制御の流れ
（手続き呼出）

（教科書3.6節）

手続き呼出の基本(1)

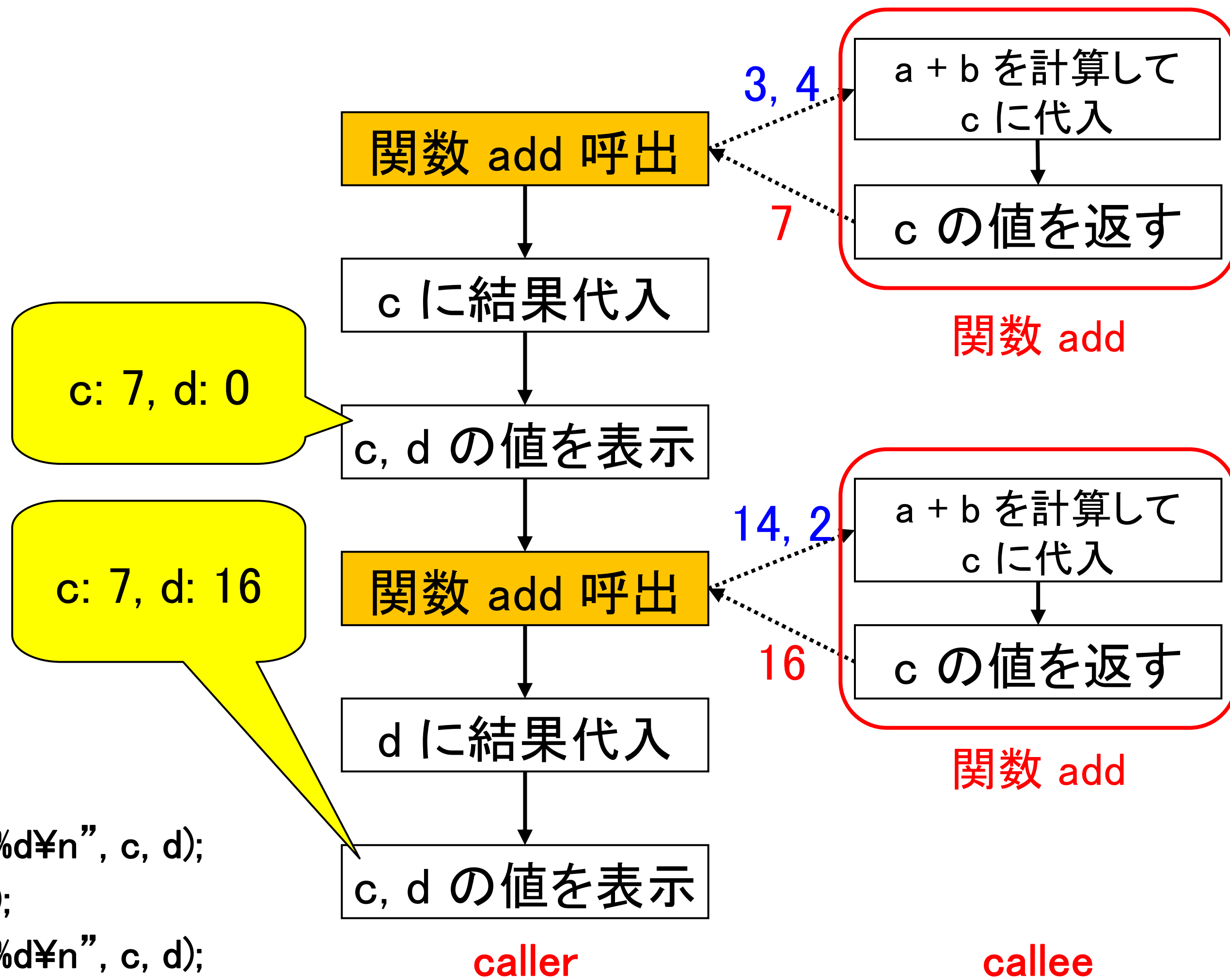
```
#include <stdio.h>

int add (int a, int b)
{
    int c;

    c = a + b;
    return c;
}

int main ()
{
    int c = 0, d = 0;

    c = add (3, 4);
    printf ("c: %d, d: %d¥n", c, d);
    d = add (c * 2, 2);
    printf ("c: %d, d: %d¥n", c, d);
    …
}
```



