計算機構成論

Lecture 6
手続き・関数呼び出しの処理

2023年度春学期
情報理工学部 Rクラス担当
越智裕之

※ このレジュメの中で「教科書」とは、一昨年度まで教科書に指定されていた書籍のことです（Lecture 0 参照）。この書籍を入手できなくても支障なく学習できるよう、レジュメに加筆修正を行っています。

内容

- 手続き(Procedure)の実行方法
  - 手続きの実行の概要
  - 手続きの実行：他の手続きを呼ばない場合
  - 手続きの実行：他の手続きを呼ぶ場合

•メモリ上でのプログラムとデータの配置
•スタック
•手続きフレームとフレーム・ポインタ(発展)

- 教材：教科書2.8節

C言語の変数の記憶クラス

		生存期間（時間的な有効範囲）	
		関数やブロックの開始時に生成され、終了時に破棄される	**プログラム起動後、終了までずっと生存し続ける**
スコープ（場所的な有効範囲）	局所的（関数やブロック内でのみアクセス可）	**自動変数**	静的ローカル変数
	大域的（プログラム中のどの関数からもアクセス可）		**グローバル変数**

※スコープはプログラミングにおいては重要ですが、この授業では深入りしません。
※**生存期間はメモリ上でのデータ配置と密接に関連する**ので、この授業では重要です。

静的な局所変数の挙動を説明せよ（この授業では深入りしない）

```
#include <stdio.h>

int total(int a)
{
  static int  t = 0;     // static variable 静的な局所変数

  t = t + a;
  printf("total=%d¥n", t);
}

int main(void)
{
  total(1);            // total=    と出力
  total(2);            // total=    と出力
  total(3);            // total=    と出力
  return 0;
}
```

- static宣言された変数は、プログラムの起動から終了まで、ずっと生存し続ける（次に total() が呼ばれた時も前の値が残っている）
- static宣言された変数の初期化（この例では t=0 ）は、プログラム起動時に1回だけ行われる（関数 total() が呼ばれるたびに t が 0 になるわけではない）

手続き　手続き、サブルーチン（Cでいえば関数）

```
int array[100];
main()
{
        int cost1, cost2, x1, x2, y1, y2, temp;
        ……
        cost1 = cost_func (x1, y1);
        cost2 = cost_func(x2, y2);
         ……
}
```

と呼ばれるデータを受け取り、定められた通りの処理を実行して結果を返す一連の命令群。

```
int cost_func (int x, int y)
{
        int temp;
        static int count =0;
        count++;
        /* xとyからあるコスト関数を計算して ansに*/
        int *data = (int *) malloc(sizeof(int) *count);
    ……
        free (data);
        return ans;
}
```

ミニクイズ: 前のページの，以下のそれぞれの変数の記憶クラスは何か？また，それぞれ，次のページのどの部分に格納されるか？

temp, array, count

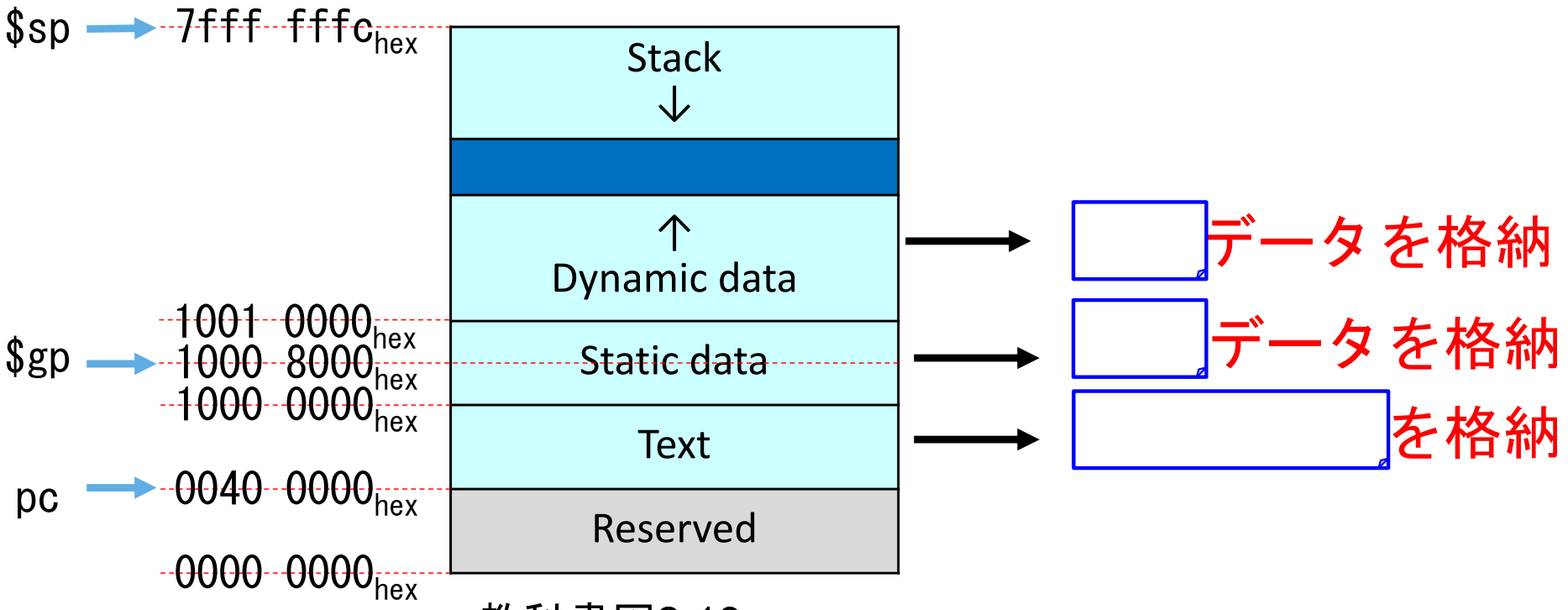
```
int array[100];
main()
{
        int cost1, cost2, x1, x2, y1, y2, temp;
        ……
        cost1 = cost_func (x1, y1);
        cost2 = cost_func(x2, y2);
         ……
}
```

```
int cost_func (int x, int y)
{
            int temp;
            static int count =0;
            count++;
            /* xとyからあるコスト関数を計算して ansに*/
            int *data = (int *) malloc(sizeof(int) *count);
          ……
            free (data);
            return ans;
}
```

- 自動変数（局所変数） ..... [ ]
- 外部変数（グローバル変数，大域変数）... [ ]
- 静的変数 ..... [ ]
- 配列dataの要素は，動的データ　に格納

メモリ上でのプログラムとデータの配置（MIPSの場合）

＊C言語の記憶クラスの概念とは異なる話のため混乱しないように注意

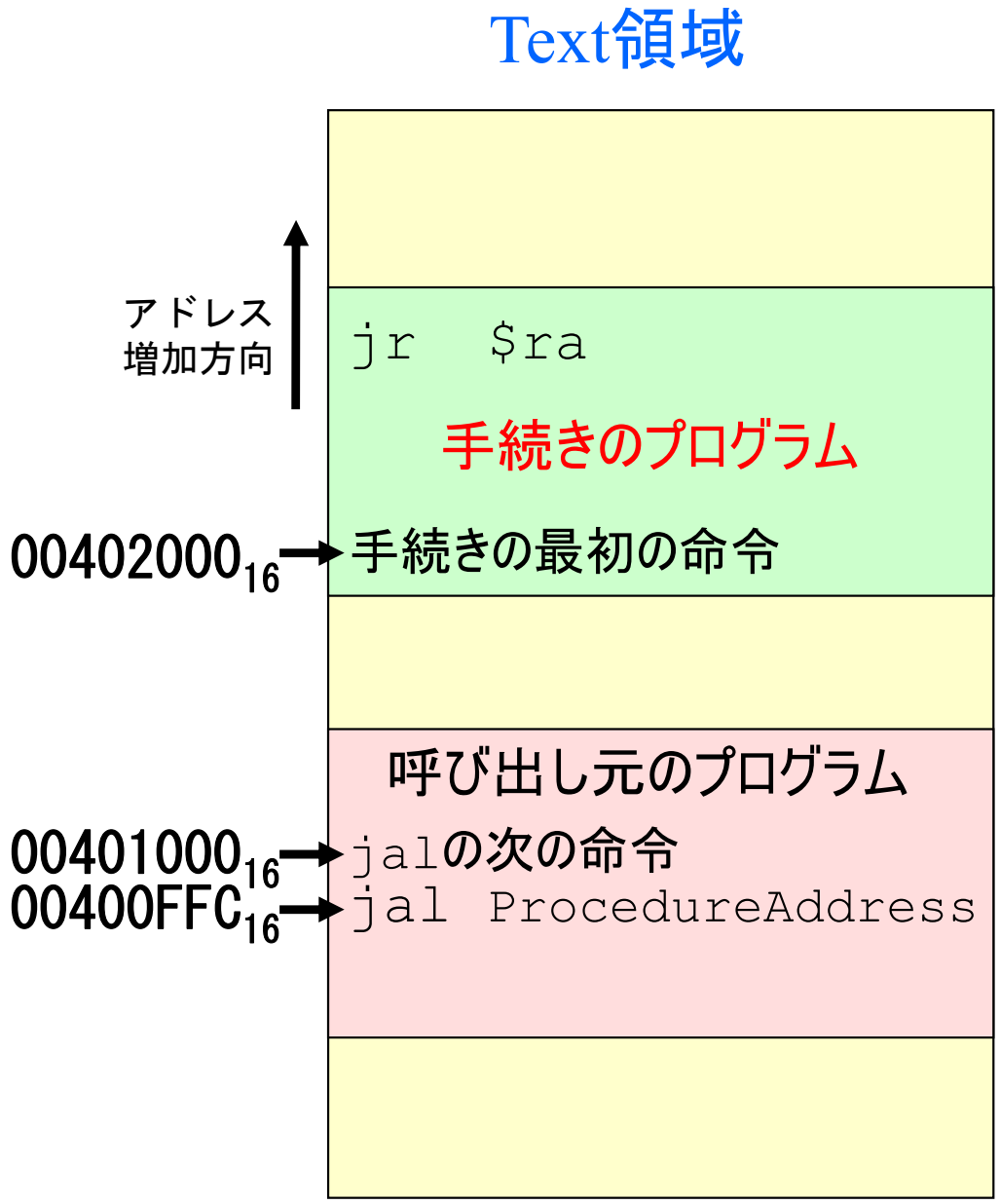


教科書図2.13

手続きの実行手順の概略（教科書p95）

1. 手続きからアクセスできる場所に パラメータを置く
2. 手続きに制御を移す
3. 手続きに必要なメモリ資源を入手
4. 必要なタスクを実行する
5. 呼び出し元のプログラムからアクセスできる場所に結果を置く
6. 制御を元の位置に戻す

手続き終了後には、何もなかったように復帰



手続き読出しのための具体的手順（教科書p95 ）

①手続き用の引数を$a0～$a3に格納

②手続きを開始する命令：jal(jump and link)命令

```
jal   ProcedureAddress
```

手続きの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$raに保持する
（　　をレジスタ$raに格納）

③（必要があればメモリから必要なデータをロードなどをする）
（必要があればレジスタのデータをメモリに退避する）

データのロードや退避の順序は変わる場合もある。

④手続き本体

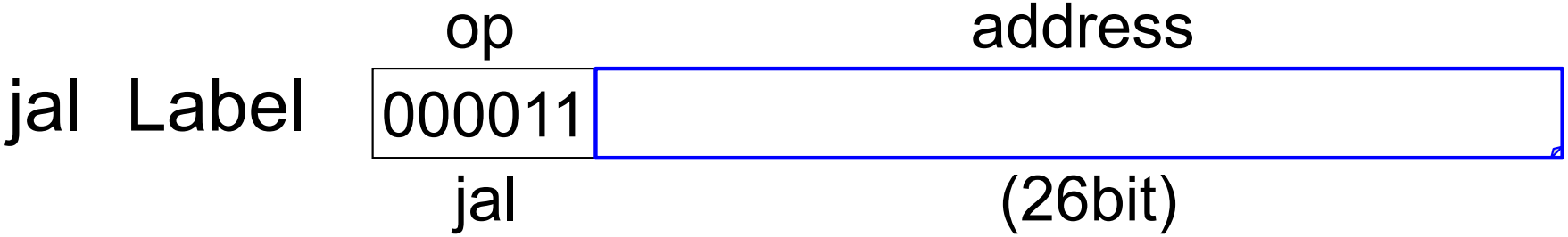
⑤手続きの戻り値を$v0, $v1に格納

⑥手続きを終了する `jr  $ra` 次の命令アドレス（レジスタ$raに格納されている）へジャンプ

- $a0～$a3 ：4個の引数レジスタ、パラメータ（引数）を渡すために使用
- $v0～$v1：2個の戻り値レジスタ、結果の値を返すために使用
- $ra：1個の戻りアドレス・レジスタ、制御を元に戻すために使用

前のスライドの状況で,
問い①　jrを実行するときの$raの値を述べよ
問い②　`jal ProcedureAddress` を機械語にせよ。

jal命令とjr命令



jal命令における飛び先アドレスの指定方法は，j命令と同様（次スライド参照）

前ページの問い②の答え

飛び先アドレスが 00402000_{16}
=0000 **0000 0100 0000 0010 0000 0000 00**00_2 の場合、
その上位4ビットと下位2ビットを取り除いた26ビットが
jal 命令の address フィールドになる

jr $ra

000000	11111	00000	00000	00000	001000

jr $ra (31番レジスタ）

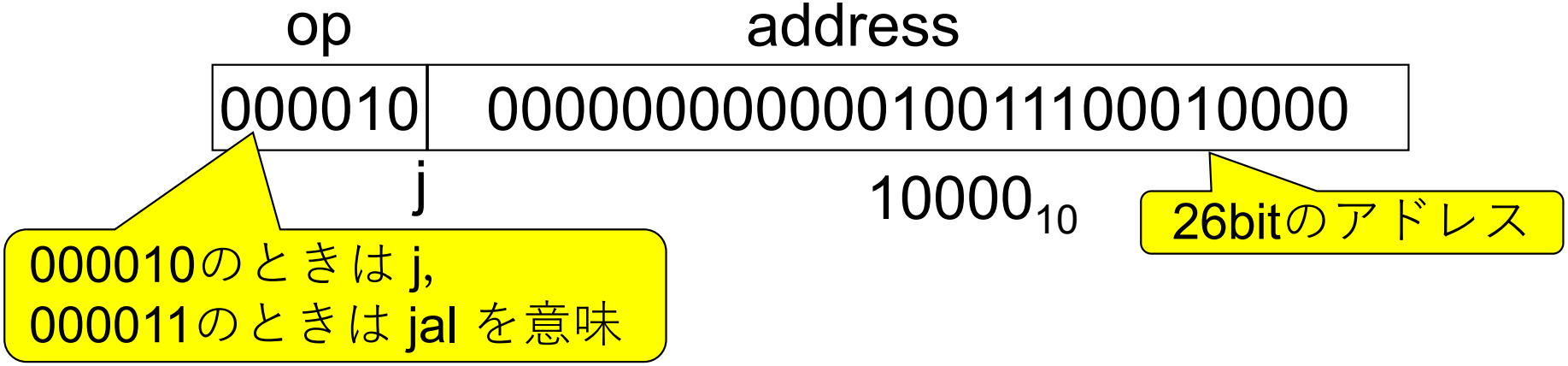
jal命令: $ra = “$pc” +4
jr命令: $pc = $ra

前ページの問い①の答え

PC(プログラムカウンタ)：　　を保持するレジスタ

j Label

ラベルのアドレスが40000_{10}のときは以下の機械語となる



PCに
“現在のPCの上位4ビット” と “26ビットのaddress” と “00” を連接した値をセット

内容

- 手続き(Procedure)の実行方法
  - 手続きの実行の概要
  - 手続きの実行：他の手続きを呼ばない場合
  - 手続きの実行：他の手続きを呼ぶ場合

> •メモリ上でのプログラムとデータの配置
> •スタック
> •手続きフレームとフレーム・ポインタ(発展)

- 教材：教科書2.8節

手続きの実行と「スタック」

- 手続き(Procedure)の実行では以下のような様々なデータの保持が必要
  - 呼び出し元のレジスタの値
    - 破壊されたくないレジスタは退避が必要
  - 4個を越えた引数
    - 4番目までの引数は $a0～$a3 で渡す
  - 2個を越えた戻り値
    - 2番目までの戻り値は $v0～$v1 で渡す
  - 手続きの中で使用される自動変数
    - （静的でない）ローカル変数は、手続きの起動時に領域が確保され、手続き終了時に破棄される
- 上記のようなデータの保持にスタックを使う

【復習】スタック

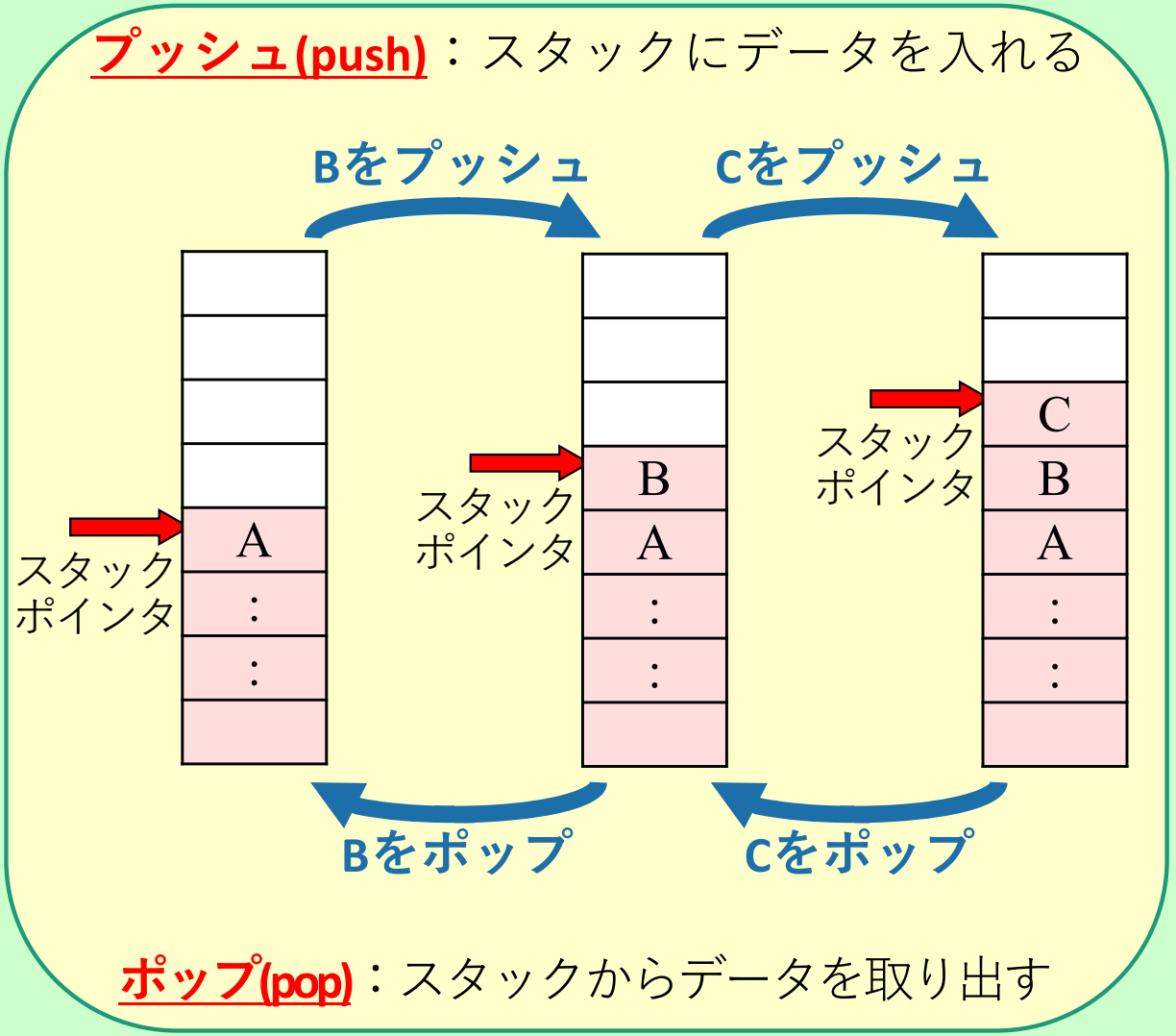
情理の学生として重要

- 「データ構造とアルゴリズム」で習った筈ですが、この授業に関連する内容を概説します

“スタック”とは

スタックの内容には自由にアクセスできず、スタックトップへのデータのプッシュとポップのみしかできない

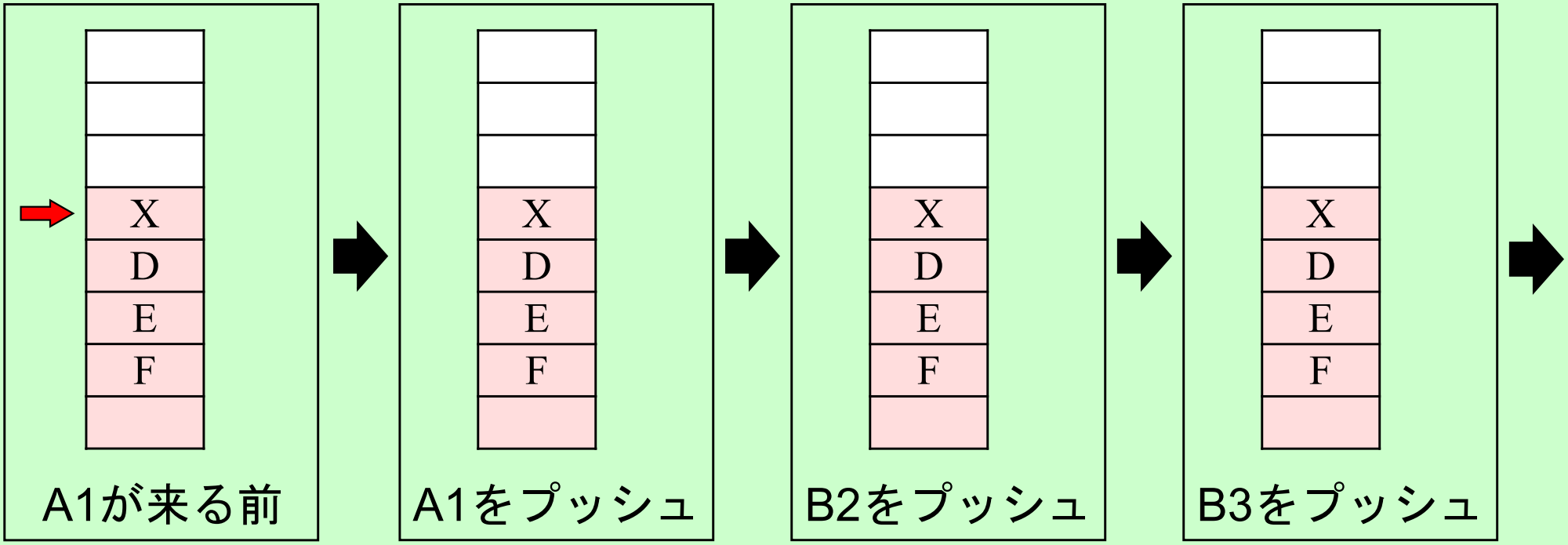
⇒単純な構造（アドレス指定なし）で、データを格納した順と逆に取り出せるので、良く利用される。



※一般的なスタックの説明図の流儀で図示しています（プッシュされたデータは「上」に積まれています）

スタックの動作例 (1/3)

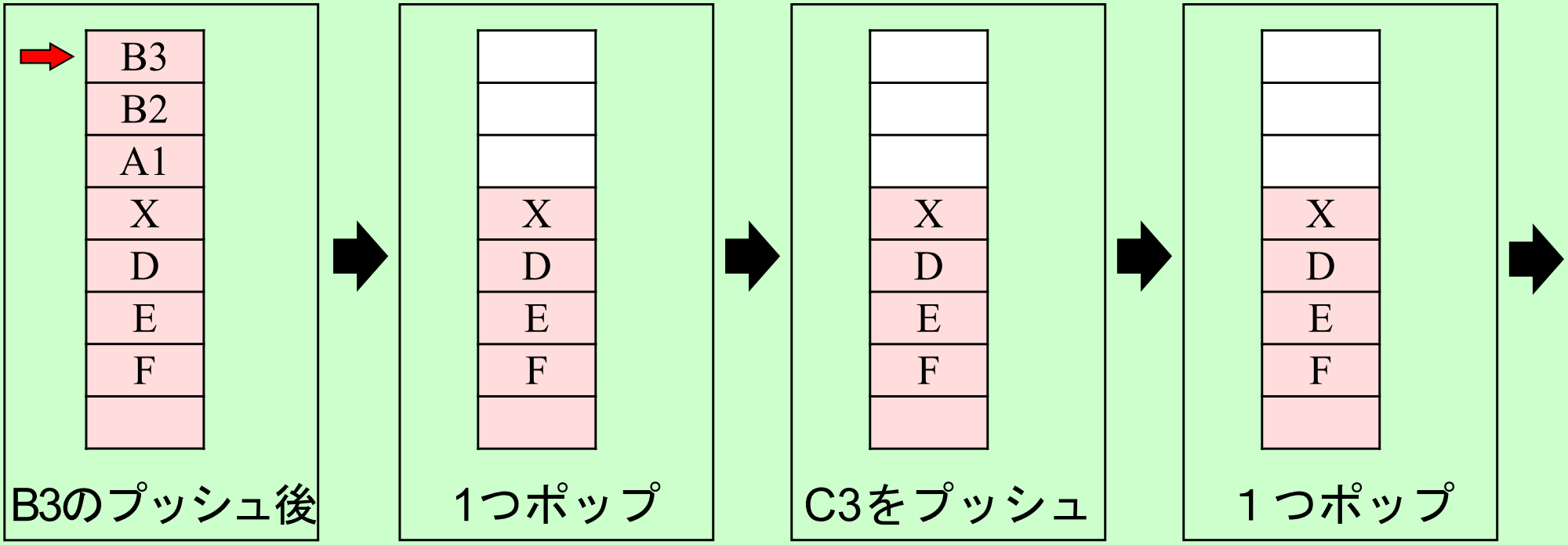
ミニクイズ：
以下のような動作をスタックにした時に、スタックがどのように変化するか示せ。
•A1, B2, B3の順でデータが来る（それを順に覚える：プッシュ）
•その時点で最新のものを一つ取り出す（ポップ）
•次にC3のデータが来るのを覚える。
•次に最新のものから順にデータを取り出す。



※一般的なスタックの説明図の流儀で図示しています（プッシュされたデータは「上」に積まれています）

スタックの動作例 (2/3)

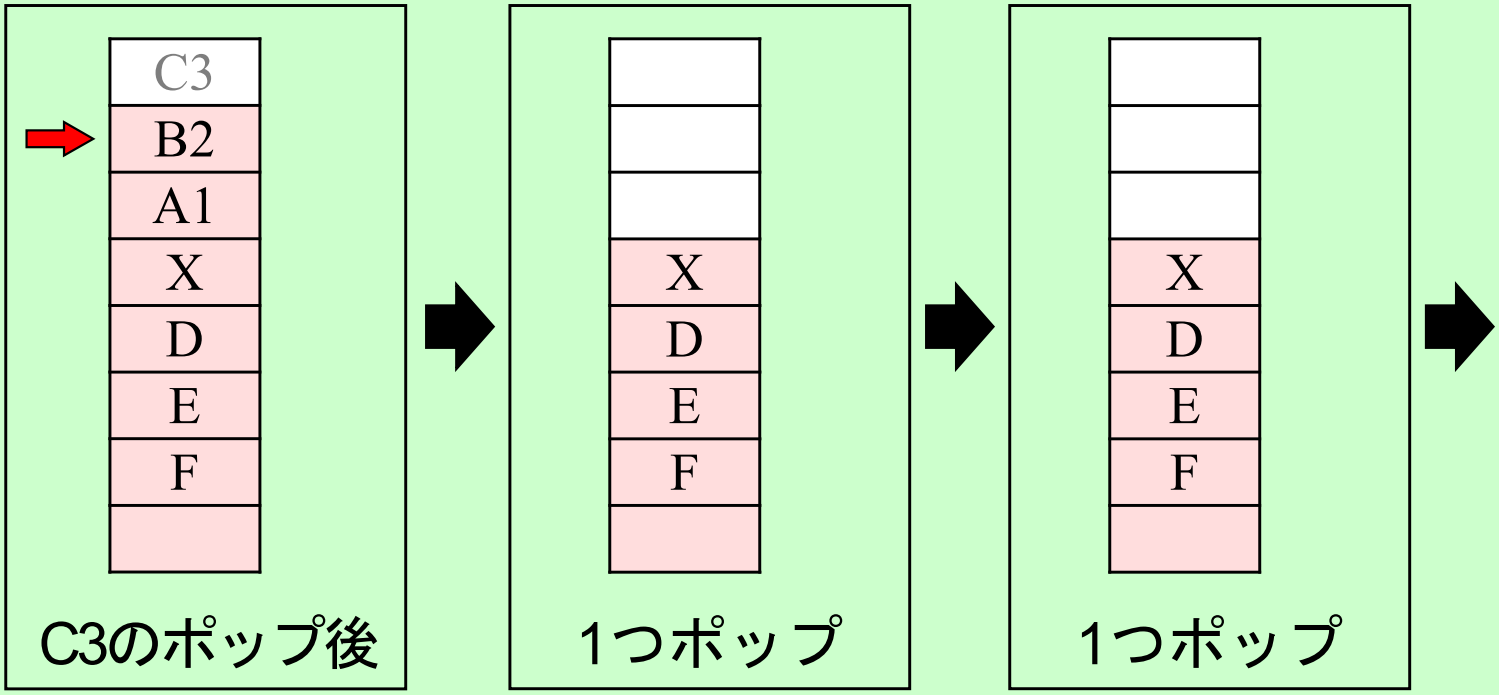
ミニクイズ：
以下のような動作をスタックにした時に、スタックがどのように変化するか示せ。
•A1, B2, B3の順でデータが来る（それを順に覚える：プッシュ）
•その時点で最新のものを一つ取り出す：ポップ
•次にC3のデータが来るのを覚える。
•次に最新のものから順にデータを取り出す。



※一般的なスタックの説明図の流儀で図示しています（プッシュされたデータは「上」に積まれています）

スタックの動作例 (3/3)

ミニクイズ：
以下のような動作をスタックにした時に、スタックがどのように変化するか示せ。
•A1, B2, B3の順でデータが来る（それを順に覚える：プッシュ）
•その時点で最新のものを一つ取り出す：ポップ
•次にC3のデータが来るのを覚える。
•次に最新のものから順にデータを取り出す。



※一般的なスタックの説明図の流儀で図示しています（プッシュされたデータは「上」に積まれています）

スタックの実現方法

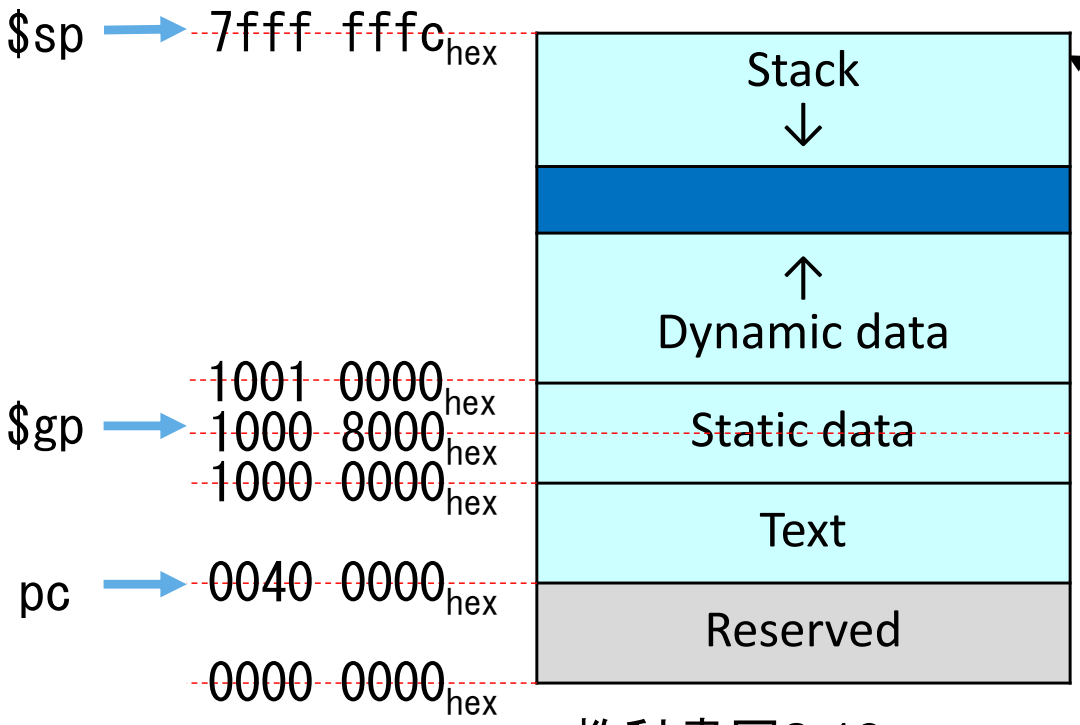
[　　] で管理する

- スタックの中で最も新しく格納したデータのアドレス。
- 論理的にはスタックトップに相当

（MIPSではメモリの下位になるので、“トップ”という言葉に注意）

注意：MIPSでは、メモリの上位から下位にスタックトップが「伸びて」いく

MIPSアーキテクチャでは、スタックポインタを覚えるのに専用のレジスタ **$sp** がある。
このアドレスを **$sp** に格納



教科書図2.13

※以降のスタックの説明図では、プッシュされたデータが「下」に積まれるので留意して下さい。

実は、呼び出された側でさらに必要な処理がある

- ルーチンＡから呼び出されたルーチンＢはいろんなレジスタを使う。
- もしＢが使用するレジスタ$s0にAがまだ使いたい値が入っていたら…

そこで、呼び出されたルーチンＢは、
- 最初に、$s0の値をスタックに　　して値を保存（退避）
- 自分の処理が終わってから、スタックから　　して$s0の値を復元

実際は、退避を必要とする複数のレジスタに対して行う。
なお、MIPS のソフトウェア規約では、
- $t0～$t9 は、呼び出された側でたとえ利用しても退避する必要なし
- $s0～$s7 は、呼び出された側で使用するなら、退避する必要あり

演習問題 その①

教科書P96の例題

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

のMIPSアセンブリ・コードを示せ。
処理の一時レジスタとしては、$t0, $t1, $s0を使い、それらの値を最初に退避すること

呼び出し側では、呼ぶ前に、
- jalで$raを正しく設定
- 引数g, h, i, jを$a0-$a3に格納している

（注意）ここでは、一時的に使うレジスタも、スタックを用いて保存するようにしているが、普通のコンパイラでは、$t0, $t1は保存しない。

解答

```
leaf_example:
    addi    $sp,  $sp,  -12
    sw      $t1,  8($sp)
    sw      $t0,  4($sp)
    sw      $s0,  0($sp)
    add     $t0,  $a0,  $a1
    add     $t1,  $a2,  $a3
    sub     $s0,  $t0,  $t1
    add     $v0,  $s0,  $zero
    lw      $s0,  0($sp)
    lw      $t0,  4($sp)
    lw      $t1,  8($sp)
    addi    $sp,  $sp,  +12
    jr      $ra
```

演習問題 その①

つづき

さらに、この手続き呼び出しの前、途中、後のスタックの状況を説明せよ

（図2.10のような図を描いて説明できればOK）

回答：図2.10　(英語スライド)

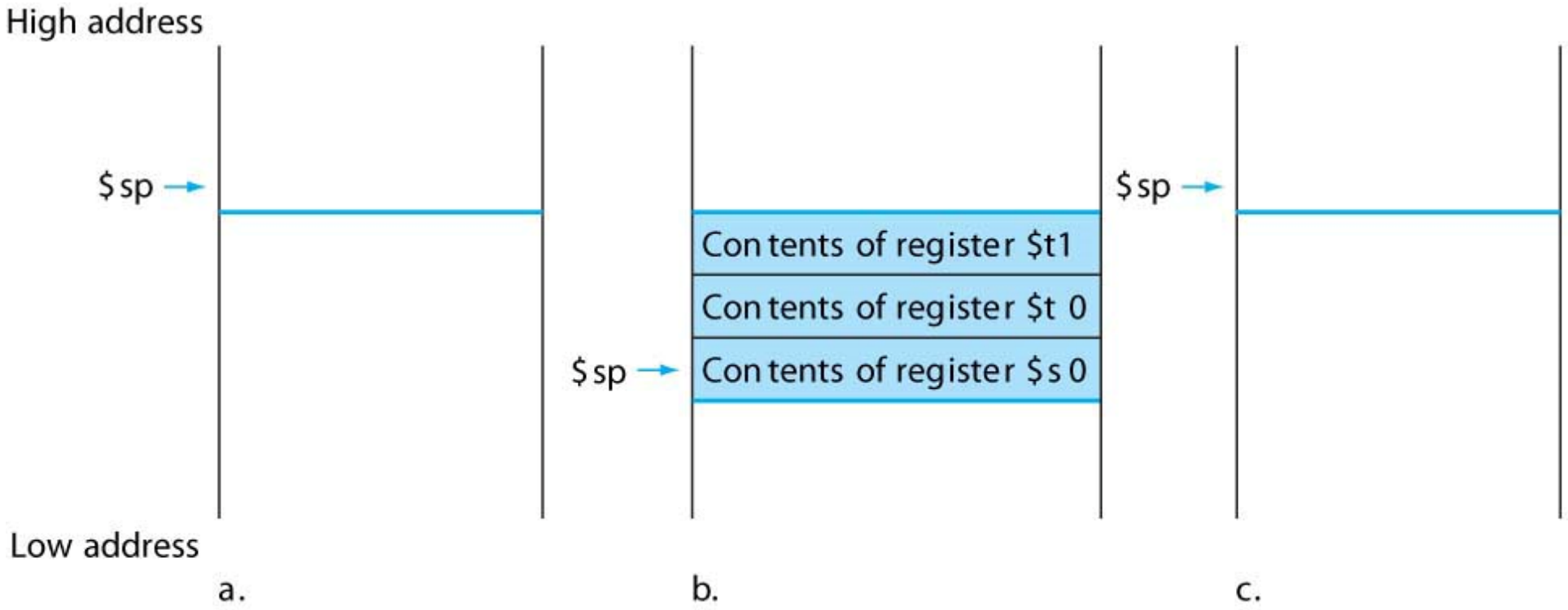


FIGURE 2.10: The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the “top” of the stack, or the last word in the stack in this drawing.

※しつこいようですが、このスタックの説明図では、プッシュされたデータが「下」に積まれているので留意して下さい。また、一番下のデータの位置を「トップ」と呼んでいます。

演習問題 その①

つづきのつづき

更なる質問:「⑤手続きの返り値を$v0, $v1に格納」 を行っているのはどこか？

更に考えよう:ちなみに、この例では、その行をなくして、全体で命令3つ減らせませんか？

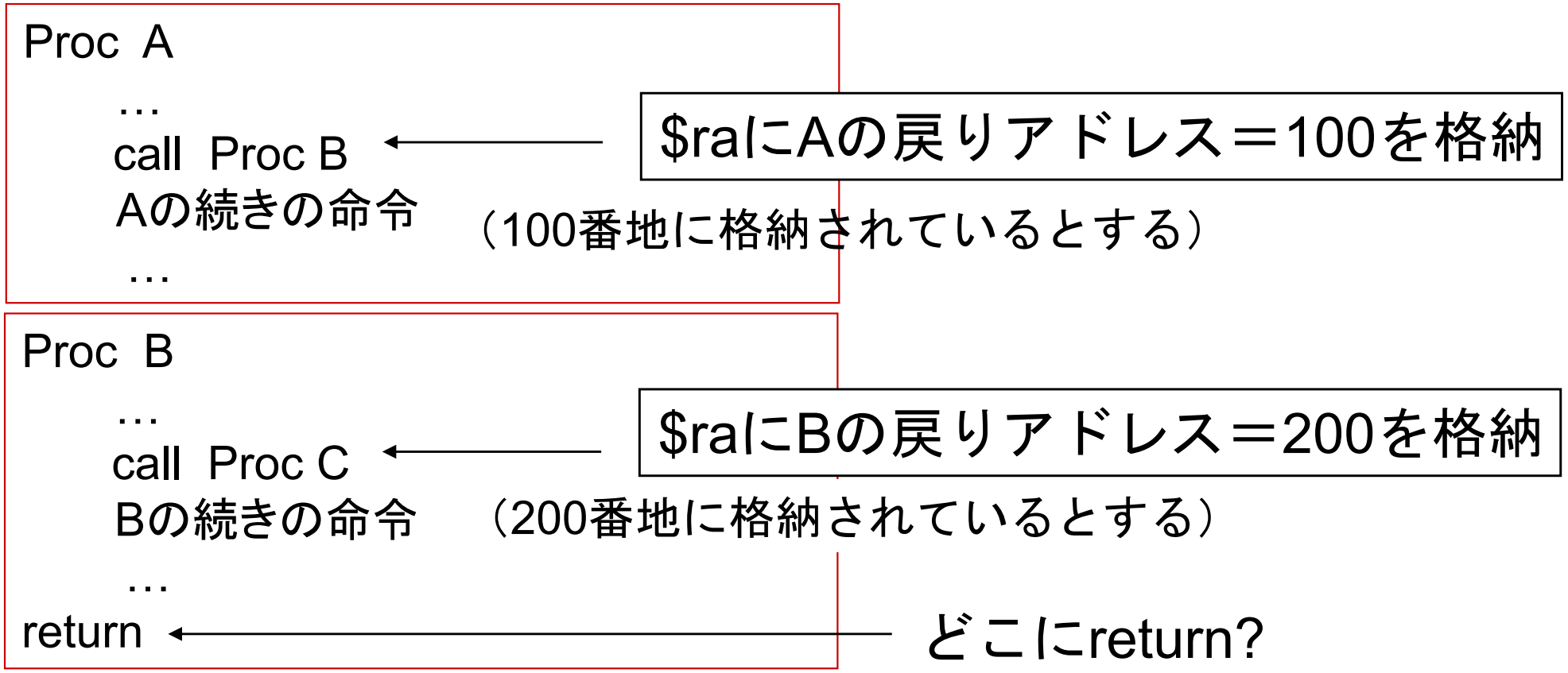
内容

- 手続き(Procedure)の実行方法
  - 手続きの実行の概要
  - 手続きの実行：他の手続きを呼ばない場合
  - 手続きの実行：他の手続きを呼ぶ場合

•メモリ上でのプログラムとデータの配置
•スタック
•手続きフレームとフレーム・ポインタ(発展)

- 教材：教科書2.8節

入れ子に手続きを呼ぶ場合：問題点



まともにやると、

手続き用の引数レジスタ $a0～$a3
関数内で計算に用いられているレジスタ
戻りアドレスレジスタ $ra

が競合してしまう。

手続きの呼び出し元に復帰不能になる

入れ子に手続きを呼ぶ場合：解決方法

手続きCを呼ぶときには、手続きBであとで必要になるであろう全てのレジスタの値をスタック上にプッシュしておいて、あとで必要になったときにスタックからポップしてレジスタの値を復元する

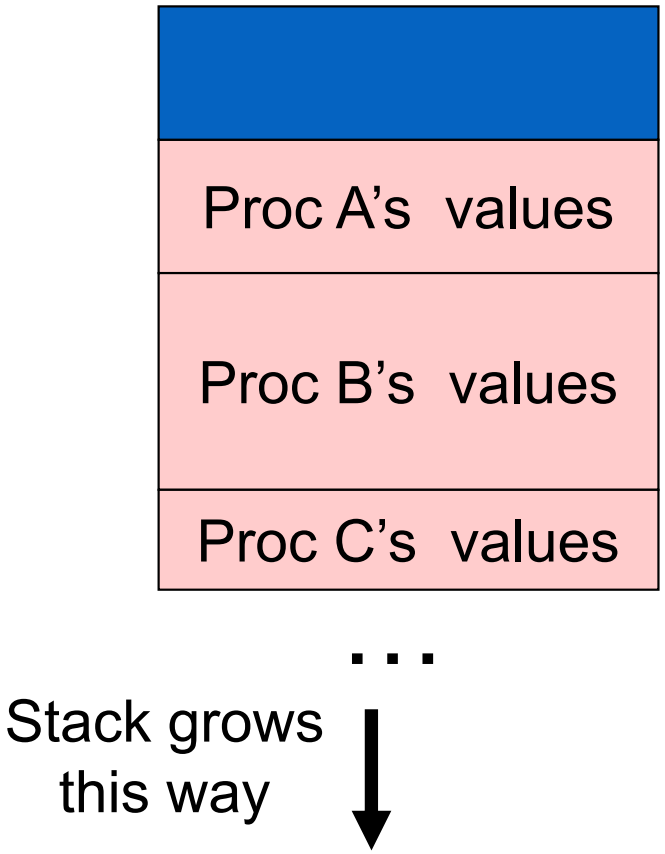
以下が実際の手続き：演習問題がすべてできればOK

呼び出し側 ： 呼び出し後に必要となる引数レジスタ$a0～$a3、一時レジスタ$t0～$t7があれば、プッシュ。（$spはもちろん調整）

被呼び出し側: 戻りアドレスレジスタ$ra及び被呼び出し側で使用する退避レジスタ$s0～$s7をプッシュ。（$spはもちろん調整）
手続きから戻るときは、レジスタをメモリから復元し（ポップ）、スタック・ポインタを再調整

(注意：必ずしもこの原則でない例も多い。また、演習問題の例などでは、一つの関数が呼び出し側でも呼び出され側でもあるので，どちらが呼び出し側とそもそも切り分けにくい。ただ、必要な保存と復元はどこかでは必ず行われていることは重要。）

スタックを利用すると容易に実現可能




```
Proc  A

        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```

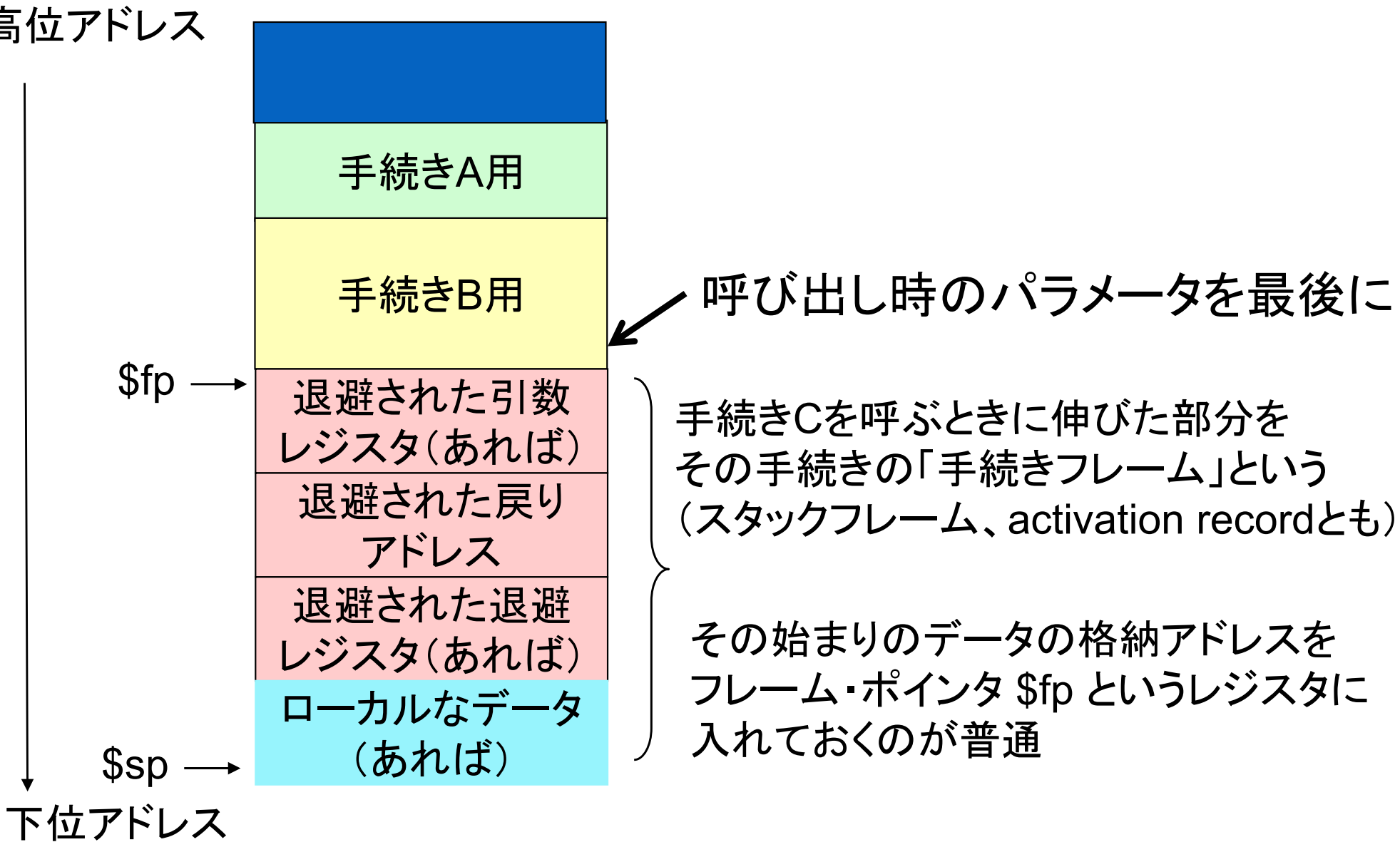
内容

- 手続き(Procedure)の実行方法
  - 手続きの実行の概要
  - 手続きの実行：他の手続きを呼ばない場合
  - 手続きの実行：他の手続きを呼ぶ場合

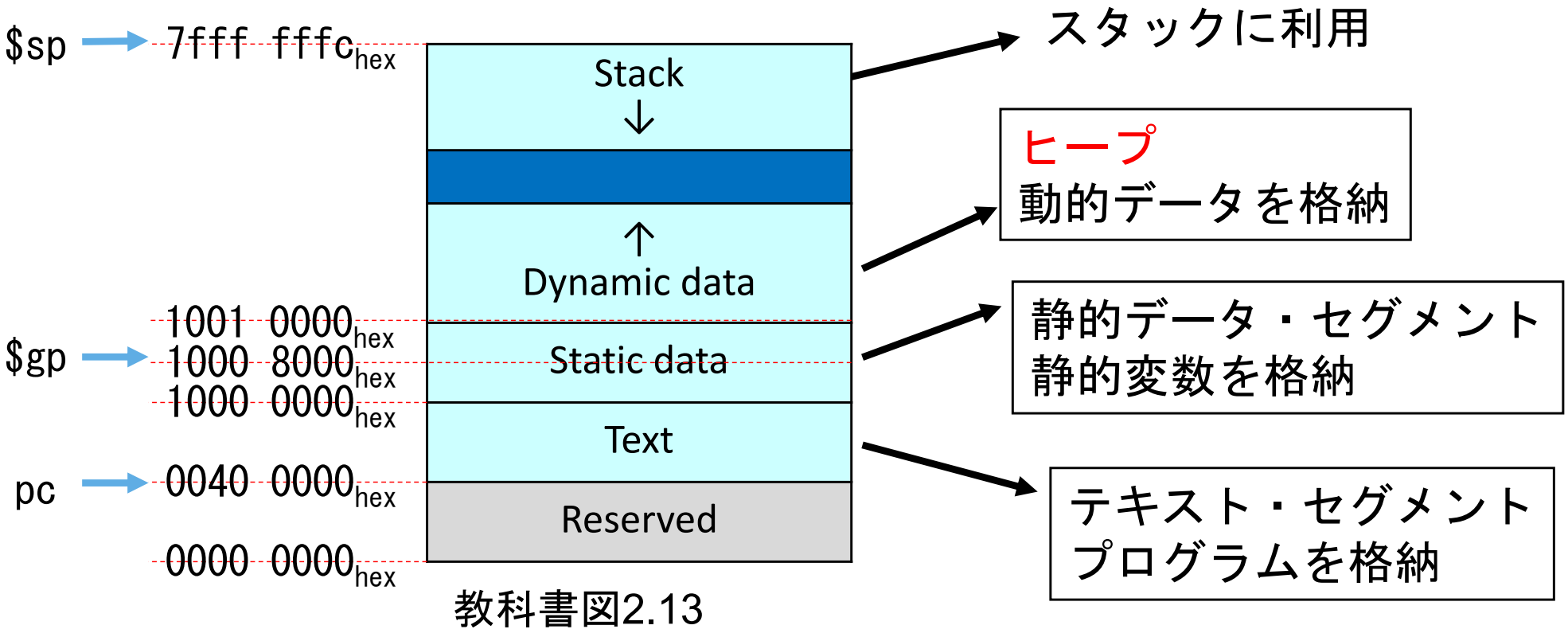
•メモリ上でのプログラムとデータの配置
•スタック
•手続きフレームとフレーム・ポインタ

- 教材：教科書2.8節

スタックの中身の詳細：図2.12



メモリ上でのプログラムとデータの配置



教科書図2.13

ミニクイズ：MIPSで以下のレジスタが保持するアドレスは通常は何か？

$pc：現在実行中の命令の格納アドレス

$gp：静的データのアクセスの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

Lec. 6での要チェック用語集

意味ぐらいはチェクすべし

Procedure
戻りアドレス
引数レジスタ
スタック
プッシュ
ポップ
スタック・ポインタ
静的データ
動的データ
フレーム・ポインタ
手続きフレーム
ヒープ
グローバル・ポインタ

演習問題 その②　（教科書そのままの内容）

以下のもので、被呼び出し側で保存と復元をするものは？

(point!) 被呼び出し側で保存と復元をしないものは、呼び出し側でもし必要があるならば、呼び出し側で保存と復元をする

- $s0-$s7
- $sp
- $ra
- スタックポインタより上のデータ
- スタックポインタより下のデータ
- $t0-$t9
- $a0-$a3
- $v0-$v1

教科書に答えはありますが、解説を少し書きます。自分で理解してください。

- $s0-$s7
- $sp
- $ra
- スタックポインタより上のデータ

これらだけ、被呼び出し側で、（もし自分が書き換えるなら）保存と復元をしないといけない

- スタックポインタより下のデータ
- $t0-$t9
- $a0-$a3
- $v0-$v1

とりあえず、呼び出し側、被呼び出し側でスライド22のように役割分担していますが、実際は、「破壊されたら困る情報は、破壊されるまえに保存して、次に使用する前に復元」すれば、どちら側でやっても問題はありません。つまり，**教科書の方針は、とりあえずの目安ということで理解してください。**

（ポイント）

- スタックポインタより下（教科書やスライドの図で下）のデータは、被呼び出し側が使うスタック領域（スタックフレーム）は呼び出し側に戻るともう誰も使わない→つまり保存の必要なし。
- $v0-$v1　は返り値を入れるレジスタ。保存などそもそも不要
- それ以外は、被呼び出し側で一時的に書き込んで破壊するのにあとで使用する場合は、「保存と復元」が必要

- その中で、スライド22にあるとおり、$t0-$t9、$a0-$a3　は、呼び出し側で、（もし被呼び出し側が書き換えて困る場合は）保存と復元をする約束にしている。

演習問題 その③

関数の呼び出し時のメモリ操作の手順をまとめよ（以下は回答例）

1. jalを実行する前に、呼び出し側は、呼出し後に必要となるレジスタの(退避が必要な引数レジスタと一時レジスタ)内容をスタックに［　　］
2. 引数を$a0-$a3に用意して、［　　］を実行（戻りアドレスが設定される）
3. 呼び出された側は、自分が使用する退避レジスタ $s0-$s7, 戻りアドレス［　　］をスタックに［　　］
4. 呼び出された側は、自分が使う一時的な変数はスタック上に確保(［　　］を更新)して使用。もし、大域変数を使用する場合は、［　　］の値を用いてそのアドレスを計算してアクセス
5. 呼び出された側は、計算が終わると、以下の処理をしてから［　　］
   - 自分が呼ばれた時の戻りアドレスをスタックから $ra に復元
   - 必要なレジスタの値をスタックから復元し，スタックを解放（$fpの値を用いて $sp を復元）
6. 呼び出し側は1.で保存した値を［　　］から復元して処理を進める。

＊上記のステップが、演習④のどこにあたるかを考えてみよう

（以下の演習④は呼び出し・呼び出され側が同じコードになっているので必ずしも上記の原則通りではないかもしれないことを注意。またこれらは必ずしもこの順序でなくても良いものもあるので、順序は目安です。）

演習問題 その④

教科書p99の例題

次の階乗計算のC言語のステートメントを、MIPSコードに変換せよ

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

これは呼び出し側でもあり被呼び出し側でもあるため、少し複雑であるが、どのような動作をすればいいのかを考えて理解しましょう。

(似たような問題の穴埋めとか試験に出るかも)

パラメータ変数nは引数レジスタ$a0に格納
戻り値は、$v0に格納

`mul  $s0,$s1,$s2` は乗算命令

演習問題 その④追加

変換したＭＩＰＳコードの各行で，演習③の1から6のどの部分に対応しているかを考えよ．

解答:　これをいきなり全部書けという問題はでないけど、
　　一部の空欄埋めとかの問題はありえる

```
fact :
    addi  $sp,$sp,-8      # スタックに2語分のスペースをとる
    sw    $ra,4($sp)      # 戻りアドレスの退避
    sw    $a0,0($sp)      # 引数レジスタ$a0の退避
    slti  $t0,$a0,1       # n<1であるかをチェック
    beq   $t0,$zero,L1    # n≧1であれば、L1に分岐
    addi  $v0,$zero,1     # $v0に1を返す.
    addi  $sp,$sp,8       # スタックから2語分のスペースを落とす
    jr    $ra             # 呼び出し元に戻る
L1: addi  $a0,$a0,-1      # n=n-1
    jal   fact            # 引数n-1を使ってfactを呼び出す
    lw    $a0,0($sp)      # 引数nの復元
    lw    $ra,4($sp)      # 戻りアドレスを復元する
    addi  $sp,$sp,8       # スタックから2語分のスペースをおとす
    mul   $v0,$a0,$v0     # n * fact(n - 1)を返す
    jr    $ra             # 呼び出し元に戻る
```

プッシュ操作

$spの復元

ここで再帰終了

ポップ操作

演習問題 その④追加
変換したＭＩＰＳコードの各行で，演習③の1から6のどの部分に対応しているかを考えよ．

演習問題　その④のアセンブリコード

```
fact :
    addi $sp,$sp,-8      # スタックに2語分のスペースをとる
    sw   $ra,4($sp)      # 戻りアドレスの退避                3
    sw   $a0,0($sp)      # 引数レジスタ$a0の退避             1
    slti $t0,$a0,1       # n<1であるかをチェック
    beq  $t0,$zero,L1    # n≧1であれば、L1に分岐

    addi $v0,$zero,1     # $v0に1を返す．
    addi $sp,$sp,8       # スタックから2語分のスペースを落とす 5
    jr   $ra             # 呼び出し元に戻る                  5

L1: addi $a0,$a0,-1      # n=n-1                             2
    jal  fact            # 引数n-1を使ってfactを呼び出す     2
    lw   $a0,0($sp)      # 引数nの復元                       6
    lw   $ra,4($sp)      # 戻りアドレスを復元する            5
    addi $sp,$sp,8       # スタックから2語分のスペースをおとす 5
    mul  $v0,$a0,$v0     # n * fact(n - 1)を返す
    jr   $ra             # 呼び出し元に戻る                  5
```

操作手順（演習問題　その③）

関数の呼び出し時のメモリ操作の手順をまとめよ（以下は回答例）

1. jalを実行する前に、呼び出し側は、呼出し後に必要となるレジスタの(退避が必要な引数レジスタと一時レジスタ)内容をスタックにプッシュ
2. 引数を$a0-$a3に用意して、jal　を実行（戻りアドレスが設定される）
3. 呼び出された側は、自分が使用する退避レジスタ$s0-$s7, 戻りアドレス$ra　をスタックにプッシュ
4. 呼び出された側は、自分が使う一時的な変数はスタック上に確保($sp を更新)して使用。もし、大域変数を使用する場合は、$gp の値を用いてそのアドレスを計算してアクセス
5. 呼び出された側は、計算が終わると、以下の処理をしてからjr $ra
   - 戻りアドレスを（自分が呼ばれた時の）$raにスタックから復元
   - 必要なレジスタの値をスタックから復元し，スタックを解放（$fpの値を用いて$spを復元）
6. 呼び出し側は1. で保存した値をスタックから復元して処理を進める。

＊上記のステップが、演習④のどこにあたるかを考えてみよう