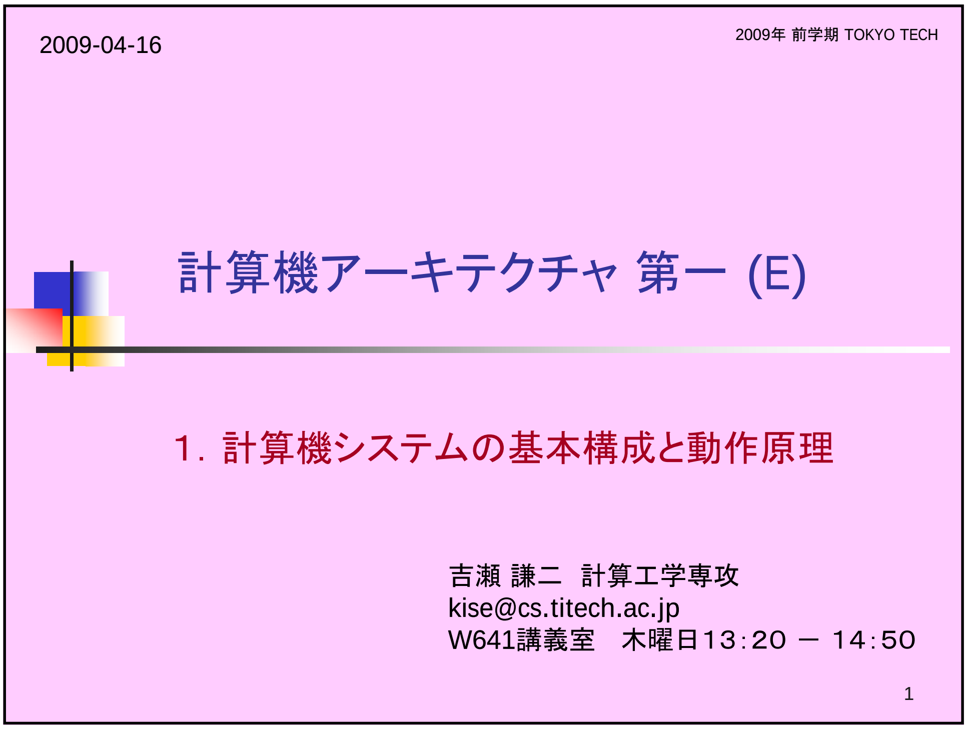


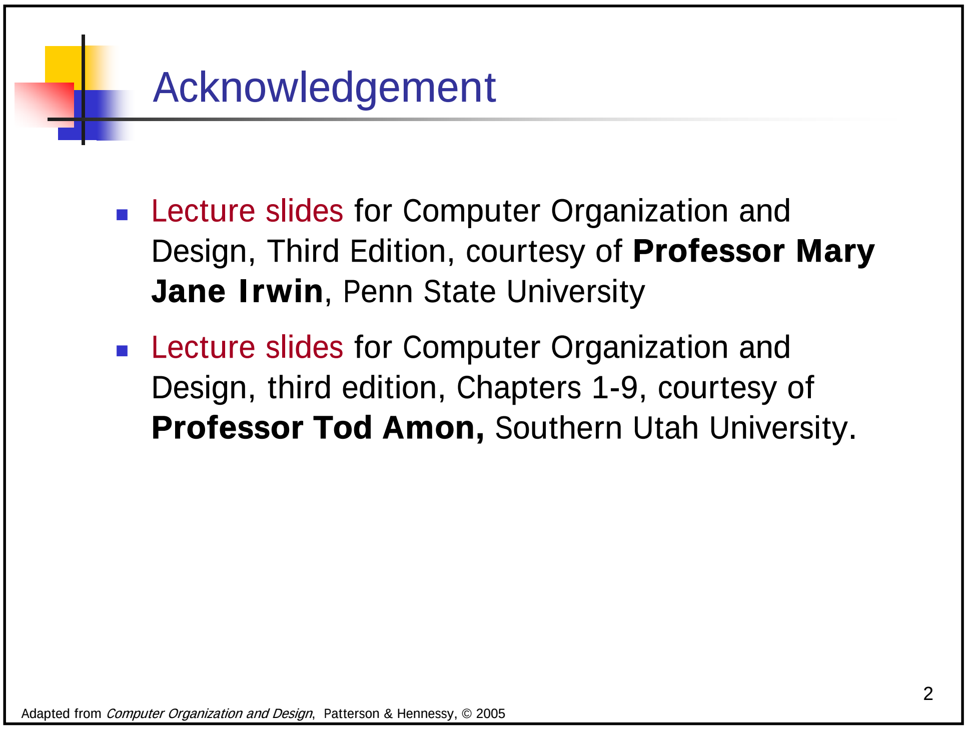
2009-04-16

2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 1. 計算機システムの基本構成と動作原理

吉瀬 謙二　計算工学専攻
kise@cs.titech.ac.jp
W641講義室　木曜日13:20 ー 14:50



## Acknowledgement

- Lecture slides for Computer Organization and Design, Third Edition, courtesy of **Professor Mary Jane Irwin**, Penn State University
- Lecture slides for Computer Organization and Design, third edition, Chapters 1-9, courtesy of **Professor Tod Amon,** Southern Utah University.

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

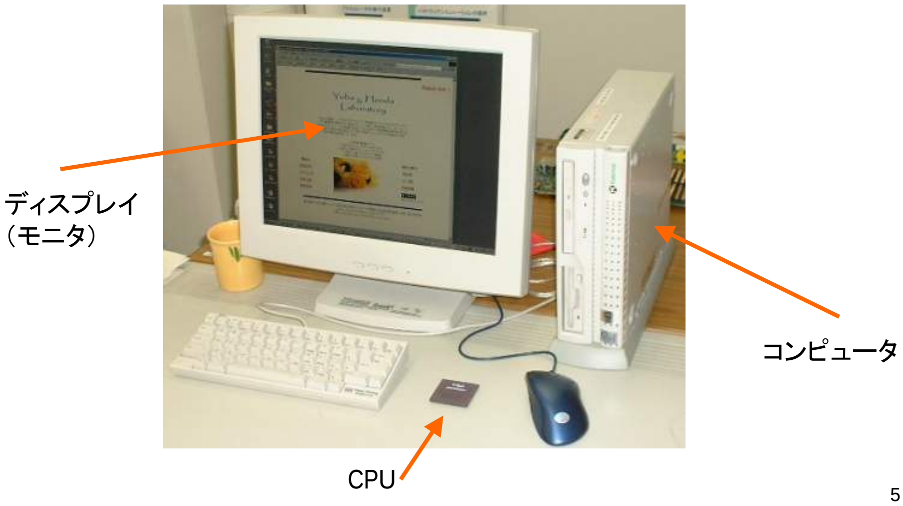
## 関連科目・履修条件等

- 4学期：計算機論理設計
  - 計算機を構成するプロセッサとその制御部に関し，具体構成と設計の原理を講義する．特に，レジスタトランスファ言語を用いて計算機の内部動作を記述し，簡単な計算機の設計を行う．
- 5学期：計算機アーキテクチャ第一
  - CPU を含め，メモリ，チャネル，入出力，通信制御，等の計算機システムを構成する各種装置について，その役割，動作原理について講義する．
- 6学期：計算機アーキテクチャ第二
  - 最新の計算機システムに採り入れられている高速プロセッサ制御方式，構成方式について述べ、これらの技術を駆使したパイプラインプロセッサ，スーパコンピュータ，超並列計算機，データフロー計算機，等の先端的なアーキテクチャについて講義する．
- 計算機アーキテクチャ特論（大学院）

