
TDK DOLGOZAT

Lázár Szilvia Melánia

„Tetszőleges” bitméretű egész számokkal végzett műveletek a moduláris aritmetika szabályai szerint, determinisztikus és pszeudó prímtesztek

Szerző:

Lázár Szilvia Melánia

Programtervező Informatikus szak

III. évfolyam

Témavezetők:

Dr. Liptai Kálmán

főiskolai tanár, rektor

Keresztes Péter

címzetes főiskolai docens

Tartalomjegyzék

1. Köszönetnyilvánítás	4
2. Kutatásom céljai	5
3. RSA titkosítás.....	6
4. Prímszámok és prímtesztok	7
5. Assembly programozási nyelv	9
5.1. A 32 bites és a 64 bites Assembly programozási nyelv összehasonlítása.....	10
5.1.1 Különbségek a szóhosszban	10
5.1.2. Különbségek a függvényhívási konvenciókban	10
5.1.3. Különbségek a regiszterkészletben	11
6. LNGINT & RLNGINT függvénygyűjtemény Assemblyben.....	12
6.1. Lucas-Lehmer prímteszt és az RLNGINT	13
7. Karatsuba-Ofman szorzó algoritmus.....	13
8. Lucas-Lehmer prímteszt 32 bites és 64 bites Assemblyben.....	14
9. Hagyományos általános iskolás szorzó algoritmus	14
9.1. Hagyományos általános iskolás szorzó algoritmus implementálása Assemblyben	18
9.2. Hagyományos általános iskolás szorzó algoritmus és a Karatsuba-Ofman szorzó algoritmus összehasonlításának története és eredményei	29
10. Kutatásom eredményei	33
11. Irodalomjegyzék.....	40

Köszönetnyilvánítás

Ezúton köszönöm meg témavezetőimnek hogy elkészülhetett ez a dolgozat:

Köszönöm Keresztes Péternek, aki végtelen türelmével minden szakmai segítséget megadott és inspirációt biztosított. Az Assembly iránti lelkesedése az elejétől kezdve magával ragadott, és nem is volt kérdés számomra, hogy ezzel a területtel akarok foglalkozni.

Köszönöm Dr. Liptai Kálmán rektor úrnak, aki az inspiráció mellett a munka matematikai részében vállalt jókora szerepet. A vele folytatott hosszú és tartalmas beszélgetések jelölték ki a dolgozat elméleti irányát.

Végül, de nem utolsósorban köszönöm az EKF számítástechnika és matematika tanszék minden oktatójának a segítséget és törődést, amit tanulmányaim alatt kaptam.

Köszönöm, hogy általuk közelebb kerülhettem ehhez a szép és hasznos tudáshoz.

Azt hiszem, hogy az életem egyik legjobb döntése volt az, hogy Assemblyvel foglalkozzak.

Kutatásom céljai:

- Az LNGINT és az RLNGINT függvénygyűjteményen keresztül összehasonlítani a 32bites, illetve a 64bites Assemblyben implementált algoritmusok futásának időbeli különbségét, különösen a Karatsuba-Ofman rekurzióval.
- A Karatsuba-Ofman rekurzió és a hagyományos szorzó algoritmus futási idejének összehasonlítása 64bites Assembly programozási nyelven.
- Lucas-Lehmer prímteszt futási idejének összehasonlítása kétféle szorzó algoritmussal, (a Karatsuba-Ofman rekurzióval, illetve a hagyományos szorzó algoritmussal) 64bites Assembly programozási nyelven implementálva.

Az RSA titkosításról röviden

Az RSA titkosítás nyilvános kulcsú aszimmetrikus titkosítás.

A nyilvános kulcsú azt jelenti, hogy a titkosításhoz szükséges egy nyilvános és egy titkos kulcs. Az aszimmetrikus azt jelenti, hogy a visszafejtő algoritmus nem a titkosító algoritmus megfordítása. Ennek a titkosításnak az erőssége a prímfaktorizáció nehézségében rejlik. Nem létezik olyan prímfaktorizációs algoritmus, amely nagy prímszámokra elegendően gyors.

A titkosításhoz szükségesek megfelelően nagy prímszámok ($n > 256$ bit) a p és q értékének kiválasztásához. Az n szám a p és q prímek szorzata. Ha eldobjuk a p és q számokat, de megtartjuk az n számot, nincs hatékony algoritmus arra, hogy visszaszerezzük a p és q számokat.

$\Phi(n)$ szám, a $(p-1)(q-1)$ számok szorzata, értéke az n -nél nem nagyobb relatív prímek száma. Két szám akkor relatív prím, ha az 1-en és (-1) -en kívül nincs más közös osztójuk.

Választunk egy $d > 1$ számot, úgy, hogy a d és a $\Phi(n)$ LNKO-ja 1 legyen.

Meghatározzuk e értékét. e értékének

Nagyobbnak kell lenni, mint 1

Kisebbnek kell lennie, mint $\Phi(n)$ és

$e \cdot d$ elosztva $\Phi(n)$ -el 1 maradékot kell kapjunk.

Prímszámok és prímtesztek

Ahogy írtam az RSA egyik erőssége, hogy időigényes nagy prímszámokat tesztelni és előállítani, valamint számokat faktORIZálni. Az időigényes nem azt jelenti itt, hogy 1-2 másodperc alatt, hanem azt, hogy akár évek, évtizedek alatt sem kapunk eredményt.

A prímszámok tesztelésére több prímtesztet is ismerünk, viszont egyetlen biztos prímtesztünk van egy prímszám prímségének meghatározására, az osztogató módszer.

A tesztelendő számot a gyökétől kisebb minden prímszámmal végigosztjuk, és ha minden osztásra nem nullát kapunk maradékkul, akkor a szám prím. Prím sorozatok előállítására is csak egyetlen biztos módszerünk van, az Eratoszthenész szitája

Vannak pszeudó prímtesztek, valószínűségi prímtesztek, amelyek nem tudnak garantált eredményt adni, csak valamilyen valószínűséggel állítják a számról, hogy prím. Ezeknek a prímteszteknek a bizonytalansága a vizsgálatok számától függ, a ciklusszám növelésével az csökken. Ilyen teszt például a Miller-Rabin prímteszt.

Vannak különleges alakú prímszámok, mint például a Mersenne prímek.

A Mersenne prímek alakja $2^n - 1$, tehát bináris számrendszerben csupa egyesekkel leírhatók. A különleges alakú prímszámokra különleges prímtesztek születtek. A Mersenne prímekre determinisztikus prímteszt a Lucas-Lehmer teszt.

A prímtesztek közül a Lucas-Lehmer prímteszt keltette fel legjobban az érdeklődésemet. Igazából az egyszerűsége tetszett meg. Egy négyzetre emelés, egy kivonás és egy maradékos osztás. Mi nehéz lehet ebben? Gondoltam...

Lucas-Lehmer prímteszt

A Lucas-Lehmer prímteszt a speciális alakú Mersenne prímszámokra íródott determinisztikus prímteszt. A Mersenne prímszámok $2^p - 1$ alakú prímszámok, vagyis bináris számrendszerben ábrázolva minden számjegye egyes. További feltétel, hogy p legyen prímszám. A Mersenne számok $2^p - 1$ alakú számok, de nem feltétlen prímek.

A Lucas-Lehmer prímteszt algoritmus:

A függvény bemenő paraméterei a tesztelendő Mersenne szám és a Mersenne szám kitevője. A függvény visszatérési értéke aszerint igaz vagy hamis, hogy a bemenő paraméterként kapott Mersenne szám prímszám vagy nem.

A függvényben szükségünk van egy tömbre, amit majd a ciklus után vizsgálunk. Először is ebbe az 's' tömbbe teszünk 4-et. A ciklus annyiszor fut le, amennyi a kitevő-2. Az s értékét négyzetre emeljük, kivonunk belőle kettőt, majd maradékosan elosztjuk a tesztelendő számmal, vagyis „ $s = ((s \times s) - 2) \bmod \text{number}$ ” Növeljük a ciklusszámlálót. Ezt addig ismételjük, amíg a ciklusszámláló értéke el nem éri a kitevő-2 t. Ennek az algoritmusnak a legidőkritikusabb pontja a négyzetre emelés, ezért kellett nagyon hatékony szorzó algoritmust találni. Ha vége lett a ciklusnak vizsgálni kell az s tömböt. Ha s értéke 0, akkor a számunk prím, ellenben összetett. Az így kapott logikai értékkel visszatérünk a függvényből.

Mivel nagyon nagy számokkal, hosszú egészekkel végzünk műveleteket, sokat és nagyon sokszor, ezért gondoltam, hogy ehhez a feladathoz kevés lesz bármelyik magas szintű programozási nyelv. Az Assembly programozási nyelv erősségeit kihasználva kell megírni ezt az algoritmust, hogy minél hatékonyabb eredményt érjünk el.

Arra volt szükség, hogy tényleg én mondjam meg, hogy a processzor melyik regiszterébe mikor milyen szám kerüljön, én mondjam meg, melyik alacsony szintű utasítással történjen a kívánt művelet. Hozzáféresem legyen a stack-hez és a flagekhez, megmondhassam, hogy hova és mit szeretnék írni a memóriába.

Nem csak az Assembly időbeli hatékonyságát, kis méretét akartam igénybe venni, hanem komoly kihívásnak is éreztem annak használatát. Alacsony szintű nyelven hatványozottabban érződik az, hogy milyen apró finomságokon múlik egy program gyorsasága, vagy tárterület igénye.

Az Assembly programozási nyelv

„A számítógépes problémamegoldás során a kitűzött célt megvalósító algoritmust mindig valamilyen programozási nyelven (programming language) írjuk, kódoljuk le. A nyelvet sokszor az adott feladat alapján választjuk meg, míg máskor aszerint döntünk egy adott nyelv mellett, hogy az hozzánk, az emberi gondolkodáshoz mennyire áll közel. Ez utóbbi tulajdonság alapján csoportosíthatók a számítógépes programozási nyelvek: megkülönböztetünk alacsony szintű (low-level) és magas szintű programozási nyelveket²

Az Assembly tehát egy alacsony szintű programozási nyelv, még hozzá nagyon alacsony szintű, ebből következően pedig sokkal közelebb áll a hardverhez, mint bármely más nyelv.

Főbb jellemzői:

- nagyon egyszerű, elemi műveletek
- típusfüggetlenség
- rögzített utasításkészlet
- világos, egyszerű szintaxis
- kevés vezérlési szerkezet”

Regiszterkészlet:

Általános célú regiszterek:

(E/R)AX : Akkumulátor: Ebbe a regiszterbe kell írni a függvények visszatérési értéket. Szorzás (mul) művelet után ebbe a regiszterbe érkezik a szorzat, pontosabban (e/r)dx:(e/r)ax regiszterbe, ha nem fér el az ax regiszterben.

(E/R)BX : teljesen általános célú regiszter.

(E/R)CX : Általános célú regiszter, de számláló regiszterként is szoktuk használni például ciklusszámlálóként vagy kötelezőként számláló a másolásnál.

(E/R)DX : Általános célú regiszter.

Mutató (pointer) regiszterek:

(E/R)SI : Source Index : Másolásnál a forrás címét kell ebbe a regiszterbe tenni.

(E/R)DI : Destination Index : Másolásnál a cél címét kell ebbe a regiszterbe tenni.

(E/R)IP : Instruction Pointer : Ebben a regiszterben van a következő utasítás címe.

(E/R)SP : Stack Pointer:

(E/R)BP: Bázis Pointer:

A 32 bites és a 64 bites Assembly programozási nyelv összehasonlítása

A 64 bites Assembly nyelven írt programok gyorsabbak, mint 32 bites Assemblyben megírt változatuk. A következő tényeket mondhatjuk a két implementáció közötti futási idők közötti eltérésekről.

- Különbség a szóhosszban:

Lényeges különbség, hogy a 32 bites Assemblyben 32 bites, 4 bájtos az egy szó hossza, amíg 64 bites Assemblyben 64 bites, 8 bájtos egy szó hossza. Dupla hosszúságú szavakkal dolgozik. Egy szóhosszon azt a bitszámot értjük, amekkora legnagyobb bites számot tudunk címezni a memóriában, illetve ekkora a legnagyobb szám, amelyeket egyes regiszterekbe beletehetünk. Ettől a számtól kisebb méretet is tehetünk a regiszterekbe, illetve címezhetünk a memóriában. Ha 64 bit a szám hossza, akkor a neve qword, a 32 bites neve dword. Ha 16 bites a szám, akkor word, ha 8 bites, akkor bájt. A bájtól kisebb értéket nem címezhetünk a memóriában.

- Különbség a függvényhívási konvencióban:

A függvényhívási konvenció azt jelenti, hogy milyen megszokott műveleteket kell elvégezni a függvény paramétereivel a függvényhívás előtt, illetve utána. A 64 bites Assemblynek és a 32 bites Assemblynek különböző a standard függvényhívási konvenciója.

32 bites Assemblyben a függvény paramétereit a stack-be tesszük jobbról balra, vagyis az utolsó paramétert tesszük a stack-be először. Ha a függvény paramétereit beletettük a stack-be, akkor hívhatjuk a függvényt: call fuggveny. A függvényhíváskor tehát nem csak azt a memóriacímet tároljuk a stack-ben, ahova a hívás után vissza kell térni, hanem az összes paramétert. Ezen kívül a stack-ben tároljuk a függvények lokális változóit is, de erre később térek vissza.

64 bites Assemblyben a függvény első négy paraméterét regiszterekbe tesszük, rendre az rcx, rdx, r8, r9 regiszterekbe. Ezek szerint ha a paraméterek száma kevesebb 5-nél, a stack-be nem kerül semmi a paraméterekből. A 4-en felüli paramétereket a stack-be tesszük. A függvény meghívása, a lokális változók kialakítása hasonlóan történik, mint 32 bites Assemblyben.

A 64 bites Assembly és 32 bites Assembly nyelven írt programok futási ideje közötti különbséget a különböző függvényhívási konvenció is befolyásolja. Ugyanis a stackművelet lényegesen lassabb, mint a regiszterművelet. Regiszterből regiszterbe kevesebb időbe telik adatot mozgatni, mint regiszterből memóriába. Mivel a stackművelet memóriaművelet, ezért

lassabb a 32 bites Assembly esetén a paraméterátadás. A 64 bites Assemblyben kiválthatjuk az első négy stack műveletet négy gyors regiszterműveletre a függvényhívási konvencióból adódóan.

```
function sumnum(szam1, szam2, szam3, szam4:UInt64):UInt64;
// rcx - szam1
// rdx - szam2
// r8 - szam3
// r9 - szam4
asm
    mov rax,rcx // beleteszem az első számot az rax-be
    add rax,rdx // hozzáadogatom a számokat az rax-hez
    add rax,r8
    add rax,r9
    // rax - ebben térek vissza az eredménnyel
end;

procedure caller();
asm
    mov rcx,1 // sumnum 1.paraméter
    mov rdx,2 // sumnum 2.paraméter
    mov r8,3 // sumnum 3.paraméter
    mov r9,4 // sumnum 4.paraméter
    call sumnum // sumnum függvény hívása
end;
```

- Különbség a regiszterkészletben:

64 bites Assemblyben sokkal több általános célú regiszter áll rendelkezésünkre, mint 32 bites Assemblyben (plusz általános célú regiszterek: r8, r9, r10, r11, r12, r13, r14). Ha egy regisztert el kellene mentenünk a stack-be, hogy ne romoljon el, helyette egy másik regiszterbe tesszük, azzal megint időigényes stack műveleteket takaríthatunk meg. Nem időbeli tényező, de a programozó munkáját is könnyebbé teszi, hogy nem a stack-be, kell menteni, vagy még több lokális változót létrehozni – amely elérése a továbbiakban szintén időigényes memóriaművelet – helyette csak gyors és kényelmes regiszterműveleteket végzünk, esetleg nem is kell menteni.

LNGINT & RLNGINT

Az LNGINT egy publikált 32 bites Assembly nyelven íródott függvénygyűjtemény a hosszú egész számokkal való műveletvégzésekhez a moduláris aritmetika szabályai szerint.

Azért született meg ez a függvénykönyvtár, mert a prímtesztokban sok és bonyolult számításokra van szükség hosszú egész számokkal.

Az RLNGINT függvénygyűjtemény az LNGINT függvények párja.

A függvénykönyvtárak nevei is utalnak az implementációjukra. 64bites Assemblyben a regiszterek nevei 'r' betűvel kezdődnek, azért kapta ezt a nevet az RLNGINT függvénykönyvtár. Az LNG a long, mint a hosszú rövidítése, az INT pedig hogy egész számokkal végzett műveleteket tartalmaz.

Az rutinok segítségével „hosszú” előjelnélküli egész számokkal lehet műveleteket végezni a moduláris aritmetika szabályai szerint. Az egész értékek hossza 32 bit (Dword hossz), illetve 64 bit (Qword hossz) egész számú többszöröse lehet. A tárolt értékek 32 illetve 64 bites szavakból álló tömbökben helyezkednek el a „LITTLE-ENDIAN” elv szerint, azaz az alacsonyabb helyiértékű részek az alacsonyabb című tárolóban vannak. A műveletekben szereplő „hosszú” érték(ek)et minden esetben a legkisebb helyiértékű elemnek a címével kell megadni.

A 64bites RLNGINT függvénygyűjtemény azért született, mert kíváncsi voltam mik a különbségek a 32bites és a 64bites Assembly nyelv között a gyakorlatban. Másrészt pedig kíváncsi voltam arra, hogy mik a futási időbeli különbségek a két implementáció között. Abban biztos voltam, hogy a 64bites Assembly gyorsabb lesz, de pontosan számszerűen látni szerettem volna a gyorsaság mértékét. Gyakorlati szempontból pedig a hosszú prímszámok tesztelésére születettek a függvények.

A rutinokkal szintén hosszú előjel nélküli számokkal lehet műveleteket végezni. Az értékek hossza 64 többszöröse lehetnek. Itt már nem 32bites, hanem a névből is adódóan 64bites szavakból álló tömbökben elhelyezkedő számokkal dolgozunk.

A Lucas-Lehmer prímteszt és az RLNGINT

A Lucas-Lehmer prímteszt egyik kardinális pontja a szorzás, ezért a szorzás optimalizálására fektettem a legnagyobb hangsúlyt. Első lépésként választani kellett egy viszonylag hatékony szorzó algoritmust. Azért nem az SRSA, Standard Rekurzív Szorzó Algoritmusra esett a választás, mert Virasztó Tamás azt írta a Titkosítás és adatrejtés című könyvében, hogy a Karatsuba-Ofman rekurzió gyorsabb, mint az SRSA. Amikor megszületett az RLNGINT függvénygyűjtemény a függvényeiből megírtam a Lucas-Lehmer prímtesztet. ¹

A Karatsuba-Ofman rekurzió

A Karatsuba-Ofman rekurzív szorzó algoritmus két egész szám szorzatának előállítására szolgáló eljárás. A függvény bemenő paraméterei az eredmény, a szorzandó, a szorzó, és az aktuális hossz.

1. Első lépésben a tényezőket fel kell bontani két-két egyenlő bitnyi hosszúságú részre.

A h az így kiadódó félhossz

$$a=2^h a_1+a_0 \text{ és } b=2^h b_1+b_0$$

2. Az eredményül kapott számokat szorozzuk össze:

$$t_0 = a_0 b_0$$

$$t_2 = a_1 b_1$$

$$t_1 = a_1 b_0 + a_0 b_1 = (a_1 + a_0)(b_1 + b_0) - t_0 - t_2$$

Addig kell rekurzívan elvégezni ezt a felbontást, amíg olyan méretű számot nem kapunk, amit már direkt művelettel össze tudunk szorozni, jelen esetben a szorzását már egyszerűen elvégezhetjük a `mul` utasítással.

```
Function KOSA(a,b)
    t0 := KOSA(a0, b0)
    t2 := KOSA(a1, b1)
    u0 := KOSA(a1 + a0, b1 + b0)
    t1 := u0 - t0 - t2
return ( [t2 << k] + [t1 << h] + t0 )
```

// Virasztó Tamás - Titkosítás és adatrejtés c. könyve

Az algoritmus lépéseinek száma: $O(k)=k^{1.53}$

A rekurzió miatt a stackméret jelentős növekedésére is számítanunk kell. Minden függvényhíváskor a stackben tárolódik a visszatérés helyének memóriacíme és a paraméterek, valamint a függvény lokális változói. A referencia típusú változóknak az értékeit is a stackben kell tárolni függvényhíváskor.

Lucas-Lehmer prímteszt futási idő Assembly(x86) – Assembly(x64)

Összehasonlítottam Lucas-Lehmer teszt futási idejét 32 bites Assemblyben és 64 bites Assemblyben megírva. Biztos voltam benne, hogy a 64 bites Assembly gyorsabb, mint a 32 bites, de a pontos értékekre voltunk kíváncsiak, hogy mennyivel gyorsabb. A sebességkülönbségben összeadódnak a 64 bites Assembly előnyei: a dupla méretű szóhossz, a gyors regiszterműveletes függvényhívási konvenció és a sokkal több általános célú regiszter. Ebben a Lucas-Lehmer prímtesztben a Karatsuba-Ofman rekurzió szorzó algoritmus került implementálásra a szorzás elvégzésére.

A futási időket 1279 és 5000 kitevők közötti számokra néztem. Átlagosan 4-5x gyorsabb a 64 bites Assembly, mint a 32 bites. Legnagyobbbrészt a dupla szóhosszú operandusok miatt. A diagramon a nagy változások azt jelentik, hogy akkor duplázódik meg a számok szóhossza. Ez azért látszódik így, mert dupla szóhosszú számot több időbe telik letesztelni, hogy prím-e, mert dupla annyi rekurzív hívásra van szükség a tényezők hosszának megfelelésekor, ami több részművelettel is jár.

Mérési környezet

A méréseket Intel Core i5 2500K @4GHz processzorral és 8GB RAM-mal végeztem, Windows 8 operációs rendszeren asztali számítógépen vízhűtéssel. A vízhűtésnek a zajcsökkentésén és az érdekességen kívül nem volt más gyakorlati haszna.

A programokat Embarcadero Delphi XE2 környezetben inline Assemblyben írtam. Ebben a környezetben nagyon kényelmes a fejlesztés és debugolás. Látom a regiszterek, a memória, a stack és a flagek tartalmát lépésenként futtatáskor.

A teszteket és a méréseket 1 szála írtam. A mérésekkor érzékenységgel is mértem, ami alatt azt értem, hogy egy mérést 5000x futtattam le, ezáltal század részére csökkentve a zavaró tényezőket. A mérések indítása előtt a taszkok prioritását magasnak kell jelölni, hogy kevesebb zavaró tényező szólhasson közbe, ne vegye el más művelet a processzoridőt.

A néhány mérést thread-ben kellett indítani, mert meg kellett növelni a stackméretet és az így volt könnyen lehetséges. A Karatsuba-Ofman rekurzió extra stack méretet igényelt az alapértelmezett 1MB-on felül. Azért kell ilyen sok hely a stack-ben, mert nem csak a függvényhívások memóriacímei, de ugye a függvények nem kisméretű paraméterei is ide kerülnek.

A hagyományos általános iskolás írásbeli szorzás tízes számrendszerben

A hagyományos általános iskolás írásbeli szorzást általános iskolás korunkban tanultuk, melyet nevezhetünk eltol-összead módszernek is.

Amikor tanultuk ezt az algoritmust, általában először tízes számrendszerben végeztük a műveleteket. Sorra vettük a szorzó számjegyeit. A szorzó minden számjegyével végigszoroztuk az eredeti szorzandót, majd az eredmény után írunk egy nullát. Mondhatnánk úgy is, hogy eltoljuk eggyel balra, ami tízes számrendszerben tízzel való szorzást jelent. Ha egyes van a szorzóban, akkor nem kell végigszorozni a szorzandót a szorzó aktuális számjegyével, mert az eggyel való szorzás a szorzandót adja vissza eredményül. Ebben az esetben, csak össze kell adni a részeredményt, az eredmény területén levő a számot az eredeti szorzandóval. Ha nulla a szorzó aktuális számjegye, akkor nem kell szorozni, csak a részeredmény után nullát írni. Ha sem nem nulla, sem nem egyes a szorzónk aktuális számjegye, akkor más az eljárás. A szorzó aktuális számjegyével végig kell szorozni az eredeti szorzandót. Itt kezelni kell a számjegyek között az átvitelt is. Átvitel az a szám, amely akkor keletkezik, ha összeszorozunk kettő egyjegyű számot és a szorzatban a legkisebb helyiértékű szám felett keletkezik, vagyis a legnagyobb helyiértékű szám a szorzatban. Mivel itt egyjegyű számokat szorzunk össze, az átvitel is maximum egy jegyű lehet. Ha elfogytak a szorzó számjegyei, akkor megkapjuk az eredeti szorzandó és szorzó szorzatainak részeredményeit, amelyeket össze is kell adni. Tízes számrendszerben így néz ki a hagyományos szorzó algoritmus.

A hagyományos általános iskolás írásbeli szorzás kettes számrendszerben

A szorzandót és a szorzót bináris számrendszerben látjuk, és az eredményt is kettes számrendszerben várjuk.

A szorzandót leírjuk az eredmény területére. Ezzel elvégeztük a szorzó legfelső helyiértékén levő egyessel való szorzást. Sorra vesszük a szorzó minden számjegyét a második jegytől kezdve. A szorzó minden számjegyén a következő műveleteket végezzük el: Az eredmény területén levő számot eltoljuk balra egy helyiértékkel, vagyis kettes számrendszerben megszorozzuk kettővel. Meg kell vizsgálni a szorzó aktuális jegyét, hogy nulla vagy egy. Ha egyes, akkor az eredeti szorzandót hozzáadjuk az eredmény területén számhoz. Ha nulla, akkor nincs szükség összeadásra, vehetjük a szorzó következő számjegyét. Addig kell ismételnünk ezt a két lépést, amíg a szorzónak vannak érintetlen jegyei. Ha már nincsenek jegyei a szorzónak, akkor az eredmény területén megkaptuk az eredeti szorzandó és szorzó szorzatát.

Bináris számrendszerben lényegesen egyszerűbb dolgunk van, mint tízes számrendszerben. Ami azt is jelenti, hogy az algoritmusunk sokkal egyszerűbb, hatékonyabb, időben gyorsabb. Nem kell végigszoroznunk minden szorzóbeli számmal a szorzandót. Pontosabban, végig kell szoroznunk, csak gyakorlatban nem, mert az eggyel való szorzás önmagát adja eredményül és mivel bináris számrendszerben a nullán kívül csak egyes számjegy létezik, ezért csak eggyel való szorzásaink lehetnek.

$$\begin{array}{r} 10101100 \text{ X } 101101 \\ 00000000 \\ 10101100 \\ 10101100 \\ 00000000 \\ 10101100 \\ + \\ \hline 111100011100 \end{array}$$

A hagyományos általános iskolás szorzó algoritmus implementálása bináris számrendszerben, 64 bites Assembly programozási nyelven

Az algoritmus implementálásnál a függvény neve oldmul lett, de néhány segédfüggvényre is szükségem volt. Azért nem egyetlen függvényben írtam meg, mert több függvénnyel logikailag áttekinthetőbb és a hatékonyság növelése érdekében nem kellett foglalkoznom a regiszterek tartalmának stack-be való mentésével az elrontás miatt.

Az oldmul függvénynek három bemenő és egy kimenő paramétere, valamint logikai típusú visszatérési értéke van. Paraméterei: eredme: kimenő paraméter, PUInt64 típusú. Ez a pointer az eredmény memória területére mutat. A bemenő paraméterek nem változnak. A következő két paramétere, a szorza és a szorzo a szorzandó memória területére, illetve a szorzó memória területére mutat a memóriában. Lehet, hogy kényelmesebb lenne a szorzó algoritmus, ha hagynám megváltozni ezeket a memóriaterületeket, de ez azért nem jó, mert további problémákat kellene kezelnem. Például nem lehetne egyszerűen ismételt szorzást végezni. A következő paramétere a lenakt ami a tényezők hosszát jelenti szóhosszban. Ha tegyük fel, összeszorozunk egy 4 szóhosszú és egy 2 szóhosszú számot, akkor az eredmény maximum 6 szóhosszú lehet, vagyis a lenaknak 6-ot kell adni.

Delphiből az alábbi módon kell meghívni ezt a függvényt:

```
oldmul(@eredme, @szando, @szorzo, lenakt);
```

Amikor Delphiből, vagy esetleg más magasszintű programozási nyelvből hívjuk ezt a függvényt, nem kell foglalkoznunk a függvényhívási konvencióval, mert azt a Delphi elintézi, de nem árt, ha számítunk arra, mi történik a háttérben.

Természetesen nem az első ötletem lett a leggyorsabb megoldás, hanem a harmadik. Először is a legjobban sikerült szorzást mutatnám be, majd később az első kettő változatról is fogok írni, hogy milyen gondolatokon keresztül jutottam el ehhez a megoldáshoz.

```
push rbp      // Elmenteni a bázispointert
mov  rbp, rsp  // Egyenlővé tenni a bázispointert és a stackpointer
```

A függvényben el kell mentenünk a függvényben használt regisztereket, hogy ne romoljanak el. Ez azért fontos, mert ha egy másik programból alprogramként hívjuk a függvényt, akkor a visszatérés után a használt regiszterekben nem a függvényhívás előtti értékek lesznek, ami meglepetésként érheti a függvényhívót, ha nem számít rá. Igazából nem is kell erre számítani, mert a függvény dolga, hogy elmentse a használt regisztereit, a hívónak ez nem feladata.

```
push r10      // Elmentem az r10 regisztert
push r11      // Elmentem az r11 regisztert
push r12      // Elmentem az r12 regisztert
push r13      // Elmentem az r13 regisztert
push r14      // Elmentem az r14 regisztert
push r15      // Elmentem az r15 regisztert
push rbx      // Elmentem az rbx regisztert
push rsi      // Elmentem az rsi regisztert
push rax      // Elmentem az rax regisztert
```

A függvényekben ezután az r10, r11, r12, r13 regiszterekbe elteszem a függvény első négy paraméterét, már ha van annyi. Ha több, mint négy paraméterem van, akkor lokális változóként teszem a memóriába, hogy ha meg is változtatom az értékét, ne romoljon el az eredeti paraméter értéke. De most csak négy paraméterünk van, ezért erre a részre nem kell figyelni.

```
mov  r10, rcx  // Az eredmény terület címe
mov  r11, rdx  // A szorzandó terület címe
mov  r12, r8   // A szorzó terület címe
mov  r13, r9   // A lenakt (szóhosszak) értéke
```

Nullázni kell az eredmény memóriaterületét.

```
mov    rdi,r10    // Másolat a cél címéről
mov    rcx,r13    // Ennyi hosszan nullázok
xor    rax,rax    // Ezzel fogok nullázni
rep    stosq      // Eredmény helyének nullázása
```

Mivel szorzóalgoritmusról van szó, nagyon fontos a nullavizsgálat. Megvizsgálom, hogy a szorzás tényezői közül valamelyik nem-e nulla esetleg. Ugyanis, ha valamelyik nulla, akkor nincs értelme a szorzásnak. Ez esetben ugrok a függvény végére, az eredmény memóriaterületén pedig a nullázás utáni nullák lesznek, vagyis nullát kapunk eredményül.

Kétféle nullavizsgálat rutint hívok meg, különbözöt a szorzandóra, illetve a szorzóra. Nem azért, mert különbség van a két szám nullavizsgálatának elvégzésében, hanem azért, mert a szorzó nullavizsgálatának rutinjába más műveleteket is beleírtam az időhatékonyság növelésének érdekében. Igazából nem kellett volna erre a részre külön rutint írnom, viszont logikailag áttekinthetőbb, másrészt ez a rutin nagyon kiemelendő része a szorzó algoritmusnak.

Nullavizsgálat a szorzandóra függvény meghívása:

```
mov    rcx,r11    // Szorzandó címe
mov    rdx,r13    // lenakt értéke
call   bitnum2    // Meghívom a szorzandót nullavizsgáló függvényt
or     rax,rax    // Nulla-e a bitszám?
je     @vegeez    //ha igen, vége. már nulláztam az eredmény területét
```

A szorzandó nullavizsgáló függvénye tényleg nem csinál más nullavizsgálaton kívül. Viszont a szorzó nullavizsgáló függvénye ennél egy kicsit több. Az r14 regiszterben visszaadja, hogy a szorzandó mennyi szóból áll. Az r15 regiszterben visszaadja a legfelső értékes szó bitszámát. Visszatérési értéke pedig a szorzó teljes bitszáma.

Nullavizsgálat a szorzóra függvény meghívása:

```
mov    rcx,r12        // Szorzó címe
mov    rdx,r13        // Lenakt értéke
xor     r14,r14        // Ide várom az offszetet
xor     r15,r15        // Ide várom a bitszámot
call   nuvizl_bitnum   // Nullavizsgálat, offszetvárás, bitszámvárás
or     rax,rax         // Nulla-e a bitszám?
je     @vegeez        // Na nulla, akkor az eredmény is nulla
```

Másolatot készítünk a szorzandóról az eredmény memóriaterület helyére. Ez azért szükséges, mert ezzel a lépéssel az első szorzóciklust, vagyis az első eggyel való szorzást – mert a szorzó első bitje mindig egy – kiválthatjuk. Ezzel eggyel kevesebbszer kell végrehajtani majd a szorzó ciklust. De ha belegondolunk, amikor papíron végezzük az írásbeli szorzást, és a szorzó legfelső helyiértékén egyes van, akkor sem végezzük el az eggyel való szorzást.

```
mov    rsi,r11        // A szorzandó címe
mov    rdi,r10        // Az eredmény terület címe
mov    rcx,r13        // Ennyi hosszan másolok
rep    movsq          // Elvégzem a másolást
```

Másolatot készítek a szorzóról, hogy ne az eredetit rontsam el. Mivel majd műveletet kell végezni a szorzóval, ezért szükséges más memóriaterületen módosítani a szorzót.

```
mov    rsi,r11        // A szorzandó címe
mov    rdi,r10        // Az eredmény terület címe
mov    rcx,r13        // Ennyi hosszan másolok
rep    movsq          // Elvégzem a másolást
```

Kialakítom az első vizsgálót

Az rbx regiszterben lesz a vizsgáló szám. Ezzel vizsgálom, hogy a kért helyiértéken 0 vagy van a szorzó számban.

```
mov    rsi, rax    // A szorzó bitszáma
mov    rbx, 1      // Ezt kell eltolni
mov    rcx, r15     // Ennyivel kell eltolni
@leptet:
shl     rbx, 1      // Eltolom
dec     rcx         // Csökkentem a ciklusváltozót
jnz     @leptet     // Kell még léptetni?
```

A szorzó nullavizsgáló függvényéből megtudtam, hogy hány bitből áll a szorzó. Ez a szám azért szükséges, mert ennyiszor mínusz egyszer kell elvégezni a szorzóciklust. Azért mínusz egy, mert az első szorzás minden esetben eggyel való szorzás, de azt igazából nem kell elvégeznünk

Meg kellett számolnom azt is, hogy hány szóból áll a szorzó. Ezt a számot szintén a nullavizsgáló függvényből tudtam meg. Ez a szám majd a szorzó felléptetéséhez fog kelleni.

A szorzóciklusban kiveszem regiszterbe a memóriából a vizsgálandó szót a szorzóból. Azt kell vizsgálni, hogy a megadott helyiértéken nulla vagy egy van. Ha egy van, akkor kell összeadni is, az eredmény területén levő számot az eredeti szorzandóval, egyébként ha nulla, akkor nem kell ezt az összeadást elvégezni. Csökkenteni kell a vizsgáló helyiértékszámát, majd kialakítani a vizgálót léptetéssel. A vizsgáló helyiértékszámát azt a számot jelenti, hogy a szorzó legfelső értékes szavában hol van a legfelső értékes bit. Ha nulla a vizsgáló, akkor csökkenteni kell az offset értéket. Ez az offset a vizsgálandó szóra mutat a szorzóban.

```

@ciklus:
    mov    rax,[r12+(r14*8)-8]    //kiveszem a vizsgálandót
    and    rax,rbx                // rbx-el vizsgálók
    jnz    @sumnum               // Ha nem nulla lett, akkor összeadok
    mov    rcx,r10                // Nem kell az összeadás, csak elolom az eredme
területén lebő számot
    mov    rdx,r10                // Az eredmény terület címe
    mov    r8,r13                 // lenakt értéke
    call   shllon                 // Kettővel szorzó függvény hívása
    jmp    @cfelte               // Ugorj a ciklusfeltétel vizsgálatához
@sumnum:
    mov    rcx,r10                // Az eredmény terület címe
    mov    rdx,r10                // Az eredmény terület címe
    mov    r8,r13                 // lenakt értéke
    call   shllon                 // Felfelé léptetem az eredmény területét
    mov    rcx,r10                // Az eredmény terület címe
    mov    rdx,r10                // Az eredmény terület címe
    mov    r8,r11                 // A szorzandó terület címe
    mov    r9,r13                 // lenakt értéke
    call   addlon                 // Összeadás hívása
@cfelte:
    shr    rbx,1                 // Léptetem a vizsgálót
    jz     @valtas               // Ha nulla lett, akkor csökkenteni kell az
offszetet és változik a vizsgáló
    dec    rsi                   // Csökkentem a szorzó bitszámát
    jnz    @ciklus               // Ha nem nulla, akkor újra ciklus
    jmp    @vegeez               // Vége a ciklusnak

```

```

@valtas:
    dec    r14        //csökkentem az offszetet
    rcr    rbx,1      //ez lesz az új vizsgáló (rbx := $8000000000000000)
    dec    rsi        //csökkentem a szorzó bitszámát
    jnz    @ciklus    //ha nem nulla, akkor még kell ciklus

@vegeez:           // Visszaveszem a használt regisztereket
    pop    rax        // Visszaveszem az rax regisztert
    pop    rsi        // Visszaveszem az rsi regisztert
    pop    rbx        // Visszaveszem az rbx regisztert
    pop    r15        // Visszaveszem az r15 regisztert
    pop    r14        // Visszaveszem az r14 regisztert
    pop    r13        // Visszaveszem az r13 regisztert
    pop    r12        // Visszaveszem az r12 regisztert
    pop    r11        // Visszaveszem az r11 regisztert
    pop    r10        // Visszaveszem az r10 regisztert

    mov    rsp,rbp    // A bázispointert beleteszem a stackpointerbe
    pop    rbp        // Visszaveszem a bázispointert

```

A hagyományos szorzó algoritmushoz felhasznált függvények:

addlon: Az addlon függvényre akkor van szükségem, amikor a szorzóban egyes bit következik a szorzó ciklus során és az eredeti szorzandót hozzá kell adnom a részeredményekhez.

```

push    r10
    xor    r10,r10                // Offszet <- 0, CY <- 0
    @adccik:
        mov    rax,[ossze1 + (r10 * 8)]    // Az egyik összeadandó címe
        adc    rax,[ossze2 + (r10 * 8)]    // A másik összeadandó címe
        mov    [eredme + (r10 * 8)],rax    // Az eredmény címe
        inc    r10                  // Következő offszet
        dec    lenakt               // Csökkentem a cilusváltozót
        jnz    @adccik             // Ciklusban összeadok
        pop    r10

```

Az **shllon** függvény, vagyis a balra tolás egy helyiértékkel, az írásbeli szorzás egyik alapja az összeadáson kívül. Tíz-es számrendszerben való szorzáskor az eltolás tízzel való szorzást jelent, bináris számrendszerben pedig kettővel való szorzást. A kettővel való szorzás/osztás, vagyis az eltolás jobbra, illetve balra alacsony szinten nagyon gyorsan elvégezhető művelet.

```
push r10
xor r10,r10
@fefeci:
    mov rax,[rdx + (r10 * 8)] // Ez a léptetendő
    rcl rax,1 // Léptetem
    mov [rcx + (r10 * 8)],rax // Kiteszem
    inc r10 // Offset növelése
    dec lenakt // Csökkentem a ciklusváltozót
    jnz @fefeci // Léptetgetek a ciklusban
pop r10
```

bitnum: Ez a függvény azért szükséges, mert meg kell számolni, hány bites a szorzandó.

```
mov rax,erteke // A cél címe
mov rcx,lenakt // Ennyi hosszan vizsgállok
xor rdx,rdx // Kinullázom, mert ide szorzat jön
imul rdx,rcx,64 // RDX <- bitszám
@atmaos:
    cmp qword ptr [rax + (rcx * 8) - 8],0 //Nulla vizsgálathoz
    jnz @kihnuv // Nem is nulla a szám
    sub rdx // Csökkentem a ciklusszámot
    jnz @atmaos // Az osztandót átmásoltam
    jmp @minuva // Kész, nincs bit sem

@kihnuv:
    bsr rcx,[rax + (rcx * 8) - 8] // RDX- sorszám
    sub rcx,63 // RDX <- negatív bitszám
    add rdx,rcx // Bitszám véglegesítés
@minuva:
    mov rax,rdx // Bitszám átadva
```

A **nuvizr_bitnum** a nuvizr és a bitnum függvény ellátja azokat a feladatit, amiket külön-külön kellene elvégeznie és még egy kicsivel több feladatot adtam neki. Az eredeti nuvizr függvény azt mondja meg a számról, hogy nulla-e vagy sem. Az eredeti bitnum függvénnyel pedig arról kapunk információt, hogy a szám hány bitből áll. Külön-külön meghívva a függvényt sok időt jelentene, ugyanis kétszer kéne végigvizsgálni a vizsgálandó számot. Ráadásul ezen a két információn kívül még mást is meg kell tudnunk a számról, például azt, hogy hány szóból áll.

Ezt a függvényt igazából nem kellett volna kiemelni az oldmul rutinból, de azért tettem így, mert kardinális pontja az algoritmusnak és így logikailag jól elkülöníthető.

A függvény bemenő paraméterei a vizsgálandó szám és a vizsgálandó szám szóhossza. A függvény visszatérési értéke a szám bitszáma. Ezen kívül még van két „kimenő” paraméterünk, de az nem a magasszintű programozásban megszokott módokon. Az r14 regiszterben a szószámot, az r15 regiszterben pedig a legfelső értékes szó bitszámát kapjuk meg. Ezt úgy csináltam, hogy ezt a két regisztert nem mentem el függvényhíváskor, illetve nem veszem vissza a függvény visszatérés végén. Viszont függvényhívás előtt és után erre a két regiszter értékére számítani kell.

Azért szabad ilyet tenni a függvények között, mert egyrészt Assemblyben lehet ilyet, másrészt pedig mind a hívó és a hívott oldalon számítani kell ennek a két regiszternek a megváltozására. Ha ilyen tulajdonságú függvényünk van és függvénykönyvtárban használjuk, akkor ezt a tulajdonságát mindenképp fel kell tüntetni, vagy csak úgy használni, hogy az „kívülről” nem elérhető, vagyis más publikus függvényből meghívva.

```

push  rbp
mov   rbp, rsp

push  r10
push  r11

mov   r10, rcx //a vizsgálandó szám
mov   r13, rdx //lenakt

    mov   rax, r10      //a vizsgálandó szám
    mov   rcx, r13      //lenakt
    //xor   rdx, rdx     //törlöm az rdx-et, mert majd ide szorzok
    imul  rdx, rcx, 64   //ennyi bites a szám összesen

    xor   r14, r14

@atmaos:
    cmp   [rax+(rcx*8)-8], r14 //nulla-e az aktuális szó?
    jnz   @kihnuv        //nem nulla
    sub   rdx, 64        //csökkentem a bitszámlálót
    dec   rcx            //csökkentem a ciklusváltozót
    jnz   @atmaos        //van még szó
    jmp   @vegeez        //kész, nincs több szó, mint nulla
@kihnuv:
    //sub   r13, rcx
    //mov   r14, r13      //kiteszem az offset-et
    mov   r14, rcx        //r14-be a ciklusváltozó
    bsr   rcx, [rax+(rcx*8)-8] //bitszámvizsgálat
    sub   rdx, 64        //csökkentem 1 qword-nyi bittel
    add   rdx, rcx        //bitszám összesen
    mov   r15, rcx        //kiteszem a legfelső értékes szó bitszámát

    dec   r15            //csökkentem a bitszámot
    jnz   @minuva        //ha nem nulla lett, akkor ugorjon
    dec   r14            //csökkentem az offset-et
    mov   r15, 63        //ez lett a bitszám

```

@minuva:

```
    cmp    r15,$FFFFFFFFFFFFFFFF    //ha nulla, vagyis nincs is értékes  
bitje a legfelső szónak
```

```
    jne    @vegeez    //ha a legfelső nem nulla
```

```
    mov    r15,63    //ez lett a bitszám
```

```
    dec    r14    //csökkentem az offsetet
```

@vegeez:

```
    mov    rax,rdx
```

```
    pop    r11
```

```
    pop    r10
```

```
    mov    rsp,rbp
```

```
    pop    rbp
```

Hagyományos szorzó algoritmus megírásának története

Elhittem, hogy a Karatsuba Ofman szorzó algoritmus nagyon nagyon gyors. Viszont kíváncsi voltam arra, hogy ha nagyon gyors a KOSA, akkor mennyire lehet lassú a hagyományos általános iskolás szorzó algoritmus. A saját szememmel látni szerettem volna ezt a különbséget számokkal alátámasztva grafikonon ábrázolva. Nagy meglepetésünkre ennek pont az ellenkezője igazolódott. 😊

Amikor elkezdtem a méréseket furcsának találtam, úgy érzésre is, hogy a mullon (Karatsuba-Ofman rekurzív rutinja) sokkal tovább fut, mint az oldmul (Hagyományos szorzó algoritmus rutinja). Amikor megnyitottam a txt fájlt, amibe írtam a futási időket, akkor tárult a szemem elé az óriási időkülönbség. Ahogy lefelé görgettem, a nagyobb szóhosszú számok szorzási ideje felé, egyre nagyobb különbségek voltak a mullon és az oldmul futási ideje között.

Gyorsan grafikonon is látni akartam az időbeli különbséget. Az arány nagyjából már feltételezhető volt, de a görbék alakját még nem tudtam megsejteni. Ábrázoltam közös grafikonon a mullon és az oldmul futási idejét. A mullon futási ideje gyorsan emelkedve az egekbe csapott, míg az oldmul futási ideje alig emelkedett el az x tengelytől. 🙄

Hihetetlennek tűnt, hogy helyes az algoritmus ekkora futási idő különbséggel, tehát ellenőriznem kellett nagyon nagy számokra is az algoritmus helyességét. Ugyanazokkal a véletlenszámokkal feltöltöttem a mullon és az oldmul szorzandójához, illetve szorozójához tartozó tömböt és 100-tól 1000 szóhosszú véletlenszámokra ellenőriztem az algoritmust. Szerencsére minden lefutás után ugyanazokat a számokat adta szorzatul ugyanazokra a tényezőkre. Minél nagyobb szóhosszú tényezőkre mértem a hagyományos szorzó algoritmus futási idejét, annál gyorsabb volt. 🙌

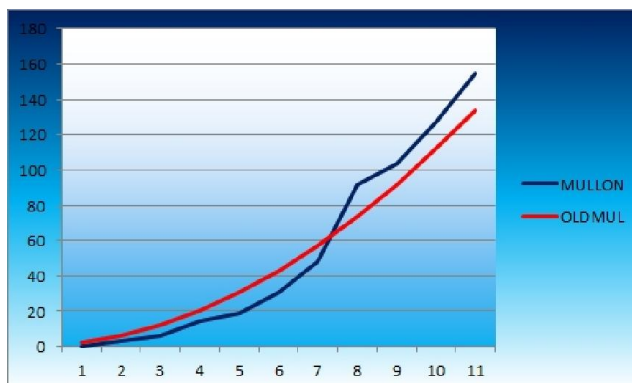
Ahogy már említettem, nem ez volt az első algoritmus amit kitaláltam a hagyományos szorzás minél hatékonyabb implementálására. Az első algoritmus is sokkal gyorsabb volt, mint a Karatsuba-Ofman rekurzió, de nem álltam meg ennél a megoldásnál, mert jöttek újabb, még hatékonyabb ötletek.

Az első szorzás abban különbözött az utolsótól, hogy a szorzót felléptettem balra, amíg a tömb első elemének legfelső helyiértékére nem került a szám legfelső értékes bitje. A szorzó ciklusban az összeadás elvégzésének feltételét úgy vizsgáltam, hogy megnéztem a szorzó legfelső bitjét, hogy nulla vagy egy. Ha nulla, akkor nem kell összeadnom, ha egy, akkor kell összeadnom. Azért toltam fel a legfelső helyiértékre a szorzó számjegyeit, hogy mindig a tömb 0. értékét kelljen vizsgálnom.

A második szorzás az első szorzásra épül, annyi különbséggel, hogy nem bitenkénti shifteléssel alakítom ki a szorzó ciklus előtt a vizsgálandó szorzót, hanem memóriaterület másolással. Ez úgy történik, hogy forráscímként megadom a szorzó legfelső értékes bitjét tartalmazó címet, célcímként pedig a szorzó másolatát tartalmazó tömb legelső helyiértékre mutató címet. Tudom kell, hogy hány szóból áll a szorzó, mert ennyi hosszan másolok. Mivel itt nem áthelyezésről, hanem másolásról van szó, ezért gondoskodnom kell a törlésről is.

A második szorzás az első szorzáshoz képest átlagosan kb 1,5x volt gyorsabb, a harmadik szorzás a másodikhoz képest átlagosan kb. 50%-al volt gyorsabb.

A Karatsuba-Ofman rekurzió hatékonyságát azzal is szokták növelni, hogy egy bizonyos szóhossz alatt áttérnek a hagyományos szorzó algoritmusra. Mint tudjuk, csak egy bizonyos szóhossz után hatékonyabb a Karatsuba-Ofman rekurzió, a leírások szerint. A hagyományos szorzó algoritmus 10 szóhossz felett volt minden tesztelt esetben gyorsabb, mint a Karatsuba-Ofman rekurzió. Érzékenységi teszteléssel mértem, hogy mérhető értéket kapjunk.



Gondoltam, az is érdekes eredményt adhat, ha különleges számokra futtatva mérem le a szorzó algoritmus futási idejét. Különleges számok alatt olyan számokat értek, hogy a bitjei között valamiféle szabályszerűség fedezhető fel. Először is minden szó csupa egyesből állt a szorzandóban és a szorzóban egyaránt bináris számrendszerben. Ahogy vártam itt is magasról verte a hagyományos szorzó algoritmus a Karatsuba-Ofman rekurziót. 100 szóhosszra 20x volt gyorsabb a hagyományos szorzó algoritmus, 1040 szóhosszra pedig 141x volt gyorsabb.



Következőleg olyan számra végeztem el a szorzás mérését, amelyben minden második bit egyes, illetve nulla volt a szorzandóban és a szorzóban is. Ezzel azt akartam megnézni, hogy változik a futási idő, ha az értékes bitek száma felére csökken a tényezőkben. Ahogy vártam, kevesebb volt a futási idő, ha a felére csökkent az értékes bitek száma a tényezőkben. Ez azért fontos vizsgálási pont, mert az írásbeli szorzásban van éles kritérium a tényezők értékes bitjeinek számára. Minél több értékes bit van a szorzóban, annál lassabb lesz a szorzás, mert annál több összeadást kell elvégezni az algoritmusban.

A másik meglepetés akkor ért, amikor megláttam a Karatsuba-Ofman rekurzió futási idejének görbéjét olyan tényezőkre amelyekben minden számjegy egyes bináris számrendszerben, illetve minden második bitje egyes. Arra ugye számítottam, hogy az oldmul futási idejének görbéje nagyon más lesz a különböző tényezőkre, de arra nem számítottam, hogy a mullon görbéi ennyire mások lesznek. Ugyanis a mullonban nincs ekkora szigorú kritérium az értékes bitek számára, mint az oldmulban.

A Lucas-Lehmer prímtesztben kicseréltem a Karatsuba-Ofman rekurziót a hagyományos szorzó algoritmusra. Kíváncsi voltam, hogy ha külön csak a szorzást mérve ekkora az időbeli különbség a két algoritmus között, akkor milyen eredményt kapok. Azt vártam, hogy az oldmullal gyorsabb lesz, mint a mullon és ezt is kaptam

A sok mérést 1279 és 5000 kitevők közötti Mersenne számokra végeztem. A $2^{1279}-1$ Mersenne szám tesztelésének futási ideje a Karatsuba-Ofman rekurzióval 1,53x volt lassabb, mint a hagyományos szorzó algoritmussal. $2^{4959}-1$ tesztelése 3,66x volt gyorsabb az oldmullal, mint a mullonnal. Egyértelműen látható a növekvő tendencia.

A görbék alakjait megvizsgálva, itt szépen látható a hagyományos szorzó algoritmus görbéje lineáris, a Karatsuba-Ofman rekurzióé pedig nem lineáris. Minél nagyobb Mersenne számot tesztelünk, annál hatékonyabb lesz az oldmul használata, mint a mullon.

Kutatásom eredményei:

- Átlagosan 4-5x gyorsabb a 64 bites Assemblyben írt Lucas-Lehmer prímteszt, mint 32 bites Assemblyben, 1279 és 5000 kitevők között. Minél nagyobb a kitevő, annál gyorsabb a 64 bites változata. Legnagyobbrészt a dupla szóhosszú operandusok miatt. Kiemelve néhány mérést: 1279 kitevőnél 4,72x-es, 4959-es kitevőnél 5,34x-es volt a sebességkülönbség a két implementáció között.
- A hagyományos szorzó algoritmus sokkal gyorsabb, mint a Karatsuba-Ofman rekurzó 64 bites Assembly programozási nyelven implementálva.
A tapasztalatok szerint nem igaz, hogy a hagyományos algoritmus $O(t^2)$ alatt fut, hanem lineáris, ami további vizsgálatokat igényel még.³
300 szóhosszra 35x volt gyorsabb a hagyományos szorzó algoritmus, mint a Karatsuba-Ofman rekurzó; 1000 szóhosszra pedig 141x volt gyorsabb a hagyományos szorzó algoritmus. Görbén ábrázolva a hagyományos szorzó algoritmus görbéje alig emelkedik el az x tengelytől, a Karatsuba-Ofman szorzó algoritmus görbéje hirtelen növekedve az egekbe emelkedik.
- A hagyományos szorzó algoritmussal a Lucas Lehmer prímteszt 1279-es kitevőre 1,53x volt gyorsabb, mint a Karatsuba-Ofman rekurzió, 4959-es kitevőnél pedig 3,66x a sebességkülönbség.

Mullon & Oldmul 64bit0xFFFFFFFFFFFFFFFF

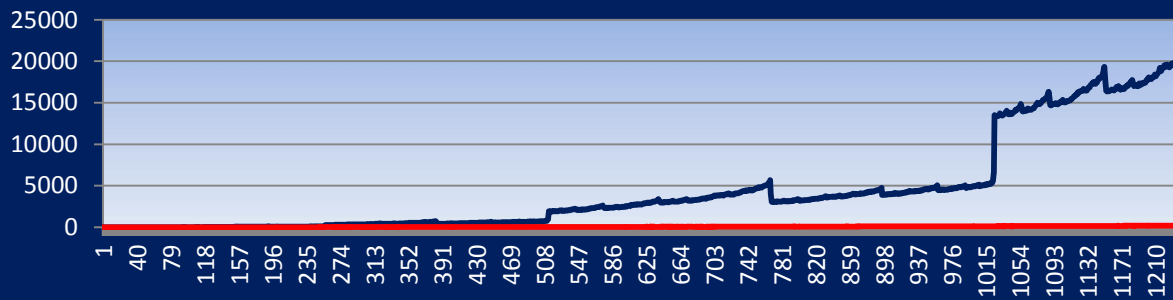


MULLON

OLDMUL

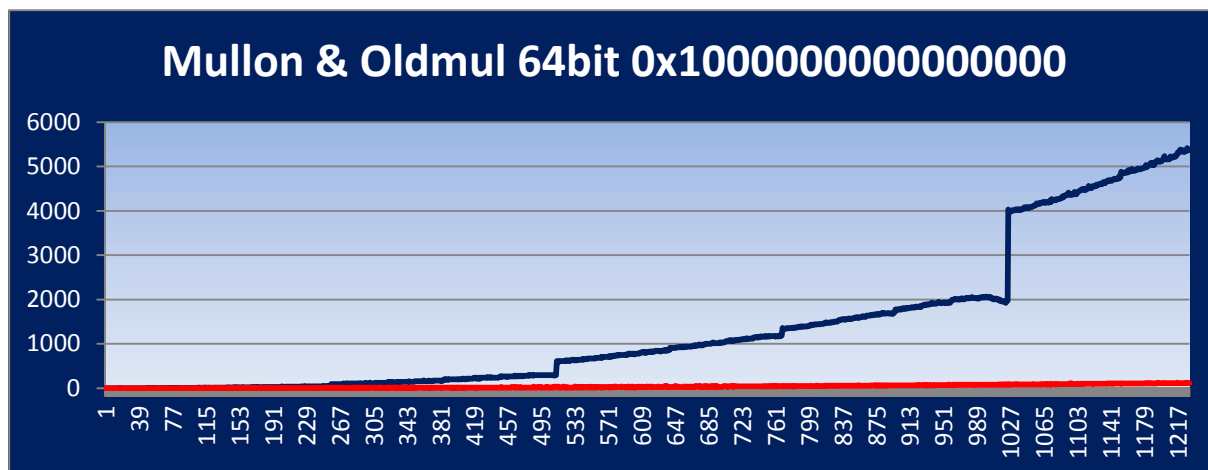
	0xFFFFFFFFFFFFFFFF		
szóhossz	MULLON	OLDMUL	arány
100	20	0	0,00
101	20	0	0,00
200	160	10	16,00
201	155	5	31,00
300	562	16	35,13
301	563	16	35,19
400	1266	31	40,84
401	1281	31	41,32
500	2797	31	90,23
501	2797	47	59,51
600	4688	63	74,41
601	4719	63	74,90
700	6625	78	84,94
701	6673	78	85,55
800	10619	106	100,18
801	10646	105	101,39
900	15509	135	114,88
901	15568	135	115,32
1000	23419	166	141,08
1001	23332	165	141,41
1100	33502	203	165,03
1101	33237	203	163,73
1200	38264	236	162,14
1201	38311	240	159,63
1300	46303	283	163,61
1301	46380	284	163,31
1349	49809	307	162,24
1350	49499	307	161,23

Mullon & Oldmul 64bit 0xAAAAAAAAAAAAAAAA



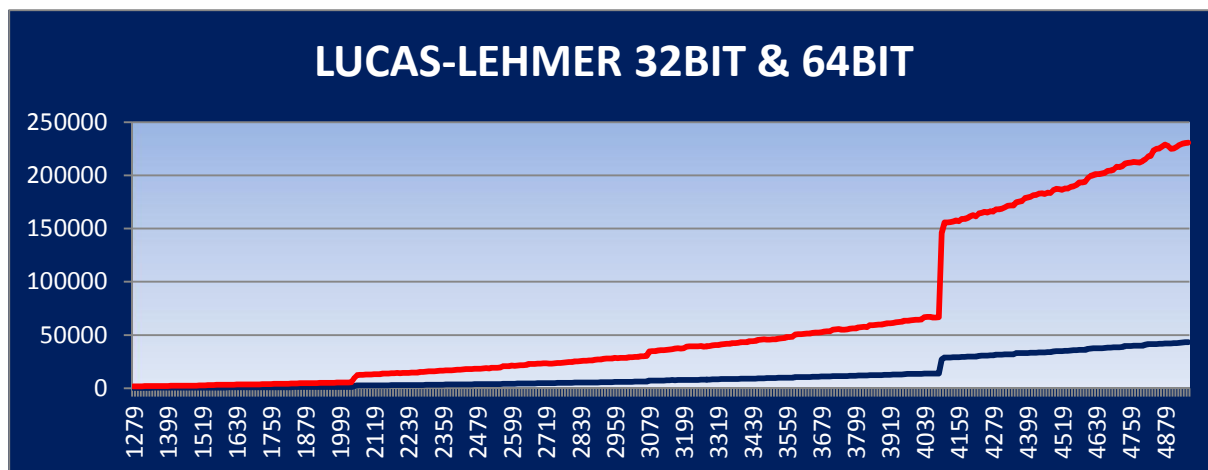
MULLON
OLDMUL

	0xAAAAAAAAAAAAAAAA		
szóhossz	MULLON	OLDMUL	arány
100	10	5	2,00
101	10	0	0,00
200	65	5	13,00
201	65	5	13,00
300	344	16	21,50
301	344	15	22,93
400	454	15	30,27
401	454	31	14,65
500	687	31	22,16
501	704	31	22,71
600	2469	47	52,53
601	2515	47	53,51
700	3641	62	58,73
701	3672	47	78,13
800	3260	74	44,05
801	3216	78	41,23
900	3945	100	39,45
901	3940	95	41,47
1000	4930	122	40,41
1001	4945	118	41,91
1100	15064	156	96,56
1101	15048	156	96,46
1200	17673	176	100,41
1201	17849	173	103,17
1300	21643	207	104,56
1301	21649	207	104,58
1349	23210	224	103,62
1350	23124	225	102,77



MULLON
OLDMUL

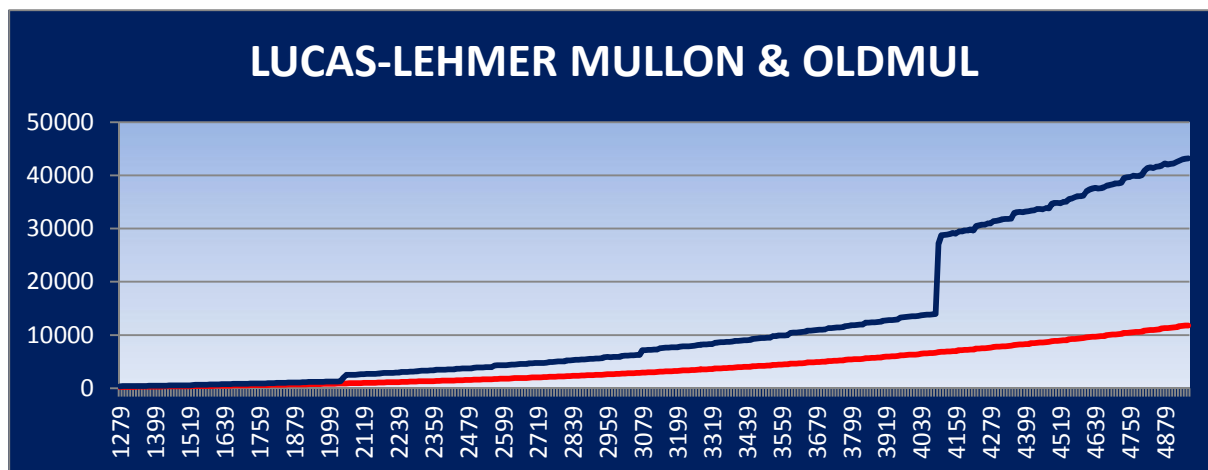
	0x10000000000000000		
szóhossz	MULLON	OLDMUL	arány
100	5	0	0,00
101	0	0	0,00
200	30	0	0,00
201	35	5	7,00
300	110	15	7,33
301	125	0	0,00
400	203	16	12,69
401	203	15	13,53
500	297	16	18,56
501	297	31	9,58
600	782	31	25,23
601	766	16	47,88
700	1032	31	33,29
701	1031	31	33,26
800	1411	51	27,67
801	1425	51	27,94
900	1772	63	28,13
901	1778	63	28,22
1000	2062	80	25,78
1001	2056	78	26,36
1100	4422	94	47,04
1101	4406	94	46,87
1200	5155	114	45,22
1201	5167	116	44,54
1300	6050	131	46,18
1301	6064	135	44,92
1349	6537	144	45,40
1350	6507	145	44,88



64BIT

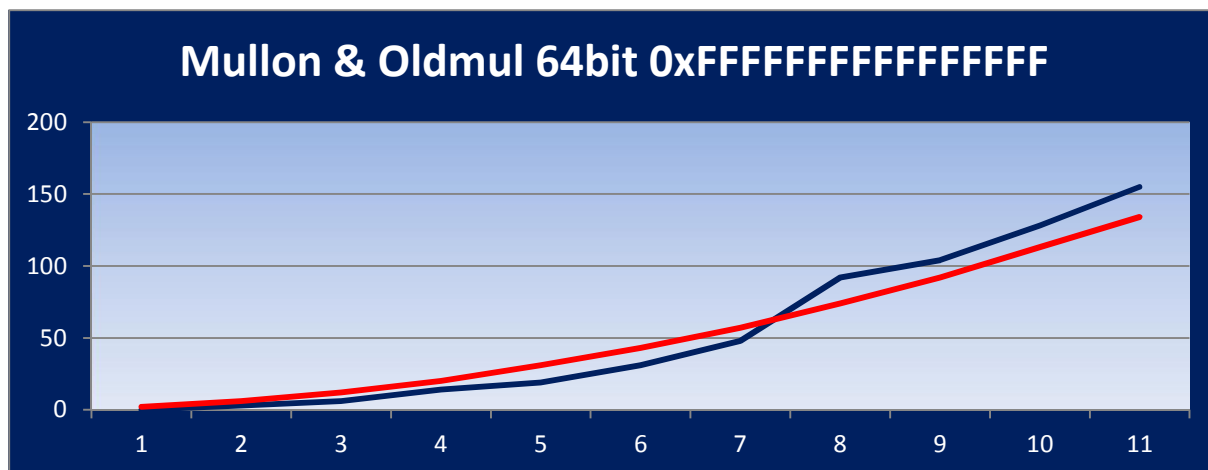
32BIT

kitevő	64bit	32bit	arány
1279	362	1710	4,72
1359	441	2035	4,61
1459	532	2363	4,44
1659	765	3469	4,53
1759	907	3891	4,29
1859	1063	4625	4,35
1959	1203	5157	4,29
2159	2719	13720	5,05
2259	3078	14845	4,82
2359	3360	16486	4,91
2459	3703	18126	4,89
2559	3985	19454	4,88
2659	4547	22705	4,99
2859	5360	25970	4,85
2959	5860	28392	4,85
3059	6219	30112	4,84
3159	7642	36627	4,79
3359	8657	41768	4,82
3459	9282	45409	4,89
3559	9907	48176	4,86
3859	12329	59254	4,81
4059	13798	67146	4,87
4159	29049	156887	5,40
4359	32909	174762	5,31
4459	33628	182607	5,43
4559	35690	189764	5,32
4659	37612	201749	5,36
4759	39722	211968	5,34
4859	41675	225266	5,41
4959	43191	230641	5,34



MULLON
OLDMUL

kitevő	MULLON	OLDMUL	arány
1279	362	236	1,53
1359	441	287	1,54
1459	532	342	1,56
1659	765	488	1,57
1759	907	587	1,55
1859	1063	696	1,53
1959	1203	793	1,52
2159	2719	1043	2,61
2259	3078	1204	2,56
2359	3360	1345	2,50
2459	3703	1525	2,43
2559	3985	1694	2,35
2659	4547	1911	2,38
2859	5360	2351	2,28
3059	6219	2854	2,18
3159	7642	3159	2,42
3359	8657	3774	2,29
3459	9282	4139	2,24
3559	9907	4457	2,22
3759	11423	5212	2,19
3859	12329	5663	2,18
4059	13798	6556	2,10
4159	29049	6983	4,16
4359	32909	8121	4,05
4459	33628	8606	3,91
4559	35690	9238	3,86
4659	37612	9783	3,84
4759	39722	10465	3,80
4859	41675	11046	3,77
4959	43191	11808	3,66



MULLON

OLDMUL

érzékenységgel mérve

	0xFFFFFFFFFFFFFFFF	
szóhossz	MULLON	OLDMUL
1	0	2
2	3	6
3	6	12
4	14	20
5	19	31
6	31	43
7	48	57
8	92	74
9	104	92
10	128	113
11	155	134

Irodalomjegyzék:

1. Virasztó Tamás: Titkosítás és adatretjtés: szószerinti idézés a 13. oldalon
2. Diós Gábor, Rodek Lajos: Assembly programozás: szószerinti idézés a 9. oldalon
3. http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations