第3回目
2012/4/23
「アセンブラ（アセンブリ言語）と機械語の続き」

レジスタの使用に関する慣例

MIPSは32本の算術・論理演算に使用するレジスタを持つが、その役割をコンパイラ・ローダで慣例的に共通仕様へ
(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
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

他のISAでは、異なるものもあるが32本は相場
(IA32: 8本, x64:16本, Sparc, PowerPC: 32本。。。
IA64 256本 , GPUは64-256だが全体で万以上)

Move: レジスタ間の値の移動

- レジスタ $s0の値を$1に移動

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

- MIPS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 ($zero)
  - → 他の命令で代用
  - `add  $s1, $s0, $zero`

- 先ほどのbltと同じように、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 operands)

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

- 考慮できる解決法
  - 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 の命令: immediate (イミィディエート、即値)

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

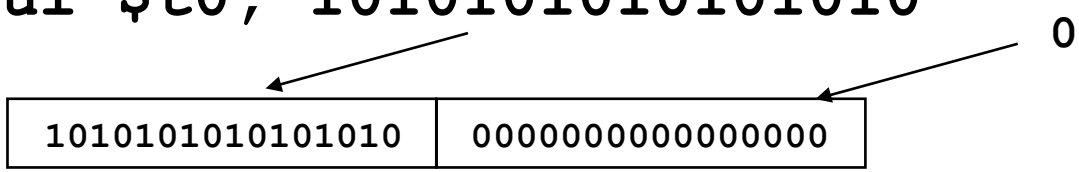
1ワードの命令の中
に定数値を埋め込む

- どのような命令形式にすれば良い?→ I形式

大きい定数はどうする?

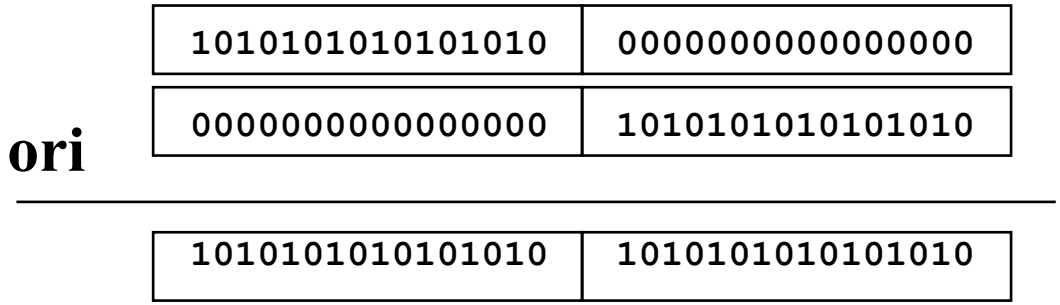
- 32-bitの定数をレジスタにロードしたい
- 新しい命令を使わなくてはならない: “load upper immediate” 命令

```
lui $t0, 1010101010101010
```



- Then そして、下位のビットを設定しなくてはならない, i.e.,

```
ori $t0, $t0, 1010101010101010
```



アセンブリ言語 対. マシン語

- アセンブリ言語はシンボリックで読みやすい形式をプログラマに提供
  - 数字を書き下すよりずっと簡単 (16進数のダンプと比較)
  - e.g., デスティネーションが最初のオペランド、など
- いかし、マシン語が現実に実行されるものである
  - MIPSではすべて1ワードのビット列
  - e.g.,デスティネーションが最初のオペランドではない
- アセンブラは疑似命令を提供できる (pseudo instructions)
  - 複数の命令により一つの命令を実現
  - その他、分岐ラベル、条件アセンブル、定数値の格納など
- 性能評価のためには、実際の命令数を数えなくてはならない
  - (後述のスーパスカラプロセッサだと命令数のカウントは難しいが)

