計算機システム
第2回
2018/12/04(火)

「コンピュータ・アーキテクチャへの入門
→MIPSアセンブリ前半」

コンピュータアーキテクチャへの入門

- 計算機は急激に進歩してきた
  - 真空管 -> トランジスタ -> IC -> LSI, VLSI

    半導体(シリコン)
  - 1.5 年で倍増 (Moore's Law)

    メモリ容量

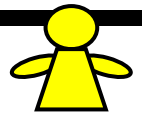
    プロセッサ速度　(テクノロジと構成<u>両方</u>の進歩)
  - *現状で5000万～数百億トランジスタ/chip*
- 何を学ぶか:
  - コンピュータの動作の原理
  - 性能をどう評価するか
  - 現代のプロセッサデザインに関して (caches, pipelines, SuperScalar…)

講義の予定

- アセンブリと機械語
- コンピュータの性能
- コンピュータにおける演算とALU
- 命令をいかに実行する?プロセッサ
- パイプラインを用いた性能向上
- メモリ: キャッシュと仮想メモリ
- 入出力とネットワーク

参考書
Patterson and Hennessy, Computer Organization and Design: The Hardware/Software Interface, Morgan Kaufmann Publishers
(コンピュータの構成と設計(上下)第五版、成田(訳)、日経BP社、2014)

計算機システムの俯瞰図・ソフトウェアとハードウェア



ユーザ

アプリケーション

アルゴリズム

高級言語 (C, Python, Java, …)

コンパイラ

ライブラリ

インタプリタ

アセンブラ・ローダ

オペレーティングシステム

機械語・ISA

I/Oデバイス

マイクロアーキテクチャ

物理実装(シリコンや物理配線など)

コンピュータとは?

- 部品:
  - プロセッサ/CPU
  - メモリ memory
  - ストレージ storage (HDD/SSD/CD/DVD)
  - 入力 input (mouse, keyboard)
  - 出力 output (display, printer)
  - ネットワーク network
  - GPU

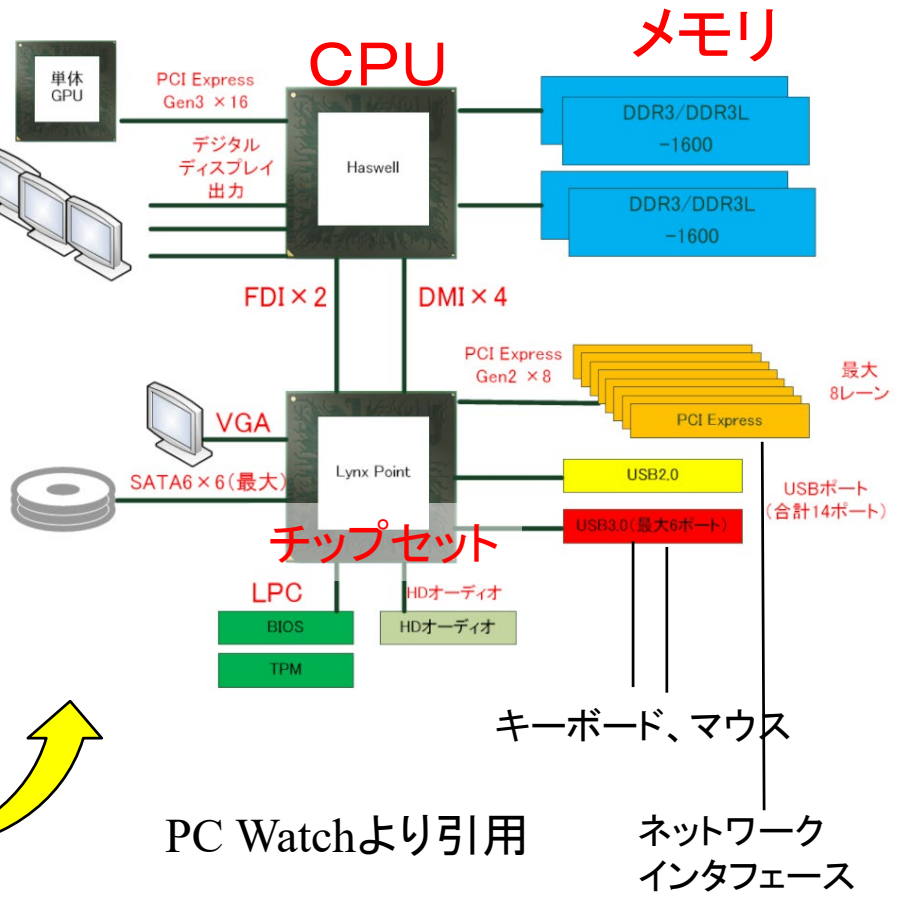
- 本講義での主役: プロセッサ processor (データパスdatapath とコントロール control)
  - 数千万～数百億トランジスタによる実装
  - 個々のトランジスタを眺めるだけではわからない
    - c.f. 細胞と人間

コンピュータの中身(デスクトップPC)