手続き呼出の基本(2)

```
#include <stdio.h>

int add (int a, int b)
{
      int c;

      c = a + b;
      return c;
}

int main ()
{
      int c = 0, d = 0;

      c = add (3, 4);
      printf ("c: %d, d: %d¥n", c, d);
      d = add (c * 2, 2);
      printf ("c: %d, d: %d¥n", c, d);
      …
}
```

仮引数

実引数

callee の作業:

1. 実引数の仮引数への代入.
2. 内部の実行.
3. 復帰: caller へのジャンプ.

caller の作業:

1. 実引数式の値の計算.
2. 呼出: callee へのジャンプ.
3. 結果の受け取り.

caller は複数ありうるので単純なジャンプでは caller へ復帰できない.

手続き呼出の基本(3)

- 実引数の受け渡し: レジスタ $a0～$a3 (レジスタ番号 4～7)を使用.
- 結果の受け取り: レジスタ $v0, $v1(レジスタ番号 2, 3)を使用.
- callee へのジャンプ: jal 命令を使用.
- caller へのジャンプ: jr 命令を使用.

jal 命令: レジスタ $ra(レジスタ番号 31)に jal 命令直後の命令の番地(= PC + 4)を格納し, ラベルで指定される callee へジャンプ. 復帰時は jr 命令で $ra が格納する番地にジャンプ.

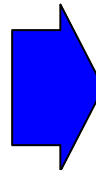
```
jal    callee    # callee 呼出
jr     $ra       # caller へ復帰
```

手続き呼出の基本(4)

caller 側のコード

```
int main ()
{
    …
    c = add (3, 4);
    …
    d = add (c * 2, 2);
…
}
```

変数名	レジスタ
c	$s0
d	$s1



```
…
addi    $a0, $zero, 3
addi    $a1, $zero, 4
jal     _add
add     $s0, $v0, $zero
…
add     $a0, $s0, $s0
addi    $a1, $zero, 2
jal     _add
add     $s1, $v0, $zero
…
```

第一実引数 ($a0)
第二実引数 ($a1)
返り値 ($v0)

【注意】addi 命令は，レジスタと定数値を加算して，その結果をレジスタに格納する命令である．（後述）

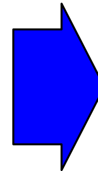
手続き呼出の基本（5）

callee 側のコード

```
int add (int a , int b)
{
    int c;

    c = a + b;
    return c;
}
```

変数名	レジスタ
c	$s0



```
_add:
add     $s0, $a0, $a1
add     $v0, $s0, $zero
jr      $ra
```

第一仮引数

第二仮引数

返り値

復帰先番地

caller が，呼出前に復帰先番地を $ra に設定していることに注意すること．

手続き呼出とスタック(1)

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

int dbl (int a)
{
    int c;
    c = add (a, a);
    return c;
}

int main ()
{
    int c;
    …
    c = dbl (100);
    foo (a, 3, b, 4, c);
    …
}
```

```
_add:
add     $v0, $a0, $a1
jr      $ra

_dbl:
        #add 第一実引数
add     $a0, $a0, $zero
        #add 第二実引数
add     $a1, $a0, $zero
jal     _add
add     $s0, $v0, $zero
jr      $ra

_main:
…
addi    $a0, $zero, 100
jal     _dbl
add     $s0, $v0, $zero
…
```

☹ $ra が上書きされる.（＝main に戻れない！）

☹ caller で使っていたレジスタが上書きされる.

☹ 引数が4つ以上ある場合は？

手続き呼出とスタック（2）

MIPS のレジスタ使用に関するルール（＝紳士協定！）

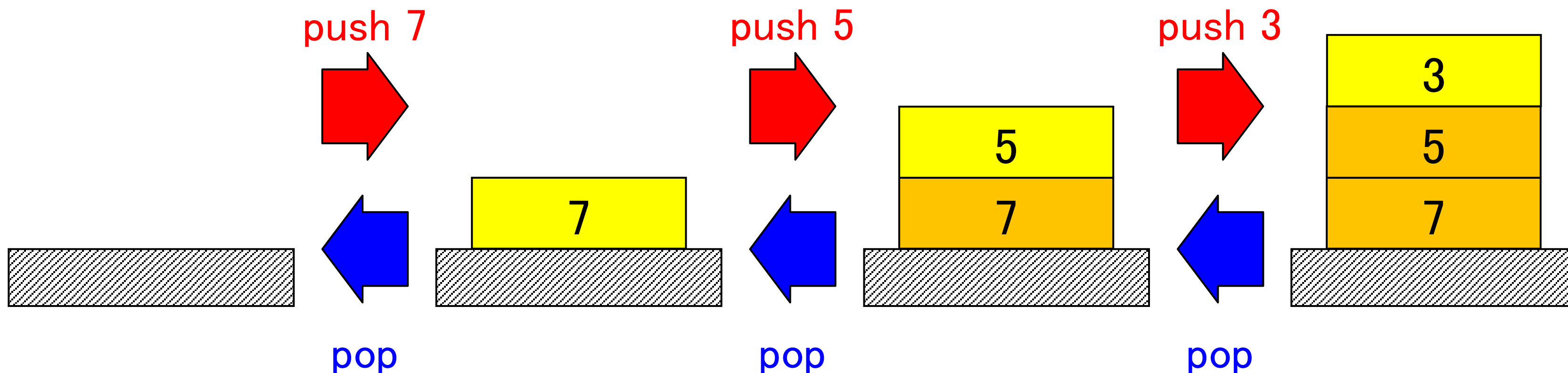
レジスタ名	レジスタ番号	用途	関数呼出時のレジスタ破壊
$zero	0	定数値 0，不要な結果の破棄	(NA)
$v0～$v1	2～3	関数の返り値	可
$a0～$a3	4～7	引数用	不可
$t0～$t7	8～15	一時保存用	可
$s0～$s7	16～23	変数用	不可
$t8～$t9	24～25	一時保存用	可
$sp	29	スタックポインタ（後述）	不可
$fp	30	フレームポインタ（後述）	不可
$ra	31	戻り番地	不可

関数呼出時のレジスタ破壊が「不可」のレジスタは，関数呼出直前と復帰直後の値が同じになることを，プログラマが保証しなければならない．

手続き呼出とスタック(3)

上書き問題の解決策　＝　スタックの利用

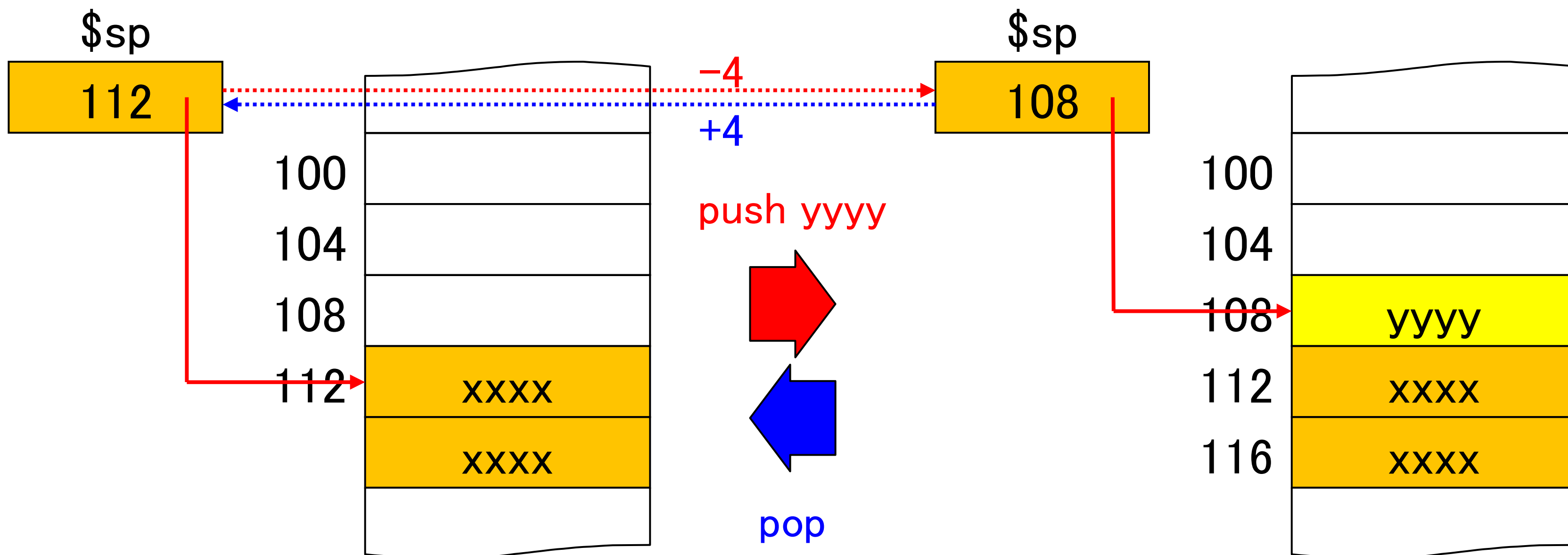
スタック: 他のデータを後入れ先出し型(**LIFO**: Last-In First-Out)で格納するデータ構造. データの追加(**push**), データの取り出し(**pop**), 最後に格納したデータの参照(**top**)のみが許される.



手続き呼出とスタック(4)

MIPS におけるスタック:

- メモリ領域の一部をスタックのために使用する.
- MIPS では，29番のレジスタに **$sp** と名前をつけ，スタックの一番上の番地を記憶するために使用する.（**スタックポインタ**）



手続き呼出とスタック(5)

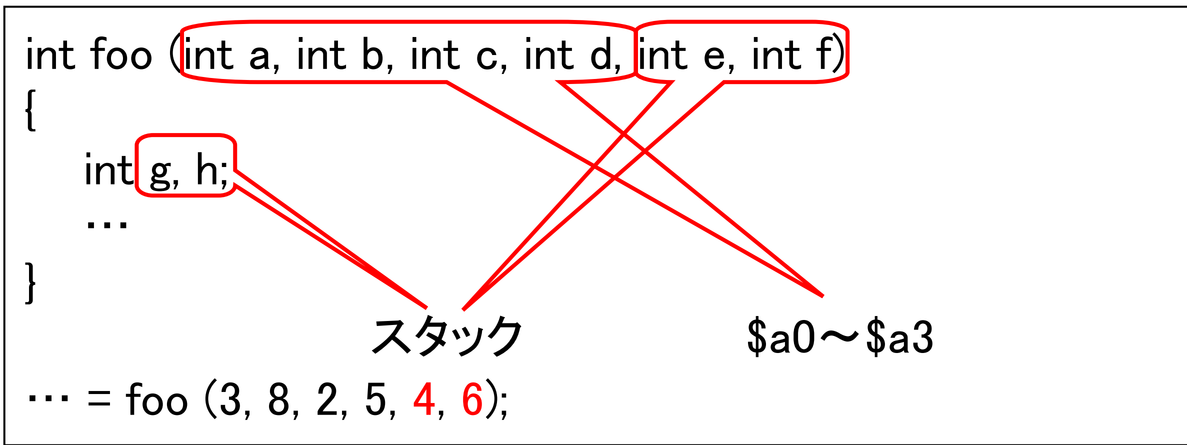
手続き foo の呼出手順(不完全版):

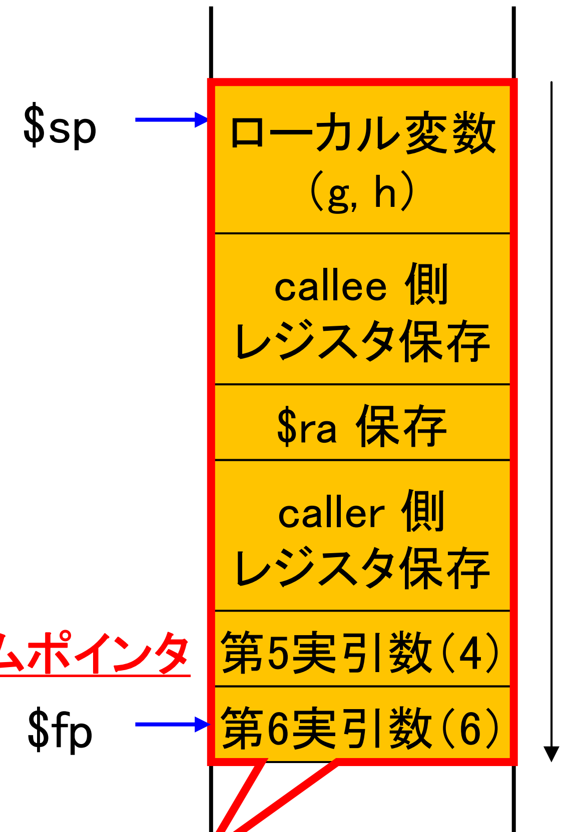
1. 関数呼出によって破壊されうるレジスタ($t0～$t9, $v0, $v1)の値を push する.(後に参照するレジスタのみでよい.) — caller
2. foo を呼び出す.(jal foo 命令) — caller
3. foo の内部で他の関数を呼び出す場合は $ra を push する. — callee
4. 関数呼出によって破壊されてはならないレジスタ($a0～$a3, $s0～$s7)の値を push する.(foo 内部で更新するレジスタのみでよい.) — callee

手続き foo からの復帰手順(不完全版):