# 計算機アーキテクチャ 第一 (E)

## 計算機システムの基本構成

## 計算機（デスクトップコンピュータ）

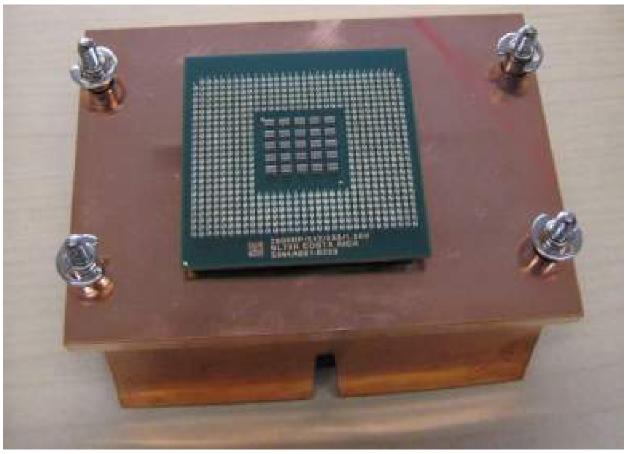


ディスプレイ
（モニタ）

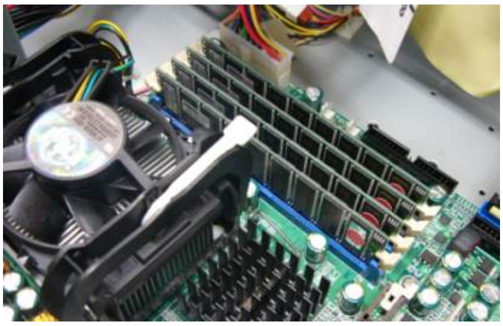
コンピュータ

CPU

## マイクロプロセッサ，CPU

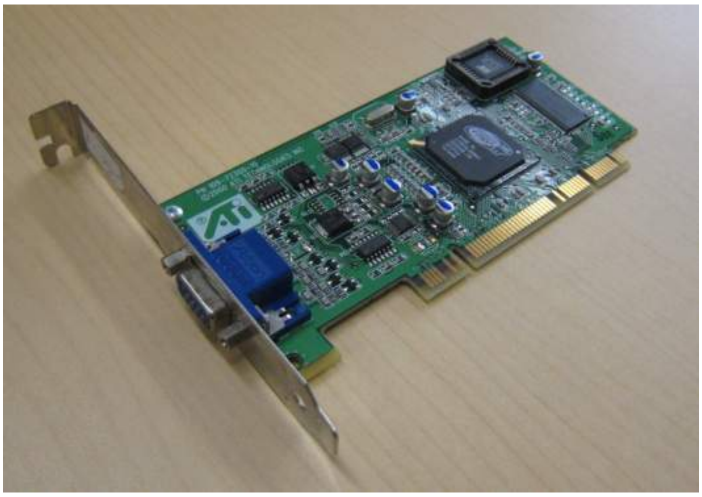


## メモリ
## DRAM (dynamic random access memory)



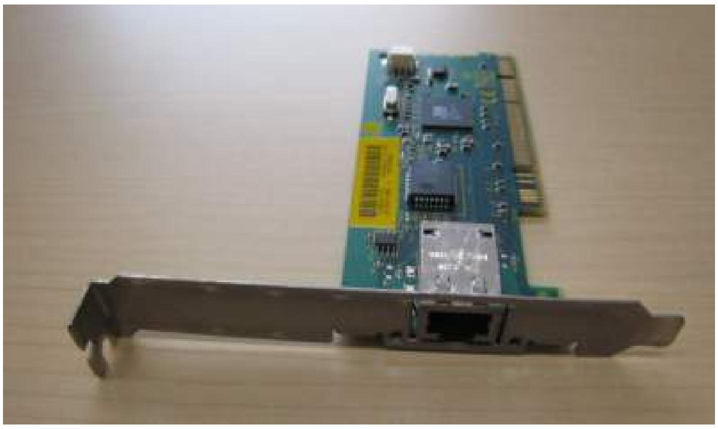
## ディスク，磁気ディスク

# グラフィックカード



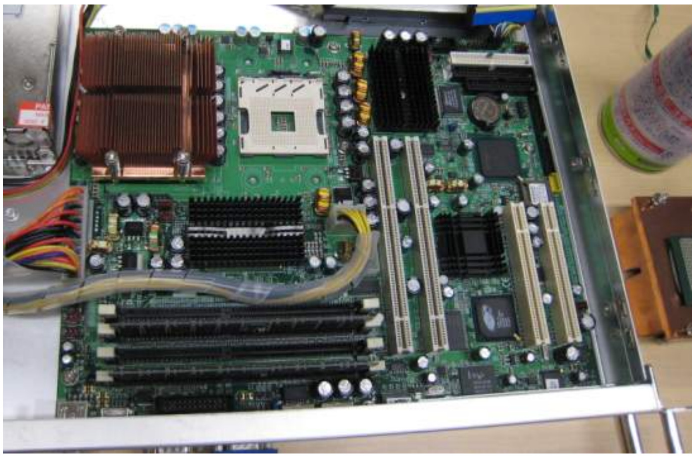
9

# ネットワークカード



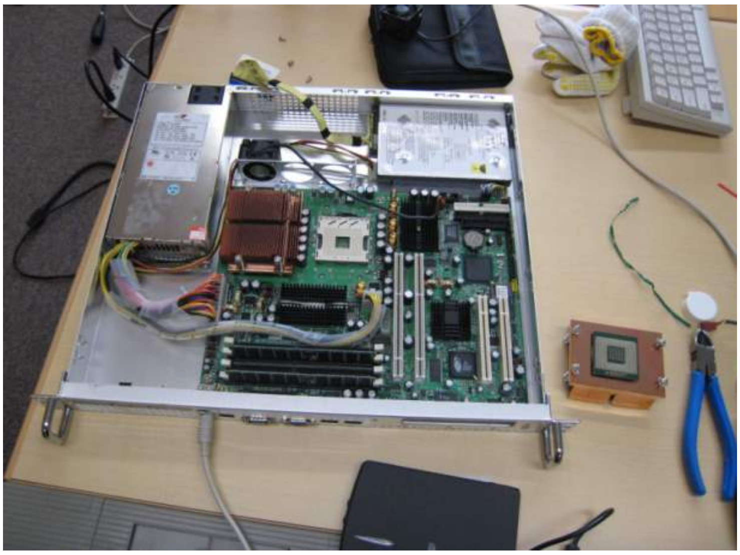
10

## マザーボード



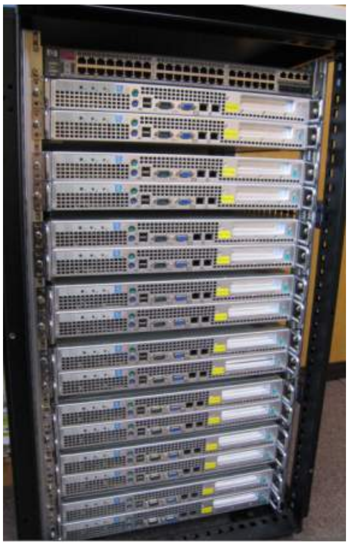
## など

## 計算機



13

## 補足: クラスタ型（並列）計算機



14

## 計算機アーキテクチャとは？

- アーキテクチャ
  Architecture
- 計算機アーキテクチャ
  Computer Architecture

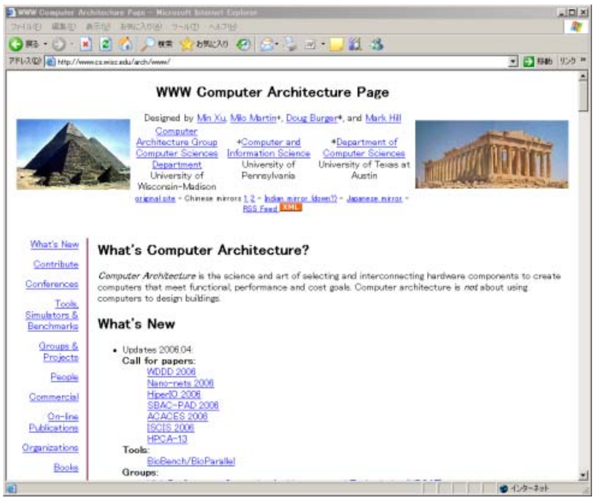
## アーキテクチャ（建築）
## Architecture

パルテノン神殿

世界最大のクフ王のピラミッド
1個約2.5tのブロックを 230～250万 個
積み重ねて造られている。

写真は計算機アーキテクチャのホームページから http://www.cs.wisc.edu/arch/www/

## 計算機アーキテクチャ
## Computer Architecture


WWW Computer Architecture Page – Microsoft Internet Explorer
ファイル(F) 編集(E) 表示(V) お気に入り(A) ツール(T) ヘルプ(H)
戻る
検索
お気に入り
アドレス(D)
http://www.cs.wisc.edu/arch/www/
移動
リンク
インターネット

### WWW Computer Architecture Page

Designed by Min Xu, Milo Martin+, Doug Burger*, and Mark Hill

Computer Architecture Group
Computer Sciences Department
University of Wisconsin-Madison

+Computer and Information Science
University of Pennsylvania

*Department of Computer Sciences
University of Texas at Austin

original site - Chinese mirrors 1 2 - Indian mirror (down?) - Japanese mirror - RSS Feed XML

What's New
Contribute
Conferences
Tools, Simulators & Benchmarks
Groups & Projects
People
Commercial
On-line Publications
Organizations
Books

#### What's Computer Architecture?

*Computer Architecture* is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. Computer architecture is *not* about using computers to design buildings.

#### What's New

- Updates 2006.04:
  **Call for papers:**
  WDDD 2006
  Nano-nets 2006
  HiperIO 2006
  SBAC-PAD 2006
  ACACES 2006
  ISCIS 2006
  HPCA-13
  **Tools:**
  BioBench/BioParallel
  **Groups:**

17

## 計算機アーキテクチャ

### What's Computer Architecture?

*Computer Architecture* is the science and art of selecting and interconnecting hardware components to create computers that meet functional, performance and cost goals. Computer architecture is *not* about using computers to design buildings.

