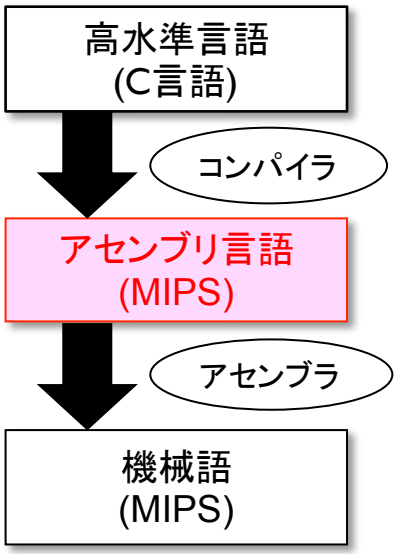
2012年度
計算機システム演習　第5回
2012.05.25

白幡　晃一



今日の内容

サブルーチンの実装

Outline

- ジャンプ、分岐命令
  - j, jr, jal
- レジスタ衝突、回避
  - caller-save
  - callee-save

分岐命令 (復習)

- **j label**
  - Jump
  - ラベルの命令へジャンプ

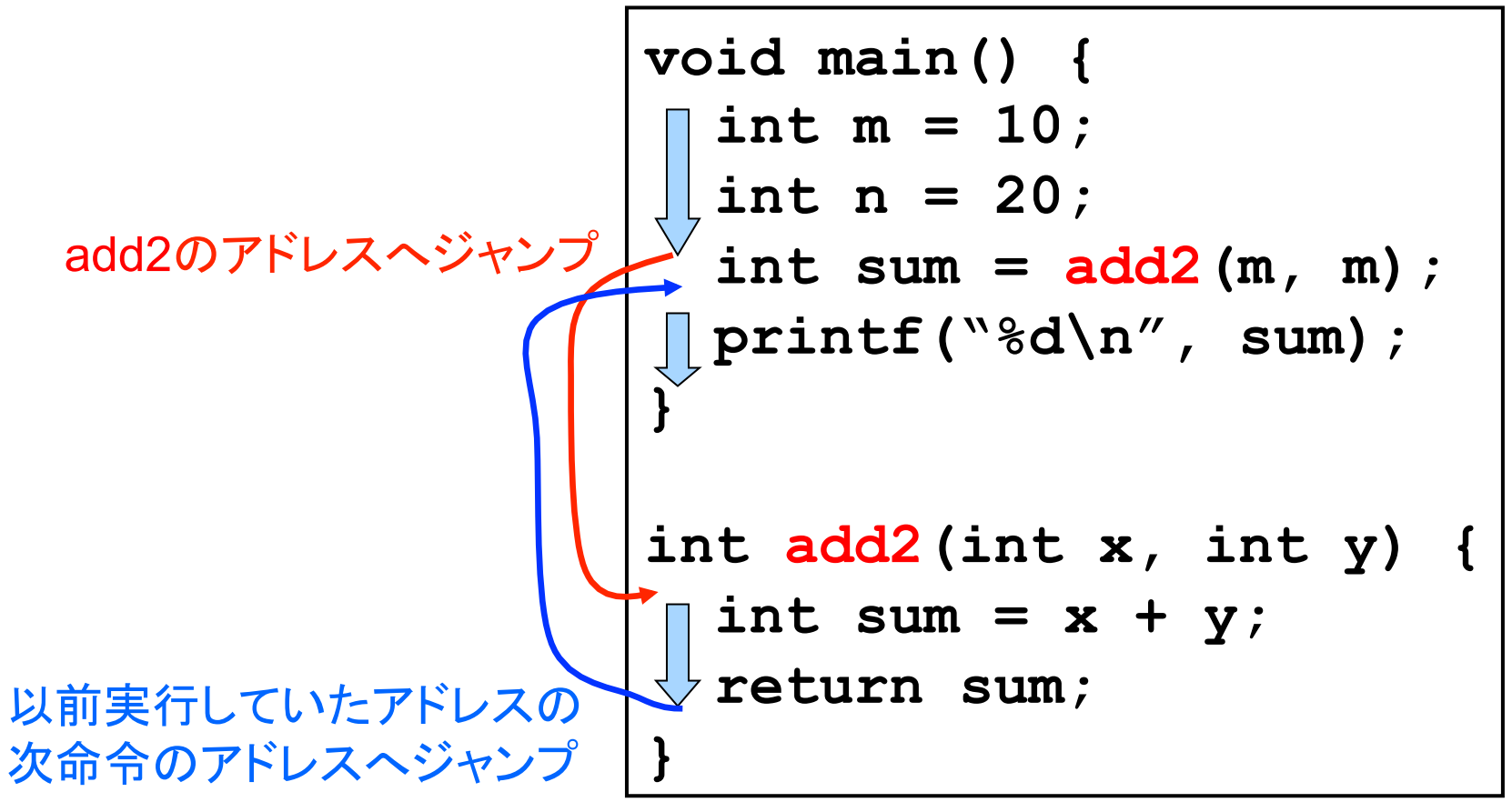
```
        j next
          :
next:
          :
```

- **jr $A**
  - Jump Register
  - レジスタ $A の値の指すアドレスにジャンプ
  - 例: jr $ra

```
      la   $t0, next
      jr   $t0
        :
next:
```

サブルーチン (C言語)

- mainルーチンからサブルーチンadd2を呼び出す



サブルーチン呼び出し

- **jal Label**
  - **Jump and Link**
  - Labelにジャンプすると同時に $raに次の命令のアドレス（リターンアドレス）を保存

