno z fronty k zápisu přečte první prvek a zapíše ho do paměti. Vlákno vyčká na dokončení zápisu a odebere právě zapsaný prvek z fronty.

Jestliže fronta k zápisu není prázdná, znamená to že řídící buňka vložila do zapisovací fronty další prvek během právě probíhajícího zápisu. Pokud tedy zapisovací fronta není prázdná, pokračuje vlákno v činnosti se zápisem dalšího prvku. Jakmile je fronta vyprázdněná, vlákno vyvolá událost určenou k indikaci toho, že již byla fronta vyprázdněná a že byly zapsány všechny hodnoty z do paměti. Na tuto událost případně čeká řídící buňka, která může pokračovat v činnosti.

5.6 Realizace ALU buněk

V simulované architektuře se vyskytují 2 druhy ALU buněk. Buňka pro výpočet v řádku pivota a buňky pro eliminaci hodnot v ostatních řádcích. Pozice těchto buněk není pevně daná a data jsou k nim přiváděny průběžně. Řídící buňka však musí zajistit, aby nedocházelo k přivedení hodnot k buňkám před dokončením aktuálního výpočtu.

Implementačně je pro každou ALU buňku realizováno vlastní vlákno, které je pomocí staticky dynamických událostí probouzeno a uspáváno.

Simulace doby trvání výpočtu je zajištěna, tak že daná ALU buňka nezapíše na výstupní porty vypočtenou hodnotu dříve než za časový rámec určený v konfiguračním souboru.

5.6.1 Realizace buňky pro výpočet v řádku pivota

Úkolem této buňky je výpočet inverze a následné vynásobení řádku touto inverzí. Buňka má oddělenou část pro výpočet inverze a vynásobení inverzí. To je realizováno proto aby nedocházelo k předčasnému probuzení eliminačních buněk po výpočtu inverze. Dále buňka z důvodu potřeby probuzení ostatních výpočetních buněk umožňuje zkopírování dříve vypočteného výsledku na výstup. Tabulka 5.2 shrnuje funkce buňky, vstupní, výstupní porty a metody jimž je buňka citlivá na změny na vstupních portech. Buňka si ve své paměti drží výsledek poslední operace.

Port	**Citlivost metodou**	**Matematické vyjádření operace**
in_k_inverzi	*pocitej-Inverzi*	$out\_vystup\_inverze = \lvert in\_k\_inverzi^{-1} \rvert_m$
in_vstup	*pocitej*	$out\_vystup = \lvert in\_vstup \cdot inverze \rvert_m$
	kopiruj-Vysledek	$out\_vystup = vysledek$
out_vystup_inverze		
out_vystup		

Tabulka 5.2: Realizace buňky pro výpočet v řádku pivota

5.6.2 Realizace eliminačních buněk

Úkolem těchto buněk je eliminace řádků, jenž nejsou pivotem. Buňka potřebuje ke své činnosti tři hodnoty. Hodnotu z řádku pivota, hodnotu z aktuálně prvního sloupce daného řádku a samotnou vstupní hodnotu. Aktuálně první sloupec je dán aktuální fází výpočtu. Buňka nemůže zahájit činnost dříve než jsou k dispozici všechny 3 hodnoty a musí být schopna odlišit zda jsou již k dispozici tyto hodnoty, nebo ne.

Buňka je citlivá výpočetní metodou na změny na všech 3 portech. Zároveň pro každý port je zde zvláštní metoda starající se o načtení hodnot z daného portu. Jestliže je výpočetní metoda probuzena změnou na jednom z portů, vyčká na dokončení načítání dat ze vstupního portu. Poté zkontroluje zda jsou již k dispozici všechny 3 potřebné hodnoty, jestliže ještě nejsou k dipozici, znamená to že buňka musí dále vyčkat na další probuzení, další hodnotou. Pokud již jsou data k dispozici, pokračuje v činnosti k samotnému výpočtu. Pokud stále nejsou k dispozici ostatní hodnoty, buňka dále vyčká na jejích načtení.

Pro každý port je zde samostatné vlákno zajišťující čtení z daného portu. Tyto vlákna jsou citlivé na změnu na daném portu. Jestliže nastane změna na portu, čtecí vlákno načte hodnotu a vyvolá událost indikující, že již načetlo hodnotu z portu.

5.7 Průběh nalezení řešení

Při využití simulačního modelu se můžeme omezit na několik případů, každý případ bude rozebrán podrobněji popsán z hlediska rozdílů průběhu nalezení řešení soustavy. Dále se můžeme omezit na určité předpoklady a specifické vlastnosti simulačního modelu:

- Máme k dispozici alespoň tolik ALU jednotek, jako je šířka paměťové sběrnice. Maximálně máme k dispozici tolik ALU jednotek, jako je počet řádků ve sloupci.
- Počet řádků soustavy je vždy násobkem šířky sběrnice. Pokud by tomu tak nebylo, samotná simulace by nemusela proběhnout korektně a bylo by nutné v ní ošetřovat výpočty zbývajících hodnot.
- Může docházet k neužitečnému využití jednotek. To je způsobeno nutností vždy dopočítat vždy celý sloupec. Neužitečná jednotka je taková pro kterou v daném kroku již nemáme vhodná data.
- Výpočet probíhá vždy pouze na úrovni jednoho sloupce. Není možné započíst výpočet v dalším sloupci, dokud není vypočten celý sloupec.
- Z paměti nelze zároveň číst a zapisovat. Je vždy potřeba dokončit jednu paměťovou operaci před provedením další.
- Řídící buňka má lokální paměť určenou k uložení inverze a hodnot eliminačních členů. Tato paměť má kapacitu takovou aby to ní bylo možné uložit celý tento řídící sloupec.
- Počet ALU jednotek je počet aktivních ALU jednotek nezávisle na funkci. Pokud by tomu tak nebylo, musela by ALU jednotka provádějící výpočet v řádku pivota být schopná stejného výpočtu jako ostatní ALU jednotky. Díky tomuto omezení a omezení na výpočet v aktivním sloupci nelze vypočíst modulární inverzi dříve než je započato se zpracováváním prvního sloupce.
- ALU jednotky začnou výpočet okamžitě jakmile jsou k nim přivedeny data. Výpočet trvá všem ALU jednotkám stejnou dobu definovanou v parametrech simulačního modelu.

- Řídící buňka při načítání z paměti ví, kde se nalézá příští pivot a nepotřebuje pro jeho načtení procházet celý sloupec. Pouze při načtení dané hodnoty předá tuto hodnotu buňce v řádku pivota.

- Řídící buňka nemůže udržovat dříve načtené hodnoty z paměti ve svojí paměti, udržuje pouze řídící sloupec výpočtu, determinant, pomocné indexy a pomocné indexy. Hodnoty z paměti nelze přednačíst a předat ALU buňkám během jejich činnosti.

Samotné hledání řešení se skládá z několika téměř stejných fází (eliminačních kroků). Každá fáze se skládá z několik různých kroků, tyto kroky se vždy mírně liší. Je potřeba vyřešit chování během čtyř různých situací v závislosti na počtu řádků soustavy, počtu ALU jednotek a šířky paměťové sběrnice. Nyní podrobně rozebereme průběh nalezení řešení pro jednotlivé situace. Každá situace je definována následujícími proměnnými:

- n Počet neznámých soustavy
- $\#_{ALU}$ Počet ALU jednotek
- $\#_{Bus}$ Šířka paměťové sběrnice

Tyto proměnné mezi sebou mají následující vztahy (mod zde značí zbytek po dělení):

- $n \geq \#_{ALU}$
- $\bmod(n, \#_{ALU}) = 0$
- $\#_{ALU} \geq \#_{Bus}$
- $\bmod(\#_{ALU}, \#_{Bus}) = 0$
- $\#_{Bus} \geq 2$

Tabulka 5.3 ukazuje, jaké byly realizované simulační modely v závislosti na proměnných. Tabulka 5.4 ukazuje, příklad pro soustavu lineárních kongruencí o 16 neznámých.

U všech realizovaných simulačních modelů je vždy společná inicializační fáze, během které se načítá první sloupec soustavy, vypočítává se inverze a případně pokud není pivot na daném řádku, probíhá zde tento přesun.

Situace	**Počet neznámých soustavy, počet ALU jednotek, šířka paměťové sběrnice**
Situace 1	$n = \#_{ALU} = \#_{Bus}$
Situace 2	$n > \#_{ALU} = \#_{Bus}$
Situace 3	$n = \#_{ALU} > \#_{Bus}$
Situace 4	$n > \#_{ALU} > \#_{Bus}$

Tabulka 5.3: Realizované simulační modely s ohledem na proměnné modelu

Situace	**Počet neznámých soustavy, počet ALU jednotek, šířka paměťové sběrnice**
Situace 1	$(n = 16) = (\#_{ALU} = 16) = (\#_{Bus} = 16)$
Situace 2	$(n = 16) > (\#_{ALU} = 8) = (\#_{Bus} = 8)$
Situace 3	$(n = 16) = (\#_{ALU} = 16) > (\#_{Bus} = 4)$
Situace 4	$(n = 16) > (\#_{ALU} = 12) > (\#_{Bus} = 4)$

Tabulka 5.4: Realizované simulační modely - příklad pro soustavu s 16 neznámými

5.7.1 Inicializační fáze

Pro každou situaci je v každé fázi nejdříve provedena inicializační fáze. Během této fáze je načten řídící sloupec, vypočtena inverze a determinant je vynásoben pivotem.

1. Načti řídící sloupec.
2. V závislosti na šířce paměťové sběrnice vždy vkládej do fronty ke čtení adresu k načtení. Do fronty adres buněk vkládej adresy v řídícím sloupci.
3. Vlákno zpracovávající frontu ke čtení, místo poslání načtené hodnoty eliminačním buňkám, uloží hodnoty do řídícího sloupce.
4. **if** Je-li známá pozice pivota, pošli danou hodnotu buňce pro výpočet inverze.
5. **if** Není-li známá pozice pivota, najdi jeho pozici v řídícím sloupci.
6. Počkej na dokončení výpočtu inverze.
7. **if** Není-li fronta ke čtení prázdná, počkej na dokončení načítání.
8. **if** Není-li pivot na řádku odpovídající aktuální fázi,

zaměň řádek s pivotem s řádkem odpovídající aktuální fázi.
9. Vynásob determinant pivotem, zmoduluj a případně uprav znaménko.

V kroku 8 vidíme, že zde může docházet k záměně řádků s pivotem, respektive části řádku s pivotem.

5.7.2 Situace 1: $n = \#_{ALU} = \#_{Bus}$

Jedná se o nejjednodušší situaci. Řídící buňka a ALU jednotky v provedou celkem n fází a v každé fázi k provedou následující kroky:

1. $i = k$
2. Inicializační fáze, popis viz výše.
3. $i++$
4. **while** $i \neq n + 1$
5. Z paměti načti celý sloupec i.
6. Předej ALU jednotkám sloupec i, řídící sloupec a hodnotu inverze.
7. ALU jednotka pro výpočet v řádku pivota vynásobí přijatou hodnotu inverzí.
8. Ostatní ALU jednotky počkají na výpočet z řádku pivota, po obdržení této hodnoty odečtou eliminační člen od násobku z řádku pivota s hodnotou v řídícím sloupci.
9. Řídící buňka počká na dokončení výpočtu ALU jednotkami.
10. Řídící buňka zapíše do paměti celý vypočtený sloupec i.
11. Řídící buňka během zápisu zkontroluje zda bude v další fázi potřeba zaměnit řádek na pozici $k + 1$. Pokud ano, při ukládání nalezne prvek jenž může být pivotem a uloží si jeho pozici do lokální paměti.
12. $i++$
13. **end while**

Výpočet se skládá z celkem n fází popsaných výše. Tyto fáze jsou stejné, jen první fáze je odlišná tím, že není známá pozici pivota. V každé další fázi řídící buňka již ví, zda bude potřeba zaměňovat řádky mezi sebou nebo ne. Každá fáze začíná o sloupec později. Po provedení poslední fáze je výsledek uložen v posledním sloupci.

