計算機システム
第2回
2011/05/02(月)

「コンピュータ・アーキテクチャへのいざない」

第一章

コンピュータアーキテクチャへの入門

- 急激に進歩している領域:
  - 真空管 -> トランジスタ (シリコン)-> IC -> VLSI
  - 1.5 年で倍増 (Moore's Law)
    メモリ容量
    プロセッサ速度　(テクノロジと構成両方の進歩)
  - *現状で5000万～数十億トランジスタ/chip, もうすぐ100億トランジスタへ、*
- 何を学ぶか:
  - コンピュータの動作の原理
  - 性能をどう評価するか (または、間違ったやり方)
  - 現代のプロセッサデザインに関して (caches, pipelines, SuperScalar…)
- どうしてこの授業を受けるの?
  - 自分を「情報科学者」と呼びたいから
  - 他の人々が使えるソフトを使いたい(ハードウェアの知識での性能要求)
  - 将来、偉くなったときに必要な知識

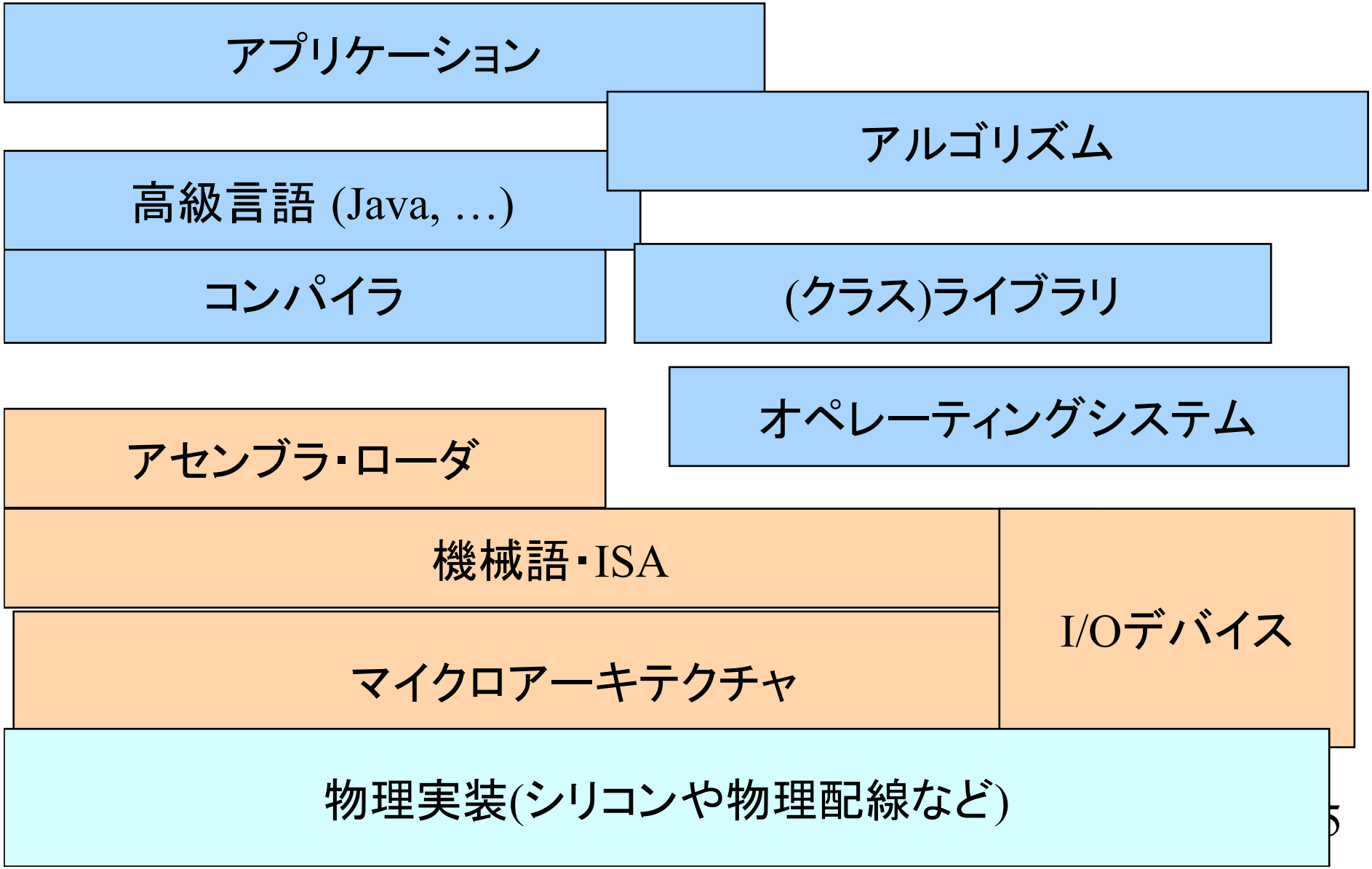
コンピュータとは?

