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 □□□□

- □□□□□□□ □□ (instruction) □□□□
- □□□□□□□□□□ □□□□□ (instruction set) □□□
- □□□□□□□□□□□□
  - MIPS
  - ARMv8
  - Intel x86
  - ARMv7

□□ 1: □□ ARMv9 □□□

□□ 2: □□□□□□□□□ RISC-V □□□□

2.1 □□□□

- □□□□□□□□□□□□□□□□
- □□□□□□□□□□□□
  - □□□□□□□□□□□□□□□□□□□□□□□□□□
  - □□□□□□□□□
  - □□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  - □□□□□□□□□□ □□□

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 □□□□□□□□□□□□□□□□□□

□□

- □□□ C □□□□□□□□ MIPS □□□□□□□□□□□

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

- □□

```
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 □□□□□□□□□□□□□□□□□□□□□□

- □□□□□
  - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  - □□□□□□□□□□ 8 □□□□ □□□ □□□□□□□
  - MIPS □□□□□□□□□ 4 □□□□□□□□□□□□□
    - □□□□□□□□□□ □□□□□
  - □□□□□□□□□□□□□□ □□□□□□□□□□ □
    □□□□□□□□□□ □□□□MIPS □□□□□□□□□□□
  - □□□□□□□□□□□□□□ 8 □□□□□□□
    □□□□ $4 \times 8 = 32$ □□□□□□□□□□□

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 □□□□□□□□□□□□□□□□□□□□

- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□ □□□□□□□ (spilling) □□□
- □□□□
  - □□□□□□□□
  - □□□□□□□□□□□□□□□□□□□□
- □□□
  - □□□□□□□□□□□□
  - □□□□□□□□□□□□□□□□□□□□□□□□

2.3 コンピュータ・ハードウェアのオペランド

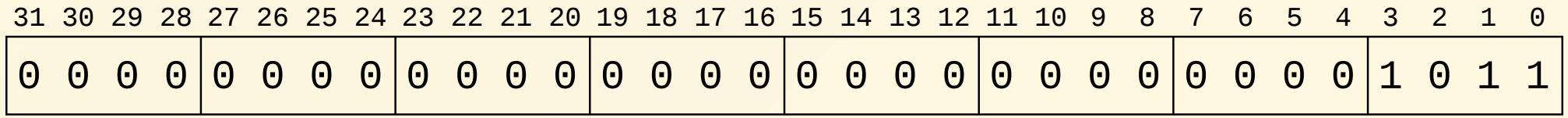
- 命令操作に定数を使うことは多い
- メモリに定数を置いてそれをロードするのではなく、

  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

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

- 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 □□□□□□□□□□□□

- □□□□□□□□□□
- □□□□□□□□□□
  - 1 □□□□□□□□□□ (□□□□□□□□□□)
- □□□□□□□□□□□□□□
  - □□□□□□□□□□□□□□
  - □□□□□□□□□□□□□□□□
  - □□□□□□□□ 2 □□□□□□□□□□

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

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

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 [illegible]

- [illegible]
  - [illegible] 1 [illegible]
- [illegible]: 2_{10} [illegible]

```
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} [illegible]

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 □□□□□□□□□□□

- □□□□□□□□
- □: MIPS □□□□□ 32 □□□□□□□□□□□□□□□
- `lb` (load byte)
  - □□□□□□□□□□□□□□□□
  - 24 □□□□□□□□□□
- `lbu` (load byte unsigned)
  - □□□□□□□□□□□□□□□□
  - 24 □□□□ 0 □□□□

2.4 □□□□□□□□□□□□

- □□: □□□□□□□□□□□□□□□□□□
  - □□□□□□□□□□
  - 2 □□□
  - 1 □□□
  - □□□□□□□

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 □□□□□□□□□□□□□□□□

- □□: □□□□□□□□□□□□□

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

□□□□□□□□□□□□□□□□□□□□□

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

□□□□□□□□□□□

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 □□□□

MIPS □□□□□ ②

- `and` (and)
- `andi` (and immediate)
- `or` (or)
- `ori` (or immediate)
- AND □ OR □□□□□□□□□□□□□□□□□□□
  - □□□□□□□□

□□: `xor` □ `xori` □□□

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 □□□□□□□□

- `beq` (branch if equal)

