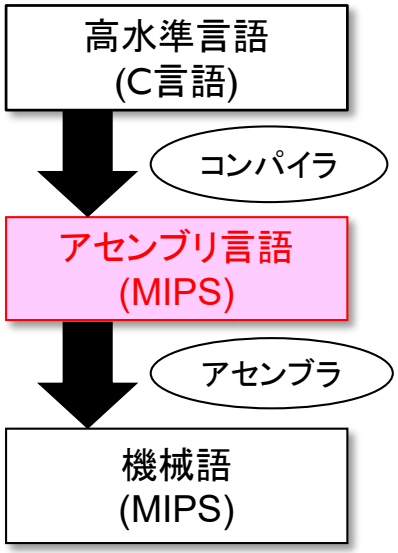
2018年度　計算機システム（演習）
第4回
2018.12.21

遠藤 敏夫（学術国際情報センター/数理・計算科学系 教授）
野村 哲弘（学術国際情報センター/数理・計算科学系 助教）



今日の内容

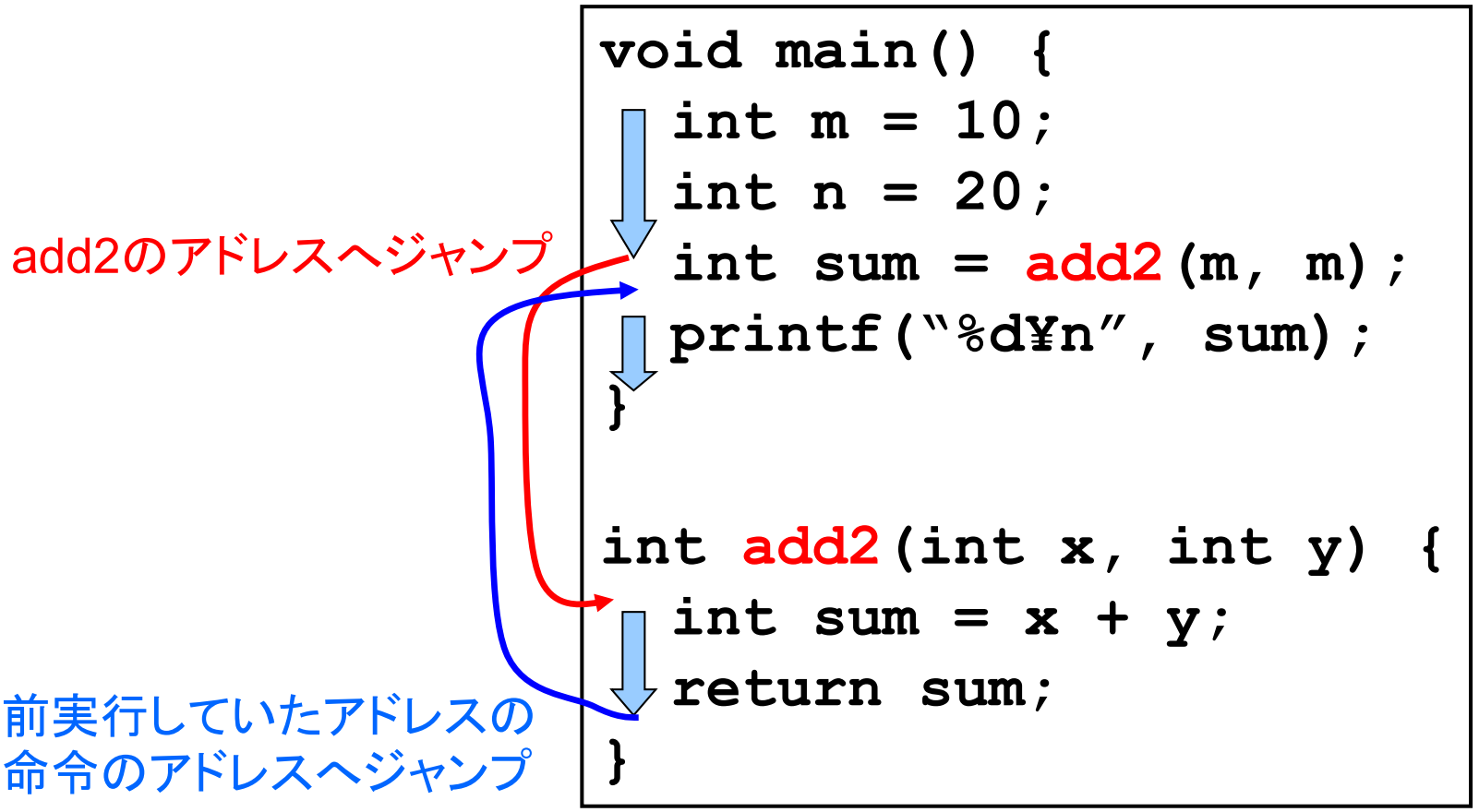
サブルーチンの実装、動的配列

Outline

- サブルーチンの実現
  - ジャンプ、分岐命令
    - `j, jr, jal`
  - レジスタ衝突、回避
    - caller-save
    - callee-save
      - c.f.) $spを用いてスタック領域上にデータを格納
- 動的メモリ割り当て: ヒープ領域にデータを確保
  - 動的配列： 動的に配列の領域を確保

C言語での関数呼び出し (復習)

- mainルーチンからサブルーチンadd2を呼び出す



C言語での関数呼び出し (復習)

