Rechnen mit großen ganzen Zahlen

M. Anton Ertl, TU Wien

Große Zahlen

- mehrere Maschinenwörter (> 128 bits): Multi-precision (MP)
- Anwendung?
  Kryptographie: 256b–4096b (performancekritisch)
  y-cruncher (performancekritisch)
  Überlauf kleiner Zahlen (selten)
  Weitere Anwendungen?
- Notation:
  $b = 2^{\text{bits}}$: Basis, das Maschinenwort als Ziffer
  s, s_0, `s_t`, `s`, `s1`: kleine Zahlen ($< b$)
  d, d_0, `d_t`, `d`, `s1`: doppelt so lange Zahlen ($< b^2$)
  l, l_0, `s_t *`, `l`, `l1`: große Zahlen ($\geq b$)

... in höheren Programmiersprachen

- Bignums
- im Normalfall klein
  oft dafür optimiert
- große Zahlen brauchen boxing und unboxing
  Ineffizienz beim Rechnen ist dann auch schon egal

... in Low-Level Sprachen

- keine direkte Unterstützung für große Zahlen
- etwas Unterstützung auf Maschinenwort-Ebene
- GMP-Bibliothek: niedrigster Level: duzende `mpn_`-Funktionen
  $l = l_1 + l_2$: `mpn_add_n()` (gleiche Längen)
  $l = sl_1$: `mpn_mul_1()`
  $l \leftarrow l + sl_1$: `mpn_addmul_1()`
  $l = l_1 l_2$: `mpn_mul()`
- Maschinenebene: verschiedene Architektureigenschaften
  verschiedene Anzahl von Carry-Flags (RISC-V: 0, ARM A64: 1, Intel ADX: 2)
  verschiedene Multiplikationsbefehle
  verschiedene Divisionsbefehle

Addition: $l = l1 + l2$

AMD64:

```
L: movq (l1,i,8),tmp
   adcq (l2,i,8),tmp
   movq tmp, (l,i,8)
   incq i
   decq n
   jnz  L
```

clang -Os mit Intel intrinsic

```
L: movq (l1,i,8), tmp
   addb $-1, carry
   adcq (l2,i,8),tmp
   setb carry
   movq tmp, (l,i,8)
   incq i
   cmpq i, n
   jne  L
```

RISC-V (RV64G):

```
L: ld summand1, 0(l1p)
   ld summand2, 0(l2p)
   add  tmp1, carry, summand1
   sltu carry1, tmp1, summand1
   add sum, tmp1, summand2
   sltu carry2, sum, tmp1
   add carry, carry2, carry1
   sd sum, 0(lp)
   addi n, n, -1
   addi lp, lp, 8
   addi l2p, l2p, 8
   addi l1p, l1p, 8
   bnez n, L
```

In C mit `uint128_t` ausdrückbar

```
for (i=0; i<n; i++) {
  s_t s1=l1[i];
  s_t s2=l2[i];
  d_t d=s1+(d_t)s2+carry;
  carry = d>>64;
  l[i] = d;
}
```

AMD64: schlechter Code

RV64G: ähnlicher Code

Kogge-Stone-Addition mit AVX-512:

http://www.numberworld.org/y-cruncher/internals/addition.html

langsam auf Intel

Addition: $l = l1 + l2 + l3$

Keine GMP-Funktion (stattdessen: $l4 = l1 + l2; l = l4 + l3$)

Intel ADX (seit 2014):

```
L: movq  (l1,i,8),tmp
   adcxq (l2,i,8),tmp
   adoxq (l3,i,8),tmp
   movq tmp, (l,i,8)
   incq i
   decq n
   jnz  L
```

RV64G (nur Zentrum)

```
   add tmp1, summand1, carry
   sltu carry1, tmp1, summand1
   add tmp2, summand2, tmp1
   sltu carry2, tmp2, tmp1
   add carry12, carry1, carry2
   add sum, summand3, tmp2
   sltu carry3, sum, tmp2
   add carry, carry12, carry3
```

aus C-code mit `uint128_t` `__builtin_addc`: schlechter

C-code mit `uint128_t` (nur Zentrum):

```
d_t d = s1+(d_t)s2+(d_t)s3+carry;
carry = d>>64;
```

Multiplikationsschritt: $l = sl_1 + l_2$

C-Code mit `uint128_t`:

```
for (i=0; i<n; i++) {
  s_t s1 = l1[i];
  s_t s2 = l2[i];
  d_t d =
    s*(d_t)s1+(d_t)s2+carry;
  carry = d>>64;
  l[i] = d;
}
```

$(b-1)^2 + 2(b-1) = b^2 - 1$

Immer noch Übertrag in d

Intel ADX (2 Iterationen)

```
# implicit operand
#  of mulx: s
mulxq s1a, dloa, carrya
adcxq carryb, dloa
adoxq s2a, dloa
# carry=carrya+c+o
mulxq s1b, dlob, carryb
adcxq carrya, dlob
adoxq s2b, dlob
# carry = carryb+c+o
```

RV64G (1 Iteration)

```
mulhu carry0, s1, s
mul   dlo, s1, s
add   carry2, s2, carry
sltu  carry1, carry2, s2
add   carry4, dlo, carry2
sltu  carry5, carry4, carry2
add   carry3, carry1, carry0
add   carry, carry3, carry5
```

Division

- Eigener Vortrag (oder mehrere)
- Unterschiede in der Architektur: d/s (AMD64) vs. s/s (die meisten)

Neuer Ansatz: große Zahlen in der Sprache

```
/* modeled on gcc vector extensions */
typedef unsigned uint4096 __attribute__ ((multi_precision (512)))
uint4096 l1, l2, l3, l4;
/* modeled on variable-lenth arrays */
typedef unsigned long vlint __attribute__ ((multi_precision));
vlint l5[n+1], l6[n], l7[n], l8[n];
unsigned long s[3];
l5 = l6+l7+l8;
l1 = s[0]*l2+(s[1]*l3<<64)+(s[2]*l4<<128);
```

- Speicher wird angegeben
  kein Boxing/Unboxing
  Überlauf möglich
- Programmierer drückt direkt aus, was gemeint ist
- Compiler erzeugt für die Architektur passenden Code
  vermeidet ggf. Befehle, die Carry-flags zerstören
- Lohnt es den Aufwand?
- Alternative: Auto-MP-isierung
  Compiler erkennt typischen multi-precision Code
  Programmierer sollte wissen, was er nicht schreiben darf

Zusammenfassung

- Multi-precision: beliebig viele Maschinenwörter für eine Zahl
- Unterschiede in den Architekturen (0–2 Carry-bits)
- Bignums in höheren Programmiersprachen ineffizient
- GMP-Bibliothek: Ineffizient für Kombinationen von Aufrufen
- C-Code mit `uint128_t`: Ineffizienter Code für Addition
- Existierende C-Erweiterung, Intrinsic: auch ineffizienter Code
- Vorschlag: Erweiterung um große Zahlen mit bestimmtem Speicher