第二回パタヘネ輪読会 前半

2021/05/21 (金)

今日やること

- 前半では 2.1 ~ 2.10 とそれに関連する演習問題をやります

2.1　はじめに

- コンピュータに **命令** (instruction) を伝える
- コンピュータの語彙を **命令セット** (instruction set) という
- 本書で使用する命令セット
  - MIPS
  - ARMv8
  - Intel x86
  - ARMv7

補足 1: 今年 ARMv9 が出る

補足 2: 一番馴染みが深いのは RISC-V だと思う

2.1　はじめに

- 命令セット間には類似性がある
- 設計者が抱く共通の目標
  - ハードウェア、コンパイラの開発を容易にする
  - 性能を最大化する
  - コストと消費電力を最小化する
- 命令もデータも、メモリ内に収まった数値である
  - **プログラム内蔵方式** という

2.2　コンピュータ・ハードウェアの演算

- MIPS では加算を以下のように書く

```
add a, b, c
```

  - `b` と `c` の和を `a` に収めることを意味する

- ４つの変数 `b` と `c` と `d` と `e` の和を `a` に収めたいとしたら？

```
add a, b, c
add a, a, d
add a, a, e
```

2.2　コンピュータ・ハードウェアの演算

- 例えば加算では、必要な **オペランド** (operand) が 3 つある

```
add a, b, c
```

  - 加算対象の 2 つと、それを収める 1 つ
- すべての命令でオペランドを 3 つにすれば単純性が保たれる

設計原則 1: 単純性は規則性につながる。

2.2　コンピュータ・ハードウェアの演算

例題

- 以下の C 言語のコードを MIPS にコンパイルしてみる

```
a = b + c;
d = a - e;
```

- 結果

```
add a, b, c
sub d, a, e
```

2.2　コンピュータ・ハードウェアの演算

例題

- 以下の C 言語のコードを MIPS にコンパイルしてみる

```
f = (g + h) - (i + j);
```

- 結果

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

2.3　コンピュータ・ハードウェアのオペランド

- オペランドは **レジスタ** (register) の中から選ぶ必要がある
  - 数に限りがある特殊な記憶領域
- MIPS においてはレジスタ長は 32 ビット
  - 32 ビットをひとまとめにしたものはよく使うので、
    1つの単位として **語** (word) と呼ぶ
- 一般に、現在のコンピュータに備えられているレジスタの数は 32

2.3　コンピュータ・ハードウェアのオペランド

設計原則 2: 小さければ小さいほど高速になる。

- レジスタ数が 32 本である理由
  - クロック・サイクル数を高速に保つこととのバランス
  - 命令形式中のビット数のため
- MIPS では、レジスタに以下の名前がついている

```
$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-v1, $gp, $fp, $ra, $at
```

2.3　コンピュータ・ハードウェアのオペランド

例題

- 先程のプログラムで、変数 `f`, `g`, `h`, `i`, `j` をそれぞれレジスタ `$s0`, `$s1`, `$s2`, `$s3`, `$s4` に割り付けたときのコンパイル結果は？

```
f = (g + h) - (i + j);
```

- 結果

```
add $t0, $s1, $s2   # g + h の結果を $t0 に代入
add $t1, $s3, $s4   # i + j の結果を $t1 に代入
sub $s0, $t0, $t1   # $t0 - $t1 の結果を f に代入
```

2.3　コンピュータ・ハードウェアのオペランド

- 変数が配列や構造体など複雑なものだったら？
  - 含まれる要素の数はレジスタ数よりも多いかもしれない
- **データ転送命令** (data transfer instruction) を使い、
  メモリとレジスタの間でデータを転送する必要がある
  - **アドレス** (address) を指定してメモリ内の語にアクセス
- MIPS では `lw` (load word) を使ってメモリからレジスタへ転送
  - 命令操作の名前
  - データをロードする先のレジスタ
  - メモリにアクセスするための定数とレジスタ

2.3　コンピュータ・ハードウェアのオペランド

- 例題: 以下をコンパイルせよ。

```
g = h + A[8];
```

  - 変数 `g` と `h` は `$s1` と `$s2` に割り付ける
  - `A` は 100 語からなる配列で、**ベースアドレス** は `$s3` に収められている

- 結果

```
lw  $t0, 8($s3)     # A[8] を $t0 に代入
add $s1, $s2, $t0
```

  - `8` をオフセット、`$s3` をベース・レジスタと呼ぶ

2.3　コンピュータ・ハードウェアのオペランド

- 細々した話
  - 配列や構造体をメモリにどう配置するか決めるのはコンパイラ
  - 今日ではほぼすべて 8 ビットの **バイト** 単位で考える
  - MIPS では語アドレスは 4 の倍数でなければならない
    - アーキテクチャによる **整列化制約**
  - 語の中のバイトの並べ方には **リトル・エンディアン** と **ビッグ・エンディアン** があり、MIPS はビッグ・エンディアン
  - 先の例題では、オフセットを 8 にするために
    実際には $4 \times 8 = 32$ を収めなければならない