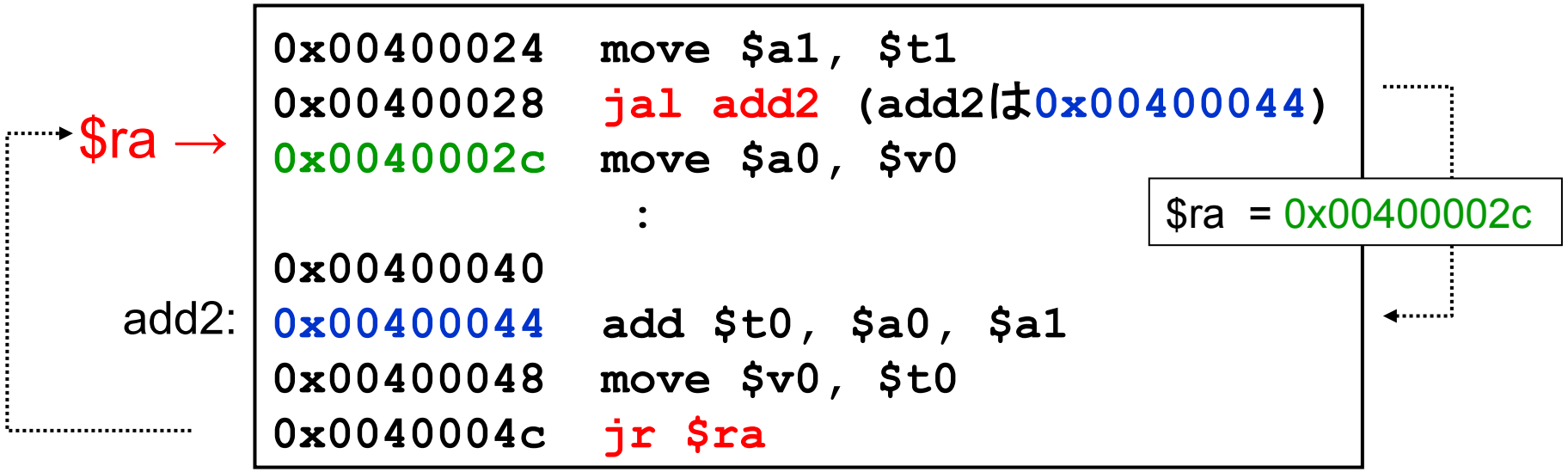
  ※j Labelは$raの内容を変えない

- サブルーチン呼び出し用レジスタ
  - 引数：$a0 ～ $a3
  - 返り値：$v0 ～ $v1
  - syscallと同じような使い方
- add2: $v0 = $a0+$a1

```
        .text
main:
        li      $t0, 10     # m=10
        li      $t1, 20     # n=20
        move    $a0, $t0
        move    $a1, $t1
        jal     add2        # call

        move    $a0, $v0
       # syscallで出力

        move    $a0, $t0
        move    $a1, $t1
        jal     add2        # call

        move    $a0, $v0
      # syscallで出力

      jr    $ra

add2:               # a0->x, a1->y
        add     $t0, $a0, $a1
        move    $v0, $t0
        jr      $ra
```

サブルーチンからの復帰

- **jr $ra**
  - $ra レジスタの指すアドレスにジャンプして戻る



レジスタの衝突

- 呼び出し元と呼び出し先で同じレジスタを使う場合
  - サブルーチンが呼び出し元が使っているレジスタを上書き（$t0）
- 別のレジスタを使用すればよい？
  - 外部命令を呼び出す場合は？

```
main:
        li      $t0, 10
        li      $t1, 20
        move    $a0, $t0    #$t0=10
        move    $a1, $t1
        jal     add2
          :（syscallでプリント）
        move    $a0, $t0    #$t0=?
        move    $a1, $t1
        jal     add2
          :
add2:           # a0->x, a1->y
        add     $t0, $a0, $a1
        move    $v0, $t0
        jr      $ra
```

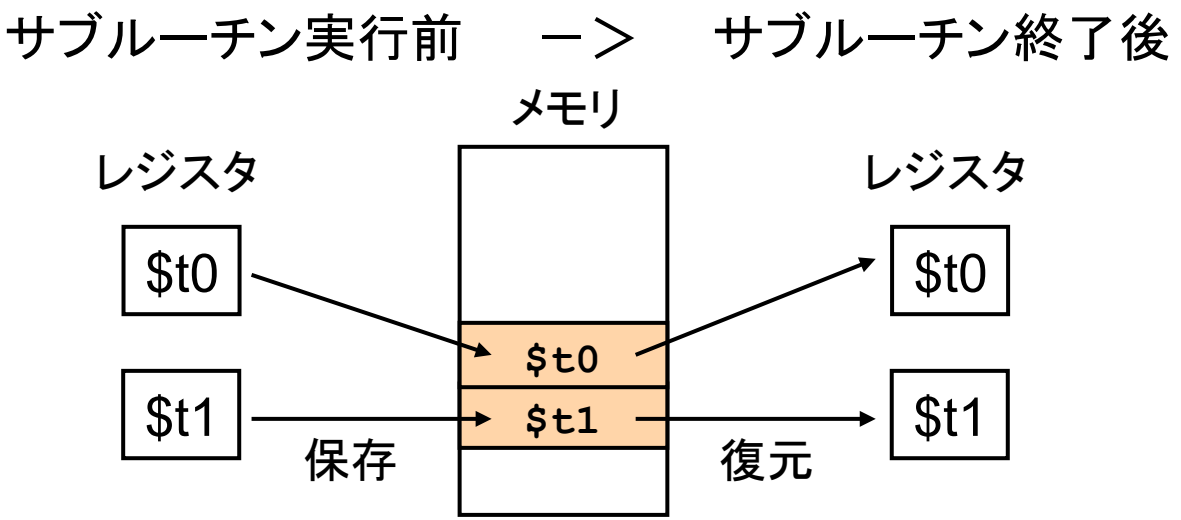
$t0を上書き

レジスタの使用規則

- **規則に違反したからといってプログラムが実行できないわけではない**
  - もちろん意図しない動作をする可能性はある
- **一時レジスタには2種類の使用規則がある**
  - ルーチン内で使用する場合、上書きしてよいレジスタ
    - 上書きされたくない場合、サブルーチンを呼び出す前に退避が必要 (caller-save)
    - $t0 ~ $t9, $v0 ~ $v1, $a0 ～$a3
  - ルーチン内で使用する場合、上書きする前に退避しなければならないレジスタ
    - 使用する場合、ルーチン内で退避が必要 (callee-save)
    - $s0～$s7, $ra

