東北大学 工学部 機械知能・航空工学科
2020年度 クラス C D

情報科学基礎 I

7. MIPSの命令と動作 — 分岐・ジャンプ・関数呼出し

(教科書7章
命令一覧は p.113)

大学院情報科学研究科
鏡 慎吾

MIPSの分岐命令・ジャンプ命令

分岐・ジャンプ命令

条件文や繰り返し文などを実現するには，命令の実行順の制御が必要

(C言語)

```
if (x == y) {
    x = x + 1;
}
...
```

ただし，変数 x, y の内容がそれぞれレジスタ s0, s1 に置かれているとする

(MIPSアセンブリ言語)

```
    bne   $s0, $s1, L1     # $s0 ≠ $s1 ならば L1 へ分岐
    addu  $s0, $s0, 1
L1: ...
```

ラベル

資料: 主な分岐命令，比較命令

命令	説明
beq $a, $b, L	$a = $b ならばラベル L へ分岐 (branch on equal)
bne $a, $b, L	$a ≠$b 〃 (branch on not equal)

slt $c, $a, $b	符号つきで $a < $b ならば $c ← 1; さもなくば $c ← 0
sltu $c, $a, $b	符号無しで $a < $b ならば $c ← 1; さもなくば $c ← 0
sgt $c, $a, $b	符号つきで $a > $b 〃 (set on greater than)
sgtu $c, $a, $b	符号無しで $a > $b 〃 (set on greater than unsigned)
sle $c, $a, $b	符号つきで $a≦$b 〃 (set on less than or equal)
sleu $c, $a, $b	符号無しで $a≦$b 〃 (set on less than or equal unsigned)
sge $c, $a, $b	符号つきで $a≧$b 〃 (set on greater than or equal)
sgeu $c, $a, $b	符号無しで $a≧$b 〃 (set on greater than or equal unsigned)

- slt, sltu 以外の比較命令はマクロ命令

例: if – else 文

(C言語)

```
if (x < y) {
    x = x + 1;
} else {
    x = x + 2;
}
...
```

ただし，変数 x, y の内容がそれぞれレジスタ s0, s1 に置かれているとする

(MIPSアセンブリ言語)

```
    slt   $t0, $s0, $s1
    beq   $t0, $zero, L1  # $s0 < $s1 でないなら L1へ分岐
    addu  $s0, $s0, 1
    j     L2              # L2 へジャンプ
L1: addu  $s0, $s0, 2
L2: ...
```

例: while文

(C言語)

```
while (x < y) {
    x = x + 1;
}
```

ただし，変数 x, y の内容がそれぞれレジスタ s0, s1 に置かれているとする

(MIPSアセンブリ言語)

```
L1: slt  $t0, $s0, $s1      # x < y なら $t0 ← 1; さもなくば $t0 ← 0
    beq  $t0, $zero, L2     # 比較結果が偽(ゼロ)なら L2 へ
    addu $s0, $s0, 1        # x ← x + 1
    j    L1
L2: ...
```

MIPSの関数呼出し機構

関数呼出し

(C言語)

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

int main() {
    int x, y, z;

    x = func(5, 1);
    y = func(8, 2);
    z = func(x, y);
    ...
}
```

関数呼出しは単に j 命令でジャンプするだけでは実現できない
（元の位置に戻って来る必要がある）

ただし，関数main内の変数 x, y, z の内容がそれぞれレジスタ s0, s1, s2 に置かれているとする

関数呼出しの実行例

(MIPSアセンブリ言語)

```
func: addu $a0, $a0, $a1
      move $v0, $a0          # 戻り値には  $v0, $v1 を使う
      jr   $ra               # $ra のアドレスへジャンプ
main:                        # 初期化省略
      li   $a0, 5            # 引数には $a0～$a3 を使う
      li   $a1, 1
      jal func               # func へジャンプ; $ra ← 戻り先アドレス
      move $s0, $v0