2.3　コンピュータ・ハードウェアのオペランド

- 逆にレジスタからメモリへデータを転送するのは **ストア** 命令
- MIPS では `sw` (store word)
  - 命令操作の名前
  - データをストアする元のレジスタ
  - 配列要素を指定するオフセットとベース・レジスタ

2.3　コンピュータ・ハードウェアのオペランド

- 例題: 以下をコンパイルせよ。

```
A[12] = h + A[8];
```

  - 変数 `h` は `$s2` に割り付ける
  - 配列 `A` のベースアドレスは `$s3` に収められている

- 結果

```
lw  $t0, 32($s3)   # A[8] を $t0 に代入
add $t0, $s2, $t0  # h + A[8] を $t0 に代入
sw  $t0, 48($s3)   # h + A[8] を A[12] に収める
```

2.3　コンピュータ・ハードウェアのオペランド

- たいていのプログラムでは、レジスタ数よりも変数の数が多い
- 頻繁に使われる変数はレジスタに保持し、それ以外はメモリに置く
- これを **スピル・アウト** (spilling) という
- レジスタ
  - 数が限られている
  - アクセスが速い、エネルギー消費が少ない
- メモリ
  - アクセスに時間がかかる
  - 読み書きするのにデータ転送命令が余分に要る

2.3　コンピュータ・ハードウェアのオペランド

- 命令操作に定数を使うことは多い
- メモリに定数を置いてそれをロードするのではなく、
  専用の命令を用意する方が効率が良い
- 即値加算 `addi` 命令

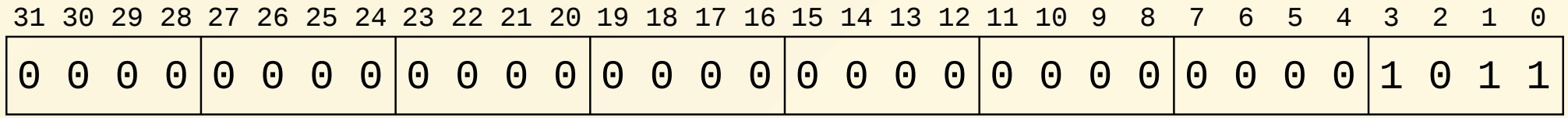
```
addi  $s3, $s3, 4   # $s3 = $s3 + 4
```

「一般的な場合を高速化する」

補足: `subi` は無い

2.4　符号付き数と符号なし数

- コンピュータは電気信号の高低による 2 進数を使う
- 数値 1011_2 は以下のように配置される



- 右端を **最下位ビット** (least significant bit)、
  左端を **最上位ビット** (most significant bit) という

2.4　符号付き数と符号なし数

- 符号なし数

```
0000 0000 0000 0000 0000 0000 0000 0000₂ = 0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ = 1₁₀
0000 0000 0000 0000 0000 0000 0000 0010₂ = 2₁₀
...
1111 1111 1111 1111 1111 1111 1111 1101₂ = 4,294,967,293₁₀
1111 1111 1111 1111 1111 1111 1111 1110₂ = 4,294,967,294₁₀
1111 1111 1111 1111 1111 1111 1111 1111₂ = 4,294,967,295₁₀
```

$$\sum_{i=0}^{31} x_i \times 2^i$$

2.4　符号付き数と符号なし数

- 負の数を扱うには？
- ひとつ考えられる方法
  - 1 ビットを符号に使う (符号付き絶対値表現)
- 符号付き絶対値表現の欠点
  - 符号ビットはどこに置く？
  - 加算器に余分なステップが必要
  - ゼロが正と負の 2 つあり、バグの原因に

2.4　符号付き数と符号なし数

- **2 の補数** (two's complement) 表現

```
0000 0000 0000 0000 0000 0000 0000 0000₂ =  0₁₀
0000 0000 0000 0000 0000 0000 0000 0001₂ =  1₁₀
0000 0000 0000 0000 0000 0000 0000 0010₂ =  2₁₀
...
0111 1111 1111 1111 1111 1111 1111 1110₂ =  2,147,483,646₁₀
0111 1111 1111 1111 1111 1111 1111 1111₂ =  2,147,483,647₁₀
1000 0000 0000 0000 0000 0000 0000 0000₂ = -2,147,483,648₁₀
1000 0000 0000 0000 0000 0000 0000 0001₂ = -2,147,483,647₁₀
...
1111 1111 1111 1111 1111 1111 1111 1101₂ = -3₁₀
1111 1111 1111 1111 1111 1111 1111 1110₂ = -2₁₀
1111 1111 1111 1111 1111 1111 1111 1111₂ = -1₁₀
```