- 部品:
  - 入力 input (mouse, keyboard)
  - 出力 output (display, printer)
  - メモリ memory (disk drives, DRAM, SRAM, CD)
  - ネットワーク network

- 本講義での着目点: プロセッサ processor (データパスdatapath とコントロール control)
  - 数千万～数億トランジスタによる実装
  - 個々のトランジスタを眺めるだけではわからない
    - c.f. 細胞と人間

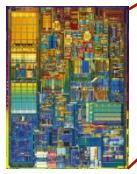
- メモリ, I/O, C言語などもカバー

計算機システムの俯瞰図

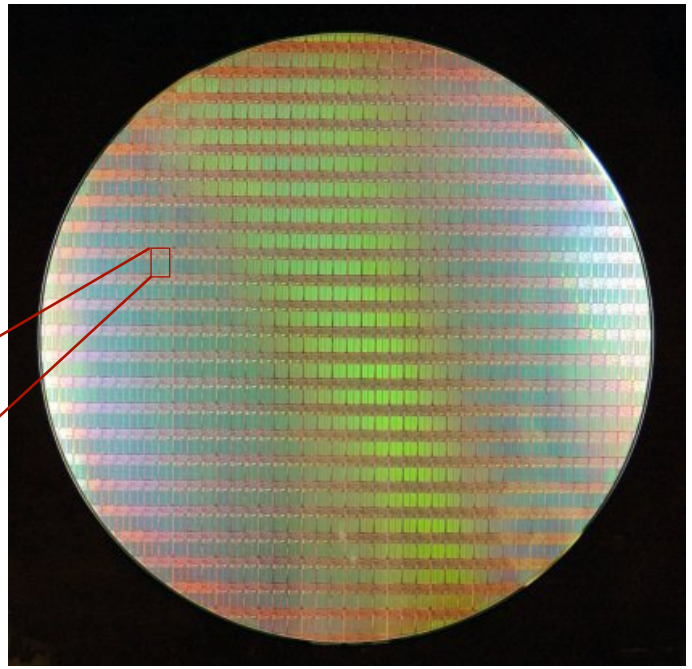


シリコンダイ

- 20cm-30cm シリコン「ウェーハー」
  - 複数のチップを採取
- PCのCPUは60mm2から600mm2まで
- 多くはチップあたり80mm2から200mm2程度
- 面積が少ない⇒
  歩留まりが高い
  - とれる個数が多い
  - 欠損確立が少ない