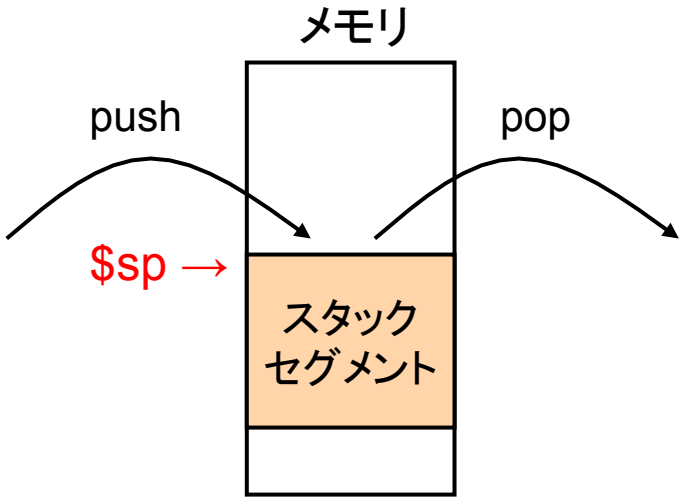
レジスタの退避（caller-save）

- **呼び出し元**がサブルーチンを呼ぶ前にメモリにレジスタの内容を保存し、終了したら元に戻す



レジスタ退避場所：スタックセグメント

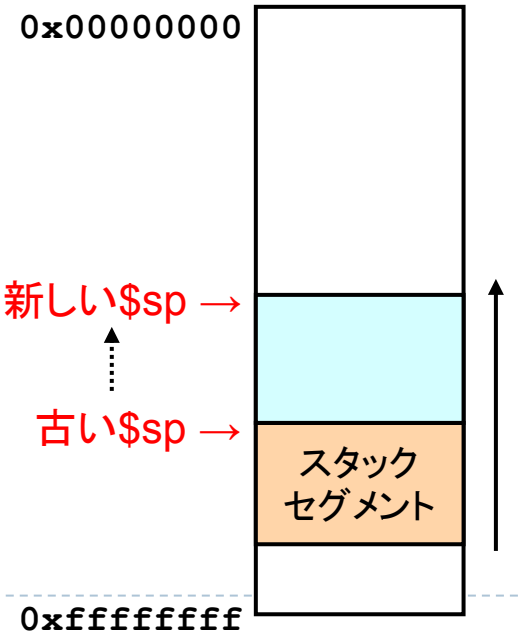
- メモリのどこにレジスタを保存するか？
  - 保存用のスタックが用意されている　=> スタックセグメント
    - 保存する時にスタックに積む（push）
    - 復元する時にスタックから取り出す（pop）
  - $sp レジスタがスタックの先頭アドレスを指している
    - スタックの先頭に push, pop すればよい



レジスタの退避

- caller-saveではサブルーチン実行前に行う
- レジスタ退避方法

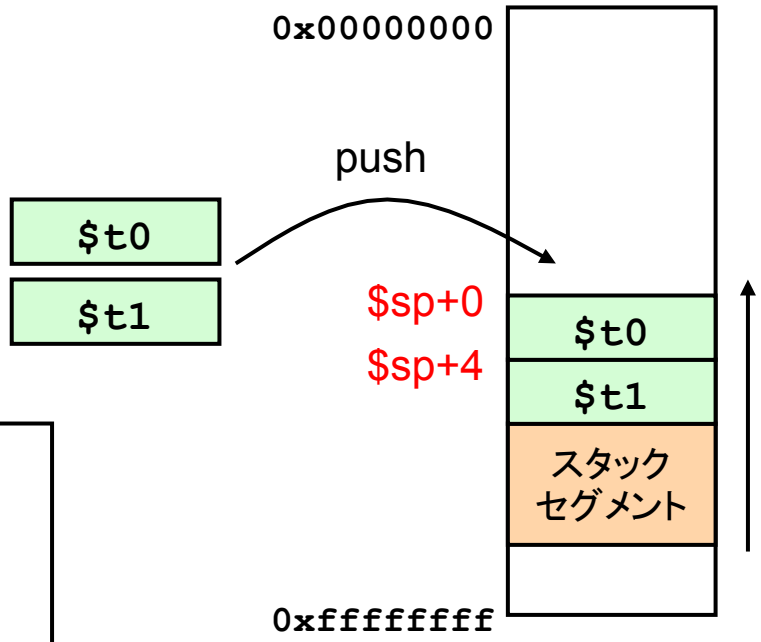
1. スタックに push する場所を確保
   - `addi $sp, $sp, -n`
   - m 個のレジスタを push する時
     n = m*4
     - レジスタ： 4バイト
   - -n するのはスタックはアドレス
     の高い方（上位）から低い方（下位）へ
     伸びるため



レジスタの退避　(cont'd)