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

  - □□□□ 1 □□□□□□□ 2 □□□□□□□□□
    □□□ `L1` □□□□□□□□□□□□

- `bne` (branch if not equal)

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

  - □□□□ 1 □□□□□□□ 2 □□□□□□□□□□
    □□□ `L1` □□□□□□□□□□□□

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 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□ (procedure)
  - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
- □□□□□□□□□□□□
  1. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  2. □□□□□□□□□□
  3. □□□□□□□□□□□□□□□□□□□□
  4. □□□□□□□□□□□□□
  5. □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
  6. □□□□□□□□□□□□
     - □□□

2.8 [illegible]

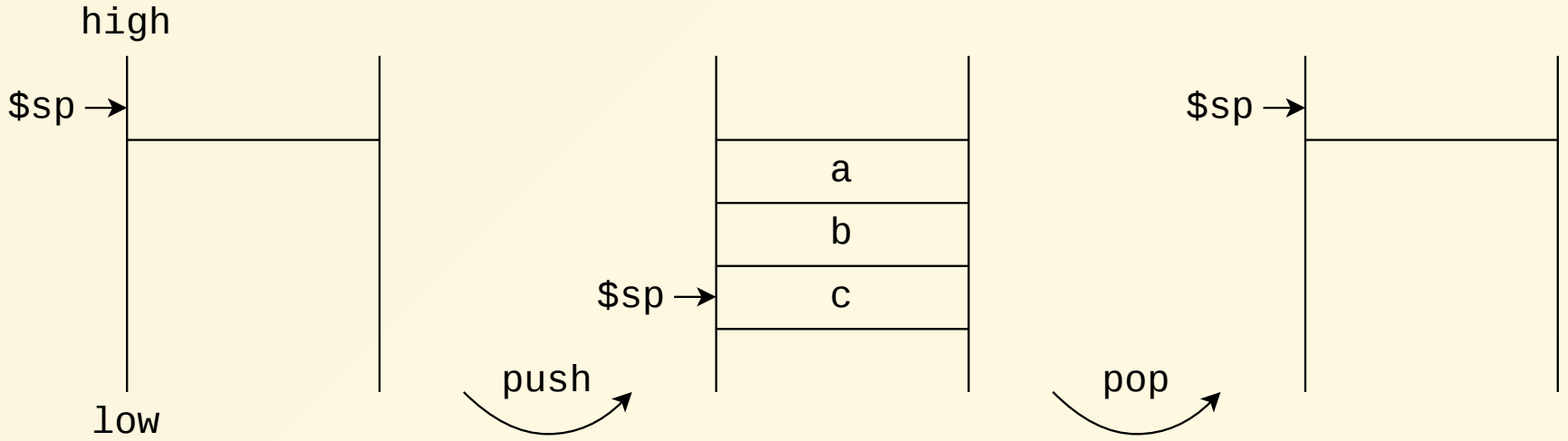
- [illegible]
  - `$a0-$a3`: [illegible]
  - `$v0-$v1`: [illegible]
  - `$ra`: [illegible]
- `jal` (jump-and-link)
  - `jal [illegible]` [illegible] [illegible] (PC + 4) [illegible] `$ra` [illegible]
    - PC: [illegible]
- `jr` (jump register)
  - `jr $ra` [illegible]

2.8 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□
  - □□□□□ (caller)
  - □□□□□□ (callee)
- □□□□□□□□□□□□□□□
  - □□□□□□□□□□□□□ `$a0-$a3` □□□□
  - `jal X` □□□□ X (□□□□□□□) □□□□□
  - □□□□□□□□□□□□□
  - □□□ `$v0-$v1` □□□□
  - `jr $ra` □□□□□□□□□□□□□