http://catprogram.hatenablog.comより引用

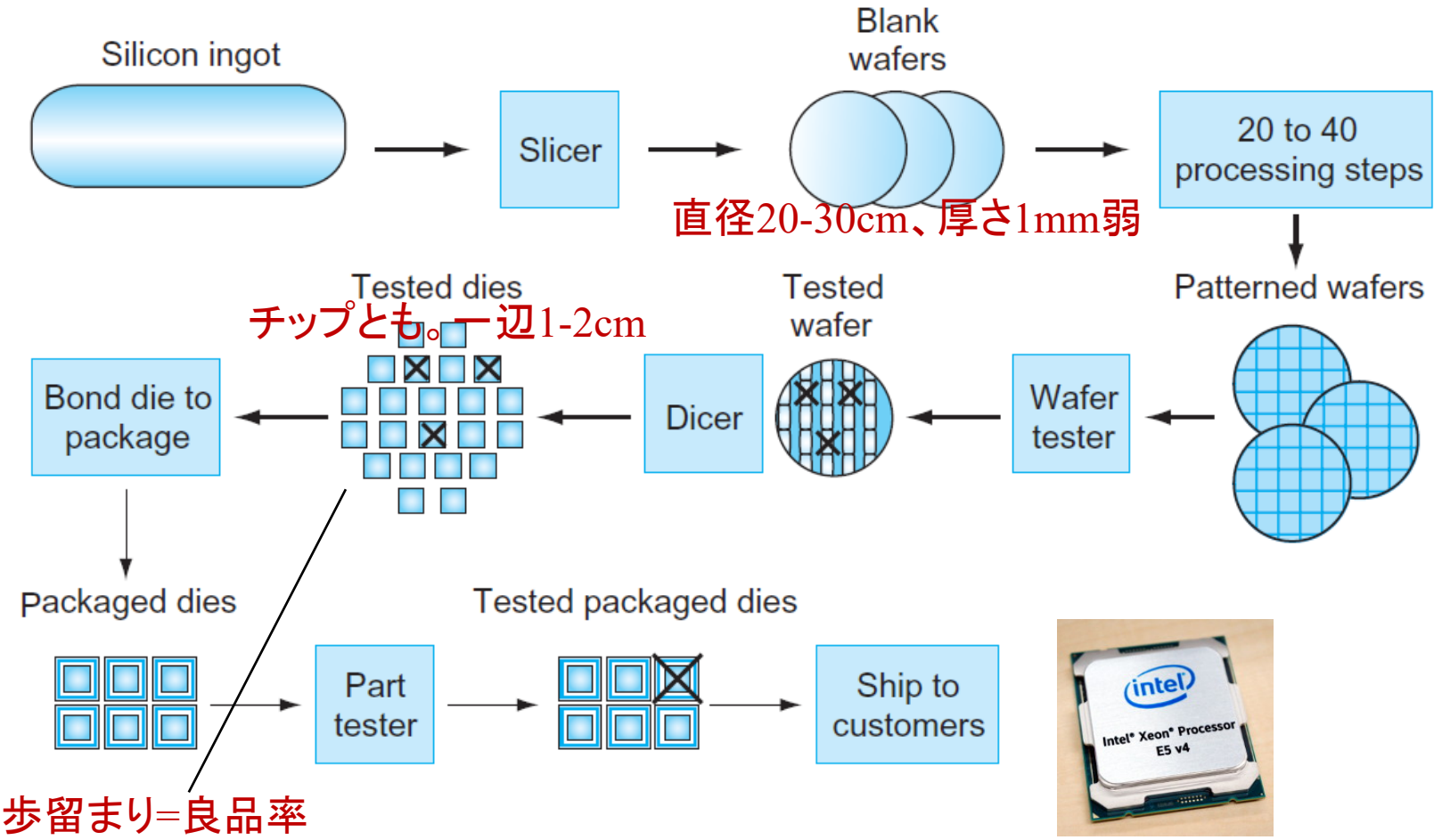
抽象化

ブロック図



PC Watchより引用

CPUの製造



(Computer Organization and Design 5th edition p.26に追加)

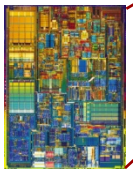
シリコンダイ

- 20cm-30cm シリコン「ウェーハー」
  - 複数のチップを採取
- PCのCPUは60mm^{2}から600mm^{2}まで
- 多くはチップあたり80mm^{2}から200mm^{2}程度
- 配線幅：10nm~65nm