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のアドレスに強制的に分岐
```

(注:実際のLabelのアドレスはアセンブラ等が後で自動計算する)

- 命令フォーマット:

- 注意:これらの命令ではアドレスは 32 ビット未満である
  — どのようにしてロードストア命令を用いてこの不足を補うのか?

分岐命令におけるアドレスの指定法(続き)

- 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:

- lw, swと同様に、レジスタ指定でアドレス値に加算、が可能であれば良い
  - ここで、現在実行中の命令のアドレスを示すレジスタとしてPC(Program Counter, または Instruction Address Register)を導入
    → *PC相対アドレッシング*
  - ほとんどのブランチは局所的である(局所性の原則)
- 無条件ジャンプ命令では、PCの高位ビットをそのまま用いる
  - 256 MBのアドレス指定の範囲が可能
- さらに32ビット全部指定することを可能にする
  - 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
```

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.

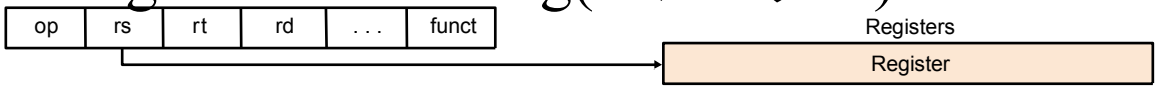
MIPS assembly language

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

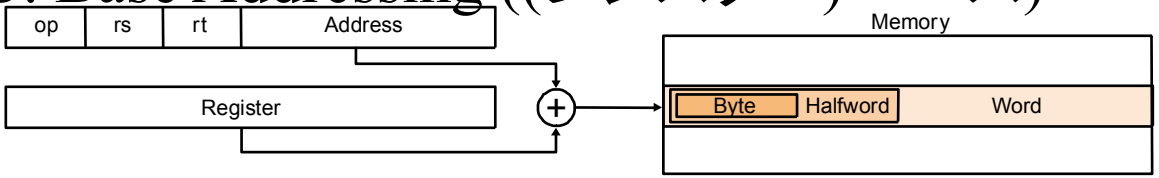
MIPSアドレッシングモードのまとめ

1. Immediate Addressing(直接)

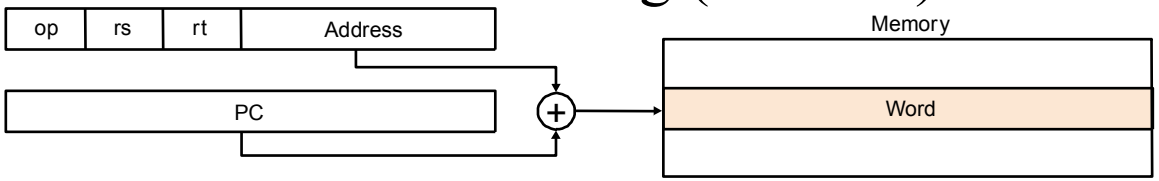
2. Register Addressing(レジスター)



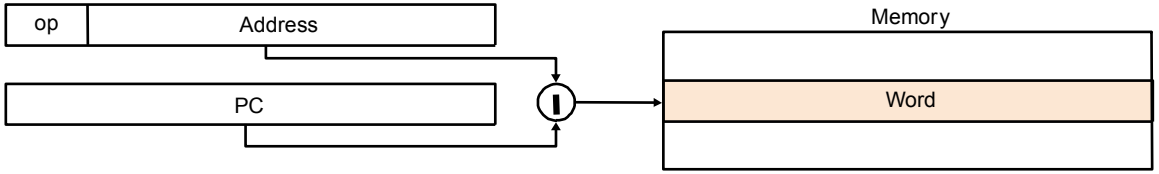
3. Base Addressing ((レジスター)ベース)