5.7.3 Situace 2: $n > \#_{ALU} = \#_{Bus}$

Jedná se o situaci kdy máme méně ALU jednotek, ale každá ALU jednotka má k dispozici svojí část paměťové sběrnice. Řídící buňka a ALU jednotky v provedou celkem n fází a v každé fázi k provedou následující kroky:

1. $i = k$
2. Inicializační fáze, popis viz výše.
3. $i++$
4. **while** $i \neq n + 1$
5. **while** Dokud nejsou zpracovány všechny části sloupce
6. Z paměti načti část sloupce i. Začni s částí obsahující řádek k.
7. Předej ALU jednotkám část sloupce i a řídícího sloupce.
8. ALU jednotka pro výpočet v řádku pivota vynásobí přijatou hodnotu inverzí.
9. Ostatní ALU jednotky buď počkají na výpočet z řádku pivota nebo obdrží uloženou hodnotu z řádku s pivotem, po obdržení této hodnoty odečtou eliminační člen od násobku z řádku pivota s hodnotou v řídícím sloupci.
10. Řídící buňka počká na dokončení části výpočtu ALU jednotkami.
11. Řídící buňka zapíše do paměti vypočtenou část sloupce i.
12. Řídící buňka během zápisu zkontroluje zda bude v další fázi potřeba zaměnit řádek na pozici $k + 1$. Pokud ano, při ukládání nalezne prvek jenž může být pivotem a uloží si jeho pozici do lokální paměti.
13. Řídící buňka si případně uloží výsledek z řádku s pivotem do lokální paměti.
14. **end while**
15. $i++$
16. **end while**

Výpočet se skládá z celkem n fází popsaných výše. Tyto fáze jsou stejné, jen první fáze je odlišná tím, že není známá pozici pivota. V každé další fázi řídící buňka již ví, zda bude potřeba zaměňovat řádky mezi sebou nebo ne. Každá fáze začíná o sloupec později. Po provedení poslední fáze je výsledek uložen v posledním sloupci.

Výpočet je dělen do částí, podle šířky paměťové sběrnice. Pro každý sloupec musíme provést několik částí výpočtu v závislosti na tom jakou část výpočtu můžeme provést v dané fázi. Tuto část výpočtu musíme vždy

uložit do paměti než načteme a přivedeme k ALU jednotkám další hodnoty. Výpočet vždy začíná v části sloupce, kde je obsažen řádek s pivotem, tak aby další výpočty v další části sloupce využili lokální hodnotu uloženou v řídící buňce.

5.7.4 Situace 3: $n = \#_{ALU} > \#_{Bus}$

Tato situace nastane, pokud máme dostatek ALU jednotek, ale užší paměťovou sběrnici. Řídící buňka musí postupně zásobovat ALU jednotky a po dokončení zapsat dokončený sloupec do paměti. Řídící buňka a ALU jednotky v každé fázi k provedou následující kroky:

1. $i = k$
2. Inicializační fáze, popis viz výše.
3. $i++$
4. **while** $i \neq n + 1$
5. **while** Dokud nejsou načteny všechny části sloupce
6. Z paměti načti část sloupce i. Začni s částí obsahující řádek k.
7. Předej ALU jednotkám část sloupce i a řídícího sloupce.
8. ALU jednotka pro výpočet v řádku pivota vynásobí přijatou hodnotu inverzí.
9. Ostatní ALU jednotky počkají na výpočet z řádku pivota po obdržení této hodnoty odečtou eliminační člen od násobku z řádku pivota s hodnotou v řídícím sloupci.
10. Řídící buňka si uloží výsledek z řádku s pivotem do lokální paměti.
11. Řídící buňka postupně načte další části sloupce a předá je odpovídajícím ALU jednotkám a ty zahájí činnost v okamžiku obdržení odpovídající hodnoty. Řídící buňka nyní postupuje při načítání z paměti odshora sloupce.
12. **end while**
13. Řídící buňka zapíše do paměti sloupec i. Při zápisu postupuje, odshora sloupce, ale vždy kontroluje zda jsou výpočty ALU jednotkami již dokončeny.
14. Řídící buňka během zápisu zkontroluje zda bude v další fázi potřeba zaměnit řádek na pozici $k + 1$. Pokud ano, při ukládání nalezne prvek jenž může být pivotem a uloží si jeho pozici do lokální paměti.
15. $i++$
16. **end while**

Výpočet se skládá z celkem n fází popsaných výše. Tyto fáze jsou stejné, jen první fáze je odlišná tím, že není známá pozici pivota. V každé další fázi řídící buňka již ví, zda bude potřeba zaměňovat řádky mezi sebou nebo ne. Každá fáze začíná o sloupec později. Po provedení poslední fáze je výsledek uložen v posledním sloupci.

Každá fáze je ještě rozdělena na několik částí. Nejdříve načteme část sloupce s řádkem pivota, abychom získali hodnotu z tohoto řádku. Jakmile je načtena tato část sloupce, budeme načítat hodnoty od prvního řádku v daném sloupci. Protože jsem vycházeli z předpokladu, že načtení dalších buněk v paměti je řádově kratší než načtení nesousedních buněk, v sekvenčním zápisu znovu načteme i řádek s pivotem ale jeho hodnoty nepředáme ALU jednotkám. Tímto ušetříme nutnost náhodného čtení z paměti. Kdybychom pokračovali v načítání v části až pod částí řádkem pivota, museli bychom načítat, že se bude jednat o náhodný přístup do paměti.

Abychom nemuseli na hodnoty čekat, zvolili jsme, že nejdříve načteme řádek s pivotem a pak budeme zásobovat ALU jednotky ve vrchní části sloupce. Zápis pak může probíhat bez nutnosti náhodného přístupu do paměti.

5.7.5 Situace 4: $n > \#_{ALU} > \#_{Bus}$

Tato situace nastane, pokud máme dostatek ALU jednotek, ale užší paměťovou sběrnici. Řídící buňka musí postupně zásobovat ALU jednotky a v průběhů výpočtu zapisovat již dokončené části sloupce do paměti. Řídící buňka a ALU jednotky v každé fázi k provedou následující kroky:

1. $i = k$
2. Inicializační fáze, popis viz výše.
3. $i++$
4. **while** $i \neq n + 1$
5. Načti nejdříve z paměti takovou část dat, kterou lze naplnit všechny ALU jednotky a zároveň obsahuje data pro výpočet v řádku pivota.
6. Počkej na dokončení výpočtu zásobenými ALU jednotkami.
7. Ulož tuto již vypočtenou část zpět do paměti.
8. **while** Dokud jsou k dispozici nevypočítané části sloupce:
9. Zpracuj nejdříve část matice pod již vypočteným sloupcem.
10. Vždy načti potřebný počet buněk z paměti a počkej na výsledek ALU jednotek a ten ulož.
11. Pokud již není k dispozici dostatek dat ze spodní části matice, přejdi na horní část matice.

12. **end while**
13. Jakmile již nezbývají žádná další data pro výpočet v horní ani spodní části matice, přesuň se do dalšího sloupce.
14. Řídící buňka během zápisu kontroluje zda bude v další fázi potřeba zaměnit řádek na pozici $k + 1$. Pokud ano, při ukládání nalezne prvek jenž může být pivotem a uloží si jeho pozici do lokální paměti.
15. **end while**

Výpočet se skládá z celkem n fází popsaných výše. Tyto fáze jsou stejné, jen první fáze je odlišná tím, že není známá pozici pivota. V každé další fázi řídící buňka již ví, zda bude potřeba zaměňovat řádky mezi sebou nebo ne. Každá fáze začíná o sloupec později. Po provedení poslední fáze je výsledek uložen v posledním sloupci.

Každá fáze je ještě rozdělena na několik částí. Nejdříve načteme část sloupce s řádkem pivota a k tomu odpovídající počet dalších dat z paměti. Získáme hodnotu z řádku s pivotem a vytížíme i další ALU buňky. Předejde se tak tomu, aby nebyly všechny dostupné ALU zapojené do výpočtu v této části výpočtu. Jakmile je zpracována tato část, budeme pokračovat ve výpočtu v sekci pod touto vypočtenou částí. Jakmile již není dostatek dat ve spodní části sloupce, pokračujeme ve výpočtu s částí sloupce nad, v prvním kroku, vypočtenou části sloupce. Jakmile v částech pod i nad částí, kterou jsme vypočetli jako první, přesuneme se do dalšího sloupce. Vždy střídáme čtecí, výpočetní a zapisovací fázi.

Vzhledem k tomu, že u této situace je vždy méně ALU buněk než je šířka paměťové sběrnice, během výpočtu mohou nastat 3 různé situace vzhledem k tomu v jaké fázi výpočtu se nacházíme. V každé této situaci je v prvním kroku potřeba získat výsledek z řádku s pivotem, proto je potřeba tuto hodnotu vypočítat jako první. Protože je však potřeba naplnit více ALU buněk než je šířka paměťové sběrnice a předpokládáme, že načtení sousedních dat z paměti je řádově rychlejší než načtení nesousedních dat, je žádoucí i v prvním kroku načíst hodnoty pro všechny ALU buňky, nejen pro buňku, která zajišťuje výpočet v řádku pivota.

Toho je dosaženo rozdělením sloupce na úseky o velikosti rovnající se počtu buněk. První úsek vždy obsahuje řádek s pivotem. Výpočet pak probíhá v úseku nad úsekem s pivotem, a v úseku pod ním. Protože počet ALU buněk nemusí nutně dělit počet řádků matice, je nutné ošetřit výpočet při pozdějších fázích výpočtu, tak abychom využili všechny ALU buňky již v prvním kroku.

Tabulka 5.5 ukazuje na příkladu pro n=16, $\#_{\text{ALU}}$=6, $\#_{\text{sběrnice}}$=2, které části sloupce jsou v závislosti na aktuální fázi vypočteny jako první.

Fáze 0-5		**Fáze 6-11**		**Fáze 11-15**
0		0		0
1		1		1
2		2		2
3		3		3
4		4		4
5		5		5
6		**6**		6
7		**7**		7
8		**8**		8
9		**9**		9
10		**10**		**10**
11		**11**		**11**
12		12		**12**
13		13		**13**
14		14		**14**
15		15		**15**

Tabulka 5.5: První vypočtená sekce v různých fázích výpočtu pro n=16, $\#_{\text{ALU}}$=6, $\#_{\text{sběrnice}}$=2

Pro lepší optimalizaci vytížení ALU buněk jsou pro pozdější fáze výpočtu realizována další vylepšení. Pro využití všech ALU buněk je realizováno načtení dat z horní části pro ALU buňky, které již nemají data nalézající se pod dříve vypočtenou sekcí. V posledních fázích výpočtu, kdy by již nebylo dostatek dat v části obsahující data z řádku s pivotem, je tato část vždy posunuta, tak aby vždy v prvním kroku byly zásobeny daty všechny ALU buňky.