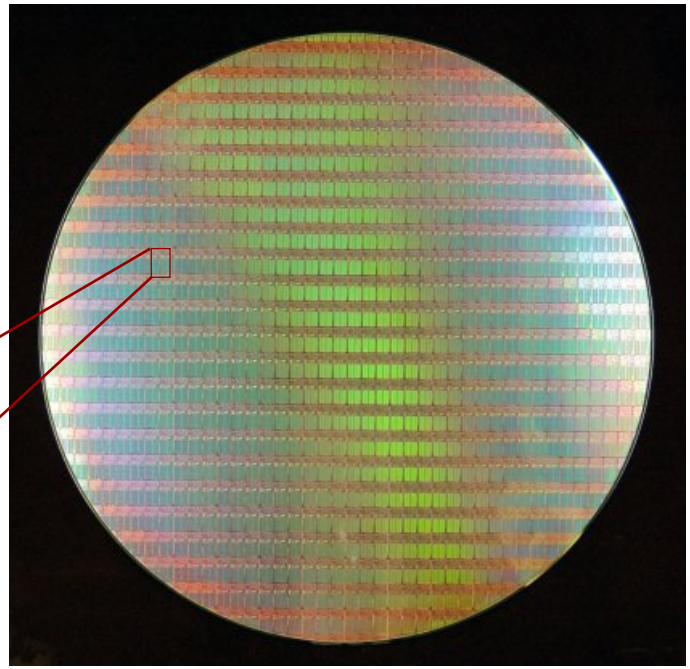
1チップの面積が大きいほうが高性能にできるが・・・

- 欠損確立が高くなり、歩留まりが下がる

→ 最近はマルチコアを利用して歩留まりを上げている

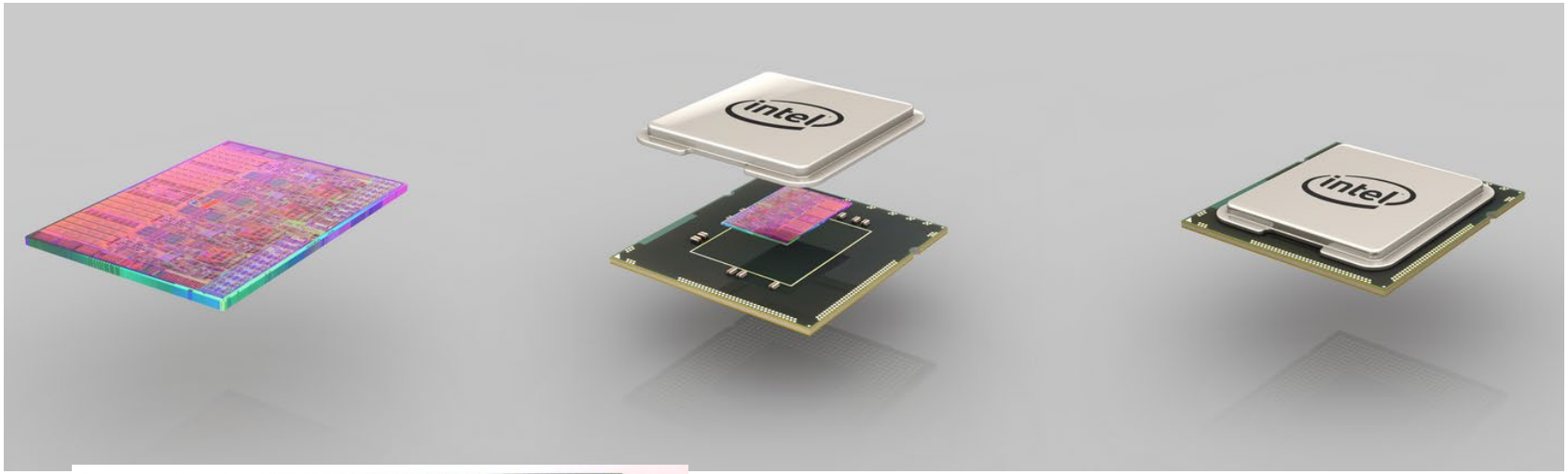


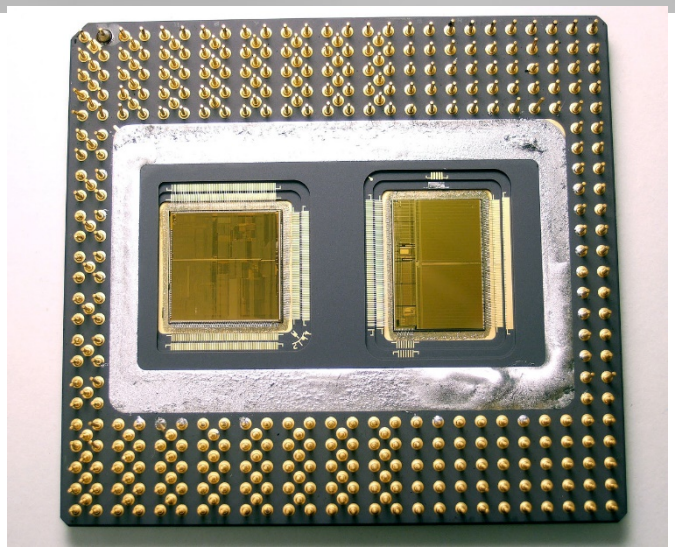
CPUチップ

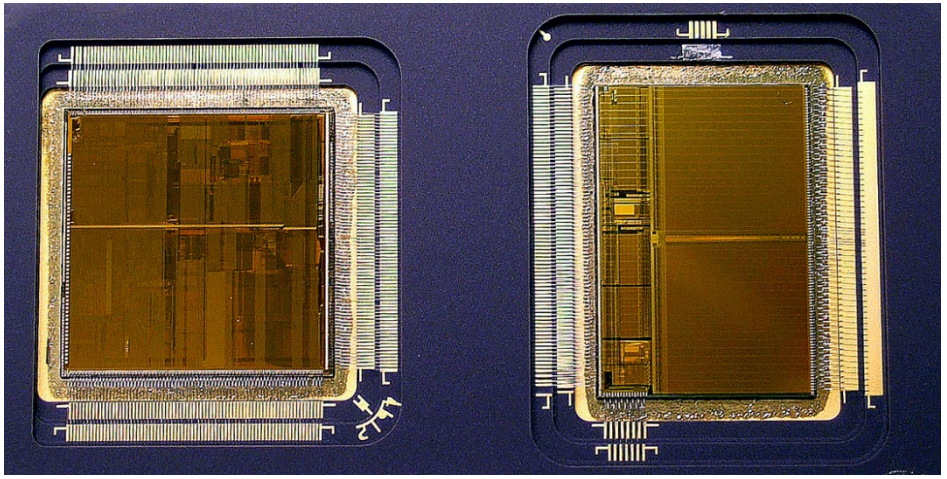


CPU のパッケージング

- シリコンの端をパッケージにボンディング

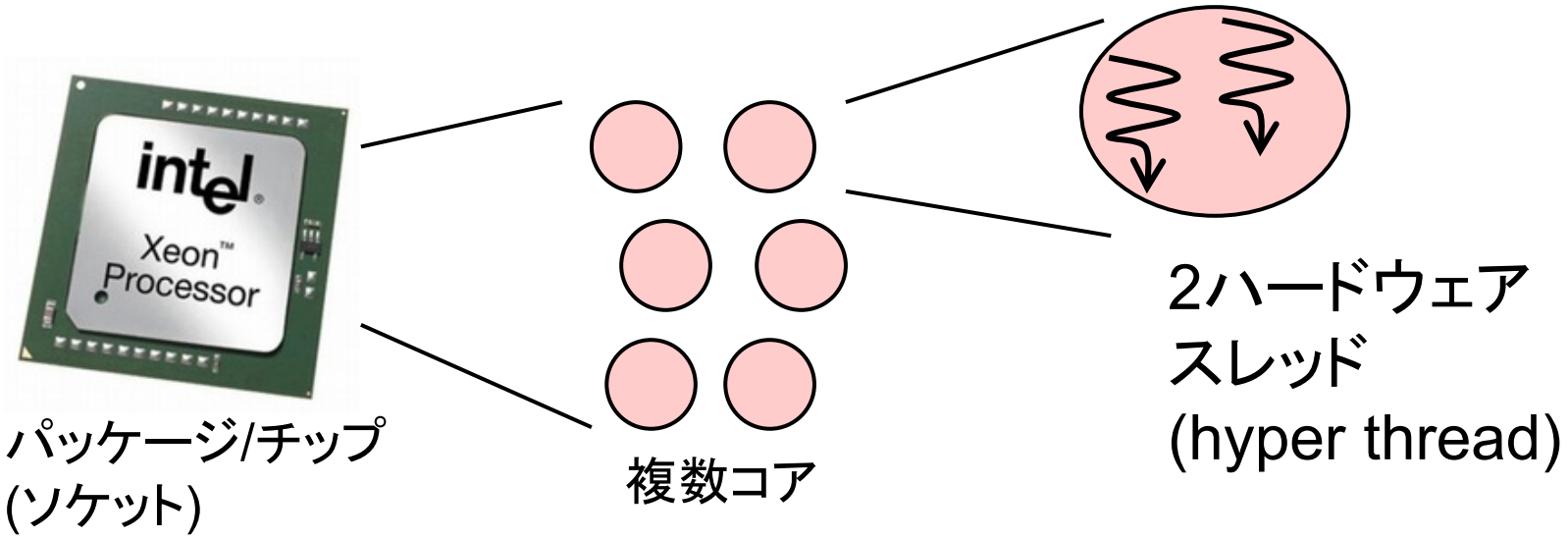






用語：パッケージやコア

- かつての定義：プロセッサ = 機械語を読取・解釈・実行する機能をもつもの
- 2005年ごろからマルチコア時代になり，プロセッサの定義が複雑に
  - 上記の機能を持つ「コア/CPUコア」を複数，パッケージに詰め込むようになった
  - HyperThreadingによりさらに複雑に．1コアを2つのハードウェアスレッドが共有する．OSからはハードウェアスレッドがプロセッサに見える



- さらにさらに、1パッケージに複数チップを詰め込む場合も。AMD EPYCなど
- この講義・実習では、1コアのみ(hyperthreadなし)を主に扱います

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

- **マシン語のビットパターンを標準化したもの**
  - ハードウェアと下位のレベルのソフトウェアとのInterface

- **非常に重要な抽象化**
  - **良い点:** ***同じアーキテクチャの、異なる実装が可能***