4. PC-Relative Addressing (PC相対)

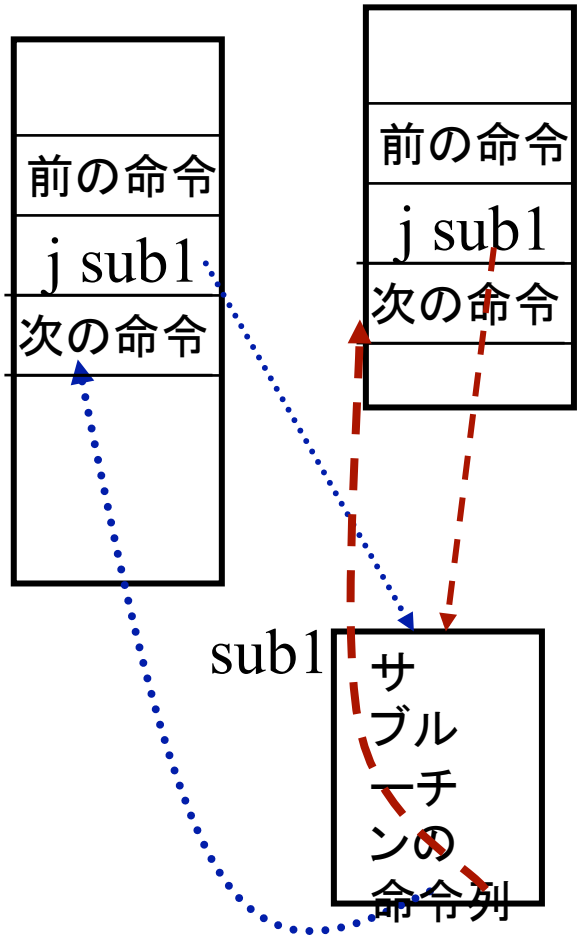


5. Pseudodirect Addressing (疑似直接)



手続き・サブルーチン・関数・メソッド呼び出しのサポート

- 手続き・サブルーチン・関数・メソッドはプログラム言語において尤も重要な手続きの抽象化
- 機械語には、サブルーチンの概念はない→分岐で実現
  - 1. 戻りアドレスの扱い (複数コールサイトからの呼び出し)
  - 2. 引数の渡し方 (戻り値も)
  - 3. 局所変数のサポート
  - 4. 再帰手続きのサポート
- デザインの選択肢は? あまり良くない例
  - 1. 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?
  - 2. 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?
  - 3.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?
  - 4. ??? うまくいかない!