計算機アーキテクチャのホームページから http://www.cs.wisc.edu/arch/www/

18

# 計算機

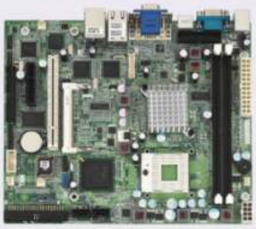
19

# 計算機アーキテクチャ，ブロック図

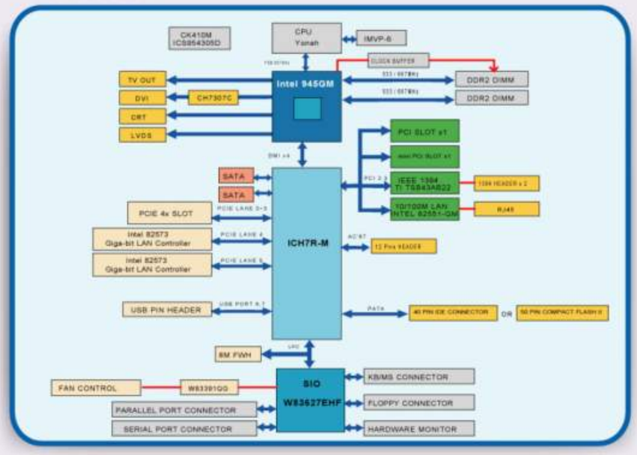
**Tomcat i945GM**

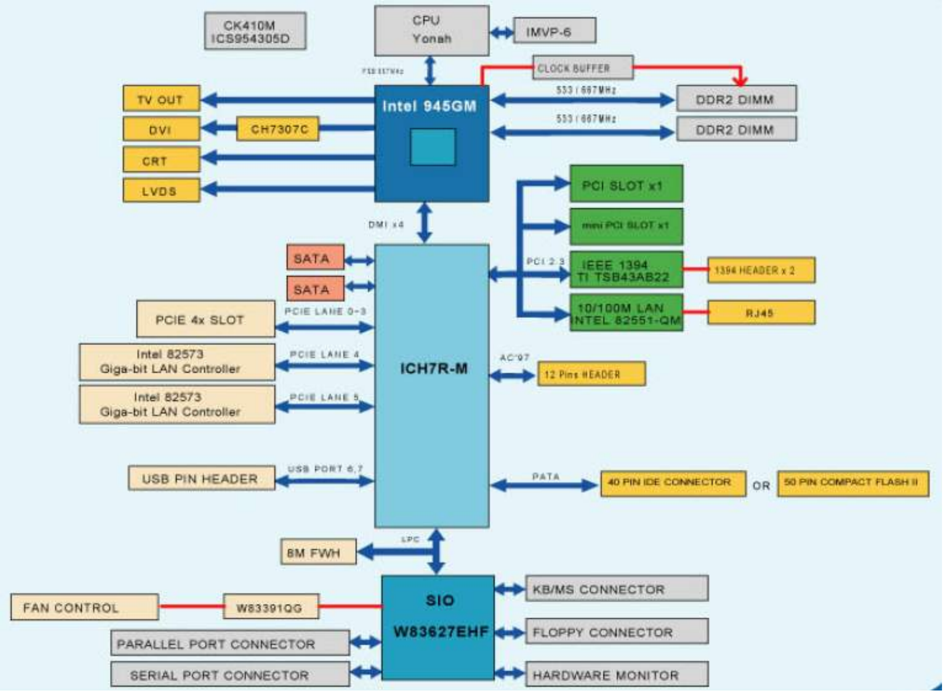
S3095G3NR

▶ TOP View



▼ Block Diagram



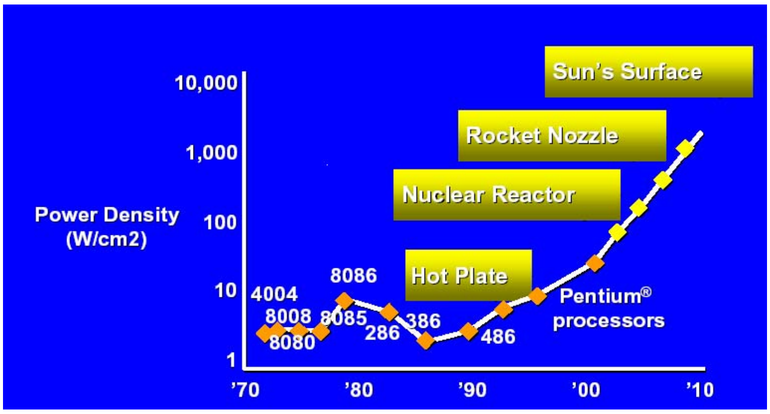


## 計算機アーキテクチャへの要求

- 速度
- 消費電力
- 発熱
- 音
- 価格
- 安定性，信頼性

# 増加を続けるプロセッサのエネルギー消費

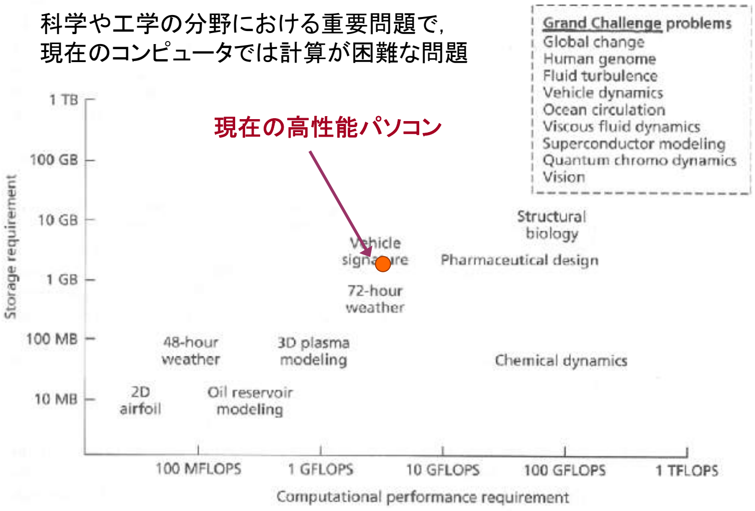
このままでは，プロセッサの熱は核反応，ロケットの噴射口，太陽の表面のエネルギー消費に近づいていく.



出典：Gelsinger's Slide from ISSCC 2001

# 人類にとって重要な問題 グランドチャレンジ

科学や工学の分野における重要問題で，
現在のコンピュータでは計算が困難な問題



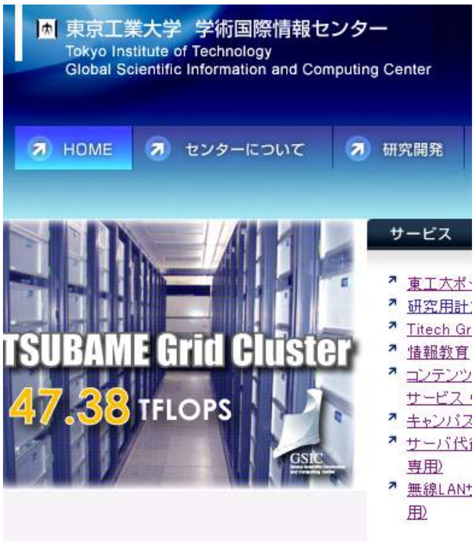
現在の高性能パソコン

出典：David E. Culler, Jaswinder Pal Singh, Parallel Computer Architecture (p.7)

## スーパーコンピュータのダウンサイジング

Titech
TSUBAME

~80+ racks
350m2 floor area
1.2 MW (peak)



東京工業大学 学術国際情報センター
Tokyo Institute of Technology
Global Scientific Information and Computing Center

HOME センターについて 研究開発

TSUBAME Grid Cluster
47.38 TFLOPS

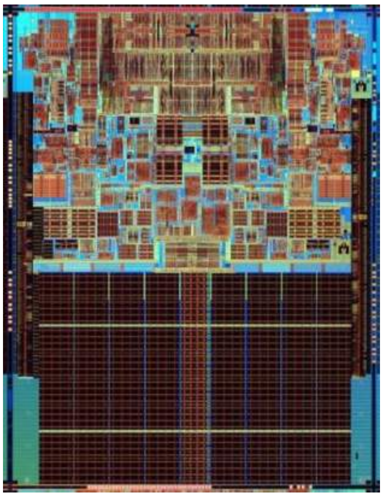
GSIC

サービス

- 東工大ポ
- 研究用計
- Titech Gr
- 情報教育
- コンテンツ
- サービス
- キャンパス
- サーバ代行専用)
- 無線LAN用)

## 先端マイクロプロセッサ Intel Core 2 Duo

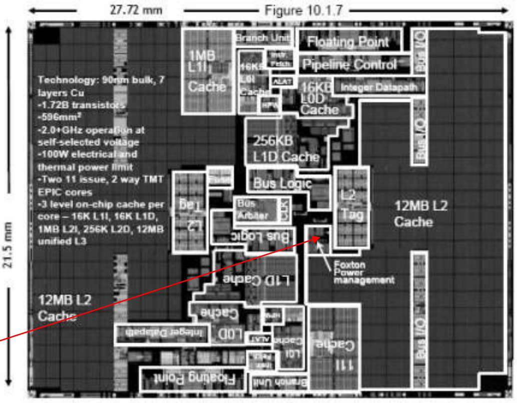
- (2006年7月発表)
  - 65nmプロセス
  - 143mm$^2$
  - 291M トランジスタ
  - 65W
- Core Micro Architecture
  - Intelligent power capability
  - Micro-Fusion
    - RISC vs CISC
  - Advanced Smart Cache



Intel Developer Forum

## 先端マイクロプロセッサ Intel Montecito

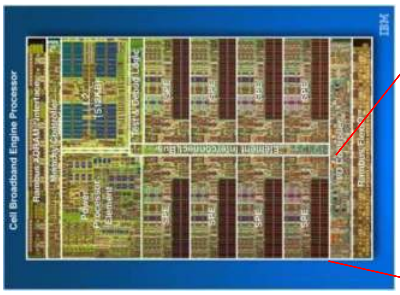
- ２個のEPICプロセッサコア
- 1MB L2, 12MB L2キャッシュ
- EPICコアは11 issue, 2way Temporal MT
- 初の10億超トランジスタ
  - 1.72BTrs
  - 21.5mm x 27.7mm
  - 90nm
  - 100W
- パワー制御用の専用チップFoxtonを搭載