2. スタックにレジスタの値を push する

- **sw $x, d($sp)**
- スタックの先頭から d = 0, 4, ... でアクセスできる

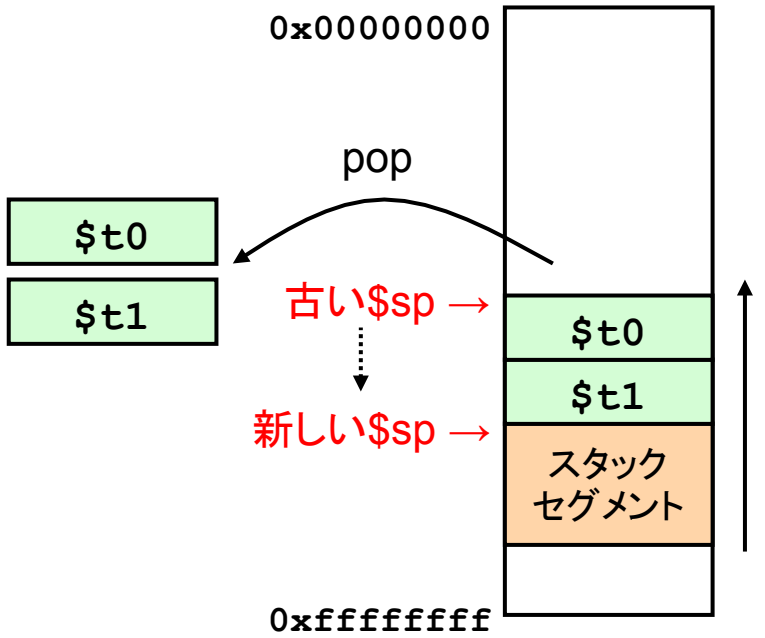



```
main:
        addi    $sp, $sp, -8   # 2レジスタ*4byte
          :
        sw      $t0, 0($sp)    # push $t0
        sw      $t1, 4($sp)    # push $t1
        jal     add2
          :
```

レジスタの復帰

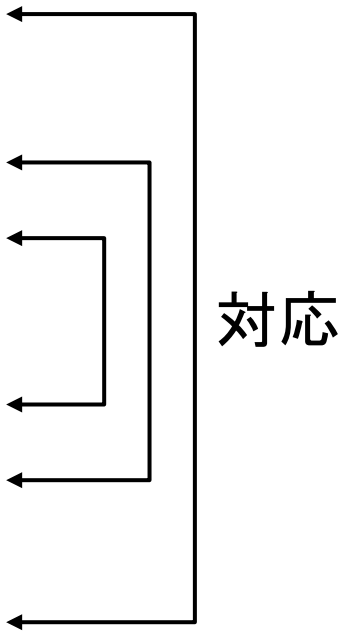
- サブルーチン終了後に行なう
- スタックから pop した値をレジスタに代入
  1. スタックから値を読み出す
     - `lw $x, d($sp)`
       - d = 0, 4, …
  2. 不要になったスタック領域を開放する
     - `addi $sp, $sp, n`

```
    :
jal     add2
lw      $t1, 4($sp)     # pop $t1
lw      $t0, 0($sp)     # pop $t0
    :
addi    $sp, $sp, 8     # 2レジスタ*4byte
jr      $ra
```



レジスタ保存のコード例

```
main:
      addi $sp, $sp, -8   # 2レジスタ*4byte
        :
      sw   $t0, 0($sp)    # push $t0
      sw   $t1, 4($sp)    # push $t1
      jal  add2
      lw   $t1, 4($sp)    # pop $t1
      lw   $t0, 0($sp)    # pop $t0
        :
      addi $sp, $sp, 8    # 2レジスタ*4byte
      jr   $ra
add2:
        :
```



add2を修正($t0, $t1を退避)

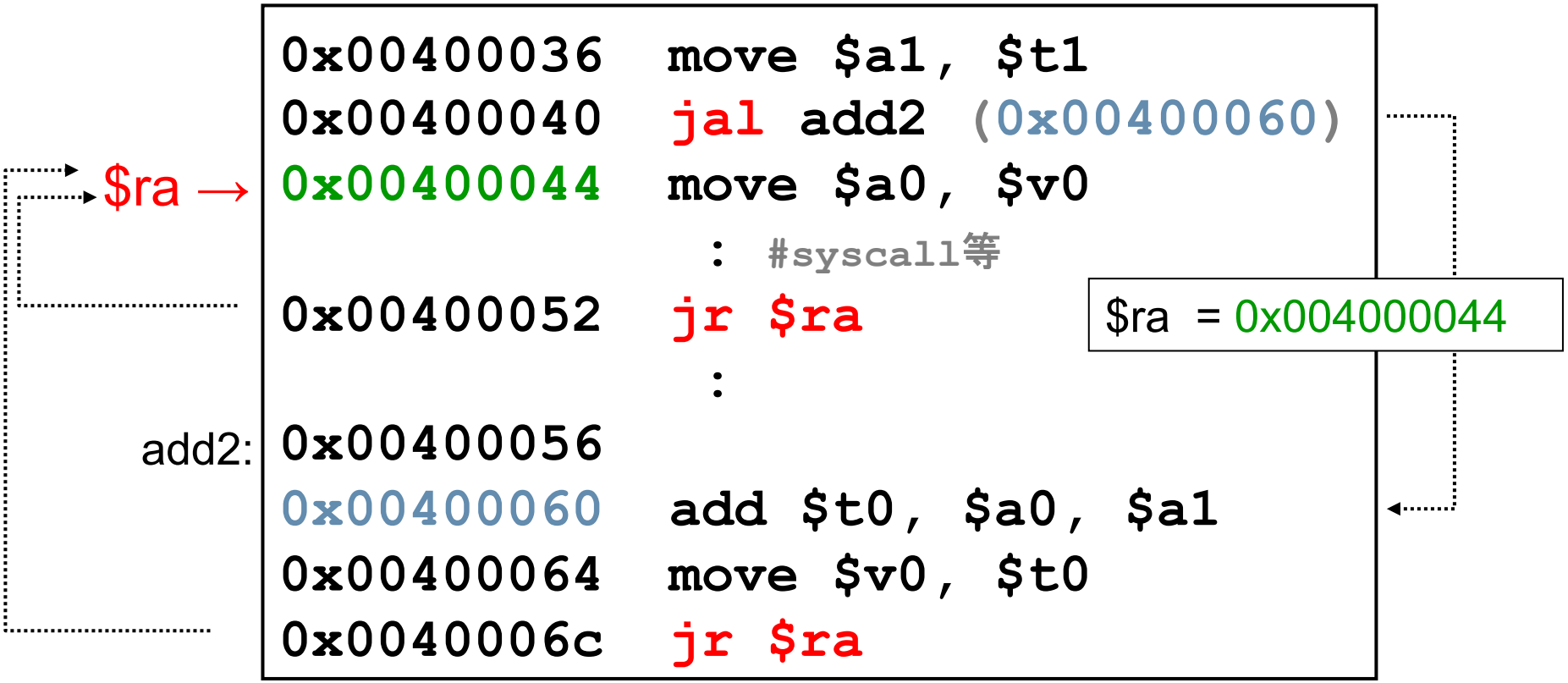
m6.s

```
  .text
main:
  addi  $sp, $sp, -8  # 退避領域作成

  li    $t0, 10
  li    $t1, 20

  move  $a0, $t0        # sum = add2(m, m)
  move  $a1, $t0
  sw    $t0, 0($sp)
  sw    $t1, 4($sp)
  jal   add2
  lw    $t1, 4($sp)
  lw    $t0, 0($sp)
  move  $a0, $v0        # printf
  li    $v0, 1
  syscall
```

```
  move  $a0, $t0        # sum = add2(m, n)
  move  $a1, $t1
  sw    $t0, 0($sp)
  sw    $t1, 4($sp)
  jal   add2
  lw    $t1, 4($sp)
  lw    $t0, 0($sp)
  move  $a0, $v0        # printf
  li    $v0, 1
  syscall

  addi  $sp, $sp, 8   # 領域開放
  jr    $ra
add2:
  add   $t0, $a0, $a1
  move  $v0, $t0
  jr    $ra
```