      li $a0, 8
      li $a1, 2
      jal func
      move $s1, $v0

      move $a0, $s0
      move $a1, $s1
      jal func
      move $s2, $v0
```

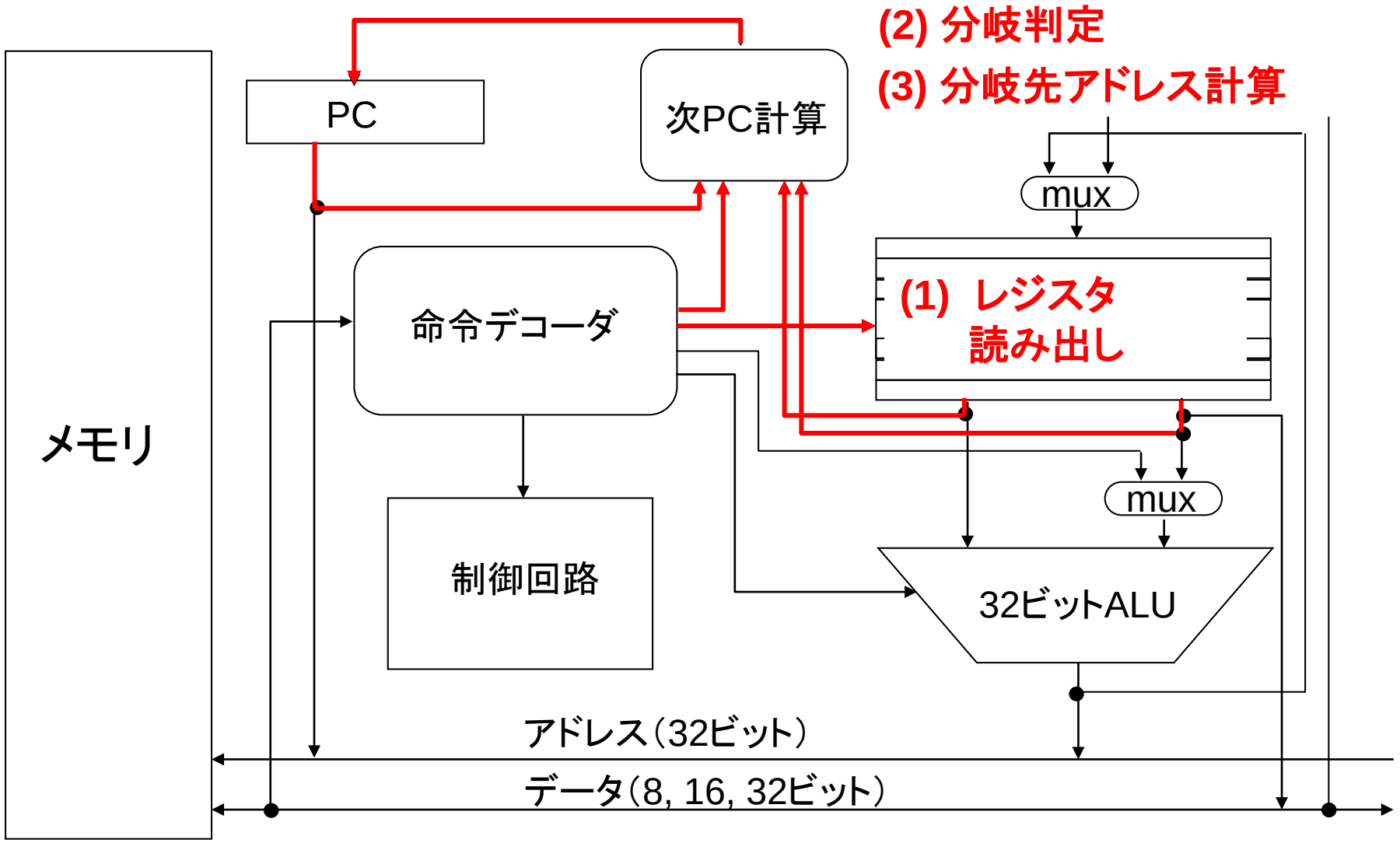
再掲: レジスター覧

番号表示	別名	説明
$0	**$zero**	**常にゼロ**
$1	$at	アセンブラ用に予約
$2, $3	**$v0, $v1**	**関数からの戻り値用**
$4 ～ $7	**$a0 ～ $a3**	**関数への引数用**
$8 ～ $15	**$t0 ～ $t7**	**（主に）一時レジスタ**
$16 ～ $23	**$s0 ～ $s7**	**（主に）変数割り当て用**
$24, $25	**$t8, $t9**	**（主に）一時レジスタ**
$26, $27	$k0, $k1	OS用に予約
$28	$gp	グローバルポインタ
$29	**$sp**	**スタックポインタ**
$30	**$s8**	**（主に）変数割り当て用**
