アセンブリ言語

20B30790 藤井 一喜

概要

以下では`calc1.c`, `calc2.c`, `calc3.c`のそれぞれについて、どのような発想のもとで提出したコードのような実装に至ったのかについて記述する。

`calc1.c`

コード自体は`calc2.c`と同様であるので、ここでは`calc1.c`の課題要件を満たすために必要な箇所についてのみ説明する。

入力された文字列の処理について

標準入力から受け取った文字列を`null`文字になるまで`while`ループで読み込み続けるという形で実装した。この際、`int state`という変数を用意し、`0`と`1`の２値で、演算キーを処理した後なのか、そうではないのかを判別できるようにした。読み込んだ文字(char型)の扱いについては、`if`, `else if`を用いて想定される文字ごとに合わせた処理を行えるようにしたが、想定されない文字については、処理を施さずにポインタ変数`p`をインクリメントすることで無視することにした。

計算処理について

現在の計算結果(=acc)を保持しておくために`%r8d`レジスタを、演算子が施される前の数字(=num)を保持しておくために`%r9d`レジスタを、memory機能のために`%r10d`レジスタを使用することにした。演算子を作用させる箇所については、演算子(=`last_op`)に合わせて処理を行う関数である`calc`を用意した。`calc`関数内部では、掛け算、割り算などの`imull`,`idivl`などにおいて特定のレジスタに値を置く必要がある演算に関しては、その動作に合わせた処理を記述した。具体的には、掛け算の場合は、`%eax`レジスタに`%r8d`(=acc)の値を`movl`にてコピーしたのちに`imull %r9d`が行われるようにすることや、割り算の場合は`%r8d`(=acc)の正負に合わせて、符号反転処理を行い、`%eax`レジスタにコピーされる値が常に正になるような処理を記述した。なお、前述の符号反転処理を行う場合は、割り算命令を施す前に割る数である`%r9d`に対しても符号反転処理を行うことで辻褄合わせを行なうようにしている。

メモリ機能について

メモリ機能の実装のために、`%r10d`レジスタに値を保存しておき、`*p == 'p'`や`*p == 'M'`の際には、`%r8d`レジスタ(=acc)と適切な演算を行うようにした。

0除算について

`idivl`の前に、割る数が0かどうか判定し、0であったらEと出力し終了する動作を実現するために`print_E_floating_exception`という関数を作成している。

ラベル

条件分岐の際に使用するラベルが重複することがないように、global変数として`int cnt = 0; int count = 0;`を定義している。`cnt`と`count`の区別は、オーバーフローと０除算が存在するかどうか判別し、存在する場合は`E`

を出力して終了するというアセンブリコードを出力する箇所においてのみ使用する変数が`count`で、それ以外の割り算を実現するための箇所と、値を`%r9d`に読み込んでいく箇所の計算処理に使用しているのが`cnt`である。

結果

`test1.csh`でのテストは、問題なく通過した。

calc2.c

`calc1.c`のsectionにて説明した箇所以外でcalc2.cの課題要件を満たすために必要な箇所について説明を加える。

オーバーフロー検知

オーバーフローが発生すると考えられる、足し算、引き算、掛け算、割り算、符号反転などを行うアセンブリコードの直後に、オーバーフローフラグ(以下OF)がたっているかどうかを判定し、OFがたっている場合は、printfをcallし`E`を出力して終了するという一連の動作を実現するアセンブリコードを出力する関数`print_E`を置くことで、オーバーフローの発生を検知するようにした。

ゼロ除算の検知については、`calc1.c`と同様

結果

`test2.csh`でのテストは、問題なく通過した。

calc3.c

乗算命令

`imull`命令の代わりに、`imull`と同様の結果をもたらすことができるアセンブリコードを出力する関数`mul`を`imull`の代わりに呼ぶことで乗算を計算する。 自作の`mul`関数では、ローテート命令`rorl`を用いて、かける数を下位ビットから1ビットずつ処理することで掛け算命令を実現している。具体的には、`rorl`命令の後に、`CF`がたっているかどうかを判定し、立っている場合は、`%eax`レジスタに`%r11d`の値を足すという処理を行う。またかける数または、かけられる数の片方もしくは両方が負の数である場合は、符号反転処理などを適切に行い、絶対値同士の乗算を求め、最後に辻褄合わせのために再度符号反転処理を行うという処理を挟んでいる。なお、かける数、かけられる数は32bitなので、`for`文で、32回分の一連の処理を行うアセンブリコードを出力するようにして実現している。

除算命令

`idivl`命令の代わりに、`idivl`と同様の結果をもたらすことが可能なアセンブリコードを出力する関数`div`を`idivl`の代わりに呼ぶことで除算を計算する。 自作の`div`関数では、`%r9d`レジスタに`割る数`を、`%r13d`レジスタに`割られる数`を、`%r11d`レジスタに`割られる数の一時保管（シフト処理が施される側）`を、`%r12d`レジスタに`答え(商)`をいれることで実装している。

`div`関数の中では、`%r9d`(=割る数)が正か負かで場合わけを行なっている。これは、負の数が割る数に存在する場合は、絶対値で計算したのち符号反転処理を商に対して行うことで辻褄合わせを行うためである。なお、`%r11d`レジスタに格納される値は、`calc1.c`の処理の段階で必ず正となることが確定しているので、こちらについては場合わけをする必要はない。

割り算命令に相当する部分の詳細な実装は、32bit整数の割り算であることから1bitずつ上位ビットから`%r11d`の値をシフトし、その際CFがたっているならば、`%r13d`(=割られる数の現状値)に1を足し、その値が割る数である`r9d`レジ

スタに格納されている値以上かどうかで場合わけを行い、`r9d`レジスタに格納されている値以上である場合は、`%r13d`レジスタの値から`%r9d`レジスタの値を引き、そして`r12d`レジスタの値をインクリメントするによって実現されている。

ラベル

`imull`や`idivl`を作る際、場合分けが多数生じるので、ラベルが衝突しないように`int mul_cnt = 0; int mul_sign_count1 = 0; int mul_sign_count2 = 0;`や`int div_cnt = 0; int div_in_cnt = 0; int div_op_cnt = 0;`などのグローバル変数を定義し、適切なタイミングで`count += 2;`のような形で演算を施すことで、数字が被らないようにしている。

結果

`test1.csh`,`test2.csh`でのテストは問題なく通過した。

工夫点

`calc1.c`〜`calc3.c`のすべての課題において、値の保持機能（計算途中なども含む）は、すべて汎用レジスタを用いて実装されている。 これは、高速なアクセスを可能にするための工夫である。