2.4　符号付き数と符号なし数

- **2 の補数** (two's complement) 表現
- 最上位が 0 の場合は正の数 ($0 \ldots 2{,}147{,}483{,}647$) を表し、
  最上位が 1 の場合は負の数 ($-2{,}147{,}483{,}648 \ldots -1$) を表す

$$x_{31} \times \left(-2^{31}\right) + \sum_{i=0}^{30} x_i \times 2^i$$

補足: $-2{,}147{,}483{,}648$ に対応する正の数がないという問題はある

2.4　符号付き数と符号なし数

- 正負反転
  - ビット反転させてから 1 を加えればよい
- 例: 2_{10} を正負反転する

```
2₁₀ = 0000 0000 0000 0000 0000 0000 0000 0010₂

      1111 1111 1111 1111 1111 1111 1111 1101₂
    +                                       1₂
    ──────────────────────────────────
    = 1111 1111 1111 1111 1111 1111 1111 1110₂ = -2₁₀
```

  - -2_{10} も同様

2.4　符号付き数と符号なし数

- **符号拡張** (sign extension)
  - より大きいビット長に変換する
  - 最上位ビットを空きビットに繰り返しコピーする
- 例: 16 ビットの 2_{10} と -2_{10} をそれぞれ 32 ビットに符号拡張する

```
                    0000 0000 0000 0010₂ =  2₁₀
                    ↓ 符号拡張
0000 0000 0000 0000 0000 0000 0000 0010₂ =  2₁₀

                    1111 1111 1111 1110₂ = -2₁₀
                    ↓ 符号拡張
1111 1111 1111 1111 1111 1111 1111 1110₂ = -2₁₀
```

2.4　符号付き数と符号なし数

- 符号拡張の必要性
- 例: MIPS でバイトを 32 ビットのレジスタにロードする
- `lb` (load byte)
  - バイトを符号付き整数としてロード
  - 24 ビットを符号拡張する
- `lbu` (load byte unsigned)
  - バイトを符号なし整数としてロード
  - 24 ビットを 0 で埋める

2.4　符号付き数と符号なし数

- 補足: 符号付き数の表現方法は色々ある
  - 符号付き絶対値表現
  - 2 の補数
  - 1 の補数
  - ゲタばき表現

2.5　コンピュータ内での命令の表現

- コンピュータ内では命令は数値
- 例: 以下の記号表現の命令を、数値として表す

```
add $t0, $s1, $s2
```

  - レジスタ `$s1` は 17、`$s2` は 18、`$t0` は 8 に対応する
- MIPS の命令形式に従い 10 進数で表すと

0	17	18	8	0	32

  - 区切られた各部分をフィールドという

2.5　コンピュータ内での命令の表現

- さらに 2 進数にすると

000000	10001	10010	01000	00000	100000

  - 1 命令は 32 ビットであることがわかる

- 命令を数値で表したものを **機械語** (machine language) という
- 機械語はたいてい 16 進数で表される
  - 例: `1110 1100 1010 1000 0110 0100 0010 0000`$_2$ → `eca8 6420`$_{16}$
  - 例: `0001 0011 0101 0111 1001 1011 1101 1111`$_2$ → `1357 9bdf`$_{16}$
  - 2 進数の 4 ビットを `0123456789abcdef` で置き換える

2.5　コンピュータ内での命令の表現

- MIPS のフィールド

op	rs	rt	rd	shamt	funct
6 ビット	5 ビット	5 ビット	5 ビット	5 ビット	6 ビット

  - **op**: 命令操作コード (opcode, オペコード)
  - **rs**: 第 1 ソース・オペランドのレジスタ
  - **rt**: 第 2 ソース・オペランドのレジスタ
  - **rd**: ディスティネーション・オペランドのレジスタ
  - **shamt**: シフト量 (後述)
  - **funct**: 機能コード (function code)。命令操作のバリエーション

2.5　コンピュータ内での命令の表現

- `lw` 命令はこれだと困る
  - レジスタ 2 つとアドレス 1 つを指定
  - アドレスに 5 ビットを当てるとすると 32 までしか指定できず、役に立たない

設計原則 3: 優れた設計には適度な妥協が必要である。

- 前のは **R 形式** ということにして、これとは別に **I 形式** も用意する

op	rs	rt	constant または address
6 ビット	5 ビット	5 ビット	16 ビット

  - 即値およびデータ転送命令用

2.5　コンピュータ内での命令の表現

命令	形式	op	rs	rt	rd	shamt	funct
					constant / address		
`add`	R	0	レジスタ	レジスタ	レジスタ	0	32_{10}
`sub`	R	0	レジスタ	レジスタ	レジスタ	0	34_{10}
`addi`	I	8_{10}	レジスタ	レジスタ	定数		
`lw`	I	35_{10}	レジスタ	レジスタ	アドレス		
`sw`	I	43_{10}	レジスタ	レジスタ	アドレス		