5.8 Měření charakteristik simulované architektury

Měření charakteristik je provedeno na základě analýzy simulačního času potřebného pro nalezení řešení soustavy dané velikosti. Pro zjištění charakteristik námi simulované architektury bylo změřeno celkem 6 různých parametrizovaných situací. Všechny měření byly provedeny pro řešení náhodně vygenerované soustavy o 512 neznámých v modulu $2^{20}+7$. Tento modul byl zvolen, záměrně, protože se jedná o 21 bitové číslo, proto je v měření velikost operandu vždy 21. Všechny měření mají společnou periodu hodinového

signálu 10 ns.

Tabulka 5.6 zachycuje zvolené fixní parametry vzhledem k charakteru daného problému.

Počet neznámých soustavy	$n = 512$
Velikost modulu	$m = 1048583$ $(2^{20} + 7)$
Velikost operandu	$k = 21$ bitů
Perioda hodinového signálu	$T_{clk} = 10$ ns
Čas výpočtu modulární inverze	$T_{^{-1}} = \lceil (0 \cdot 21^2 + 2, 2 \cdot 21 + 0) \rceil \cdot T_{clk} = 470$ ns
Čas výpočtu modulárního odčítání	$T_{-} = \lceil (0 \cdot 21^2 + 0 \cdot 21 + 1) \rceil \cdot T_{clk} = 10$ ns

Tabulka 5.6: Fixní parametry simulačního modelu využité při měření charakteristik

Tabulka 5.7 zachycuje zvolené vstupní konfigurační parametry modelu využitého při měření charakteristik.

Čas výpočtu modulární násobení	$T_{\cdot} = \lceil (a \cdot k^2 + b \cdot k + c \rceil \cdot T_{clk}$
Doba sekvenčního čtení	$T_R = c \cdot T_{clk}$ ns
Doba sekvenčního zápisu	$T_W = c \cdot T_{clk}$ ns
Doba náhodného čtení	$T_{nR} = c \cdot T_{clk}$ ns
Doba náhodného zápisu	$T_{nW} = c \cdot T_{clk}$ ns

Tabulka 5.7: Konfigurační parametry simulačního modelu využité při měření charakteristik

Bližší informace k daným parametrům lze nalézt v sekci 5.1. U všech měření je potřeba vzít v úvahu, že se stoupajícím počtem ALU buněk, stoupá i šířka paměťové sběrnice. Tabulka 5.14 zobrazuje naměřené simulační časy pro všechny provedené měření.

5.8.1 Vyhodnocení vlivu rychlosti násobení operandu na rychlost výpočtu

Pro vyhodnocení vlivu závislosti doby výpočtu na počtu ALU buněk bylo provedeno měření při podmínkách zaznamenaných v tabulce 5.8. Naměřené

výsledky pro vybrané počty ALU buněk a šířku paměťové sběrnice jsou zaznamenány v tabulce 5.9. Situace kdy je doba výpočtu závislá na velikosti operandu je zaznamená jako „Násobení s vel. operandu“, situace kdy je doba násobení konstantní je označena jako „Rychlé násobení“. Vzhledem k charakteru problému jsme vycházeli z předpokladu, že největší vliv na dobu výpočtu má rychlost provedení operace modulární násobení.

Násobení s vel. operandu	**Rychlé násobení**
$T_{\cdot} = (0 \cdot 21^2 + 1 \cdot 21 + 0) \cdot T_{clk}$	$T_{\cdot} = \lceil (0 \cdot 21^2 + 0 \cdot 21 + 4) \rceil \cdot T_{clk}$
$T_R = 1 \cdot T_{clk}$ ns	$T_R = 1 \cdot T_{clk}$ ns
$T_W = 1 \cdot T_{clk}$ ns	$T_W = 1 \cdot T_{clk}$ ns
$T_{nR} = 5 \cdot T_{clk}$ ns	$T_{nR} = 5 \cdot T_{clk}$ ns
$T_{nW} = 5 \cdot T_{clk}$ ns	$T_{nW} = 5 \cdot T_{clk}$ ns

Tabulka 5.8: Konfigurační parametry simulačního modelu využité při měření charakteristik pro soustavu o 512 neznámých

V tabulce 5.9 vidíme námi naměřené hodnoty. Čas_1 je čas pro výpočet, kdy doba k provedení operace násobení závisí na velikosti operandu. Čas_2 je čas pro výpočet, kdy doba násobení nezávisí na velikosti operandu, ale je konstantní.

Z tabulky vidíme, že pokud je doba násobení závislá na velikosti operandu, je tato situace nevhodná pokud se jedná o větší operand. Pro tuto situaci jsou výpočetní časy nejvíce závislé pouze na počtu ALU buněk. Toto je způsobeno, tím že v námi realizovaném experimentu byla doba potřebná k výpočtu násobení 210 ns (21 taktů). Náhodný zápis/čtení z paměti však trvalo 50 ns (5 taktů). Reálně tedy nezáleží na tom zda architektura přistupuje sekvenčně nebo náhodně do paměti, a jak často. Proto je výsledná doba výpočtu závislá především na počtu ALU buněk, nikoliv na šířce paměťové sběrnice. Ve sloupci „Zlepšení“ je o kolik se výpočet zrychlí využijeme-li násobičku takovou, kdy doba výpočtu nezávisí na velikosti operandu.

Pro situaci kdy využijeme násobičku, takovou kde doba výpočtu nezávisí na velikosti operandu, vidíme že doba výpočtu je ovlivněna mnohem více rychlostí přístupu do paměti než u předchozí situace. Ve posledním sloupci tabulky 5.9 označeném jako „Zpomalení“ je uvedeno prodloužení doby výpočtu při stejném počtu ALU buněk ale pro užší paměťovou sběrnici.

Graf 5.2 zobrazuje srovnání pro použití rychlostí pro různé násobičky.

V grafu jsou modře znázorněny výpočetní časy při použití rychlé násobičky a oranžově při použití pomalé násobičky, kdy je rychlost závislá na velikosti operandu. Jednotlivé datové řady jsou zobrazeny podle poměrů $\frac{\texttt{počtu ALU buněk}}{\texttt{šířce paměťové sběrnice}}$.

$\#_{ALU}$	$\#_{Bus}$	Čas$_1$ $[ms]$	Čas$_2$ $[ms]$	Zlepšení	Zpomalení
64	16	363,53 ms	162,60 ms	123,57%	22,17%
64	32	341,23 ms	140,29 ms	143,22%	5,41%
64	64	334,02 ms	133,09 ms	150,97%	0,00%
128	16	221,86 ms	118,11 ms	87,84%	64,39%
128	32	198,41 ms	86,78 ms	128,63%	20,78%
128	64	186,69 ms	75,06 ms	148,71%	4,47%
128	128	183,48 ms	71,85 ms	155,36%	0,00%
256	16	151,61 ms	100,11 ms	51,44%	170,24%
256	32	126,18 ms	61,83 ms	104,08%	66,90%
256	64	113,48 ms	46,50 ms	144,03%	25,52%
256	128	107,15 ms	40,17 ms	166,73%	8,43%
256	256	104,02 ms	37,05 ms	180,80%	0,00%
512	16	110,34 ms	101,15 ms	9,09%	406,60%
512	32	88,97 ms	58,77 ms	51,40%	194,34%
512	64	77,85 ms	37,14 ms	109,62%	86,02%
512	128	71,52 ms	27,61 ms	159,05%	38,28%
512	256	67,25 ms	22,93 ms	193,33%	14,83%
512	512	64,62 ms	19,97 ms	223,64%	0,00%

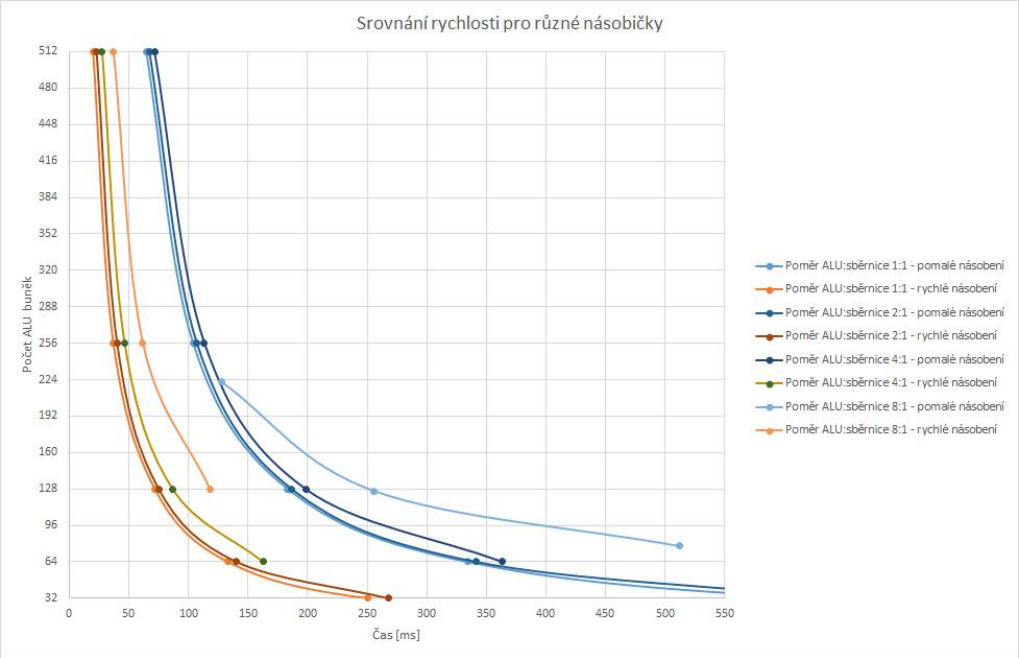
Tabulka 5.9: Naměřené výsledky pro analýzu doby výpočtu v závislosti na rychlosti násobení pro soustavu o 512 neznámých

Z grafu jasně vidíme, že rozdíl mezi rychlostí výpočtu, při použití různých násobiček je velmi výrazný.

5.8.2 Vliv rychlosti paměťových operací

Pro další měření bylo vycházeno z předpokladu o výrazné závislosti simulačního času na rychlosti provedení modulárního násobení, proto byla pro následující měření využívána násobička s konstantní dobou výpočtu nezávislé na velikosti operandu, aby rychlost výpočtu nebyla ovlivněna rychlostí modulárního násobení.

Samotná měření byla zaměřena na parametry související s přístupem do paměti, na vyhodnocení vlivu zpomalení sekvenčního a náhodného přístupu do paměti.



Obrázek 5.2: Graf srovnání rychlosti výpočtu při použití rychlé a pomalé násobičky

5.8.2.1 Vyhodnocení vlivu zpomalení sekvenčního přístupu do paměti

Tabulka 5.10 zobrazuje zvolené konfigurační proměnné pro námi provedené měření. V tomto měření bylo účelem vyhodnotit vliv zpomalení sekvenčního přístupu do paměti a navrhnout ideální poměr počtu ALU buněk k šířce paměťové sběrnice.