Source: ISSCC 2005 papers

## 先端マイクロプロセッサ
## Cell Broadband Engine

- ヘテロジニアス チップマルチプロセッサ
  - PowerPC Processor Element (PPE) １個
  - Synergistic Processor Element (SPE) ８個



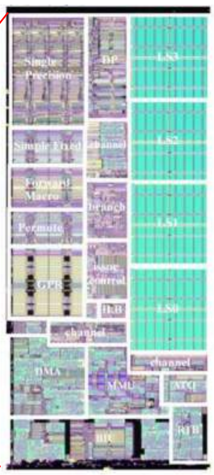


Diagram created by IBM to promote the CBEP, ©2005
WIKIPEDIAより

# 先端マイクロプロセッサ **SUN Rock**

- A Third-Generation 65nm 16-Core 32-Thread Plus 32-Scout-Thread CMT SPARC® Processor

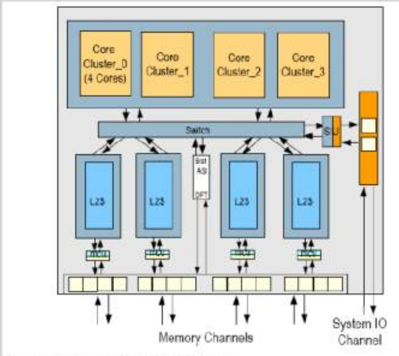


Figure 4.1.1: Top level logical block diagram.

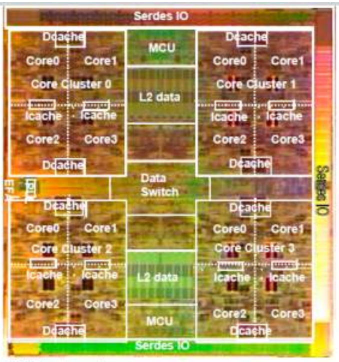
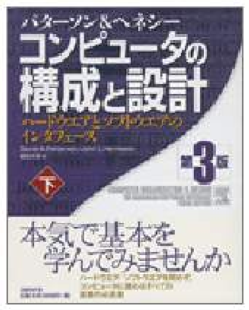


Figure 4.1.3: Chip micrograph.

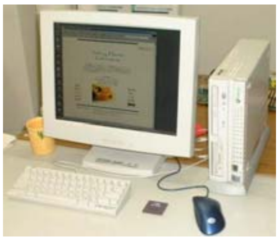
Source: ISSCC 2008 papers

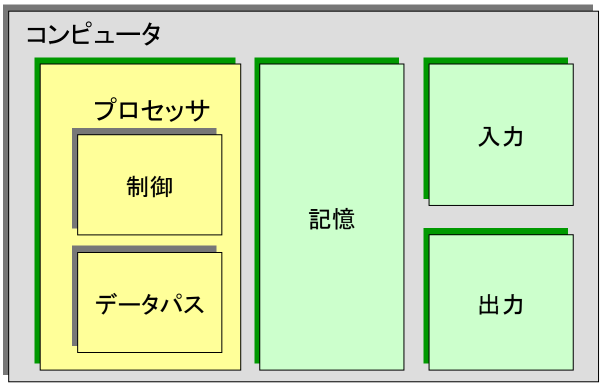
# 計算機アーキテクチャ 第一 (E)


パターソン&ヘネシー
コンピュータの構成と設計
ハードウエアとソフトウエアのインタフェース
第3版
下
本気で基本を学んでみませんか

計算機システムの動作原理

## コンピュータ(ハードウェア)の古典的な要素





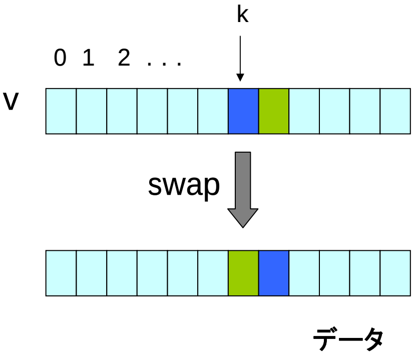
プロセッサは記憶装置から命令とデータを取り出す。入力装置はデータを記憶装置に書き込む。出力装置は記憶装置からデータを読みだす。制御装置は、データパス、記憶装置、入力装置、そして出力装置の動作を指定する信号を送る。

出典: パターソン & ヘネシー、コンピュータの構成と設計

## 高水準言語からハードウェアの言語へ

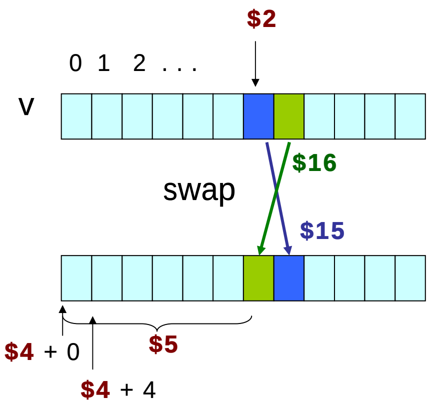
```
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

C言語で記述したプログラム



## 高水準言語からハードウェアの言語へ

```
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
```

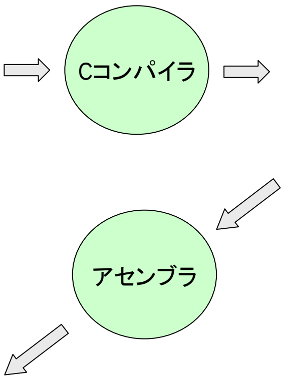


MIPSのアセンブリ言語に変換されたプログラム

## 高水準言語からハードウェアの言語へ

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

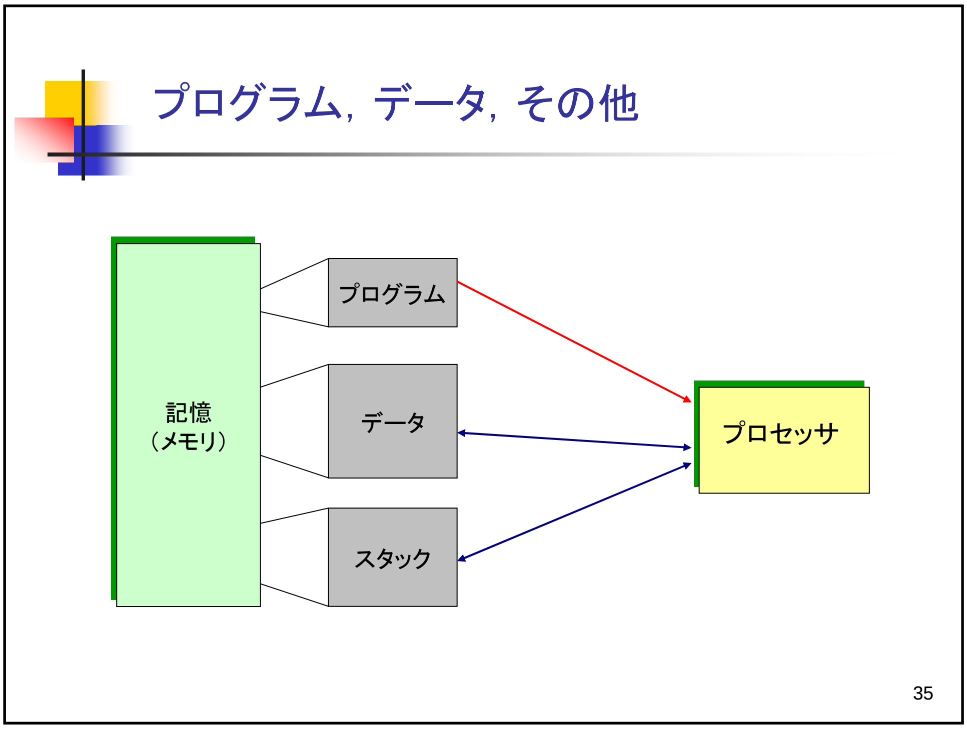
C言語で記述したプログラム



