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 19 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，より良い実装方法の考察など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 言語による電卓の実装について述べる。

1.1 全体の実装

入力された式については，値を `p` というポインタ型の変数に格納して先頭から 1 文字ずつ `while` で読んでいくことにより，電卓の入力を 1 文字ずつ受け付けていくことを再現した。

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`，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。

ここで，全体の処理の流れを擬似的なフローチャートとして図 1 に示しておく。

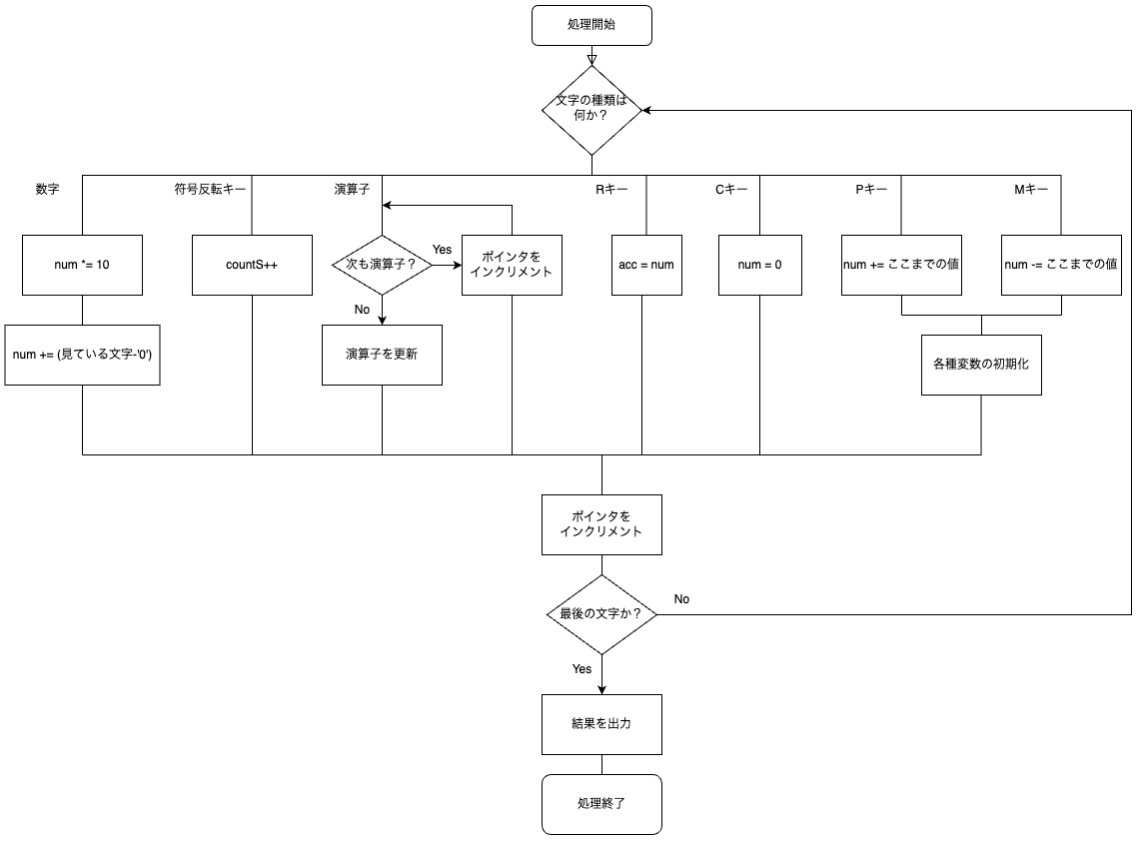

図 1　電卓コンパイラの全体の処理の流れ

以下，入力要素の種類ごとに，それぞれに対して行う処理について詳しくおねべておく述べる。

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， 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，最後に入力された演算キーによってそれ以前の演算子の入力は上書きされる，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結果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，countS が偶数の時には num にマイナス 1 をかけて符号を反転し，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。

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。

1.1.4 符号反転キーの管理

符号反転キーについては，S が来るたびに都度 countS をインクリメントするようにした。ただし，演算子の後ろにきた符号反転キーは無視する，という使用を実現するために，num が 0 の時は入力を無視するように，つまり countS をインクリメントしないようにした。

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。

また，計算処理の部分は関数として main 関数から切り分けたのだが，これによりコードの可読性が向上した。後述する calc1-3 についても，計算処理の部分は重複しがちな上アセンブリが最も複雑で長くなる部分だったので，計算処理のためのアセンブリを出力する部分を関数として切り分け，コード長を短縮して理解しやすいものにすることができた。

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衝突してしまう，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うなどといった問題が生じてしまった。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，スタック上の値を間接アドレッシングで参照するようにしたりした。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，意味として似通っている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，そのような操作を行う多くの部分において，単に `mov` 命令で即値 0 を代入するのではなくて `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ので，calc0 通りの実装のまま C 言語で有効な演算子を取得するようにした。

それ以外の部分の処理，つまり演算子に基づく計算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，すなわち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ことができ，それゆえ符号反転キーが奇数回押されている時のみ ecx に入っている値の符号を反転させることができる。以上のようにして，符号反転キーの処理を行った。

なお符号処理の部分でジャンプ用のラベルとして”1”というのを用いているが，これは数値ラベルである。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. `rcrl $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，乗算における桁上がりを表現できる。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。また，この命令により CF はクリアされる。

上記の動作について，図 2 に示すような例 ($11 \times 5 = 55$) を用いて説明する。

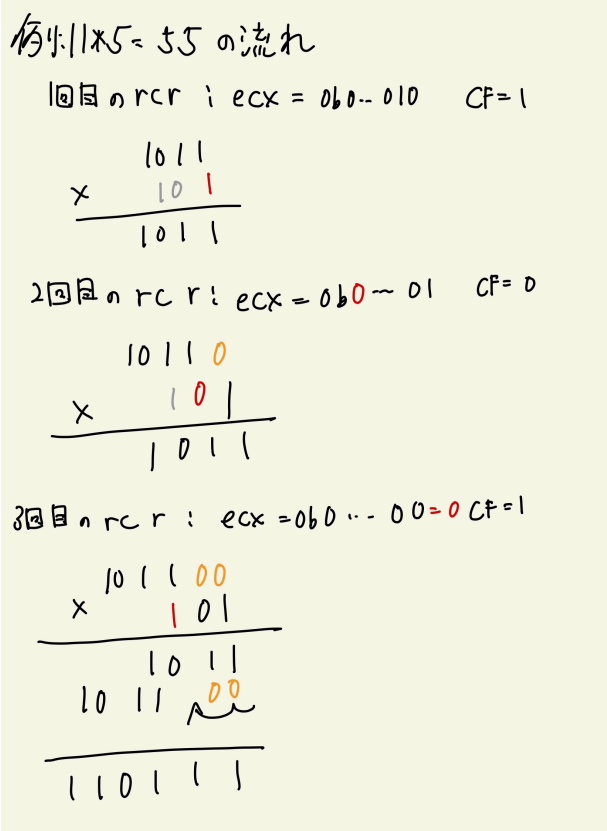

図 2　ビットシフトによる掛け算の動作例

まず，1 回目の `rcr` 命令により，乗数の 1 桁目の数字がわかる。この場合，乗数の 1 桁目は 1 であるため，`addl` 命令により被乗数を `edx` に加算する。また，それに続いて `shl` 命令により被乗数を 1 ビット左シフトしておく。

2 回目の `rcr` 命令により，乗数の 2 桁目が 0 であるということがわかり，`addl` 命令はスキップされる。また，2 回目の `rc` 命令の前に `test` 命令を実行しているため，キャリーフラグがリセットされている故，`ecx` の最上位ビットは 1 ではなく 0 になっていることに注意する。その後，`shl` 命令により被乗数を 1 ビット左シフトしておく。

3 回目の `rcr` 命令により，乗数の 3 桁目が 1 であるということがわかり，`addl` 命令により被乗数を `edx` に加算する。ここで，これ以前に被乗数は 2 回左シフトされているが故に，足し込まれる値も最初と比べて 2 ビット分左にずれており，これが筆算の位取りを表していることがわかる。

今回は 3 回 `rcr` 命令を実行すると `ecx` が 0 になるので，ここで掛け算の作業は終了し，答えとして $110111_2 = 55$ が得られる。

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 edi に移動させる。これは，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:edi と除数 edx を比較する。
9. jl 3f: edi 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:edi 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: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。また，この命令により CF はクリアされる。

上記の動作のうち，割り算のロジックに直接関連するステップ 5 以降について，図 3 に示すような例

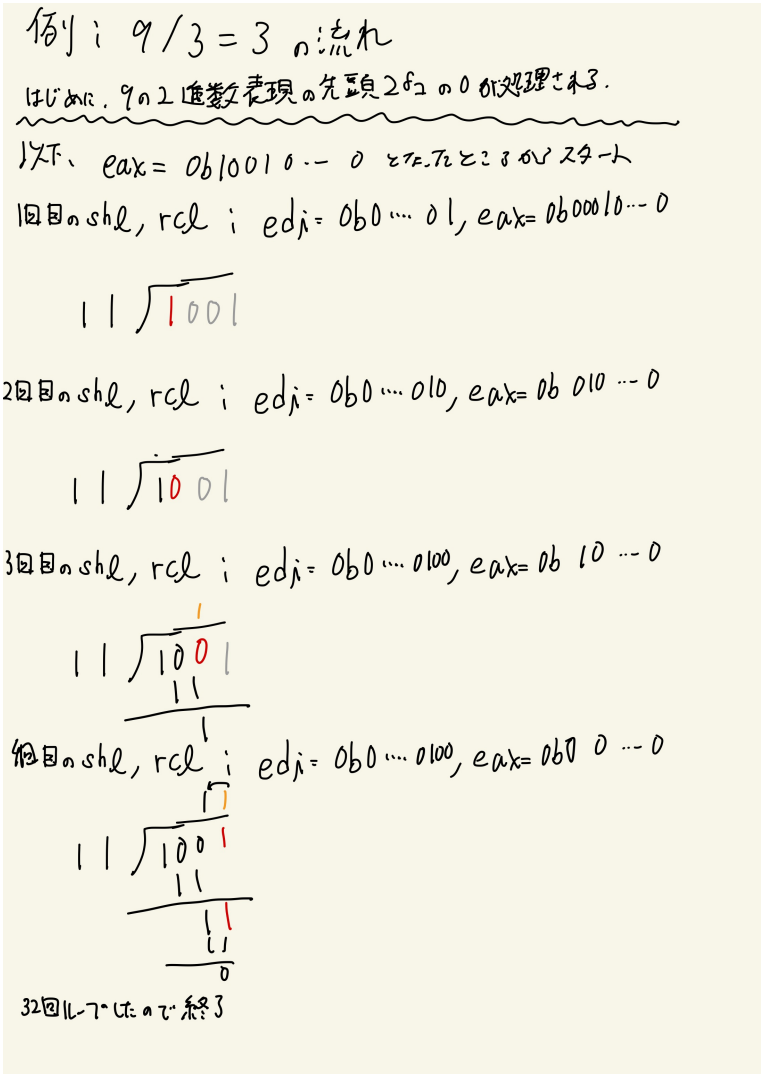

図 3　ビットシフトによる割り算の動作例

$(9/3 = 3)$ を用いて説明する。

ここで，$9 = 0\mathrm{b}0 \cdots 01001$ であり，9 の二進数表記の先頭 28 個は 0 である。これは `shl` 命令が 28 回実行されることにより，特に結果に影響を与えずスキップされる。なぜならば，先頭の 0 を取り出してそれを `edi` に移している間，`edi` の値は 0 で，常に除数よりも小さいためである。よって，以下の説明では 28 回ループを回した後から説明する。

まず，29 回目の `shl` 命令と `rcl` 命令により，CF を通じて 1 が移動し，`edi` の値が 1 となる。この時，まだ除数の方が大きいため，減算処理はスキップされる。このような動作は，割り算の筆算を行う時に商に数字が立つかどうかを上から一桁ずつ見ていく動作に相当する。その後，`shl` 命令が実行される。これは，そこまでに立てていた商を 1 つシフトすることで，位取りの動作を再現するためである。

次に，30 回目の `shl` 命令と `rcl` 命令により，`edi` の値が 2 となる。この時，まだ除数の方が大きいため，減算処理はスキップされる。

次に，31 回目の `shl` 命令と `rcl` 命令により，`edi` の値が 4 となる。この時，除数が `edi` 以下になったので，

商に 1 を立てるために esi に 1 を加算する。その後，subl 命令により，edi から除数を減算する。これは，割り算の筆算において商に数字が立った時にそこまでの段階で見ている数から除数を引く動作に相当する。

最後に，32 回目の shl 命令と rcl 命令により，edi の値が 11 となる。この時，除数が edi 以下であるので，商に 1 を立てるために esi に 1 を加算する。その後，subl 命令により，edi から除数を減算する。

これによりループが終了し，結局答えとして $11_2 = 3$ が得られる。

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　より良いテストケースの提案

この章では，私がテストケースに加えた変更について述べる。

まず，calc3 ではビットシフトによる掛け算や割り算の実装をしたので，そのロジックがうまく機能していることを確かめるために，掛け算や割り算を用いる計算のテストケースを増やした。具体的には，符号キーがつく場所のパターンを複数にして見たり，オーバーフロしないギリギリの大きさになるような計算を行うテストケースを追加した。

また，今回はテストケースを仕様として実装を進めてきたため，無意識のうちに与えられたテストケースだけを満たすような実装をしてしまっている可能性があると考えたため，ランダムなテストケースを生成するプログラムを作成した。具体的には，生成 AI を活用し，ランダムな数値と演算子を出力し，その演算結果を Python で計算し，testcases1.txt や testcases2.txt に記載されているテストケースと同様の形式でファイルに書き込むプログラムファイルを作成し，それでランダムなテストケースを作成した。このようにしたことで，配布されたテストケースや自分で考えて作成するテストケースだけに頼る場合よりもプログラムの信頼性を高めることができたと言えるだろう。

6　全体の実装についての反省・考察

この章では，私の実装を振り返り，その良し悪しや改善すべき点について考察する。

初めに，良かった点について述べる。まず，calc1 から calc3 までそれぞれに課せられた課題を達成することができた (少なくとも配布 + 自分で設定したテストケースが全て通った) ことについては満足している。既知のバグなども存在していない。

また，calc3 の乗算や除算の実装において，ローテート命令を適切に活用することにより掛け算/割り算の筆算を直感的に表せたことについては，比較的よい実装ができたと言えるだろう。

一方で，このコードには課題が多くある。第一に，レジスタとスタックの使い方に課題があるだろう。今回の私の実装では，累積値と途中に入力された数値はレジスタで管理しており，また演算子については C 言語で管理していた。しかしながら，講義の資料で紹介された木構造による計算式の処理では入力された記号を木構造で管理し，数値を push し，記号が来たら pop してその記号に応じた計算をするというような実装をしていた。

私の実装は今回のようにコードが行う動作が限定的であれば問題ないが，より複雑な作業を行ったり複数の作業をする必要があったりする場合には，レジスタの数が不足してしまい，不足するたびにスタックに値を積む必要が生じ，スタック上の値の管理が煩雑になってしまうだろう。一方で講義で紹介された実装であれば，常に記号が来るたびに 2 つの数を pop して計算しその結果を push するという処理を繰り返すだけでよいため，無駄なレジスタを使用することがなく，またスタックの管理も簡単になるであろうと考えられる。よって，授業で紹介された方法の方が，拡張性という点で私の実装よりも優れていると考えられる。

第二に，割り算の実装について，仕様は達成しているものの，小数の計算やあまりを求める処理を扱っていないという点において通常の電卓に劣っている。あまりの処理については，calc1,calc2 については `idiv` 命令ではあまりが edx レジスタに格納されることを利用すればよい。また，calc3 ではレジスタ `edi` で割られる数を保持しているので，最終的なあまりはこのレジスタに格納された値ということになる。これらをうまく実装することであまりの計算も実装できそうであるが，形にするのに十分な時間がなかった。これは次回以降の課題として取り組むべきである。小数の計算については，それ専用の命令を使うことが必要そうであるが，これについてはまだ知識が不足しているため，知識を身につけるところから始めていきたい。

第三に，コードの実行速度についても疑問がある。例えばレポート中で，`xor` 命令による 0 クリアは `mov` 命令によるものよりも高速であると知られている，というように述べたが，これは実際に計測した結果ではなく，環境によって異なる可能性も十分に考えられる。よって，よりより設計をするためには計測が不可欠であると考えるが，私は本課題でそれを怠ってしまった。文献や口伝による情報を信じるばかりでなく，実際のユースケースにおける計測を行うことが重要であるのだから，それをするべきであった。

第四に，前に述べた内容と関連する部分もあるが，私のコードはまだ C 言語に依存している部分が多いと感じる。例えば式の入力や演算子の管理は C 言語で行なったり，C 言語で出力するアセンブリの条件分岐を行ったり，関数呼び出しを行ったりしている。これらの部分をアセンブリで実装することができれば，より実際の C コンパイラに近いものを作ることができるだろう。

7　感想

この授業で初めて本格的にアセンブリに触れたが，プログラミング言語が裏でどのようにコンピュータが解釈できるように変換されているのか，ということをほんの少しではあるが理解することができた。特に，ビット演算を用いた乗算や除算の実装をしたことにより，数値やその正負が根本的にはどのように表現されているかということを深く考えることができ，まるでコンピュータになったような気持ちで演算の設計をすることができた。

しかし，私がこのコードで扱ったアセンブリ命令はそれほど複雑なものではなく，またレジスタやスタックの適切な利活用をしているとは到底言い難いものになってしまった。他の人のコードを読んだり，ドキュメントを少しずつでも読み込んだりすることにより，効率的でかつ言語の思想にきちんと従っている，あるいは世間で一般的とされているお作法に従っているアセンブリの書き方を学んでいきたい。

それに関連して，課題の提出期間が終わった後でも良いから他の人の実装を見ることができると自らの実装を反省したり到底思い付かなかった実装をみられたりして嬉しいなと思った。