Tabulka 5.11 zobrazuje námi naměřené hodnoty. Hodnoty ve sloupci $\frac{T_{R/W=1}}{T_{nR/nW=10}}$ jsou pro konfiguraci kdy sekvenční přístup do paměti trval 1 takt a náhodný přístup 10 taktů. Obdobně pro sloupce $\frac{T_{R/W=2}}{T_{nR/nW=10}}$, $\frac{T_{R/W=1}}{T_{nR/nW=32}}$ a $\frac{T_{R/W=2}}{T_{nR/nW=32}}$. Sloupec „Zp. $\frac{1/10}{2/10}$“ vyjadřuje o kolik se zpomalí výpočet pokud doba sekvenčního přístupu nebude 1 takt, ale 2 takty.

Z tabulky vidíme, že čím máme užší paměťovou sběrnici tím je vliv rychlosti sekvenčního přístupu větší. Stejný počet výpočetních ALU buněk potřebuje přistupovat k paměti, častěji, protože paměťová sběrnice není schopná vše přenést najednou.

Dále vidíme, že zvětšíme-li šířku paměťové sběrnice dvakrát, můžeme, při zachování přibližně stejného výpočetního času, využít datovou sběrnici s dvakrát pomalejším sekvenčním přístupem do paměti.

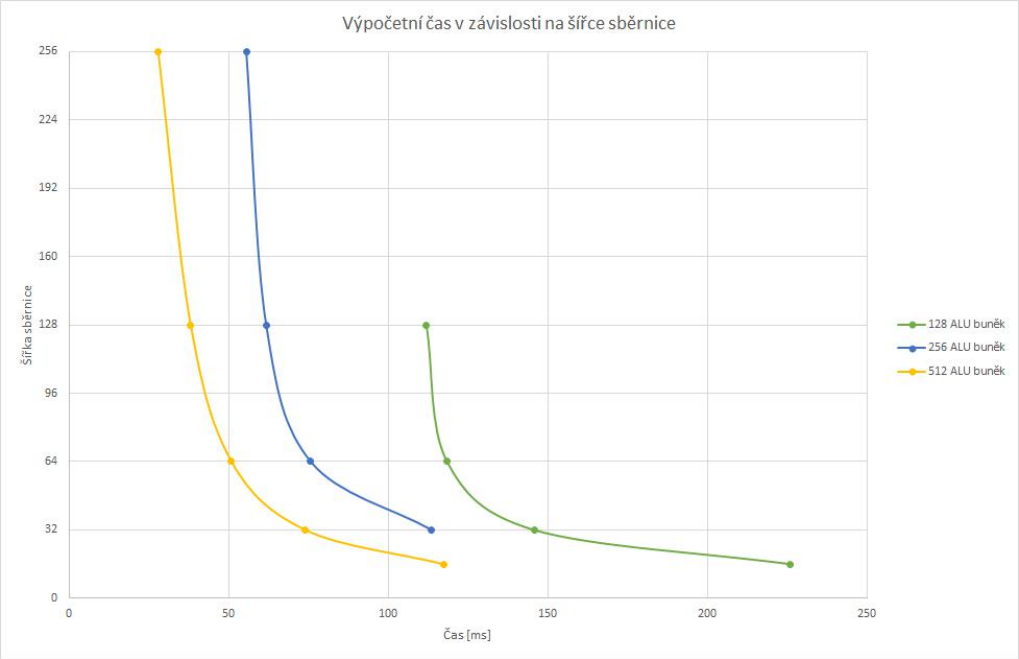
Sekvenční paměťová operace	**1 takt**	**2 takty**	**1 takt**	**2 takty**
Náhodná paměťová operace	**10 taktů**	**10 taktů**	**32 taktů**	**32 taktů**
$T.$	$4 \cdot T_{clk}$ ns	$4 \cdot T_{clk}$ ns	$4 \cdot T_{clk}$ ns	$4 \cdot T_{clk}ns$
T_R	$1 \cdot T_{clk}$ ns	$2 \cdot T_{clk}$ ns	$1 \cdot T_{clk}$ ns	$2 \cdot T_{clk}$ ns
T_W	$1 \cdot T_{clk}$ ns	$2 \cdot T_{clk}$ ns	$1 \cdot T_{clk}$ ns	$2 \cdot T_{clk}$ ns
T_{nR}	$10 \cdot T_{clk}$ ns	$10 \cdot T_{clk}$ ns	$32 \cdot T_{clk}$ ns	$32 \cdot T_{clk}$ ns
T_{nW}	$10 \cdot T_{clk}$ ns	$10 \cdot T_{clk}$ ns	$32 \cdot T_{clk}$ ns	$32 \cdot T_{clk}$ ns

Tabulka 5.10: Konfigurační parametry simulačního modelu využité při měření charakteristik pro sekvenční přístup do paměti pro soustavu o 512 neznámých

$\#_{ALU}$	$\#_{Bus}$	$\frac{T_{R/W}=1}{T_{nR/nW}=10}$	$\frac{T_{R/W}=2}{T_{nR/nW}=10}$	**Zp.** $\frac{1/10}{2/10}$	$\frac{T_{R/W}=1}{T_{nR/nW}=32}$	$\frac{T_{R/W}=2}{T_{nR/nW}=32}$	**Zp.** $\frac{1/32}{2/32}$
64	16	225,21 ms	276,38 ms	22,72%	500,72 ms	551,88 ms	10,22%
64	32	202,91 ms	222,56 ms	9,69%	478,41 ms	498,07 ms	4,11%
64	64	200,63 ms	208,16 ms	3,75%	497,80 ms	505,33 ms	1,51%
128	16	151,80 ms	225,93 ms	48,83%	300,00 ms	374,13 ms	24,71%
128	32	120,46 ms	145,74 ms	20,98%	268,67 ms	293,95 ms	9,41%
128	64	108,75 ms	118,37 ms	8,85%	256,95 ms	266,57 ms	3,74%
128	128	108,82 ms	111,94 ms	2,87%	271,47 ms	274,60 ms	1,15%
256	16	116,56 ms	196,08 ms	68,22%	188,93 ms	268,45 ms	42,09%
256	32	78,28 ms	113,62 ms	45,15%	150,65 ms	185,99 ms	23,46%
256	64	62,95 ms	75,69 ms	20,24%	135,32 ms	148,06 ms	9,41%
256	128	56,62 ms	61,71 ms	9,00%	128,99 ms	134,09 ms	3,95%
256	256	53,49 ms	55,46 ms	3,68%	125,86 ms	127,84 ms	1,57%
512	16	120,47 ms	202,06 ms	67,73%	205,48 ms	287,07 ms	39,71%
512	32	77,70 ms	117,29 ms	50,96%	160,99 ms	200,59 ms	24,60%
512	64	55,33 ms	74,04 ms	33,82%	135,35 ms	154,06 ms	13,82%
512	128	43,89 ms	50,87 ms	15,91%	118,05 ms	125,03 ms	5,92%
512	256	34,78 ms	38,06 ms	9,44%	99,91 ms	103,19 ms	3,29%
512	512	26,56 ms	27,87 ms	4,95%	55,56 ms	56,88 ms	2,36%

Tabulka 5.11: Naměřené výsledky pro analýzu doby výpočtu v závislosti na rychlosti sekvenčního přístupu do paměti pro soustavu o 512 neznámých

Graf 5.3 zobrazuje srovnání doby výpočtu pro různou šířku paměťové sběrnice pro různý počet ALU buněk.



Obrázek 5.3: Graf rychlosti výpočtu v závislosti na šířce paměťové sběrnice

V grafu jsou znázorněny výpočetní časy při použití daného počtu ALU s různou šířkou paměťové sběrnice. Vidíme stejný trend jako u předchozí grafu. Dále vidíme, že rozdíl mezi situací, kdy máme k dispozici 256 ALU buněk oproti situaci kdy máme k dispozici 512 není tak velký, jako rozdíl kdy máme k dispozici 256 ALU buněk oproti 128 ALU buňkám.

5.8.2.2 Vyhodnocení vlivu zpomalení náhodného přístupu do paměti

Tabulka 5.12 zobrazuje zvolené konfigurační proměnné pro námi provedené měření. V tomto měření bylo účelem vyhodnotit vliv zpomalení náhodného přístupu do paměti a navrhnout ideální poměr počtu ALU buněk k šířce paměťové sběrnice.

Tabulka 5.13 zobrazuje námi naměřené hodnoty. Hodnoty ve sloupci $\frac{T_{R/W=1}}{T_{nR/nW=5}}$ jsou pro konfiguraci kdy sekvenční přístup do paměti trval 1 takt a náhodný přístup 5 taktů. Obdobně pro sloupce $\frac{T_{R/W=1}}{T_{nR/nW=10}}$ a $\frac{T_{R/W=1}}{T_{nR/nW=32}}$. Sloupec „Zp. $\frac{1/5}{1/10}$“ vyjadřuje o kolik se zpomalí výpočet pokud doba náhodného přístupu nebude 5 taktů, ale 10 taktů.

Sekvenční paměťová operace	**1 takt**	**1 takt**	**1 takt**
Náhodná paměťová operace	**5 taktů**	**10 taktů**	**32 taktů**
$T.$	$4 \cdot T_{clk}$ ns	$4 \cdot T_{clk}$ ns	$4 \cdot T_{clk}$ ns
T_R	$1 \cdot T_{clk}$ ns	$1 \cdot T_{clk}$ ns	$1 \cdot T_{clk}$ ns
T_W	$1 \cdot T_{clk}$ ns	$1 \cdot T_{clk}$ ns	$1 \cdot T_{clk}$ ns
T_{nR}	$5 \cdot T_{clk}$ ns	$10 \cdot T_{clk}$ ns	$32 \cdot T_{clk}$ ns
T_{nW}	$5 \cdot T_{clk}$ ns	$10 \cdot T_{clk}$ ns	$32 \cdot T_{clk}$ ns

Tabulka 5.12: Konfigurační parametry simulačního modelu využité při měření charakteristik pro náhodný přístup do paměti pro soustavu o 512 neznámých

$\#_{ALU}$	$\#_{Bus}$	$\frac{T_{R/W=1}}{T_{nR/nW=5}}$	$\frac{T_{R/W=1}}{T_{nR/nW=10}}$	$\frac{T_{R/W=1}}{T_{nR/nW=32}}$	**Zp.** $\frac{1/5}{1/10}$	**Zp.** $\frac{1/10}{1/32}$	**Zp.** $\frac{1/5}{1/32}$
64	16	162,60 ms	225,21 ms	500,72 ms	38,51%	122,33%	207,94%
64	32	140,29 ms	202,91 ms	478,41 ms	44,63%	135,78%	241,01%
64	64	133,09 ms	200,63 ms	497,80 ms	50,75%	148,12%	274,03%
128	16	118,11 ms	151,80 ms	300,00 ms	28,52%	97,63%	153,99%
128	32	86,78 ms	120,46 ms	268,67 ms	38,81%	123,03%	209,60%
128	64	75,06 ms	108,75 ms	256,95 ms	44,87%	136,29%	242,32%
128	128	71,85 ms	108,82 ms	271,47 ms	51,45%	149,47%	277,82%
256	16	100,11 ms	116,56 ms	188,93 ms	16,43%	62,09%	88,72%
256	32	61,83 ms	78,28 ms	150,65 ms	26,60%	92,45%	143,65%
256	64	46,50 ms	62,95 ms	135,32 ms	35,37%	114,97%	191,00%
256	128	40,17 ms	56,62 ms	128,99 ms	40,95%	127,82%	221,11%
256	256	37,05 ms	53,49 ms	125,86 ms	44,40%	135,29%	239,76%
512	16	101,15 ms	120,47 ms	205,48 ms	19,10%	70,57%	103,15%
512	32	58,77 ms	77,70 ms	160,99 ms	32,21%	107,20%	173,95%
512	64	37,14 ms	55,33 ms	135,35 ms	48,97%	144,64%	264,44%
512	128	27,61 ms	43,89 ms	118,05 ms	58,97%	168,97%	327,57%
512	256	22,93 ms	34,78 ms	99,91 ms	51,69%	187,29%	335,79%
512	512	19,97 ms	26,56 ms	55,56 ms	33,02%	109,22%	178,29%