1. 呼出後 4 で push した値を元のレジスタに pop する. — callee
2. 呼出後 3 で $ra を push した場合は, その値を $ra に pop する. — callee
3. foo を呼び出した jal 命令の次の命令にジャンプする.(jr $ra 命令) — callee
4. 呼出前 1 で push した値を元のレジスタに pop する. — caller

手続き呼出とスタック(6)





手続き foo の呼出手順(完全版):

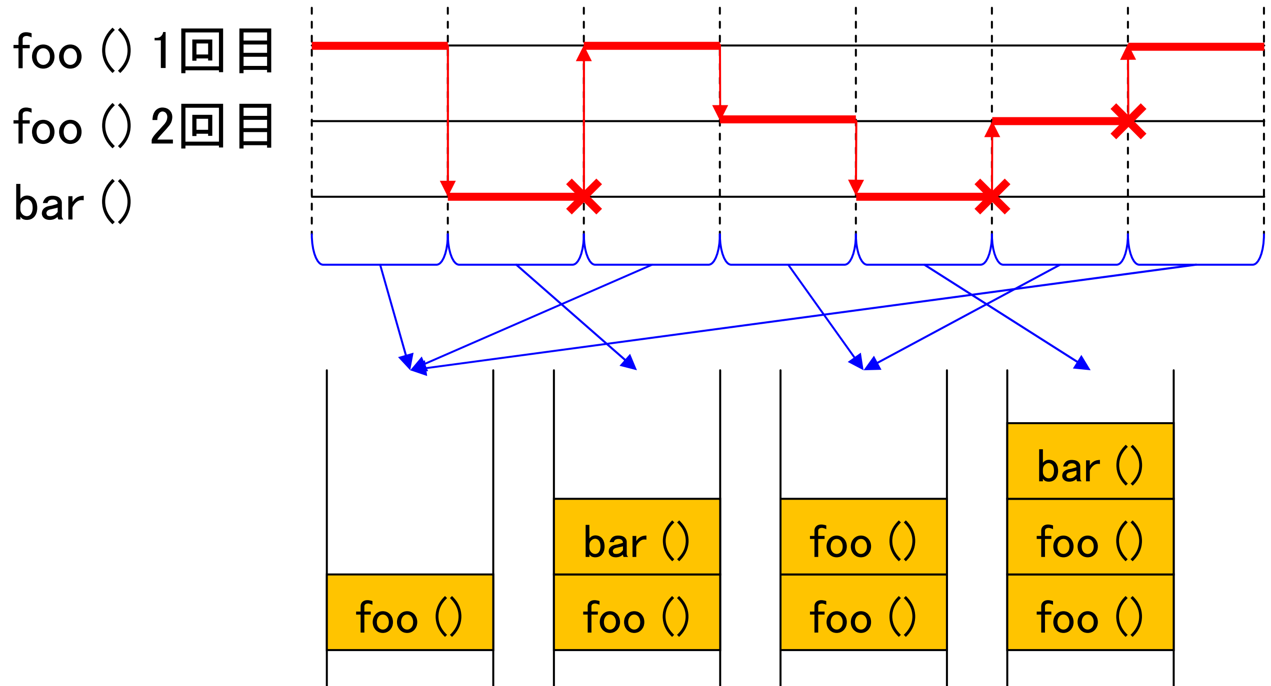
1. 5番目以降の実引数を push
2. caller 側レジスタ保存($t0～$t7 *etc.*)
3. foo 呼出
4. $ra を push
5. callee 側レジスタ保存($s0～$s9 *etc.*)
6. ローカル変数のための領域をスタック上に確保
7. $fp をスタックフレームの底に設定.

手続き呼出とスタック（7）

手続きフレームは，関数の呼出ごとに生成される.

```
int bar ()
{
    …
}

int foo ()
{
    int x, y;
    …
    bar ();
    …
    foo ();
    …
}
```



- 呼出ごとに独立したローカル変数の実現.
- 再帰呼出が可能.