サブルーチン等のMIPS ISAにおける実現

- 1. 戻りアドレスの算定
  - MIPSでは jal (jump and link)命令

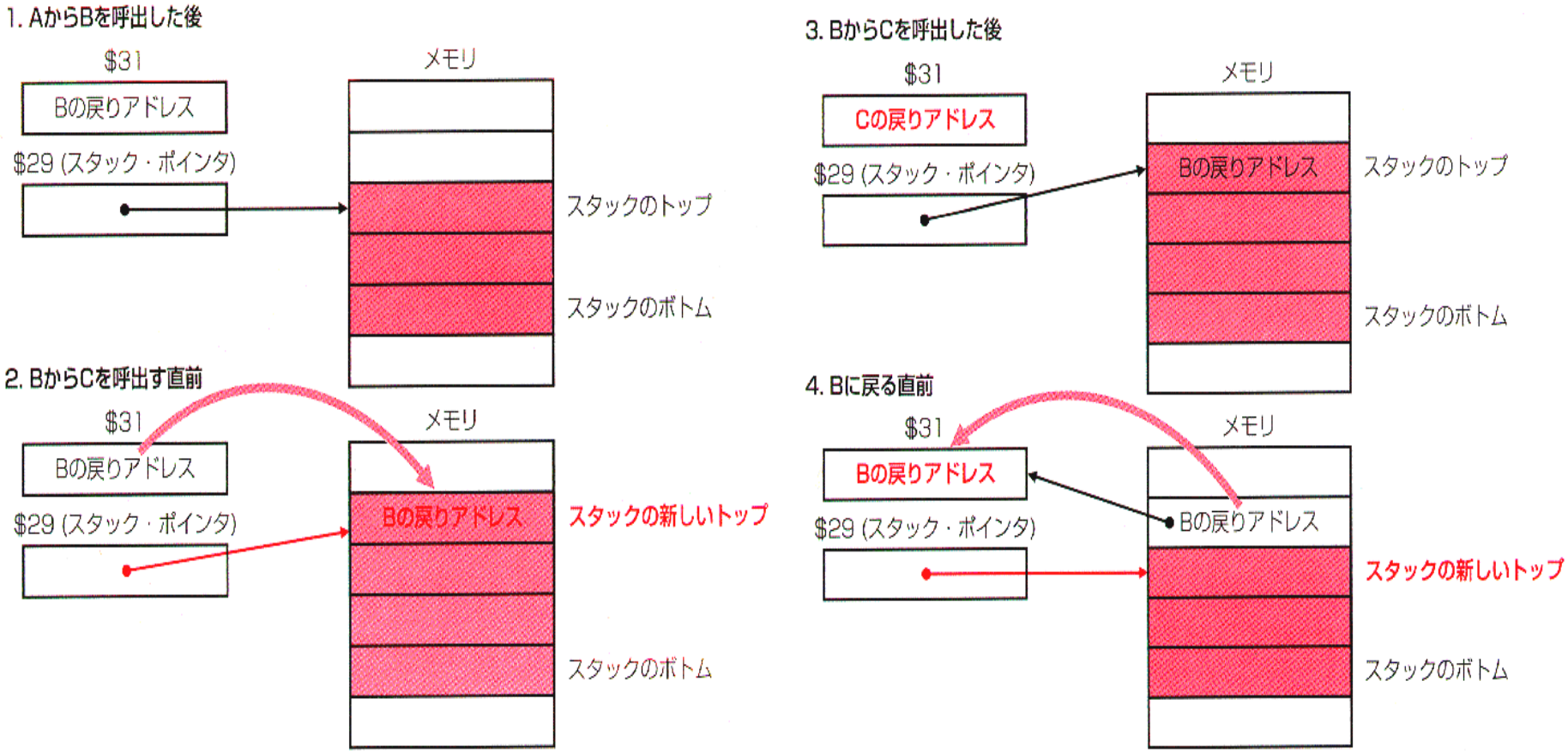
    `jal Label  #$ra (31)にPC+4(自命令の次)を格納し、Labelへ分岐`

  - スタックの活用
    - $raの戻り番地をスタックにpush
    - MIPSでは、専用のレジスタ $sp (29)がスタックポインタ
    - スタックは、上位アドレスから下位アドレスに伸びる
- 2. 引数の渡し方 (戻り値も)
  - レジスタ+スタック上のフレーム　(stack frameスタックフレーム)
- 3. 局所変数のサポート
  - レジスタ+スタック上のフレーム
- 4. 再帰手続きのサポート
  - スタック上にフレームをとることによって、自然にサポート