- （C言語で）
  関数が呼び出されると何が起こるか？
  - 関数 ≒ 実体は関数へのポインタ
    ＝ 関数の命令列が格納されているアドレス

1. 引数が「渡される」
2. 関数のローカル変数（自動変数・局所変数）のための
   スタックフレームが作られる
3. 関数の命令の先頭に実行が移る
4. 関数がreturnすると、元の実行位置に戻ってくる
5. 呼び出し元の関数は、関数の戻り値を使って処理をする

MIPSでのサブルーチン呼び出し

- （MIPSで）
  サブルーチンが呼び出されると何が起こるか？
  - サブルーチン　≒　呼び出し規約に従って書かれた
    プログラムの一部分

1. 引数を$aレジスタに設定する（$a0, $a1,…）
2. スタックポインタ $sp （スタックのトップを指すレジスタ）を
   動かしてスタックフレームを確保する
3. 現在位置（呼び出し元）の位置を $ra に保存する
4. jal命令でサブルーチンの先頭に制御を移す
5. サブルーチンは戻り値を$vレジスタに入れる
6. jr $ra命令で元の場所に戻る

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

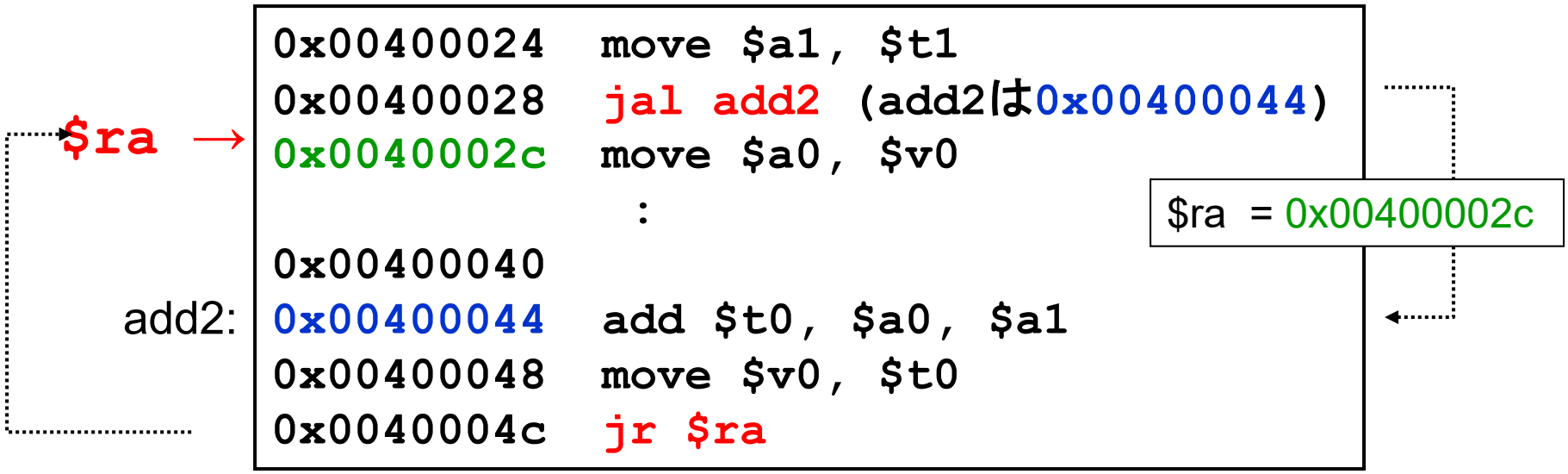
※ j Labelは$raの内容を変えない

- 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

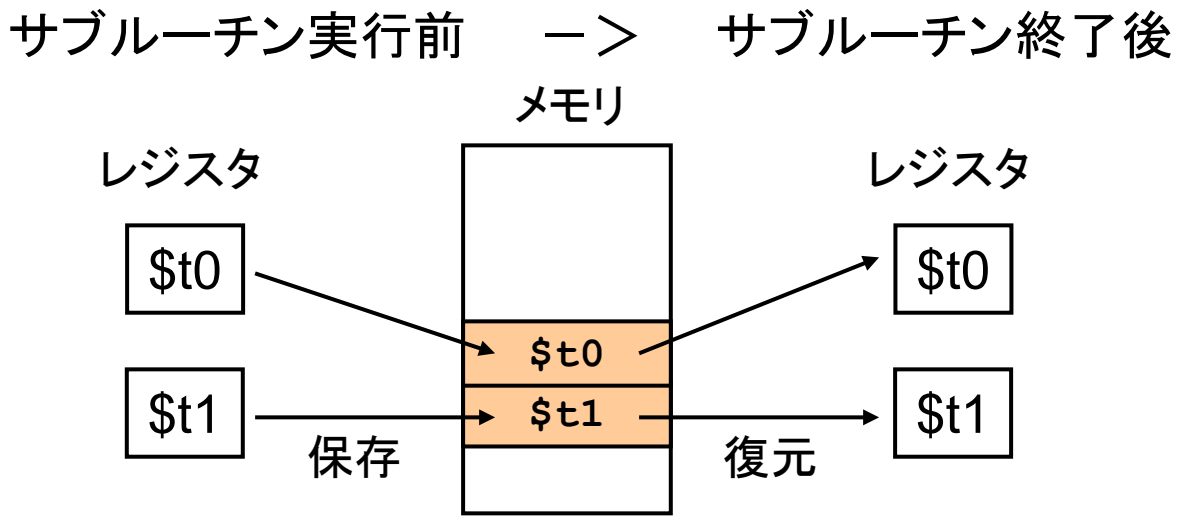


レジスタの使用規則

- **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`
- **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**
  - **意図しない動作をする可能性はある**

