コンピュータアーキテクチャ
第6回　入出力の演習

天野　hunga@am.ics.keio.ac.jp

演習6.1

入力装置から入力した数字をASCIIコードに直して出力するプログラムenshu61.asmを書け

ioディレクトリで行うこと

io.asmを改造すれば良い

test_mpise61.vをtest_mipse.vにコピーして使え（元のも取っておいた方がいい）

演習6.2

- lbu を実装せよ　この命令は8ビットのデータをゼロ拡張してレジスタに格納する点以外はlbと同じである
- opcodeは100100とせよ
- 提出物：改造したmipse.v
- ioのディレクトリで行うこと
- テストにはlbutst.asmを用いよ

演習6.3

- 割り込み処理ルーチンではメインルーチンで利用するレジスタを破壊しないために利用する全てのレジスタをスタックに保存しなければならない。
  - test.asmではフィボナッチ数列を計算している
  - testint2.asmでは入出力を行う
  - この両者がきちんと動作するために、testint2.asmにレジスタ保存と復帰のコードを付け加えよ
  - スタックポインタは$30とする

演習のコマンドは例題3と同じ intのディレクトリで行うこと

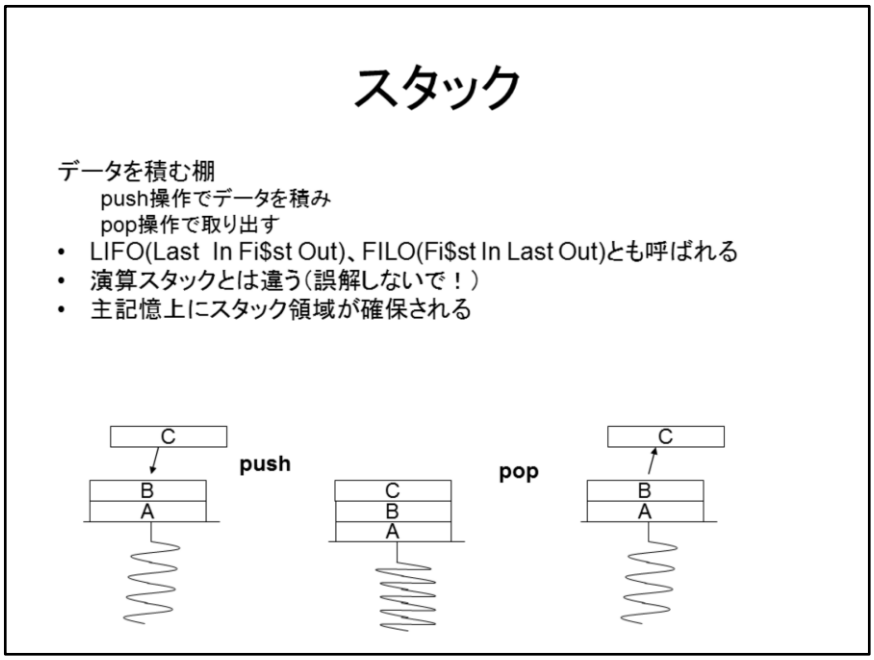
```
./asm.pl test.asm –o imem.dat
make intmem.dat
./a.out > tmp
```

例題3：割り込みのシミュレーション

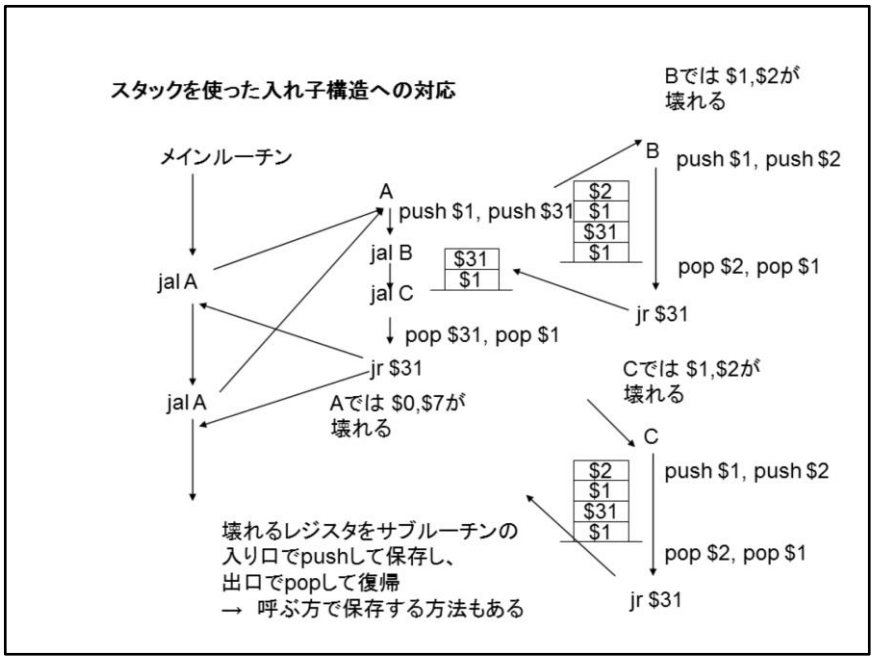
ta$ xvf int.ta$で割り込み演習用ディレクトリが生成される
make
./asm.pl test.asm –o imem.dat
make intmem.dat
./a.out > tmp
でtmpを見てみよう

test.asmは、フィボナッチ数列を計算するメインプログラム
$F_0=0, F_1=1, F_{n+2}=F_{n+1}+F_n$
順に$1,$2,$3に答が入る