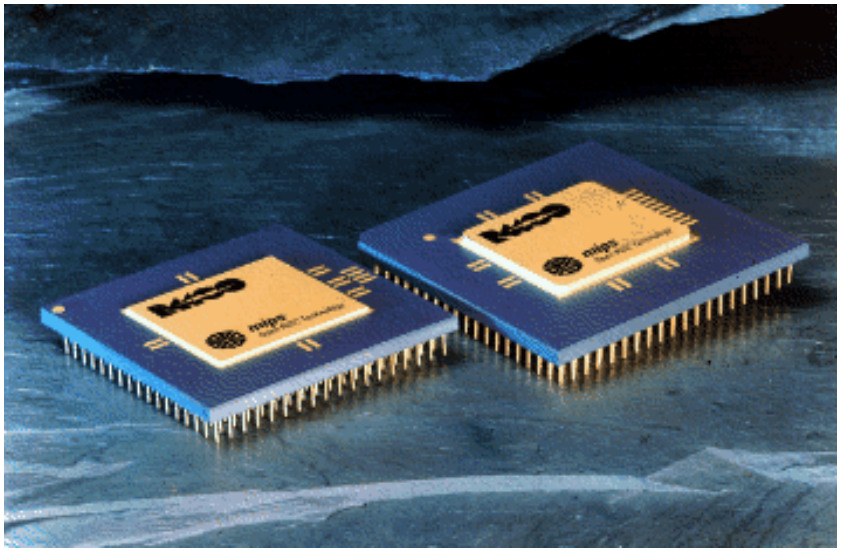


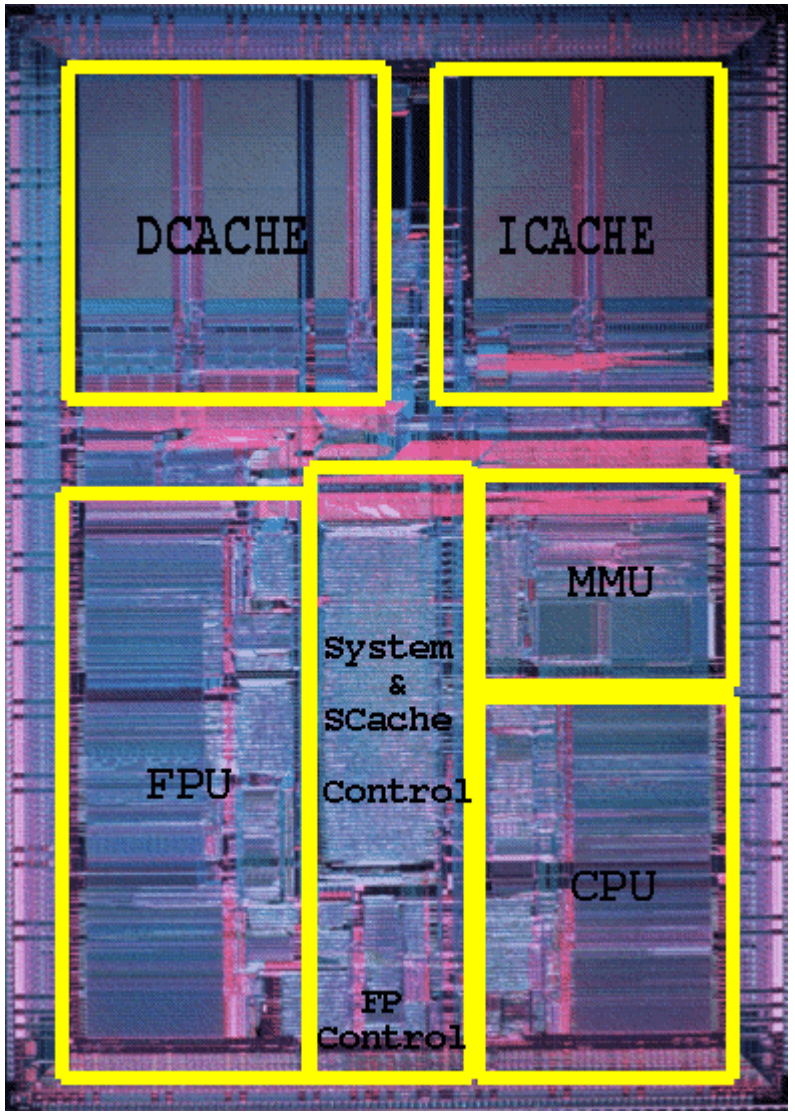
CPUチップ



物理的なプロセッサの構成

- 数十万～数億トランジスタ
- シリコンウェーハー
- 例: MIPS R4400
- 複数の「機能ブロック」により構成
  - しかし、見ただけではわからない





現在では数十nmの配線長

必要なもの: 抽象化

- 情報科学の根元
- より深くなればなるほど、より情報量が多くなる

- 抽象化により、不必要な情報を削減し、
  全体を見渡すことができるようになる

- 右はプログラム言語が実行にいたるまでの
  さまざまな抽象化のレベル
  何が見えてくるか?

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
  temp = v[k];
  v[k] = v[k+1];
  v[k+1] = temp;
}
```

高級言語
C, Java

C compiler

Assembly language program (for MIPS)

```
swap:
    muli $2, $5,4
    add  $2, $4,$2
    lw   $15, 0($2)
    lw   $16, 4($2)
    sw   $16, 0($2)
    sw   $15, 4($2)
    jr   $31
```

アセンブラ
X86, MIPS

抽象化

Assembler

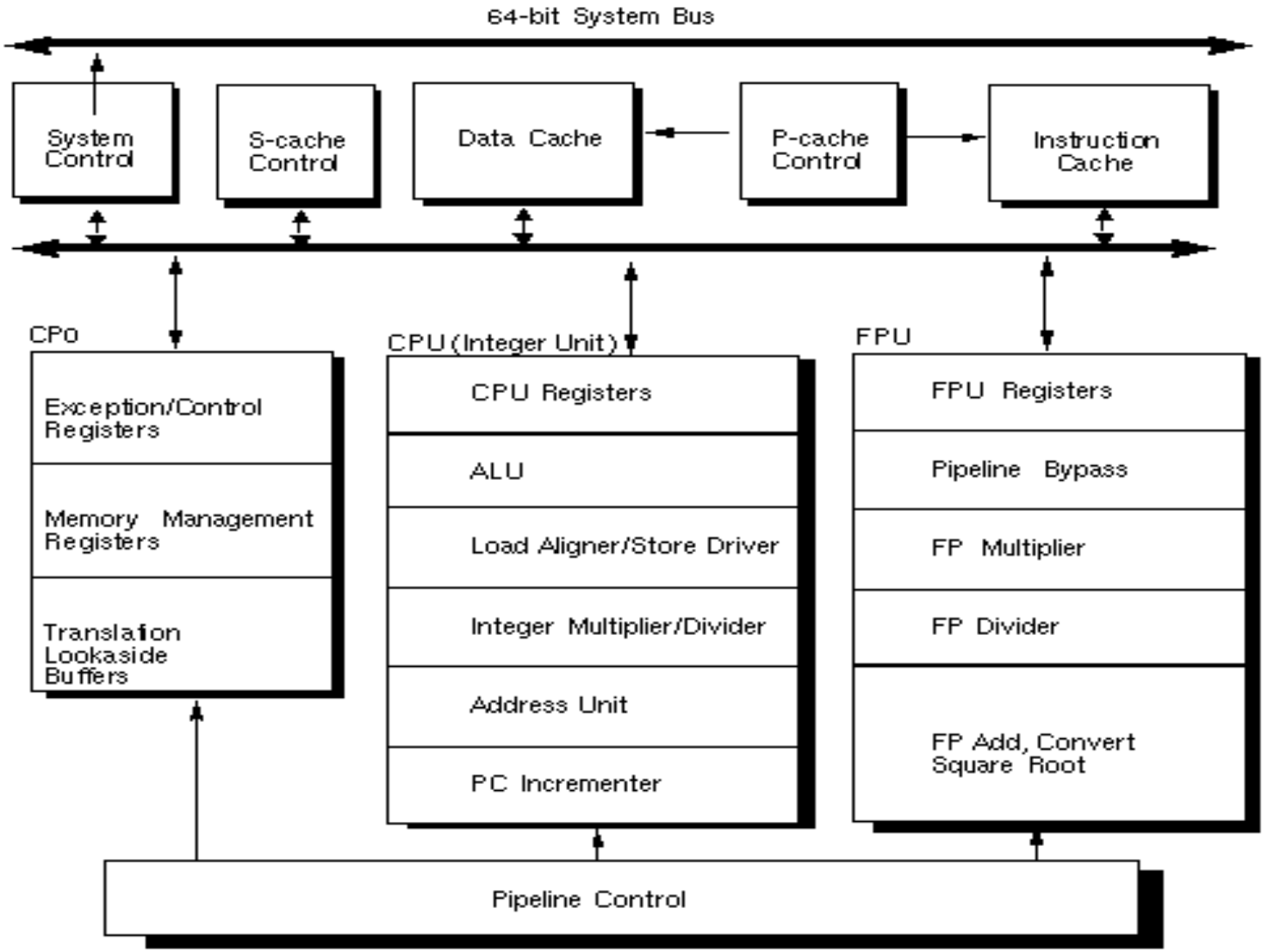
ハードウェアが「解釈」し実行

Binary machine language program (for MIPS)