`lw` においては rt がディスティネーション・レジスタである

2.5　コンピュータ内での命令の表現

- 例題: 以下のプログラムがある。

```
A[300] = h + A[300]
```

これは以下のようにコンパイルされる。

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

これの機械語コードは？

2.5　コンピュータ内での命令の表現

アセンブリ → 機械語

```
lw  $t0, 1200($t1)
add $t0, $s2, $t0
sw  $t0, 1200($t1)
```

op	rs	rt	rd	shamt	funct
			constant / address		
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

2.5　コンピュータ内での命令の表現

10 進表現 → 2 進表現

op	rs	rt	rd	shamt	funct
			constant / address		
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

2.5　コンピュータ内での命令の表現

- **プログラム内蔵方式**
  1. 命令は数値として表現される
  2. プログラムをメモリ中に格納して、データと同様に読み書きできる
- 使いたいプログラムとデータをメモリにロードし、
  指定の場所から実行すれば、色々なプログラムを実行可能
- ハードウェアとソフトウェアの単純化

2.6　論理演算

MIPS の論理演算 ①

- `sll` (shift left logical)
- `srl` (shift right logical)

```
sll $t2, $s0, 4  # $t2 = $s0 << 4
```

op	rs	rt	rd	shamt	funct
0	0	16	10	**4**	0

- shamt (shift amount) を使う

2.6　論理演算

MIPS の論理演算 ②

- `and` (and)
- `andi` (and immediate)
- `or` (or)
- `ori` (or immediate)
- AND と OR は定数の使用が多いので即値版がある
  - 符号拡張はしない

補足: `xor` と `xori` もある

2.6　論理演算

MIPS の論理演算 ③

- `nor` (nor)
- NOR 演算
  - A NOR B = NOT (A OR B)
- NOT は代わりにこれを 0 と組み合わせて使う
  - A NOR 0 = NOT (A OR 0) = NOT A

補足 1: 即値版はない

補足 2: レジスタしか指定できないので、`$zero` のことだと思われる

2.7　条件判定用の命令

- `beq` (branch if equal)

```
beq register1, register2, L1
```

  - レジスタ 1 の値とレジスタ 2 の値が等しいときに
    ラベル `L1` に制御の流れを分岐させる

- `bne` (branch if not equal)

```
bne register1, register2, L1
```

  - レジスタ 1 の値とレジスタ 2 の値が等しくないときに
    ラベル `L1` に制御の流れを分岐させる

2.7　条件判定用の命令

- 例: 以下のコードをコンパイルせよ

```
if (i == j) f = g + h; else f = g - h;
```

  - 変数 `f`, `g`, `h`, `i`, `j` はレジスタ `$s0` から `$s4` に割り付ける

- 結果

```
      bne $s3, $s4, Else  # i ≠ j なら Else に分岐
      add $s0, $s1, $s2   # f = g + h (i ≠ j ならスキップ)
      j   Exit            # Exit へジャンプ
Else: sub $s0, $s1, $s2   # f = g - h (i = j ならスキップ)
Exit:
```

2.7　条件判定用の命令

- 例: 以下のコードをコンパイルせよ

```
while (save[i] == k) i += 1;
```

- 結果

```
Loop: sll  $t1, $s3, 2      # $t1 = i << 2
      add  $t1, $t1, $s6    # $t1 = (save[i] のアドレス)
      lw   $t0, 0($t1)      # $t0 = save[i]
      bne  $t0, $s5, Exit   # save[i] ≠ k なら Exit へ
      addi $s3, $s3, 1      # i = i + 1
      j    Loop             # Loop へ
Exit:
```

2.7　条件判定用の命令

- ラベルを使うことで、(コンパイラやアセンブリ言語プログラマは)
  分岐先アドレスの計算にわずらわれずに済む
  - もっと言えば、高水準言語プログラマはラベルについて考えなくても
    コンパイラが勝手に分岐を生成する
- 分岐も分岐先・ラベルもない一続きの塊を **基本ブロック** という

2.7　条件判定用の命令

- `slt` (set on less than)

```
slt  $t0, $s3, $s4  # $s3 < $s4 なら $t0 を 1 にする
```

- `slti` (set on less than immediate)

```
slti $t0, $s2, 10   # $s2 < 10 なら $t0 を 1 にする
```

- `sltu` (set on less than unsigned)
- `sltiu` (set on less than immediate unsigned)
  - 数値を符号付きとして比べるか、符号なしとして比べるか

2.7　条件判定用の命令

- MIPS では「より小」「以下」「より大」「以上」の分岐はすべて `slt`, `slti`, `beq`, `bne`, `$zero` の組み合わせで生成される
  - 単純性を保つべし
- 配列の境界チェック `0 ≦ x < y` を一発で行うテクニック

```
sltu $t0, $s1, $s2   # $s1 >= $t2 または $s1 < 0 なら $t0 を 0 にする
beq  $t0, $zero, IndexOutOfBounds   # 範囲外ならエラー処理に分岐
```