$31	**$ra**	**リターンアドレス**

資料: 主なジャンプ命令

命令	説明
j L	ラベル L へジャンプ (jump)
jal L	ラベル L へジャンプすると同時に，次の命令アドレスを $ra ($31) に保存 (jump and link)
jr $r	レジスタ r に保存されたアドレスへジャンプ (jump register)

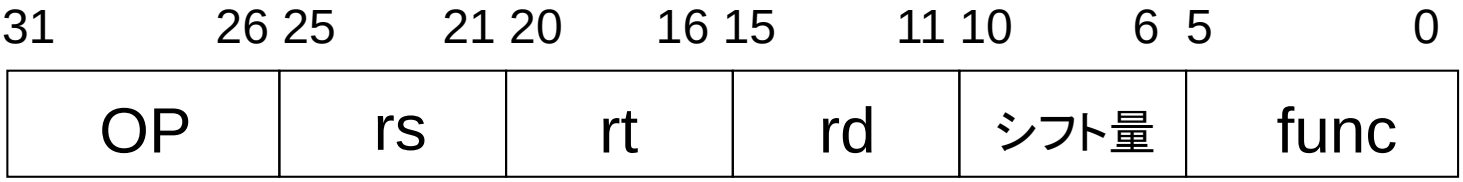
分岐・ジャンプ命令の動作



MIPS の命令フォーマット

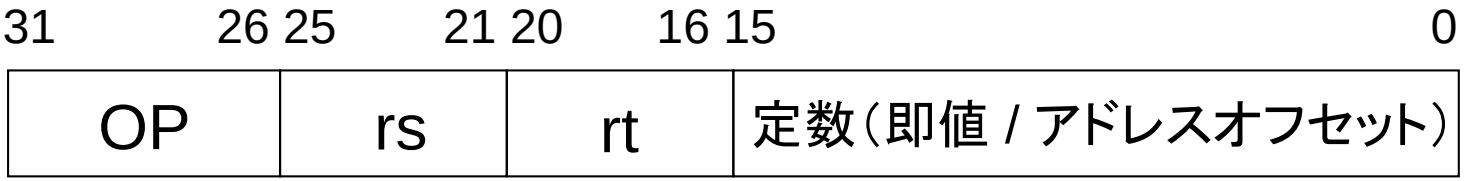
MIPSの命令は基本的に以下の3フォーマットに分類される

R形式



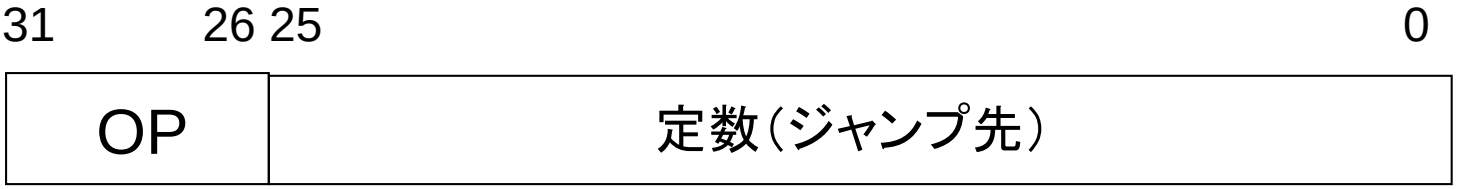
レジスタ間演算，シフト命令（即値版も含む）など

I形式



レジスタ-即値間演算，ロード・ストア，分岐命令など

J形式



ジャンプ命令など

参考:「次PC計算」の処理

分岐条件が成立していないなら,

- PC ← PC + 4　　# 次の命令のアドレス

成立しているなら,