```
00000000101000010000000000011000
00000000100011100001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

機械語
X86, MIPS

抽象化されたMIPS R4400の構成



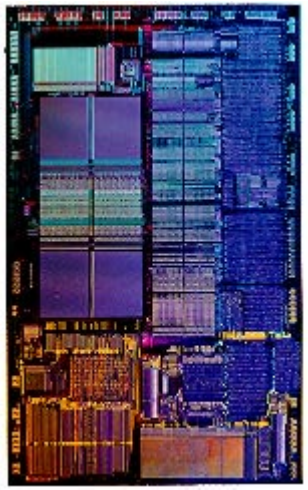
命令セットアーキテクチャ
Instruction Set Architecture

- 非常に重要な抽象化
  - ハードウェアと下位のレベルのソフトウェアとのInterface
  - マシンに対する命令や、マシン語のビットパターンを標準化する
  - 良い点: *同じアーキテクチャの異なる実装が可能*
    - *例Intel i386, Pentium, Pentium Pro, Pentium II, PIII, P4, Core2, Nehalem, Westmere Sandy Bridge…*
  - 悪い点: *時たま、革新を妨げることがある*

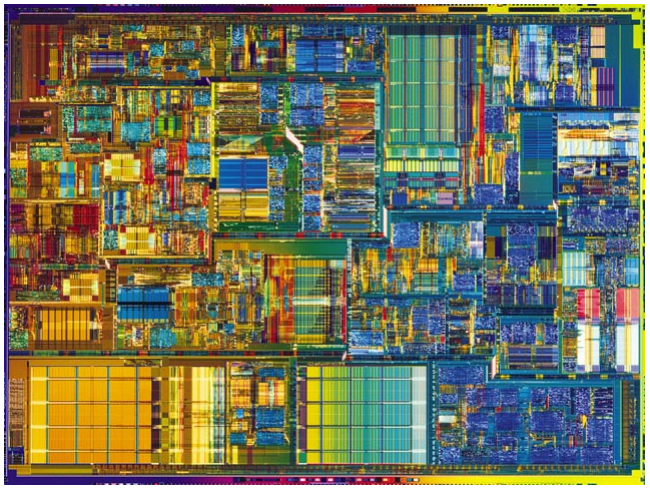
    本当?: バイナリ互換性はもっとも重要?