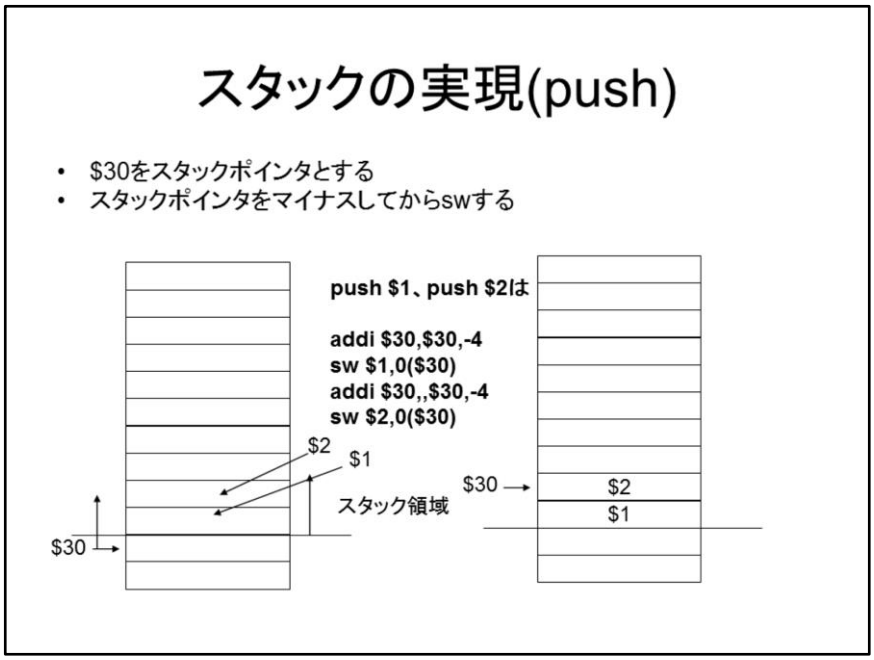
testint.asmは、例題2の1文字入出力プログラム



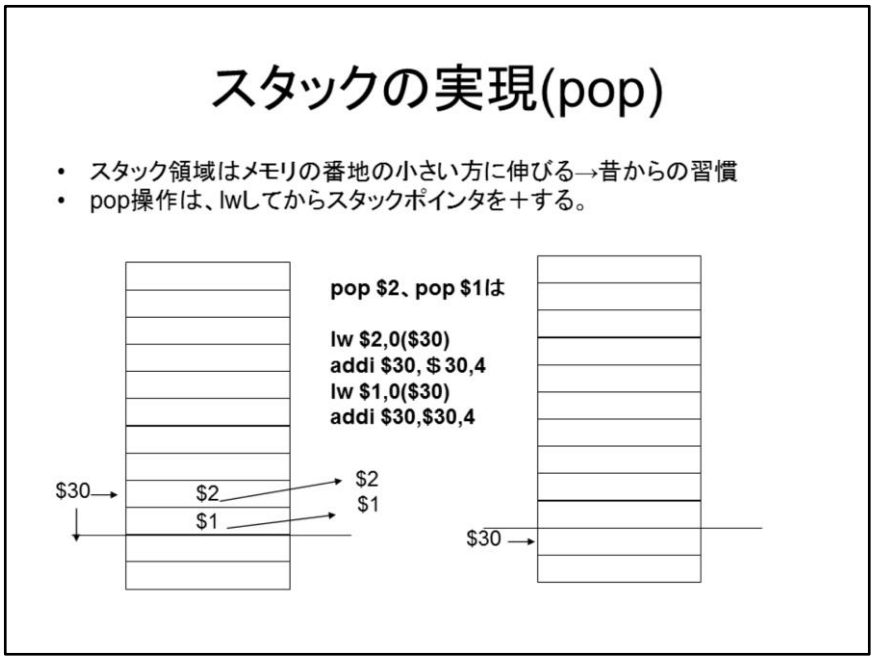
スタックとは、データを積む棚です。この棚にデータを積む操作をpush、棚から取り出す操作をpopと呼びます。先に積んだものが後から取り出されることからLIFO (Last In Fi$st Out)と呼びます。逆に考えると、後に積んだものが先に取り出されるのでFILO（Fi$st In Last Out)と呼ぶ場合もあります。この積んだ逆順に取り出すことのできる性質からサブルーチンコール時にレジスタを退避するのに適しています。
以前紹介したスタックマシンで利用した演算スタックは、演算用の特殊なメモリですが、サブルーチンコールのレジスタの退避用のスタックは主記憶上に確保するのが普通です。スタックは棚ですが、ばねがついているイメージがあります。データを積むときは押し込むイメージからpushと呼び、取り出すときは、飛び出すイメージからpopと呼ばれます。



