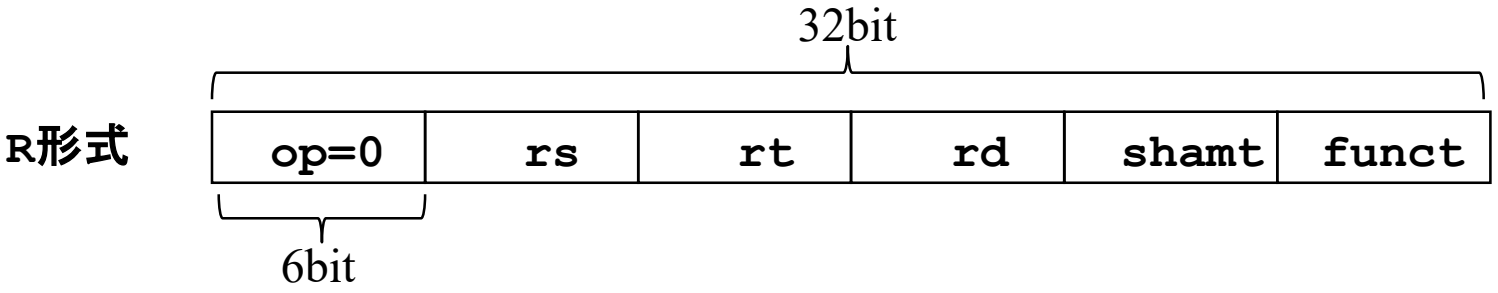
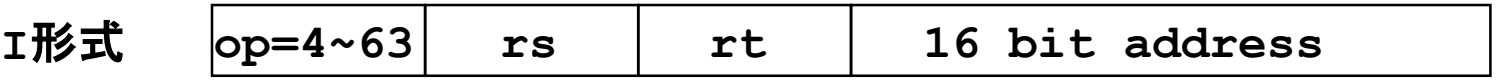
第3回目
2018/12/07
「アセンブリ言語と機械語の続き」

MIPSの機械語の命令形式

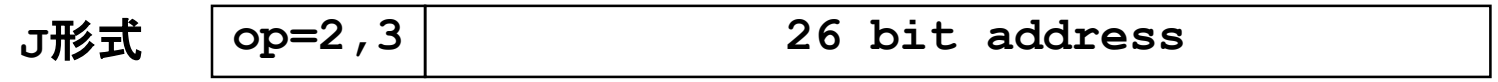
- どの命令も32bit長で一定 (RISCタイプの特徴)



例: add (op=0, funct=0x20), sub (op=0, funct=0x22), and (op=0, op=0x24),
sll (op=0, funct=0x00), jr (op=0, funct=0x08) …



例: lw (op=0x23), sw (op=0x2b), lb (op=0x20), sb (op=0x28),
addi (op=0x08), andi (op=0x0c), lui (op=0x0f) …



例: j (op=0x02), jal (op=0x03)

レジスタの使用に関する慣例

MIPSは32本のレジスタを持つ
しかし、一部は慣例的に(コンパイラ・ローダにより)決められた役割を持つ
(ハードウェア的には$zeroと$ra以外は特殊扱いされない)

Name	Register number	Usage
$zero	0	the constant value 0 - cannot be changed
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments to functions
$t0-$t7	8-15	function temporaries
$s0-$s7	16-23	function temporaries (saved)
$t8-$t9	24-25	function temporaries
$k0-$k1	26-27	reserved for OS kernel
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (for jal instruction)

この他に、PC (program counter)=今実行中の命令アドレス

MIPS以外では？ISAに依存するが、コアあたりレジスタ数十本

Move: レジスタ間の値の移動

- 例: int a = b; を実行したいとき
- レジスタ $s0の値を$s1に移動

```
move $s1, $s0
```

- MIPS機械語にはmove命令が存在しない
- 0番のレジスタは、必ず0が入っていることを利用 ($zero)

→ 他の命令で代用

```
add  $s1, $s0, $zero
```

- 前回のbltやmoveのように、アセンブラでは用意されているが、実際には他の命令に変換される命令を pseudo instruction（疑似命令）と呼ぶ

定数値の表現法

- 小さい値の定数は頻繁に用いられる (50% of all operands)

  e.g., A = A + 5;
  B = A + 1;
  C = C - 18;

- ありうる解決法

  ?? 典型的な定数を表としてメモリに入れておいて、ロードする。

  ?? $zero同様、決まった定数が常に格納されているレジスタを用意

  – Q: これらは実際には用いられないか、無駄が多い。なぜ?