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 □□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
leaf_example:
    addi $sp, $sp, -12      # □□□□□ 3 □□□□
    sw   $t1, 8($sp)        # $t1 □□□
    sw   $t0, 4($sp)        # $t0 □□□
    sw   $s0, 0($sp)        # $s0 □□□
    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, $zero    # f □□□ ($v0 = $s0 + 0)
    lw   $s0, 0($sp)        # $s0 □□□
    lw   $t0, 4($sp)        # $t0 □□□
    lw   $t1, 8($sp)        # $t1 □□□
    addi $sp, $sp, 12       # □□□□□□ 3 □□□□
    jr   $ra                # □□□□□□□□
```

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 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□□□□□□□□□□□□□□□ □□□ (leaf) □□□□□□
- □□□□□□□□□□□□□□□□□
- □: □□□□□□□□□□□□□□□□□□□

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

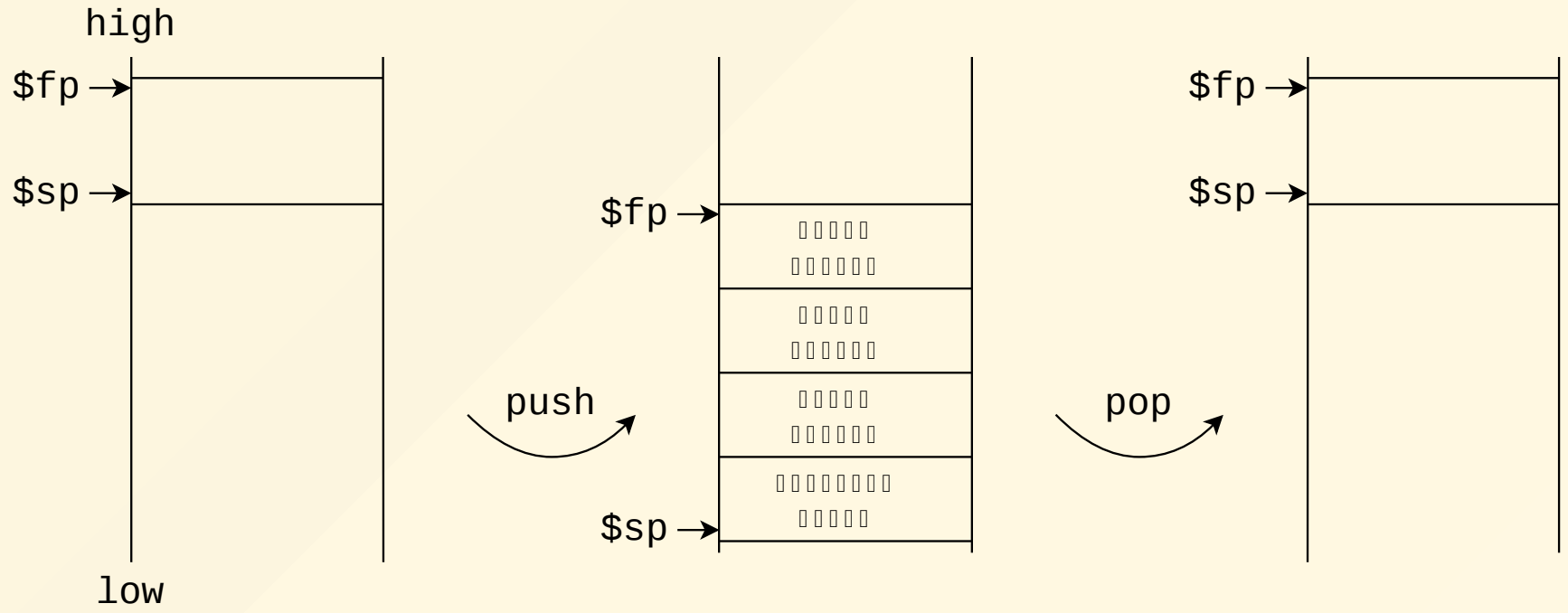
  - □□□□□□□□□□□□□□□□□□□□□□□□□
  - □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

2.8 □□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
fact:
    addi $sp, $sp, -8     # □□□□□ 2 □□□□
    sw   $ra, 4($sp)      # □□□□□□□□□□
    sw   $a0, 0($sp)      # □□ n □□□
    slti $t0, $a0, 1      # n < 1 □□□□□□
    beq  $t0, $zero, L1   # n ≧ 1 □□□□ L1 □□□
    addi $v0, $zero, 1    # 1 □□□
    addi $sp, $sp, 8      # □□□□□□ 2 □□□□
    jr   $ra              # □□□□□□□□
L1: addi $a0, $a0, -1     # n ≧ 1 □□□□□□□ (n - 1) □□□
    jal  fact             # (n - 1) □□□□ fact □□□□□
    lw   $a0, 0($sp)      # □□ n □□□
    lw   $ra, 4($sp)      # □□□□□□□□□□
    addi $sp, $sp, 8      # □□□□□□ 2 □□□□
    mul  $v0, $a0, $v0    # n * fact(n - 1) □□□
    jr   $ra              # □□□□□□□□
```

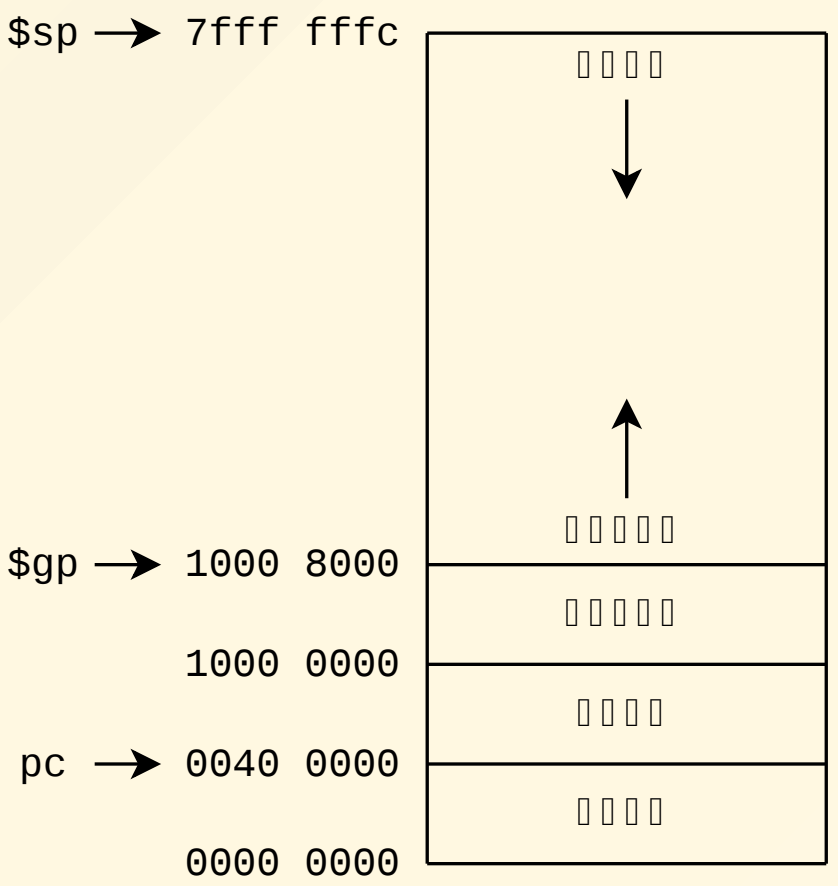
2.8 コンピュータ・ハードウェア内での手続きのサポート

- 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 □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

- □□□□□□□□□□□□
  - □□□□□□□
- □□□□□□□□□□□□□□
  - □□□□□□□□□□□□
- □□□□
  - `malloc()` □□□□□□□□□
  - `free()` □□□□□□□
- □□□□□



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 □□□□□□□

- □□□□□□: ASCII □□□

	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 MIPS

- 32
- `lui` (load upper immediate)
  - 16 16
    16
  - 16 `ori`

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

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

- 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

<table>
<tr><td>80000</td><td>0</td><td>0</td><td>19</td><td>9</td><td>2</td><td>0</td></tr>
<tr><td>80004</td><td>0</td><td>9</td><td>22</td><td>9</td><td>0</td><td>32</td></tr>
<tr><td>80008</td><td>35</td><td>9</td><td>8</td><td colspan="3">0</td></tr>
<tr><td>80012</td><td>5</td><td>8</td><td>21</td><td colspan="3">2</td></tr>
<tr><td>80016</td><td>8</td><td>19</td><td>19</td><td colspan="3">1</td></tr>
<tr><td>80020</td><td>2</td><td colspan="5">20000</td></tr>
<tr><td>80024</td><td colspan="6">...</td></tr>
</table>

  - 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

□□□□

□□□□

□□□□□□□□□□

- 2.27
  - □□□□□□□□□□
- 2.31
  - □□□□□□□□□

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`

□□□□ 2.31

□□□ C □□□□ MIPS □□□□□□□□□□□□□□□□□

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