  - 符号付き整数の負の数は、符号なし整数として解釈すれば
    大きな整数のように見える

2.7　条件判定用の命令

case 文と switch 文

- if-then-else をつなげて実現する
- **ジャンプ・アドレス・テーブル** によって実現する
  - 分岐先候補のアドレスを表にして、該当処理へジャンプ
  - レジスタで指定したアドレスにジャンプする命令が必要
  - `jr` (jump register)

2.8　コンピュータ・ハードウェア内での手続きのサポート

- **手続き** (procedure)
  - 要するに関数。プログラムの抽象化を実現する。
- 手続きを実行する手順
  1. 手続きからアクセスできる場所にパラメータを置く
  2. 手続きに制御を移す
  3. 手続きに必要なメモリ資源を入手する
  4. 必要なタスクを実行する
  5. 呼び出し元のプログラムからアクセスできる場所に結果を置く
  6. 制御を元の位置に戻す
     - 手続きはプログラム内のいろいろな場所から呼び出される可能性がある

2.8　コンピュータ・ハードウェア内での手続きのサポート

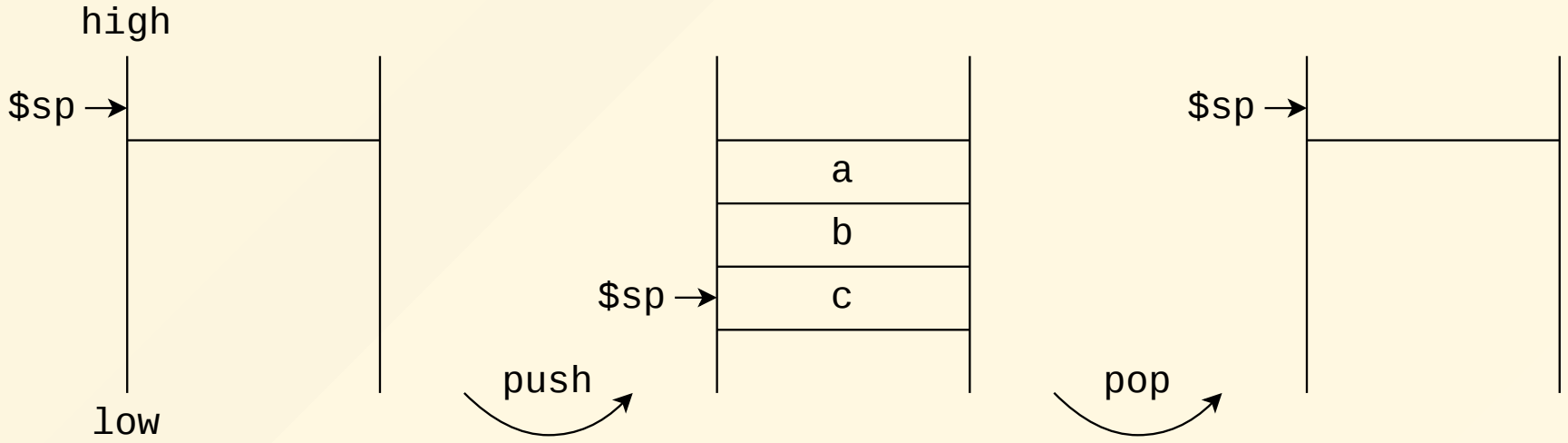
- 受け渡しは極力レジスタを使う
  - `$a0-$a3`: 引数レジスタ。パラメータを渡す。
  - `$v0-$v1`: 値レジスタ。結果の値を返す。
  - `$ra`: 戻りアドレス・レジスタ。制御を元に戻す。
- `jal` (jump-and-link)
  - `jal [手続きのアドレス]` で手続きのアドレスにジャンプすると同時に、次の命令のアドレス (PC + 4) を `$ra` に退避する
    - PC: **プログラム・カウンタ**
- `jr` (jump register)
  - `jr $ra` で制御を元に戻す

2.8　コンピュータ・ハードウェア内での手続きのサポート

- 用語
  - **呼び出し側** (caller)
  - **被呼び出し側** (callee)
- 手続きを実行する手順まとめ
  - 呼び出し側はパラメータを `$a0-$a3` に収める
  - `jal X` で手続き X (被呼び出し側) にジャンプ
  - 被呼び出し側は計算を実行
  - 結果を `$v0-$v1` に収める
  - `jr $ra` で制御を呼び出し側に戻す

2.8　コンピュータ・ハードウェア内での手続きのサポート

- 受け渡すデータがもっと多い場合は？
- **スタック** (stack) を使う
  - `$sp` (stack pointer) がスタックのトップを指し示す
  - レジスタを 1 つ退避すれば、`$sp` は 1 語分下に伸びる (**プッシュ**)
  - レジスタを 1 つ復元すれば、`$sp` は 1 語分上に縮む (**ポップ**)



2.8　コンピュータ・ハードウェア内での手続きのサポート

- 例: 以下の手続きをコンパイルせよ

```
int leaf_example(int g, int h, int i, int j)
{
  int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - 引数 `g`, `h`, `i`, `j` は `$a0`, `$a1`, `$a2`, `$a3` に相当
  - 変数 `f` は `$s0` に相当
  - 代入の前にレジスタを退避する必要がある

2.8　コンピュータ・ハードウェア内での手続きのサポート

```
leaf_example:
    addi $sp, $sp, -12      # スタックを 3 語分確保
    sw   $t1, 8($sp)        # $t1 を退避
    sw   $t0, 4($sp)        # $t0 を退避
    sw   $s0, 0($sp)        # $s0 を退避
    add  $t0, $a0, $a1      # $t0 = g + h
    add  $t1, $a2, $a3      # $t1 = i + j
    sub  $s0, $t0, $t1      # f = (g + h) - (i + j)
    add  $v0, $s0, $zero    # f を返す ($v0 = $s0 + 0)
    lw   $s0, 0($sp)        # $s0 を復元
    lw   $t0, 4($sp)        # $t0 を復元
    lw   $t1, 8($sp)        # $t1 を復元
    addi $sp, $sp, 12       # スタックから 3 語分除く
    jr   $ra                # 呼び出し元に戻る
```

2.8　コンピュータ・ハードウェア内での手続きのサポート

- MIPS ではレジスタを以下のように分けている
  - `$t0-$t9`: 手続き呼び出しの際に被呼び出し側で保存しない
  - `$s0-$s7`: 手続き呼び出しの際に保存されなければならない
    - 使用する場合、被呼び出し側で退避および復元する

2.8　コンピュータ・ハードウェア内での手続きのサポート

- 他の手続きを呼び出さない手続きを **リーフ** (leaf) 手続きと呼ぶ
- 実際はリーフ手続きばかりではない
- 例: 以下の再帰的手続きをコンパイルせよ

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

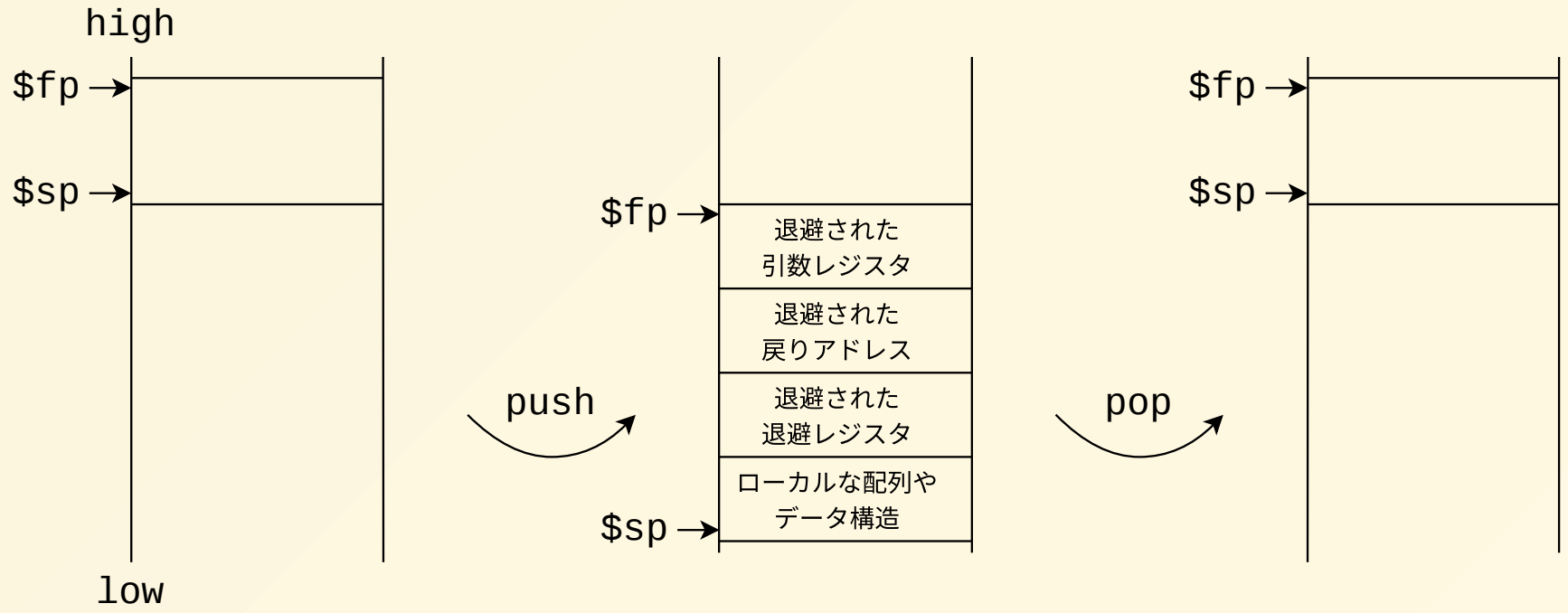
  - 呼び出し側は引数レジスタと一時レジスタを退避
  - 被呼び出し側は戻りアドレス・レジスタと退避レジスタを退避

2.8　コンピュータ・ハードウェア内での手続きのサポート