```
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
```

MIPSのアセンブリ言語に
変換されたプログラム

機械語に落とされたプログラム(機械命令の集まり)



## プログラム，データ，その他

記憶
（メモリ）

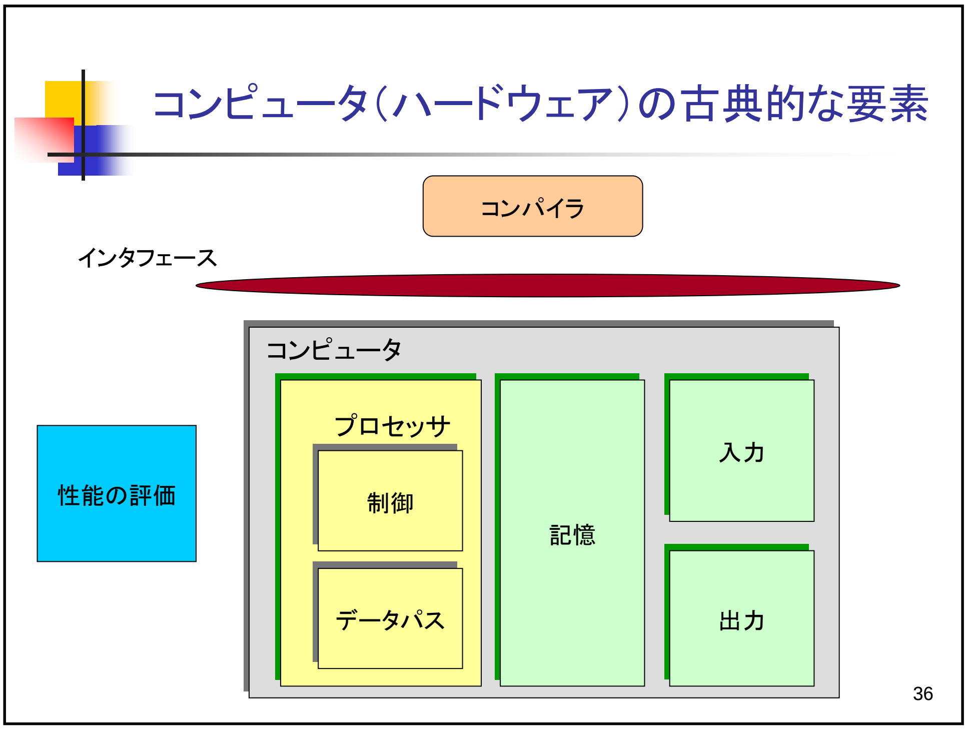
プログラム

データ

スタック

プロセッサ

35



## コンピュータ（ハードウェア）の古典的な要素

コンパイラ

インタフェース

性能の評価

コンピュータ

プロセッサ

制御

データパス

記憶

入力

出力

36

# 講義項目

- **計算機システムの基本構成と動作原理**
- データ形式，命令形式，アドレス指定形式
- **メモリ1**：半導体メモリシステム，ファイルメモリシステム
- **メモリ2**：記憶階層，キャッシュシステム
- **メモリ3**：仮想記憶システム（セグメンテーション，ページング，等）
- **メモリ4**：主記憶とファイルメモリの管理，多重仮想記憶，記憶保護
- **割り込み1**：割り込みの必要性，割り込みの種類
- **割り込み2**：割り込み処理の流れ
- **入出力制御1**：チャネル，チャネルプログラム方式
- **入出力制御2**：入出力動作の流れ，チャネル動作の効率化
- **入出力制御3**：チャネルの種類，通信制御

レポートと期末試験により評価、今年度はちょっと修正

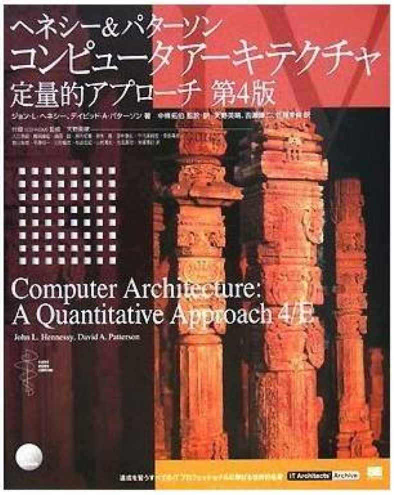
# 参考書

- **コンピュータの構成と設計 第3版**、パターソン＆ヘネシー（成田光彰訳）、日経BP社、2006
- コンピュータアーキテクチャ 定量的アプローチ 第4版
  翔泳社, 2008
- コンピュータアーキテクチャ,
  村岡 洋一 著， 近代科学社, 1989
- 計算機システム工学,
  富田 真治, 村上 和彰 著, 昭晃堂, 1988
- コンピュータハードウヱア,
  富田 真治, 中島 浩 著, 昭晃堂, 1995
- 計算機アーキテクチャ,
  橋本 昭洋 著, 昭晃堂, 1995

パターソン&ヘネシー
コンピュータの構成と設計
ハードウエアとソフトウエアのインタフェース
David A.Patterson/John L.Hennessy
第3版
上
下
日経BP社

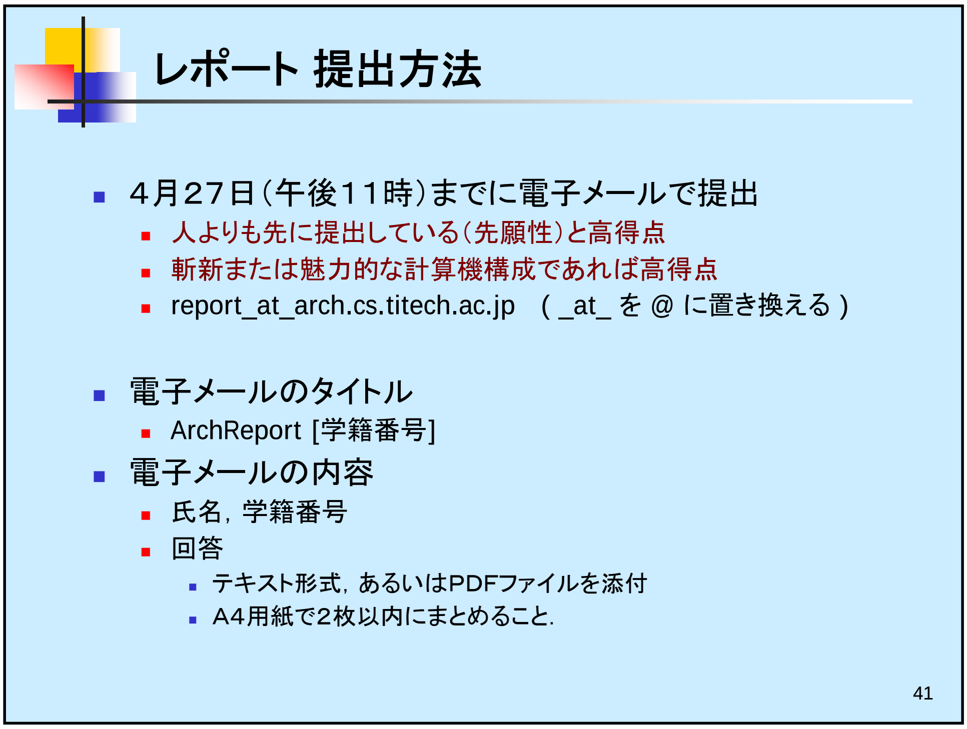
# 参考書

- **コンピュータの構成と設計 第3版、パターソン＆ヘネシー（成田光彰訳）、日経BP社、2006**
- **コンピュータアーキテクチャ 定量的アプローチ 第4版**
  翔泳社, 2008
- コンピュータアーキテクチャ,
  村岡 洋一 著, 近代科学社, 1989
- 計算機システム工学,
  富田 真治, 村上 和彰 著, 昭晃堂, 1988
- コンピュータハードウエア,
  富田 真治, 中島 浩 著, 昭晃堂, 1995
- 計算機アーキテクチャ,
  橋本 昭洋 著, 昭晃堂, 1995


ヘネシー&パターソン
コンピュータアーキテクチャ
定量的アプローチ 第4版
Computer Architecture:
A Quantitative Approach 4/E
John L. Hennessy, David A. Patterson

# レポート 問題

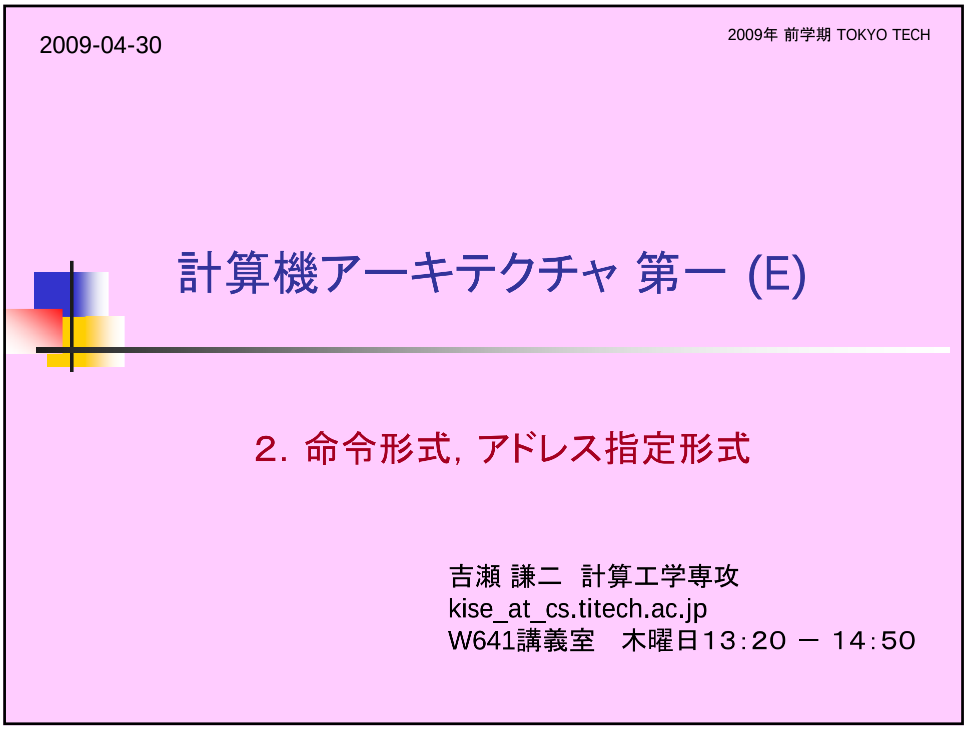
1. 部品を組み合わせて計算機（パソコン）を自作したい.
   適切な（個人の主観でかまわない）部品と構成を提案せよ.
   提案構成できちんと動作することを説明せよ.
   また, 構成の特徴を魅力的に説明せよ.
   1. 予算は5万円以内とする.
      それぞれの部品の価格をWebにて調査すること.
   2. オペレーティングシステムとして Linux が動作すること.
      利用目的とその意義を明確にすること.
   3. 計算機本体のみとする. ディスプレイやキーボードは不要.
   4. レポートはA4用紙 2枚以内にまとめること.



## レポート 提出方法

- ４月27日（午後１１時）までに電子メールで提出
  - 人よりも先に提出している（先願性）と高得点
  - 斬新または魅力的な計算機構成であれば高得点
  - report_at_arch.cs.titech.ac.jp　( _at_ を @ に置き換える )
- 電子メールのタイトル
  - ArchReport [学籍番号]
- 電子メールの内容
  - 氏名，学籍番号
  - 回答
    - テキスト形式，あるいはPDFファイルを添付
    - A4用紙で2枚以内にまとめること.

41



2009-04-30

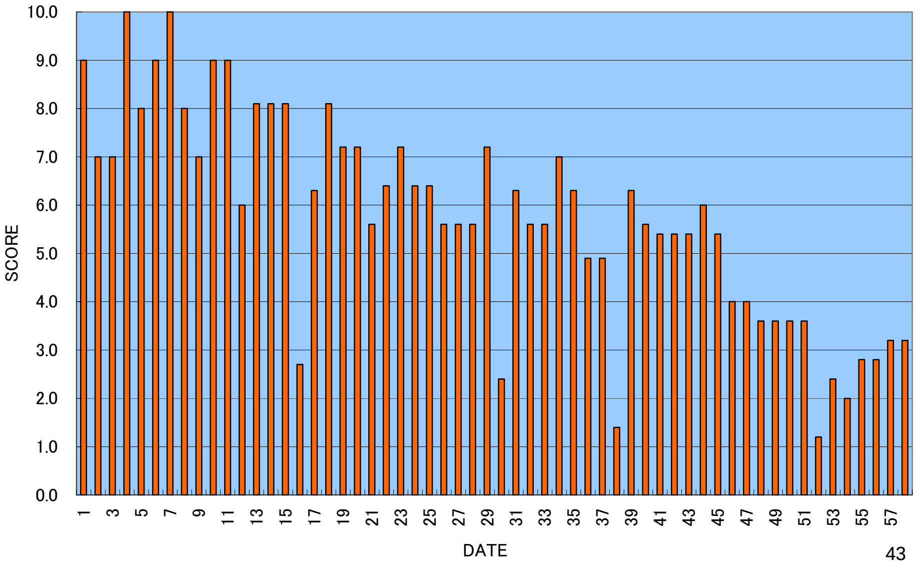
2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 2. 命令形式，アドレス指定形式

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

## 第1回 レポートの提出状況

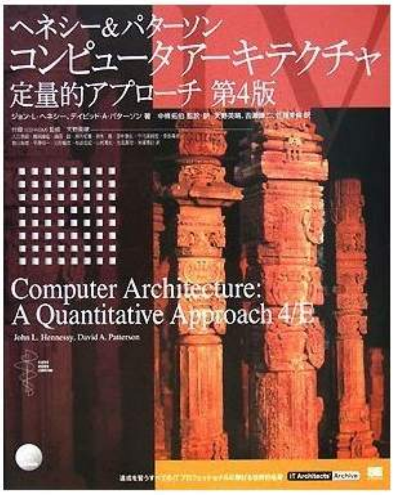


## 参考書（読んでください）

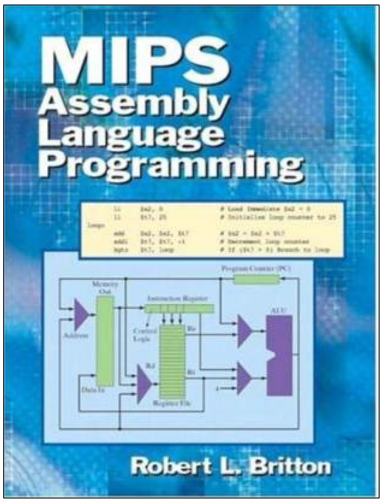
- **コンピュータの構成と設計 第3版**、パターソン＆ヘネシー（成田光彰　訳）、日経BP社、2006
- コンピュータアーキテクチャ 定量的アプローチ 第4版<br>翔泳社, 2008
- コンピュータアーキテクチャ,<br>村岡 洋一 著, 近代科学社, 1989
- 計算機システム工学,<br>富田 真治, 村上 和彰 著, 昭晃堂, 1988
- コンピュータハードウエア,<br>富田 真治, 中島 浩 著, 昭晃堂, 1995
- 計算機アーキテクチャ,<br>橋本 昭洋 著, 昭晃堂, 1995

## 参考書（大学院生がターゲット，興味があれば）

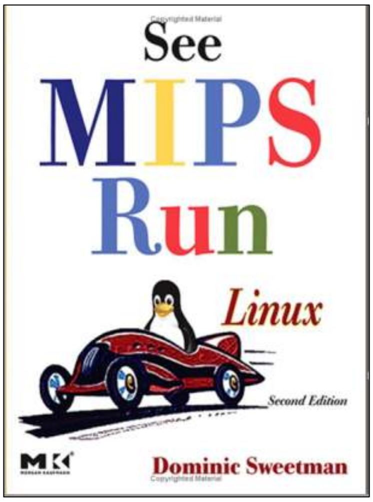
- コンピュータの構成と設計 第3版、パターソン＆ヘネシー（成田光彰　訳）、日経BP社、2006
- **コンピュータアーキテクチャ 定量的アプローチ 第4版**
  翔泳社, 2008
- コンピュータアーキテクチャ,
  村岡 洋一 著，近代科学社，1989
- 計算機システム工学,
  富田 真治，村上 和彰 著，昭晃堂，1988
- コンピュータハードウエア,
  富田 真治，中島 浩 著，昭晃堂，1995
- 計算機アーキテクチャ,
  橋本 昭洋 著，昭晃堂，1995


ヘネシー&パターソン
コンピュータアーキテクチャ
定量的アプローチ 第4版
Computer Architecture:
A Quantitative Approach 4/E
John L. Hennessy, David A. Patterson