- 条件分岐命令のとき:

  PC ← PC + 4×定数フィールド値

  # 現在位置からの相対アドレス

- ジャンプ命令のとき:

  PC の下位28ビット ← 4×定数フィールド値

  # 256 Mbytes 境界内でジャンプ

- ジャンプレジスタ命令のとき:

  PC ← 入力オペランドレジスタの値

※ 命令長は常に4バイトなので，分岐・ジャンプ先のアドレスは必ず 4の倍数になる

※ この講義では「遅延分岐」は無視する

関数呼出しとスタックメモリ

実際の関数呼出し

前の関数呼出しの例は，簡単に済むように巧妙に作られた例である．実際には以下のようなことを考えなくてはならない．

- 呼出された関数はどのレジスタを使えばよいのか？特に，呼出された関数が呼出し元のレジスタを破壊しないためにはどうすればよいのか？
- レジスタ ra は一つしかないが，関数を多重で呼出す場合は?
- 4つを超える引数の受け渡しは?

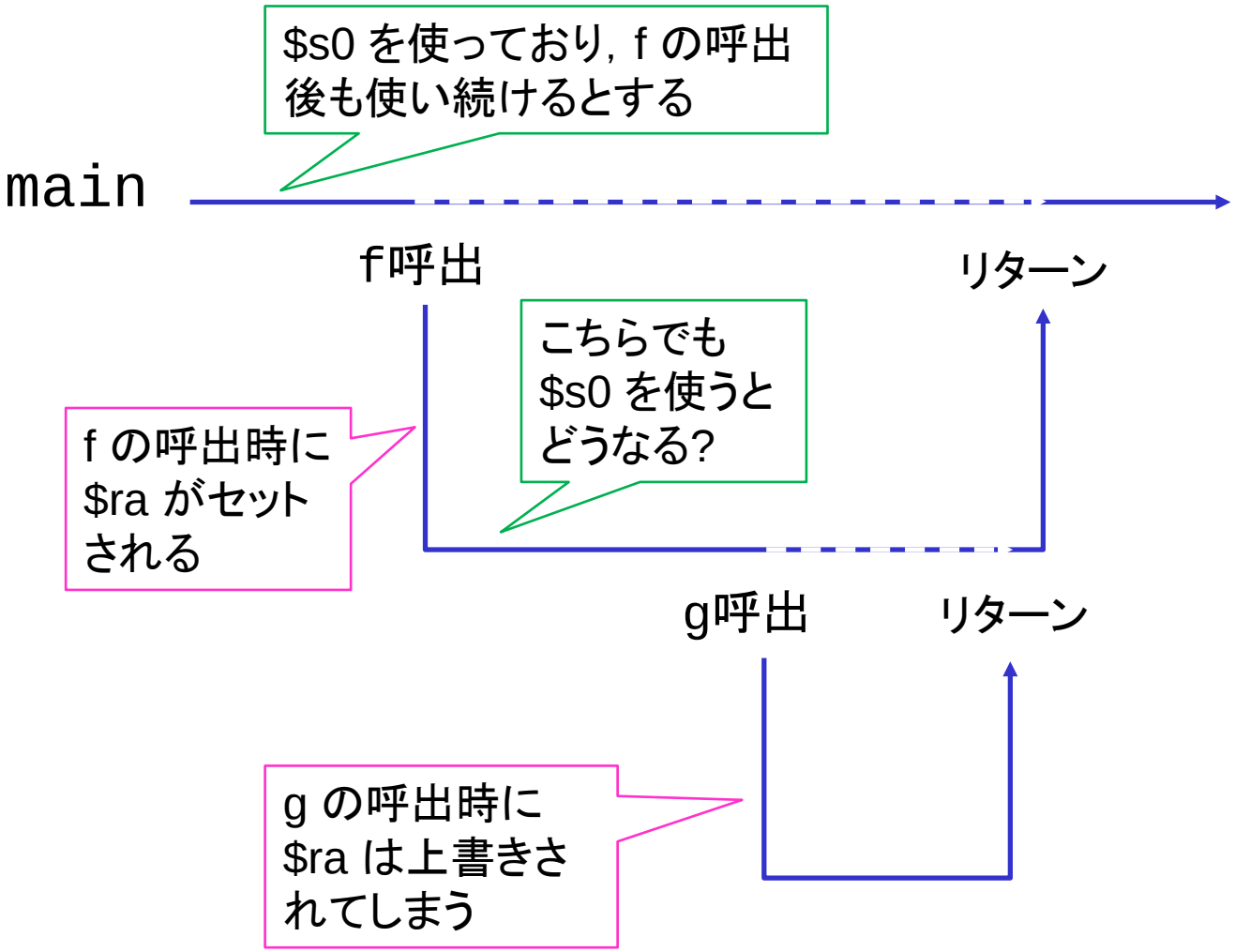
これらは**スタック**と呼ばれるデータ構造をメモリ内に構築することで取り扱われる

関数呼出し時のプログラムの流れ

```
int g() {
    …
}

int f() {
    …
    g();
}

int main() {
    …
    f();
    …
}
```



スタックメモリ

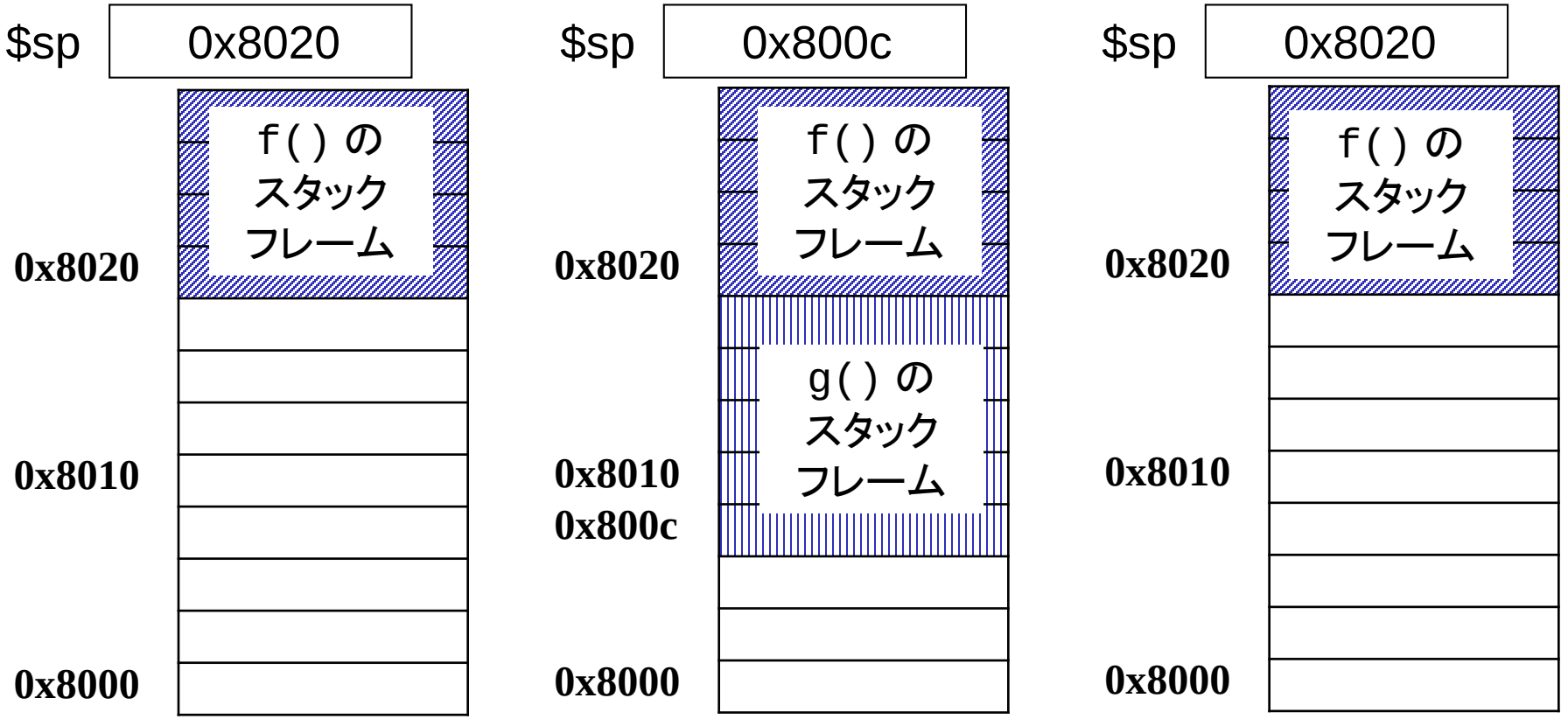
```
int f() {
    g();
}
```

addu $sp, $sp, -20

push

addu $sp, $sp, 20

pop



参考: MIPSのレジスタ・スタック利用ルール

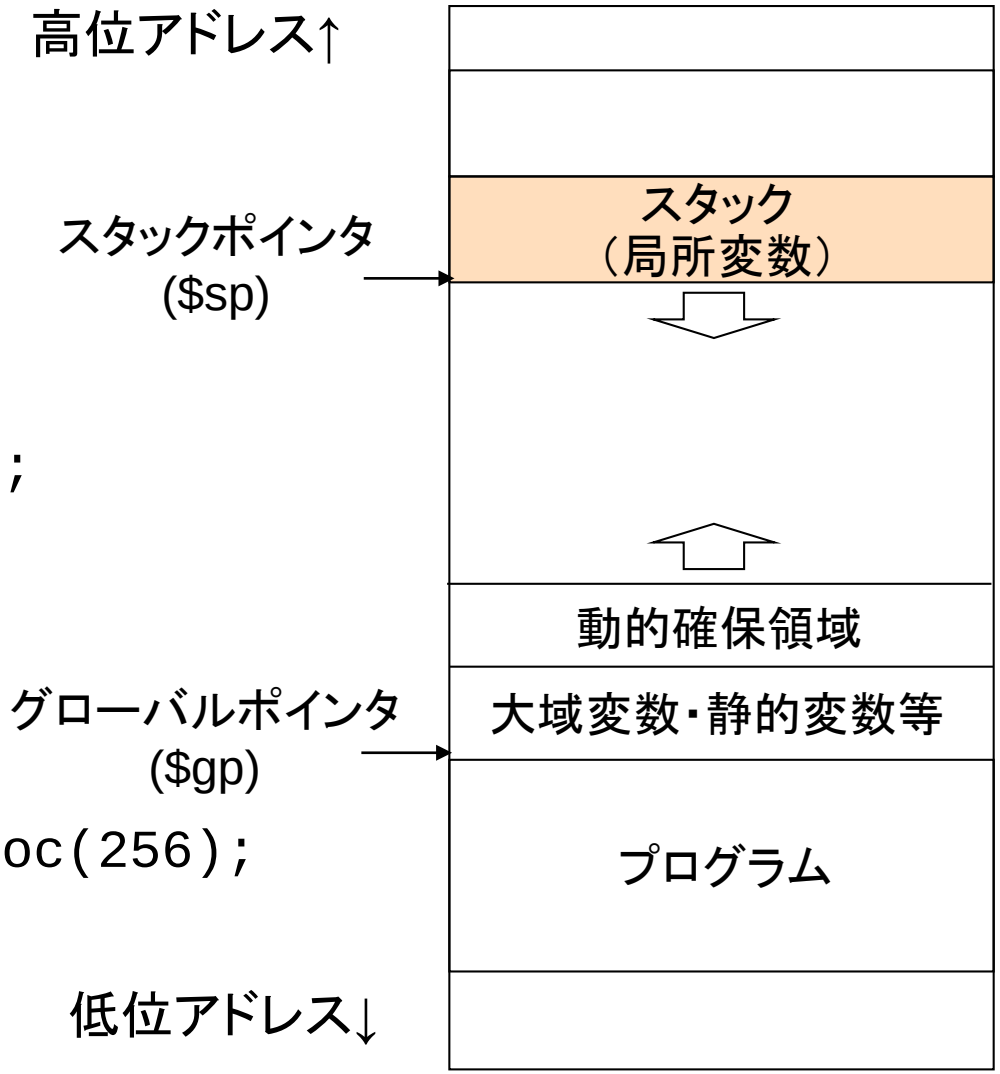
- 関数呼出時のレジスタ利用規約
  - t0～t9 は，壊されたくないなら呼出し側がスタックに退避（すなわち，主にテンポラリ用と想定されている）
  - s0～s8 は，呼出された側が使いたいならスタックに退避してリターン時に原状回復する
- 関数を多重で呼出す場合は ra もスタックに退避
- 4つを超える引数はスタックに積んでから関数を呼出す

典型的なメモリマップ

```
int global_var;

void func() {
    int a, b;
    static int static_var;
    ...
}

int main() {
    int x, y;
    int *p = (int *) malloc(256);
}
```



例: C言語あるある

```
int *add_vector(int a[3], int b[3]) {
    int c[3];
    for (int k = 0; k < 3; k++) {
        c[k] = a[k] + b[k];
    }
    return c; /* ローカル配列 c[] の先頭アドレスを返す （返すな） */
}

int main() {
    int x[3] = { 1, 2, 3 };
    int y[3] = { 4, 5, 6 };
    int *z = add_vector(x, y);
    printf("z[0] = %d¥n", z[0]);
    printf("z[0] = %d¥n", z[0]);
    return z[0];
}
```

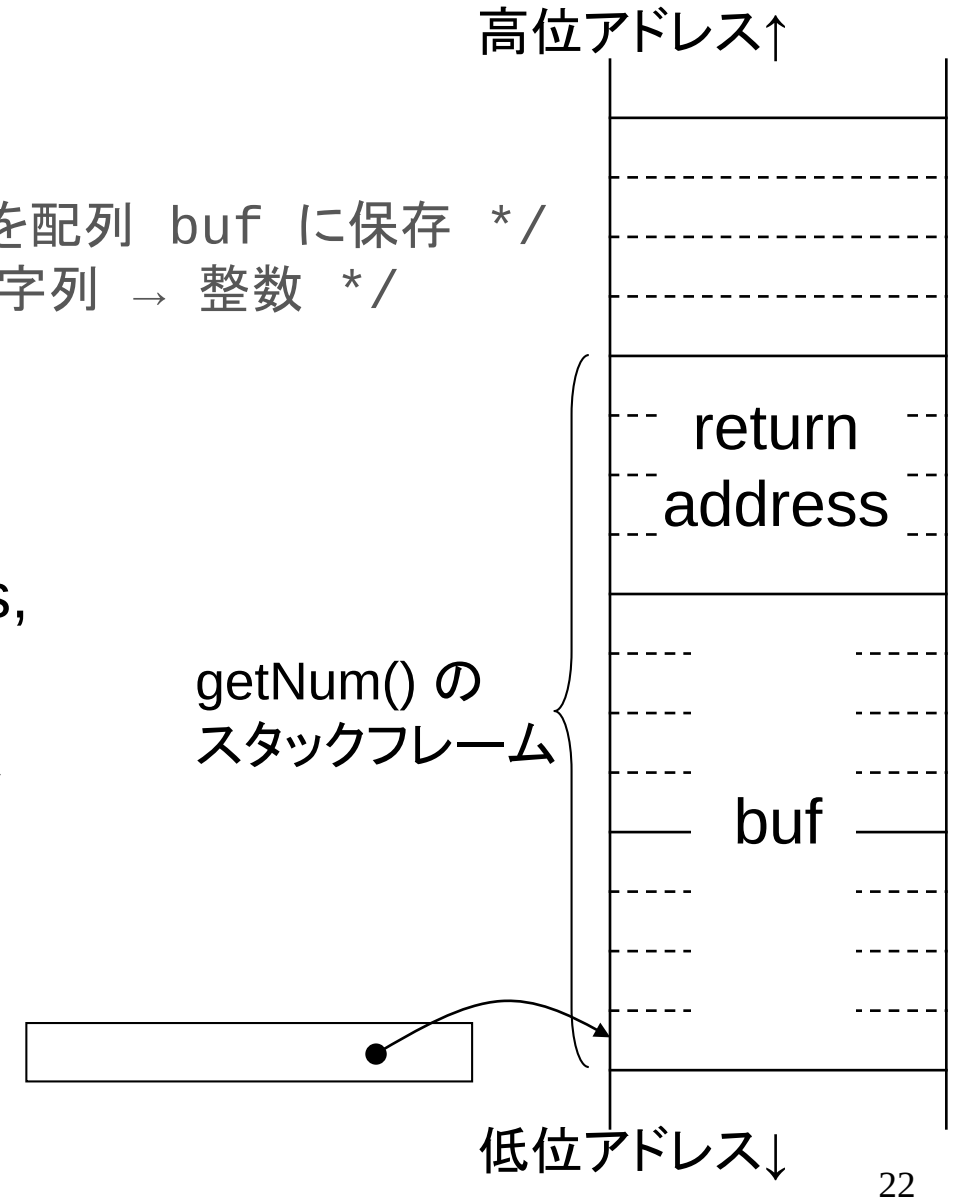
1回目と2回目の printf で結果が違ったり，そもそも異常終了で動かなかったりなど，初心者には不可解に見える挙動をする

https://wandbox.org/permlink/O1fswIR2bUVM0DKJ

例: バッファ・オーバラン攻撃

```
int getnum() {
    char buf[8];
    gets(buf); /* 入力文字列を配列 buf に保存 */
    return atoi(buf); /* 文字列 → 整数 */
}
```

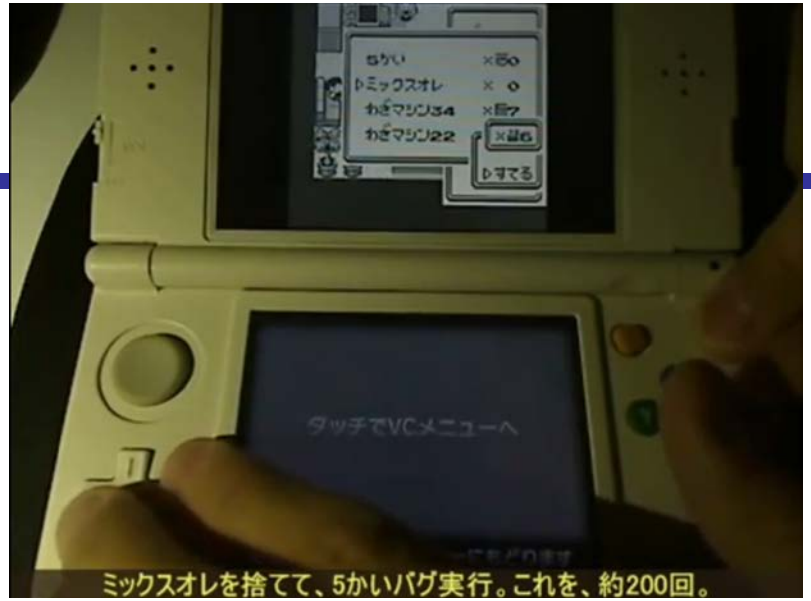
このような問題があるので，gets, scanf のように配列長を指定せずに入力を読み込む関数は，使用が推奨されない

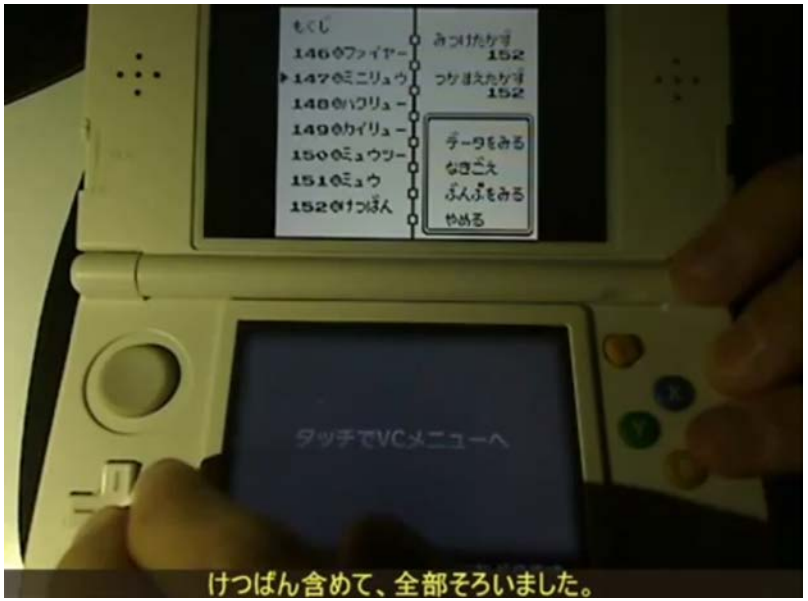


余談: ポケットモンスター「5かい」バグ

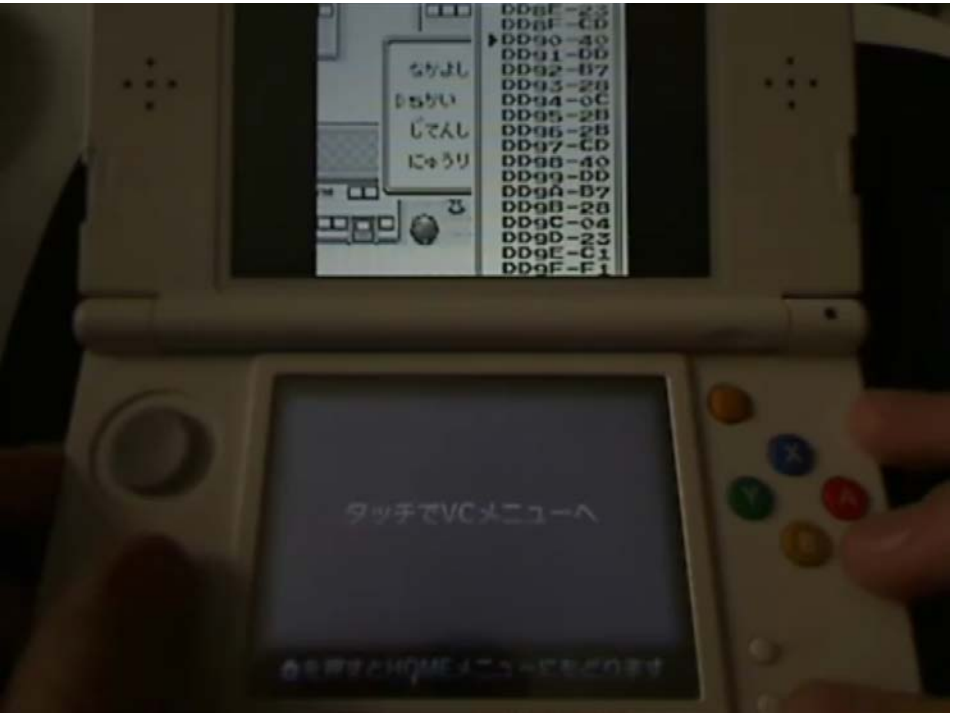
ゲームボーイ用ゲーム「ポケットモンスター 赤」「ポケットモンスター 緑」（任天堂, 1996）には，プレイヤが保有している道具やポケモン（ゲーム世界内の架空生物）などを並べ替える機能があったが，この機能の実装に不具合があり，特定の操作によって本来は存在しないはずの道具（俗に「バグアイテム」）を入手することができた．

道具を使ったときに生じる効果は，道具ごとに定められているアドレスに対する呼出しによって実現されており，バグアイテムを使用すると想定外のアドレスへの呼出しが生じた．特に「5かい」という名称で表示されるバグアイテムの場合は，呼出されるアドレス以降がポケモンの種類やその状態等を保持するメモリ領域となっており，プレイヤがある程度自由に設定することができた．結果として，任意のプログラムを作成し実行することが可能となっていた．





https://www.youtube.com/watch?v=IJ7mRJISeO0



https://www.youtube.com/watch?v=baXxP6b7ANQ

実際のコンパイラが出力するコードを見てみる

https://godbolt.org/

左側の言語として C を選択 → 適当な C の関数を入力
右側のコンパイラとして（例えば）MIPS gcc 5.4 を選択

- その右の入力欄（コンパイラへのオプション指定）に:

```
-O -fomit-frame-pointer -fno-delayed-branch
```

-O0 とすると最適化がオフになる

```
int test(int x, int y) {
    if (x == y) {
        x = x + 1;
    }
    return x;
}
```

$2 は $v0, $31 は $ra

$4, $5, $6, $7 は引数

```
test:
        move    $2,$4
        bne     $4,$5,$L2
        nop

        addiu   $2,$4,1
$L2:
        j       $31
        nop
```

nop は気にしない

練習問題(1)

以下に示すプログラムは，ある値を 2 進数で表した際に，その中に含まれる「1」のビットの数を数えるものである.

```
        lw   $s0, 0($s1)
        move $t0, $zero
L1:     and  $t1, $s0, 1
        addu $t0, $t0, $t1
        srl  $s0, $s0, 1
        bne  $s0, $zero, L1
        sw   $t0, 0($s1)
```

レジスタ s1 の内容が指すアドレスに値13 が格納されている状態でこのプログラムを実行した.

(1) 実行終了時の，レジスタ s0，t0，t1 の内容を示せ.

(2) ラベル L1 で指される命令は，何回実行されるか答えよ.

練習問題(2)

以下に示すプログラムは配列の中から最大値を探すものである.

```
        move    $v0, $zero
L1:     lw      $t0, 0($s0)             … (※1)
        sltu    $t1, $v0, $t0
        beq     $t1, $zero, L2
        move    $v0, $t0                … (※2)
L2:     addu    $s0, $s0, 4
        addu    $s1, $s1, -1
        bne     $s1, $zero, L1
```

いま, 符号なし整数 (4バイト) が, メモリ上のアドレスが増える方向に
10, 20, 3, 22, 5
の順に並んでおり, この配列の先頭アドレスをレジスタ s0, 配列長 5 をレジスタ s1 に与えてこのプログラムを実行した.

(1) プログラムの実行が終わった時点でのレジスタ s1, v0, t0, t1 の内容を示せ.

(2) ※1 及び ※2 の命令がそれぞれ何回実行されるか答えよ.

解答例 (1)

```
     lw $s0, 0($s1)          $s0 = mem[$s1 + 0]; /* = 13 */
     move $t0, $zero         $t0 = 0;
L1:                          do {
     and  $t1, $s0, 1          $t1 = $s0 & 1;
     addu $t0, $t0, $t1        $t0 = $t0 + $t1;
     srl $s0, $s0, 1           $s0 = $s0 >> 1;
     bne $s0, $zero, L1      } while ($s0 != 0);
     sw $t0, 0($s1)          mem[$s1 + 0] = $t0;
```

(1) s0 は計算対象で，1ビットずつシフトして行き，最後は 0 になる
t0 は最終結果で，ビット1の数になる．よって 3
t1 は計算途中で使用し，ループごとに s0 の最下位ビットを取り出すのに使われる．最後は 1 になる
(2) 4回

解答例 (2)

```
    move $v0, $zero
L1: lw $t0, 0($s0)
    sltu $t1, $v0, $t0
    beq $t1, $zero, L2
    move $v0, $t0

L2: addu  $s0, $s0, 4
    addu  $s1, $s1, -1
    bne $s1, $zero, L1
```

```
$v0 = 0;
do {
   $t0 = mem[$s0 + 0];
   if ($v0 < $t0) {
       $v0 = $t0;
   }
   $s0 = $s0 + 4;
   $s1 = $s1 – 1;
} while ($s1 != 0)
```

(1) s1は処理すべき配列要素の残り数で，終了時は0になる．v0 はその時点までの最大値を保持するレジスタで，終了時は全体の最大値22になる．t0 は読み出した配列要素を格納するレジスタで，終了時は最後の要素 5 になる．t1 は，それまでの最大値が読み出した配列要素より小さければ1になるレジスタで(これが0のときはL2に分岐するため，最大値の更新がスキップされる)，最後のループでは 0 になる．

(2) ※1 は，配列長が5なので5回実行される．※2 は，最大値を更新するときのみ実行されるため3回実行される．