サブルーチンの実現(2)

- ```
  A: ...
     jal B              #サブルーチン呼び出し
    ...
  B: subi $sp, $sp, 32   #フレームの確保
    sw   $ra, 0($sp)    #戻り番地の待避
    jal  C
      ...
      lw   $ra, 0($sp)    #戻り番地の復元
    addi $sp, $sp, 32   #フレームの消去
    jr   $ra            #サブルーチンから戻る
  C: ...
     jr   $ra
  ```
- 注
  - 実際は、引数の受け渡しなども行う
  - MIPSでは、正式には戻り番地は20($sp)に格納する
  ```

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

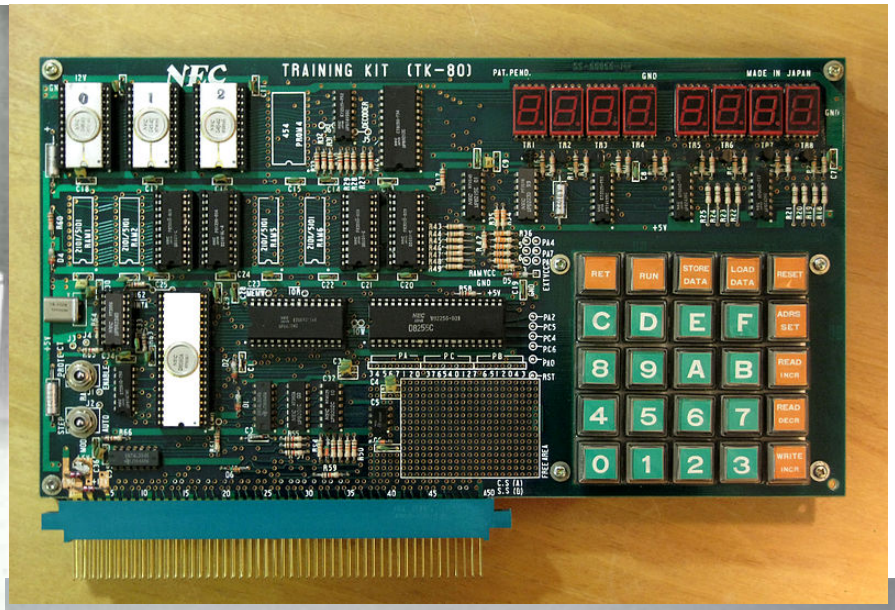
値の受け渡し方と局所変数

- どこに引数を入れて渡すか
  - スタック渡し
  - レジスタ渡し
  - 渡し方に一定の規約(convention)
    - MIPSでは, $a0-$a3 (4-7) が引数、$v0, $v1 (1,2)が戻り値
    - ハードウェアによる制限はないが、コンパイラ、ライブラリやOS等は必ずこの規約を守ることにする
- 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 (呼び出し側保存) レジスタ
    - $t0-$t9 (8-15,24,25)
  - callee-save (呼ばれた側保存)レジスタ
    - $0-$s7 (16-23)
  - 残りはスタック上に
  - Q: Caller-save と Callee-saveはどのように使い分けるか?

「代わりの」アーキテクチャデザイン

- 代わりのデザイン:
  - より強力な命令を提供する
  - 目標は命令数の削減→ 単位時間あたりの命令数を削減すれば速くなる?
  - 計算機の速度は、クロックサイクル時間とCPI (Clocks per Instruction, 命令あたりの平均クロック数)が本質的に重要(後述)
    - クロックサイクルの遅滞と、CPIの増加を招く
- “RISC 対 CISC”論争とも呼ばれる
  - 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(ARM)。
  - 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!*
- Power & PowerPC (RISC)と80x86(CISC)を見てみよう

VAX-11/780(VAX), NEC TK-80 (8080), IBM-PC(8088), PowerMac8100(PowerPC)



比較:Power & PowerPCの命令体系

- スパコン・メインフレーム(Power, BlueGene)からゲーム機(Wii-Broadway, PS3-CellBE, XBOX360-PX)、組込み(PowerPC)まで
  - Macintoshも1994-2006年はPowerPC (2006年から x86へ)
