

GMP small operands optimization  
or  
Is arithmetic assembly automatically optimal?

Torbjörn Granlund

Swox AB, Sweden

SPEED 2007

# GMP

## **What is GMP?**

- ▶ A low-level bignum library for integers, rationals, and floats.
- ▶ Not an algebra system

# GMP

## What is GMP?

- ▶ A low-level bignum library for integers, rationals, and floats.
- ▶ Not an algebra system

## GMP goals

- ▶ Correctness
- ▶ Performance
- ▶ Performance
- ▶ Performance
- ▶  $\vdots$

## About this presentation

- ▶ Just about small operands
- ▶ Operands of 4097 bits are huge
- ▶ Quadratic algorithms

## About this presentation

- ▶ Just about small operands
- ▶ Operands of 4097 bits are huge
- ▶ Quadratic algorithms
- ▶ Karatsuba, who's that?

# GMP primitives

Simple basic computation primitives:

- ▶ `mpn_add_n`
- ▶ `mpn_sub_n`
- ▶ `mpn_mul_1`
- ▶ `mpn_addmul_1`
- ▶ `mpn_submul_1`
- ▶ `mpn_lshift`
- ▶ `mpn_rshift`

## GMP primitives

<b>name</b>	<b>operation</b>	<b>return value</b>
<code>mpn_add_n</code>	$A \leftarrow B + C$	carry-out
<code>mpn_sub_n</code>	$A \leftarrow B - C$	borrow-out
<code>mpn_mul_1</code>	$A \leftarrow B \times c$	most significant limb
<code>mpn_addmul_1</code>	$A \leftarrow A + B \times c$	most significant limb
<code>mpn_submul_1</code>	$A \leftarrow A - B \times c$	most significant limb
<code>mpn_lshift</code>	$A \leftarrow B \times 2^c$	“shifted out” limb
<code>mpn_rshift</code>	$A \leftarrow B / 2^c$	“shifted out” limb

## Use of primitives (example)

- ▶ Schoolbook multiplication made of:  
`mpn_mul_1, {mpn_addmul_1}*`
- ▶ Schoolbook left-to-right division made of:  
`{q̂-comp, mpn_submul_1, (mpn_add_n) or mpn_sub_n}*`
- ▶ Schoolbook right-to-left division made of:  
`{q-comp, mpn_addmul_1}* , mpn_add_n, (mpn_sub_n)`



GMP coding strategies

## Use assembly code

GMP can use assembly for critical operations.

## Use basic algorithms!

### **BAD:**

```
if  $n = 1$  then  
     $A[0] = B[0] * C[0]$   
else  
    mul_sledgehammer (...)
```

### **GOOD:**

```
if  $n < 30$  then  
    mpn_mul_basecase (...)  
else  
    mul_sledgehammer (...)
```

” All GMP code is rubbish”

All GMP code is due for replacement.

We’re happy with improvements just for a day, or two.

# Assembly in GMP

- ▶ C with inline assembly (GMP 1)
- ▶ + simple assembly loops (GMP 2)
- ▶ + complex assembly loops (GMP 3)
- ▶ + nested loops (GMP 4)
- ▶ + *OSP* loops (GMP 5)

## Assembly strategies (1)

- ▶ Loop recurrency "shallowing"
- ▶ Loop unrolling
- ▶ Software pipelining
- ▶ Loop unrolling + Software pipelining
- ▶ OSP(tm)

## Recurrency shallowing — 3 operation mpn\_add\_n

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
        {
            uword = u[i];
            vword = v[i];
            sum0 = uword + cy;
            cy0 = sum0 < uword;
            sum1 = sum0 + vword;
            cy1 = sum1 < sum0;
            cy = cy0 + cy1;
            r[i] = sum1;
        }
}
```

## Recurrency shallowing — 2 operation mpn\_add\_n

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
        {
            uword = u[i];
            vword = v[i];
            sum0 = uword + vword;
            cy0 = sum0 < uword;
            sum1 = sum0 + cy;
            cy1 = sum1 < sum0;
            cy = cy0 + cy1;
            r[i] = sum1;
        }
}
```



## Recurrency shallowing — 1 operation mpn\_add\_n

```
add (word *r, word *u, word *v, size_t n)
{
    cy = 0;
    for (i = 0; i < n; i++)
        {
            uword = u[i];
            vword = v[i];
            sum0 = uword + vword;
            cy0 = sum0 < uword;
            sum1 = sum0 + cy;
            cy = cy0 | (cy & (sum0 == ~0));
            r[i] = sum1;
        }
}
```

## Software pipelining of a tight loop

```
for (i = 1; i < n; i++)  
  {  
    a0 = ap[0];  
    b0 = bp[0];  
    r0 = a0 * b0;  
    rp[i] = r0;  
  }
```

## Software pipelining of a tight loop

```
a0 = ap[0];  
b0 = bp[0];  
  
for (i = 1; i < n; i++)  
{  
    r0 = a0 * b0;  
    a0 = ap[i];  
    b0 = bp[i];  
    rp[i-1] = r0;  
}  
  
r0 = a0 * b0;  
rp[n - 1] = r0;
```

## Loop unrolling + Software pipelining

feed-in	pipelined loop	wind-down
-----	-----	-----
a0 = ap[0];	for (i = 4; i < n, i+=2)	
b0 = bp[0];	{	
a1 = ap[1];	rp[i-4] = r0;	rp[n-4] = r0;
b1 = bp[1];	r0 = a0 * b0;	r0 = a0 * b0;
	a0 = ap[i];	rp[n-3] = r1;
r0 = a0 * b0;	b0 = bp[i];	r1 = a1 * b1;
a0 = ap[2];	rp[i-3] = r1;	
b0 = bp[2];	r1 = a1 * b1;	rp[n-2] = r0;
r1 = a1 * b1;	a1 = ap[i+1];	rp[n-1] = r1;
a1 = ap[3];	b1 = bp[i+1];	
b1 = bp[3];	}	

## Assembly strategies (2)

- ▶ Explore CPU pipelines
- ▶ (Read manufacturers' pipeline docs)
- ▶ Code for each pipeline
- ▶ Find "micro algorithms" for ISA/pipeline
- ▶ Work around pipeline flaws
- ▶ Avoid the branch misprediction death

Pentium 4 has a 10 cycle carry flag latency. . .

## Unpredictable branches – BAD

```
if (carry_from_add)
    do something
else
    do some other thing
```

## Unpredictable branches – OK

```
if (very_unlikely_condition)
    do something
else
    do some other thing
```



## Assembly strategies (3)

Move away from `mpn_addmul_1!`

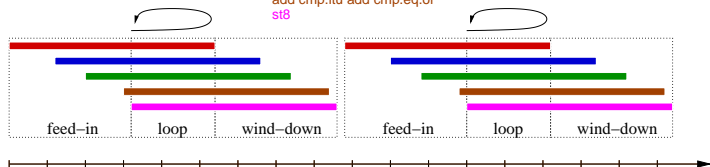
## Assembly strategies (3)

Move away from `mpn_addmul_1`!

- ▶ Make `mpn_addmul_2` the main primitive
- ▶ Or `mpn_addmul_3`, etc.
- ▶ Base `mpn_mul_basecase` on these
- ▶ Base a `mpn_redc_N` for each `mpn_addmul_N`

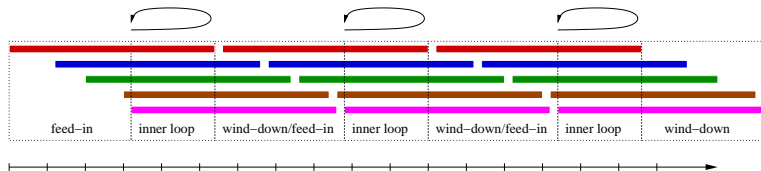
# OSP: Simple Software Pipelining

ldf8 ldf8  
xma.l xma.hu  
getf.sig getf.sig  
add cmp.ltu add cmp.eq.or  
st8



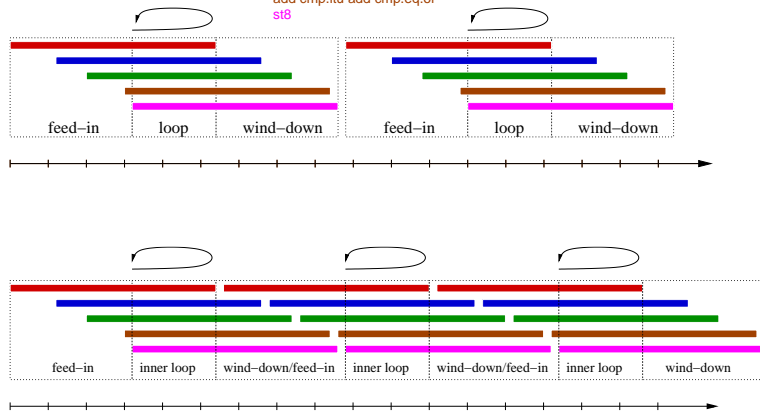
# Overlapped Software Pipelining (OSP)

ldf8 ldf8  
xma.l xma.hu  
getf.sig getf.sig  
add cmp.ltu add cmp.eq.or  
st8



# OSP: Simple s/w Pipelining vs OSP

ldf8 ldf8  
xma.l xma.hu  
getf.sig getf.sig  
add cmp.ltu add cmp.eq.or  
st8



# OpenSSL's and assembly

- ▶ Inadequate primitives
- ▶ Assembly is fast, period
- ▶ More is better

The answer to the initial question: NO.

## GMP 4.2 division

	Left-to-right	Right-to-left/REDC
Schoolbook on addmul_1	Y	Y
Schoolbook on addmul_N	–	Y
Burnikel-Ziegler $O(M(n) \log n)$	–	Y
Barrett-MUA $O(M(n))$	–	–



## GMP 5 division

	Left-to-right	Right-to-left/REDC
Schoolbook on addmul_1 + OSP	Y	Y
Schoolbook on addmul_N + OSP	Y	Y
Burnikel-Ziegler $O(M(n) \log n)$	Y	Y
Barrett-MUA $O(M(n))$	Y	Y

# Burnikel-Ziegler division

```
mp_limb_t
mpn_dc_bdiv_qr (mp_ptr qp, mp_ptr np, mp_srcptr dp, mp_size_t n, mp_limb_t dinv)
{ lo = n >> 1; /* floor(n/2) */
  hi = n - lo; /* ceil(n/2) */

  if (BELOW_THRESHOLD (lo, DC_BDIV_QR_THRESHOLD))
    cy = mpn_sb_bdiv_qr (qp, np, 2 * lo, dp, lo, dinv);
  else
    cy = mpn_dc_bdiv_qr_n (qp, np, dp, lo, dinv, tp);

  mpn_mul (tp, dp + lo, hi, qp, lo);

  mpn_incr_u (tp + lo, cy);
  rh = mpn_sub (np + lo, np + lo, n + hi, tp, n);

  if (BELOW_THRESHOLD (hi, DC_BDIV_QR_THRESHOLD))
    cy = mpn_sb_bdiv_qr (qp + lo, np + lo, 2 * hi, dp, hi, dinv);
  else
    cy = mpn_dc_bdiv_qr_n (qp + lo, np + lo, dp, hi, dinv, tp);

  mpn_mul (tp, qp + lo, hi, dp + hi, lo);

  mpn_incr_u (tp + hi, cy);
  return rh + mpn_sub_n (np + n, np + n, tp, n);
}
```

## Relevance to cryptography

- ▶ All RSA sizes benefit from
  - ▶ `mpn_addmul_2` (`mpn_addmul_3`, ...) in multiplication
  - ▶ `mpn_addmul_2` (`mpn_addmul_3`, ...) in REDC
  - ▶ OSP in multiplication
  - ▶ OSP in REDC
- ▶ Larger RSA sizes (2048-) benefit from
  - ▶ Burnikel-Ziegler REDC

## 64-bit vs 32-bit cryptography

	512	1024	2048	4096
Athlon32	566950	3051800	19415000	130360000
Athlon64	170370	759670	4690800	30137000
A32/A64	3.33	4.02	4.14	4.33

Athlon32:

MUL\_KARATSUBA\_THRESHOLD 26 (832 bits)

SQR\_KARATSUBA\_THRESHOLD 52 (1664 bits)

Athlon64:

MUL\_KARATSUBA\_THRESHOLD 25 (1600 bits)

SQR\_KARATSUBA\_THRESHOLD 80 (5120 bits)