- MIPSの解決法: immediate (イミィディエート、即値)を用いた命令

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

1ワードの命令の中に定数値を埋め込む

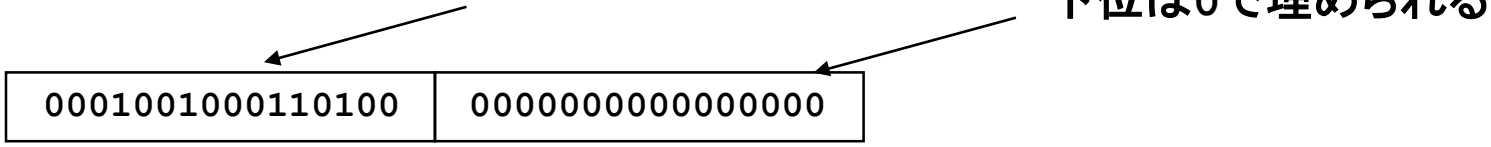
→ I形式。即値は16bitで表せる範囲(−32768～32767)

- Q: subi命令はない。C=C-18; を実現したいときは？
- Q: A=5; を実現したいときは?

大きい定数はどうする?

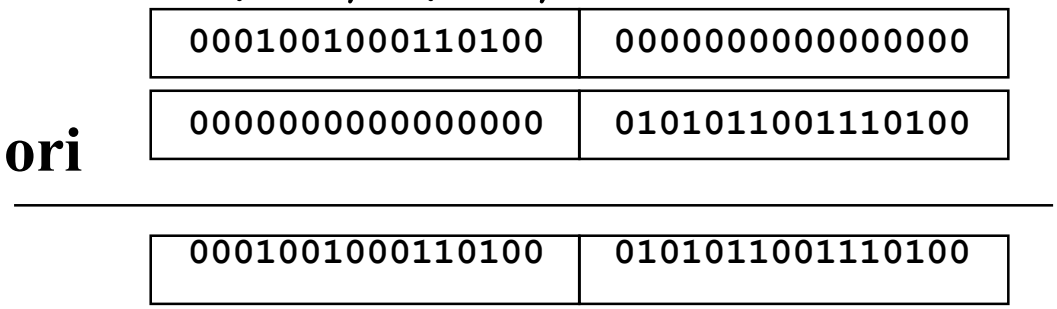
- A=0x12345678; を行いたい場合？
- 32-bitの定数をレジスタにロードしたい
- 新しい命令を使わなくてはならない: “load upper immediate” 命令

```
lui $t0, 0x1234
```



- そして、下位のビットを設定しなくてはならない

```
ori $t0, $t0, 0x5678
```



分岐命令におけるアドレスの指定法

- 命令:

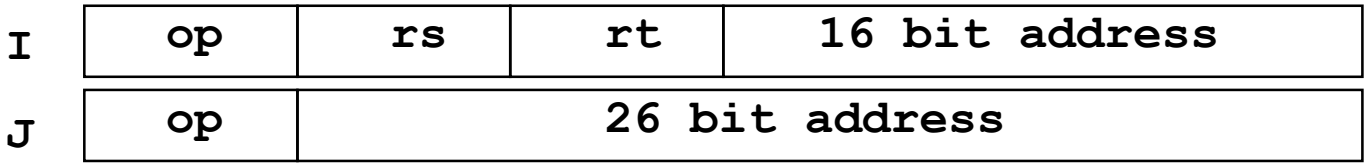
  `bne $t4,$t5,Label` もし $t4≠$t5なら Labelに分岐・移動

  `beq $t4,$t5,Label` もし $t4=$t5なら Labelに分岐・移動

  `j Label` Labelのアドレスに無条件でジャンプ

  (注:実際のLabelのアドレスはアセンブラ等が自動計算する)

- MIPSで即値を使える命令フォーマット:



- 注意:I形式でもJ形式でも即値は 32 ビット未満である

  プログラムのサイズは必ず2^{16}バイト以下、というのはさすがにひどい

  → どのようにしてこの不足を補うのか?