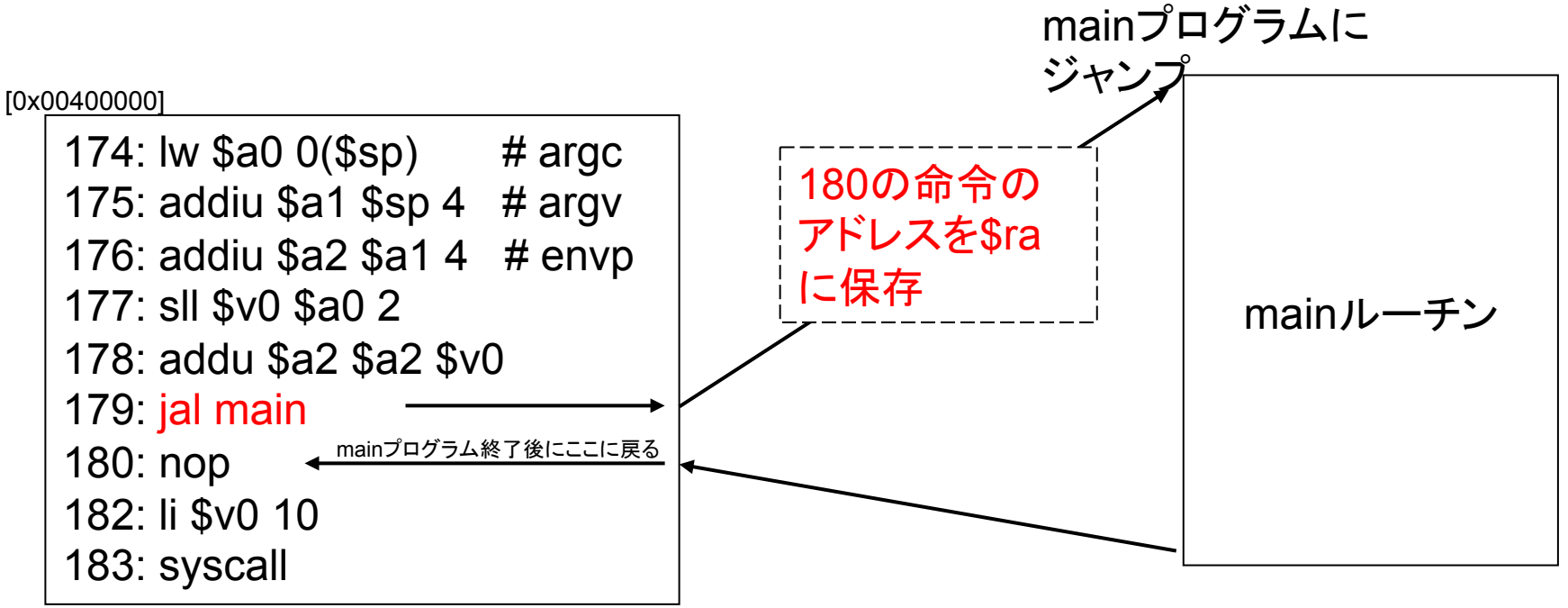
サブルーチンからの復帰

- mainルーチンの戻りアドレスが上書きされてしまう



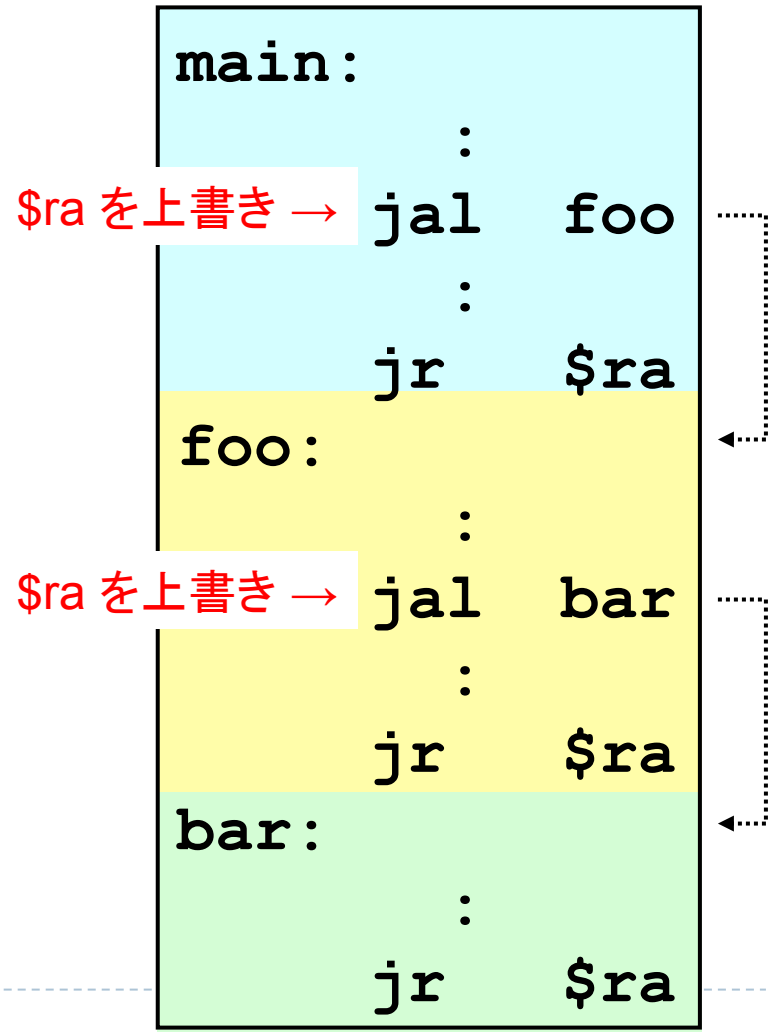
mainルーチンの流れ

main(argc, argv, envp)



サブルーチンを呼ぶ場合 $ra の退避

- $raはcallee-save
  - mainルーチンもサブルーチン
- jal命令を呼ぶ時, $ra を退避
  - jal 命令で $ra が上書きされてしまうため



完全なレジスタ保存コード例

```
main:
        :
      addi $sp, $sp, -12
      sw   $ra, 0($sp)
        :
      sw   $t0, 4($sp)
      sw   $t1, 8($sp)
      jal  bar
      lw   $t1, 8($sp)
      lw   $t0, 4($sp)
        :
      lw   $ra, 0($sp)
      addi $sp, $sp, 12
        :
```

- $ra をサブルーチンの先頭で退避
  - 最後で復帰
  - サブルーチンを呼ばないときは不要
- 必要に応じて$a0 ～ $a3も退避
  - 呼び出し元ルーチンに引数が無い場合は不要

add2 (完成版)

m7.s

```
  .text
main:
  addi  $sp, $sp, -12   # 退避領域作成
  sw    $ra, 0($sp)     # ra退避
  li    $t0, 10
  li    $t1, 20

  move  $a0, $t0        # sum = add2(m, m)
  move  $a1, $t0
  sw    $t0, 4($sp)
  sw    $t1, 8($sp)
  jal   add2
  lw    $t1, 8($sp)
  lw    $t0, 4($sp)
  move  $a0, $v0        # printf
  li    $v0, 1
  syscall
```

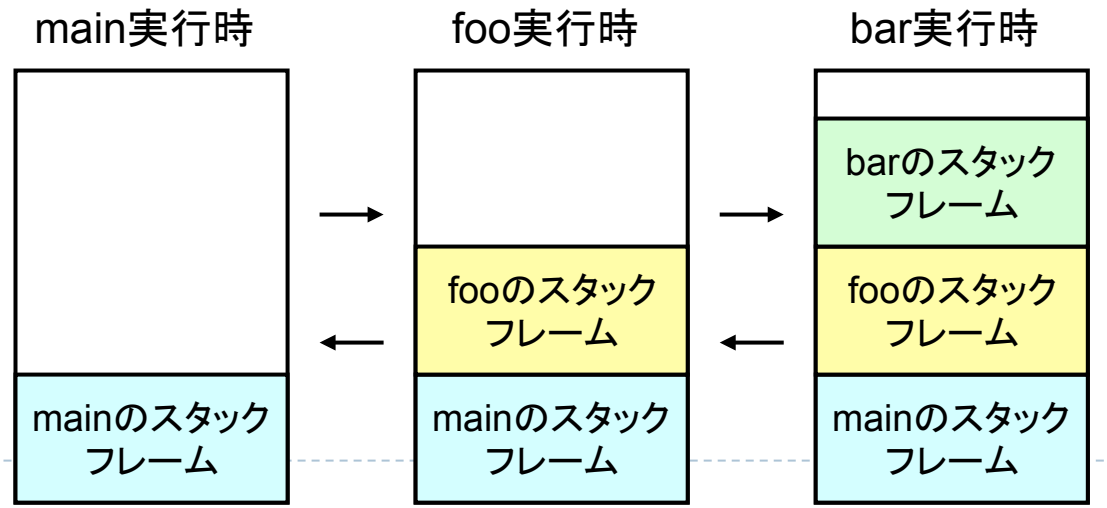
```
  move  $a0, $t0        # sum = add2(m, n)
  move  $a1, $t1
  sw    $t0, 4($sp)
  sw    $t1, 8($sp)
  jal   add2
  lw    $t1, 8($sp)
  lw    $t0, 4($sp)
  move  $a0, $v0        # printf
  li    $v0, 1
  syscall

  lw    $ra, 0($sp)     # ra復帰
  addi  $sp, $sp, 12    # 領域開放
  jr    $ra
add2:
  add   $t0, $a0, $a1
  move  $v0, $t0
  jr    $ra
```