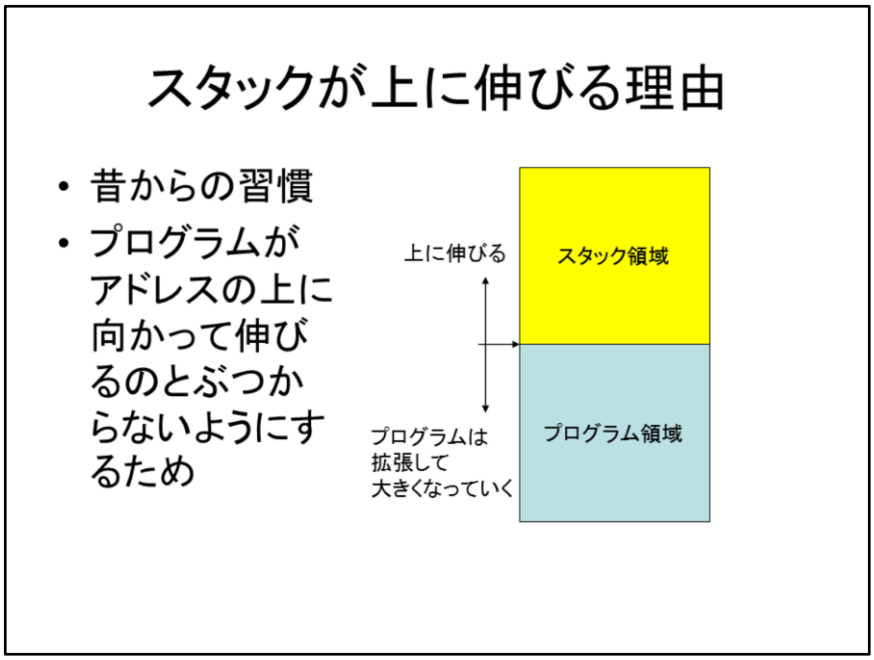
スタックを使ってレジスタを退避する様子を示します。この例では、サブルーチンの入り口で、中で使って壊れるレジスタを退避し、リターンする直前に復帰する方法を示します。これはコーリーサーブと呼びます。逆に呼ぶ側で、壊れて困るレジスタをスタックに積んでからサブルーチンを呼び出す方法（コーラーセーブ）もあります。サブルーチンAでは$1を使います。中で別のサブルーチンを呼ぶので$31も退避します。サブルーチンBを呼んだ際に$1、$2を退避します。この2つのレジスタはサブルーチンAを呼んだ際の$1、$31の上に積まれます。サブルーチンBの中でさらに別のサブルーチンを呼ぶ場合、さらにこの上に積み重なります。サブルーチンからリターンする直前に、pushしたのと逆順にpopします。そのようにすると、スタックの内容は呼ばれた時と同じになります。さらに別のサブルーチンCを呼んだ場合、サブルーチンの入れ子になった場合も同様に対処できます。再帰呼び出し（リカーシブコール）を行った場合も、スタックの容量が許す限り、スタックにレジスタを積み続けることができます。（再帰呼び出しのプログラムにバグがあるとセグメンテーションフォルトになるのは、スタックが溢れてしまうためです）
上記の説明はサブルーチンコール時のものですが、割り込み時も同じです。



スタックをメモリ上に実現するには、スタックポインタを使います。ここでは$30をスタックポインタの役割に使います。スタックポインタは、スタック領域の一番上の番地＋4の所に初期化します。push操作は、まずスタックポインタを減らし、空いた領域にレジスタを書き込みます。スタック領域はメモリ上の番地が減る方向に伸びていきます。この図はpush＄1, push＄2を順に実行した様子を示します。



逆にpop操作はまずスタックポインタの指し示す番地から取り出して、スタックポインタを増やします。図はpop $1, pop $0を順に実行した様子を示します。スタックポインタは最初の位置に戻ります。



スタックは番地の小さくなる方向に（この図では上に向かって）伸びるのが普通ですが、これは昔からの習慣に基づいています。昔はある場所に境界を引いて、それより小さい番地はスタック領域とし、大きな番地はプログラム領域としました。これはプログラムというのは開発を進めると大きくなるので、番地が増える方向に伸びます。スタックを番地の増える方向に成長させると方向が同じになり、成長したスタックがプログラム領域を食い荒らす可能性があります。そこで、スタックは番地が小さい方向に伸ばす習慣ができました。もちろん領域を食いつぶしてしまえば、どちらの方向でもトラブルになります。

演習のやり方

- testint2.asmをtestint.asmにコピー
- make intmem.datでintmem.datを生成
- フィボナッチ数列が、ちゃんと計算され、出力もちゃんと行われていることを確認

新たなtestint.asmを提出