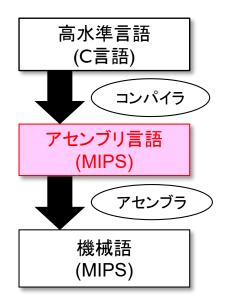
#### 2018年度 計算機システム(演習) 第4回 2018.12.21

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

```
void main() {
                    int m = 10;
                    int n = 20;
 add2のアドレスへジャンプ
                    int sum = add2(m, m);
                   printf("%d\n", sum);
                  int add2(int x, int y) {
                    int sum = x + y;
                    return sum;
以前実行していたアドレスの
次命令のアドレスへジャンプ
```



## 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<u>実行が移る</u>
  - 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<mark>戻り値</mark>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 ≒ <u>呼び出し規約</u>に従って書かれた プログラムの一部分
  - 1. 引数を<u>\$a</u>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
- $\rightarrow$  add2: v0 = a0+a1

```
.text
main:
                $t0, 10
                          \# m=10
        li
        1 i
                $t1, 20
                          \# n = 20
                $a0, $t0
        move
                $a1, $t1
        move
                add2
                          # call
        jal
                $a0, $v0
        move
       # syscallで出力
        move $a0, $t0
        move $a1, $t1
                add2
                          # call
        jal
                $a0, $v0
        move
       # syscallで出力
              $ra
       jr
add2:
                \# a0->x, a1->y
        add
                $t0, $a0, $a1
                $v0, $t0
        move
        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



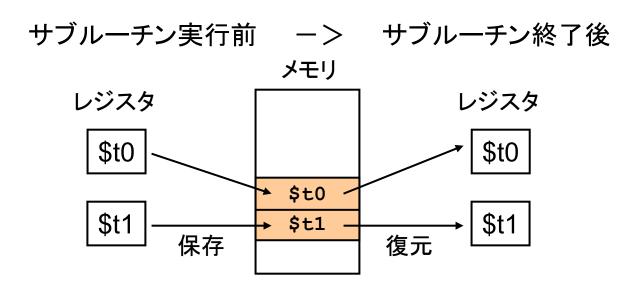
## レジスタの使用規則

- ▶ 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)
    - \$\$0~\$\$7, \$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)

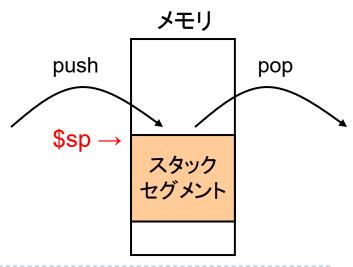
▶ 呼び出し元がサブルーチンを呼ぶ前にメモリにレジスタの内容を保存し、終了したら元に戻す





## レジスタ退避場所:スタックセグメント

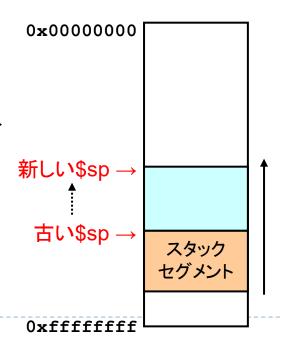
- メモリのどこにレジスタを保存するか?
  - ・ 保存用のスタックが用意されている => スタックセグメント
    - ▶ 保存する時にスタックに積む(push)
    - ▶ 復元する時にスタックから取り出す(pop)
  - \$sp レジスタがスタックの先頭アドレスを指している
    - スタックの先頭に push, pop すればよい





## レジスタの退避

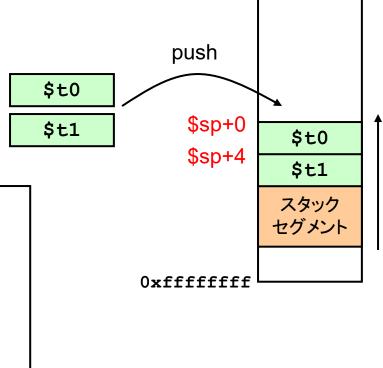
- ▶ caller-saveではサブルーチン実行前に行う
- ▶ レジスタ退避方法
  - I. スタックに push する場所を確保
    - addi \$sp, \$sp, -n
    - m 個のレジスタを push する時 n = m\*4
      - □ レジスタ: 4バイト
    - ▶ -n するのはスタックはアドレス の高い方(上位)から低い方(下位)へ 伸びるため



# 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



0x00000000

```
main:

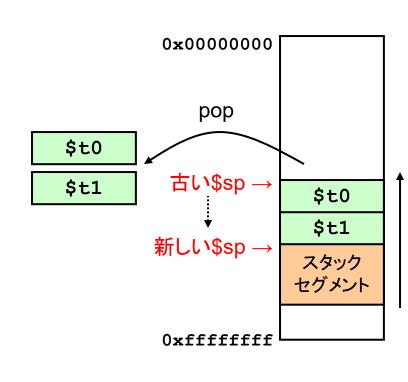
addi $sp, $sp, -8 #2レジスタ*4byte

:
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
  - スタックから値を読み出す
    - lw \$x, d(\$sp)
      - $\Box$  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
                                       対応
    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
                                                 $a0, $t0
                                                               \# sum = add2(m, n)
                                           move
main:
                                                 $a1, $t1
                                           move
                                                 $t0, 0($sp)
  addi
       $sp. $sp. -8 # 退避領域作成
                                           SW
                                           sw $t1, 4($sp)
  Ιi
       $t0. 10
                                           jal add2
       $t1, 20
                                           lw $t1, 4($sp)
                                           lw $t0, 0(\$sp)
                \# sum = add2(m. m)
                                           move $a0, $v0
  move
       $a0, $t0
                                                               # printf
       $a1, $t0
                                           Ιi
                                                 $v0. 1
  move
       $t0, 0($sp)
                                           syscall
  SW
       $t1, 4($sp)
                                           addi
     add2
                                                 $sp, $sp, 8 # 領域開放
  ial
     $t1, 4($sp)
                                           ir
                                                 $ra
       $t0, 0($sp)
                                         add2:
  move $a0, $v0
                                                $t0, $a0, $a1
                      # printf
                                           add
       $v0, 1
                                                 $v0,
  Ιi
                                                     $t0
                                           move
                                                 $ra
  syscall
                                           jr
```

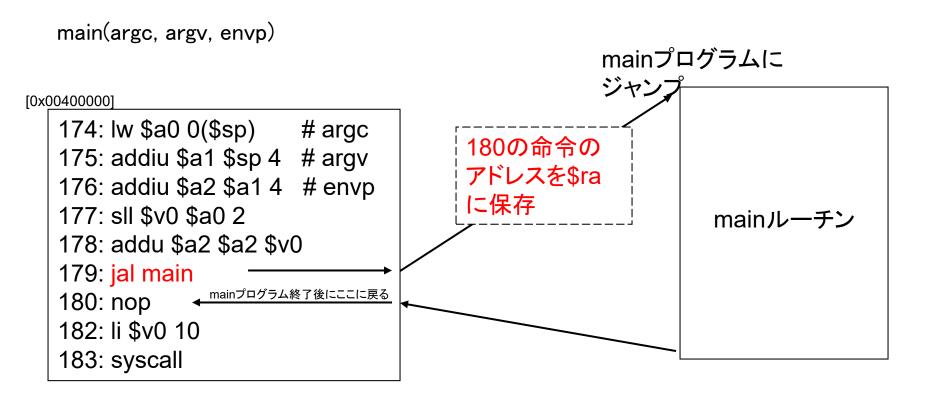


## サブルーチンからの復帰

▶ mainルーチンの戻りアドレスが上書きされてしまう

```
0 \times 00400036
                      move $a1, $t1
     0 \times 00400040
                      jal add2 (0x00400060)
     0 \times 00400044
                      move $a0, $v0
                        : #syscall等
     0 \times 00400052
                      jr $ra
                                        rac{1}{3} = 0x004000044
     0x00400056
add2:
      0 \times 00400060
                      add $t0, $a0, $a1
                      move $v0, $t0
     0 \times 00400064
     0 \times 0040006c
                      jr $ra
```

## mainルーチンの流れ



## サブルーチンを呼ぶ場合 \$ra の退避

- > \$raltcallee-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

```
main:
$ra を上書き → jal
                 foo
                 $ra
     foo:
$ra を上書き → jal
                 bar
                 $ra
    bar:
                 $ra
```

## 完全なレジスタ保存コード例

```
main:
    addi $sp, $sp, -12
     sw $ra, 0($sp)
         $t0, 4($sp)
     SW
     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
                                                 $a0. $t0
                                                               \# sum = add2(m. n)
                                           move
                                                 $a1, $t1
main:
                                           move
       $sp, $sp, -12 # 退避領域作成
                                                 $t0, 4($sp)
  addi
                                           SW
       $ra, 0($sp) # ra退避
                                                $t1, 8($sp)
                                           SW
  SW
  Ιi
       $t0, 10
                                              add2
                                           ial
       $t1, 20
                                                $t1, 8($sp)
                                           l w
                                                $t0, 4($sp)
                                           l w
                \# sum = add2(m, m)
                                                               # printf
       $a0. $t0
                                                $a0, $v0
 move
                                           move
       $a1, $t0
                                           Ιi
                                                $v0, 1
 move
       $t0, 4($sp)
                                           syscall
  SW
       $t1, 8($sp)
  SW
                                                $ra, 0($sp) # ra復帰
  ial
      add2
                                           l w
       $t1, 8($sp)
                                           addi
                                                 $sp, $sp, 12 # 領域開放
       $t0, 4($sp)
                                           ir
                                                 $ra
                                         add2:
       $a0, $v0
                      # printf
  move
                                                $t0, $a0, $a1
  Ιi
       $v0, 1
                                           add
                                                 $v0, $t0
  syscall
                                           move
                                                 $ra
                                           ir
```



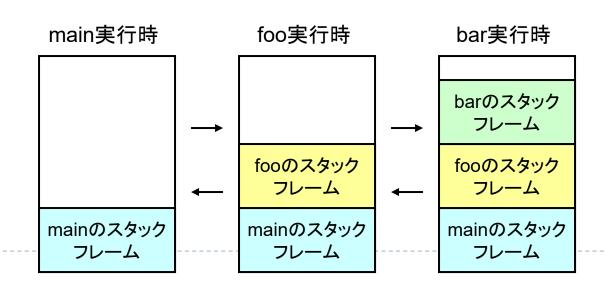
## 復習:スタックフレーム

- ▶ 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

```
アドレス
                           値
User data segment [10000000]..[10040000]
                                                          .dataで定義した
[10000000]..[10040000]
                        00000000
                                                               データ
User Stack [7ffffdc4]..[80000000]
                        7ffffe06
[7ffffdc4]
              00000001
                                  00000000
[7ffffdd0]
              7fffffdd
                        7ffffffa4
                                  7ffffff94
                                             7ffffff7e
                        7ffffff5d
                                  7ffffff3a
                                             7ffffff0f
[7ffffde0]
              7ffffff6f
[7ffffdf0]
              7ffffece
                       7ffffe99
                                  7ffffe83
                                             7ffffe60
[7ffffe001
              00000000 552f0000
                                 73726573
                                             6968732f
7ffffe101
              61686172 442f6174
                                  6d75636f
                                            73746e65
                                 e9a6ade5
              888ee62f 2fada5e6
[7ffffe201
                                            b0e5a883
                                                            退避された
[7ffffe30]
              8096e982 e791a7e7
                                  e82fae9b
                                             aee788a8
[7ffffe40]
              9fa9e697 e3b782e3
                                  83e3b982
                                            a083e386
                                                          レジスタの内容
              3130322f 494d2f32 6d2f5350
[7ffffe50]
                                             00732e31
                                             455f5458
[7ffffe60]
              46435f5f
                        4553555f
                                  45545f52
[7ffffe70]
              444f434e 3d474e49
                                  46317830
                                             3a313a35
[7ffffe80]
              43003431
                        414d4d4f
                                 4d5f444e
                                             3d45444f
[7ffffe90]
                                             505f656c
              78696e75 33303032
                                  70704100
[7ffffea0]
              75536275 6f535f62
                                  74656b63
                                             6e65525f
[7ffffeb0]
              3d726564
                       706d742f
                                  75616c2f
                                             2d68636e
7fffffec01
              74724d66
                        522f4461
                                             70410072
                                  65646e65
[7ffffed0]
              5f656c70
                       71696255
                                  79746975
                                             73654d5f
[7ffffee0]
              65676173
                        6d742f3d
                                  616c2f70
                                            68636e75
                                  6c707041
[7ffffef0]
              314b6c2d
                       2f6a3654
                                             62555f65
[7fffff00]
              69757169 4d5f7974
                                  61737365
                                             53006567
7ffffff101
              415f4853
                        5f485455
                                  4b434f53
                                             6d742f3d
              616c2f70
                                  41794d2d
7fffff201
                        68636e75
                                             2f7a4f55
```



#### 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₺ callee-save



## add2をcallee-saveで実装

```
$a0, $v0
. text
                                                          # printf
                                       move
main:
                                        Ιi
                                             $v0. 1
       $sp, $sp, -12 # 退避領域作成
 addi
                                        syscall
       ra, O(\$sp) # raltcallee save
 SW
       l w
                                             $s1, 8($sp)
                                                         # s1復帰
 SW
       $s1, 8($sp) # s1/dcallee save
                                             $s0, 4($sp) # s0復帰
                                        l w
 SW
                                             $ra, 0($sp) # ra復帰
                                        lw
                                             $sp, $sp, 12 # 領域開放
 Ιi
     $s0, 10
                                        addi
       $s1. 20
                                             $ra
                                        ir
                   \# sum = add2(m.
                                      add2:
 move
       $a0, $s0
       $a1, $s1
                                        addi
                                             sp, sp, -4
 move
 ial
       add2
                                             $s0. 0($sp)
                                        SW
       $a0, $v0
                   # printf
 move
 Ιi
       $v0. 1
                                             $s0, $a0, $a1
                                        add
                                             $v0. $s0
 syscall
                                       move
       $a0, $s0
                   \# sum = add2(m, n)
                                             $s0, 0($sp)
 move
                                        l w
       $a1, $s1
                                             $sp, $sp, 4
                                        addi
 move
       add2
                                             $ra
  ial
                                        ir
```

### 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~\$vI

```
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
          $ra
     jr
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
           $s0, 0($sp)
     SW
     # $50を使った処理
     lw
          $s0, 0($sp)
     addi $sp, $sp, 4
           $ra
     jr
```

# 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	# \$xに代入 jal foo jal bar # \$xを使って計算	# \$xに代入 jal foo # \$xを使って計算 jal bar # \$xを使って計算		
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)			



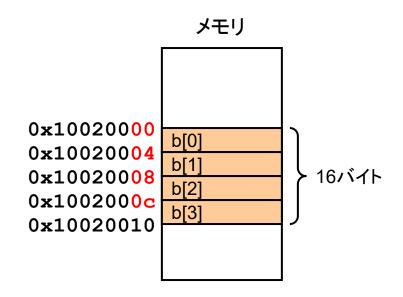
## 動的配列

- ▶ 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



## 動的配列のアドレス

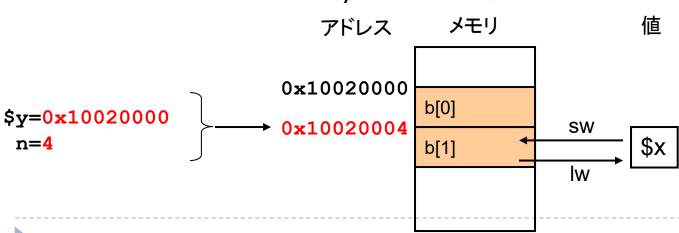
- ▶ sbrk した結果...
  - $\Rightarrow$  \$v0 = 0x10020000
- ▶ bの先頭アドレス: 0x10020000
  - b[0] のアドレス 0x10020000
  - b[I] のアドレス 0x10020004
  - **:**





## 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 
\displaystar $t2 = $t0 + 4*$a0
\lw $v0, 0($t2)
```

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
}
```

.text main: li \$v0, 5 syscall \$t1: 配列のサ move **\$t1**, \$v0 イズ add \$a0, \$v0, \$v0 add \$a0, \$a0, \$a0 \$v0\*4 Byte li \$v0, 9 syscall \$t0: 配列の先 move \$t0 ,\$v0 頭アドレス while: beq \$t1, \$zero, end addi \$t1, \$t1, -1

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:

ir \$ra

\$t0 + \$t1\*4 Byte

## 補足:全32本のレジスタ

#### 使用できるレジスタ

#### 使用できないレジスタ

Name	Register number	Usage	_	
\$zero	0	the constant value 0	/ <u> </u>	
\$v0-\$v1	2-3	values for results and expression evaluation	\	\$at
\$a0-\$a3	4-7	arguments	•	(assembler temporary)
\$t0-\$t7	8-15	temporaries		reserved by the assembler
\$s0-\$s7	16-23	saved	4	,
\$t8-\$t9	24-25	more temporaries	26.27	
\$gp	28	global pointer	20-21	\$k0-\$k1
\$sp	29	stack pointer	1	reserved for use by the
\$fp	30	frame pointer		interrupt/trap handler
\$ra	31	return address		1 1



### 課題1

- caller-save で書かれた次ページのコードを <u>callee-save で書き直</u>し、コードの効率の違いを<u>論じよ</u>
  - ▶ 0または I を 8回入力させて 8ビットの2進数とみなし、それを I0 進数に変換するプログラム
    - (入力I)×2<sup>7</sup>+(入力2)×2<sup>6</sup>+(入力3)×2<sup>5</sup>+(入力4)×2<sup>4</sup>+(入力5)×2<sup>3</sup>+(入力6)×2<sup>2</sup>+(入力7)×2<sup>1</sup>+(入力8)×2<sup>0</sup>
    - ▶ 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
                                  # 結果を表示
#必要に応じてプロンプト(count=など)
#を表示させること
                                 create array
                                  # $a0: 作成する配列のサイズ
 text
                                  # $v0: 配列の先頭アドレス
main:
                                  # read_int syscallを $a0 回呼び出す
                                  # Ii $v0, 5
 # 要素数を読込む
 li $v0, 5
                                  # syscall
 syscall
                                  # move $t0. $v0
 move $t0. $v0
 # create_arrayを呼び出す
                                 calc sum
 move $a0, $t0
                                  # $a0: 配列の先頭アドレス
 jal create_array
                                  # $a1: 配列のサイズ
 # calc_sumを呼び出す
                                  # $v0: 和の結果
 move $a0, $v0
 move $a1, $t0
 ial calc sum
 # ナビゲーション(sum=など)を表示
```

### 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I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Ⅰの議論
    - ト各課題の実行結果
    - 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0I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)