復習：スタックフレーム

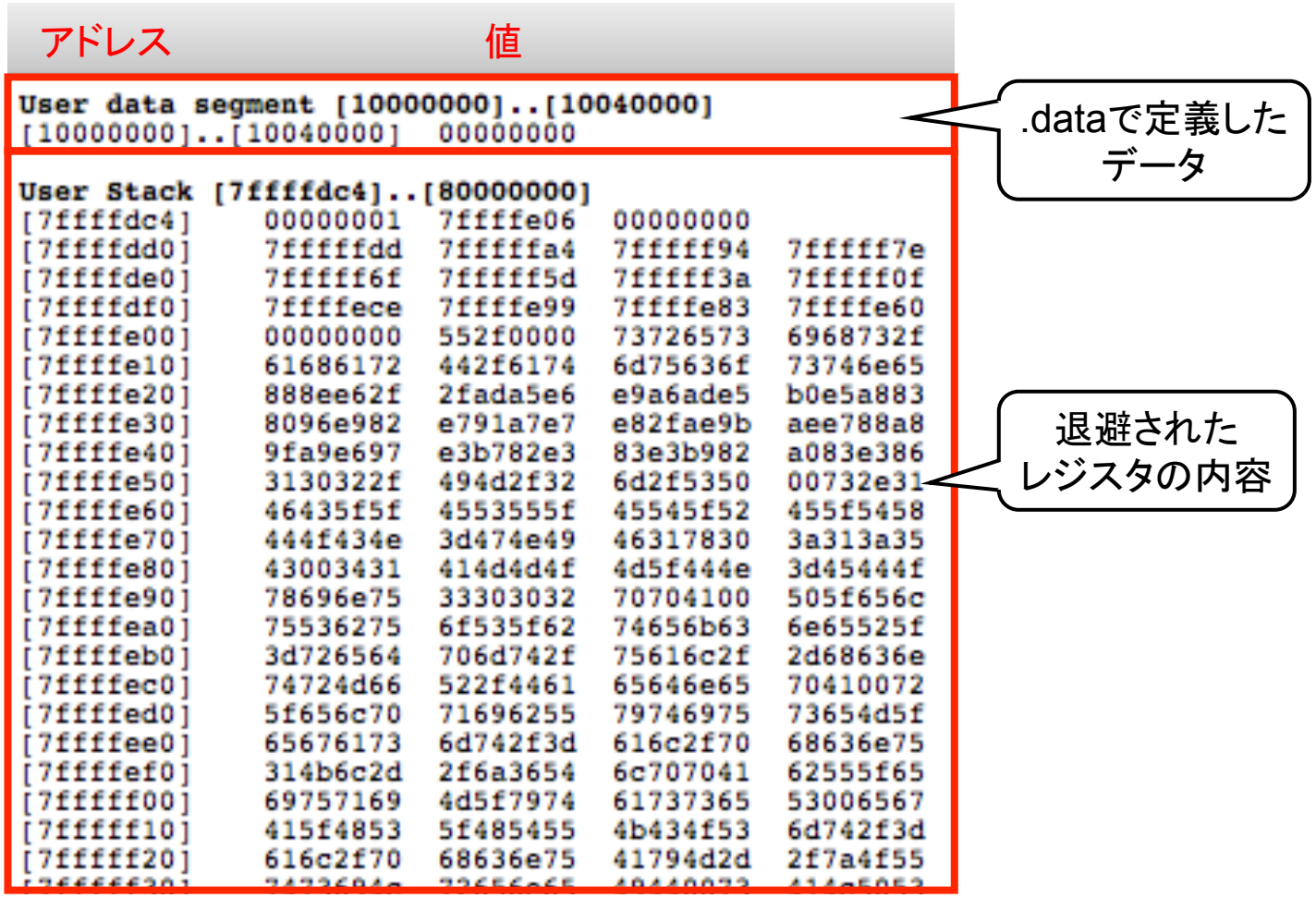
- １つのサブルーチンが使うスタック領域
  - 退避されたレジスタの内容
    - $ra, $t0, $t1, ...
  - ローカル変数
    - そのサブルーチンの中でだけ使える変数
    - 高級言語が使用
  - 他のサブルーチンを呼ぶ時の5つ目以降の引数
    - 4つ目まではレジスタ $a0 ～ $a3 に入れられる

スタックフレームの作成・破棄

- サブルーチンを呼ぶ時にスタックフレームを作り、終了した時に破棄する
  - サブルーチンの最初と最後で $sp のアドレスを変更することで明示的に行う
    - **`addi $sp, $sp, ±n`**



QtSpimにおけるスタック



Callee-save

- **呼び出される**サブルーチンが、自分が使用するレジスタの値を退避・復帰する
  - 「**callee-save**」という
  - レジスタ **$s0 ～ $s7**
    - 呼び出し元では$s0 ～ $s7は自由に使える
  - $raも callee-save

add2をcallee-saveで実装

```
 .text
main:
  addi  $sp, $sp, -12 # 退避領域作成
  sw    $ra, 0($sp)   # raはcallee save
  sw    $s0, 4($sp)   # s0はcallee save
  sw    $s1, 8($sp)   # s1はcallee save

  li    $s0, 10
  li    $s1, 20

  move  $a0, $s0      # sum = add2(m, m)
  move  $a1, $s1
  jal   add2
  move  $a0, $v0      # printf
  li    $v0, 1
  syscall

  move  $a0, $s0      # sum = add2(m, n)
  move  $a1, $s1
  jal   add2
  move  $a0, $v0      # printf
  li    $v0, 1
  syscall

  lw    $s1, 8($sp)   # s1復帰
  lw    $s0, 4($sp)   # s0復帰
  lw    $ra, 0($sp)   # ra復帰
  addi  $sp, $sp, 12   # 領域開放
  jr    $ra

add2:
  addi  $sp, $sp, -4
  sw    $s0, 0($sp)

  add   $s0, $a0, $a1
  move  $v0, $s0

  lw    $s0, 0($sp)
  addi  $sp, $sp, 4
  jr    $ra
```

まとめ：caller-save

- 上書きする場合、退避せず自由に使ってよい
  - 一時レジスタ：$t0～$t9
  - 戻り値レジスタ：$v0～$v1

```
foo:
        addi   $sp, $sp, -8
           :
        # $t0に代入
           :
        sw     $t0, 4($sp)
        jal    bar
        lw     $t0, 4($sp)
           :
        # $t0を使った処理
           :
        addi   $sp, $sp, 8
        jr     $ra

bar:
        # $t0を使った処理
```

まとめ：callee-save

- 上書きする場合、退避が必要
  - 退避レジスタ: $s0～$s7
  - 引数レジスタ: $a0 ~ $a3
  - 戻りアドレスレジスタ: $ra

```
foo:
        jal    bar

bar:
        addi   $sp, $sp, -4
        sw     $s0, 0($sp)
           :
        # $s0を使った処理
           :
        lw     $s0, 0($sp)
        addi   $sp, $sp, 4
        jr     $ra
```

まとめ：一時レジスタ($t, $s) の違い

- t レジスタ (t0-t9)
  - ルーチン内で使用する場合、上書きしてよい

  ⇒ サブルーチンを呼び出す前に退避が必要 (caller-save)

- s レジスタ (s0-s8)
  - ルーチン内で使用する場合、上書きする前に退避

  ⇒ 使用する場合、ルーチン内で退避が必要 (callee-save)

- 使用するレジスタによって、プログラムのパフォーマンスに影響

まとめ：効率の違い

- レジスタを退避・復帰する回数の違い
  - どっちのレジスタ・退避法を使うかはパフォーマンス向上のカギ

プログラム例	<pre># $xに代入
	jal foo
	jal bar
	# $xを使って計算</pre>
		<pre># $xに代入
		jal foo
		# $xを使って計算
		jal bar
		# $xを使って計算</pre>

caller-save ($xの退避回数)	$xはfooの前で退避し、barの後で復帰（1回）	$xはfooとbarの前後で保存・復帰（2回）
callee-save ($xの退避回数)	$xの退避・復帰はfooとbarで使うかどうかに依存 1回(main)+0～2回 (foo, bar)	

課題

（遅れても採点いたします。。。）

課題1

- 1つの整数値を入力させ、入力された行数だけ数値を図のように表示するプログラムを書け
  - “num=“というプロンプトを表示させてから行数を入力させること
  - 1行を表示するサブルーチンを作り、繰り返し呼び出すこと
    - 例）$a0: 表示する値&表示文字数
  - $t0～$t9を使ってcaller-saveで実装せよ

```
num=5
55555
4444
333
22
1
```

- 補足：整数値入力の実装

```
li  $v0, 5
syscall
move $t0, $v0
```

← 実行後、$v0に入力値が入っている

課題2

- 以下のアルゴリズムに対応するアセンブリコードを書け
  - main ルーチンからサブルーチン(fact)を呼んで計算
    - nは入力プロンプトを設けること
  - かけ算命令は **`mul $x, $y, $z`**（$x = $y * $z）

```
int fact(int n) {
    if (n == 0)
        return 1;
    else
        return n * fact(n - 1);
}
```

課題3

- caller-save で書かれた次ページのコードを callee-save で書き直し、コードの効率の違いを論じよ
  - 0 または 1 を 8 回入力させて 8 ビットの2進数とみなし、それを 10 進数に変換するプログラム
    - (入力1)×2^7 + (入力2)×2^6 + (入力3)×2^5 + (入力4)×2^4 +(入力5)×2^3 + (入力6)×2^2 + (入力7)×2^1 + (入力8)×2^0
    - 結果を2倍しながら入力を足していく
- 注意:
  - callee-saveなのでサブルーチン内で使わないレジスタを退避する必要はない
  - main ルーチンもサブルーチンであることに注意せよ
    - main で $s0～$s7 を使うなら、最初と最後で退避・復帰する必要がある

２進１０進変換プログラム

```
        .text
main:
        addi $sp, $sp, -12
        sw $ra, 0($sp)

        li $t0, 0   # 結果
        li $t1, 0   # i=0
loop:
        sw $t0, 4($sp)
        sw $t1, 8($sp)
        jal read1bit
        lw $t1, 8($sp)
        lw $t0, 4($sp)

        sll $t0, $t0, 1   # 2倍
        add $t0, $t0, $v0
        addi $t1, $t1, 1
        blt $t1, 8, loop

        move $a0, $t0
        li $v0, 1
        syscall

        lw $ra, 0($sp)
        addi $sp, $sp, 12
        jr $ra

# サブルーチン
read1bit:
        li $v0, 5
        syscall   # read_int
        jr $ra
```

課題提出

- 〆切 : 6/8 (金) 23:59
- 提出物 : 以下のファイルを圧縮もの
  - ドキュメント(pdf,plain txt,wordなんでも可)
    以下の内容を含めること
    - 課題3の議論
    - 課題1,2,3の実行結果
    - (もしあれば) 感想、質問等
  - プログラムソース
    - 課題1,2,3 (適宜コメントを記述すること)