GMP small operands optimization
or
Is arithmetic assembly automatically optimal?

Torbjörn Granlund

Swox AB, Sweden

SPEED 2007
What is GMP?

- A low-level bignum library for integers, rationals, and floats.
- Not an algebra system
What is GMP?
- A low-level bignum library for integers, rationals, and floats.
- Not an algebra system

GMP goals
- Correctness
- Performance
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?
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

<table>
<thead>
<tr>
<th>name</th>
<th>operation</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>mpn_add_n</td>
<td>$A \leftarrow B + C$</td>
<td>carry-out</td>
</tr>
<tr>
<td>mpn_sub_n</td>
<td>$A \leftarrow B - C$</td>
<td>borrow-out</td>
</tr>
<tr>
<td>mpn_mul_1</td>
<td>$A \leftarrow B \times c$</td>
<td>most significant limb</td>
</tr>
<tr>
<td>mpn_addmul_1</td>
<td>$A \leftarrow A + B \times c$</td>
<td>most significant limb</td>
</tr>
<tr>
<td>mpn_submul_1</td>
<td>$A \leftarrow A - B \times c$</td>
<td>most significant limb</td>
</tr>
<tr>
<td>mpn_lshift</td>
<td>$A \leftarrow B \times 2^c$</td>
<td>“shifted out” limb</td>
</tr>
<tr>
<td>mpn_rshift</td>
<td>$A \leftarrow B/2^c$</td>
<td>“shifted out” limb</td>
</tr>
</tbody>
</table>
Use of primitives (example)

- Schoolbook multiplication made of:
  \( \text{mpn\_mul\_1}, \{\text{mpn\_addmul\_1}\}^* \)

- Schoolbook left-to-right division made of:
  \( \{\hat{q}\text{-comp, mpn\_submul\_1, (mpn\_add\_n) or mpn\_sub\_n}\}^* \)

- Schoolbook right-to-left division made of:
  \( \{q\text{-comp, mpn\_addmul\_1}\}^*, \text{mpn\_add\_n, (mpn\_sub\_n)} \)
GMP coding strategies
Use assembly code

GMP can use assembly for critical operations.
Use basic algorithms!

BAD:
if \( n = 1 \) then
    \( A[0] = B[0] \times C[0] \)
else
    \text{mul\_sledgehammer} (\ldots)

GOOD:
if \( n < 30 \) then
    \text{mpn\_mul\_basecase} (\ldots)
else
    \text{mul\_sledgehammer} (\ldots)
"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
- + simple assembly loops
- + complex assembly loops
- + nested loops
- + OSP loops
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

```c
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

```c
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

```c
for (i = 1; i < n; i++)
{
    a0 = ap[0];
    b0 = bp[0];
    r0 = a0 * b0;
    rp[i] = r0;
}
```
Software pipelining of a tight loop

```c
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

<table>
<thead>
<tr>
<th>feed-in</th>
<th>pipelined loop</th>
<th>wind-down</th>
</tr>
</thead>
<tbody>
<tr>
<td>a0 = ap[0];</td>
<td>for (i = 4; i &lt; n, i+=2)</td>
<td></td>
</tr>
<tr>
<td>b0 = bp[0];</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a1 = ap[1];</td>
<td>rp[i-4] = r0;</td>
<td>rp[n-4] = r0;</td>
</tr>
<tr>
<td>b1 = bp[1];</td>
<td>r0 = a0 * b0;</td>
<td>r0 = a0 * b0;</td>
</tr>
<tr>
<td></td>
<td>a0 = ap[i];</td>
<td>rp[n-3] = r1;</td>
</tr>
<tr>
<td>r0 = a0 * b0;</td>
<td>b0 = bp[i];</td>
<td>r1 = a1 * b1;</td>
</tr>
<tr>
<td>a0 = ap[2];</td>
<td>rp[i-3] = r1;</td>
<td></td>
</tr>
<tr>
<td>b0 = bp[2];</td>
<td>r1 = a1 * b1;</td>
<td>rp[n-2] = r0;</td>
</tr>
<tr>
<td>b1 = bp[3];</td>
<td>a1 = ap[i+1];</td>
<td>rp[n-1] = r1;</td>
</tr>
<tr>
<td>a1 = ap[3];</td>
<td>b1 = bp[i+1];</td>
<td></td>
</tr>
<tr>
<td>b1 = bp[3];</td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
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
Overlapped Software Pipelining (OSP)
OSP: Simple s/w Pipelining vs OSP
OpenSSL’s and assembly

- Inadequate primitives
- Assembly is fast, period
- More is better
The answer to the initial question: NO.
<table>
<thead>
<tr>
<th>Method</th>
<th>Left-to-right</th>
<th>Right-to-left/REDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schoolbook on addmul_1</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Schoolbook on addmul_N</td>
<td>–</td>
<td>Y</td>
</tr>
<tr>
<td>Burnikel-Ziegler $O(M(n) \log n)$</td>
<td>–</td>
<td>Y</td>
</tr>
<tr>
<td>Barrett-MUA $O(M(n))$</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
GMP 5 division

<table>
<thead>
<tr>
<th>Method/Algorithm</th>
<th>Left-to-right</th>
<th>Right-to-left/REDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schoolbook on addmul_1 + OSP</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Schoolbook on addmul_N + OSP</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Burnikel-Ziegler $O(M(n) \log n)$</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Barrett-MUA $O(M(n))$</td>
<td>Y</td>
<td>Y</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th></th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Athlon32</td>
<td>566950</td>
<td>3051800</td>
<td>19415000</td>
<td>130360000</td>
</tr>
<tr>
<td>Athlon64</td>
<td>170370</td>
<td>759670</td>
<td>4690800</td>
<td>30137000</td>
</tr>
<tr>
<td>A32/A64</td>
<td>3.33</td>
<td>4.02</td>
<td>4.14</td>
<td>4.33</td>
</tr>
</tbody>
</table>

**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)