    - *例 x86 ISA: Intel i386, Pentium, Core-i7/i5/i3, Xeon, Xeon Phi, AMD Ryzen, EPYC …*

    ISAが共通なため、1980年代にコンパイルされたプログラムが、現代のPC上で(原理的には)動く
  - **悪い点:** ***時たま、革新を妨げることがある***
    - *x86 ISAは30年以上つぎはぎされてきた*
  - ***互換性を持たせつつも、時々拡張はされた***
    - *x86 (32bit) → x86_64 (64bit), AVX系(SIMD)命令の追加、VT系(仮想マシン対応)命令の追加・・・*

代表的なISA

x86/x86_64 (Intel)

- Intel: Pentium, Core-i, Xeon, Xeon Phi…
- AMD: Ryzen, Epyc, Opteron…

ARM v7/v8/v9 (ARM)

- Qualcomm: Snapdragon
- Apple: A5
- Cavium: Thunder X2
- Fujitsu: A64fx

MIPS (MIPS)
ヘネシー教授による

本講義・演習

RISC-V (UCB)

SPARC (Sun→Oracle)

PA-RISC (HP)

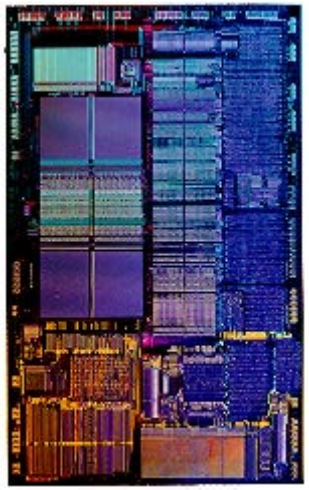
Power (IBM)

68000 (Motorola)

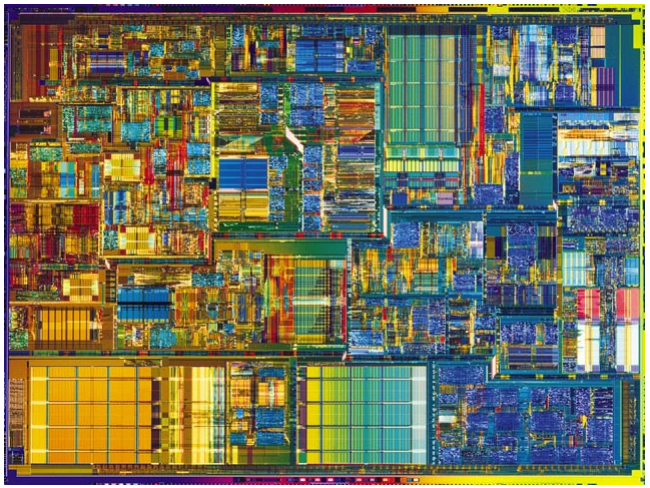
Java byte code ??
(Sun→ Oracle)

← ハード実装も想定されたが、通常はソフトで実装

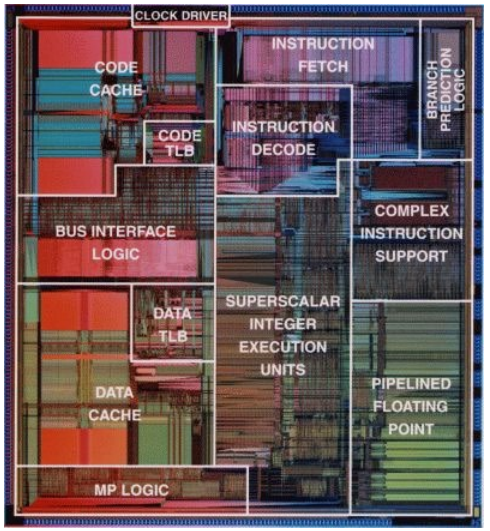
様々x86 ISAのCPU (主にIntel製)