```
fact:
    addi $sp, $sp, -8      # スタックを 2 語分確保
    sw   $ra, 4($sp)       # 戻りアドレスを退避
    sw   $a0, 0($sp)       # 引数 n を退避
    slti $t0, $a0, 1       # n < 1 かをチェック
    beq  $t0, $zero, L1    # n ≧ 1 であれば L1 に分岐
    addi $v0, $zero, 1     # 1 を返す
    addi $sp, $sp, 8       # スタックから 2 語分除く
    jr   $ra               # 呼び出し元に戻る
L1: addi $a0, $a0, -1      # n ≧ 1 であれば引数を (n - 1) にする
    jal  fact              # (n - 1) を用いて fact を呼び出す
    lw   $a0, 0($sp)       # 引数 n を復元
    lw   $ra, 4($sp)       # 戻りアドレスを復元
    addi $sp, $sp, 8       # スタックから 2 語分除く
    mul  $v0, $a0, $v0     # n * fact(n - 1) を返す
    jr   $ra               # 呼び出し元に戻る
```

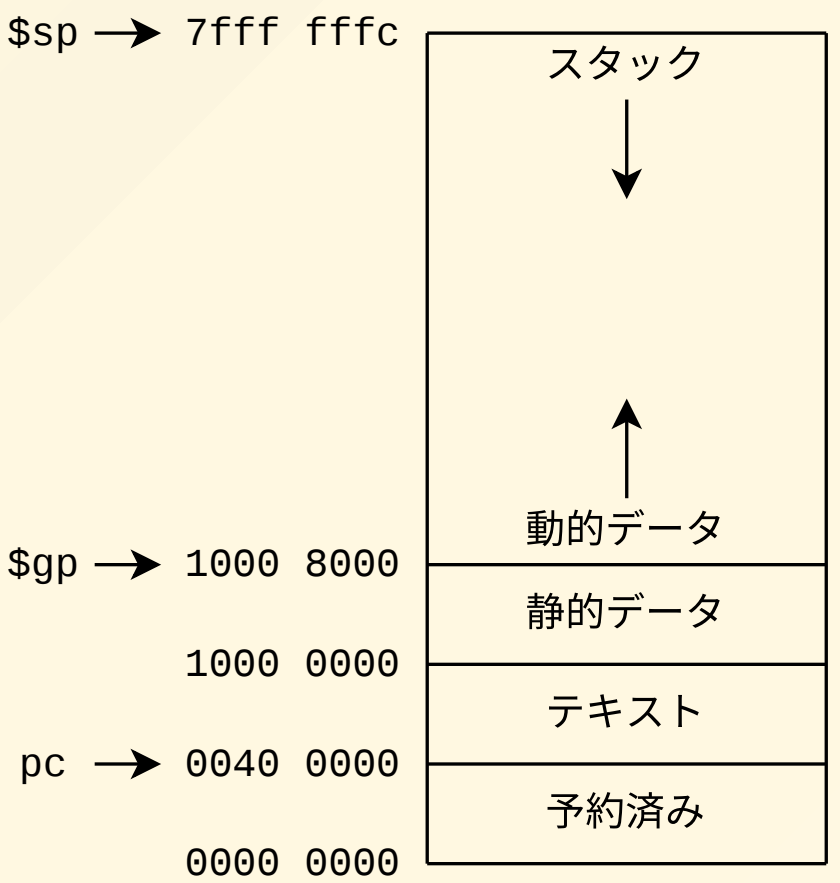
2.8　コンピュータ・ハードウェア内での手続きのサポート

- 手続きは退避されたレジスタやローカル変数をスタックに置く
  - この領域を **手続きフレーム** または **アクティベーション・レコード** という
- `$fp` (frame pointer) はフレームの先頭を指す



2.8　コンピュータ・ハードウェア内での手続きのサポート

- テキスト・セグメント
  - 機械語を置く
- 静的データ・セグメント
  - 定数や静的変数を置く
- ヒープ
  - `malloc()` で割り当てられる
  - `free()` で解放される
- スタック



2.8　コンピュータ・ハードウェア内での手続きのサポート

- 補足: 末尾呼び出し (tail call) の最適化

```
int sum(int n, int acc) {
  if (n > 0) return sum(n - 1, acc + n);
  else return acc;
}
```

```
sum:      slti $t0, $a0, 1           # n ≦ 0 かチェック
          bne  $t0, $zero, sum_exit  # n ≦ 0 なら sum_exit へ分岐
          add  $a1, $a0              # n を acc に累計
          addi $a0, $a0, -1          # n を 1 繰り下げ
          j    sum                   # sum にジャンプ
sum_exit: add  $v0, $a1, $zero       # 戻り値 acc を設定
          jr   $ra                   # 呼び出し元に戻る
```

2.9　人との情報交換

- 文字の表し方: ASCII コード

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	

2.9　人との情報交換

- 文字列 (string)
- 文字列の長さを示す方法
  - 最初の文字を文字列の長さにする
    - Java ではこれ。
  - データ構造として付随する変数に長さを持たせる
  - 最後の文字を終端を表す特定の文字にする
    - C 言語ではこれ。値がゼロのバイト (null 文字) で表す。

余談: C 言語ではこれのせいで `strlen` に $O(|S|)$ かかる (定数倍高速化されている)

2.9　人との情報交換

- 現在では Unicode を使用
  - 世界のあらゆる文字体系をコード化したもの
- Java 内部では 1 文字が 16 ビットで表されている
  - UTF-16 という符号化方式を使用
- MIPS で 8 ビットや 16 ビットを読み書きしたい場合
  - 1 バイト読み書き: `lb` (load byte), `lbu` (load byte unsigned), `sb` (store byte)
  - 2 バイト読み書き: `lh` (load half), `lhu` (load half unsigned), `sh` (store half)

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- 32 ビットの定数やアドレスを扱いたい
- `lui` (load upper immediate)
  - 16 ビットの即値定数フィールドの値をレジスタの上位 16 ビットに入れ、残りの下位 16 ビットをゼロで埋める
  - 16 ビットを超える定数がほしいときに `ori` と組み合わせて使用

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- ジャンプ命令 `j 10000` (J 形式)

2	10000
6 ビット	26 ビット

- 一方、分岐命令 `bne $s0, $`

5	16	17	Exit
6 ビット	5 ビット	5 ビット	16 ビット

- アドレスが 16 ビットなのは今日では小さすぎる

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- 分岐先はだいたい分岐命令の近くにある

$$\mathrm{PC} \leftarrow (\mathrm{PC} + 4) + \text{定数}$$

- この形式を **PC 相対アドレシング** という
  - **「一般的な場合を高速化する」**
  - ただし jump と jump-and-link の場合は呼び出す対象の手続きが呼び出し元の近くにあるとは限らないので、J 形式により長いアドレスを指定する
- さらに命令はすべて 4 バイトなので、
  MIPS ではバイト数ではなく語数を指定する
  - 2 ビットお得

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- 例: 以下をアセンブルせよ。

```
Loop: sll  $t1, $s3, 2      # $t1 = i << 2
      add  $t1, $t1, $s6    # $t1 = (save[i] のアドレス)
      lw   $t0, 0($t1)      # $t0 = save[i]
      bne  $t0, $s5, Exit   # save[i] ≠ k なら Exit へ
      addi $s3, $s3, 1      # i = i + 1
      j    Loop             # Loop へ
Exit:
```

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- 結果

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	…					

  - 20000 × 4 = 80000 にジャンプ

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- MIPS のアドレシング・モードのまとめ
  - **即値アドレシング**
    - 命令中の定数
  - **レジスタ・アドレシング**
    - レジスタの値
  - **ベース相対アドレシング**
    - 命令中の定数とレジスタの値の和
  - **PC 相対アドレシング**
    - PC と命令中の定数の和
  - **疑似直接アドレシング**
    - PC の上位 4 ビットと命令中の 26 ビットを連結

2.10　32 ビットの即値およびアドレスに対する MIPS のアドレシング方式

- 例: 以下の機械語をアセンブリ言語に変換せよ。

```
00af8020
```

- 結果

```
0000 0000 1010 1111 1000 0000 0010 0000
```

```
op      rs     rt     rd     shamt  funct
000000  00101  01111  10000  00000  100000
```

```
add $s0, $a1, $t7
```

以上です

余談: Compiler Explorer

- https://godbolt.org/
- 左の言語を C に、右のコンパイラを MIPS gcc 5.7 に設定
- 適当にプログラムを入れるとコンパイル結果が表示される

演習問題

演習問題

以下を解きましょう

- 2.27
  - ループのコンパイル
- 2.31
  - 関数のコンパイル

演習問題 2.27

下記の C コードを MIPS のアセンブリ・コードに翻訳せよ。

```
for (i = 0; i < a; i++)
  for (j = 0; j < b; j++)
    D[4 * j] = i + j;
```

変数 `a`, `b`, `i`, `j` はレジスタ `$s0`, `$s1`, `$t0`, `$t1`

配列 `D` のベースアドレスはレジスタ `$s2`

補足: 変数は多分すべて `int`

演習問題 2.31

下記の C コードを MIPS のアセンブリ・コードに翻訳せよ。

```
int fib(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n - 1) + fib(n - 2);
}
```