無条件ジャンプ命令の場合

- 無条件ジャンプ命令はJ形式→ 2^{26}までの即値を用いることができる

J形式

op	26 bit address

- さらに、「有効な命令アドレスは4の倍数」という事実も利用

→ 26bit即値×4を、ジャンプ先アドレスとする

→ 2^{28}Byte=256 MBのアドレス指定の範囲が可能

- どうしても32ビット全体(4GB)にジャンプしたい場合
  - 任意のレジスタの値をアドレス→ レジスタ間接ジャンプ

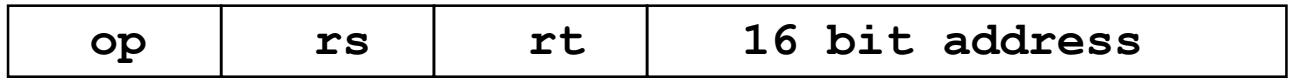
```
jr $t0      #t0レジスタの中身のアドレスにジャンプ
```

条件分岐命令の場合

- 命令:

```
bne $t4,$t5,Label      もし $t4≠$t5なら Labelに分岐・移動
beq $t4,$t5,Label      もし $t4=$t5なら Labelに分岐・移動
```

- 命令フォーマット:
  - レジスタ2つ指定する必要があるので、I形式である必要



- 前ページ同様4倍テクは使えるが、2^{18}バイトまでのプログラムしか作れない？

MIPSの解決法： lw, swと同様に、(レジスタ＋即値)の考え方を用いる

  - ただし、現在実行中の命令のアドレスを示す特殊レジスタである PC(Program Counter)を基準 → *PC相対アドレッシング*
  - 現在の命令を基準に -131072～+131068バイトへ飛べる
  - これでよいか？ → ほとんどのブランチは局所的である(局所性の原則)

Q： もっと遠くへ条件分岐したいときは？ hint: j命令と組合わせ

今までのMIPS命令の一覧:

MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. Register $at is reserved for the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], ..., Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential words differ by 4. Memory holds data structures, such as arrays, and spilled registers, such as those saved on procedure calls.

MIPS assembly language

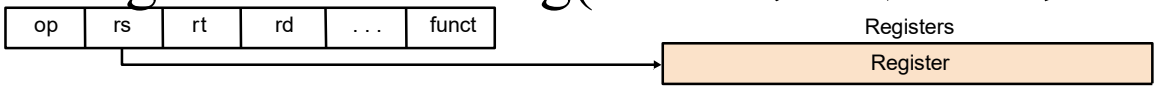
Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

MIPSのアドレッシングモードのまとめ

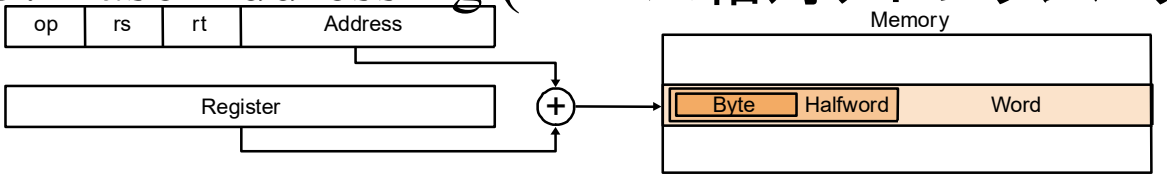
アドレッシングモード：演算・操作対象の「値」をどう指定するか

1. Immediate Addressing(即値アドレッシング)

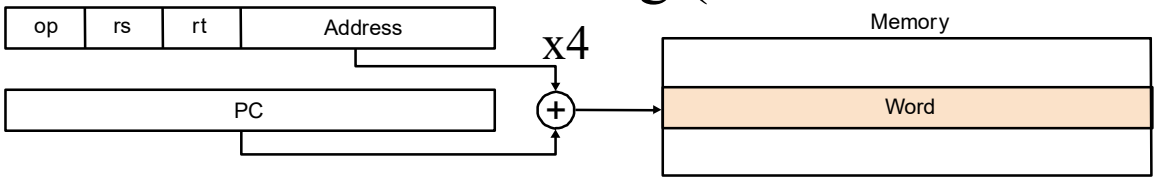
2. Register Addressing(レジスターアドレッシング)



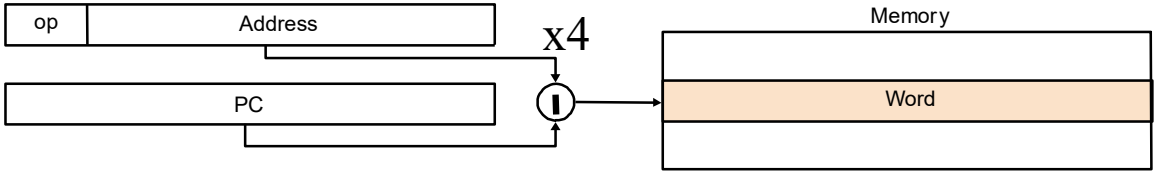
3. Base Addressing (ベース相対アドレッシング)