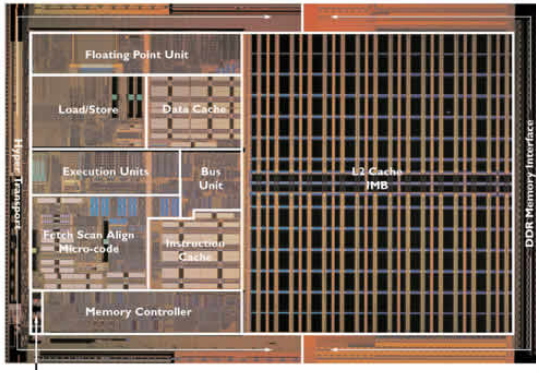
486



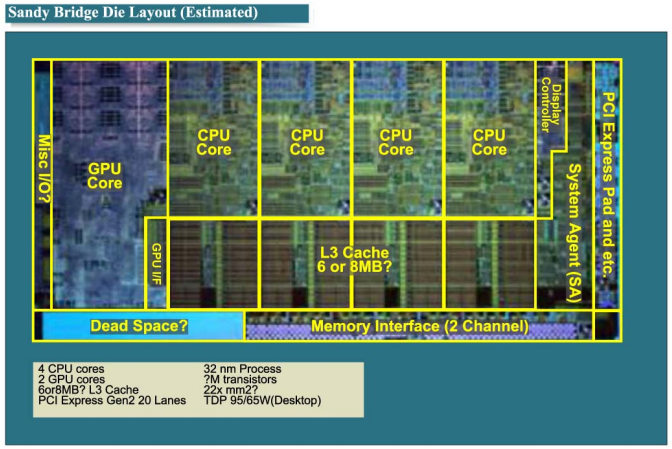
Pentium VI



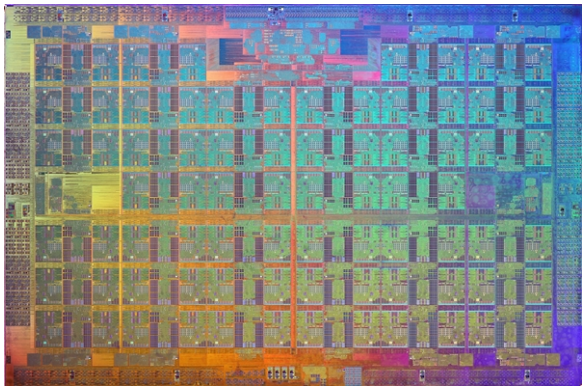
Pentium MMX



AMD Opteron (2 core)



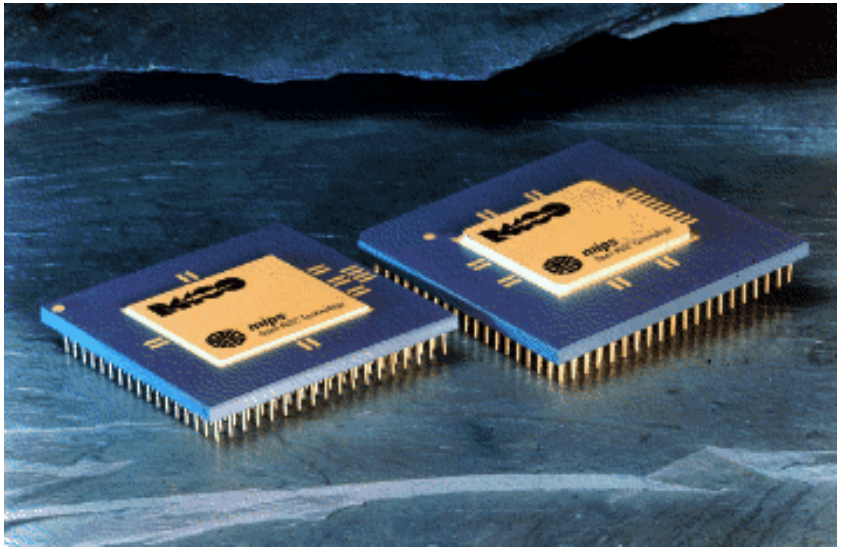
Sandy Bridge (4 core)

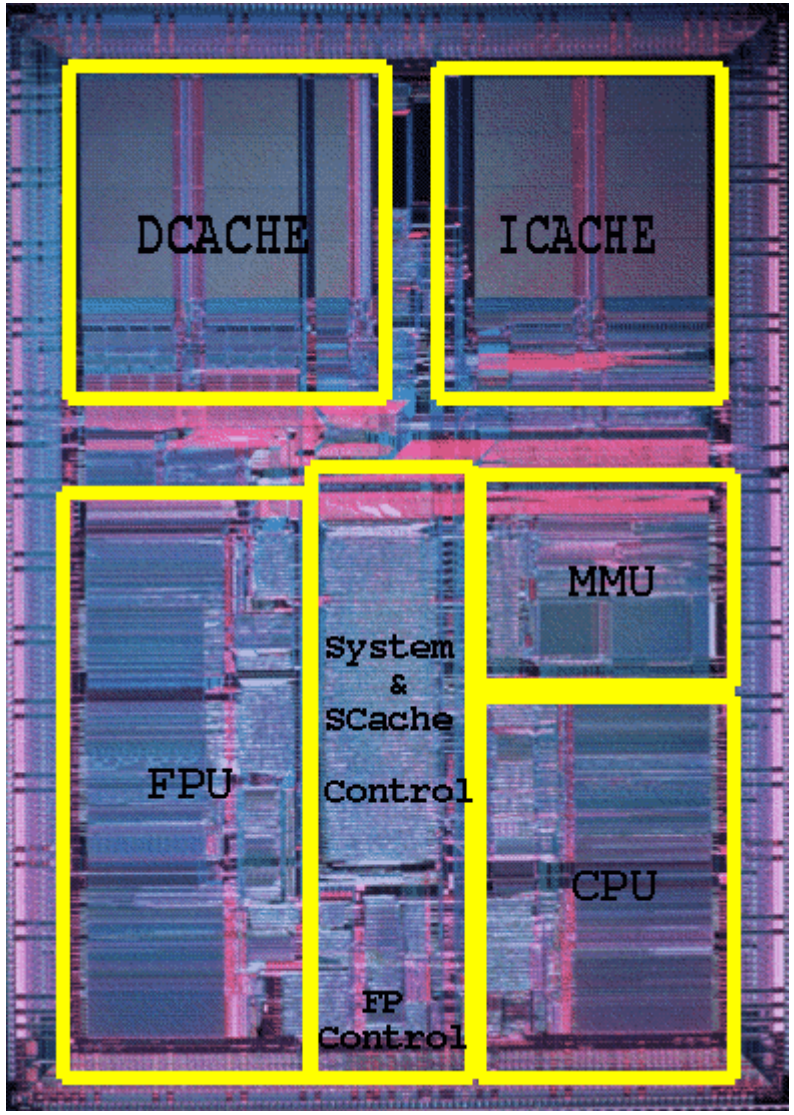


Xeon Phi (72/68/64core)

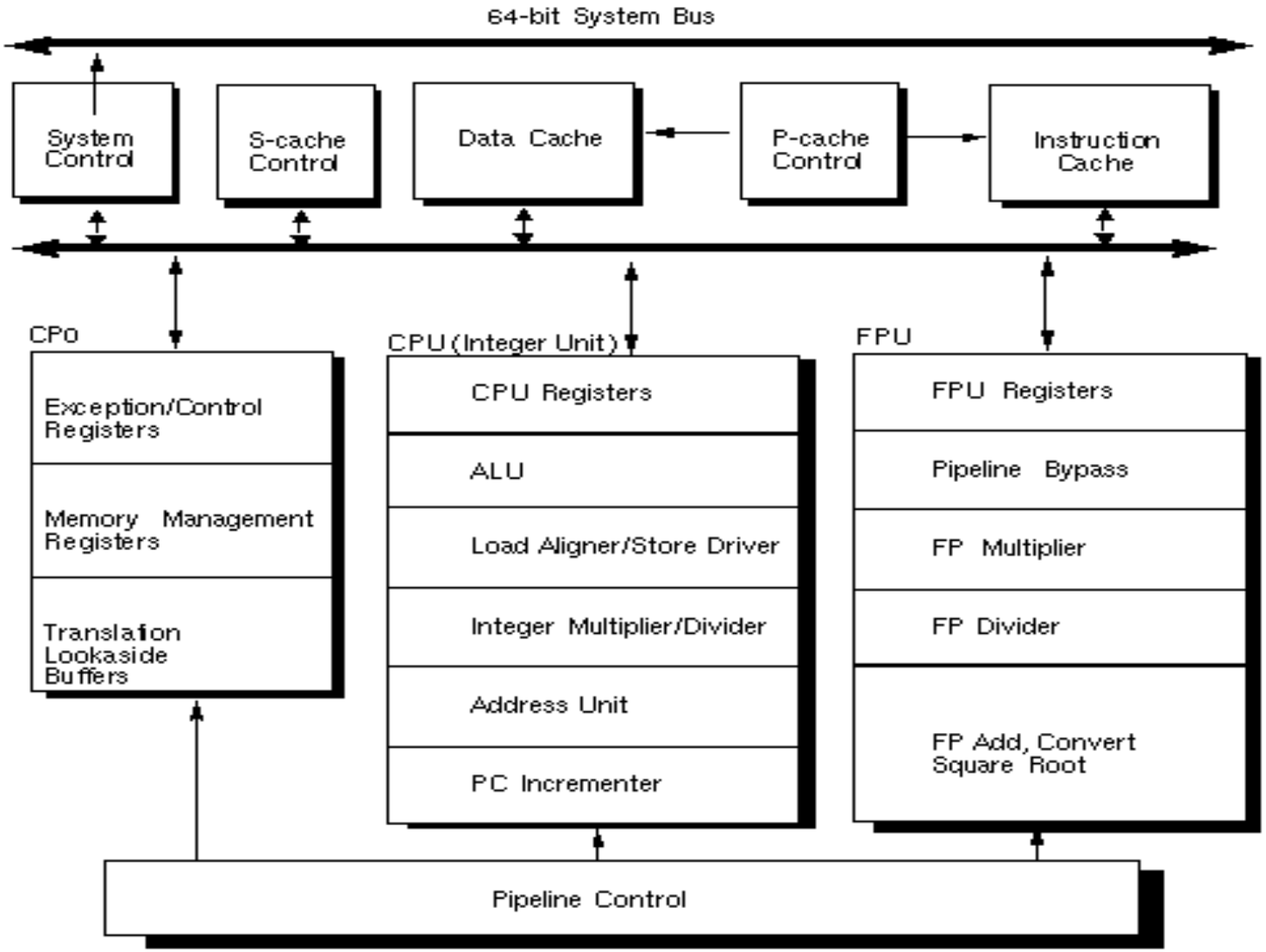
物理的なプロセッサの構成

- 数百万～数百億トランジスタ
- シリコンウェーハー
- 例: MIPS R4400
  - 1991年のプロセッサであり、1コアのみ
- 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





抽象化されたMIPS R4400の構成



高級言語・アセンブラ・機械語

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, ARM, 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, ARM, MIPS

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
  - 主な理由:単純化による効率化

- 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, 4は異なる)
- 機械語 (0/1パターン) はさすがに人間の解釈が難しいので、アセンブリで議論する
  - アセンブリの1行≒機械語の1命令

MIPSの主な機械語/アセンブリ命令の種類

- 算術演算命令: add, sub, addi…
- 論理演算命令: and, or…
- ロードストア(転送)命令: load, store…
- 制御命令
  - 無条件ジャンプ: jump
  - 条件分岐: beq, slt…

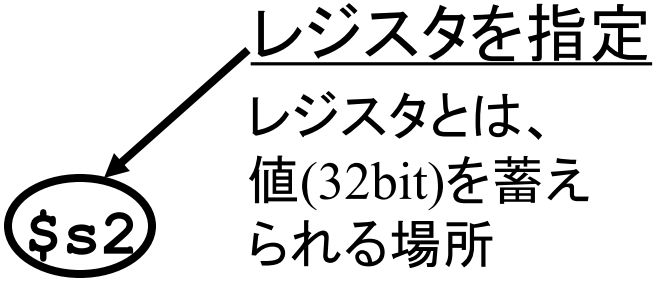
MIPS 算術命令

- 各算術命令は 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...

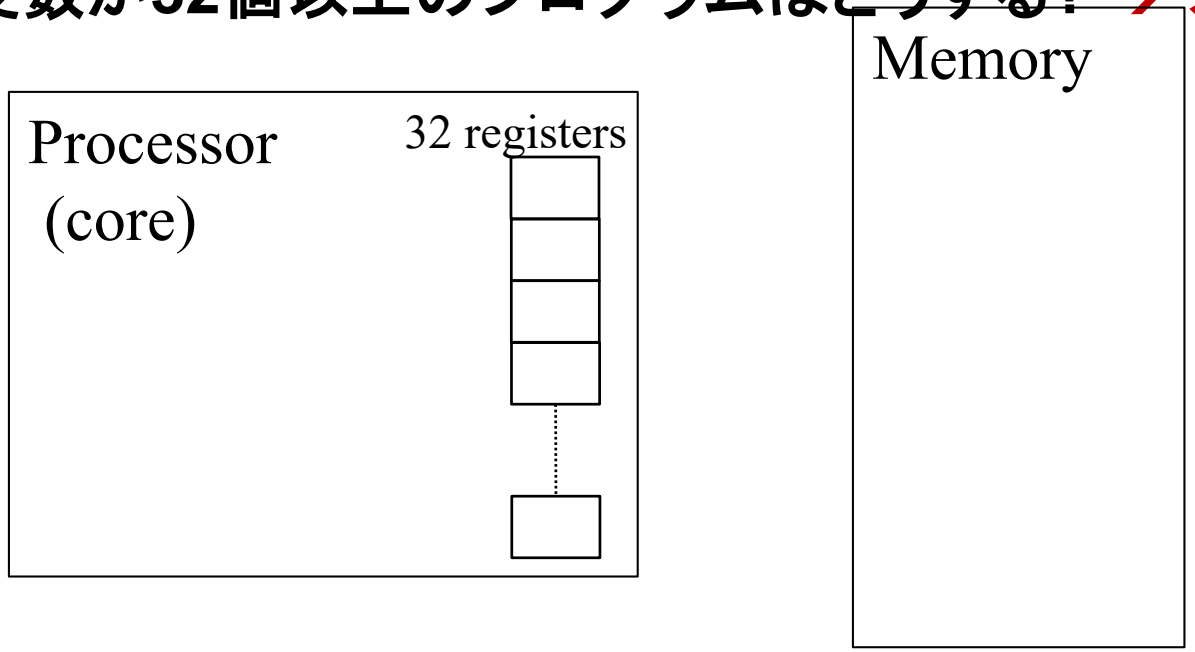
C code:
```
A = B + C + D;
E = F - A;
```

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個のどれか)でなくてはならない

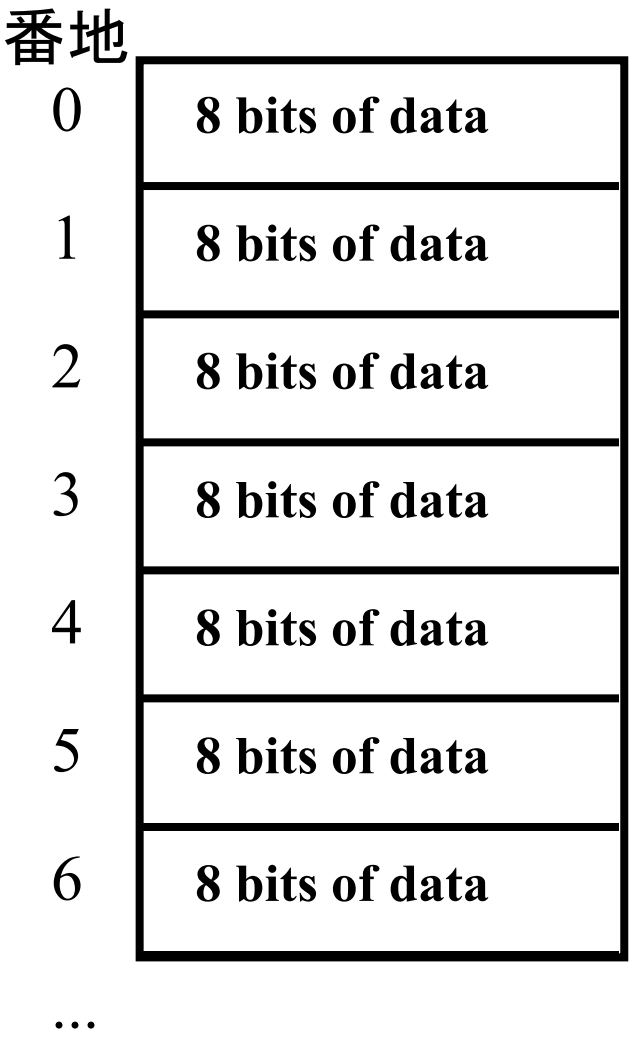
レジスタ とメモリ

- 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個
  - $s0…$s7, $t0…$t9, $a0…$a3, $zeroなど
- 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? → メモリを使う



メモリの構成

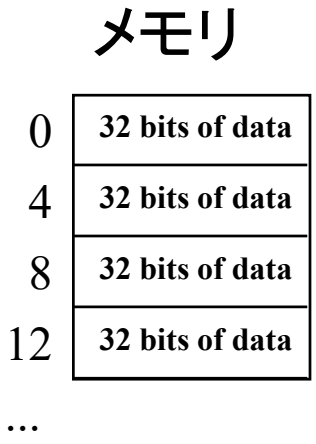
- 巨大な一次元の配列とみなせる。要素のそれぞれのメモリセルには番地が振ってある。
  - MIPSでは 32-bit = 約40億番地
- メモリのアドレスが配列へのインデックスとなる
- “Byte addressing (バイトアドレッシング)” メモリはバイト(8bit)単位で番地が振られる



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、多くのデータは “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.

レジスタも 32 bitのデータを保持



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。オペランドは2つ
- 例:

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

```
// $s3にAの番地が入っているとする
lw $t0, 32($s3)    // 32 = 4 * 8
add $t0, $s2, $t0
sw $t0, 40($s3)
```

MIPS code:

アドレス($s3+32)のメモリ内容を$t0へ転送

$t0の内容をアドレス($s3+40)へ転送

先ほどの例では、なぜ三命令必要？

?? add 40($s3), $s2, 32($s3) ➔ これはMIPS ISAで不可能！！

- 算術命令のオペランドはレジスタのみ
  - 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

- また、ロードストア命令では、$s1($s2) のようなアドレス指定も不可能
  32($t1) のような、ベースレジスタ + (定数)オフセットのみ可能
  Q: C言語のA[i] にアクセスしたい場合はどうする？

ここまでのまとめ

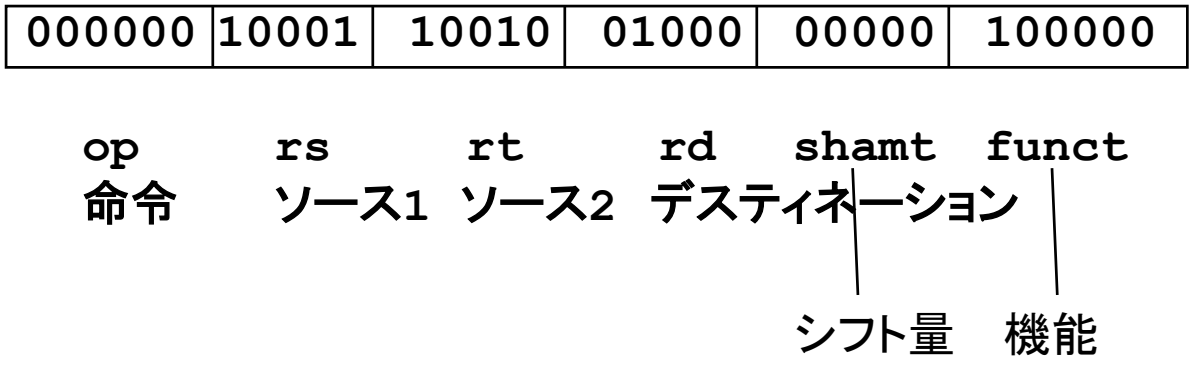
- **MIPS**
  — ロードストアの対象はワードだが、アドレッシングはバイト単位
  — 算術命令のオペランドはレジスタのみ
  - 典型的なRISC (Reduced Instruction Set Computer) アーキテクチャ
    (c.f. CISC (Complex Instruction Set Computer))
- **命令** **意味**

```
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

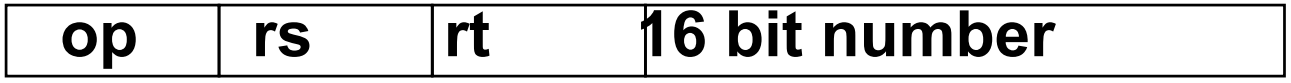
- それぞれの機械命令は、レジスタ同様、1ワード(32bit)長
  - Example: `add $t0, $s1, $s2`
  - レジスタには番号を割り振る, `$t0=9, $s1=17, $s2=18`
  - `c.f., CISC`アーキテクチャ→ 命令は可変長

- 命令フォーマットの例 (R形式):



機械語(続き)

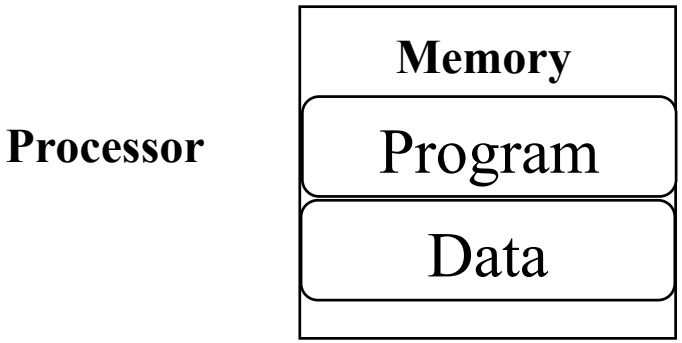
- Load-word (lw)と store-word(sw)命令を考えてみよう
  - 均一性の原則からは、どのようなデザインが芽生える?
    - R形式だけではメモリ番地の指定が難しい
  - 新原則:「良いデザインには妥協も必要」
- 新しい命令形式
  - データ転送のためのI形式
- 例: `lw $t0, 32($s2)`



- MIPS設計者はどこを妥協した?

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



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。

Q: メモリ内のプログラムとデータは区別されない。このことにより起こるセキュリティリスクは？

- 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

制御命令 (Control instruction)

- 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`　　ジャンプ先に26bit指定可能(J形式)
- 例　if--then--else:

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

```
    beq $s4, $s5, Lab1
    add $s3, $s4, $s5
    j Lab2
Lab1: sub $s3, $s4, $s5
Lab2: ...
```

- *Q: While文はどのように?*

```
while (i!=j)
    i=i+j;
```

```
Lab1: beq $s4, $s5, Lab2
      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):

```
                                  if  $s1 < $s2 then
                                     $t0 = 1
      slt $t0, $s1, $s2                 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: