計算機構成論 第7回
―命令の実行(1)―

大連理工大学・立命館大学 国際情報ソフトウェア学部

大森 隆行

講義内容

- 命令の実行
  - 手続きの実行とスタック

手続きの実行方法

- 手続き(procedure)
  - プログラムの一部を構成する一連の処理
  - C言語だと「関数」

```
int main(){
   int result = function(100, 200);
   ...
}
```

関数呼び出し　　実引数

戻り値(の型)　関数名　　仮引数

```
int function(int a, int b){
   return a + b;
}
```

関数の定義

※戻り値： 返り値とも言う

手続きの実行方法

- 手続きを実行するために必要なこと
  - 手続きからアクセスできる場所に引数の値を置く
  - 手続きに制御を移す
  - 手続き内部での処理に必要なメモリ資源を確保する
  - 手続き内部の処理を実行する
  - 呼び出し元からアクセスできる場所に結果(戻り値)を置く
  - 制御を元の位置に戻す

手続きの実行方法

- 手続きを実行するために必要なこと
  - 手続きからアクセスできる場所に引数の値を置く
  - 手続きに制御を移す
  - 手続き内部での処理に必要なメモリ資源を確保する
  - 手続き内部の処理を実行する
  - 呼び出し元からアクセスできる場所に結果(戻り値)を置く
  - 制御を元の位置に戻す

$a0-$a3

jal命令でPC, $ra操作

メモリ内

$v0-$v1

jr $ra

手続き実行のためのレジスタ

$zero	0	常にゼロ
$at	1	アセンブラが一時的に使用
$v0-v1	2-3	戻り値用
$a0-a3	4-7	引数用
$t0-t9	8-15, 24-25	一時レジスタ (一時変数用)
$s0-s7	16-23	退避レジスタ (変数用)
$k0-k1	26-27	OSカーネル用に予約
$gp	28	グローバルポインタ
$sp	29	スタックポインタ
$fp	30	フレームポインタ
$ra	31	リターンアドレス

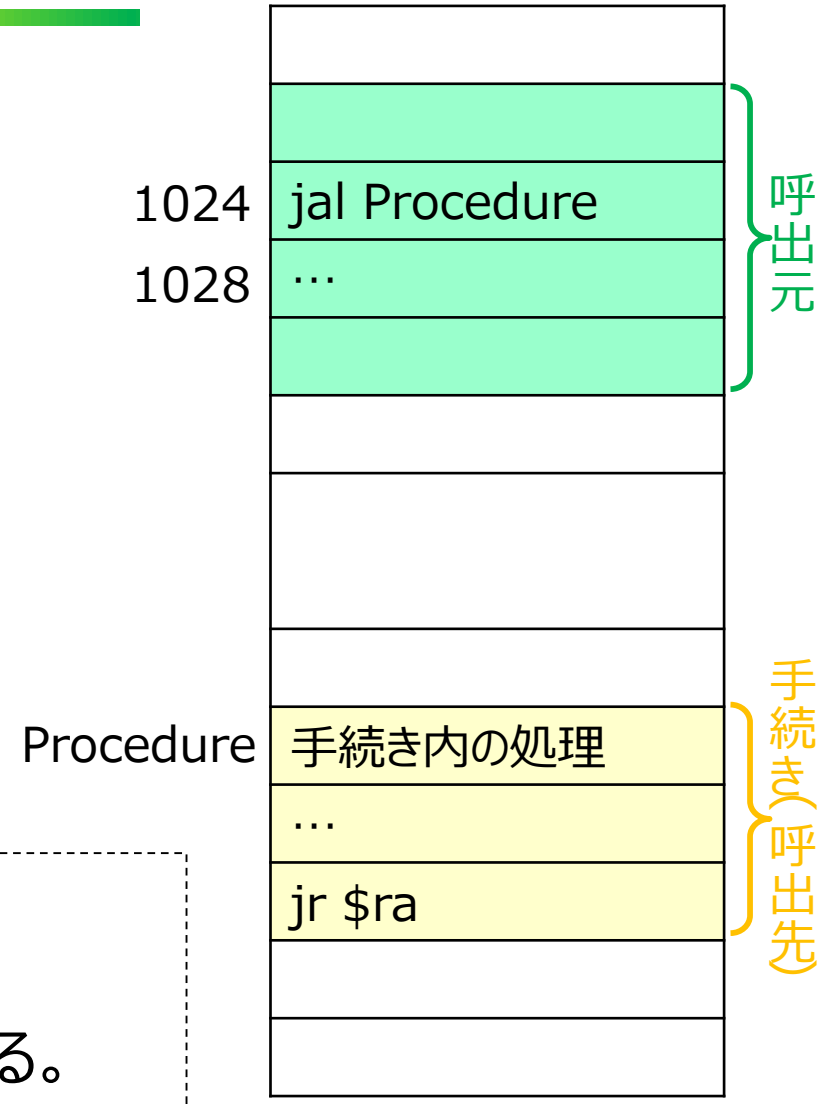
手続きの実行

`jal Label`

…jump and link
指定されたアドレスにジャンプすると
同時に、次の命令のアドレスを
レジスタ$raに保存　(J形式)

`jr $ra`

…jump register
指定されたレジスタに格納された
メモリアドレスにジャンプ (R形式)



右の例だと、1024番地のjal命令が
実行された後、$raは **1028** になる。
PCは1024の次は、**Procedure** になる。
手続き内のjr命令の後、PCは **1028** になる。

手続きの実行方法

引数が4個以上の場合は？

- 手続きを実行するために必要なこと
  - 手続きからアクセスできる場所に引数の値を置く
  - 手続きに制御を移す
  - 手続き内部での処理に必要なメモリ資源を確保する
  - 手続き内部の処理を実行する
  - 呼び出し元からアクセスできる場所に結果(戻り値)を置く

$a0-$a3

jal命令でPC, $ra操作

戻り値が2個以上の場合は？

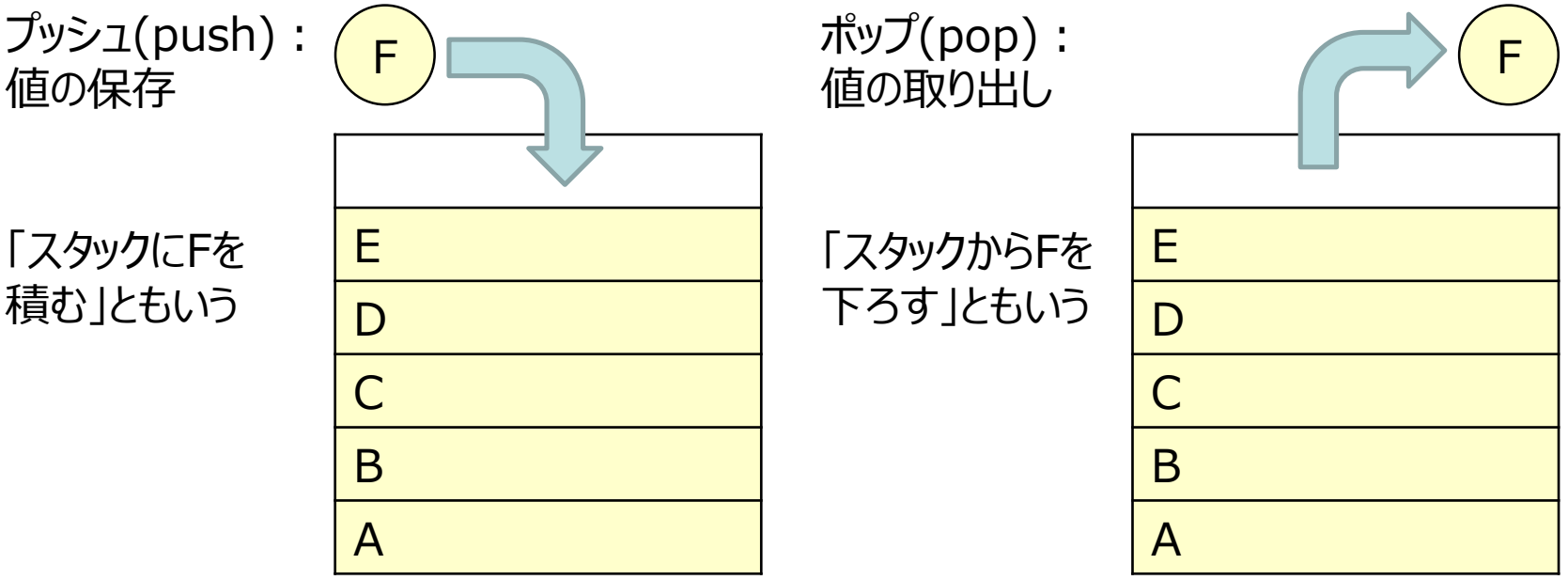
$v0-$v1

手続きを何度も呼び出す場合は？

jr $ra

スタック (stack)

- 後から入れたデータを先に取り出すデータ構造
  - LIFO (last in, first out)
- スタックの先頭(トップ)にあるデータ(＝最後に入れたもの)にしかアクセスできない

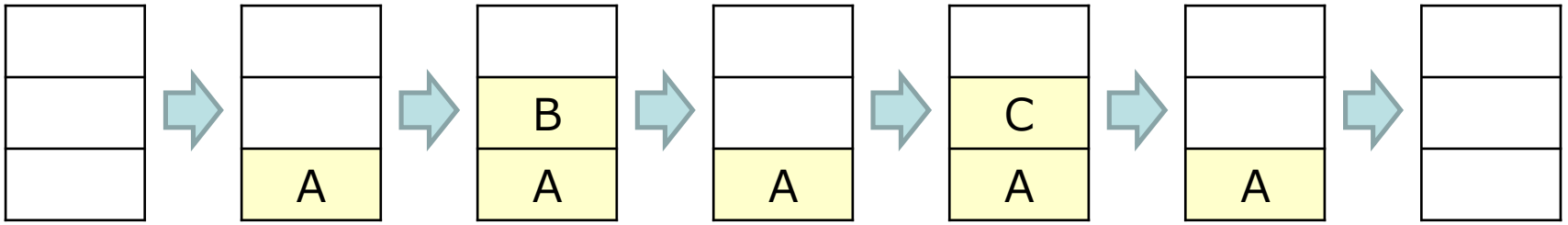


スタックの動作

- 空のスタックを考える。このスタックに対して、以下の操作を行ったときのスタックの変化を示せ。
  - (1) データAをプッシュ
  - (2) データBをプッシュ
  - (3) データをポップ
  - (4) データCをプッシュ
  - (5) データをポップ
  - (6) データをポップ

スタックの動作

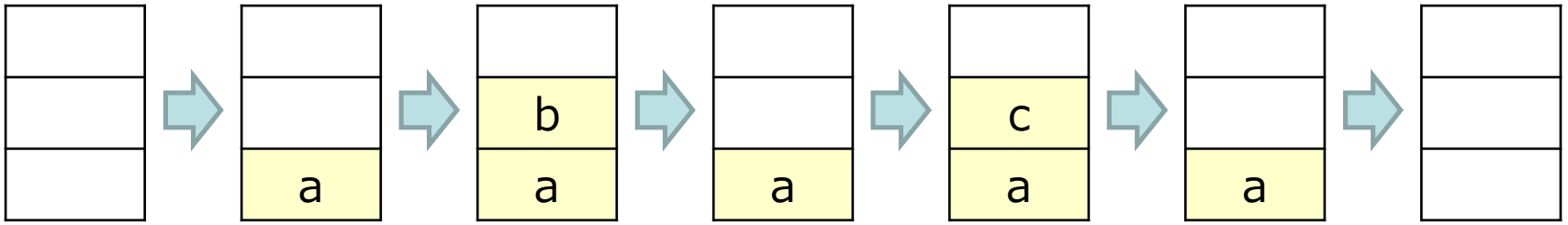
- 空のスタックを考える。このスタックに対して、以下の操作を行ったときのスタックの変化を示せ。
  - (1) データAをプッシュ
  - (2) データBをプッシュ
  - (3) データをポップ
  - (4) データCをプッシュ
  - (5) データをポップ
  - (6) データをポップ



手続き呼び出しとスタックの関係

- 現在実行中の手続きの名前を
  スタックに積むとすると
  - (1) a をプッシュ
  - (2) b をプッシュ
  - (3) ポップ
  - (4) c をプッシュ
  - (5) ポップ
  - (6) ポップ
  - ※mainは省略

```
int main(){
  a();
}

void a(){
  b();
  c();
}
```



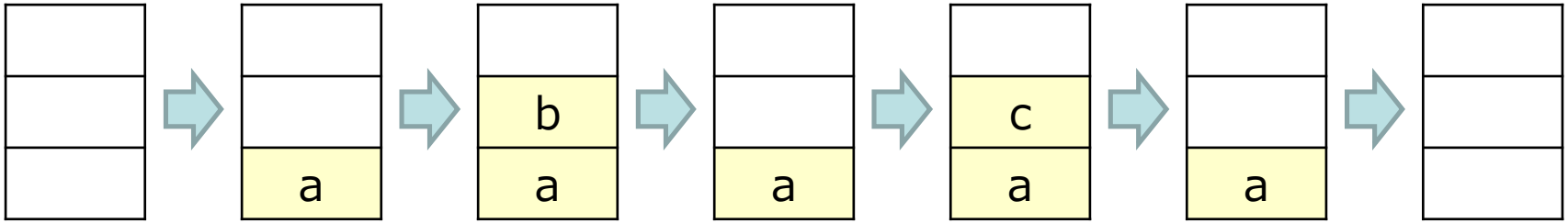
レジスタに入りきらない引数や戻り値は
メモリ中のスタックに積まれる (スピルアウト)

手続き呼び出しとスタックの関係

他の手続きを呼ばない手続きを
リーフ(leaf)手続きと呼ぶ
(この例だとb()やc())

すべてリーフだと、スタック管理は
必須ではない

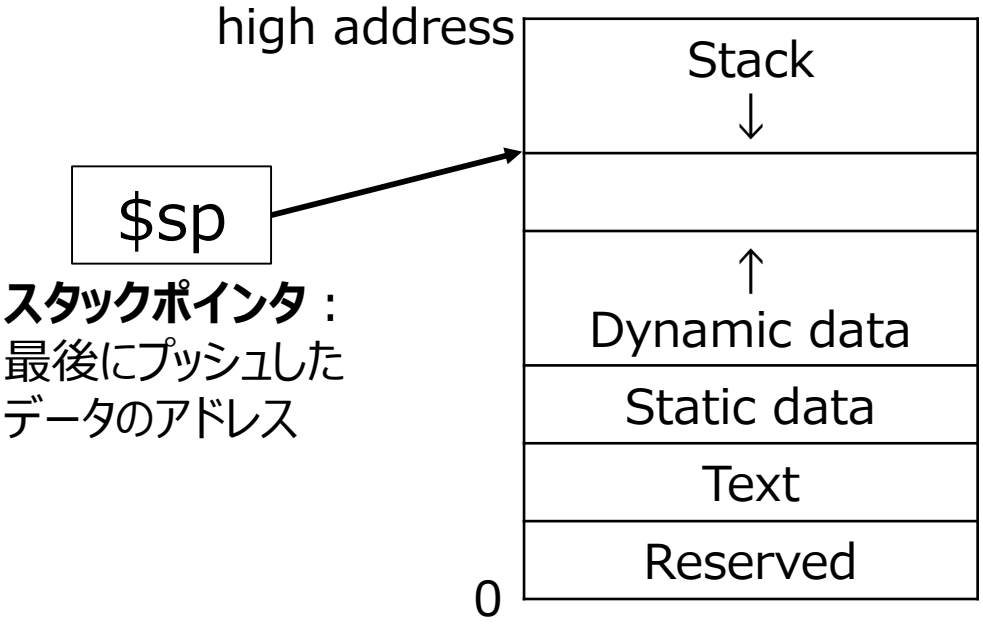
```
int main(){
  a();
}

void a(){
  b();
  c();
}
```



レジスタに入りきらない引数や戻り値は
メモリ中のスタックに積まれる (スピルアウト)

スタックの実現

- スタックの先頭アドレスをレジスタ$spで記憶
- メモリアドレスが大きい方がスタックの底
  ＝メモリアドレスが小さい方がスタックトップ
  (スタックの一番上)

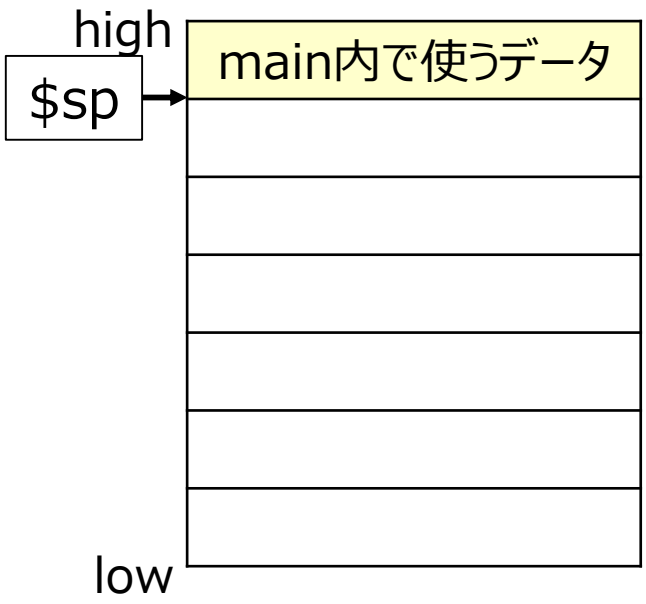


手続き呼び出し時のスタック操作

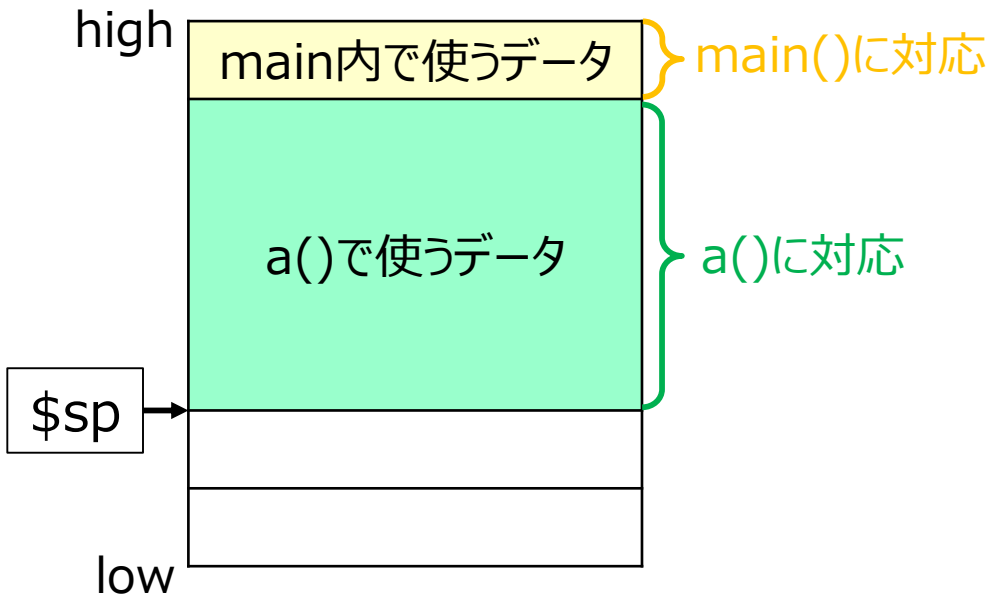
①
②

```
int main(){
   a();
}

void a(){
   int x;
   b(x);
   c();
}
```

①



②

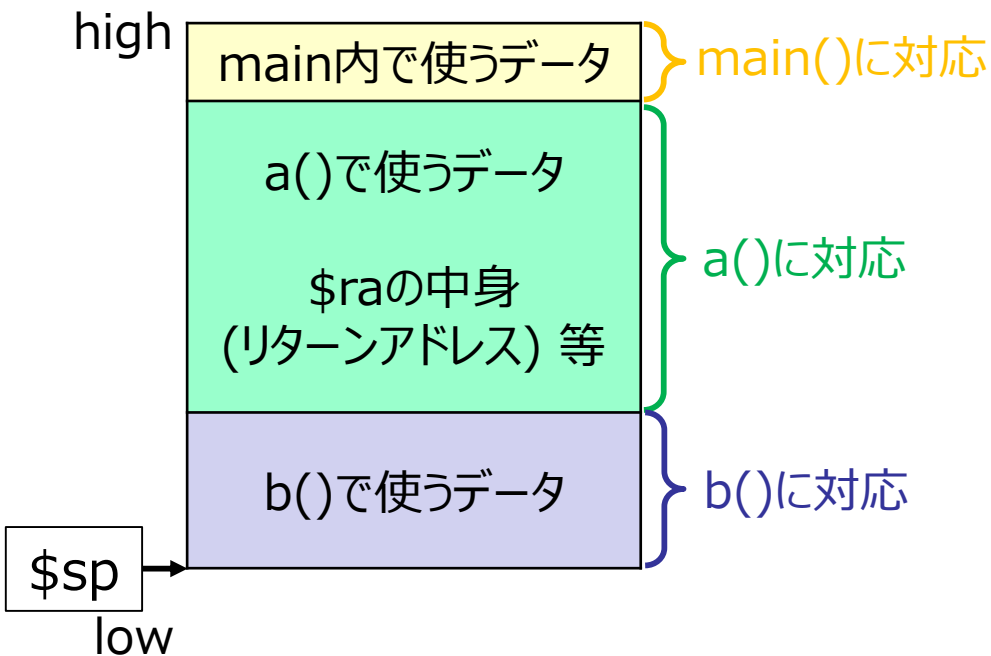


手続き呼び出し時のスタック操作

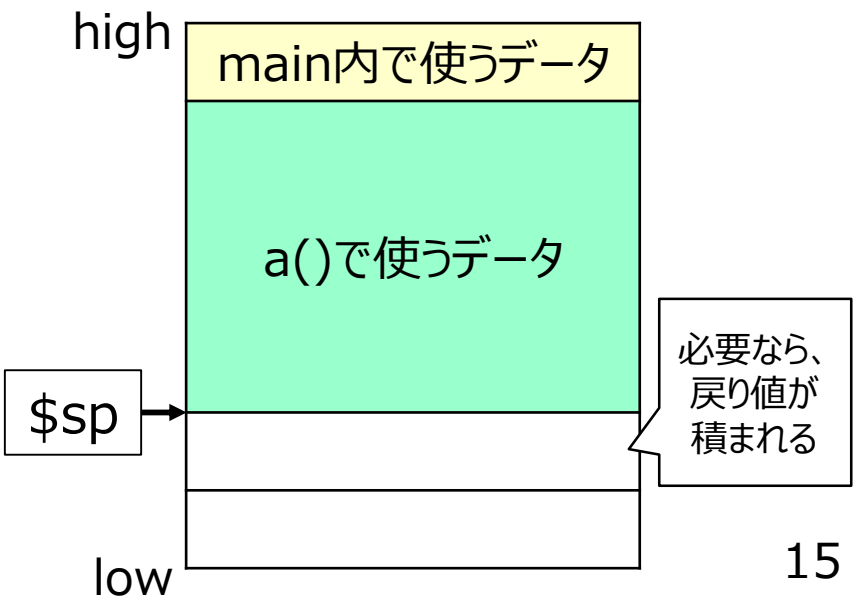
```
int main(){
   a();
}

void a(){
   int x;
   b(x);
   c();
}
```

③
④

③(b()内部実行中)



④

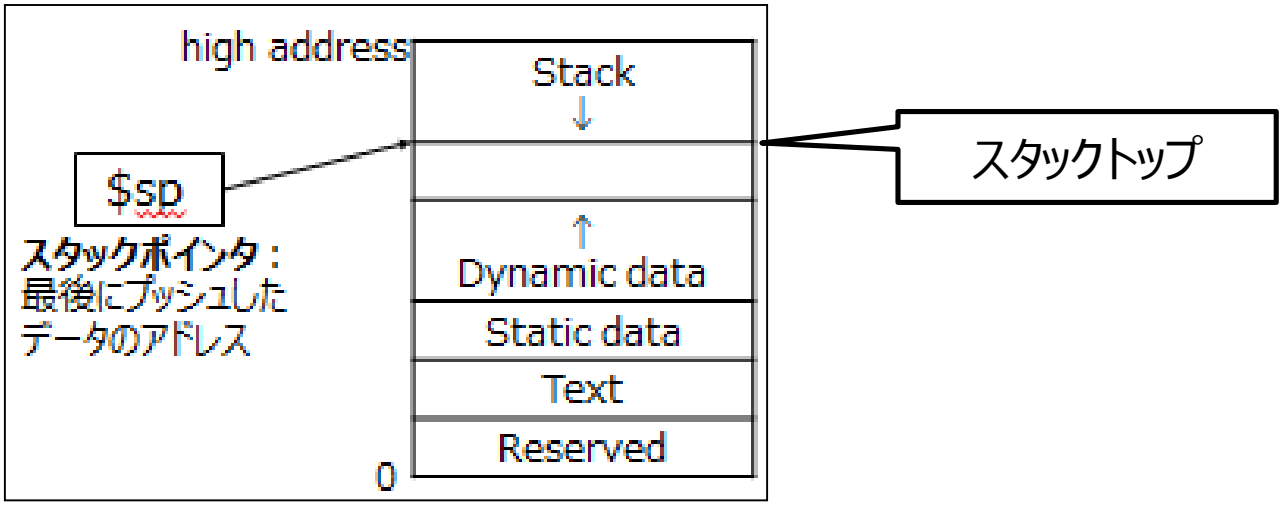


手続き呼び出し時のスタック操作

- 手続きAから手続きBが呼ばれたとき、
  Aの中で使っていたレジスタ内のデータを
  メモリ(スタック)に退避させないといけない
  - レジスタによって、退避させるかどうか
    決まっている
    - $s0-$s7(退避レジスタ), $ra(リターンアドレス) 等は
      退避させる
    - $t0-$t9(一時レジスタ)は退避しない
    - 上書きされないなら、退避しなくても良い
  - Bが呼ばれたとき、上記レジスタの値を
    スタックにプッシュ
  - Bの実行が終了したら、スタックから
    ポップして値をレジスタに戻す

手続き呼び出し時のスタック操作

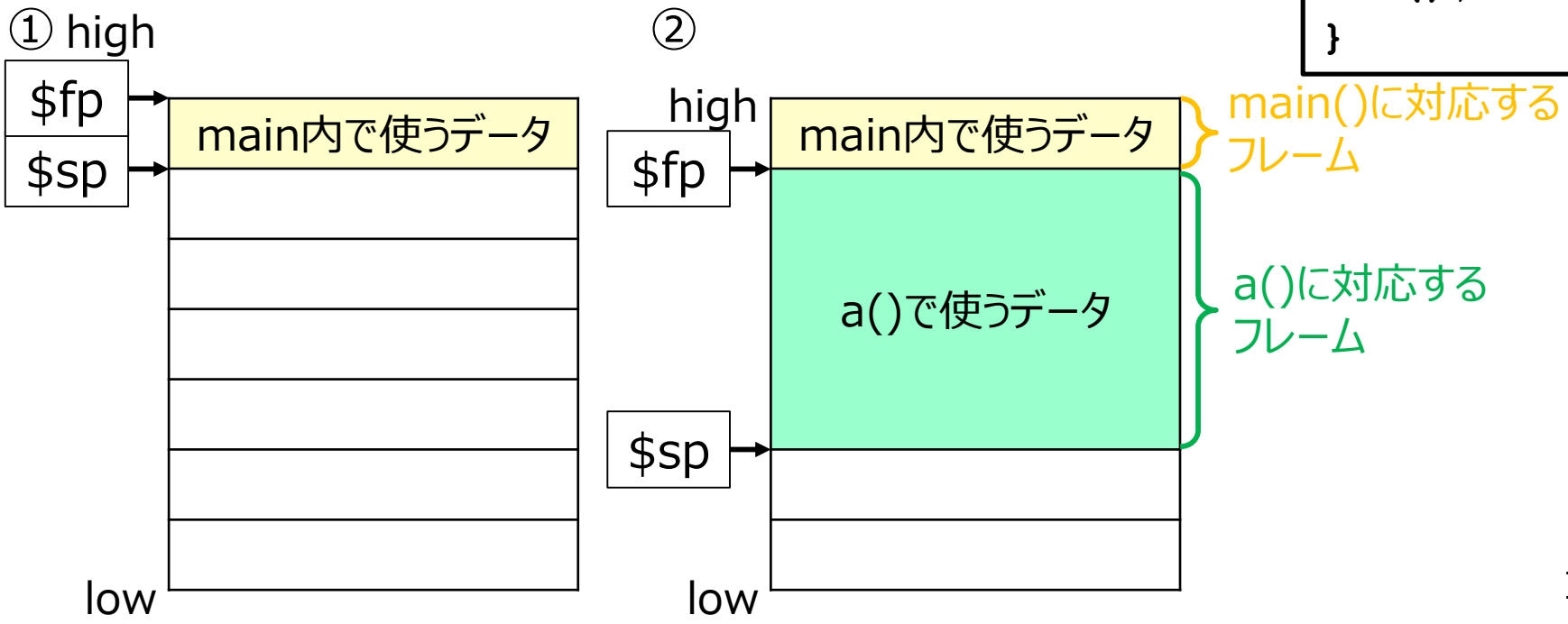
- レジスタ値の退避方法
  - まず $sp の値を加算 (**負の数**)
  - sw命令で退避するレジスタの中身を メモリ(スタック)に書き込む



手続き呼び出し時のスタック操作

- 手続きフレーム
  - その手続きを呼ぶときに新しく確保された スタック上の領域 (スタックフレームとも)
- フレームポインタ $fp
  - フレームの開始アドレスを指す

```
①  int main(){
      a();
    }

    void a(){
      int x;
②     b(x);
      c();
    }
```



確認問題

以下のC言語のソースコードをコンパイルした結果のMIPSアセンブリコードはどうなるか

```
int leaf_example(int i, int j){
  int f, g, h;
  g = i+j;
  h = i-j;
  f = g+h;
  return f;
}
```

レジスタの対応
f:$s0 g:$s1 h:$s2 i:$a0 j:$a1

```
leaf_example:
 addi $sp, $sp, (1)     #3語分確保
 sw   $s2, 8($sp)       #退避
 sw   $s1, 4($sp)       #退避
 sw   $s0, 0($sp)       #退避
 add  $s1, (2), (3)     #g=i+j
 sub  $s2, (2), (3)     #h=i-j
 add  $s0, $s1, $s2     #f=g+h
 add  (4), $s0, $zero   #戻り値設定
```

```
(5)  $s0, 0($sp)    #レジスタ復元
(6)  $s1, 4($sp)    #レジスタ復元
(7)  $s2, 8($sp)    #レジスタ復元
(8)  $sp, $sp, (9)  #3語分取り除く
jr   (10)           #呼び出し元へ戻る
```

```
leaf_example:
  addi $sp, $sp, -12     #3語分確保
  sw   $s2, 8($sp)       #退避
  sw   $s1, 4($sp)       #退避
  sw   $s0, 0($sp)       #退避
  add  $s1, $a0, $a1     #g=i+j
  sub  $s2, $a0, $a1     #h=i-j
  add  $s0, $s1, $s2     #f=g+h
  add  $v0, $s0, $zero   #戻り値設定
```

(i,j)=(3,2)
($s0,$s1,$s2)=(9,8,7)と仮定

```
  lw   $s0, 0($sp)   #レジスタ復元
  lw   $s1, 4($sp)   #レジスタ復元
  lw   $s2, 8($sp)   #レジスタ復元
  addi $sp, $sp, 12  #3語分取り除く
  jr   $ra           #呼び出し元へ戻る
```

レジスタ
$a0: 3 $a1: 2
$s0: 9 $s1: 8
$s2: 7 $v0: ?
$ra: 呼出元PC

スタック

呼出側フレーム

レジスタ
$a0: 3 $a1: 2
$s0: 9 $s1: 8
$s2: 7 $v0: ?
$ra: 呼出元PC

スタック

呼出側フレーム
7
8
9

レジスタ
$a0: 3 $a1: 2
$s0: 6 $s1: 5
$s2: 1 $v0: 6
$ra: 呼出元PC

スタック

呼出側フレーム
7
8
9

```
leaf_example:
 addi $sp, $sp, -12     #3語分確保
 sw   $s2, 8($sp)       #退避
 sw   $s1, 4($sp)       #退避
 sw   $s0, 0($sp)       #退避
 add  $s1, $a0, $a1     #g=i+j
 sub  $s2, $a0, $a1     #h=i-j
 add  $s0, $s1, $s2     #f=g+h
 add  $v0, $s0, $zero   #戻り値設定
```

```
lw   $s0, 0($sp)   #レジスタ復元
lw   $s1, 4($sp)   #レジスタ復元
lw   $s2, 8($sp)   #レジスタ復元
addi $sp, $sp, 12  #3語分取り除く
jr   $ra           #呼び出し元へ戻る
```

レジスタ

$a0: 3 $a1: 2
$s0: 6 $s1: 5
$s2: 1 $v0: 6
$ra: 呼出元PC

スタック

呼出側フレーム
7
8
9

レジスタ

$a0: 3 $a1: 2
$s0: 9 $s1: 8
$s2: 7 $v0: 6
$ra: 呼出元PC

スタック

呼出側フレーム

変数f($s0)に代入したあと、
すぐにreturnするだけなので、
いきなり$v0へ代入しても良い。

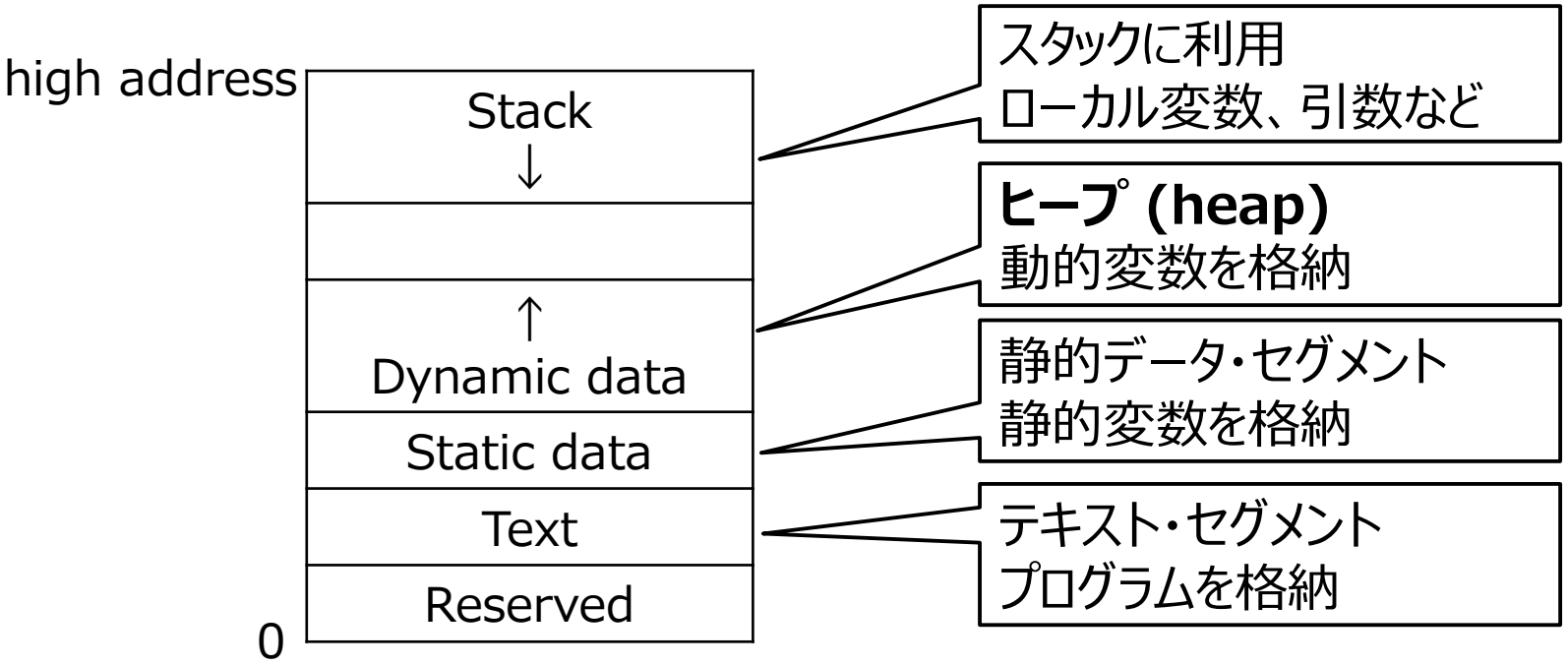
→$s0を退避させる必要はない。

この例の場合、
変数g,hを$t0, $t1で代用しても良い。

→…となると、何も待避させる
必要がない。

「コンパイラの最適化」により、
効率の良いコードを生成するべし。

変数とスタック



- 以下の各レジスタが保持しているものは？
  - $pc 現在実行中の命令のアドレス
  - $gp 静的データにアクセスするための基準アドレス
  - $sp 現在のスタックの一番下位のアドレス
  - $fp 現在の手続きフレームの一番上位のアドレス

手続き呼び出し間で保持されるもの

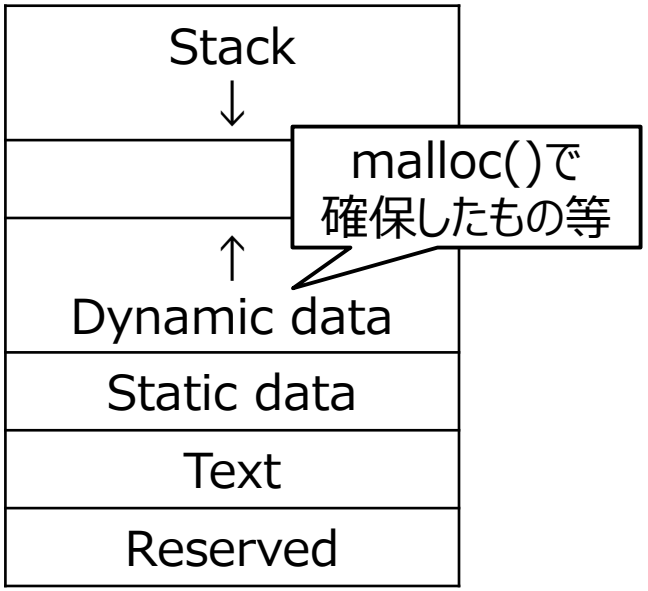
- 保持される
  - 退避レジスタ $s0-$s7
  - スタックポインタ $sp
  - フレームポインタ $fp
  - グローバルポインタ $gp
  - リターンアドレス $ra
- 保持されない
  - 一時レジスタ $t0-$t9
  - 引数レジスタ $a0-$a3
  - 戻り値レジスタ $v0-$v1

C言語の変数の記憶クラス

- 自動変数 (局所変数)
  - ブロック内のみで存続
  - 宣言したスコープ内からアクセス可能
  - スタックに置かれる
- 外部変数
  - プログラム実行中ずっと存続
  - どこからでもアクセス可能
  - 静的データ・セグメントに置かれる
- 静的変数
  - プログラム実行中ずっと存続
  - 宣言したスコープ内からアクセス可能
  - 静的データ・セグメントに置かれる

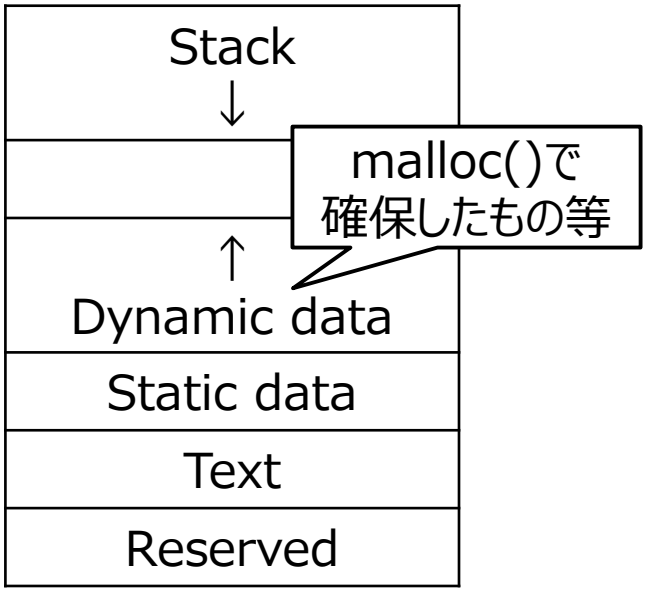
※ レジスタ変数

- レジスタに割り当てる自動変数



メモリの解放

- 動的データのメモリ解放
  - C言語だとfree()
  - 解放し忘れ
    → メモリ・リーク(memory leak)
    → メモリ・リークが増えると、
    新たに確保できる領域が
    なくなる
  - 解放が早すぎる
    → 宙ぶらりんなポインタ
    (dangling pointer)が発生



- Java等の場合はガベージ・コレクション
(garbage collection) により、
適切に・自動的に解放が行われる

確認問題

以下のC言語のソースコードをコンパイルした結果のMIPSアセンブリコードはどうなるか

```
int fact (int n){
  if(n<1)
    return 1;
  else
    return n * fact(n-1);
}
```

fact呼び出し時、$a0, $raの値を退避する必要があるとする。

レジスタの対応　n:$a0

```
fact:
 addi $sp, (1), (2)    #2語分確保
 sw   $ra, 4($sp)      #退避
 sw   $a0, 0($sp)      #退避
 slti $t0, $a0, 1      #n<1か？
 beq  $t0, $zero, (3)  #分岐
 addi $v0, $zero, 1    #戻り値設定
 addi $sp, (4), (5)    #2語取り除く
 jr   (6)              #return
```

```
L1:
 addi $a0, $a0, -1     #引数n-1
 jal  (7)              #fact呼出
 lw   $a0, 0($sp)      #レジスタ復元
 lw   $ra, 4($sp)      #レジスタ復元
 addi $sp, (8), (9)    #2語取り除く
 mul  $v0, $a0, $v0    #n*fact(n-1)
 jr   (10)             #return
```

参考文献

- コンピュータの構成と設計 上 第5版
  David A.Patterson, John L. Hennessy 著、
  成田光彰 訳、日経BP社
- 山下茂 「計算機構成論 1 」講義資料