Tabulka 5.13: Naměřené výsledky pro analýzu doby výpočtu v závislosti na rychlosti náhodného přístupu do paměti pro soustavu o 512 neznámých

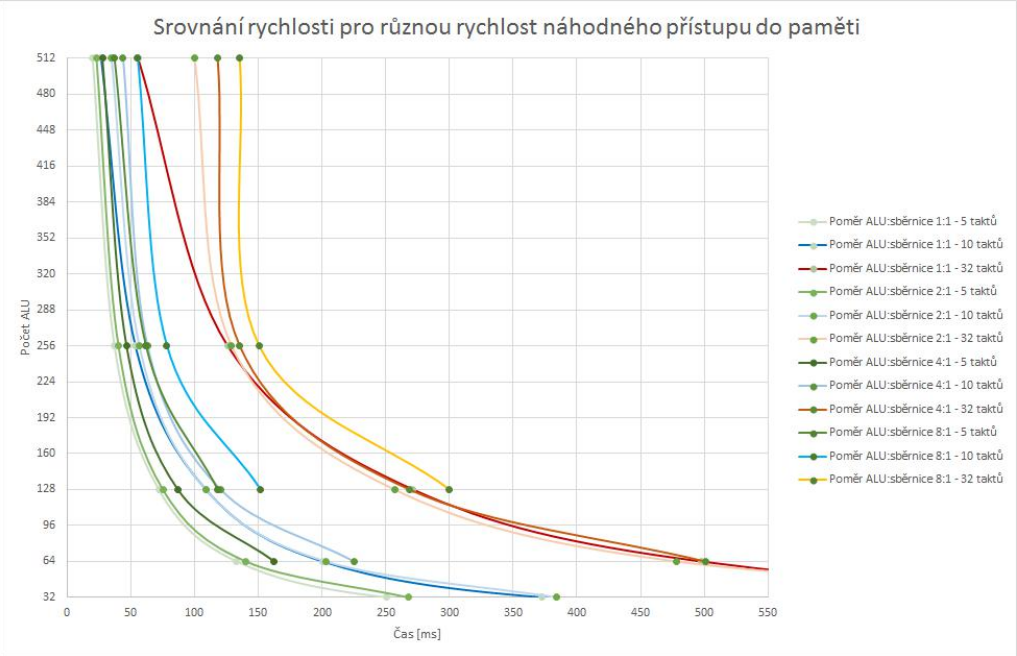
Z tabulky vidíme, že jako ideální poměr počtu ALU buněk k šířce paměťové sběrnice se jeví $\frac{1}{4}$, tzv. že pokud rozšíříme paměťovou sběrnice o 1 slovo, je ideální abychom přidali další 4 ALU buňky.

Pokud byl poměr menší dochází k tomu, že ALU buňky nedostávají data včas a musí na ně příliš čekat.

Pokud byl poměr větší, architektura naopak musí čekat na dokončení ALU operací a data jsou do paměti zapisována téměř okamžitě. Architektura tak na přístup do paměti téměř nečeká.

Zároveň u takového poměru nedochází k tak znatelnému prodloužení doby výpočtu pokud bude doba náhodného přístupu do paměti delší.

Graf 5.4 zobrazuje srovnání doby výpočtu pro různou dobu trvání náhodného přístupu do paměti.



Obrázek 5.4: Graf srovnání rychlosti výpočtu při použití paměti s různou dobou náhodného přístupu do paměti

V grafu jsou zeleně znázorněny výpočetní časy při použití paměti s dobou náhodného přístupu 5 taktů, modře s dobou 10 taktů a oranžově s dobou 32 taktů.Jednotlivé datové řady jsou zobrazeny podle poměrů $\frac{\texttt{počtu ALU buněk}}{\texttt{šířce paměťové sběrnice}}$.

Z grafu jasně vidíme, že rozdíl mezi rychlostí výpočtu, kdy doba náhodného přístupu do paměti trvá 5 taktů, oproti 10 taktům, není tak výrazný, jako pro 32 taktů. Pro hodnotu 512 ALU buněk s poměrem 1, tzv. šířku

sběrnice 512 vidíme, že omezující zde již není doba náhodného přístupu do paměti.

5.8.3 Kompletní naměřené výsledky

Tabulka 5.14 zobrazuje všechny naměřené výsledky pro námi provedené měření na simulačním modelu architektury.

Z tabulky 5.14 vidíme, že nejvíce dobu výpočtu ovlivňují následující sledované faktory:

1. Rychlost výpočtu modulárního násobení.
2. Rychlost náhodného přístupu do paměti.
3. Rychlost sekvenčního přístupu do paměti.

Z hlediska počtu ALU lze považovat pro soustavu o 512 neznámých hodnotu $\#_{ALU} = 512$ za teoretickou hodnotu, stejně tak lze předpokládat že se nejvíce bude vyskytovat situace s užší paměťovou sběrnicí a menším počtem ALU buněk než maximální hodnoty.

Situaci 1 lze tedy považovat za čistě teoretickou hodnotu, situaci 2 za krajní hodnotu, situace 3 a 4 lze považovat za reálné pokud bude k dispozici větší poměr výpočetních buněk než je šířka paměťové sběrnice.

Situace 4 se bude se s největší pravděpodobností vyskytovat nejčastěji. To je také vidět v tabulce, kdy pro tuto situaci bylo provedeno nejvíce měření.

Jak již bylo uvedeno v sekci 5.8.1 je z počtu operací nejvíce důležité, aby reálná architektura měla co nejkratší dobu výpočtu modulárního násobení. Zatímco operace modulární inverze je provedena v dané fázi (eliminačním kroku) pouze jednou, samotné násobení je v každé fázi provedeno mnohonásobně vícekrát. I když se počet násobení v námi navržené architektuře s každou další fázi snižuje, tak i v závěrečné fázi je nutné provést vynásobení celého sloupce.

Samotná rychlost odčítání byla v námi simulované architektuře konstantní a proti rychlosti násobení zanedbatelná. Simulační model však není na tomto předpokladu závislý. Lze simulovat i jiný čas násobení pomocí parametrů popsaných v sekci 5.1.

Díky tomu, že námi navržená architektura, podle podmínek definovaných v sekci 3.6, neumožňuje současný zápis a čtení z paměti, je důležité aby architektura byla schopna data efektivně nejdříve z paměti načíst, přivést k ALU buňkám a pak vypočtené hodnoty uložit zpět do paměti. Vzhledem k tomu, že při kombinaci sekvenčního čtení sloupce dat z paměti a výpočtu

Situace	$\#_{ALU}$	$\#_{Bus}$	**Pomalé násobení**	**Rychlé násobení**	$\frac{T_{R/W=1}}{T_{nR/nW=10}}$	$\frac{T_{R/W=2}}{T_{nR/nW=10}}$	$\frac{T_{R/W=1}}{T_{nR/nW=32}}$	$\frac{T_{R/W=2}}{T_{nR/nW=32}}$
2	16	16	1219,41 ms	482,66 ms	711,29 ms	749,71 ms	1717,27 ms	1755,69ms
4	32	16	647,16 ms	267,62 ms	384,25 ms	424,69 ms	897,43 ms	937,88ms
2	32	32	630,06 ms	250,53 ms	372,90 ms	390,50 ms	911,37 ms	928,96ms
4	64	16	363,53 ms	162,60 ms	225,21 ms	276,38 ms	500,72 ms	551,88ms
4	64	32	341,23 ms	140,29 ms	202,91 ms	222,56 ms	478,41 ms	498,07ms
2	64	64	334,02 ms	133,09 ms	200,63 ms	208,16 ms	497,80 ms	505,33ms
4	96	32	269,80 ms	114,38 ms	166,79 ms	188,19 ms	397,39 ms	418,78ms
4	112	16	257,33 ms	130,44 ms	175,22 ms	244,43 ms	372,27 ms	441,48ms
4	128	16	221,86 ms	118,11 ms	151,80 ms	225,93 ms	300,00 ms	374,13ms
4	128	32	198,41 ms	86,78 ms	120,46 ms	145,74 ms	268,67 ms	293,95ms
4	128	64	186,69 ms	75,06 ms	108,75 ms	118,37 ms	256,95 ms	266,57ms
2	128	128	183,48 ms	71,85 ms	108,82 ms	111,94 ms	271,47 ms	274,60ms
4	160	32	198,03 ms	88,25 ms	123,83 ms	153,85 ms	280,40 ms	310,42ms
4	192	64	150,02 ms	61,15 ms	88,17 ms	99,27 ms	207,04 ms	218,14ms
4	256	16	151,61 ms	100,11 ms	116,56 ms	196,08 ms	188,93 ms	268,45ms
4	256	32	126,18 ms	61,83 ms	78,28 ms	113,62 ms	150,65 ms	185,99ms
4	256	64	113,48 ms	46,50 ms	62,95 ms	75,69 ms	135,32 ms	148,06ms
4	256	128	107,15 ms	40,17 ms	56,62 ms	61,71 ms	128,99 ms	134,09ms
2	256	256	104,02 ms	37,05 ms	53,49 ms	55,46 ms	125,86 ms	127,84ms
4	320	64	113,09 ms	46,11 ms	61,12 ms	73,60 ms	127,17 ms	139,65ms
4	384	64	112,66 ms	45,68 ms	59,66 ms	73,04 ms	121,19 ms	134,57ms
4	384	128	105,75 ms	38,77 ms	52,76 ms	58,92 ms	114,29 ms	120,45ms
3	512	16	110,34 ms	101,15 ms	120,47 ms	202,06 ms	205,48 ms	287,07ms
3	512	32	88,97 ms	58,77 ms	77,70 ms	117,29 ms	160,99 ms	200,59ms
3	512	64	77,85 ms	37,14 ms	55,33 ms	74,04 ms	135,35 ms	154,06ms
3	512	128	71,52 ms	27,61 ms	43,89 ms	50,87 ms	118,05 ms	125,03ms
3	512	256	67,25 ms	22,93 ms	34,78 ms	38,06 ms	99,91 ms	103,19ms
1	512	512	64,62 ms	19,97 ms	26,56 ms	27,87 ms	55,56 ms	56,88ms

Tabulka 5.14: Celkové naměřené výsledky pro soustavu o 512 neznámých

od hodnot z vrchní části sloupce, může docházet k tomu, že ALU jednotky ve vrchní části sloupce dopočítají dříve než bude dokončeno čtení pro ALU jednotky ze spodní části sloupce, bylo zvoleno řešení, že čtení z paměti má vyšší prioritu.