- 現代の命令セットアーキテクチャ instruction set architectures:
  - IA32(Pentium), x64 (Pentium4, Core2/Nehalem, Athlon), PowerPC, DEC Alpha, MIPS, SPARC, HP PA-RISC, IA64,. ARM, …

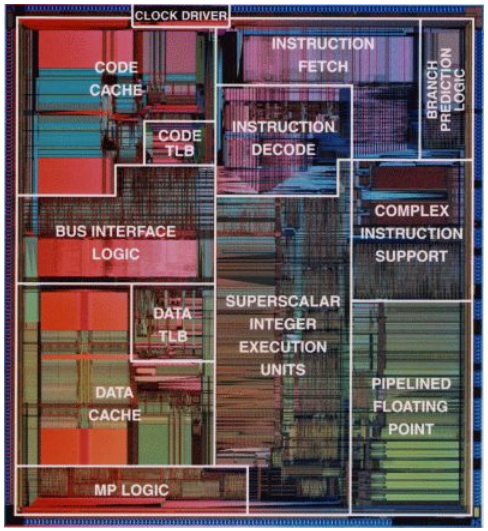
様々x86 ISAのCPU



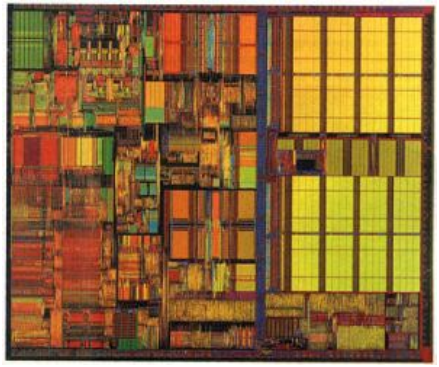
486



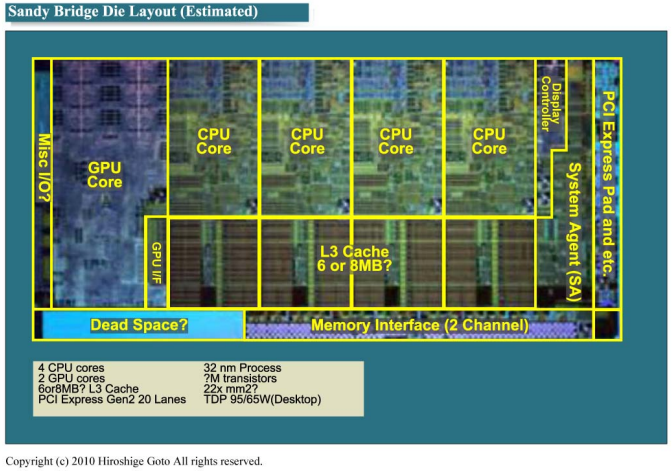
Pentium VI



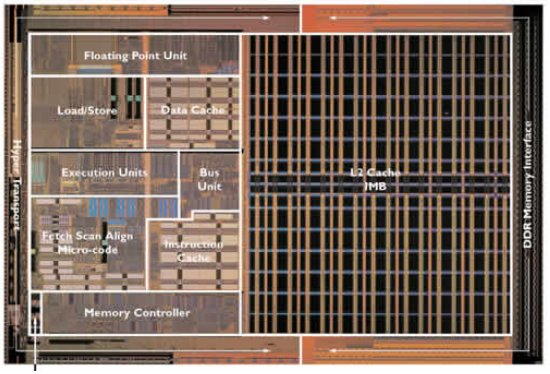
Pentium MMX



Pentium III



Sandy Bridge (4 core)



AMD Opteron (2 core)

講義の予定

- 性能の役割
  Performance issues (2)
- 命令: マシンの言葉
  A specific instruction set architecture (3)
- コンピュータにおける演算とALU
  Arithmetic and how to build an ALU (4)
- 命令をいかに実行する?プロセッサ
  Constructing a processor to execute our instructions (5)
- パイプラインを用いた性能向上
  Pipelining to improve performance (6)
- メモリ: キャッシュと仮想メモリ
  Memory: caches and virtual memory (7)
- 入出力
  I/O (8)

参考書
Patterson and Hennesy
Computer Organization and Design: The Hardware/Software Interface
Morgan Kaufmann Publishers, 1997
(コンピュータの構成と設計(上下)第二版、成田(訳)、日経BP社)

機械語とは:

- 機械をプログラミングする「言葉」
- Java, Cなどの高レベルの言語と比較して、より「原始的」
  例: 複雑な制御構造 (forやメソッド呼び出し)、データ構造などはない
- 個々の機能も限定されている
  e.g., MIPS の算術命令 => 32bit整数, 64bit 実数
- 本授業では、MIPSの命令アーキテクチャを対象とする
  - 1980代から開発されている他のISAと類似(RISC)
  - 例: Sony PlayStation1, 2, PSP, … (PlayStation3は異なる)

デザインゴール: 性能の最大化、コストの最小化, デザインタイムの高速化

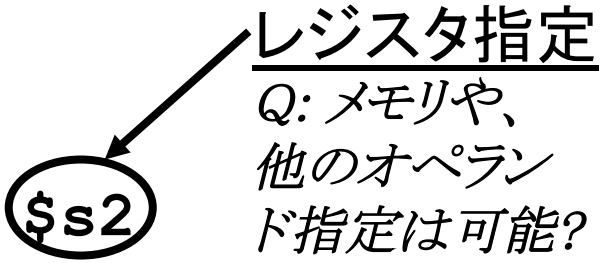
MIPS 算術命令

- 全ての命令は 3 つのオペランド(引数)を持つ

- 引数の順序は固定 (destination first)
  - 命令　　デスティネーション, ソース1, ソース2
- Example:

C code:　　`A = B + C`

MIPS code:　`add $s0, $s1, $s2`



(コンパイラーによって変数に割り付け)

MIPS 算術命令

- デザインの原則: 単純化は規則性を要求する. Why?
- これによって、一見単純な操作が複雑になるが...

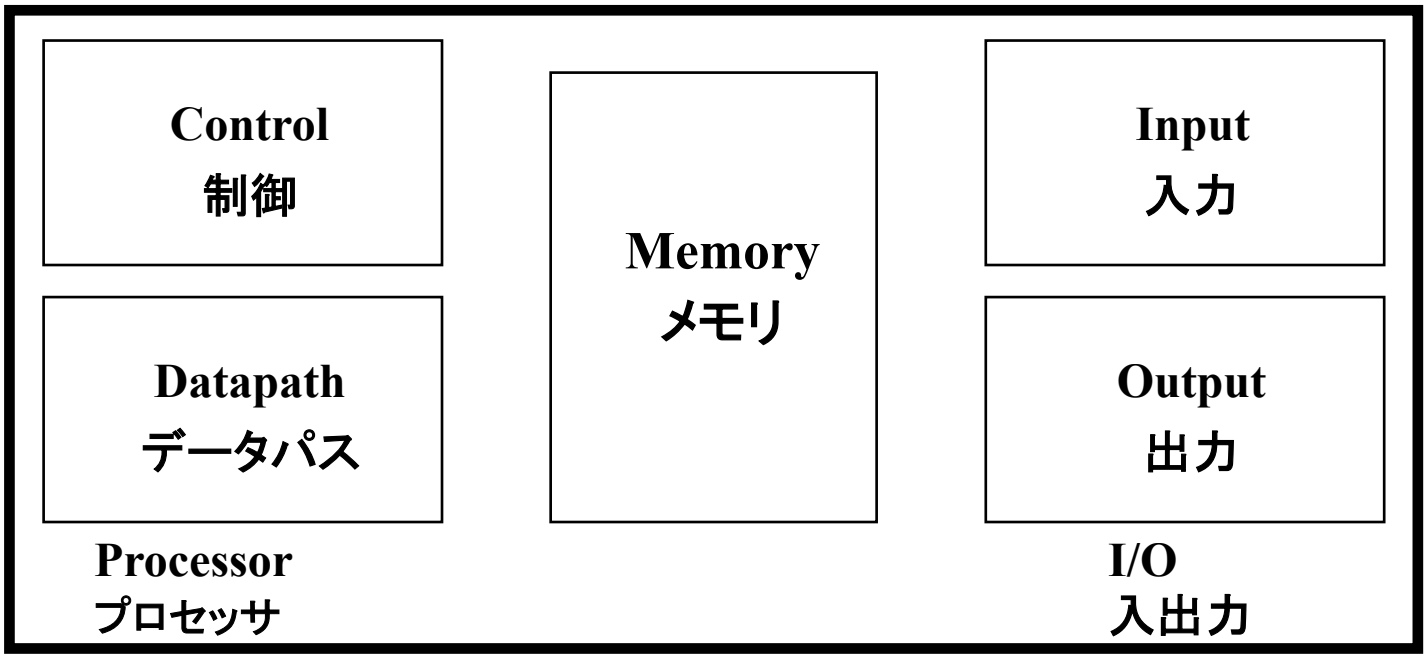
C code:
```
A = B + C + D;
E = F - A;
```

MIPS code:
```
add $t0, $s1, $s2
add $s0, $t0, $s3
sub $s4, $s5, $s0
```

- オペランドはレジスタ(32個)でなくてはならない
- デザインの原則: 小さいものは速い. Why?