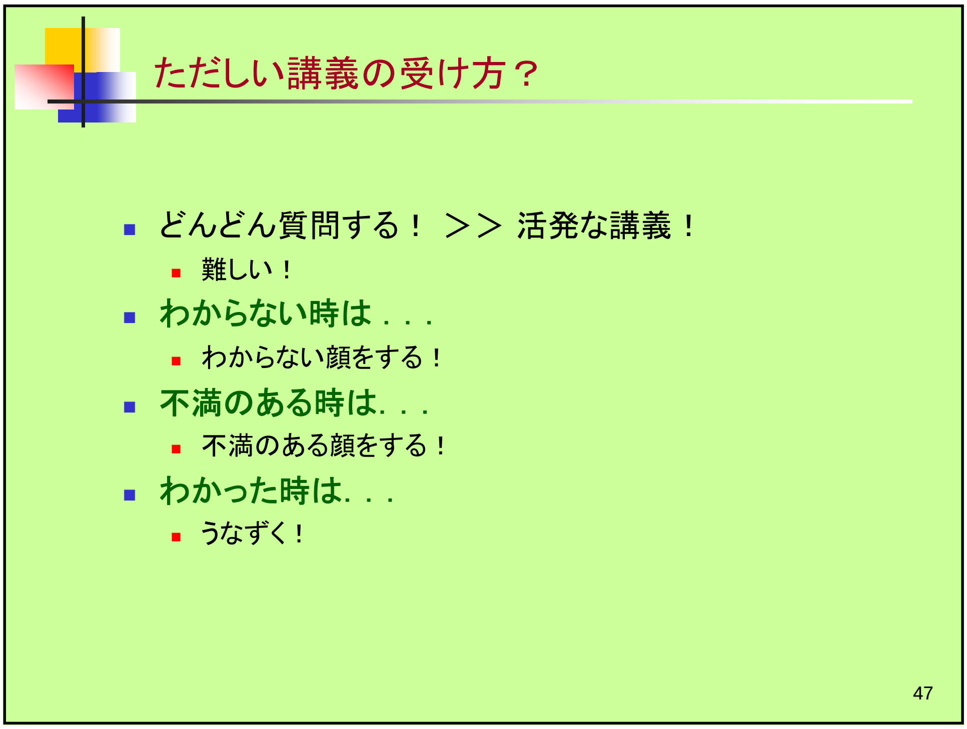
## 参考書（アセンブラに興味があれば）



MIPSのアセンブラがよくわかります．面白いです．



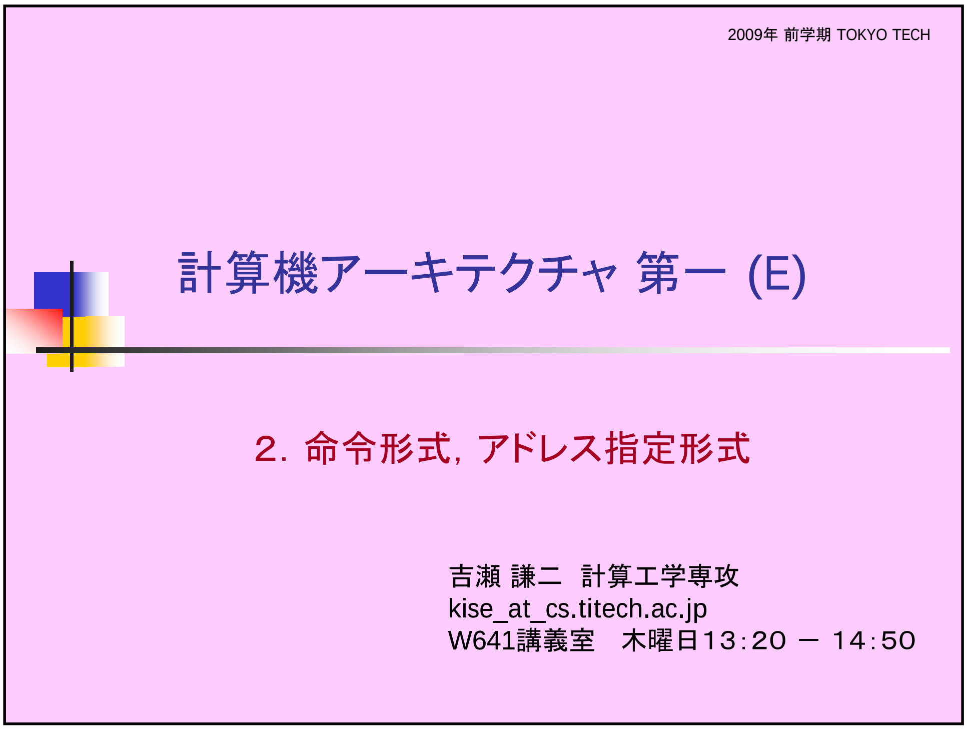
MIPSとLinuxの間がわかります．お勧め．



## ただしい講義の受け方？

- どんどん質問する！ ＞＞ 活発な講義！
  - 難しい！
- わからない時は．．．
  - わからない顔をする！
- 不満のある時は．．．
  - 不満のある顔をする！
- わかった時は．．．
  - うなずく！

47



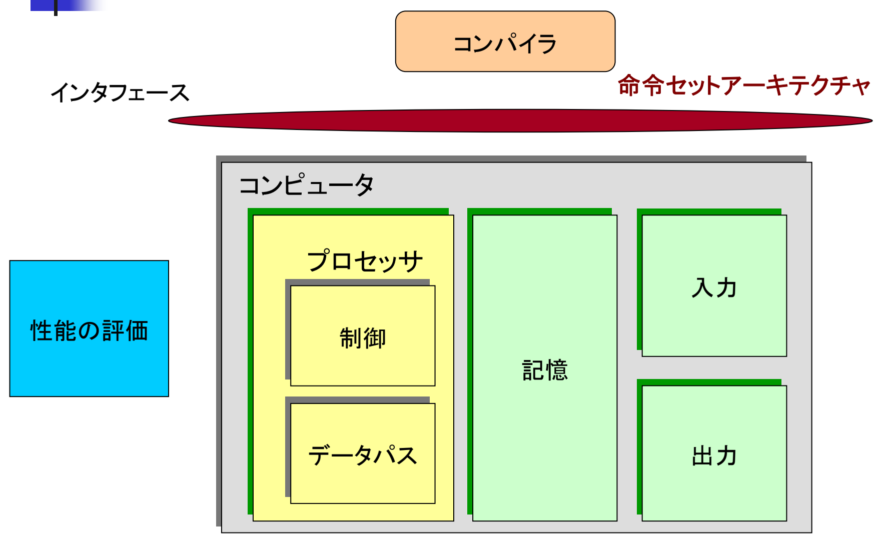
2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 2．命令形式，アドレス指定形式

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13：20 － 14：50

# コンピュータ（ハードウェア）の古典的な要素



コンパイラ

インタフェース

命令セットアーキテクチャ

性能の評価

コンピュータ

プロセッサ

制御

データパス

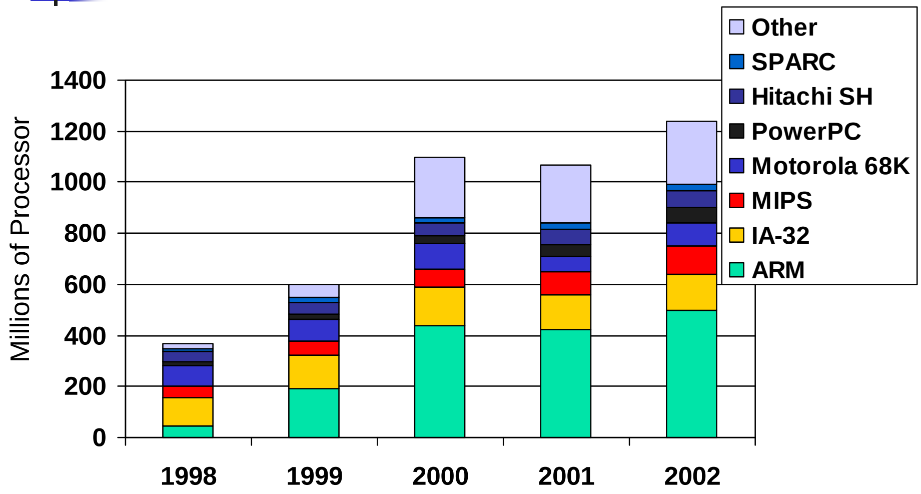
記憶

入力

出力

49

# Instruction Set Architecture (ISA) Type Sales

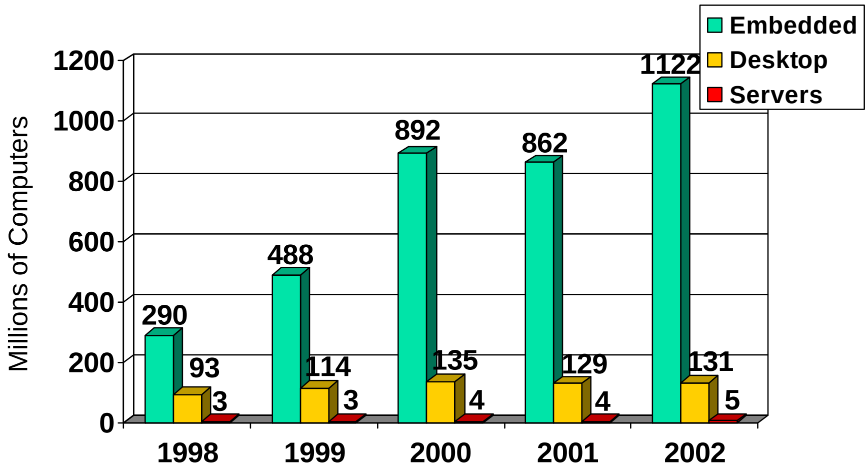


PowerPoint “comic” bar chart with approximate values (see text for correct values)

50

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Where is the Market?



Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

51

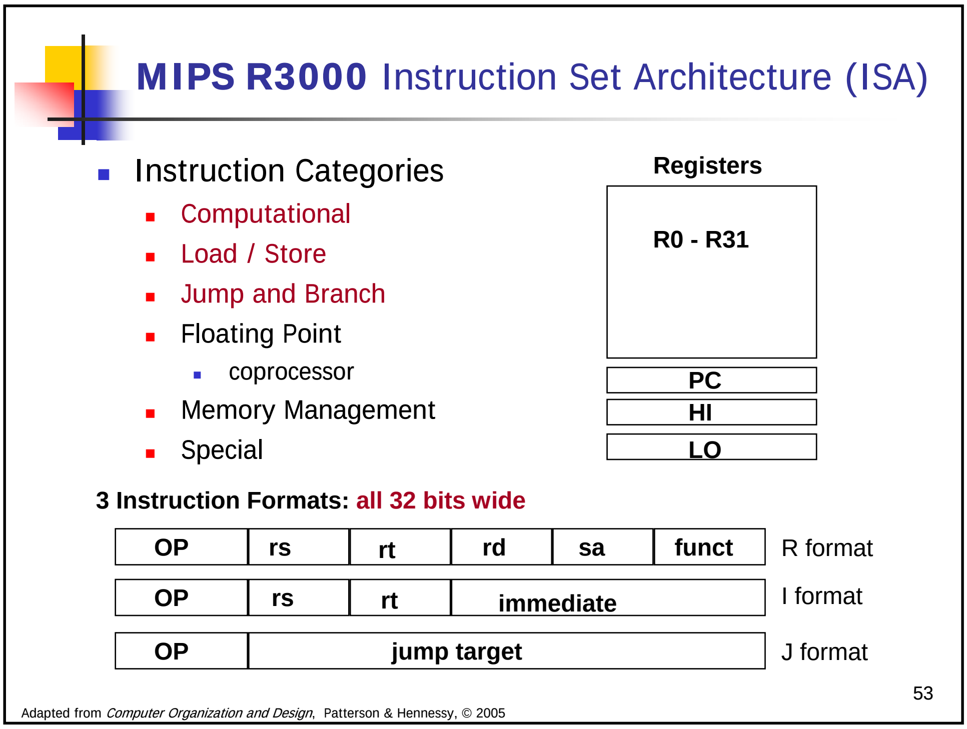
## **RISC** - Reduced Instruction Set Computer

- **RISC philosophy** ⟷ **CISC**
  **Complex Instruction Set Computer**
  - fixed instruction lengths
  - load-store instruction sets
  - limited addressing modes
  - limited operations
- Sun SPARC, HP PA-RISC, IBM PowerPC, Compaq Alpha, **MIPS**, …

*Design goals: speed, cost (design, fabrication, test, packaging), size, power consumption, reliability, memory space (embedded systems)*

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

52



## MIPS R3000 Instruction Set Architecture (ISA)

- Instruction Categories
  - Computational