V ideálním případě by tedy měl následovat jeden náhodný přístup do paměti po dokončení čtení, po tomto náhodném přístupu se sekvenčně zapíše celý sloupec.

Toto však nelze realizovat jestliže nemáme dostatek ALU buněk. Jedná se o protichůdné vlastnosti. Jako ideální se tedy jeví situace 1, která je však pro matice větší velikosti nereálná. Je velmi obtížné této situace dosáhnout z důvodů prostorové náročnosti, proto je potřeba vždy najít vhodný kompromis mezi vynaloženými prostředky (počet ALU buněk, rychlost operací, šířka paměťové sběrnice) a výpočetním časem.

5.9 Implementační omezení simulačního modelu

Simulační model je omezen velikostí použitého datového typu *long long*. Teoreticky lze tedy řešit soustavy, kde koeficienty v matici jsou $< 2^{31}$. Při použití matic s většími koeficienty může docházet k přetečení datového typu během násobení.

Kapitola 6

Realizace vizualizace výpočtu

Pro zobrazení jednotlivých kroků výpočtu byla vytvořena jednoduchá vizualizační aplikace v jazyce Java. Aplikace je určena ke sledování prováděných operací v jednotlivých krocích výpočtu. Tyto kroky jsou charakterizovány stejným simulačním časem.

Jednotlivé operace jsou barevně odlišeny. Každá buňka je znázorněna společně se jejími vstupy a výstupy. Paměť je znázorněna jen jako 2D pole buněk.

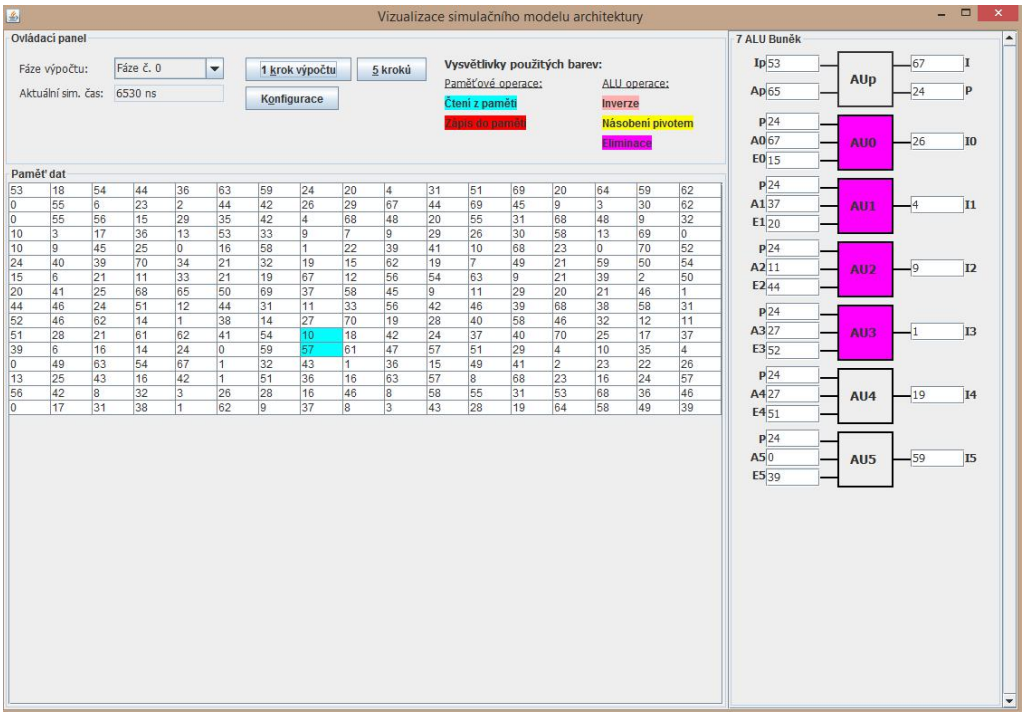
Ve vizualizaci lze procházet po jednotlivých krocích vpřed. Po průchodu každé fáze výpočtu lze přejít na další fázi výpočtu. Obrázek 6.1 zobrazuje náhled na vizualizační aplikaci. Dále je možné zobrazit použitou konfiguraci.

XML soubor je nejdříve zpracován a jsou z něj načteny jednotlivé informace o průběhu výpočtu. Poté je zobrazeno vlastní okno s vizualizací.

Výpočet je možné krokovat buď po jednotlivých krocích nebo po 5 krocích. Dále je možné zvolit fázi (eliminační krok) od které bude výpočet krokován. Po dokončení výpočtu je možné výběrem fáze výpočet krokovat znovu od zvolené fáze nebo spustit celou vizualizaci znovu.

ALU buňky jsou znázorněny schématicky, z důvodu přehlednosti není znázorněno propojení ALU buněk.

Činnost jednotlivých ALU buněk závisí na změnách na vstupních portech. Činnost ALU buněk a paměti je rozlišena barevně. V každém kroku je buď započata nebo dokončena nová ALU nebo paměťová operace, případně u operací, které nemají dobu trvání, je operace v daném kroku provedena okamžitě. Takovou operací je například zneplatnění hodnot na vstupech ALU buněk.



Obrázek 6.1: Náhled na vizualizační aplikaci

6.1 Zpracování výstupního souboru pro vizualizaci

Výstupní XML soubor získaný ze simulačního modelu je nejdříve zpracován do výsledné architektury a jednotlivých fází. Nejdříve jsou zpracovány informace o architektuře, poté je načtena první matice soustavy, dále je načtena konfigurace použitá během simulace a pak jsou zpracovány jednotlivé fáze

6.1.1 Tvorba architektury

Z XML souboru je na základě počtu eliminačních ALU buněk získán počet těchto buněk. Na základě počtu těchto buněk je vytvořeno hlavní okno vizualizace.

Toto okno je složeno ze třech základních panelů, které se dále mohou dělit. Je zde ovládací panel, panel ALU buněk a panel paměti.

6.1.1.1 Ovládací panel

Ovládací panel je složen ze 3 dalších panelů, panelu určenému k volení fáze výpočtu, panelu s kontrolními prvky a panelu s vysvětlivkami barevného odlišení jednotlivých ALU a paměťových operací.

Panel pro výběr fáze zobrazuje a umožňuje vybrat aktuální fázi výpočtu. Zároveň je zde zobrazen aktuální simulační čas.

Na panelu s kontrolními prvky se nachází tlačítka pro posun po 1/5 krocích výpočtu a tlačítko pro zobrazení použité konfigurace. Tato konfigurace se otvírá v novém dialogovém okně.

6.1.1.2 Panel ALU buněk

Na základě počtu ALU buněk je vytvořen panel ALU buněk. Ten je vždy složen z jedné buňky v řádku pivota a několika eliminačních buněk. Každá buňka, která je v činnosti je barevně zvýrazněna podle právě prováděné operace.

Eliminační buňky jsou číslovány od 0 a jsou pojmenovány jako $AU_{0\times n-2}$, kde n je vstupní počet buněk. Buňka v řádku pivota není číslována, je pojmenována jako AU_p. Vstupy/výstupy ALU buněk odpovídají vstupům/výstupům buněk na schématu 5.1.

6.1.1.3 Panel paměti

Panel paměti slouží k zobrazení aktuálních dat z paměti. Tyto data jsou průběžně aktualizována podle pořadí zápisů. Dále jsou barevně odlišeny jednotlivé paměťové operace.

6.1.2 Načtení použité konfigurace

V XML je obsažená během výpočtu použitá konfigurace. Tato konfigurace je během načítání XML souboru načtena a předána ve formě řetězce hlavnímu vizualizačnímu oknu aplikace, které jí případně po kliknutí na tlačítko *Konfigurace* zobrazí.

6.1.3 Tvorba jednotlivých fází

Po načtení architektury, konfigurace a vstupní matice soustavy následuje načtení jednotlivých fází. Každá fáze je složena z matice, nad kterou se v dané fázi pracuje a jednotlivých operací.

Vyskytují se zde operace dvou typů, ALU operace a paměťové operace (v XML jako *MEM* operace). Každá *MEM* operace má typ (čtení/zápis), stav (začátek/konec/okamžitá), čas operace v ns, směr (od/pro řídící buňku), velikost a počet hodnot. Jednotlivé hodnoty nesou informace o adrese a datové hodnotě.

ALU operace mají ID buňky, typ operace (inverze, násobení, eliminace, kopírování vstupu, zneplatnění vstupu, načtení eliminačního členu, načtení hodnoty z řádku pivota, načtení hodnoty vstupu), stav (začátek/konec/okamžitá) a čas operace v ns.

Pokud je čas právě zpracované operace stejný jako čas předchozí operace, je operace zřetězena s předchozí operací, aby bylo možné tyto operace zobrazit současně.

Takto je vytvořen list jednotlivých zpráv, které jsou reprezentovány vnitřně jako textové řetězce. Každá fáze obsahuje takovýto list a vstupní matici.

6.2 Průchod jednotlivými fázemi výpočtu

Každá fáze výpočtu vizualizace obsahuje kromě vstupní matice list zpráv. Tyto zprávy nesou informace o jednotlivých operacích a jsou procházeny pomocí krokování. Podle toho zda se jedná o paměťovou nebo ALU operaci se pracuje buď s tabulkou s daty z paměti nebo s ALU buňkami.

Pokud se jedná o čtení z paměti, pouze se barevně zvýrazní v tabulce dat dané adresy buněk v paměti ze kterých se čte. Pokud se jedná o zápis, na začátku zápisu se adresy buněk v paměti zvýrazní a když je zápis dokončen, aktualizuje se tabulka dat.

Pro ALU buňky je podle typu operace rozhodnuto zda se jedná o výpočet inverze, násobení, eliminace, kopírování vstupu, zneplatnění vstupů nebo načtení vstupu/eliminačního členu/hodnoty datového vstupu. Na základě tohoto typu a ID buňky je rozhodnuto o kterou ALU buňku se jedná.

Na základě stavu a typu operace je daná buňka nastavena jako aktivní/neaktivní a jsou nastaveny dané vstupy/výstupy.

Jakmile se takto projde na konec dané fáze, je uživateli nabídnut přechod na další fázi výpočtu, případně může vybrat, kterou fází výpočtu chce pokračovat.

Závěr

Tato závěrečná kapitola se zabývá vyhodnocení časové a prostorové složitosti navržené architektury na úrovni aritmetických jednotek a paměti, z hlediska aritmetických operací a vyhodnocení zlepšení výpočetního času v závislosti na počtu aritmetických jednotek. Dále je zde celkově zhodnocena práce.

Vyhodnocení časové a prostorové složitosti

Práce se zabývá zkoumáním na vyšší úrovni. Pro podrobnější měření je nutné upravit parametry simulačního modelu, tak aby co nejvíce odpovídali hodnotám pro danou architekturu. Toto práce umožňuje.

Navržená architektura stojí na mírně pozměněné Gaussově eliminační metodě. Navržená architektura je simulována pomocí simulačního modelu ze kterého získáme simulační čas, který odpovídá námi zvoleným parametrům pro vyřešení soustavy rovnic o daném počtu neznámých. Parametry výpočetního modelu jsou časy jednotlivých ALU operací a paměťových operací, dále počet ALU jednotek k dispozici a šířka paměťové sběrnice.

Simulační model nezohledňuje prostorovou náročnost ALU jednotek. Dále předpokládá, že ALU jednotky umějí pouze jednu sdruženou operaci.

Vyhodnocení na úrovni aritmetických jednotek

Ze získaných dat je jasně vidět, že čím více máme k dispozici ALU jednotek, je výpočet rychlejší, pokud nedochází ke zpomalení vlivem úzké paměťové sběrnice.

Další hledisko, které je potřeba zvážit při rozhodování, je že čím více použijeme ALU jednotek, tím větší plochu zaberou dané HW obvody. S tímto

také souvisí, že násobičky u nichž není rychlost závislá na velikosti operandu, jsou řádově složitější a zabírající větší plochu. V práci nebylo toto hledisko posuzováno, protože simulace architektury probíhá na vyšší úrovni, než na úrovni jednotlivých obvodů.

Vyhodnocení aritmetických operací

Navržená architektura je z hlediska počtu ALU operací ideální, nedochází ke zbytečným výpočtům. Počty jednotlivých operací jsou dány podstatou problému. Nejvíce převažuje násobení a odčítání. Inverze je zde zastoupena pouze jednou v každém eliminačním kroku.

Z toho také vyplývá, že nejvíce závisí na rychlosti násobení a odčítání. Pro soustavu o 512 neznámých, s velikostí modulu 2^{21}, vychází zlepšení při využití rychlé násobičky, kdy rychlost není závislá na velikosti operandu je zlepšení až o 130 % oproti využití násobičky s rychlostí závislou na velikosti operandu.

Vyhodnocení na úrovni paměti

Vzhledem k tomu, že navržená architektura umožňuje pouze jednu paměťovou operaci současně, je možné vždy provést jen tolik výpočetních operací jako je paměťových jednotek. Po tomto výpočtu následuje zápis do paměti, který je však nutno realizovat jako náhodné čtení z paměti.

Tuto vlastnost by bylo možné eliminovat oddělením čtení od zápisu do paměti. Paměť by pak měla oddělenou čtecí a zapisovací část. Na druhou stranu by bylo zapotřebí další logiky pro ošetření pořadí paměťových operací, abychom nedocházelo ke čtení ještě nezapsaných hodnot a podobným hazardům.

Z naměřených výsledků jasně vidíme závislost mezi dobou výpočtu a šířkou paměťové sběrnice. Dále vidíme závislost na době potřebné k vykonání paměťové operace. Čím máme širší paměťovou sběrnici, tím více ALU jednotek jsme schopni zásobovat daty. Architektura není až tak závislá na době sekvenčního přístupu do paměti, více je závislá na době náhodného přístupu do paměti. Dochází tedy k častějšímu náhodnému přístupu do paměti.

Navržená zlepšení v pořadí operací během výpočtu najdou uplatnění především v případě širších paměťových sběrnice a více ALU buněk, než u menšího počtu ALU buněk a užších paměťových sběrnic, kde dochází k serializaci výpočtu.

Vyhodnocení zlepšení výpočetního času v závislosti na počtu ALU jednotek a šířce sběrnice

Jak již bylo řečeno dříve, největší zlepšení poskytuje rychlejší násobení. Pokud je k dispozici širší paměťová sběrnice, je vhodné aby náhodný přístup do paměti byl co nejrychlejší. Bude-li k dispozici jen úzká paměťová sběrnice není možné dostatečně rychle zásobit ALU jednotky potřebnými daty. Počet ALU buněk je pak potřeba volit s ohledem na jejich prostorovou náročnost. Jako ideální poměr se jeví 4 ALU jednotky na 1 jednotku šířky paměťové sběrnice. Při tomto poměru nedochází k takovému zpomalení, například vlivem pomalého sekvenčního přístupu do paměti.

Při konfiguraci v tomto poměru (a s rychlou násobičkou) z naměřených výsledků (soustava o 512 neznámých) vyplývá, že ubráním poloviny ALU jednotek, se výpočet zpomalí průměrně o 39%. Naopak přidáním dvojnásobku ALU jednotek se výpočet průměrně zrychlí o 14%.

Na druhou stranu, stejně jako počet ALU jednotek, má i šířka paměťové sběrnice vliv na velikost obvodu. Čím širší bude paměťová sběrnice, tím bude zabírat její vedení větší plochu obvodu.

Zhodnocení práce

Účelem práce byla analýza stávajících hardwarových architektur použitelných pro řešení soustav lineárních rovnic v modulární aritmetice. Dalším cílem byl návrh takovéto architektury, její simulace a zhodnocení návrhu této architektury. Posledním cílem byla realizace vizualizace jednotlivých výpočetních kroků této architektury.

Práce se nejdříve zabývala analýzou a definicí matematických struktur. Dále analyzovala postupy pro nalezení řešení soustav lineárních rovnic. V další části analyzovala stávající hardwarové architektury pro řešení těchto soustav. Práce dále uvádí možnosti využití problému řešení soustav v různých oblastí a zaměřuje se na využití těchto soustav v oblasti počítačové bezpečnosti.

Práce se dále zabývá různými hledisky návrhu architektury, analyzuje možnosti organizace matematických operací. Na základě informací získaných z analýzy pak navrhuje jedno z možných řešení architektury. Na základě posloupnosti operací pak navrhuje jednotlivé stavební buňky architektury, jednotlivé moduly. Navržená architektura je zhodnocena.

V realizační části pak realizuje simulační model navržené architektury dodržující popsané chování. Simulační model je pak hodnocen z hlediska simulačního času nutného pro nalezení řešení soustavy lineárních rovnic v mo-

dulární aritmetice. Další částí realizace bylo vytvoření vizualizační aplikace, která na základě výstupního XML vytvořeného simulačním modelem, bude vizualizovat jednotlivé kroky výpočtu. Kromě samotného simulačního modelu a vizualizační aplikace bylo vytvořeno několik dalších nástrojů pro práci s výstupy ze simulačního modelu. Tyto nástroje slouží pro zpracování a ověření výsledků ze simulačního modelu. Dále byl vytvořen generátor dat pro generování vstupních dat pro simulační model.

Možným pokračováním by mohl být návrh architektury orientované pro výpočet po řádcích a jejich srovnání. Z hlediska simulačního modelu by bylo možné navázat simulací paměti s oddělenými sběrnicemi pro čtení a zápis. Další možností je bližší prozkoumání stávajícího simulačního modelu.

Samotná simulace je provedena na relativně vysoké úrovni hardwarové abstrakce (například použitím sdružené paměťové sběrnice). Největším problémem se jeví datová závislost výpočtu ve sloupci na hodnotě z řádku pivota. Samotná architektura předpokládá, že při větších velikostech modulu je nízká pravděpodobnost, že pivot nebude nalezen na hlavní diagonále. Proto si architektura může dovolit relativně drahé operace přesunu dat v paměti. Pokud by se jednalo o menší velikosti modulu, bylo by vhodné aby architektura dokázala již během přesunu dat tyto data využít k předvýpočtu hodnot, tak aby se eliminovala tato datová závislost. Toto však není realizováno, protože pro měření byl použit dostatečně velký modul, takže pravděpodobnost nutnosti přesunu řádků s pivotem je minimální.

Další možností navázání práce by mohla být simulace výpočtu v tělese $\mathbb{GF}(2^k)$. Tím by se jednoduše dalo odstranit implementační omezení plynoucí z použití datových typů určených pro celá čísla, do kterých jsou během výpočtu ukládány hodnoty.

Pokračováním vizualizace architektury a jejich výpočetních kroků by bylo rozšíření o vizualizaci spojenou s časovou osou. Samotná vizualizace není sice omezená velikostí matice, ale pro větší matice již není natolik přehledná. Pro vizualizaci velkých matic by například nebylo potřeba zobrazovat vždy celou datovou paměti, ale jen část dat se kterými se aktuálně pracuje. Výstupní XML soubor však pro vizualizace velkých matic poskytuje dostatek informací, tak aby bylo možné tuto vizualizaci realizovat.

Literatura

[1] Althoen, S. C.; McLaughlin, R.: Gauss-Jordan Reduction: A Brief History. *Am. Math. Monthly*, ročník 94, č. 2, Únor 1987: s. 130–142, ISSN 0002-9890.

[2] Bogdanov, A.; Kizhvatov, I.; Pyshkin, A.: Algebraic Methods in Side-Channel Collision Attacks and Practical Collision Detection. In *Progress in Cryptology - INDOCRYPT 2008*, *Lecture Notes in Computer Science*, ročník 5365, Springer Berlin Heidelberg, 2008, ISBN 978-3-540-89753-8, s. 251–265.

[3] Bogdanov, A.; Knudsen, L.; Leander, G.; aj.: PRESENT: An Ultra-Lightweight Block Cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007*, *Lecture Notes in Computer Science*, ročník 4727, editace P. Paillier; I. Verbauwhede, Springer Berlin Heidelberg, 2007, ISBN 978-3-540-74734-5, s. 450–466.

[4] Buček, J.; Kubalík, P.; Lórencz, R.; aj.: Comparison of FPGA and ASIC Implementation of a Linear Congruence Solver. In *Digital System Design (DSD), 2013 Euromicro Conference on*, Sept 2013, s. 284–287.

[5] Demlová, M.: Diskrétní matematika a logika. 2005, doprovodný text k přednáškám.

[6] Gollová, A.: Teorie informace a kódování. 2013, doprovodný text k přednáškám.

[7] Hladík, J.; Lórencz, R.; Šimeček, I.: Clock Math-a System for Solving SLEs Exactly. *Acta Polytechnica*, ročník 53, č. 2, 2013, ISSN 1805-2363.

[8] Initiative, A. S.: SystemC. 2014. Dostupné z: http://www.accellera.org/downloads/standards/systemc

[9] Jahn, J.: Simulace řešiče soustav lineárních kongruencí. 2012, diplomová práce. Dostupné z: https://dip.felk.cvut.cz/browse/pdfcache/jahnjiri_2012dipl.pdf

[10] Jones, J.: [Online; accessed 13-July-2014]. Dostupné z: https://people.richland.edu/james/lecture/m116/matrices/pivot.html

[11] Kalvoda, T.; Petr, I.: Matematika pro kryptologii. 2013.

[12] Krejčí, J.: Vybrané partie z matematiky. 08 2006. Dostupné z: http://physics.ujep.cz/~jkrejci

[13] Kužel, R.; Míková, M.: Soustavy lineárních rovnic. 2005, doprovodný text k přednáškám. Dostupné z: http://dimatia.fav.zcu.cz/2005/vyuka/zm1/soustavy.pdf

[14] Lórencz, R.; Morháč, M.: A modular system for solving linear equations exactly. *Computers and Artificial Intelligence*, ročník 1992, č. 11, 1991. Dostupné z: https://users.fit.cvut.cz/~lorencz/clanky/8_Modular_System_1_1992.PDF

[15] Microsoft: Download Visual C++ Redistributable Packages for Visual Studio 2013 from Official Microsoft Download Center. 2014, [Online; accessed 4-May-2014]. Dostupné z: http://www.microsoft.com/en-us/download/details.aspx?id=40784

[16] Nover, H.: Algebraic Cryptanalysis of AES: An Overview. 2004.

