Vysoké učení technické v Brně
Fakulta informačních technologií

Montgomeryho násobení

Ročníkový projekt

Tomáš Pirkl
2006

Prohlášení

Prohlašuji, že jsem tento ročníkový projekt vypracoval samostatně pod vedením doc. Ing. Vladimíra Drábka, CSc.

Veškeré literární prameny, ze kterých jsem čerpal jsem uvedl.

Poděkování

Poděkovat bych chtěl vedoucímu projektu doc. Ing. Vladimírovi Drábkovi, CSc., za poskytnutí veškerých materiálů a při pomoci řešení nejrůznějších problémů.

Tomáš Pirkl

Abstrakt

V této práci je popsáno několik metod Montgomeryho násobení. Montgomeryho násobení je základ pro modulární umocňování, které se hojně využívá v asymetrické kryptografii pro šifrování a podepisování digitálních dat.Všechny prezentované metody byly navrženy už dříve. Jsou zde popsány a detailně analyzovány prostorové a časové nároky všech pěti metod. Jeden z algoritmů byl implementován v jazyku C. Analýza a aktuální výkonové výsledky jsou předvedeny na metodě Coarsely Integrated Operand Scanning (CIOS).

Klíčová slova: modulární aritmetika, Montgomeryho metoda, modulární násobení a umocňování, asymetrická kryptografie, šifrovací algoritmus RSA, šifrovací algoritmus Diffie Hellman, šifrovací algoritmus založený na eliptických křivkách EC-DSA, elektronický podpis DSA.

Obsah

Abstrakt ..... - 3 -
Obsah ..... - 4 -
1 Úvod ..... - 5 -
1.1 Motivace ..... - 5 -
1.2 Cílem této práce je ..... - 5 -
1.3 Struktura ..... - 5 -
2 Modulární aritmetika ..... - 6 -
2.1 Klasické modulární násobení ..... - 7 -
2.2 Montgomeryho násobení ..... - 7 -
3 Algoritmy Montgomeryho násobení ..... - 10 -
3.1 Separated Operand Scanning (SOS) ..... - 12 -
3.2 Coarsely Integrated Operand Scanning (CIOS) ..... - 15 -
3.3 Finely Integrated Operand Scanning (FIOS) ..... - 16 -
3.4 Finely Integrated Product Scanning (FIPS) ..... - 18 -
3.5 Coarsely Integrated Hybrid Scanning (CIHS) ..... - 20 -
4 Aplikační oblasti Montgomeryho násobení ..... - 22 -
4.1 Diffie-Hellman DH ..... - 23 -
4.2 RSA ..... - 24 -
4.3 Digital Signature Algorithm (DSA) ..... - 25 -
4.4 Šifry založené na eliptických křivkách ECDSA ..... - 26 -
5 Implementace algoritmu ..... - 27 -
6 Závěr ..... - 30 -
7 Literatura ..... - 31 -
8 Přílohy ..... - 32 -

1 Úvod

1.1 Motivace

Důvodem pro studium rychlých a paměťově nenáročných algoritmů pro modulární násobení byl příchod jeho aplikací v asymetrické kryptografii. Kryptografický algoritmus RSA a Diffie Hellman využívají modulární umocňování, které je rozkládáno do sérií modulárního násobení aplikací binární nebo m-ární metody. Algoritmus Montgomeryho násobení je používán pro urychlení modulárního násobení, které je potřeba během procesu umocňování. Pomocí Montgomeryho algoritmu dostaneme

$$\mathrm{MonPro}(a,b) = a \cdot b \cdot r^{-1} (\mathrm{mod}\, n), \tag{1}$$

kde $a,b < n$ a r takové, že největší společný dělitel $gcd(n,r) = 1$. I když algoritmus pracuje se všemi r, která jsou nesoudělná s n, je lepší když r je mocninou čísla 2. V takovém případě, Montgomeryho algoritmus provádí dělení mocninou čísla 2, které je na běžných počítačích podstatně rychlejší, např. signálové procesory a mikroprocesory; toto vede k jednodušší a také rychlejší implementaci oproti obyčejnému modulárnímu násobení.

1.2 Cílem této práce je

V této práci se budu zabývat operacemi, které jsou spojeny s výpočtem Montgomeryho součinu, tedy rychlými a paměťově nenáročnými algoritmy pro výpočet MonPro(a, b), a analýzou jejich časových a paměťových požadavků. Bude uvedeno a srovnáno pět alternativ pro Montgomeryho násobení.

1.3 Struktura

Následující kapitola vysvětluje modulární aritmetiku a následně popisuje Montgomeryho násobení. Ve třetí kapitole je uvedeno 5 typů algoritmu pro Montgomeryho násobení. Čtvrtá kapitola obsahuje jeho aplikační oblasti. Pátá kapitola se věnuje implementaci algoritmu pro Montgomeryho násobení. Šestá kapitola je věnována závěru, kde jsou shrnuty veškeré dosažené výsledky.

2 Modulární aritmetika

Modulární aritmetika je upravená aritmetika nad celými čísly, kterou na přelomu devatenáctého století zformuloval K.F.Gauss (1801). V základní podobě se často označuje názvem aritmetika hodin. Výsledky operací v modulární aritmetice jsou prvky konečné množiny celých čísel, pro kterou se také používá označení *okruh*. Omezující hodnota okruhu se nazývá *modulus*. Obvykle se jako základní příklad uvádí hodinový ciferník rozdělený na 12 dílů, který však popisuje 24 hodinový (denní) cyklus. Po překročení poledne se obvykle vracíme k označení času 1 hodina, 2 hodiny, 3 hodiny atd. Modulem v tomto případě je číslo 12 a odpolední hodiny podvědomě určujeme jako zbytek celočíselného dělení.

Modulární aritmetika definuje tzv. *kongruentní třídy* celých čísel. Dvě čísla *a*,*b* jsou *modulo N* kongruentní, jestliže existuje takové *q*, že platí $(a-b)=q*N$. Jinak lze tento vztah napsat výrazem

$$a \equiv b(\bmod N).$$

Příkladem kongruentní třídy jsou čísla 2, 14, 26, 38, 50, ... Libovolná dvě čísla splňují požadavek definice z předchozí věty pro *N* = 12. Jsou tedy kongruentní modulo 12, resp. říkáme, že čísla náleží do kongruentní třídy *modulo* 12, např. $(26-2)=2*12$ nebo $(50-2)=4*12$. Kongruenci lze také vyjádřit pomocí zbytku po dělení modulem. Všechna čísla z uvedeného příkladu mají tu vlastnost, že zbytek po dělení 12 je číslo 2.

Jedna z nejdůležitějších vět modulární aritmetiky je věta označovaná jako *Fermatova věta*. Jsou-li *a, p* nesoudělná celá čísla, pak platí

$$a^{p-1} \equiv 1(\bmod p).$$

Je-li *p* prvočíslo, pak existuje právě $p-1$ nesoudělných čísel s *p*. Tuto hodnotu vyadřuje *Eulerova funkce* $\varphi(p) \equiv p-1$. Zobecnění Fermatovy věty se označuje jako *Euler-Fermatova* věta, kterou vyjadřuje následující výraz

$$a^{f(n)} \equiv 1(\bmod n).$$

Modulární aritmetika poskytuje tzv. *jednosměrné funkce*, tj. takové funkce, ke kterým nelze vytvořit funkci inverzní. Právě tyto jednosměrné funkce se využívají při vytváření kryptografických algoritmů. Obecně $a^{-1} \equiv x(\bmod n)$ má jednoznačné řešení když *a* a *n* jsou nesoudělná, pokud nejsou, pak řešení neexistuje. Pokud je tedy *n* prvočíslo, pak každé číslo mezi 1 a *n* – 1 je s *n* nesoudělné a exituje právě jedna inverze *modulo n*. Základní operace v modulo aritmetice jsou *komutativní*, *asociativní, distributivní* a jsou to:

$$\begin{aligned}
&((a \bmod n) + (b \bmod n)) \bmod n = (a + b) \bmod n \\
&((a \bmod n) * (b \bmod n)) \bmod n = (a * b) \bmod n \\
&((a \bmod n)^b) \bmod n = a^b \bmod n \\
&(a^b)^c \bmod n = a^{b*c \bmod \varphi(n)} \bmod n
\end{aligned}.$$

2.1 Klasické modulární násobení

Klasická metoda pro modulární násobení $a \cdot b \bmod n$ je založena na provedení násobení $(a \cdot b)$ a následného odečítání dělitele *n* do té doby, než je rozdíl menší než dělitel. Rozdíl se vrací jako výsledek.