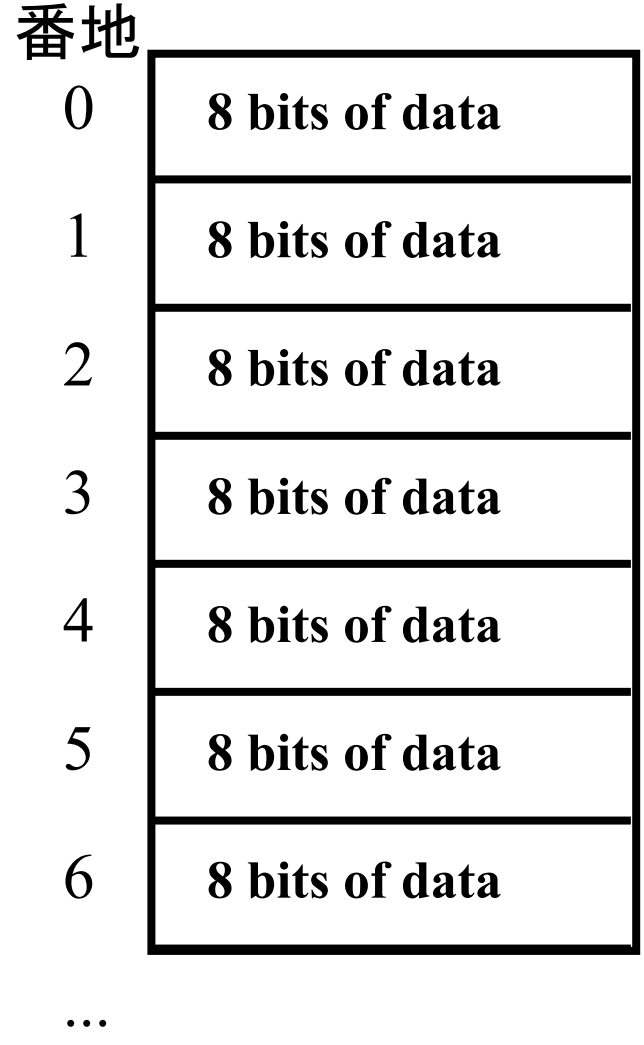
レジスタ とメモリ

- 算術命令のオペランドはレジスタでなくてはならない
  - レジスタ数は32本 ($0, $1, … , $31)
- コンパイラがレジスタを変数に割り付ける
- 変数が32個以上のプログラムはどうする?



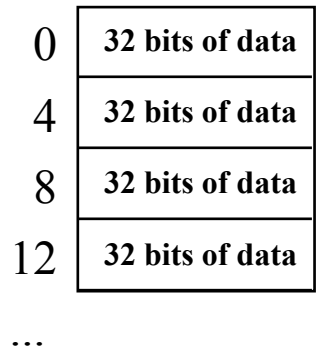
メモリの構成

- 巨大な一次元の配列とみなせる。要素のそれぞれのメモリセルには番地が振ってある。
  - MIPSでは 32-bit = 約40億番地
- メモリのアドレスが配列へのインデックスとなる
- “Byte addressing (バイトアドレッシング)” メモリはバイト(8bit)単位で番地が振られる
  - c.f., ワードアドレッシング: ワード(32bit)単位で番地



メモリの構成(続き)

- バイトは良い単位だが(ASCII英文字など), しかし、他のデータは “word”(ワード)単位で扱われる
- MIPSでは, ワードは32bit、つまり4バイト.



レジスタは 32 bitのデータを保持

- 2^{32} bytes with byte addresses from 0 to 2^{32}-1
  - = 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
- ワードはalign (整列)されている
  Q: ワードの下位2bitの値は?

命令: 算術命令、ロードストア命令

- Load/Store (ロードストア)命令 :メモリとレジスタの間のワードデータの転送
- 例:

C code: `A[10] = h + A[8];`

MIPS code:

```
lw $t0, 32($s3)    // 32 = 4 * 8
add $t0, $s2, $t0
sw $t0, 40($s3)    // $s3にAの番地
```

- Store word はデスティネーションが最後に来ることに注意
- 算術命令はレジスタがオペランド
  - メモリはオペランドにならない!
  - 例のように、メモリに対する算術演算を行いたい場合は、一度レジスタにロードして、操作後、ストアしなくてはならない

最初の例

- どのようにコードが対応するか?

```
swap(int v[], int k);
{ int temp;
   temp = v[k]
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

k → $5
v → $4
temp → $15

$2は?

```
swap:
   add $2, $5, $5
   add $2, $2, $2
   add $2, $4, $2
   lw $15, 0($2)
   lw $16, 4($2)
   sw $16, 0($2)
   sw $15, 4($2)
   jr $31
```

今まで学んだことのまとめ:

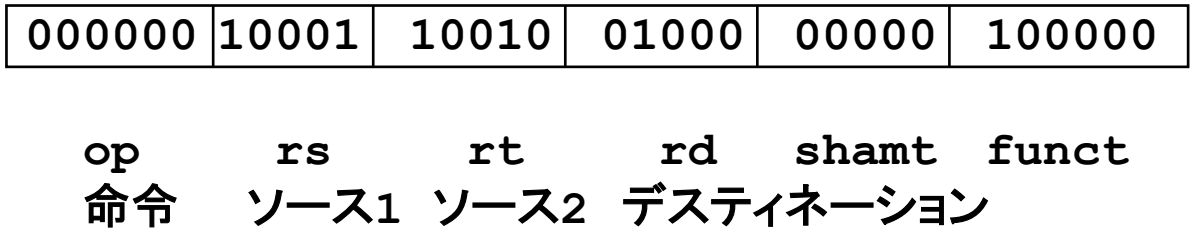
- **MIPS**
  — ロードストアの対象はワードだが、アドレッシングはバイト単位
  — 算術命令のオペランドはレジスタのみ
  - 典型的なRISC (Reduced Instruction Set Computer) アーキテクチャ
    (c.f. CISC (Complex Instruction Set Computer))
- 命令 / 意味

```
add $s1, $s2, $s3        $s1 = $s2 + $s3
sub $s1, $s2, $s3        $s1 = $s2 - $s3
lw $s1, 100($s2)         $s1 = Memory[$s2+100]
sw $s1, 100($s2)         Memory[$s2+100] = $s1
```

アセンブラ命令から機械語へ

- それぞれの機械命令は、レジスタ同様、1ワード(32bit)長
  - Example: `add $t0, $s1, $s2`
  - レジスタには番号を割り振る, `$t0=9, $s1=17, $s2=18`
  - `c.f., CISC`アーキテクチャ→ 命令は可変長

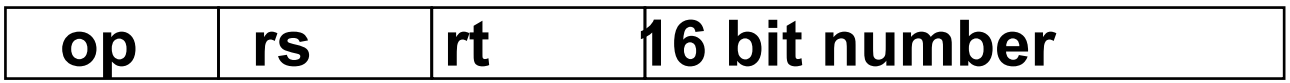
- 命令フォーマットの例 (r形式):



- *それぞれのビットフィールドの名前の意味するところは?*

機械語(続き)

- Load-word (lw)と store-word(sw)命令を考えてみよう
  - 均一性の原則からは、どのようなデザインが芽生える?
    - メモリ番地の指定が大変難しくなる!
  - 新原則:「良いデザインには妥協も必要だ」
- 新しい命令形式
  - データ転送のためのI形式
- 例: `lw $t0, 32($s2)`

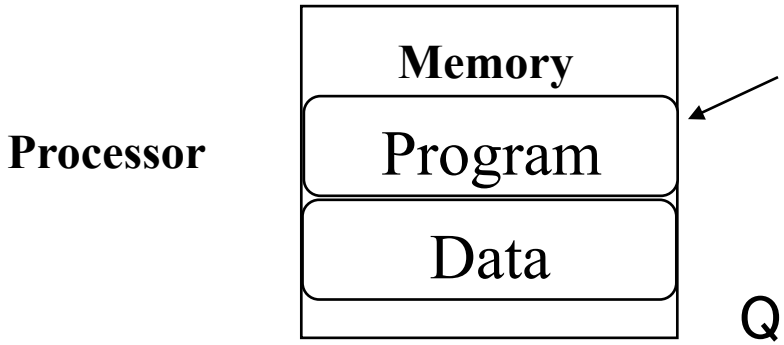


- 妥協点はどこに?

Stored Program 方式の概念

- “von Neumann アーキテクチャ”
- 命令もビット列で表現できる
- プログラムもメモリに格納される
  — データのように読み書きが可能

Q1: von Neumannアーキテクチャでないものは?



メモリには、データのみならず、OSやコンパイラやアプリケーションなどが格納されている。

Q2: メモリ内のプログラムとデータはどのように区別(可能)?

- 命令サイクル (Fetch & Execute)
  - 命令はメモリからフェッチされて、特殊なレジスタに格納される
  - レジスタ内のビットが命令の実行を制御する(命令デコード+実行)
  - 次の命令をフェッチし、続ける
  - 特殊レジスタ Program Counter (PC) の存在

Control (制御命令)

- 判断を行うための命令
  - control flow (制御の流れ)を変更する
  - i.e., 次に実行する命令を変更する

- MIPS 条件分岐命令 →二つのオペランドの比較:

```
bne $t0, $t1, Label // $t0 != $t1
beq $t0, $t1, Label // $t0 == $t1
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label:    ....
```

Control (続き)

- **MIPS 無条件分岐命令:**

```
j  label
```

- **例　if--then--else:**

```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                     j Lab2
    h=i-j;               Lab1: sub $s3, $s4, $s5
                         Lab2: ...
```

- *Q: While文はどのように?*

```
while (i!=j)             Lab1: beq $s4, $s5, Lab2
    i=i+j;               add $s4, $s4, $s5
                         j Lab1
                         Lab2: ...
```

Control Flow (制御の流れ)

- 等しいかは: beq, bne, だが、blt「値が小さければブランチ」は?
- 新命令 slt (set if less then):

```
slt $t0, $s1, $s2
```

```
if  $s1 < $s2 then
   $t0 = 1
     else
   $t0 = 0
```

- この命令を使って blt命令を実現可 "`blt $s1, $s2, Label`"
  — これによって一般的な制御構造が記述可能
- アセンブラは一時レジスタを一本要求することに注意
  — レジスタの使用に関するConvention (慣例)がある

- Q: blt を実現せよ。ただし、一時レジスタを$t0とせよ。

今まで学んだことのまとめ(2):

- アセンブラ命令　　意味

`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 – $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	もし `$s4 != $s5`ならば次の命令は `Label`
`beq $s4,$s5,L`	もし `$s4 = $s5`ならば次の命令は `Label`
`j Label`	次の命令は `Label`

- 機械語の命令形式: