x86 Assembler Übung: Rechnen mit beliebig langen Zahlen

Klaus Kusche

Gesucht sind diesmal *vier Assembler-Funktion* zum gegebenen C++-Code und Headerfile.
Sie sollen die *vier Grundrechnungsarten auf beliebig langen Zahlen* implementieren:
- Die Additionsfunktion addiert eine lange Zahl zu einer anderen langen Zahl dazu.
- Die Subtraktionsfunktion zieht eine lange Zahl von einer anderen langen Zahl ab.
- Die Multiplikationsfunktion multipliziert eine lange mit einer kurzen Zahl.
- Die Divisionsfunktion dividiert eine lange durch eine kurze Zahl.

„Lange Zahl“ bedeutet ein beliebig langes *Array* von **`uint64`**-Elementen,
wobei das *erste* Element das *höchstwertige* und das *letzte* das *niederwertigste* ist,
und eine ***„kurze Zahl“*** ist ein einzelner **`uint64`**-Wert.
Alle Zahlen und Rechnungen sind *vorzeichenlos* (nur positive Werte).

Weiters soll der Assembler-File eine *globale* **`int`**-*Variable* exportieren.
Sie muss *vor dem ersten Aufruf vom Aufrufer gesetzt* werden,
und zwar auf die *Anzahl der Elemente* in den Arrays, die die langen Zahlen darstellen
(d.h. alle langen Zahlen haben die *gleiche*, vom Aufrufer festzulegende *Länge*).

Der *Returnwert* aller vier Funktionen soll ein **`int`**-*Fehlercode* sein:
`0` für *Erfolg*
`1` bei *Überlauf* (Addition, Multiplikation), Unterlauf (Subtraktion) oder Division durch **`0`**.
`-1` bei *Aufruf-Fehler* (**`NULL`** als Parameter oder die Array-Längen-Variable ist *nicht gesetzt*).

Hinweise:

- Zum Anlegen der *globalen Variable*:
  - Der Assembler-Befehl **`.int`** reserviert Platz für einen (oder mehrere) **`int`**'s
    und *initialisiert* den **`int`** mit dem als *Operand* angegebenen Wert
    (nimm **`0`**, damit du erkennst, wenn der Aufrufer noch keinen Wert gesetzt hat).
  - Globale, schreibbare Variablen gehören in die Section **`.data`**, nicht **`.text`**.
  - Der Variablenname muss so wie die Funktionsnamen *global gemacht* werden.
- Ich schlage vor, diese Variable in allen Funktionen gleich am Beginn in
  ein *freies Register* zu laden, wir brauchen sie ja als *Array-Index und Schleifenzähler*.

  Mach dabei aus dem 32-bit-**`int`** mit Vorzeichen einen 64-bit-**`int`** mit Vorzeichen.
  Der Befehl dazu heißt **`movsx`** (*„move with sign extension“*),
  die Quelle ist in unserem Fall ein *Speicherzugriff mit Längenangabe*
  (weil die Quell-Länge ja eine andere als die Ziel-Länge ist).

  Da der *Index des letzten Array-Elementes* um **`1`** kleiner als die Anzahl der Elemente
  ist, kannst du auch gleich **`1`** von diesem Register *abziehen*.
  Wenn dabei eine *negative Zahl* herauskommt oder ein *Überlauf* (Overflow-Flag)
  auftritt (Tipp: *„Jump less“* prüft beides auf einmal),
  dann war unsere globale Variable *nicht oder falsch initialisiert*,
  und die Funktion sollte sofort mit einem *Fehler-Returnwert* enden.
- Auch die „lange Zahl“-*Parameter* der Funktionen solltest du auf **`NULL`** *prüfen*,
  bevor die eigentliche Rechnung beginnt.
- Vergiss nicht, am Ende der Funktionen immer den richtigen *Returnwert* zu setzen!

- **Addition und Subtraktion** sind relativ einfach:

  Gerechnet wird wie man das in der Grundschule gelernt hat,
  indem man *von hinten nach vorne* Element für Element addiert bzw. subtrahiert
  und dabei den *Übertrag* mitrechnet.

  Die Befehle für das Rechnen mit langen Zahlen heißen **`adc`** (*„add with carry“*)
  und **`sbb`** (*„subtract with borrow“*). Sie funktionieren wie **`add`** und **`sub`**,
  zählen aber zusätzlich das *Carry-Flag* (*„Übertrag“*) dazu bzw. weg.

  Betreffend *Carry-Flag* sind drei Dinge zu beachten:

  - *Vor der Schleife* muss das Carry-Flag für das erste **`adc`** bzw. **`sbb`**
    mit dem Befehl **`clc`** (*„clear carry“*) auf **`0`** gesetzt werden.
  - *In der Schleife* darf es *keinen* anderen Befehl geben, der das Carry-Flag *ändert*,
    weil es ja von einem **`adc`** / **`sbb`** bis zum nächsten erhalten bleiben muss.
    Es darf innerhalb der Schleife also kein weiteres **`add`** oder **`sub`**
    und auch kein **`cmp`** oder **`test`** vorkommen.

    Zum Glück hat Intel hier mitgedacht:
    Die beiden Befehle **`inc`** (*„increment“ = „zähl 1 dazu“*) und **`dec`** (*„decrement“ = „zieh 1 ab“*) setzen *alle Flags außer dem Carry-Flag* (das bleibt unverändert).
    Wir können unser *Index- bzw. Schleifenzähler-Register* also mit **`dec`** *erniedrigen*
    und dann mit dem Sign-Flag (*„Vorzeichen“*) *prüfen*, ob es *negativ* geworden ist
    (dann *endet* unsere Schleife).
  - Ist das Carry-Flag *nach dem Ende der Schleife gesetzt*,
    so ist ein *Überlauf* (bzw. Unterlauf) aufgetreten.
    Prüfe das und ende in diesem Fall mit dem Returnwert **`1`**.

  Hinweise betreffend *Speicherzugriff*:

  - Du brauchst eine *Index-Adressierung* mit dem jeweiligen Pointer-Parameter
    als Basis und unserem Index- bzw. Schleifenzähler-Register als Index,
    und zwar multipliziert mit 8 (weil ein **`uint64`** ja 8 Bytes lang ist).
  - So wie alle Rechen-Befehle auf x86 können **`adc`** und **`sbb`** *nicht* mit *zwei*
    Speicher-Operanden arbeiten. Du brauchst also *zwei Befehle*:
    Lade zuerst die Quelle in ein freies *Hilfs-Register* und
    zähle dann dieses Hilfsregister zum Ziel dazu (bzw. zieh es vom Ziel ab).

- Die **Multiplikation** ist die komplizierteste Funktion.

  Auch sie funktioniert wie in der Grundschule gelernt *elementweise von hinten nach vorne*, wobei jede Teilmultiplikation ein *doppelt langes Zwischenergebnis* liefert,
  zu dem zuerst einmal der *Übertrag der vorigen Multiplikation addiert* werden muss.
  Dann wird die *hintere Hälfte* des Zwischenergebnisses im *Array-Element*
  gespeichert, und die *vordere Hälfte* ist der *Übertrag* für die nächste Runde.

  Der vorzeichenlose **`mul`**-Befehl hat so wie der vorzeichenlose **`div`**-Befehl
  nur *einen frei wählbaren Operanden*.
  Der *zweite Operand* ist immer das **`ax`**/**`eax`**/**`rax`**-Register,
  die *untere Hälfte* des doppelt langen Ergebnisses liegt immer in **`ax`**/**`eax`**/**`rax`**
  und die *obere* in **`dx`**/**`edx`**/**`rdx`**.

Diesmal brauchen wir *drei Hilfs-Register*:
- Eines wie bisher für den *Index bzw. Schleifenzähler*.
- Eines, in das wir den *zweiten Parameter* kopieren, weil **`mul`** ja **`rdx`** verwendet.
- Und eines für den *Übertrag*.

Das *Übertrags-Register* müssen wir *vor der Schleife* auf **`0`** setzen
und *nach der Schleife* auf **`0`** prüfen: Bleibt ein Übertrag aus dem *letzten* **`mul`**,
so endet die Funktion mit *Returnwert „Überlauf“*.

In der Schleife passiert Folgendes:

- Das *aktuelle Array-Element* wird in **`rax`** *geladen*.
  Dann wird **`rax`** *mit dem Parameter-Register multipliziert*.
- Als nächstes wird das *Übertrags-Register* zum **`mul`**-Ergebnis *addiert*.
  Dabei könnte ein Übertrag aus den hinteren 64 bit der Addition entstehen,
  also sind *zwei Schritte* nötig:
  - Zuerst ein normales **`add`** vom Übertrags-Register zum **`rax`**.
  - Und dann ein **`adc`** mit dem Wert **`0`** zum **`rdx`** (Warum?).
- Dann wird das *Ergebnis gespeichert*:
  Die untere Hälfte aus **`rax`** ins *Array-Element*,
  und die obere aus **`rdx`** in unser *Übertrags-Register*.
- Schließlich wird der *Index erniedrigt und geprüft*.

- Die **Division** ist einfacher, aber insofern neu, als sie elementweise
  von vorne nach hinten arbeitet und nicht von hinten nach vorne.

  Wir erinnern uns an den **`div`**-Befehl: Er hat als *einzigen Operanden* die Zahl,
  durch die geteilt wird. Die *zu teilende Zahl hat doppelte Länge*,
  das **`div`** erwartet die *obere* Hälfte in **`rdx`** und die *untere* in **`rax`**
  und liefert das *Divisionsergebnis* in **`rax`** und den *Divisionsrest* in **`rdx`**.

  Wir verwenden wieder *drei Hilfsregister*:
  - Ich würde wie bisher ein Register als *Abwärts-Schleifenzähler* bis **`0`** verwenden.
  - Ein zweites Register würde ich als *aufwärtszählenden Index* verwenden.
    Es wird *vor der Schleife* auf **`0`** gesetzt und in jedem Durchlauf um **`1`** *erhöht*.
  - Und das dritte Hilfsregister ist wieder eine *Kopie des Parameters* aus **`rdx`**.

  Außerdem muss *vor der Schleife*
  - das *„Rest-Register“* **`rdx`** für die erste Division auf **`0`** gesetzt werden
  - und der *zweite Parameter* auf **`0`** geprüft werden
    (wenn ja: Return mit Returnwert *„Division durch 0“*)

  Die Schleife selbst ist relativ einfach:

  - Das nächste *Array-Element* wird nach **`rax`** *geladen*
    und dann *durch das Parameter-Register dividiert*.
  - Das *Divisionsergebnis* wird von **`rax`** zurück ins *Array-Element* gespeichert,
    der Divisionsrest in **`rdx`** bleibt dort gleich als obere Hälfte für das nächste **`div`**.
  - Dann werden die *beiden Schleifenzähler erhöht bzw. erniedrigt*.

    Eigentlich müsste man jede Division auf Überlauf prüfen, aber rein
    von den Zahlen und der Logik her kann bei uns *kein Überlauf* auftreten.