Další metodou může být přístup, kdy je provedeno násobení, a pak je součin $(a \cdot b) = c$ klasicky vydělen $c / n = d$. Potom dostaneme výsledek modulárního násobení jako $c - d \cdot n$. Oba přístupy jsou neefektivní a trpí nedostatkem rychlosti.

2.2 Montgomeryho násobení

Základní myšlenka Montgomeryho algoritmu pro zlepšení klasické metody spočívá v redukování bitové délky mezivýsledku na fixní délku *n*+1 bitů. Toho se dosáhne celočíselným dělením dvěma beze zbytku (pro dvojkovou soustavu), které je vloženo do výpočtu a které snižuje bitovou délku právě o jeden bit, jak je vidět na následujícím kódu.

```
Vstupy: a, b, n kde, 0 ≤ a, b < n, n je liché
m: počet bitů X
xi: i-tý bit X
```

```
p0: nejnižší bit P

P := 0;
for i := 0 to m-1 do
  P := P + xi * Y;
  P := P + p0 * n;
  P := P div 2;
if P > n then P := P - n;
```

Montgomeryho algoritmus je založen na tom, že přičítání násobku *n* k mezivýsledku neovlivní hodnotu hledaného výsledku, protože ten je počítán jako *modulo n*, kde *n* je liché číslo. Takže po každém přičtení ve vnitřním cyklu výpočtu je mezivýsledek kontrolován na lichost, a to kontrolou nejnižšího významového bitu. Je-li 1, pak přičteme *n* aby byl sudý. Sudé číslo můžeme dělit dvěma beze zbytku. Toto dělení dvěma redukuje velikost mezivýsledku na *m+1* bitů. Po *m* krocích toto dělení dá stejný výsledek jako jedno dělení číslem 2^m. Toto Montgomeryho násobení vyžaduje dva průchody násobícím procesem, takže čas výpočtu je také dvojnásobný. V prvním průchodu je spočten tzv. Montgomeryho součin

$$c = \left(a * b * \left(2^m\right)^{-1}\right) \bmod n$$

a v druhém průchodu pak hodnota

$$c = \left(c * 2^{2m} * \left(2^m\right)^{-1}\right) \bmod n = \left(a * b\right) \bmod n ,$$

která je hledaným výsledkem. Konkrétně tuto metodu je velmi snadné implementovat, protože pracuje jen s nejnižším významovým bitem a nevyžaduje žádné jiné porovnávání.

Nechť je modul *n* *k*-bitové celé číslo, to je $2^{k-1} \le n < 2^k$, a nechť $r = 2^k$. Algoritmus Montgomeryho násobení vyžaduje aby *r* a *n* byla nesoudělná, tedy aby platilo $\gcd(r, n) = \gcd(2^k, n) = 1$. Tento požadavek je splněn tak, že *n* je liché. Aby bylo možné popsat algoritmus Montgomeryho násobení, definujeme *n-residuum* jako celé číslo pro které platí $a < n$, kde $\overline{a} = a \cdot r (\bmod r)$. To přímo ukazuje, že množina

$$\{a \cdot r \bmod n \mid 0 \le a \le n-1\}$$

je kompletní *zbytkový systém*, to zn., že obsahuje všechna celá čísla v rozsahu 0 až *n*-1 včetně. Tedy, existuje jedna ku jedné zobrazení mezi čísly v rozsahu *0* až *n*-1 a čísly z množiny uvedené nahoře. Montgomeryho redukční algoritmus využívá této vlastnosti prováděním mnohem rychlejší násobící rutiny, která počítá zbytek po vydělení součinu dvou celých čísel číslem *n*. Mějme dané dvě *n-residua* $\overline{a}$ a $\overline{b}$, pak je Montgomeryho součin definován jako

$$\overline{c} = \overline{a} \cdot \overline{b} \cdot r^{-1} (\bmod n) \qquad \textbf{(2)}$$

kde r^{-1} je inverzní hodnota pro *r* modulo *n*, to zn. že je to číslo, pro které platí $r^{-1} \cdot r = 1(\bmod n)$. Výsledné číslo $\overline{c}$ ve vzorci (2) je opravdu n-residuum součinu $c = a \cdot b(\bmod n)$, protože

$$\begin{aligned} \overline{c} &= \overline{a}.\overline{b}.r^{-1}(\bmod n) \\ &= a.r.b.r.r^{-1}(\bmod n) \\ &= c.r(\bmod n). \end{aligned}$$

Aby mohl být popsán Montgomeryho redukční algoritmus potřebujeme další veličinu *n'*, což je celé číslo pro které platí $r \cdot r^{-1} - n \cdot n' = 1$. Celá čísla r^{-1} a *n'* mohou být obě spočtena rozšířeným Euclidovským algoritmem uvedeným v [1]. Výpočtu $\mathrm{MonPro}(\overline{a}, \overline{b})$ je dosaženo následovně:

```
function MonPro(ā , b̄ )
1.    t := ā . b̄
2.    u := (t + (t . n' mod r) . n) / r
3.    if u ≥ n then return u - n else return u
```

Násobení *modulo r* a dělení číslem *r* jsou podstatně rychlejší, když je *r* mocninou čísla 2. Pak je algoritmus pro výpočet Montgomeryho součinu rychlejší a jednodušší, než běžný výpočet pro $a \cdot b \bmod n$, který zahrnuje dělení číslem *n*. Nicméně konverze z normálního residua na *n-residuum*, výpočet *n'*, a konverze zpět na normální zbytek jsou časově náročné. Není dobrý nápad použít algoritmus Montgomeryho násobení, když by mohlo být užito jednoduché modulární násobení. Montgomeryho metoda je vhodná, když se provádí několik modulárních násobení s tím samým modulem. Jako v případě modulárního umocňování. Použitím binární metody pro výpočet mocnin, se nahradí operace umocňování sériemi druhých mocnin a násobením operací *modulo n*.

Následující algoritmus pro umocňování je cesta, jak vypočítat $x := a^e \bmod n$ se složitostí $\mathrm{O}(j)$. Nechť j je počet bitů exponentu e. Je to algoritmus Montgomeryho umocňování. 4. krok algoritmu modulárního umocňování vypočte x užitím $\bar{x}$ a vlastnosti Montgomeryho algoritmu: $\mathrm{MonPro}(\bar{x},1) = \bar{x} \cdot 1 \cdot r^{-1} = x \cdot r \cdot r^{-1} = x \bmod n$.

```
function ModExp(a,e,n)
1.  ā  := a . r mod n
2.  x̄  := 1 . r mod n
3.  for i = j - 1 downto 0
       x̄  := MonPro(x̄ , x̄ )
       if eᵢ = 1 then x̄ := MonPro(x̄ , ā )
4.  return x := MonPro(x̄ , 1)
```

V typických implementacích, operací s velkými čísly jsou vykonávány pomocí rozkladu čísel na slova (16b, 32b). Když w je velikost slova počítače, pak si můžeme představit, že velké číslo je složeno ze sekvence celých čísel, kde každé je řádu $W = 2^w$. Když je takhle požadováno s slov každé řádu W, pak vezmeme r jako $r = 2^{sw}$.

3 Algoritmy Montgomeryho násobení

V této kapitole bude uvedeno několik algoritmů pro provádění Montgomeryho násobení $\mathrm{MonPro(a, b)}$ a analýza jejich časové a paměťové složitosti. Časová analýza je provedena výpočtem celkového počtu násobení, sčítání (odčítání), čtení z paměti a zápisů do paměti v závislosti na velikosti vstupního parametru s. Např. následující operace

```
(C,S) := t[i+j] + a[j] . b[i] + c
```

je výpočet vyžadující tři operace čtení z paměti, dvě přičítání a jedno násobení, protože většina mikroprocesorů násobí dvě jedno-slovní čísla tak, že na výstupu je jeden dvou-slovový výsledek v jednom nebo dvou registrech. Chci poznamenat, že některé typy procesorů mohou do přičítání zapojit dvě instrukce, a to pokud je hodnota $a[j] \cdot b[i]$ double. Toto se bude v následujících závěrech ignorovat.

Velká celá čísla jsou upravena tak, aby byla v paměti během výpočtu. Protože operace přiřazení provedená bez rutiny odpovídá operacím čtení a psaní mezi registry a pamětí. Tyto operace jsou určující pro výpočet poměru času pro přístup do paměti a celkového času běhu algoritmu Montgomeryho násobení. V analýze nejsou brány v úvahu ustavení cyklů a výpočty indexů. Předpokládejme že jediné registry, které jsou k dispozici jsou ty k přechování přenosu C a součtu S (nebo stejně pro výpůjčku a rozdíl). V mnoha mikroprocesorech bude víc registrů, ale pro obecné srovnání přístupů toto dostačuje. Implementace na určitém procesoru by umožnila srovnání ve více detailech.

Analýza paměťové náročnosti je provedena spočtením celkového počtu použitých slov. Nicméně prostor požadovaný pro vstupní a výstupní hodnoty a, b, n, n'_0 a u není brán v úvahu.

Funkce ***ADD(x[i],C)***, která implementuje operaci ***x[i]:=x[i]+C*** zahrnující propagaci přenosu, požaduje jedno čtení z paměti ***(x[i])***, jedno sčítání ***(x[i]+C)*** a jeden zápis do paměti ***(x[i]:=)*** během prvního kroku. Vzhledem k tomu, že na propagaci přenosu z tohoto sčítání je potřeba v průměru jedno čtení z paměti, jedno sčítání a jeden zápis do paměti (navíc k tomu větvení a instrukce cyklu). počítá se funkce ADD jako dvě čtení z paměti, dvě sčítání a dva zápisy do paměti.

Je více možností jak provést Montgomeryho násobení, stejně jako je mnoho metod pro obyčejné násobení. Algoritmy můžeme rozdělit podle dvou kritérií. První kriterium je, zda-li násobení a redukce jsou oddělené (*separated*) nebo integrované (*integrated*). V přístupu kde jsou separované nejdříve násobíme a pak provedeme Montgomeryho redukci. V přístupu kde jsou násobení a redukce spojené se násobení a redukce prolínají. Tahle integrace může být buď hrubě strukturovaná nebo jemně strukturovaná, to záleží na tom, jak často se přepíná mezi násobením a redukcí (přesněji jde-li o zpracování pole slov nebo jednoho slova); existují i implementace, které jsou kompromisem mezi oběma přístupy.

Druhým kriteriem je podoba kroků násobení a redukce. První typ je *operand scanning*, kde vnější cyklus prochází slova jednoho z operandů; druhý typ je *product scanning*, kde vnější cyklus prochází slova samotného součinu. Tato vlastnost je nezávislá na té první; navíc je možné pro násobení mít jeden přístup a pro redukci ten druhý, i když jde o integrovaný přístup.

Pro všechny případy bude uvažováno, že algoritmy budou popisovány jako operace na libovolném počtu bitů. Takže algoritmy mohou být přímo přepsány pro danou platformu.

Celkem málo algoritmů je v praxi přijatelných a použitelných, ale v této práci bude uvedeno pět zástupců z celé řady, právě ty, které mají dobré implementační charakteristiky. Mezi pět algoritmů, o kterých bude řeč, patří:

3.1 Separated Operand Scanning (SOS),
3.2 Coarsely Integrated Operand Scanning (CIOS),
3.3 Finely Integrated Operand Scanning (FIOS)**,**
3.4 Finely Integrated Product Scanning (FIPS),
3.5 Coarsely Integrated Hybrid Scanning (CIHS)**.**

3.1 Separated Operand Scanning (SOS)

První analyzovanou metodou pro výpočet $\mathrm{MonPro}(a,b)$ je metoda zvaná Separated Operand Scanning. V této metodě se nejprve spočte součin $a \cdot b$ např. použitím:

```
for i=0 to s-1
      C := 0
      for j=0 to s-1
            (C,S) := t[i+j] + a[j] . b[i] + C
            t[i+j] := S
t[i+s] := C
```

kde *t* je inicializováno na 0. Získaná konečná hodnota je celé číslo *t*, které má 2*s* slov a je uloženo v

```
t[0], t[1], ...., t[2s-1].
```

Pak vypočteme *u* použitím vzorce $u := (t + m \cdot n)/r$, kde $m := t \cdot n' \bmod r$. Za účelem výpočtu *u*, provedeme nejdříve přiřazení $u = t$, a pak přičteme $m \cdot n$ použitím standardní násobící rutiny, a na konec to vydělíme číslem $r = 2^{sw}$, které je provedeno ignorováním nejnižších *s* slov z *u*. Když $m = t \cdot n' \bmod r$ a redukční proces zpracuje slovo za slovem, pak můžeme použít $n'_0 = n' \bmod 2^w$ místo *n'*. Toto bude aplikováno na všech pět uvedených metod. Pak po spočtení *t* násobením *a* a *b* užitím kódu uvedeným nahoře, se bude pokračovat prováděním následujícího kódu, který přepíše *t* za účelem výpočtu $t + m \cdot n$.

```
for i=0 to s-1
      C := 0;
      m :=t[i] . n'[0] mod W
      for j=0 to s-1
            (C,S) := t[i+j] + m.n[j] + C
            t[i+j] := S
      ADD(t[i+s], C)
```

V tomto kódu předvedená funkce ADD zajišťuje propagaci přenosu přičítaného C do vstupního pole daného prvním argumentem a začínající počátečním indexem $(t[i+s])$, a propagující přenos tak dlouho, dokud je generován. ADD funkce je potřeba pro propagaci přenosu nahoru do posledního slova t, který rozšíří velikost t na $2s$ slov a jeden bit. Tento bit je vložen do samostatného slova, takže nová velikost t je $2s+1$ slov. (Tento extra bit a jedno extra slovo navíc ve kterém je uložen je použito ve všech zde uvedených algoritmech. Cesta jak se většinou dá vyhnout jednomu extra slovu navíc, je definovat s jako délku ve slovech po $2n$ bitech, a ne jako *modul n*. Toto s bude stejné jako to dosavadně definované, až na to, že délka bude n násobkem délky slova a v tomhle případě jen o jedno slovo delší.) Vypočítaná hodnota t je pak vydělena číslem r, které vznikne z čísla t ignorováním s nejnižších slov. Tento krok vypadá takto:

```
for j=0 to s
      u[j] := t[j+s]
```

Konečně získáváme číslo u v $s+1$ slovech. Odečítání na libovolném počtu bitů v třetím kroku funkce $\mathrm{MonPro}(\ )$je provedeno pokud je třeba redukovat u. 3 krok může být proveden pomocí následujícího kódu:

```
B := 0
for i=0 to s-1
      (B,D) := u[i] - n[i] - B
      t[i] := D
(B,D) := u[s] - B
t[s] := D
if B=0 then return t[0], t[1], ... , t[s-1]
      else return u[0], u[1], ... , u[s-1]
```

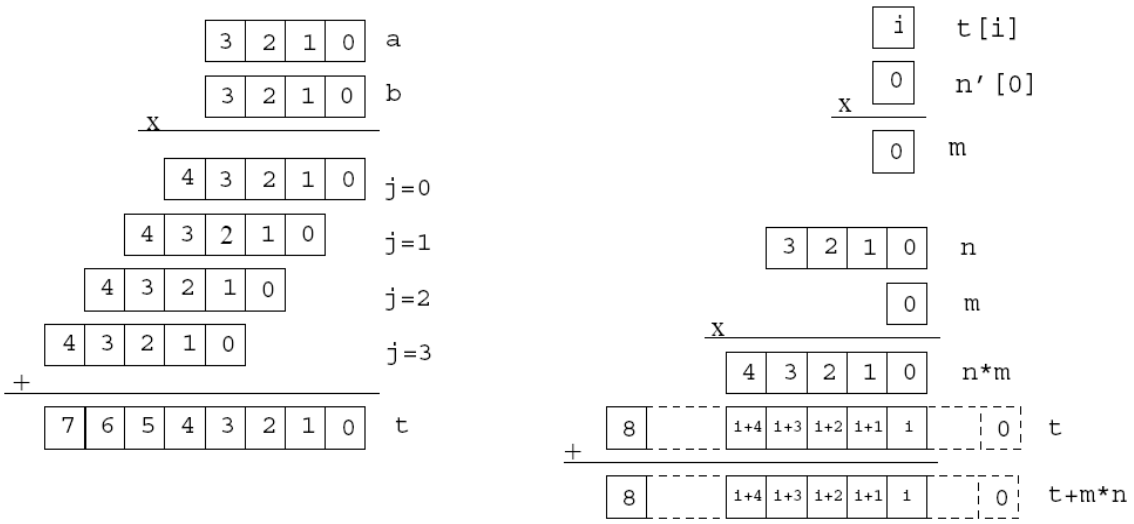
Třetí krok je pro všechny zde popsané algoritmy stejný, a proto nebude opakován v ostatních algoritmech. Ale časová a paměťová složitost se v hodnocení algoritmů vyskytne. Operace uvedené nahoře obsahují $2(s+1)$ sčítání, $2(s+1)$ čtení a $(s+1)$ zápisů do paměti.

Shrnutí metody SOS, založené na technikách pro výpočet počtu operací ukáže, že je třeba $2s^2+s$ násobení, $4s^2+4s+2$ sčítání, $6s^2+7s+3$ čtení z paměti a $2s^2+6s+2$ zápisů do paměti. Metoda SOS požaduje celkových $2s+2$ slov pro dočasné mezivýsledky, které jsou používány pro uložení $2s+1$ slovního pole *t* a jednoslovní proměnné *m*. Metoda SOS je předvedena na Obrázek 3.1.1.

Hodnota n_0', která je definována jako inverze nejmenšího významového slova $n \bmod 2^w$ tedy jako $n_0' = -n_0^{-1}(\bmod 2^w)$, může být spočtena použitím jednoduchého algoritmu uvedeného v [1]. Dále ještě důvod pro rozdělení výpočtu součinu $a.b$ od zbytku kroků pro výpočet *u* je ten, že když $a=b$ je možné optimalizovat algoritmus Montgomeryho násobení pro umocňování. Optimalizace umocňování je dosaženo, protože skoro polovina násobení může být přeskočena, když $a_i \cdot a_j = a_j \cdot a_i$. Následující jednoduchý kód nahradí první část algoritmu Montgomeryho násobení aby správně pracovalo optimalizované Montgomeryho umocňování:

```
for i=0 to s-1
(C,S) := t[i+i] + a[i]*a[i]
for j = i+1 to s-1
      (C,S) := t[i+j] + 2*a[j]*a[i] + C
      t[i+j] := S
t[i+s] := C
```

Je zde jedna choulostivá část, hodnota ***2*a[j]*a[i]*** potřebuje pro uložení víc než dvě slova; pokud nemá hodnota *C* extra bit, pak jediná cesta jak to udělat je přepsat cyklus tak, že členy ***a[j]*a[i]*** jsou přičteny jako první, bez vynásobení 2; výsledek je zdvojen a členy ***a[i]*a[i]*** jsou přičteny.



Obrázek 3.1.1: Metoda Separated Operand Scanning (SOS) pro $s = 4$. Operace násobení $t = a \times b$ je ilustrována nalevo. Aby jsme dostali *m*, je n'_9 vynásobeno každým slovem *t*. Konečný výsledek je získán sečtením posunutého $n \times m$ a *t*, jak je ukázáno napravo.

3.2 Coarsely Integrated Operand Scanning (CIOS)

Další metoda Coarsely Integrated Operand Scanning vylepšuje první algoritmus integrací násobení a redukčního kroku. Přesněji místo výpočtu úplného součinu *a.b* a pak redukce, se střídá opakování vnějšího cyklu pro násobení s redukcí. To se provádí, pokud je hodnota *m* v *i-té* iteraci vnějšího cyklu pro redukci závislá pouze na hodnotě ***t[i]***, která je kompletně vypočtena v *i-té* iteraci vnějšího cyklu pro násobení. Toto vede k následujícímu algoritmu:

```
for i=0 to s-1
      C := 0
      for j=0 to s-1
            (C,S) := t[j] + a[j]*b[i] + C
            t[j] := S
      (C,S) := t[s] + C
      t[s] := S
      t[s+1] := C
      C := 0
      m := t[0]*n'[0] mod W
      for j=0 to s-1
            (C,S) := t[j] + m*n[j] + C
            t[j] := S
      (C,S) := t[s] + C
```

```
t[s] := S
t[s+1] := t[s+1] + C
for j=0 to s
      t[j] := t[j+1]
```

Pole *t* je nejdříve nastaveno na samé nuly. Poslední *j-cyklus* je použit k posunutí výsledku doprava o jedno slovo, proto reference na **t[j]** a **t[0]** místo **t[i+j]** a **t[i]**. Nepatrné vylepšení v integraci posunu do redukce je předvedeno dále:

```
m := t[0] *n'[0] mod W
(C,S) := t[0] + m * n[0]
for j=1 to s-1
      (C,S) := t[j]  + m * n[j] + C
      t[j-1] := S
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s+1] + C
```

Pomocné pole *t* využívá pouze $s+2$ slov. Toto je způsobeno tím, že posun je prováděn vždy jen o jedno slovo, raději než o několik slov najednou, což ušetří $s-1$ slov. Konečný výsledek je prvních $s+1$ slov v poli *t*.

CIOS metoda požaduje $2s^2+s$ násobení, $4s^2+4s+2$ sčítání, $6s^2+7s+2$ čtení a $2s^2+5s+1$ zápisů a užívá během výpočtu $s+3$ slova v paměti. Redukce paměťové složitosti je hlavním vylepšením oproti SOS metodě.

Integrace v této metodě je "hrubší", protože nahrazuje iteraci vnějšího cyklu. V další metodě budě nahrazena iterace vnitřního cyklu.

3.3 Finely Integrated Operand Scanning (FIOS)

Tato metoda integruje dva vnitřní cykly metody CIOS do jednoho, počítáním násobení a sčítání ve stejném cyklu. Násobení $a_i \cdot b_i$ a $m \cdot n_i$ jsou vypočtena ve stejném cyklu, a pak je přičteno konečné *t*. V tomto případě musí být t_0 vypočteno před vstupem do cyklu, protože *m* je závislé na této hodnotě, která koresponduje s rozvinutím první iterace cyklu pro $j=0$.

```
for i=0 to s-1
(C,S) := t[0] + a[0] * b[i]
ADD(t[1],C)
m := S * n'[0] mod W
```

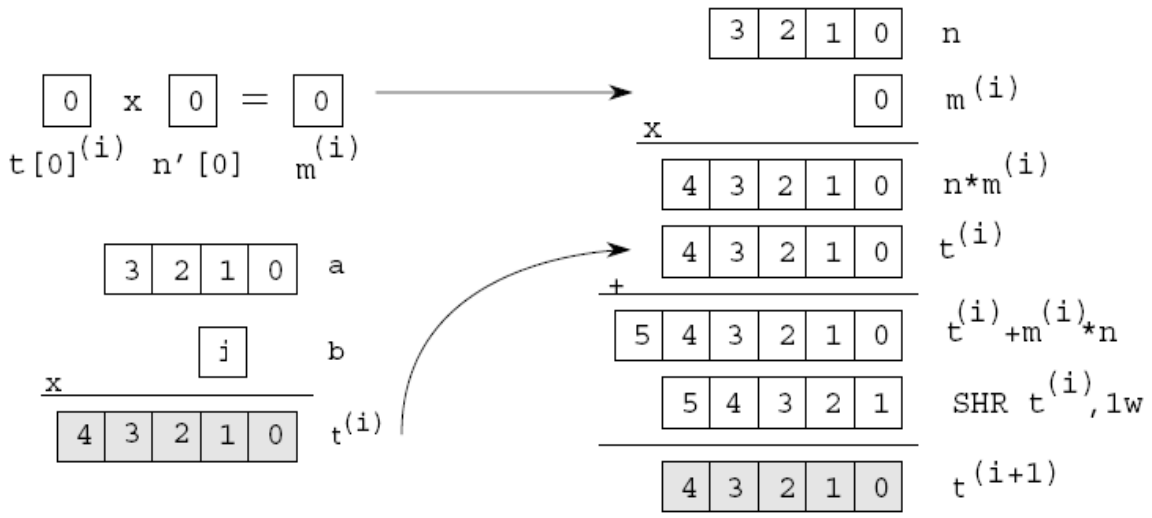
```
(C,S) := S + m * n[0]
```

Dílčí součin $a \cdot b$ je spočítán jeden po druhém pro všechny hodnoty *i*, pak je součin $m \cdot n$ přičten k dílčímu součinu. Tahle suma je dále posunuta jedno slovo doprava, a tak připraví *t* na další iteraci.

```
for j=1 to s-1
      (C,S) := t[j] + a[j] * b[i] + C
      ADD(t[j+1], C)
      (C,S) := S + m * n[j]
      t[j-1] := S
(C,S) := t[s] + C
t[s-1] := S
t[s] := t[s-1] + C
t[s-1] := 0
```

Rozdíl mezi metodou CIOS a touto metodou je ten, že metoda FIOS má pouze vnitřní cyklus. Algoritmus je předveden na Obrázek 3.3.1. ADD funkce je požadována ve vnitřním *j-cyklu* pro dva různé přenosy, jeden vyplývající z násobení $a_j \cdot b_i$ a druhý z násobení $m \cdot n_j$. (Tím se výhoda jednoho cyklu vyvažuje požadavkem ADD funkce.) Pole *t* je nastaveno po inicializaci na samé nuly.

Metoda FIOS požaduje $2s^2 + s$ násobení, $5s^2 + 3s + 2$ sčítání, $7s^2 + 5s + 2$ čtení a $3s^2 + 4s + 1$ zápisů, včetně konečného odčítání. To je asi s^2 víc sčítání, zápisů a čtení než u metody CIOS. Celkové množství dočasně potřebného prostoru je $s + 3$ slov.



Obrázek 3.3.1: Iterace metody Finely Integrated Operand Scanning (FIOS). Výpočet částečného součinu $t^{(i)} = a \times b_i$, který je na obrázku nalevo, umožní výpočet m^i v této iteraci. Pak je mezivýsledek $t^{(i+1)}$ nalezen přičtením n.m^i k tomuto částečnému součinu, jak je ukázáno na obrázku vpravo.

3.4 Finely Integrated Product Scanning (FIPS)

Tak jako předchozí, tato metoda prokládá výpočet $a \cdot b$ a $m \cdot n$, ale zde jsou oba výpočty ve formě součinu (product scanning). Metoda udržuje hodnoty *m* a *u* ve stejném poli *m*, které má *s* slov. První cyklus o kterém byla řeč už dříve spočte jednu část součinu $a \cdot b$ a pak přičte $m \cdot n$. Pole tří slov

```
t[0], t[1], 1[2],
```

se používá jako úložiště pro součiny $a \cdot b$ a $m \cdot n$. Používání tří slovního pole předpokládá, že $s < W$; obvykle potřebujeme $\log_W(sW(W-1))$ což je přibližně $2 + \log_W(s)$ slov. Algoritmus je jednoduše modifikovatelný pro zvládnutí většího akumulátoru.

```
for i=0 to s-1
      for j=0 to i-1
            (C,S) := t[0] + a[j] * b[i-j]
```

```
            ADD(t[1], C)
            (C,S) := S + m[j] * n[i-j]
            t[0] := S
            ADD(t[1], C)
      (C,S) := t[0] + a[i] * b[0]
      ADD(t[1],C)
      m[i] := S*n'[0] mod W
      (C,S) := S + m[i] * n[0]
      ADD(t[1], C)
      t[0] := t[1]
      t[1] := t[2]
      t[2] := 0
```

V tomto cyklu je *i*-té slovo *m* spočteno pomocí n'_0, a pak poslední nejméně významové slovo $m \cdot n$ je přičteno k *t*. Protože poslední významové slovo *t* bude vždy nulové, posun o jedno slovo může být uskutečněn vždy v čase jedné iterace. Pole *t* je při inicializaci nastaveno na *0*.

Druhý cyklus *i*, který jsme už viděli dříve ukončí výpočet srovnáním konečného výsledku *u,* slovo po slově, do paměti o velikosti *m*.

```
for i = s to 2s-1
      for j = i-s+1 to s-1
            (C,S) := t[0] + a[j] * b[i-j]
            ADD(t[1],C)
            (C,S) := S + m[j] * n[i-j]
            t[0] := S
            ADD(t[1],C)
      m[i-s] := t[0]
      t[0] := t[1]
      t[1] := t[2]
      t[2] := 0
```

Při zkoumání znaků druhého *i*-cyklu zjistíme, že posledních *s* nejméně významových slov výsledku *u* je umístěno v proměnné *m*. První významový bit je v **`t[0]`**. (Hodnoty **`t[1]`** a **`t[2]`** jsou na konci nulové.)

Metoda FIPS požaduje $2s^2 + s$ násobení, $6s^2 + 2s + 2$ sčítání, $9s^2 + 8s + 2$ čtení z paměti a $5s^2 + 8s + 1$ zápisů do paměti. Počet sčítání, čtení z paměti a zápisů do paměti je o něco málo větší než u předchozích metod, ale počet násobení zůstává stejný. (Všimněme si, že mnoho zápisů a čtení je kvůli slovům v akumulátoru, která by mohla být v registrech.) Potřebný paměťový prostor je $s + 3$ slov.

3.5 Coarsely Integrated Hybrid Scanning (CIHS)

Tato metoda je modifikací metody SOS. Ilustruje jiný přístup k Montgomerymu násobení. Jak už bylo ukázáno, metoda SOS požaduje $2s+2$ slov pro ukládání pomocných proměnných *m* a *t*. Zde bude vidět, že je možné použít pouze $s+3$ slov dočasné paměti, bez změny hlavního toku algoritmu. Metoda byla nazvána Hybrid Scanning, protože pracuje jak v režimu násobení Product Scanning, tak i v režimu Operand Scanning. (Redukce je jen v Operand Scanning formě.) Jako první rozdělíme výpočet $a \cdot b$ do dvou cyklů. Druhý cyklus bude posouvat mezivýsledek o jedno slovo na konci každé iterace.

Rozdělení násobení je možné, protože *m* je počítáno násobením *i*-tého slova z *t* číslem n'_0. Tak může být násobení $a \cdot b$ zjednodušeno odsunutím násobení slova potřebného pro nejvyšší významovou půlku *t* do druhého *i*-cyklu. Cyklus násobení může být integrován do druhého hlavního *i*-cyklu, který bude počítat jeden částečný součin v každé iteraci a redukovat prostor pole *t* z $2s+1$ na $s+2$ slov. V první části, je spočteno $(n-j)$ slov částečného součinu $a \cdot b$ a přičteno k *t*. Na Obrázek 3.5.1 jsou ukázány vypočtené části částečných součinů rovnými čarami a sečtené výsledky jsou ve šrafovaných blocích. Tento výpočet může být proveden následujícím kódem:

```
for i=0 to s-1
      C := 0
      for j=0 to s-i-1
            (C,S) := t[i+j] + a[j] * b[i] + C
            t[i+j] := S
      (C,S) := t[s] + C
      t[s] := S
      t[s+1] := C
```

Násobení $m \cdot n$ je pak prokládáno sčítáním $a \cdot b + m \cdot n$. Dělení číslem *r* je provedeno posunem o jedno slovo během *i*-cyklu. Protože *m* je dlouhé jedno slovo a součin $m \cdot n + C$ je dlouhý dvě slova, suma $t + m \cdot n$ spotřebuje nejvíce $s+2$ slov. Propagace přenosu do *s*-tého slova je provedena do $(s-1)$ho slova právě po posunutí. Pole *t* je inicializováno na *0*.

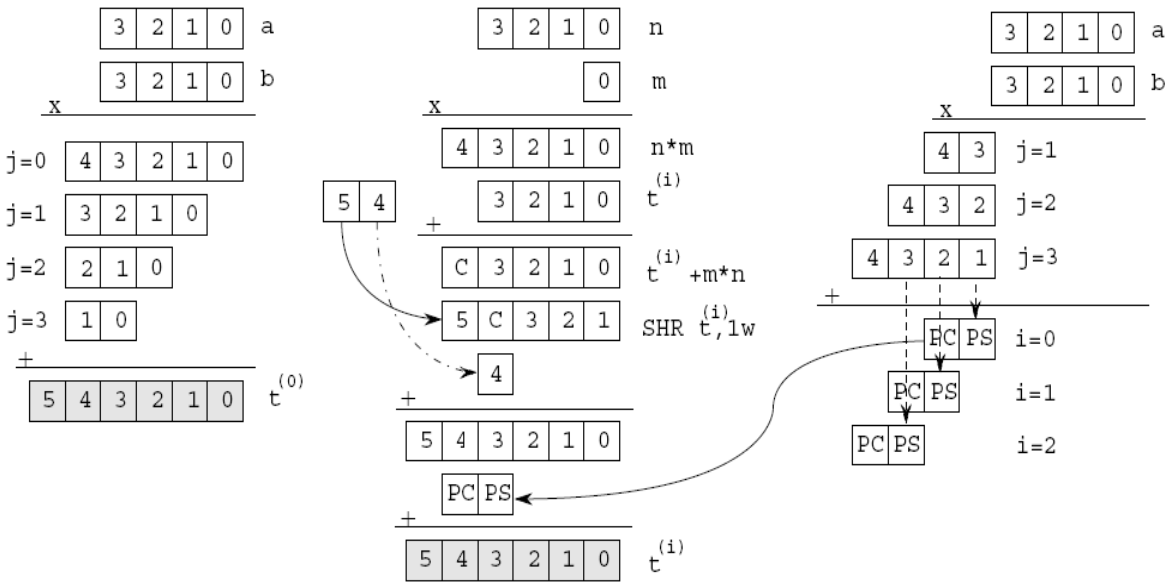
```
for i=0 to s-1
      m:=t[0] * n'[0] mod W
      (C,S) := t[0] + m*n[0]
      for j:= 1 to s-1
```

```
        (S,C) := t[j] + m*n[j] + C
        t[j-1] := S
  (C,S) := t[s] + C
  t[s-1] := S
  t[s] := t[s+1] + C
  t[s+1] := 0
```

Výpočet *m* potřebuje využívat t_0 místo t_i, jako tomu bylo v algoritmu SOS. Toto je důvod pro posunutí *t* v každé iteraci. Dvě přebytečná slova spočtená v prvním cyklu jsou použita v následujícím *j*-cyklu, který počítá $(s+1)$ní slovo součinu $a \cdot b$.

```
for j=i+1 to s-1
        (C,S) := t[s-1] + b[j] * a[s-j+i]
        t[s-1] := S
        (C,S) := t[s] + C
        t[s] := S
        t[s+1] := C
```

Poznamenejme, že čtyři řádky nahoře spočítají tři nejvyšší významové slova z *t*, to znamená slova s indexy $(s-1)$, *s* a $(s+1)$ z pole *t*. Kód nahoře je kompletní první krok pro $\mathrm{MonPro}(\mathrm{a},\mathrm{b})$. Potom pokud je $t \geq n$, je *n* odečteno od *t*. Algoritmus je předveden na Obrázek 3.5.1 pro Montgomeryho násobení dvou čísel o čtyřech slovech. Zde symboly PC a PS označují dvě extra slova požadovaná pro získání správného $(s+1)$ního slova. Na Obrázek 3.5.1 je každá dvojice $(\mathrm{PC},\mathrm{PS})$ suma slov ve sloupci spojeného přerušovanou čarou. Počet požadovaných násobení v této metodě je roven $2s^2+s$. Počet sčítání klesl na $4s^2+4s+2$. Počet čtení z paměti je $6.5s^2+6.5s+2$ a počet zápisů do paměti je $3s^2+5s+1$. Jak už bylo zmíněno dříve tento algoritmus potřebuje $s+3$ slov dočasné paměti.



Obrázek 3.5.1: Iterace metody Coarsely Integrated Hybrid Scanning (CIHS) pro $s = 4$. Levá strana obrázku ukazuje získávání pravé půlky částečného součinu $a \times b$, který je proveden v prvním *i*-cyklu. Druhý *i*-cyklus je zobrazen ve dvou částech uprostřed a napravo. Přičtení $n \times m$ k *t* a posun $t + m \times n$ jsou na obrázku uprostřed a jsou provedeny v prvním *j*-cyklu druhého *i*-cyklu. Výpočet zbylých slov částečného součinu $a \cdot b$ je na obrázku úplně napravo. Každý $(\mathrm{PC}, \mathrm{PS})$ pár je součet sloupců spojených s řádky. Jak je ukázáno dole uprostřed obrázku, pár $(\mathrm{PC}, \mathrm{PS})$ se sečte s t^i, což je provedeno v posledním j-cyklu.

4 Aplikační oblasti Montgomeryho násobení

Aplikační oblastí modulárního násobení je hlavně asymetrická kryptografie. Jsou to kryptografické algoritmy, založené na generování páru klíčů: veřejného klíče a soukromého klíče. Veřejný je volně distribuován a soukromý je udržován v tajnosti tím kdo pár vygeneroval.

Pokud odesílatel pomocí veřejného klíče šifruje data a zasílá je majiteli soukromého klíče, jde o utajení zprávy. Používá se při komunikaci přes nezabezpečený kanál pro zajištění důvěrnosti. Jenom soukromým klíčem je možné zprávu dešifrovat.

Druhý případ použití je digitální podpis, který slouží k označení odesilatele zprávy. Zpráva zašifrovaná soukromým klíčem je dešifrovatelná kýmkoliv, ale zároveň prokazuje, že ji odeslal

autor, protože nikdo jiný ji nemůže zašifrovat tak, aby byla veřejným klíčem dešifrovatelná. Soukromý klíč spojuje elektronický podpis s konkrétní osobou. Z hlediska utajení zprávy nemá elektronický podpis žádný význam. Používá se při komunikaci přes nezabezpečený kanál pro zajištění autentizace, integrity a nepopiratelnosti.

Mezi široce rozšířené algoritmy asymetrické kryptografie, které využívají výhod Montgomeryho násobení patří RSA, výměna klíčů Diffie-Hellman (DH), Digital Signature Algorithm (DSA), a další kryptografické systémy založené na eliptických křivkách (EC-DH, EC-DSA). Všechny tyto algoritmy běžně pracují s velmi dlouhými slovy. Velikost slov se pohybuje v rozmezí od 160 do 2048 bitů. Velké délky slov jsou nutné pro zajištění požadovaného stupně bezpečnosti těchto algoritmů. Jsou to algoritmy založené na modulárním umocňování, které je počítáno opakovaným modulárním umocňováním na druhou a násobením (RSA, HD, DSA). Nebo modulární aritmetika určuje požadavky na čas a zdroje (EC-DH, EC-DSA).

4.1 Diffie-Hellman DH

Problém distribuce klíče se stal klíčovým problémem symetrického šifrování a základním úkolem moderní kryptografie. Bez jeho vyřešení nebylo možné masově používat symetrické šifrovací algoritmy v počítačové komunikaci. Hledáním způsobu, jak provést bezpečnou výměnu klíče otevřenou cestou, se zabývali vědci Whitfield Diffie a Martin Hellman. Této dvojici se po několikaletém snažení podařilo řešení najít. Předpokládejme, že dvě strany A a B si chtějí vyměnit otevřenou cestou klíč, který jim umožní šifrovat a dešifrovat následnou komunikaci. Nejdříve si vymění veřejnou informaci, s jejíž pomocí vygenerují stejný symetrický klíč.

Následující popis objasňuje podstatu algoritmu a způsob použití modulární aritmetiky:

- Odesílatel a příjemce se dohodnou na dvou veřejných parametrech M a N, kde M je prvočíslo a N určí generátor a platí $N < M$.
- Odesílatel generuje náhodné privátní číslo x a vypočítá veřejnou hodnotu $O = N^x \bmod M$, kterou odešle príjemci.
- Příjemce generuje náhodné privátní číslo y a vypočítá veřejnou hodnotu $P = N^y \bmod M$, kterou odešle odesilateli.
- Odesilatel obdrží od příjemce hodnotu P, ze které vypočte klíč $K_{PO} = (P)^x \bmod M$.

- Příjemce obdrží od odesilatele hodnotu *O*, ze které vypočte klíc $K_{OP} = (O)^y \bmod M$.
- Protože platí $K_{PO} = K_{OP} = K$, mohou další komunikaci odesilatel i příjemce šifrovat tímto klíčem.

Bezpečnost algoritmu je založena na obtížnosti výpočtu diskrétních logaritmů. Což i dnes není problém výpočetně zvládnutelný v rozumném čase.

4.2 RSA

Je nejznámější a nejpopulárnější asymetrická šifra a je výsledkem úsilí pracovní skupiny, kterou tvořila trojice matematiků Donald Rivest, Adi Shamir a Leonard Adleman. Inspirováni koncepcí asymetrického šifrování, kterou zveřejnil W.Diffie. Jeho síla je založena na velmi obtížné faktorizaci velkých prvočísel, tj. na v současnosti neznámé dostatečně rychlé metodě umožňující rozklad velkých čísel na součin prvočísel.

Postup přípravy klíčů pro algoritmus RSA lze rozepsat do následujících kroků:

- Zvolíme dvě dostatečně velká prvočísla *p*, *q* a určíme jejich součin *N*.
- Určíme přirozené číslo *s* nesoudělné s $(p-1)*(q-1)$.
- Dvojice čísel $\{s, N\}$ představuje veřejný klíč.
- Určíme číslo *t* takové, že $t \cdot s = 1 \bmod (p-1)\cdot(q-1)$. Protože stejný požadavek je kladen i na veřejné číslo *s*, existuje právě jedno řešení úlohy.
- Dvojice čísel $\{t, N\}$ představuje soukromý klíč.
- Čísla *p*, *q* nejsou dále potřebná.

Prostředky modulární aritmetiky lze algoritmus RSA vyjádřit následující větou: Jsou-li *s*, *p* a *N* parametry algoritmu RSA, pak pro každé číslo *x* z intervalu $\langle 1; N)$ platí $x = x^{st} \bmod N$.

Šifrování zprávy se provádí pomocí veřejného klíče a schématu (nebo také kongruence) $SZ = OT^s \bmod N$. Pro dešifrování informace se naopak použije klíč soukromý $OT = SZ^t \bmod N$. Postup však lze i obrátit, tj. šifrovat soukromým klíčem a dešifrovat klíčem veřejným.

4.3 Digital Signature Algorithm (DSA)

Je algoritmus pro digitální podpis navržený v roce 1991 organizací The National Institute of Standards and Technology (NIST), původně National Bureau of Standards. Možným řešením problému jsou kryptografické systémy s veřejným klíčem, je-li možné je použít i opačným směrem. RSA je typický příklad a velmi často používaný algoritmus.

Postup je následující. K posílané zprávě se přiřadí kopie zprávy, která je pro objem redukovaná na hash hodnotu. Tato hash hodnota se zašifruje soukromým klíčem odesilatele a připojí se ke zprávě. Při příjmu zprávy příjemce oddělí zprávu od zašifrované hash hodnoty.
Spočte hash zprávy a dešifruje přijatou hash hodnotu. Pokud se shodují je to potvrzením o autenticitě odesilatele.

Jak algoritmus pracuje? Mějme čísla *P, Q, G, X, Y, K, M, R, S, M′, R′, S′* která splňují:

- *P* je prvočíslo, *Q* dělí $(P-1)$ a je také prvočíslo.
- $G = H^{((P-1)/Q)} \bmod P$, kde *H* je libovolné celé číslo, pro které platí $(1 < H < P-1)$ a $H^{((P-1)Q)} \bmod P > 1$.
- *X* je náhodně vygenerované a platí, že $(0 < X < Q)$, $X = G^X \bmod P$.
- *K* je náhodně vygenerované a platí, že $(0 < K < Q)$.

Pak veřejným klíčem je sada čísel $[P, Q, G, Y]$, přičemž *P*, *Q* a *G* mohou být sdíleny skupinou uživatelů, soukromým klíčem je číslo *X*. Původní zprávou je číslo *M*, digitálním podpisem dvojice (R, S), ověřovanou zprávou číslo M' a ověřovaným digitálním podpisem dvojice (R', S').

Digitální podpis (R, S) vytvoříme pomocí vzorců $R = (G^K \bmod P) \bmod Q$, $S = ((Sha(M) + X * R)/K) \bmod Q$, přičemž $Sha(\ )$ je funkce, která vrací výsledek algoritmu SHA-1 (což je 160ti bitový řetězec) převedený na celé číslo. Pokud se při výpočtu stane, že *R* nebo *S* bude rovno nule, vygeneruje se nové číslo *K* a celý postup se opakuje.

Ověření digitálního podpisu provedeme následovně. Nejdříve zjistíme zda platí dvě podmínky: $(0 < R' < Q)$ a $(0 < S' < Q)$. Pokud neplatí, podpis není platný. Pokud ano, pokračuje se dále. Dejme $W = (1/S') \bmod Q$, $U_1 = (Sha(M') * W) \bmod Q$,

$U_2 = (R' * W) \bmod Q$ a $V = ((G^{U_1} * Y^{U_2}) \bmod P) \bmod Q$. Jestliže pak platí, že $R' = V$, pak je možné prohlásit podpis i zprávu za autentické.

4.4 Šifry založené na eliptických křivkách ECDSA

Nová perspektivní oblast moderní aplikované kryptografie vzniká z čisté algebry po více než sto letech výzkumu v roce 1985 (pracemi V. Millera a N. Koblitze). V současné době pronikly eliptické kryptografické systémy do řady světových standardů a staly se alternativou ke „klasickému" RSA i DSA. Mají své výhody zejména v rychlosti a menší náročnosti na hardware i software. Bezpečnost algoritmu je založena na obtížnosti výpočtu diskrétních logaritmů.

Generování dvojice klíčů pro eliptické kryptografické systémy. Daná množina parametrů systému je asociována s dvojicí klíčů, která je vytvářena následovně:

- Zvolíme eliptickou křivku *E* nad $GF(p)$. Počet bodů křivky *E* by měl být dělitelný velkým prvočíslem *n*.
- Zvolíme bod $P \in E$ řádu *n* (norma ANSI X9.62 požaduje aby řád křivky byl větší než 2^{160}).
- Vybereme jedinečnou a nepredikovatelnou hodnotu privátního klíče, číslo $d \in [1, n-1]$.
- Vypočteme veřejný bod $Q = dP$.
- Veřejný klíč tvoří čtveřice (E, P, n, Q).

Vytvoření podpisu pomocí schématu ECDSA. Mějme zprávu *M*.

- Vybereme jedinečné a nepredikovatelné číslo $k \in [1, n-1]$.
- Vypočteme bod $kP = (x_1, y_1)kP$ a číslo $r = x_1 \bmod n$.
- Je-li $r = 0$, pak postup opakujeme od generování čísla *k* (to je nutné proto, aby v hodnotě *s* byl obsažen privátní klíč, viz dále).
- Vypočteme $k^{-1} \bmod n$.
- Vypočteme $s = k^{-1}(h(M) + dr) \bmod n$, kde *h* je hash funkce SHA-1 (viz [1]).
- Je-li $s = 0$, pak opět jdeme na první bod generování nového *k* (neexistovalo by $s^{-1} \bmod n$, viz dále proces ověření).

- Podpisem zprávy M je dvojice čísel (r,s).

Ověření podpisu ECDSA. Mějme zprávu M a její podpis (r,s).

- Důvěryhodným způsobem získáme veřejný klíč podepisujícího (E,P,n,Q).
- Ověříme, že $r,s \in \langle 1, n-1 \rangle$.
- Vypočteme $w = s^{-1} \bmod n$ a $h(M)$.
- Vypočteme $u_1 = h(M)w \bmod n$ a $u_2 = rw \bmod n$.
- Vypočteme $u_1 P + u_2 Q = (x_0, y_0)$ a $v = x_0 \bmod n$.
- Podpis je platný právě tehdy, když $v = r$.