- 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

80x86に関して—ISAの進化過程

- (1974: Intel 8080 (8 bit アーキテクチャ) 2Mhz)
- 1978: Intel 8086 が発表される (16 bit アーキテクチャ)
  - CISC, 8080の系譜(ただし、完全な互換性はない), クロック 8 Mhz
  - 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
  - 20bitセグメントアドレシング→4本のセグメントレジスタ
  - 1バイトから5バイトまでの命令 (ブロック転送用の命令などもあり)
  - 規則性、直交性に欠けるISA
- 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

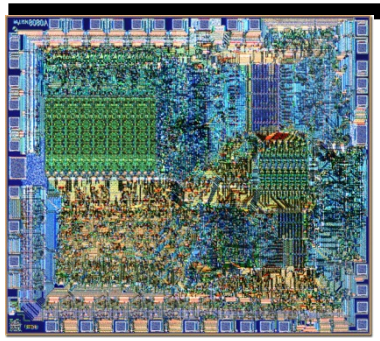
ここまでが16ビット。以下32ビット拡張

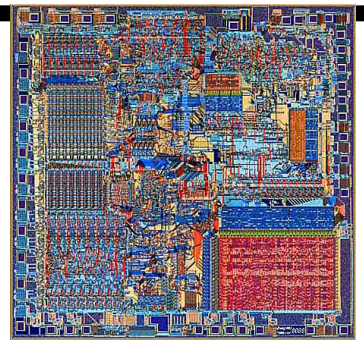
- 1985: 80386 は 32 bitアーキテクチャに, 新たなアドレッシングモード、8本の汎用レジスタ、「平滑」で連続なアドレス空間(セグメントレジスタの無効化)→RISCやVAXに相似
- 1989-1995: 80486, Pentium, Pentium Pro は若干の命令追加、しかし大幅なマイクロアーキテクチャ改変
  (ほとんど同じISA、大幅な性能向上：16Mhz→200Mhz)

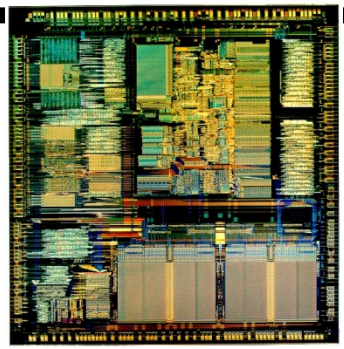
ここまでで32ビットの通常命令の拡張は終了。以下、(1) SIMDベクトル命令と、(2) 64ビット拡張

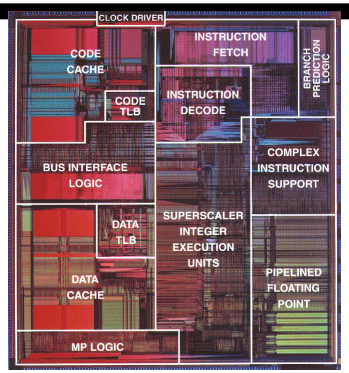
- 1997: Pentium MMX (MultiMedia Instruction Set)の追加, Pentium II 266Mhz
- 1998: AMD K6/2 3DNow! (SIMDベクトル拡張命令の追加)
- 1999: Pentium III SSE (Streaming SIMD Extensions) 600Mhz
- 2001: Pentium 4 SSE2 (Streaming SIMD Extensions 2) 1.5Ghz -> 3.8Ghz (Pentimu 4 2004)
- 2003: AMD Opteron/Athlon64 AMD64 (64bit命令体系への拡張、レジスタの倍増)、
  Intel Pentium 4 SSE3 (Streaming SIMD Extensions 3)
- 2004: AMD64/IA32-e の融合(x64 = EM64T=AMD64+SSE3)
- 2007: Intel SSE4拡張 (Core 2 45nm Penryn)
- 2007: AMD SSE5拡張 (2011年にAVXに統合しBulldozer実装)
- 2008: Intel AVX拡張 (Advanced Vector Extensions) (2011年 Sandy Bridgeに実装)を発表
- 2008: LNI (Larrabee New Instructions) を発表
