

# アセンブリ言語 期末課題 レポート

講義名: アセンブリ言語

学籍番号: 23B30032

氏名: 飯田悠太

作成日: 2024 年 11 月 7 日

本レポートは、アセンブリ言語の期末課題である calc0 から calc3 までの 4 つのバージョンの電卓コンパイラそれぞれについて、実装方法とその工夫、考察について述べるものである。

## 1 calc0

この章では calc0 の実装、つまり課題である calc1 から calc3 までのアセンブリを出力するベースとなる C のコードについて述べる。

### 1.1 全体の実装

入力された式については p というポインタ型の変数に格納し、これを先頭から 1 文字ずつ while で読んでいくことにより、電卓の入力を 1 文字ずつ読んでいくということを再現した。

入力された要素については、数字、演算子、メモリキー、符号反転キーの 4 つに分類して処理することにした。また、演算結果を処理するための変数として、現在入力されている数値を管理する変数 num と、演算子を管理する変数 lastOp、現在までの計算結果を管理する変数 acc、メモリ機能の値を管理する変数 mem、符号反転キーが入力された回数を管理する変数 countS を用意した。

以下、入力要素の種類ごとの処理について述べる。

#### 1.1.1 数値の管理

数値については、数字が入力されるたびに既存の num を 10 倍し、それに現在見ている数字 \* p から '0' を引いたものを加えることで、数値を管理することにした。これは、10 進数の数値の表現方法をそのまま利用したものであり、簡単に理解することができる。文字型から数値への変換には、数字に連続されたコードが与えられるという ASCII コードの性質を利用している。

#### 1.1.2 演算子の管理

もし現在見ている文字が演算子 (+, -, \*, /, =) である場合、まず見ている文字の次の文字が演算子でなくなるまでポインタを進める。これは、電卓で言うところの「最後に入力された記号キー」を拾ってきて、それ以外の演算子の入力は無視する、という動作を実現するためである。

演算子の入力が確定した時点で、今入力された演算子ではなく、元々持っていた演算子で計算を行う。これは、演算を行うためには、演算子の前までの計算値と演算子の直後にくる数値の 2 つが必要であるが、演算子を見た時点では演算子直後の数値が確定していないので、代わりに前回の演算子の前までの計算値と前回の演算子、そして前回の演算子直後の数値を用いて計算する必要があるからである。それゆえ、lastOp の初期値は + としておく必要がある。なぜならば、式中で初めて演算子を見た時に行うべき計算は acc に num を代入するという操作であり、その操作は acc の初期値 0 に num を加えるという操作と等価であるからである。

また、符号の反転については、num の符号について、countS が偶数の時には符号を反転し、奇数の時には反転しないようにすることで、符号反転キーが押された回数に応じて符号を反転することができる。

計算を終えた後、現在見ている演算子を lastOp に代入し、num、countS を 0 にリセットする。これにより、次の数値の入力に備えることができる。

### 1.1.3 メモリ機能の管理

メモリ管理機能について、まず、Cが入力された場合はmemにaccの値を代入し、Rが入力された場合は、accにmemの値を代入するようにしている。これらは非常にシンプルである。

次に、P、つまりメモリ加算キーが入力された場合について考える。この場合、まずPが入力される前までの演算結果を用意する必要があるので、Pが入力された時点でのacc, num, lastOp, countSの値に基づいて演算を行う。そしてその値をmemに加算する。これにより、メモリ加算キーが入力された時点までの計算結果をメモリに保存することができる。すると、mem以外の各種変数が保持している値はもはや必要がなくなる(なぜならばメモリに保存されているから)ので、これらは全て初期値にリセットする。

M、つまりメモリ減算キーが入力された場合の動作は、memへの加算が減算に変わるだけで、Pの場合とほぼ同様の処理を行うことで実現できる。

## 1.2 実装の工夫点

この実装で工夫した点はlastOpの初期値を+にした上で、演算子を遅延処理するようにしたことである。これにより、最初の数値入力も含めて全て同一の方法で処理することができ、コードの可読性向上につながっている。