4. PC-Relative Addressing (PC相対アドレッシング)



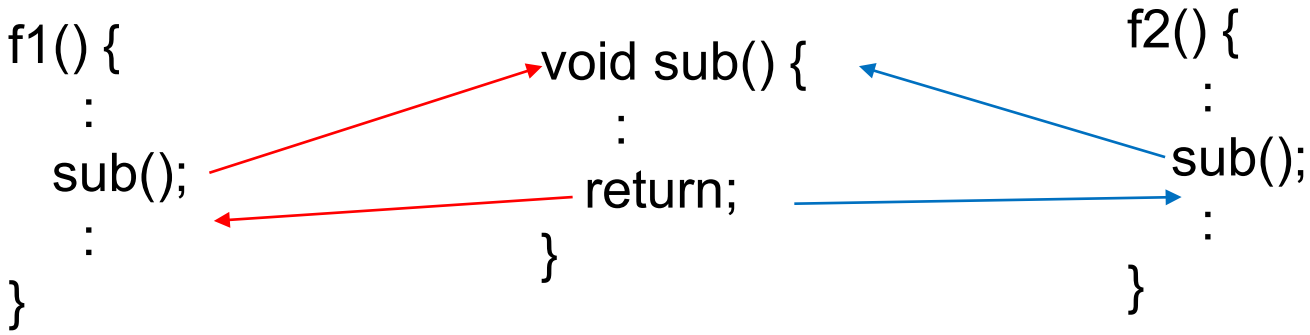
5. Pseudodirect Addressing (疑似直接アドレッシング)



サブルーチン・関数呼び出し

- サブルーチン・関数呼び出しは、プログラム言語においてもっとも重要な処理
- だが、プロセッサにとっては高度な処理！
- 機械語には、直接サブルーチンの概念はない

正しい呼び出し元へ帰る必要



再帰呼び出しにも対応する必要

```
int fact(int n) {
   if (n <= 1) return 1;
   return n * fact(n-1);
}
```

今何回目の呼び出し?

呼び出しごとに違うnでなければならない
(局所変数の問題)

ほかにも、引数・返り値の渡し方、など

サブルーチンの実現についての議論

- これまで学んだジャンプ・分岐命令だけでは実現できない
- 解決のためのデザインの選択肢は? あまり良くない例:
  - 戻りアドレスをサブルーチンの頭にストア
    - どのようにPCの値を得るかの問題
  - サブルーチン毎に固定のメモリのある領域に引数を書き込んで渡す?
  - サブルーチン毎に固定のメモリのある領域を局所変数として割り当てる?

  [Q] 以上がうまくいかない理由を考えよう。Hint 再帰呼び出しの場合

サブルーチンのMIPS ISAにおける実現：基本方針

- 一部のレジスタを、サブルーチン実現のために役割固定

$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address (for jal instruction)

  - 番外の「PC (program counter)=今実行中の命令アドレス」も大事
- スタックの活用
  - 後入れ先出しのデータ構造：サブルーチン実現専用のスタックが1つある
  - これで再帰呼び出しもok
  - スタックは、上位アドレス(大)から下位アドレス(小)に伸びる
  - MIPSでは、専用のレジスタ $sp (29)がスタックポインタ
- 専用命令(jal)あり

戻りアドレスの実現

- MIPSには jal (jump and link)命令がある

  `jal Label  #$ra (31)にPC+4(自命令の次)を格納し、Labelへ分岐`

- `jal`を、サブルーチン「呼び出し」に使う
- `jr $ra` を、サブルーチンからの「return」に使う

→ 正しい呼び出し元へ帰ることができる！

?? でもまだこれだけでは、多段呼び出しができない。一つしかない$raをかきつぶしてしまう → スタックの活用

多段呼び出しの実現

サブルーチンAからBを呼び出し、BからCを呼び出す例

- プログラム開始時に、一定のメモリ領域をスタックとしておき、その最大アドレスを$spに代入

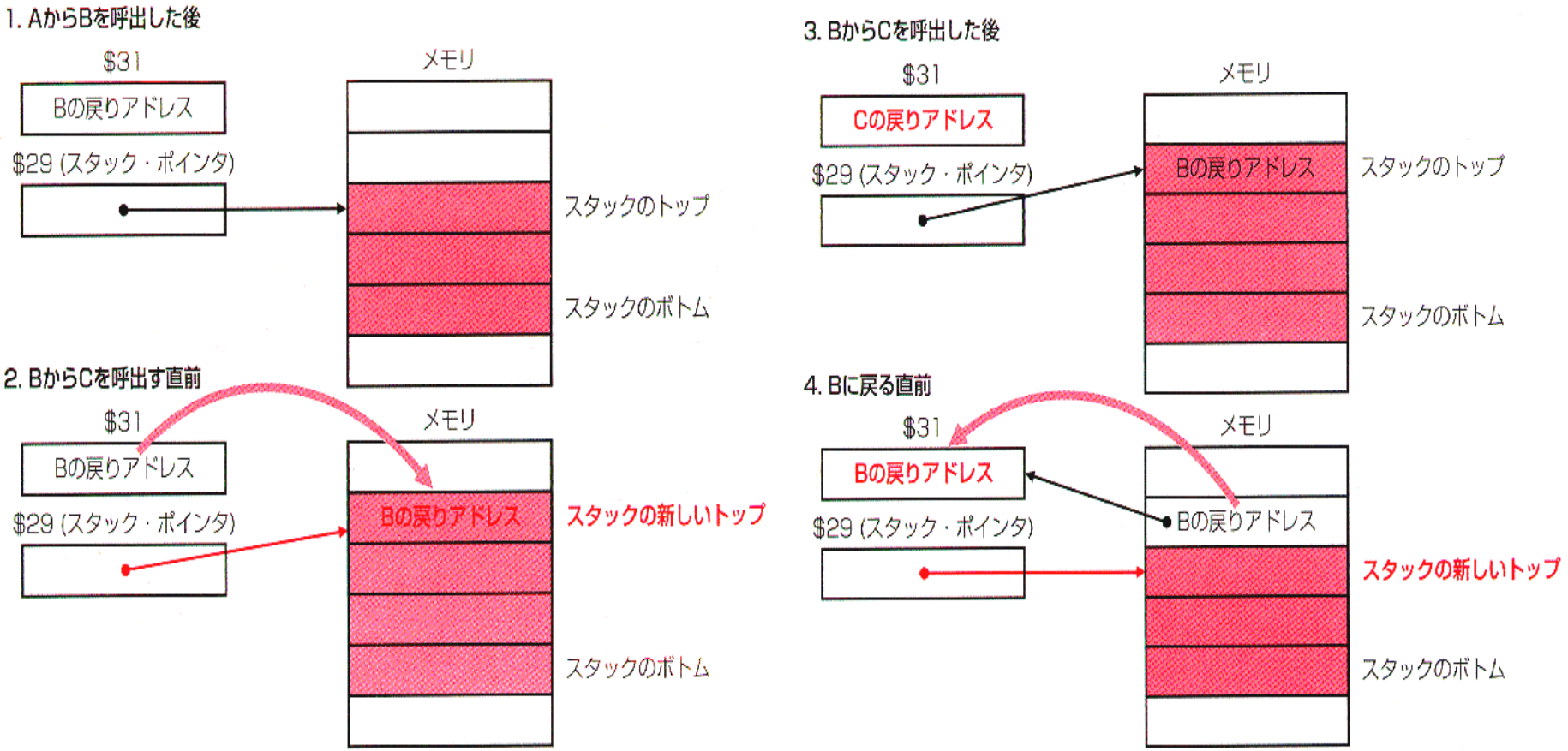
```
A: ...
   jal B              #サブルーチンB呼び出し
   ...

B: subi $sp, $sp, 32   #スタックを伸ばす
   sw   $ra, 0($sp)    #戻り番地$raを待避
    ...
   jal  C               #サブルーチンC呼び出し
   ...
   lw   $ra, 0($sp)    #戻り番地を復元
   addi $sp, $sp, 32   #スタックを縮める
   jr   $ra            #サブルーチンBから戻る

C: ...
   jr   $ra
```

- 注
  - 実際は、MIPSでは戻り番地は20($sp)に格納するルール
  - スタックサイズは有限 → 際限なく多段呼び出しを続けると実行失敗

サブルーチンの実現(3)


1. AからBを呼出した後
$31
Bの戻りアドレス
$29 (スタック・ポインタ)
メモリ
スタックのトップ
スタックのボトム
2. BからCを呼出す直前
$31
Bの戻りアドレス
$29 (スタック・ポインタ)
メモリ
Bの戻りアドレス
スタックの新しいトップ
スタックのボトム
3. BからCを呼出した後
$31
Cの戻りアドレス
$29 (スタック・ポインタ)
メモリ
Bの戻りアドレス
スタックのトップ
スタックのボトム
4. Bに戻る直前
$31
Bの戻りアドレス
$29 (スタック・ポインタ)
メモリ
Bの戻りアドレス
スタックの新しいトップ
スタックのボトム

引数・返り値・局所変数

- 引数・返り値の、MIPSでの標準の規約(convention)
  - レジスタ$a0-$a3 (4-7) が引数、$v0, $v1 (1,2)が戻り値
  - しかしそれだけでは、多段呼び出し時にレジスタを書きつぶしてしまう → これもスタックにしまう
  - 引数が4個を超えるときもスタックを利用
- 局所変数は？
  - これもスタック上のメモリを使える
  - コンパイラ最適化によりレジスタが使えることはもちろんある

→ スタック上には、返りアドレス・引数・局所変数などがしまわれる

→ 各サブルーチン開始時に、必要なサイズ(コンパイラが計算)だけスタックが伸びる。これを**スタックフレーム**と呼ぶ

- コンパイラによる局所変数の扱い
  - 自動的にレジスタとフレーム中のメモリに割り付ける
  - 数々のアルゴリズム→人間が行うより良い

Caller saveとcallee save (やや難)

- 用語：あるサブルーチン呼び出しがあるとき、
  Caller = 呼び出す側、Callee = 呼び出される側
- $raは呼び出される側(callee)でスタックに保存(save)し、壊れないようにしていた
  → callee saveと呼ぶ
- どのレジスタがcaller saveかcallee saveかも、規約で決まっている
  - Caller save: $t0-$t9, $a0-$a3(引数用レジスタ), $v0-$v1 (返り値用レジスタ)
    呼び出し側は、jalの前と後でこれらレジスタの値が変わっていると想定して動作しなければならない
  - Callee save: $s0-$s7, $sp, $ra など

[Q] すべてのレジスタがcaller saveもしくはcallee saveだと弊害はあるか？

- ここまで述べた規約はあくまで規約であり、プログラミング言語を自作する際には必ずしも守る必要はない
- ただし、標準ライブラリ(printf()など)は標準規約に則っているので注意

「代わりの」アーキテクチャデザイン

- 「僕の考える最強のISA?」
  - よくある案：より強力かつ複雑な命令を提供する → 命令数削減
  - 複雑な命令の導入により、クロックサイクルを高くできない・CPI (Clocks per Instruction, 命令あたりの平均クロック数)が増加、などを招く
- “RISC 対 CISC”論争の歴史
  - 1982年以来のほとんどの新しい命令セットはRISCか、その影響を強く受けている（例：ARM, RISC-V)
  - CISCの代表：DEC VAX-11: 命令を強力に、アセンブラを簡単に *1バイトから54バイト長の命令まで!*
  - CISCを超えたVLIW (very long instruction word)という動きもあった(IA-64など)が、あまり成功せず
- Power系ISA (RISC)とx86 ISA(CISC)を簡単に見てみよう

Power系のCPUの利用例


PowerMac

Xbox360

PlayStation3
(Cell BE CPU)

Summit supercomputer
2018現在世界一

Power系ISAの特徴

- RISC: 命令長は32bit固定
- インデックスアドレッシング (Indexed addressing)
  - 例:　`lw $t1,$a0+$s3  #$t1=Memory[$a0+$s3]`
  - Q:どのような場合に便利？ MIPSの場合、どのような操作に相当？
- アップデートアドレシング (Update addressing)
  - ロード命令の動作の一部として、レジスタを更新(配列の逐次アクセスなど)
  - 例: `lwu $t0,4($s3) #$t0=Memory[$s3+4];$s3=$s3+4`
  - Q: MIPSの場合はどのような操作に相当?
- その他:
  - load multiple/store multiple(複数のレジスタを一度にload/store)
  - 特殊なループ用のカウンタレジスタ“`bc Loop`”

decrement counter, if not 0 goto loop
カウンタを減らし、0でなければloopに分岐

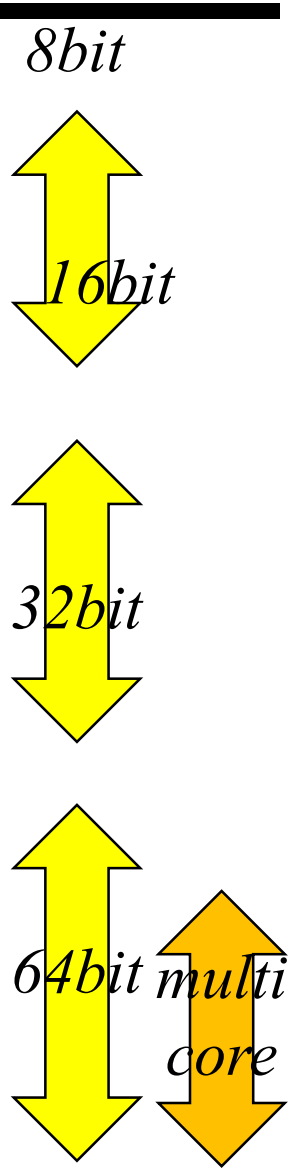
x86系のISAの進化過程

- (1974: Intel 8080 (8 bit アーキテクチャ) 2Mhz)

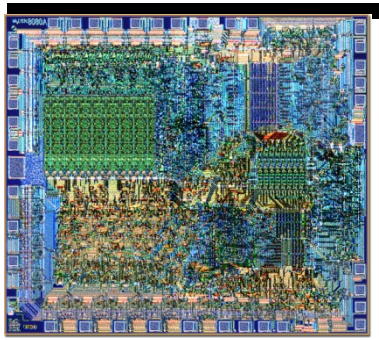
- 1978: Intel 8086 が発表
  - CISC, 命令長1~5バイト
  - 8080と完全な互換性はない
  - 4本の算術系レジスタ(機能異なる)、4本のインデクスレジスタ
- 1980: 8087 浮動小数点コプロセッサが追加
- 1982: 80286 はアドレス空間を24ビットに拡張、命令の追加、保護モード

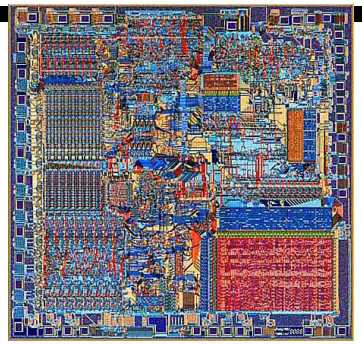
- 1985: 80386 は 32 bitアーキテクチャに, 新たなアドレッシングモード、8本の汎用レジスタ
- 1989-1995: 80486, Pentium, Pentium Pro は若干の命令追加、大幅なマイクロアーキテクチャ改変
  (ほとんど同じISAのまま、大幅な性能向上：16Mhz→200Mhz)
- 1997: Pentium MMX (MultiMedia Instruction Set)の追加, Pentium II 266Mhz
- 1998: AMD K6/2 3DNow! (SIMDベクトル拡張命令の追加)
- 1999: Pentium III SSE (Streaming SIMD Extensions) 600Mhz

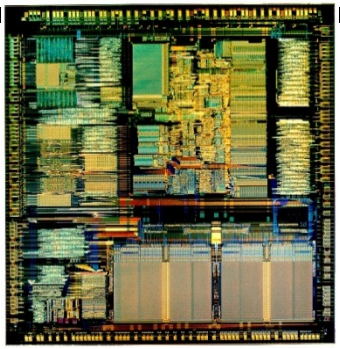
- 2003: AMD Opteron/Athlon64 AMD64 (64bit命令体系への拡張、レジスタの倍増)
- 2004: AMD64/IA32-e の融合(x86_64 = x64 = EM64T=AMD64+SSE3 ≠ IA64)
- 2005: Intel Pentium D (2コア)発売、AMD 2コアOpteron発売
- 2007: AMD Phenom (4コア)発売
- 2011: Intel Sandy Bridge世代 でAVX (Advanced Vector Extensions) +暗号化サポート命令
- 2013: Intel Xeon Phi (Knights Corner)発売
- 2016: Intel Xeon Phi (Knights Landing, 64-72コア)発売