- 2011: Sandy Bridge 発売、AVX+暗号化サポート命令
- 現在の Intel, AMD全てのCPUで、8086の16ビット命令を実行可能

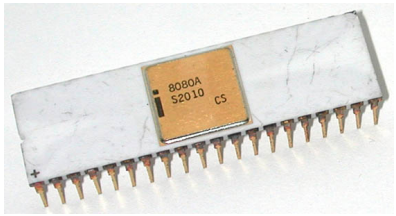
Intel プロセッサの系譜の画像

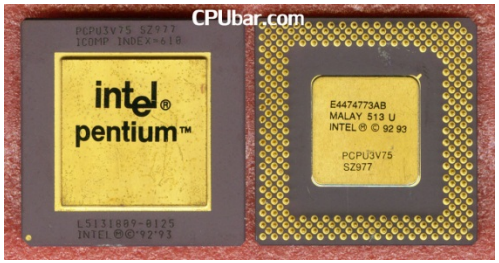


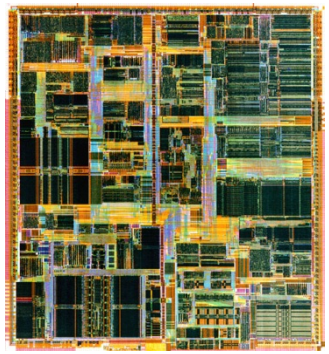


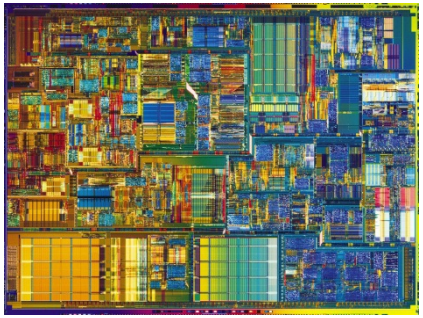


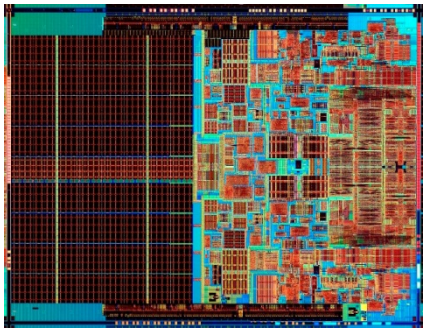


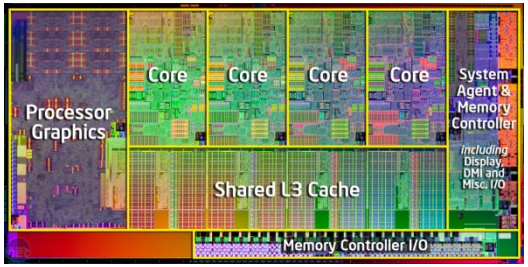


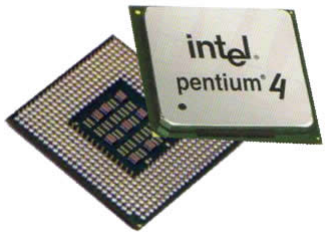


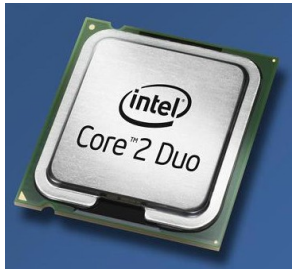


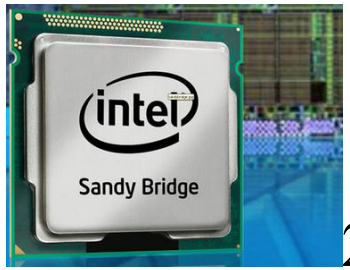












80x86 (IA32): “dominant”なISA

- 80x86(IA32)命令セットアーキテクチャの複雑さ:
  - 1バイトから17バイト長の命令 from 1 to 17 bytes long
  - 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（何故これは良くない?)
  - 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
  - 複雑なアドレッシングモード
    例, “base or scaled index with 8 or 32 bit displacement”
  - 詳しくは参考書を見よ
- どうして世の中はそれでもうまく動いているか:
  - もっとも良く実行される命令はあまり複雑ではない(レジスタ間演算やレジスタメモリ演算)→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
  - Pentium Pro/II/III/4/M/Core/Core2/Nehalem, AMD K6/Athlon/Athlon64/Phenomでは、CISC命令を内部的にRISC命令に変換
    - CISC命令→複数のRISC命令にハードウェアで変換
    - コアはRISCプロセッサ (Pentium MMX, Cyrix 6x86MX, Larrabeeまでは内部もCISC)
    - その他、レジスタリネーミング、正確なブランチ予測など、近年のプロセッサ技術の総動員(アドバンストピックなので、本授業では触れない)

まとめ

- 命令の機能面のみを追求してはいけない
  - 命令数削減 対 高いCPIとクロックレートの低下
- デザインの原則: (注:ソフトウェアも同様)
  - 単純さは規則性を好む
  - 小さいものは速い
  - 良いデザインには時には妥協も必要
  - 一般的なケースを速くすべし
- 命令セットアーキテクチャ (Instruction Set Architecture = ISA)
  - 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」
  - Von Neumann アーキテクチャ