## 2 calc1

この章ではcalc1の実装について述べる。なお、実装のベースの考え方はcalc0と同様であるので、本章ではcalc0でのC言語ベースの実装をどのようにアセンブリに落とし込んだかについて述べる。

### 2.1 変数の取り扱い

まず、課題のレギュレーションの制約上、calc0ではC言語の変数として管理していたnum, acc, mem, countSを、アセンブリで管理する必要がある。そこで、accはレジスタeax, numはレジスタecxで管理し、memとcountSはスタックで管理することにした。

このように設計した理由を述べる。そもそも、アセンブリコードを書き始めた時はこれら4つの変数は全てレジスタで管理していた。しかしながら、コードを書き進めるうちに、使用するアセンブリが混雑してきたり、誤ってcallee-saveなレジスタをスタックへの退避なしで使用してしまふミスが発生してしまったりすることがあると気づいた。そこで、比べて使用頻度の少ないmemとcountSをスタックで管理し、使用するタイミングでのみスタックから取り出して空いているレジスタに読み込んだり、直接アドレスを参照するようにした。また、numとaccに割り当てるレジスタの選定であるが、accは最終的な式の計算結果が格納されることが期待されるため、関数の返り値の格納に使われるeaxを使用することにした。numを格納するレジスタについては、caller-saveなレジスタのうちeaxでないもの、という理由でecxを選択した。

また、このコード上では変数に0を代入する操作を頻繁に行うが、多くの部分で単にmov命令で即値0を代入するのではなく、xor命令を用いて0クリアを行うようにした。これは、mov命令による即値代入よりもxor命令を用いた方が高速であるという知見を得たからである。また、そのような背景があるため0クリアにはxor命令を用いることが一般的であるという言説を聞き、慣例に則った方がより読み手に理解されやすいだろうと思いxorを用いたという側面もある。これもコードの工夫と言えるだろう。

## 2.2 数値の管理

数値の取り扱いについては、calc0 で説明した実装を単純にアセンブリに置き換えて行ったのみである。

## 2.3 演算子の管理

演算子の管理について、まず lastOp についてはレギュレーション上 C 言語での管理が認められていたもので、calc0 通りの実装のまま行った。

それ以降の処理について述べる。まず初めに符号反転キーの処理を行うことにした。符号反転キーが押された回数は常にスタックの上から 2 番目に置かれているので、これを空いている rdx レジスタに読み込む。その後 testb 命令を使うことにより、rdx&&1 の結果が 0 であるかどうか、つまり符号反転キーが押された回数が偶数か奇数かを判定するようにした。これにより、ゼロフラグがセットされている時、つまり符号反転キーが偶数回押されている時は数値の符号を反転させる negl 命令をジャンプするようにした。以上のようにして、符号反転キーの処理を行った。

ここで、ジャンプ用のラベルとして 1、というのをを用いているが、これはローカルラベルである。これを用いた理由は、この命令は複数回出力されるため、グローバルラベルを用いるとラベル名の衝突によるエラーが発生してしまうからである。

また、その後の演算は lastOp に記録された演算子に対応するアセンブリ命令を用いるだけであるが、割り算の場合においては若干実装に工夫があるので、それについて述べる。割り算のとき、負数の取り扱いのために符号拡張を行う必要がある。講義資料では eax レジスタの正負に合わせて mov 命令により edx の上位ビットをセットする方法が紹介されていたが、今回は cltd 命令を用いた。これにより、割り算の際に eax の符号に合わせて mov 命令でセットする値を変更する、というコードを書かなくて良くなり、コードが簡潔になった。

## 2.4 メモリ機能の管理

メモリ機能の管理については、calc0 で述べた実装をそのままアセンブリに置き換えたのみである。メモリの値についてはスタックの先頭に積むことにしたので、メモリの値を書き換える必要がある時は rdx レジスタに読み出して処理を行ったのち、スタックに積み直すようにした。

## 2.5 その他の部分

コードの終了部分では、計算結果を返すために、講義資料で示された例に従って printf を呼び出している。また、資料の例ではプログラムの終了のために ret 命令を用いていたが、今回は exit を呼び出すことでプログラムを終了させるようにした。コードの終了部分の工夫として、スタックに積んでいた mem と countS をポップするために、pop 命令ではなく add 命令を用いてスタックポインタを移動させるようにしたことが挙げられる。スタックポインタが指すアドレスをデータ 2 つ分スタックの下に移動することにより、pop 命令を 2 回書かずともスタックの先頭 2 つのデータを削除することができる。

## 3 calc2

この章では calc2 の実装について述べる。calc2 では calc1 での実装をベースに、オーバーフロー検知や 0 割り検知を行うようにした。本章では、calc1 から変更された実装について述べる。

### 3.1 オーバーフロー検知

オーバーフローの検知は、演算子の処理部分で各演算子に対応する命令を実行した後と数値入力の処理のための命令を実行した後に `jo overflow` を使用することで行う。これにより、種々の計算によってオーバーフローフラグがセットされた時に、オーバーフロー時の対応を行うための処理ブロックにジャンプすることができる。

`overflow` ラベル以下での処理は以下の通りである。まず、`leaq` 命令によりエラーメッセージ (今回は単に E のみ) をセットし、`call` 命令で `printf` を呼び出してエラーメッセージを表示する。その後、`exit` ステータスとして 1 をセットした上で `call` 命令で `exit` を呼び出してプログラムを終了させる。

### 3.2 0 割り検知

0 割りの検知は、`idiv` 命令を実行する前に `cmpl $0, %ecx` と `je division_by_zero` を実行することで 0 割りの検知を行う。`cmpl` 命令を実行することで、`ecx` がゼロであればゼロフラグがセットされるので、その次の `je` 命令で条件分岐することができる。

`division_by_zero` 以下の処理は、オーバーフロー検知時と同様である。

## 4 calc3

この章では calc3 の実装について述べる。calc3 では、calc1 の実装をベースに `imull` 命令と `idiv` 命令を用いずに乗除を表すことを目指した。本章では、calc1 から変更された実装について述べる。

### 4.1 乗算の実装

`imull` を用いない乗算について、正数同士の掛け算と符号の処理に分けて実装した。

まず、正数同士の掛け算の実装について説明する。この実装では、`ecx` に格納されている乗数と `eax` に格納されている被乗数を用いて、ビットシフトとキャリーフラグを用いることで乗算を実現している。少し込み入った手順であるため、以下に番号付き箇条書きで用いた命令とその意図について述べる。

1. `rcll $1, %ecx`: 乗数 `ecx` を 1 ビット右回転させる。この操作により、乗数の最下位ビットがキャリーフラグ (CF) に移動する。これは、掛け算の筆算での、下一桁を取り出して、その掛け算を考えるという動作を実現するためである。
2. `jnc 3f`: CF がクリアであれば、次の加算処理をスキップする。これは、乗数の最下位ビットが 0 の場合は被乗数に 0 を掛ける処理をすることになり、そのような処理は最終的な値に影響を与えないからである。
3. `addl %eax, %edx`: CF がセットされている場合 (乗数の最下位ビットが 1 の場合)、結果を保持する

レジスタ `edx` に被乗数 `eax` を加算する。

4. `shll $1, %eax`: 被乗数 `eax` を 1 ビット左シフトする。これは、乗数の次の桁に対応するための処理である。2 進数では、桁が 1 つ左にシフトすると値が 2 倍になるため、乗算における桁上りを表現できる。筆算で言えば、桁ごとの掛け算の結果ごとに一つずつ位をずらして書くという操作に相当する。
5. `testl %ecx, %ecx and jnz 2b`: 乗数 `ecx` が 0 になるまで、ステップ 1 から 4 を繰り返す。`ecx` が 0 になるということは見るべき桁がなくなったこと、つまり乗数の全てのビットを処理したことを意味する。

続いて、符号処理の実装方法について考える。`calc2` までのように先んじて乗数の符号を変えてしまうと、2 の補数表現を用いている関係でうまくいかないケースが出てきてしまうと考え、乗数と被乗数の符号をまとめて処理することにした。具体的な実装としては、まず、`test` 命令を用いることで、被乗数 `eax` の符号をチェックする。もし被乗数が負の数であれば、`negl` 命令を用いて被乗数を正の数に変換した上で、`countS` をインクリメントする。そして乗算が終了したタイミングで、計算結果の正負を `countS` に応じて調整する。こうすることで、被乗数の正負の情報を `countS` に渡し、まとめて処理することができる。

以上のような `imull` を用いない乗算の実装のポイントは、乗数の最下位ビットを見るために `rcr` 命令を用いたことである。これにより、例えばシフトしてから `and` 命令を用いるなどの余分な動作をせず、短いコードで確実に乗算を行えるようになった。また、掛け算の筆算における、各桁ごとの掛け算の結果を 1 桁ずつずらして書いて最後に足しこむ、という動作を被乗数を左シフトすることにより実装したことは、かなり直感的で分かりやすい実装であると考えている。

また、被乗数と乗数の符号の情報を `countS` にまとめ、最後に処理するという実装も、最も大変な掛け算の実装のロジックに複雑な場合分を入れなくて済むという点で優れている。

## 4.2 除算の実装

`idivl` を用いない除算について、乗算と同様に正数同士の割り算と符号の処理に分けて実装した。

まず、正数同士の割り算の実装について説明する。この実装では、`eax` に格納されている被除数、`ecx` に格納されている除数を用いて、ビットシフトと減算を用いることで除算を実現している。以下に番号付き箇条書きで用いた命令とその意図について述べる。

1. `xorl %edi, %edi`: 部分剰余を格納するレジスタ `edi` を 0 クリアする。
2. `movl %ecx, %edx`: 除数 `ecx` の値を `edx` にコピーする。`ecx` はループカウンタとして使用するため、除数の値を保持するために `edx` を用いる。
3. `movl $32, %ecx`: ループカウンタ `ecx` に 32 を設定する。カウンタを 32 にするのは、除算を行うために 1 ビットずつ見ていく必要があるからであり、かつ今回は 32 ビット整数を扱うからである。
4. `xorl %esi, %esi`: 商を格納するレジスタ `esi` を 0 クリアする。
5. `shll $1, %eax`: 被除数 `eax` を 1 ビット左シフトする。
6. `rcll $1, %edi:eax` の最上位ビットをキャリーフラグを介して部分剰余 `edi` に移動させる。これは、部分剰余に被除数のビットを 1 つずつ取り込んでいく処理であり、これは割り算の筆算で上の桁から一桁ずつずらして商が立つところを探す動作に相当する。
7. `shll $1, %esi`: 商 `esi` を 1 ビット左シフトする。これは、割り算の筆算で数字を立てる桁を一つずつ下にずらしていく動作に相当する。

8. `cmpl %edx, %edi`: 部分剰余 `edi` と除数 `edx` を比較する。
9. `j1 3f`: 部分剰余が除数より小さい場合、次の減算処理をスキップする。
10. `addl $1, %esi`: 部分剰余が除数以上の場合、商 `esi` に 1 を加算する。これは、割り算の筆算で数字を立てることに相当する。
11. `subl %edx, %edi`: 部分剰余 `edi` から除数 `edx` を減算する。
12. `decl %ecx`: ループカウンタ `ecx` をデクリメントする。
13. `testl %ecx, %ecx and jnz 2b`: ループカウンタ `ecx` が 0 になるまで、ステップ 5 から 12 を繰り返す。

続いて、符号処理の実装方法について説明する。乗算の場合と同様に、被除数と除数の符号をまとめて処理するために、被除数の符号を `test` 命令でチェックし、負の数であれば `negl` 命令で正の数に変換し、`countS` をインクリメントする。そして除算が終了したタイミングで、計算結果の正負を `countS` に応じて調整する。

以上のような `idiv` を用いない除算の実装のポイントは、乗算の場合と同様に、`rcl` 命令によるシンプルな手法に被除数の値を見ることができる点である。また、これらの動作は割り算の筆算の手法を直感的に表現しており、理解しやすい実装であると考えている。

## 5 反省

- 命令やレジスタが本当に高速になっているのかを計測して確かめていない。