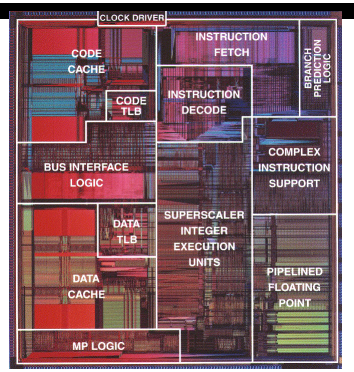


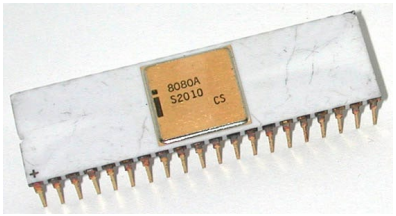
Intel プロセッサの系譜の画像

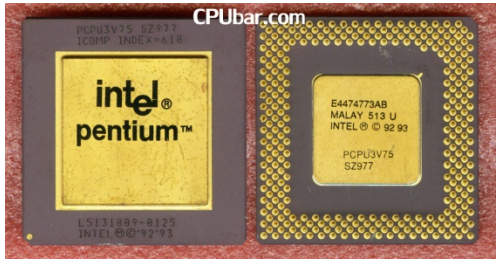


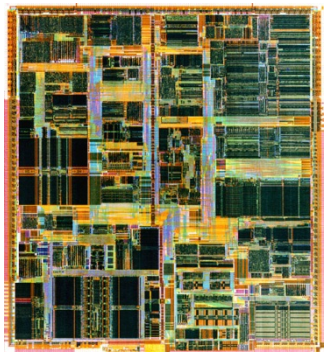


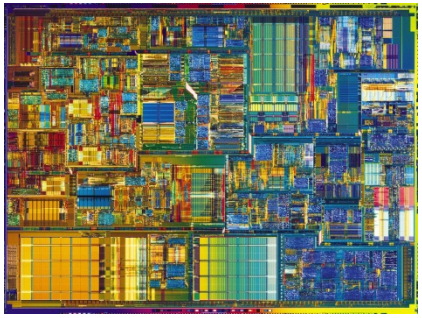


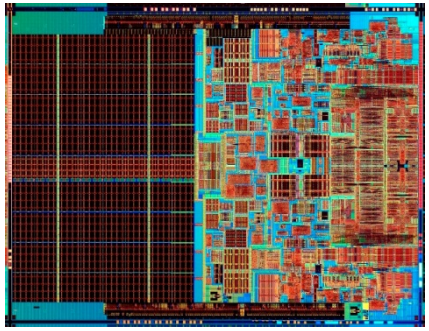


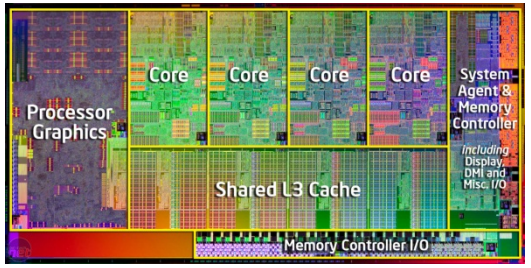


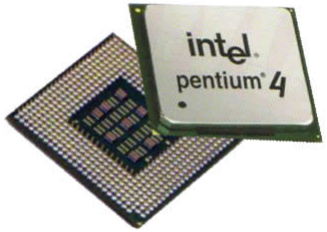


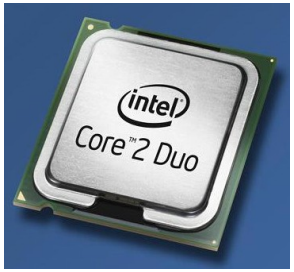


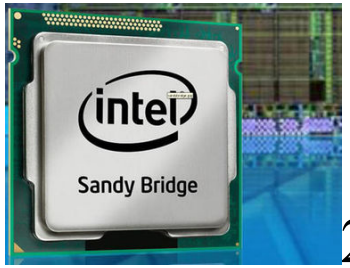












x86/x86_64 ISAの特徴

- CISC: 命令長は1～15バイト
- より多機能(複雑)な命令
  - 算術演算などでも、一つのオペランドは直接メモリを参照可能
    例: `add dword ptr[ebp+8], eax`
  - 同じオペランドがソースとデスティネーションに同時になりうる
  - 複雑なアドレッシングモード
    例: scaled indexed addressing

```
mov dword ptr[rdi+2*rax], eax
```

- どうして世の中はそれでもうまく動いているか
  - 頻繁に良く実行される命令はあまり複雑ではない→RISCと同等の複雑さ、実行効率
  - コンパイラがアーキテクチャの遅い部分を避けている
  - 1998ごろ以降のIntel/AMDプロセッサは、CISC命令を内部的にRISCのような命令(μOpと呼ばれる)に変換
    - コアはRISCプロセッサ (それ以前は内部もCISC)
    - レジスタリネーミング、正確なブランチ予測、μOp融合など、高度技術の総動員

アセンブリ・機械語のまとめ

- 命令セットアーキテクチャ (Instruction Set Architecture = ISA)
  - もっとも抽象度が低位な機械の言語を定義
  - 上位のプログラムと、ハードウェアのインターフェースとなる重要な抽象化 -「30年前のプログラムも動く」
  - RISC vs. CISC – 今は「良いとこどり」
- RISCであるMIPSを題材にした
  - 限定された種類のアセンブリ命令とアドレッシングモード
  - ビットパターン(MIPSは32bit固定)に埋め込むための制限も
  - 単純な命令を組み合わせて、より高度なプログラム言語の機能を実現 → サブルーチン呼び出しを取り上げた
    - 他の機能は？オブジェクト指向、自動メモリ管理・・・
- 機械語の複雑さと速度性能は相反することが多い
  - x86は例外？CISCでありつつ、高度な技術により性能向上を果たしてきた