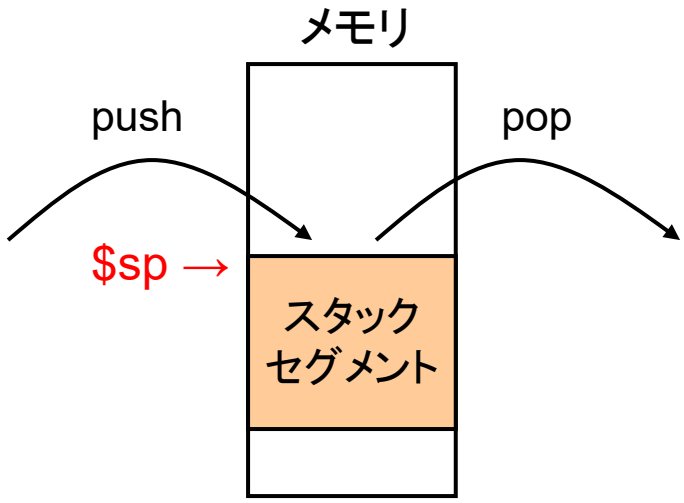
レジスタの退避 (caller-save)

- **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

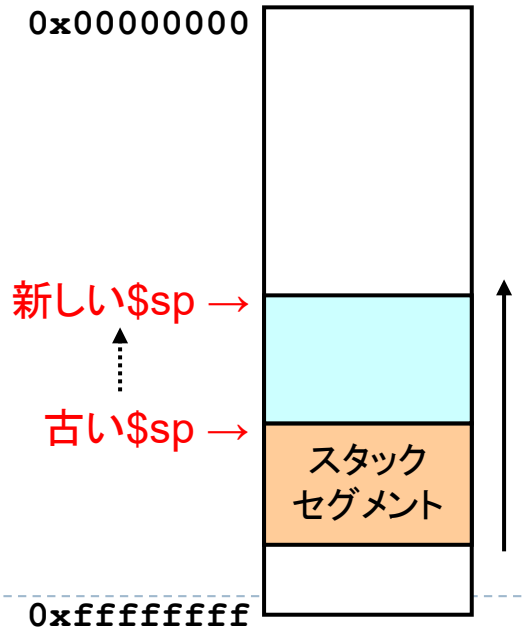
- メモリのどこにレジスタを保存するか？
  - 保存用のスタックが用意されている　=> スタックセグメント
    - 保存する時にスタックに積む（push）
    - 復元する時にスタックから取り出す（pop）
  - $sp レジスタがスタックの先頭アドレスを指している
    - スタックの先頭に push, pop すればよい



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

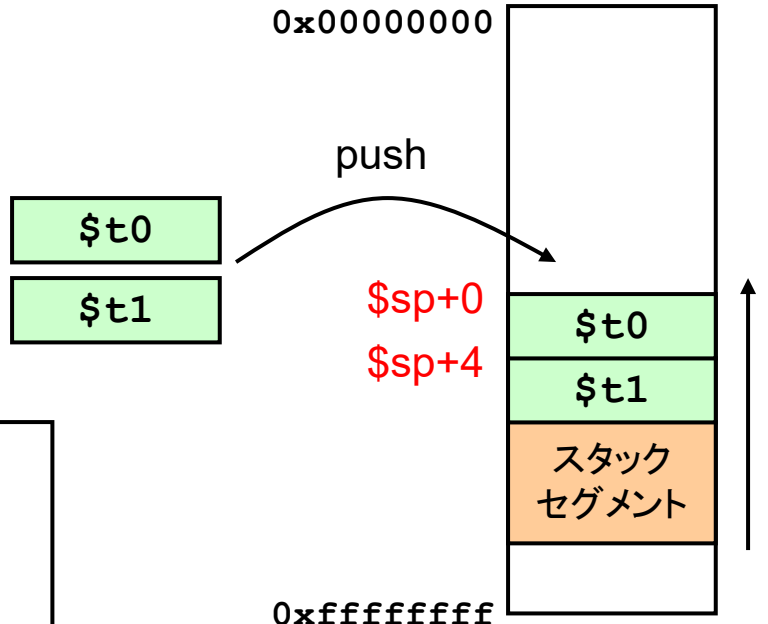
1. スタックに push する場所を確保
   - `addi $sp, $sp, -n`
   - m 個のレジスタを push する時
     n = m*4
     - レジスタ： 4バイト
   - -n するのはスタックはアドレス
     の高い方（上位）から低い方（下位）へ
     伸びるため