5 Implementace algoritmu

Algoritmus jsem implementoval pomoci knihovny LIP určené pro experimentování s kryptografickými algoritmy. Balík obsahuje celou řadu funkcí ze světa kryptografie, od základních aritmetických funkcí přes funkce pro posuny, bitové manipulace, porovnávání, logaritmy, konverze mezi typy, modulární aritmetiku, Montgomeryho modulárrní aritmetiku, generování náhodných čísel, až po funkce určené pro faktorizaci. Pro další seznámení se s knihovnou LIP doporučuji studium její dokumentace, která je dodávána s knihovnou.

Algoritmus je implementován v souboru `montgomery.c`. Jde o metodu typu FIOS, protože násobení a redukce nejsou oddělené a pracuje v režimu operand scanning, což znamená, že prochází vždy slova jednoho operandu.

Součástí algoritmu jsou funkce `Rad_rep()` a `Mon_pro()`. Pomocná funkce `Rad_rep()` provádí rozklad celého čísla u v číselné soustavě se základem r (v našem případě $r = 10$) do pole podle vah jednotlivých číslic a to vzestupně. Podle pseudokódu uvedeného níže.

Vstupy: $u,\ r;\ u \geq 0,\ r \geq 2.$
Výstup: $u = (u_0, u_1, \ldots u_s)_r;\ s \geq 0,\ u_s \neq 0$ **když** $s \geq 1.$

$i := 0$
$x := u$
$q := \lfloor x/r \rfloor$
$u_i := x - qr$

```
while q > 0 do
  i := i + 1
  x := q
  q := ⌊x / r⌋
  u_i := x − qr
  return((u_0,u_1,...u_s))
```

Funkce `Mont_mult()` se pak stará o samotné Montgomeryho násobení. Funkčnost je předvedena na následujícím kódu.

```
Vstupy: n = (n_0,n_1,...n_s)_r, a = (a_0,a_1,...a_s)_r, b = (b_0,b_1,...b_s)_r,
        0 ≤ a,a < n, R = r^s, gcd(n,r) = 1, n' = −n^(−1) mod r.
Výstup: U = abR^(−1) mod n.

U := 0                (U = (u_0,u_1,...u_s)_r)
for i = 0 to x − 1
  t_i := (u_0 + a_i * b_0)n' mod r
  U := (U + a_i * b + t_i * n) / r
if U ≥ n then U := U − n
return(U)
```

Funkčnost algoritmu jsem ověřil na příkladu. Kde

$$n = 72639,\ R = 10^5,\ a = 5792,\ b = 1229.$$

Pak platí, že

$$s = 5,\ n' = -n^{-1} \bmod 10 = 1,\ abR^{-1} \bmod n = 39796.$$

Následující Tabulka 1 zobrazuje provedené kroky a mezivýsledky v implementovaném algoritmu.

i	a_i	a_ib_0	t_i	a_ib	t_in	U
0	2	18	8	2458	581112	58357
1	9	81	8	11061	581112	65053
2	7	63	6	8603	435834	50949
3	5	45	4	6145	290556	34765
4	0	0	5	0	363195	39796

Tabulka 1: Montgomeryho násobení $a \cdot b \cdot R^{-1} \bmod n = 39796$

Aplikováním algoritmu na výstupní hodnotu $a \cdot b \cdot R^{-1} \bmod n = 39796$ po první průchodu a na hodnotu $R^2 \bmod n = 6787$ dostaneme $a \cdot b \bmod n = 72385$. Průběh výpočtu zobrazuje Tabulka 2.

i	a_i	$a_i b_0$	t_i	$a_i b$	$t_i n$	U
0	7	42	2	278572	145278	42385
1	8	48	3	318368	217917	57867
2	7	42	9	278572	653751	99019
3	6	36	5	238776	363195	70099
4	0	0	9	0	653751	72385

Tabulka 2: Montgomeryho násobení $a \cdot b \bmod n = 72385$

6 Závěr

Prezentované typy algoritmu pro Montgomeryho násobení požadují stejný počet násobení, ale počty sčítání, čtení a zápisů se lehce liší. $4s^2 + 4s + 2$ je spodní hranice počtu sčítání. Tohoto počtu přesně dosahuje metoda SOS a CIOS. Počet operací a velikost pracovní paměti, které požadují metody ukazuje Tabulka 3. Celkový počet operací je součtem všech operací uvnitř cyklu a násobkem počtu iterací daného cyklu. Jako příklad slouží Tabulka 4, ve které je naznačen výpočet pro metodu CIOS.

Samozřejmě tyto výpočty nejsou úplně přesné; nepočítá se s plným využitím registrů k ukládání mezivýsledků, s velikostí cache paměti, s chybějícími instrukcemi a speciálními instrukcemi pro násobení a akumulování. Také není započítána režie cyklu, ukazatelová aritmetika, a podobné záležitosti, které určitě ovlivňují výkon.

Implementovaný algoritmus je plně funkční. Na jeho vstup je možné zadat celá čísla typu `long` tzn., že musí být v intervalu $\langle -2147483648, 2147483647 \rangle$ a musí splňovat už výše zmíněné podmínky. Pro výpočet $a \cdot b \cdot R^{-1} \bmod n$ potřebuje $2s(s+1)$ násobení, a tak pro výpočet $a \cdot b \bmod n$ je třeba $4s(s+1)$ násobení. V úvahu nebyl brán počet násobení pro získání hodnoty $R^2 \bmod n$, je brána jako před vypočítaná.

Přínos práce vidím hlavně v tom, že jsem se seznámil s celou řadou pro mě nových skutečností. Jako je modulární aritmetika, princip Montgomeryho násobení a jeho využití. Dále jsem nastudoval základy asymetrické kryptografie a některých kryptografických algoritmů. Práce shrnuje typy metod Montgomeryho násobení a jeho uplatnění v asymetrických kryptografických systémech.

7 Literatura

[1] Menezes, A., Van Oorschot, P., Vanstone, S.: *Handbook of aplied cryptography.* CRC Press, 1996.

[2] Koc, C. K., Alcar, T., Kaliski, B. S.: *Analyzing and Comparing Montgomery Multiplication Algorithms*. IEEE micro, pp. 26-33, June 1996.

[3] Bunimov, V., J., Schimmler, M., Tolg, B.: *A Compelxity-Effective Version of Montgomery's Algorithm*. Workshop on Complexity Effective Designs (WCDE), May 2002.

[4] Bosselaers, A., Govaerts, R., Vandewalle J.: *Comaparison of Three Modular Reduction Functions*. 1993.

8 Přílohy

Metoda	Násobení	Sčítání	Čtení	Zápisy	Paměť
SOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+3$	$2s^2+6s+2$	$2s+2$
CIOS	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	$s+3$
FIOS	$2s^2+s$	$5s^2+3s+2$	$7s^2+5s+2$	$3s^2+4s+1$	$s+3$
FIPS	$2s^2+s$	$6s^2+2s+2$	$9s^2+8s+2$	$5s^2+8s+1$	$s+3$
CIHS	$2s^2+s$	$4s^2+4s+2$	$6.5s^2+6.5s+2$	$3s^2+5s+1$	$s+3$

Tabulka 3: Časové a paměťové požadavky metod.

Výrazy	Operace				Iterace
	Násobení	Sčítání	Čtení	Zápisy	
`for i=0 to s-1`	-	-	-	-	-
`C := 0`	0	0	0	0	s
`for j=0 to s-1`	-	-	-	-	-
`(C,S):=t[j]+a[j]*b[i]+C`	1	2	3	0	s^2
`t[j]:=S`	0	0	0	1	s^2
`(C,S) := t[s] + C`	0	1	1	0	s
`t[s] := S`	0	0	0	1	s
`t[s+1] := C`	0	0	0	1	s
`m := t[0]*n'[0] mod W`	1	0	2	1	s
`(C,S) := t[s] + C`	1	1	3	0	s
`for j=1 to s-1`	-	-	-	-	-
`(C,S):=t[j]+m*n[j]+C`	1	2	3	0	s(s-1)
`t[j-1] := S`	0	0	0	1	s(s-1)
`(C,S) := t[s] + C`	0	1	1	0	s
`t[s-1] := S`	0	0	0	1	s
`t[s] := t[s+1] + C`	0	1	1	1	s
`Odečtení`	0	2(s+1)	2(s+1)	s+1	1
	$2s^2+s$	$4s^2+4s+2$	$6s^2+7s+2$	$2s^2+5s+1$	

Tabulka 4: Výpočet operací metody CIOS.