目次

★1 サブルーチン呼び出しとスタック（つづき）

★1.1 サブルーチン呼び出しと復帰

前回のプログラム EX04X3 を右のように書き換えると，次のように期待通りの動作をするようになる．

1. 2 行目で A 番地の値を GR1 にロードし，3 行目で SAMBAI へ
2. 9 行目から 11 行目で GR7 に GR1 の 3 倍の値をセット
3. 12 行目の RET 命令を実行すると 4 行目へ飛ぶ
4. 4 行目で GR7 の値を AX3 番地にストア
5. 5 行目から 7 行目でも，上と同様に，6 行目で SAMBAI へ飛んで 3 倍の値を計算し，7 行目に戻ってくることができる

このように，いんちきアセンブリ言語では，CALL 命令でサブルーチンを呼び出し，RET 命令で復帰する（呼び出し元のすぐ次の命令に飛ぶ）ことができる．このような動作を実現するためには,「どこから呼び出されたか」を何らかの形で記憶しておく仕組みが必要である．また，その仕組みは，サブルーチン内からさらに別のサブルーチンを呼ぶような場合でもちゃんと復帰できるようになっていなければならない．

一般に，このような仕組みは，**スタック** (stack) と呼ばれるデータ構造を用いて実現される．

EX05X3

```
 1  EX05X3   START
 2           LD    GR1,A
 3           CALL  SAMBAI
 4           ST    GR7,AX3
 5           LD    GR1,B
 6           CALL  SAMBAI
 7           ST    GR7,BX3
 8           RET
 9  SAMBAI   LD    GR7,GR1
10           ADDA  GR7,GR1
11           ADDA  GR7,GR1
12           RET
13  A        DC    17
14  B        DC    -31
15  AX3      DS    1
16  BX3      DS    1
17           END
```

［注］プログラムの最後（上の例では 8 行目）も RET 命令である．これは，このプログラムそのものがオペレーティングシステム (OS) から呼び出されて実行を開始するので，最後に復帰するためである．

★1.2 スタックとスタックポインタ

スタック (☆ 1) とは…（板書して説明します）

サブルーチン呼び出しと復帰のためのスタックは，次のように実装されることが多い．スタックとスタックポインタの管理は CPU が行ってくれるので，プログラムの側はスタックがどうなっているかというようなことは気にしないで済む．

- メモリ中の特定の領域をスタック用と約束する
- スタック領域の番地を格納する**スタックポインタ** (SP) 用のレジスタを用意
- SP は最後にプッシュした値が格納された番地を指し，スタックが空のときは,「スタック領域の最大番地+1」を入れておく

☆ 1) スタックは，情報処理装置がデータを格納しておくやり方（**データ構造**）の中でも特に重要なものです．幅広い分野で登場するものなので，是非自分でどんなものか調べてみてください.「データ構造とアルゴリズム」等の授業で学びます．

この場合，プッシュとポップは次のような動作になる．

● プッシュ

1. SP をひとつ減らす
2. SP が指す番地に値を格納する

● ポップ

1. SP が指す番地の値を指定された場所にコピーする
2. SP をひとつ増やす

このようなスタックがあれば，前頁のプログラムの例のサブルーチン呼び出しと復帰は次のように実現することができる．

1. 3 行目の `CALL  SAMBAI` を実行．次のことが行われる．
   (a) PR がインクリメントされて次の命令（この場合は 4 行目の命令）の格納されている番地を指しているので，その値をスタックにプッシュする
   (b) ラベル `SAMBAI` に対応した番地（9 行目の命令の格納番地）を PR にセットする
2. PR の指す番地から命令をフェッチして実行（PR には 9 行目の命令の格納番地がセットされているので，9 行目の命令を実行することになる）．
3. 同様にして 10 行目の命令を実行．
4. 同様にして 11 行目の命令を実行．
5. 12 行目の `RET` を実行．次のことが行われる．
   (a) スタックからポップした値（4 行目の命令の格納番地）を PR にセットする
6. PR の指す番地から命令をフェッチして実行（PR には 4 行目の命令の格納番地がセットされているので，4 行目の命令を実行することになる）．
7. 同様にして 5 行目の命令を実行…

★ 1.3 ［発展］レジスタ値の退避

ここまでの説明では，スタックに積まれるのはサブルーチン呼び出しからの復帰先の番地のみであったが，実際のコンピュータでは，その他にも，汎用レジスタの値やフラグレジスタの値など，様々なものをスタックに積むことがある (☆2)．

例えば，サブルーチン呼び出しの前に汎用レジスタの値をスタックにプッシュしておき（レジスタの値を退避させるという），復帰直後にポップさせて復元する，という使い方をすることが多い．このようにしておけば，サブルーチン側が呼び出し元で使ってるレジスタの値をうっかり変更してしまわないよう気を使わなくてすむ．また，少ない汎用レジスタをやりくりして使うのも楽になる．

また，プログラムの実行途中に何らかの理由で実行を中断することになった場合には，オペレーティングシステムが中断時点での全てのレジスタの値（PR,FR,SP も含む）を退避させる．そうしておけば，そのプログラムに復帰する際にそれらのレジスタ値を復元して，何事もなかったかのように命令実行を再開することができる．

☆ 2) いんちきアセンブリ言語には，汎用レジスタの値をスタックにプッシュする命令 (PUSH)，スタックからポップさせる命令 (POP) が存在する

★2 性能評価

CPU の性能を評価する方法について考える．まずは，自動車を例にして,「性能」をどのように評価するか考えてみよう．自動車の場合,「人を移動させる速さ」,「燃料の消費量」,「価格」など様々な基準が考えられる.「人を移動させる速さ」に注目したとしても,「1 人の人間を移動させるなら，移動時間最短なのは HOGE-GT」（速度最大）,「30 人なら FugaX」（輸送能力最大）というように，見方によって評価が変わり得る．CPU の性能評価の場合も，基準の選び方次第で評価が変わり得るところは同じである．では，CPU の性能評価にはどんな基準を用いるのがよいだろうか.

車種・車名	定員 [人]	平均燃費 [km/l]	速度 [km/h]	輸送能力 [人・km/h]
スポーツカー HOGE-GT	2	5	200	400
普通乗用車 Hena5	5	20	150	750
大型バス FugaX	50	2	100	5000

★2.1 n-bit CPU という言い方とクロック周波数

n-bit CPU CPU を分類する際に,「これは 32-bit の CPU，あっちは 64-bit」という言い方をすることがある (☆ 3)．この「n-bit の CPU」という呼び方は，その CPU の汎用レジスタのビット長や CPU 内部のバスの幅などが n であり，n-bit のデータを効率よく処理できるよう設計されていることを表している．大雑把に言えば，n の大きなプロセッサほど性能が高い傾向があるが，CPU のアーキテクチャも様々なので，ビット長のみで評価することにはあまり意味はない．一方，n の大きな CPU は，一般に番地指定に使える bit 数が多くなるので大容量のメモリを扱えるようになるという利点をもっている (☆ 4).

クロック周波数（動作周波数） 現代のコンピュータに含まれるほとんどの回路は，**クロック**と呼ばれる周期的な信号に**同期**して（タイミングをあわせて）動作するように作られている．そのため，クロックの周波数を高くすれば，同じ時間内にできる処理の量を増やすことができる．したがって，全く同じ回路をより高いクロック周波数で動作させればより高い性能を期待できる．しかし，回路の作りが異なれば動作の仕方も異なるので，異なる CPU の性能を評価する際にクロック周波数のみを取り上げても意味がない.

［注］ クロック周波数と動作周波数 「1 秒毎に腕を上げ下げする（上げてたら下げ，下げてたら上げる）」人にタイミングを合わせて別の人がジャンプする，という場面を考えてみよう．腕を上げ下ろしする人がクロック信号を作っており，それに同期して別の人が動作するわけである．この場合，腕の上げ下ろしという動作が一巡りするには 2 秒かかるので，このクロック信号の周波数は 0.5Hz ということになる．一方，ジャンプ動作の周波数の方は，腕が上がった瞬間だけ（または下がった瞬間だけ）に合わせてジャンプするならクロックと同じ 0.5Hz であるが，腕の上がるタイミングと下がるタイミングの両方でジャンプするなら 1Hz ということになる．このように，クロック信号の周波数と回路の動作周波数は必ずしも同じではない．しかし，計算機アーキテクチャ関係の教科書や資料等では両者の違いを気にせず,「動作周波数」を使うべき所で「クロック周波数」という用語を使って説明している資料も多いので，この授業でもそのようにしている.

☆ 3) 近頃の CPU の用途を分類してみると（注：いいかげんな分類です）

- 8or16 bit – 家電製品などの組み込み用途や，一昔前の携帯ゲーム機
- 32 bit – PC や, 携帯電話・プリンタなどの比較的性能が要求される組み込みシステム
- 64bit – 大量のデータを処理するサーバなど

☆ 4) いんちき計算機は 1 語 8bit で番地を表現するので，特別な工夫なしで接続可能なメモリは最大 256 語分だった．いんちき計算機 II は 1 語 16bit なので，最大 65536 語分だった.

★ 2.2 実行時間とスループット

CPU のビット数もクロック周波数も，性能評価の基準としては使えないことがわかった．実際にコンピュータシステムの性能を評価する際には，個々のユーザの立場では**実行時間**に注目し，多数のユーザが同時使用するようなサーバを管理する管理者の立場では**スループット**に注目することが多い．

実行時間: execution time.
スループット: throughput.

- **実行時間**（応答時間）: 1 つの作業を開始してから終了するまでの時間
- **スループット**: 一定時間内にどれだけの作業をこなせるか

一般に，これらの一方を変えると他方にも影響が及ぶ．例えば，

- システムの CPU を高速なものに取り替えたり，スーパーのレジ担当者を手際のよい人に替えると，実行時間が短縮し，スループットも改善する
- 複数プロセッサを使えるシステムにプロセッサを追加したり，スーパーのレジの数を増やすと，スループットのみ改善する (☆ 5)

☆ 5) ただし，(処理能力) < (仕事量) の場合には仕事があふれて待ち行列に入っており，スループットが改善するとそれらが待ち行列にとどまっている時間が短くなるので，結局実行時間も短縮されることになる．

これからは主に実行時間を基準として CPU の性能を考えていく．

★ 2.3 CPU 実行時間

現代のコンピュータは**タイムシェアリング方式**で動作し，CPU は短時間の間に複数のプロセスを切り替えて実行することが多い．そのため，1 つのプロセスの起動から終了までに実際にかかった時間（**経過時間**）を測っても，その中には他のプロセスの実行に費やした時間や入出力待ちに要した時間も含まれてしまう．

経過時間: elapsed time

そういうことのないように正しく CPU の性能を評価するためには，CPU が該当プロセスの実行のみに費やした時間を測定する必要がある．この時間のことを，**CPU 実行時間**あるいは **CPU 時間** という．

CPU 実行時間: CPU execution time

UNIX 系の OS では，`time` コマンドを用いておよその CPU 時間を測定できる（以下の `user` あるいは `user+sys`）．

```
$ time  ./a.out      ← 実行ファイル名の前に time をつけて実行
       :
「a.out の出力」
       :
real   2m39.000s     ← 経過時間 2 分 39 秒（このプロセスの実行開始から終了までに現実に経過した時間）
user   1m30.700s     ← ユーザ CPU 時間 90.7 秒（このプロセスの実行に要した CPU 時間）
sys    0m12.900s     ← システム CPU 時間 12.9 秒（このプロセスの要求で OS が仕事を実行した時間）
```

あるプロセスを実行するのにかかる CPU 時間が，マシン A では 10 秒，マシン B では 15 秒であったとすると，実行時間の比は 1.5 である．このとき，「A は B の 1.5 倍速い」という言い方をすることがある．