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

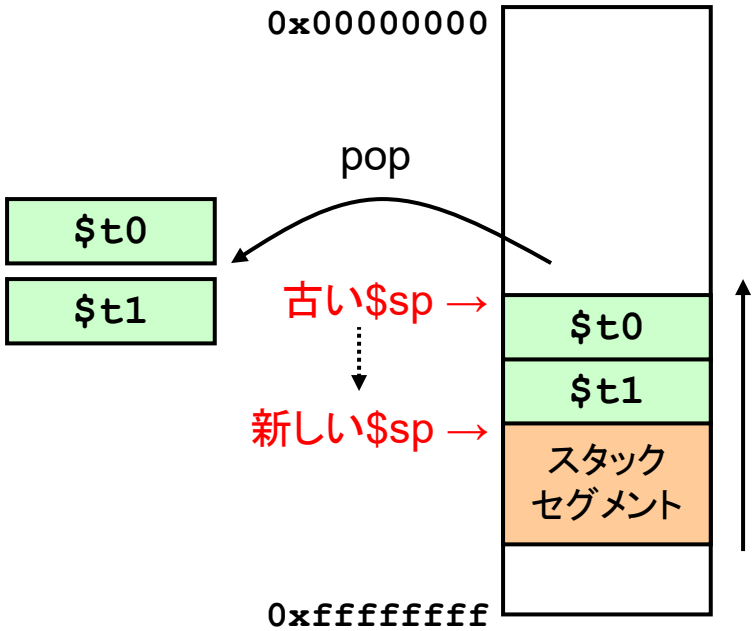
- **sw $x, d($sp)**
- スタックの先頭から d = 0, 4, ... でアクセスできる




```
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

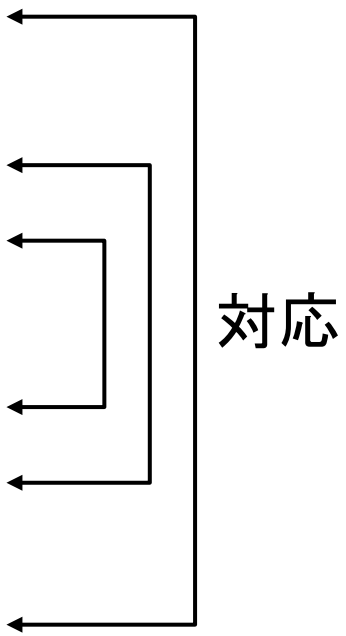
- サブルーチン終了後に行なう
- スタックから pop した値をレジスタに代入
  1. スタックから値を読み出す
     - `lw $x, d($sp)`
       - d = 0, 4, ...
  2. 不要になったスタック領域を開放する
     - `addi $sp, $sp, n`




```
   :
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)

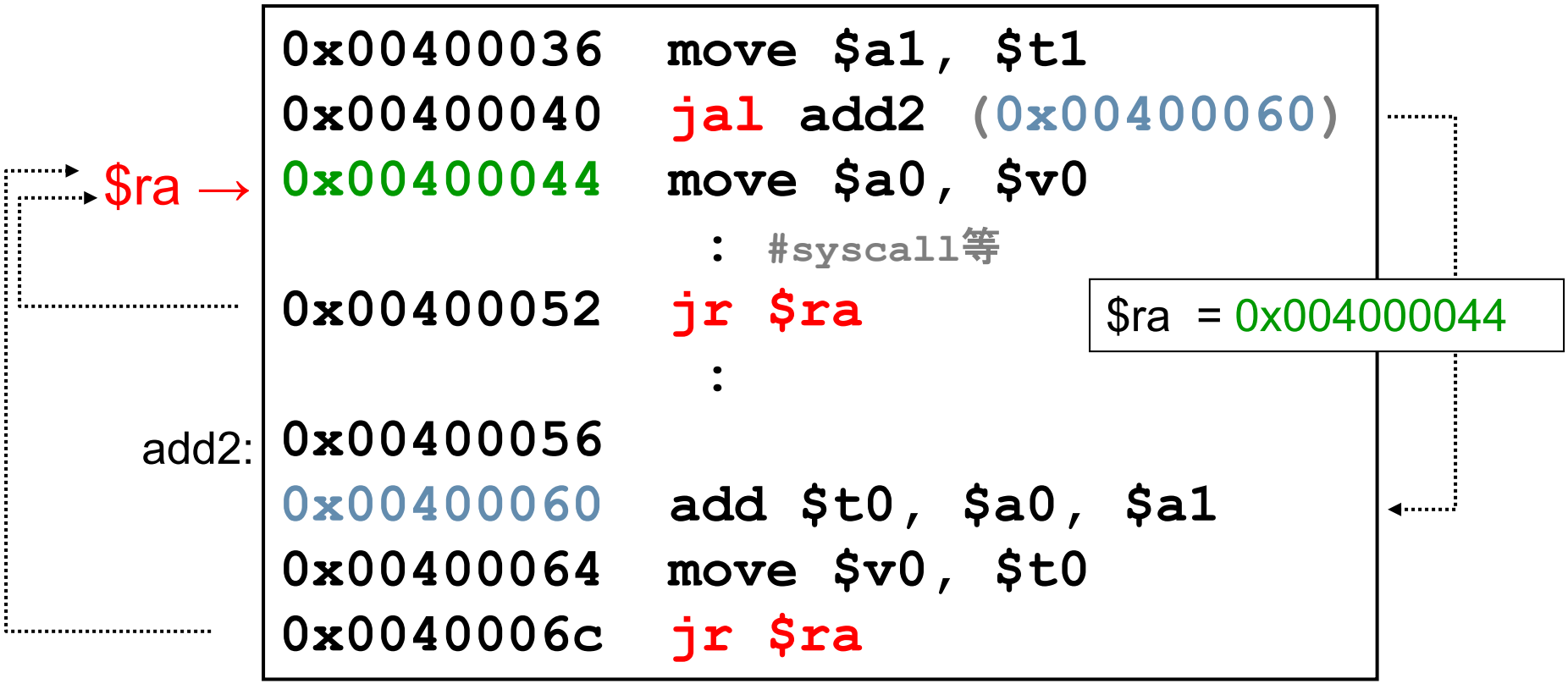
m6.s

```
  .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
```

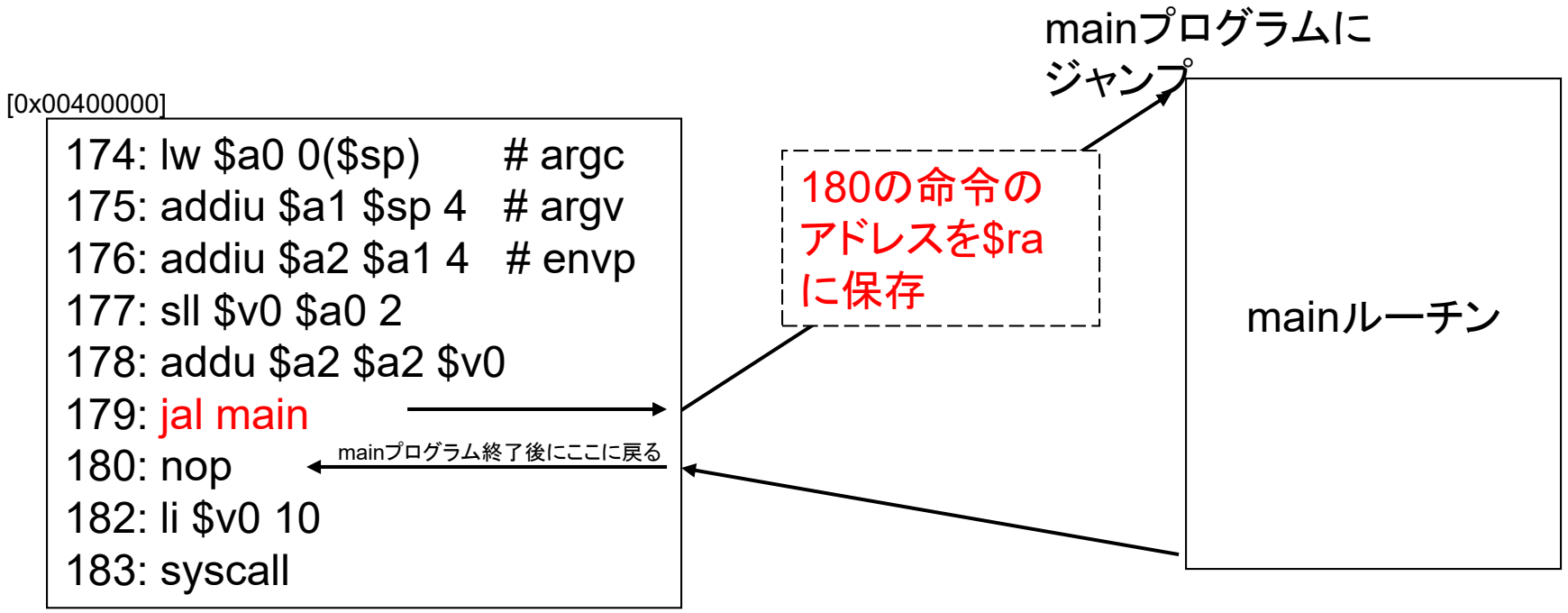
サブルーチンからの復帰

- mainルーチンの戻りアドレスが上書きされてしまう



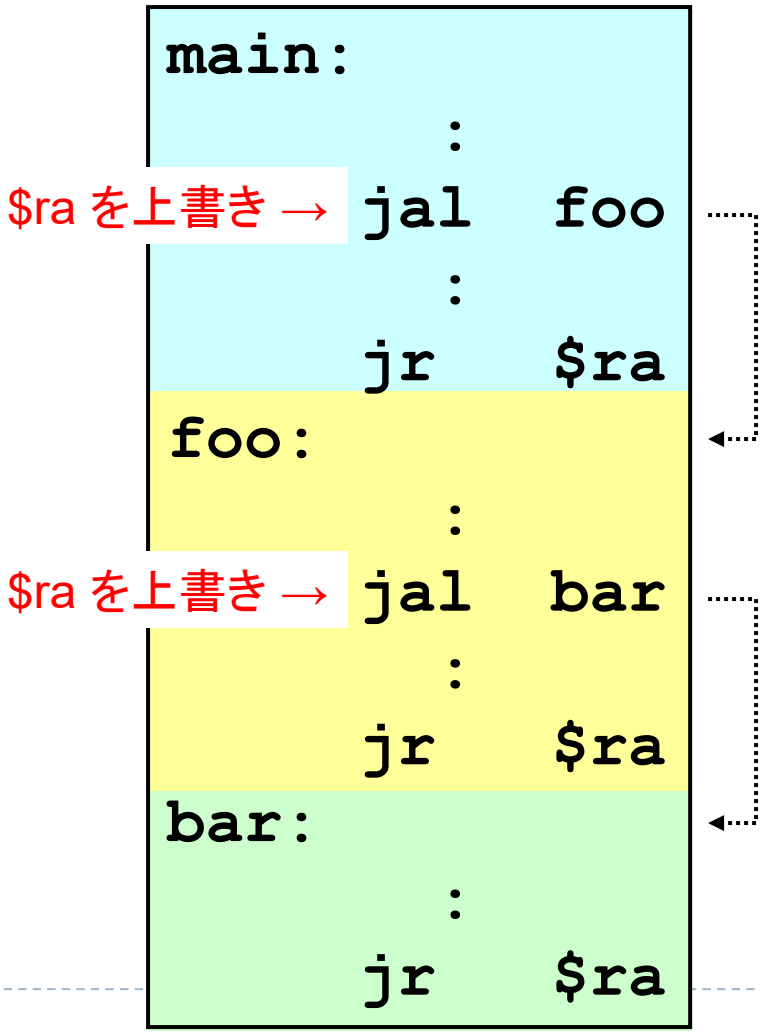
mainルーチンの流れ

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
```

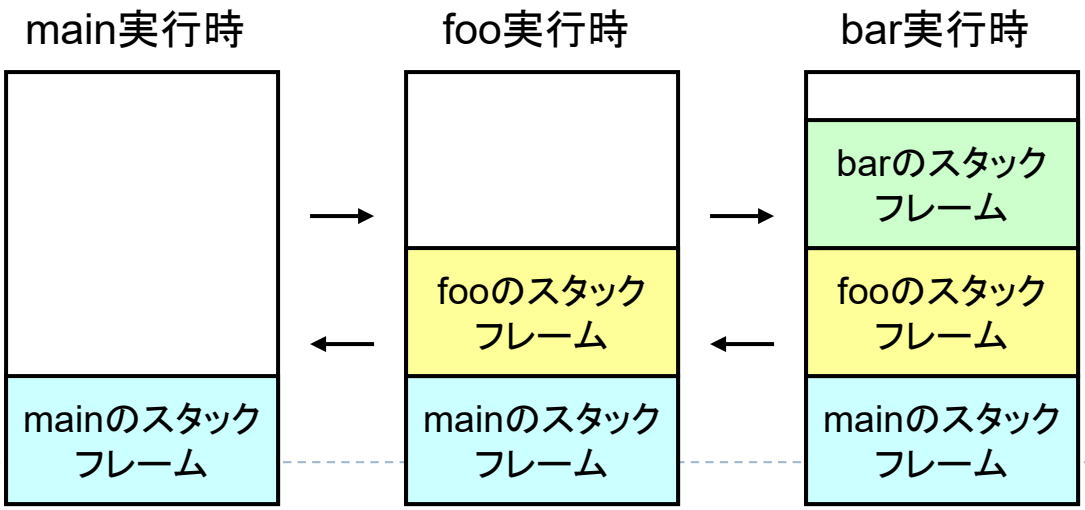
```
  move  $a0, $t0        # sum = add2(m, n)
  move  $a1, $t1
  sw    $t0, 4($sp)
  sw    $t1, 8($sp)
  jal   add2
  lw    $t1, 8($sp)
  lw    $t0, 4($sp)
  move  $a0, $v0        # printf
  li    $v0, 1
  syscall

  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

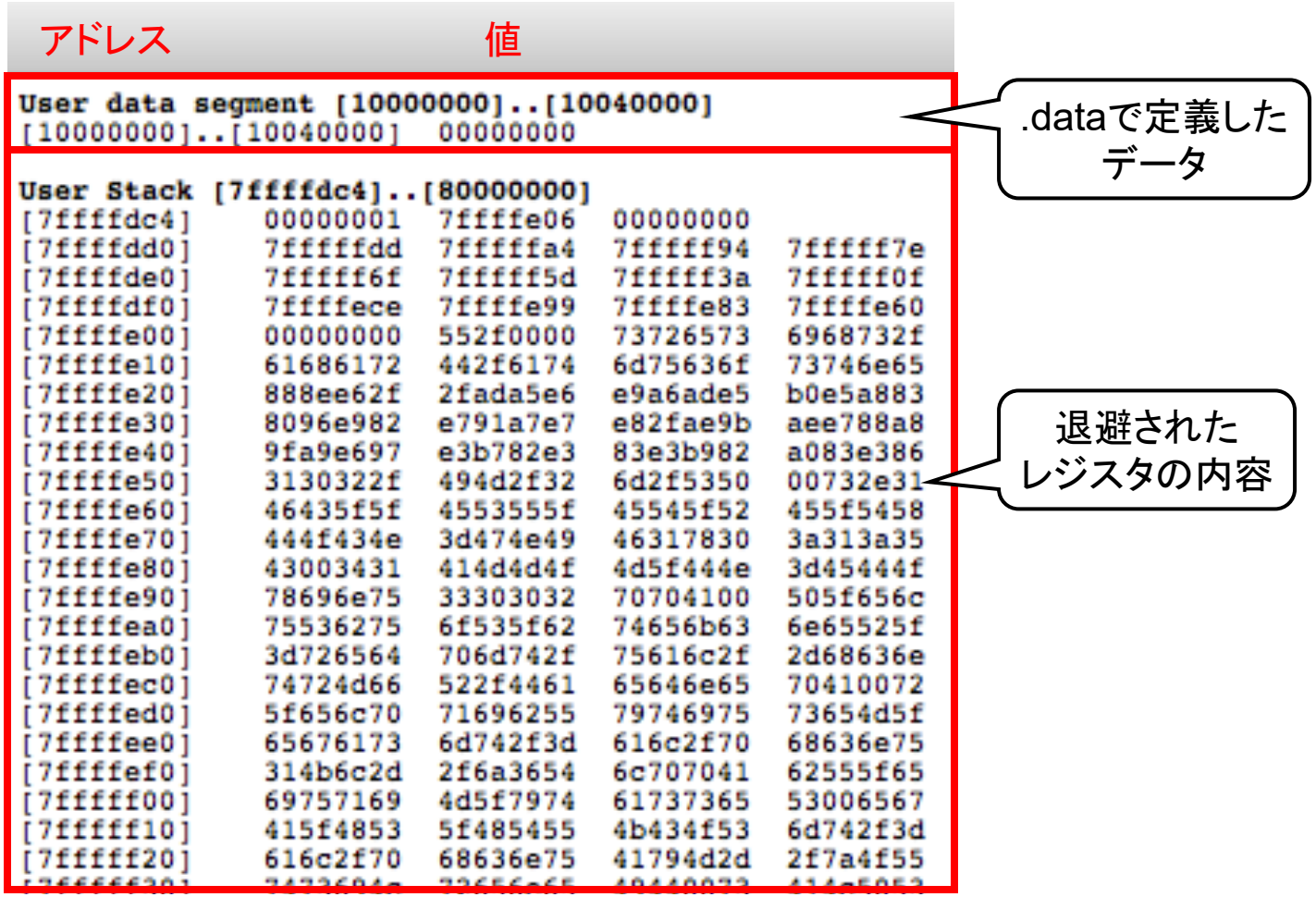
- 1つのサブルーチンが使うスタック領域
  - 退避されたレジスタの内容
    - $ra, $t0, $t1, ...
  - ローカル変数
    - そのサブルーチンの中でだけ使える変数
    - 高級言語が使用
  - 他のサブルーチンを呼ぶ時の5つ目以降の引数
    - 4つ目まではレジスタ $a0 ～ $a3 に入れられる

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

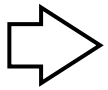
プログラム例	<pre># $xに代入
	jal foo
	jal bar
	# $xを使って計算</pre>
		<pre># $xに代入
		jal foo
		# $xを使って計算
		jal bar
		# $xを使って計算</pre>

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)	

動的配列

- sbrk システムコールで実行時(動的)に確保される配列
  - ⇔ 静的配列：データセグメントで実行前から(静的に)サイズが定義された配列
  - sbrk ($v0=9)：メモリ領域割り当てを要求
  - 引数（$a0）：確保するメモリ量（Byte）
  - 返り値（$v0）：確保されたメモリの先頭アドレス

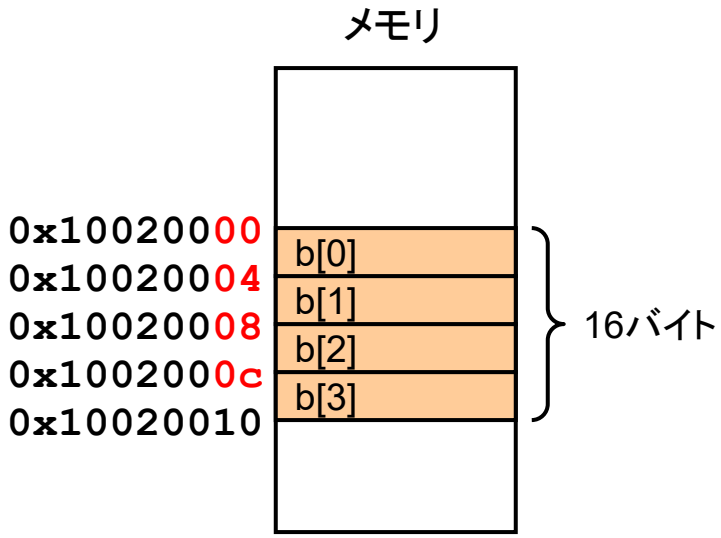
```
int* b = (int *)malloc(4*4)
```



```
li $v0, 9
li $a0, 16                    # 4*4byte
syscall                # sbrk
move $t0, $v0
```

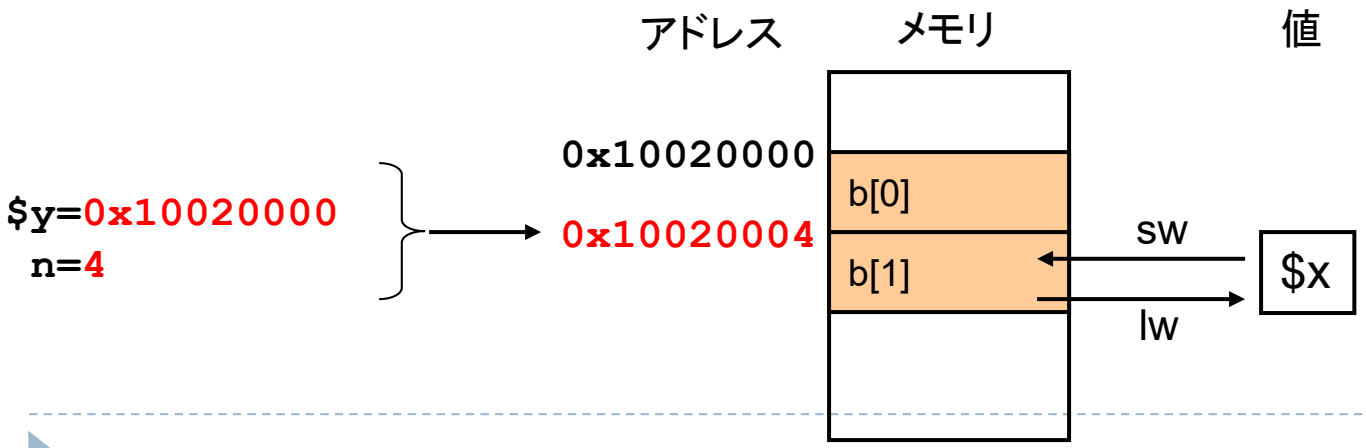
動的配列のアドレス

- sbrk した結果...
  - `$v0 = 0x10020000`
- bの先頭アドレス：**0x10020000**
  - b[0] のアドレス
    0x10020000
  - b[1] のアドレス
    0x10020004
  - :



動的配列の操作

- メモリアクセスは`sw,lw`
  - データセグメント、スタック領域（スタックセグメント）、ヒープ領域
- メモリアクセス命令
  - `sw $x, n($y)`
    - $x の値をメモリのアドレス n + $y に代入
  - `lw $x, n($y)`
    - メモリのアドレス n + $y にある値を $x に代入



動的配列の操作の例

- **b[1]**
  - 先頭のアドレス：$t0 = b
  - インデクス：I

```
# 4+b
lw $v0, 4($t0)
```

- **b[i]**
  - 先頭のアドレス：$t0 = b
  - インデックス：$a0 = i

```
add $t1, $a0, $a0
add $t1, $t1, $t1
add $t2, $t0, $t1
lw $v0, 0($t2)
```

$t1 = 4*$a0

$t2 = $t0 + 4*$a0

インデクス*i* にアクセスする場合、
*4*倍*(i*4)*すればよい

動的配列 (サンプル)

- データ数を入力させ、
  そのサイズnの配列bを
  b[i]=iと初期化するプログラム

```
while(i != 0) {
    i--;
    b[i] = i
}
```

m8.s

```
        .text
main:
        li $v0, 5
        syscall

        move $t1, $v0

        add $a0, $v0, $v0
        add $a0, $a0, $a0
        li $v0, 9
        syscall
        move $t0 ,$v0

while:
        beq $t1, $zero, end
        addi $t1, $t1, -1

        add $t2, $t1, $t1
        add $t2, $t2, $t2
        add $t2, $t0, $t2
        sw $t1, 0($t2)
        j while
end:
        jr $ra
```

$t1: 配列のサイズ

$v0*4 Byte

$t0: 配列の先頭アドレス

$t0 + $t1*4 Byte

補足：全32本のレジスタ

使用できるレジスタ

Name	Register number	Usage
$zero	0	the constant value 0
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

使用できないレジスタ

1 — $at
(assembler temporary)
reserved by the assembler

26-27 — $k0-$k1
reserved for use by the interrupt/trap handler

課題1

- 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2

- データ数、データを入力させ、全てのデータの和を計算し、出力するプログラムを書け
- 次の2つのサブルーチンを実装すること
  - create_array
    - 引数：配列の要素数($a0)
    - 戻り値：配列の先頭アドレス($v0)
    - 要素数分の動的配列を作成し、受け取ったデータで初期化する
      - 内部で read_int syscallを **データ数** 回呼び出してよい
  - calc_sum
    - 引数：配列の先頭アドレス($a0)、配列のサイズ($a1)
    - 戻り値：和($v0)
  - caller-save、あるいはcallee-saveで適切にレジスタを退避・復元すること

実行例

```
count=5
1
3
2
4
5
sum=15
```

課題2：ヒント

```
 # 必要に応じてレジスタを退避させる
 # 必要に応じてプロンプト(count=など)
 # を表示させること
  .text
main:

  # 要素数を読込む
  li    $v0, 5
  syscall
  move  $t0, $v0
  # create_arrayを呼び出す
  move  $a0, $t0
  jal   create_array
  # calc_sumを呼び出す
  move  $a0, $v0
  move  $a1, $t0
  jal   calc_sum
  # ナビゲーション（sum=など）を表示
  # 結果を表示

create_array
  # $a0: 作成する配列のサイズ
  # $v0: 配列の先頭アドレス
  # read_int syscallを $a0 回呼び出す
  # li    $v0, 5
  # syscall
  # move  $t0, $v0

calc_sum
  # $a0: 配列の先頭アドレス
  # $a1: 配列のサイズ
  # $v0: 和の結果
```

課題提出

- 〆切：2018/01/09 (火) 23:59
  - OCW-iから提出すること
  - 遅れても（減点しますが）受け付けます。
- 提出物：以下のファイルを1つのファイルにzip圧縮したもの
  - ドキュメント　(pdf, txt 形式)
    - 課題1の議論
    - 各課題の実行結果
    - プログラムソースの簡単な説明、工夫したところ
    - 感想、質問等
  - プログラムソース
  - 全てのファイル名は半角英数字でお願いします
    - レポートのファイルを含む、文字化け防止のため

課題締め切り

- 第01回
  - 遅れても減点しますが受け付けます
- 第02回
  - 12/21 (金) 本日 23:59 (日本時間)
- 第03回
  - 1/8 (火)
- 第04回
  - 1/8 (火)