[17] Olšák, P.: *Lineární algebra*. ČVUT Fakulta elektrotechnická, 2010.

[18] Renauld, M.; Standaert, F.-X.; Veyrat-Charvillon, N.: Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, *Lecture Notes in Computer Science*, ročník 5747, Springer, 2009, s. 97–111.

[19] Rothe, J.: *Complexity Theory and Cryptology: An Introduction to Cryptocomplexity*. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2006, ISBN 9783540285205. Dostupné z: http://books.google.cz/books?id=YnLmsHAtvYEC

[20] Rupp, A.; Eisenbarth, T.; Bogdanov, A.; aj.: Hardware SLE solvers: Efficient Building Blocks for Cryptographic and Cryptanalytic Applications. *Integration, the VLSI Journal*, 2011, ISSN 0167-9260, hardware Architectures for Algebra, Cryptology and Number Theory. Dostupné z: `http://www.sciencedirect.com/science/article/pii/S016792601000057X`

[21] Sadílek, M.: Paralelní řešič soustav lineárních rovnic v aritmetice kódů zbytkových tříd. 2011, bakalářská práce. Dostupné z: `https://dspace.cvut.cz/handle/10467/7705`

[22] Sen, S.; Agarwal, H.; Sen, S.: Chemical equation balancing: An integer programming approach. *Mathematical and Computer Modelling*, ročník 44, č. 7–8, 2006: s. 678 – 691, ISSN 0895-7177.

[23] Starosta, Š.: Matematika pro informatiku. 2012/2013, prezentace k přednáškám.

Příloha A

Přílohy

A.1 Manuál pro simulaci v SystemC

A.1.1 Spuštění simulace výpočtu

Simulační aplikace je na CD přiložena jako staticky nalinkovaná, aby jí bylo možné spustit. Protože se jedná o aplikaci vytvořenou v aplikaci MS Visual Studio 2013 vyžaduje ke svému běhu Visual C++ Redistributable Packages for Visual Studio 2013 dostupné z [15].

Prvním vstupním parametrem aplikace je cesta k souborem se vstupní maticí a druhým vstupním parametrem je cesta k souboru s konfigurací.

Vstupem simulace výpočtu je datový soubor s daty matice a soubor s konfigurací, struktura těchto souborů je popsána níže. Výstup je vždy na standardní výstup, který však lze přesměrovat do souboru. Výstup je buď XML soubor pro vizualizaci, nebo jen záznam o výpočtu, kdy se zaznamenává dokončení fází výpočtu a nalezené řešení společně s determinantem a simulačním časem.

Na CD, v adresáři *src*, je umístěn projekt z aplikace MS Visual Studio 2013 použitý při kompilaci. Pro novou kompilaci projektu je nutné nastavit správně adresáře s cestami ke knihovnám, případně knihovny zkompilovat znova.

Příkaz ke spuštění simulace s parametry pak může vypadat například takto:

SystemCColumnSLESolver.exe d_512_1.txt konfigurace_s4_160_32.txt

A.1.2 Struktura souboru s daty

Data jsou uložena v souboru, soustava o n neznámých je uložena v rozšířeném maticovém tvaru v matici o velikost n řádků a $n + 1$ sloupců. Sloupce matice jsou odděleny tabulátory. Pro jednoduché vygenerování souboru s daty je na CD v adresáři *tools* přiložen notebook pro Wolfram Mathematicu *generovani_dat.nb*, který do adresáře vygeneruje jak vygenerovaná data, tak odpovídající řešení. Toto řešení lze použít pro kontrolu správnosti nalezeného řešení. Pokud však dochází během výpočtu k záměně řádků v důsledku toho, že pivot nebyl nalezen na hlavní diagonále, tak vygenerovaný determinant tuto rotaci nezohledňuje. Jedná se o determinant původní matice s daty.

A.1.3 Ověření řešení a zpracování výstupních souborů

Pro ověření správnosti nalezeného řešení ze simulace architektury je na CD v adresáři skript v jazyce Perl *porovnej_s_vysledky.pl*. Jak bylo uvedeno výše, tak pokud při výpočtu došlo k záměně řádků, vypočtený determinant generátorem tuto záměnu nezohledňuje. Tento skript porovnává vypočtené řešení s řešením z generátoru dat. Vstupními argumenty skriptu je výstup z generátoru a zjednodušený výstup ze simulace architektury. Skript do soubory *casy.txt* přidá data potřebná pro další analýzu z daného výstupního souboru ze simulace. Je potřeba zachovat pořadí argumentů skriptu, tak aby první argument byl soubor s výsledky z generátoru a jako druhý argument byl výstup z architektury.

Pokud by ve výstupním souboru *casy.txt* zůstala nějaká číselná hodnota z řešení, znamenalo by to že architektura nenalezla řešení soustavy správně.

Skript *zpracuj_vysledky.pl* pak soubor *casy.txt* převede do formy dat pro další analýzu například v tabulkovém procesor.

Kompletní zpracování výstupů do zpracovatelných výsledků pomocí skriptů pak může vypadat například takto:

```
porovnej_s_vysledky.pl vysledky512_1.txt vystup_d1_512_512.txt
porovnej_s_vysledky.pl vysledky512_1.txt vystup_d1_512_256.txt
zpracuj_vysledky.pl casy.txt
```

Po dokončení průchodu všemi soubory je ve složce soubor *casy.txt* a *vystup.txt*.

A.1.4 Struktura konfiguračního souboru

Konfigurační soubor má pevnou strukturu, tak jak je uvedeno níže. Tuto strukturu je nutné dodržet. V konfiguračním souboru jsou kromě parametrů uvedených v tabulce 5.1, ještě počet řádků soustavy v maticové formě, velikost modulu, perioda hodin (využita pro výpočet doby jednotlivých operací) a zda požadujeme zjednodušený výpis nebo kompletní výpis pro vizualizaci.

```
velikostMatice=8
modul=71
velikostOperandu=7
casNasobeniA=0
casNasobeniB=1
casNasobeniC=0
casInverzeA=0
casInverzeB=2.2
casInverzeC=0
casOdecteniA=0
casOdecteniB=0
casOdecteniC=1
pocetALU=8
sirkaPametoveSbernice=8
casCteniSekv=1
casCteniNahodna=5
casZapisSekv=1
casZapisNahodna=5
periodaHodin=10
vizualizace=ne
```

A.2 Manuál pro vizualizaci

Vizualizační aplikace je aplikace vytvořená v jazyce Java. Vstupem aplikace je vizualizační XML soubor vzniklý během simulace. XML soubor je nejdříve zpracován a jsou z něj načteny jednotlivé informace o průběhu výpočtu. Poté je zobrazeno vlastní okno s vizualizací.

Výpočet je možné krokovat buď po jednotlivých krocích nebo po 5 krocích. Dále je možné zvolit fázi od které bude výpočet krokován. Po dokončení výpočtu je možné výběrem fáze výpočet krokovat znovu od zvolené fáze nebo spustit celou vizualizaci znovu.

Činnost jednotlivých modulů závisí na změnách na vstupních portech modulů. Činnosti paměti a jednotlivých modulů jsou barevně odlišeny, vysvětlivky k jednotlivým barvám jsou zobrazené přímo v aplikaci.

Jednotlivé operace jsou barevně odlišeny. Každá buňka je znázorněna společně s jejími vstupy a výstupy. Paměť je znázorněna jen jako 2D pole

buněk.

Výpočet je možné krokovat buď po jednotlivých krocích nebo po 5 krocích. Dále je možné zvolit fázi (eliminační krok), od které bude výpočet krokován. Po dokončení výpočtu je možné výběrem fáze výpočet krokovat znovu od zvolené fáze nebo spustit celou vizualizaci znovu.

Řídící buňka není znázorněna, je však připojena pomocí signálů k výpočetním buňkám a paměti, tak jak je znázorněno na schématu 5.1.

Činnost jednotlivých ALU buněk závisí na změnách vstupních portů. Činnost ALU buněk a paměti je rozlišena barevně.

A.2.0.1 Spuštění aplikace

Aplikace je zkompilována v Java JDK 1.8.0_40. Na CD je přiložena vizualizace jak formou zkompilované aplikace, tak ve formě projektu pro NetBeans IDE 8.0.2.

Spuštění se provádí předáním názvu *xml* souboru s vizualizací jako 1. argumentu příkazové řádky.

Spuštění může vypadat například takto:
java -jar JavaVizualizace.jar vystup_8_1.xml

A.2.0.2 Chod aplikace

Po výběru fáze v pravé části aplikaci lze vizualizaci krokovat buď po 1 kroku nebo po 5 krocích. Každý krok je ohraničen změnou simulačního času. Ten je zobrazen pod

Jednotlivé ALU buňky jsou znázorněny symbolicky, bez přímých propojení. Hodnoty právě provedeného kroku jsou zobrazeny na daných portech. Po provedení posledního výpočtu je v posledním sloupci matice nalezené řešení soustavy.

Výpis matice je realizován tabulkou, ve které jsou aktuální hodnoty matice a je v ní zvýrazňováno do které buňky matice se zapisuje nebo ze které se čte.

A.3 Použité nástroje a jejich verze

Použité knihovny jsou umístěny v přiloženém adresáři v *lib*.

MS Visual Studio 2013 Microsoft Visual Studio Professional 2013, Verze 12.0.31101.00 Update 4

Visual C++ 2013 Microsoft Visual C++ 2013

SystemC 2.3.1 SystemC 2.3.1-Accellera — Oct 16 2014 23:11:07 [8]

Java SE 8u40 JDK 1.8.0_40

NetBeans NetBeans IDE 8.0.2

A.4 Seznam použitých zkratek

float Pohyblivá řádová čárka

MQ Multivariate Quadratic

XML Extensible markup language

ALU Aritmeticko-logické operace

GB Gigabyte

Příloha B

Obsah přiloženého CD

- `readme.txt` ... stručný popis obsahu CD
- `data` ... Testovací data
  - `simulace` ... Testovací datové soubory pro simulaci rozdělená dle modulů
  - `vizualizace` .. Vstupní XML soubory pro vizualizaci, konfigurace a data použitá pro generování
- `exe` ... Adresář se spustitelnou formou implementace
  - `simulace` ... Spustitelná implementace simulace
  - `vizualizace` ... Spustitelná forma vizualizace
- `lib` ... Použité knihovny
- `src`
  - `impl` ... Zdrojové kódy implementace
    - `simulace` ... Zdrojové kódy simulace
    - `vizualizace` ... Zdrojové kódy vizualizace
  - `thesis` ... Zdrojová forma práce ve formátu LaTeX
- `text` ... Text práce
  - `thesis.pdf` ... Text práce ve formátu PDF
- `tools`
  - `generovani_dat.nb` Wolfram Mathematica notebook pro generování dat a výsledků
  - `reseni.nb` .. Wolfram Mathematica Notebook pro ověření výsledků
  - `porovnej_s_vysledky.pl` ... Perl skript pro porovnání souboru s řešením s výstupem ze simulace architektury
  - `zpracuj_vysledky.pl` .. Perl skript pro zpracování souboru *casy.txt* do datového souboru
- `results` ... Kompletní naměřené výsledky