  - Load / Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

**Registers**

R0 - R31

PC

HI

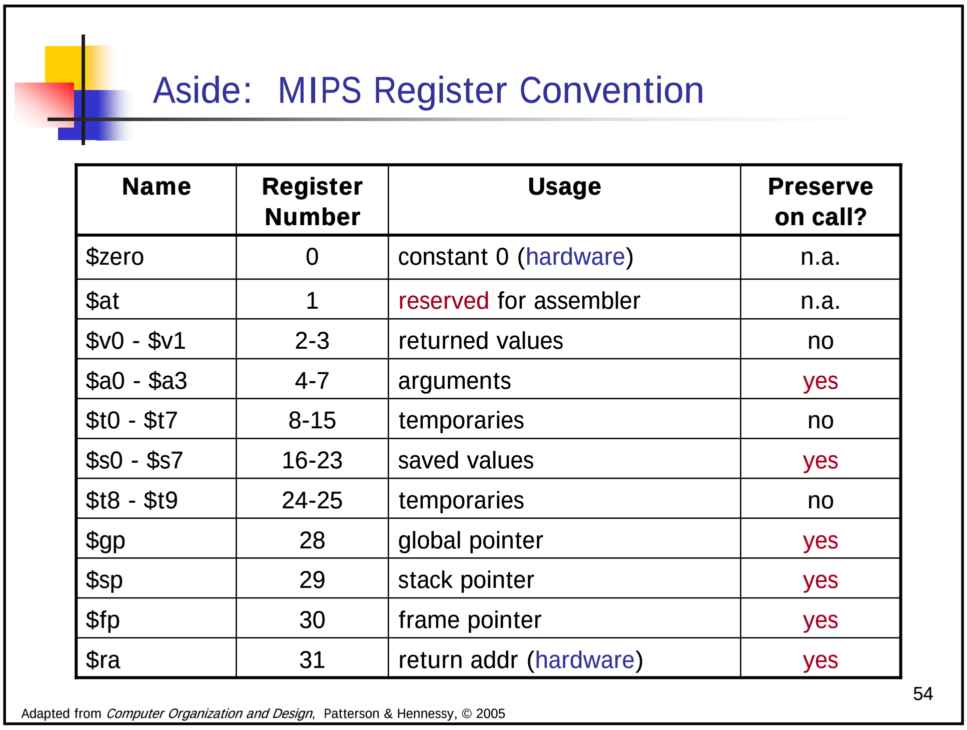
LO

**3 Instruction Formats: all 32 bits wide**

| | | | | | | |
|---|---|---|---|---|---|---|
| OP | rs | rt | rd | sa | funct | R format |
| OP | rs | rt | immediate | | | I format |
| OP | jump target | | | | | J format |

53

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

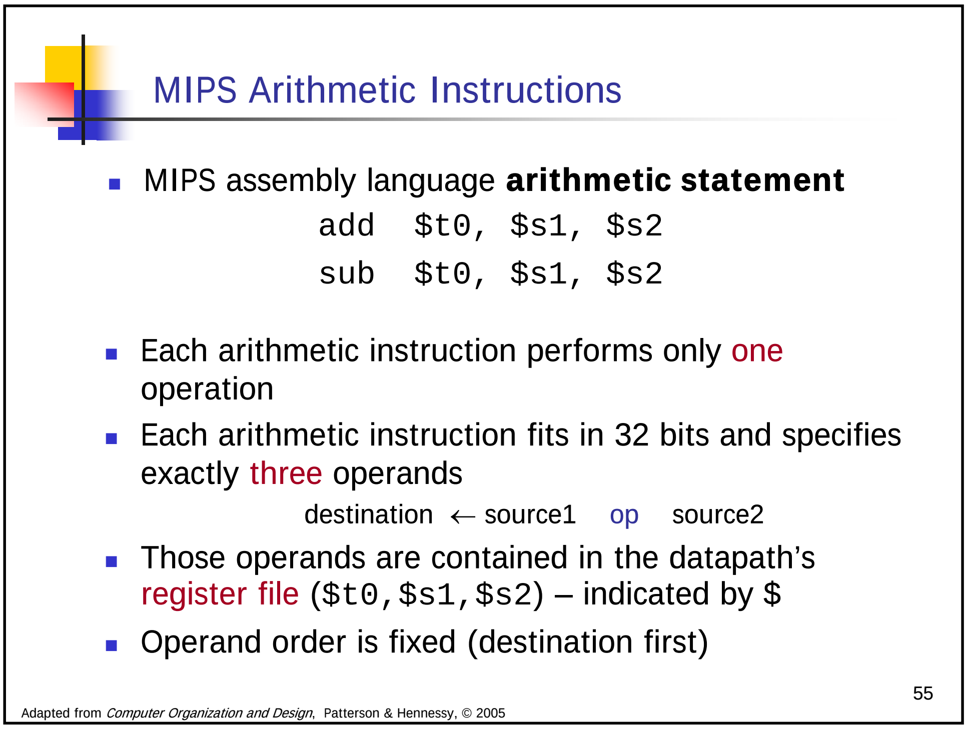


## Aside: MIPS Register Convention

| Name | Register Number | Usage | Preserve on call? |
|---|---|---|---|
| $zero | 0 | constant 0 (hardware) | n.a. |
| $at | 1 | reserved for assembler | n.a. |
| $v0 - $v1 | 2-3 | returned values | no |
| $a0 - $a3 | 4-7 | arguments | yes |
| $t0 - $t7 | 8-15 | temporaries | no |
| $s0 - $s7 | 16-23 | saved values | yes |
| $t8 - $t9 | 24-25 | temporaries | no |
| $gp | 28 | global pointer | yes |
| $sp | 29 | stack pointer | yes |
| $fp | 30 | frame pointer | yes |
| $ra | 31 | return addr (hardware) | yes |

54

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## MIPS Arithmetic Instructions

- MIPS assembly language **arithmetic statement**

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

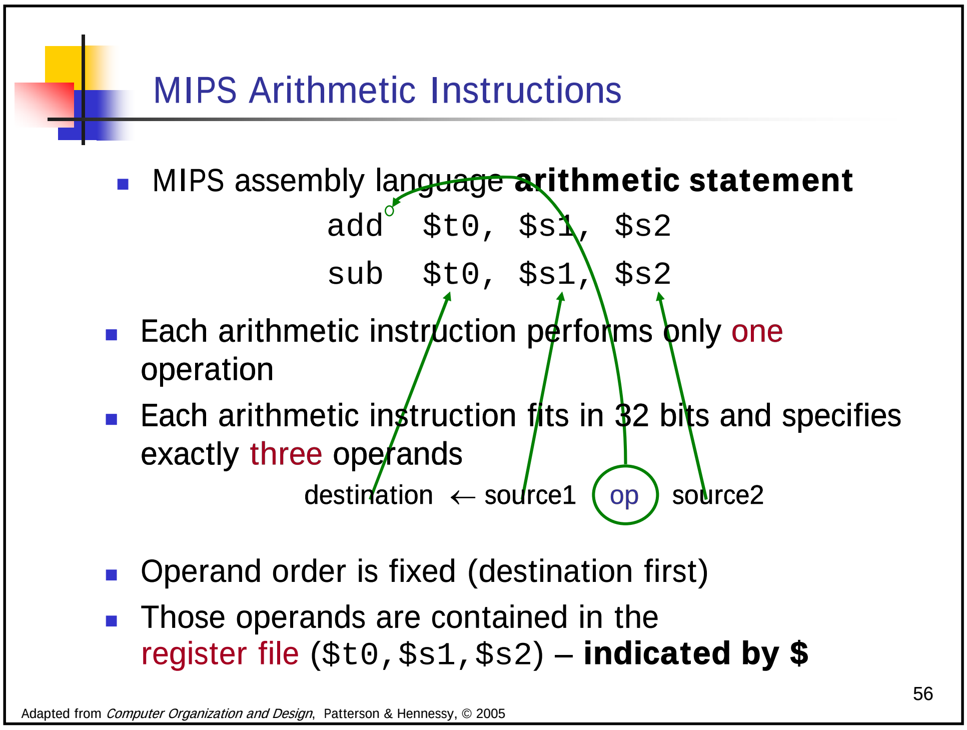
- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

  destination ← source1 op source2

- Those operands are contained in the datapath's register file (`$t0,$s1,$s2`) – indicated by $
- Operand order is fixed (destination first)

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

55



## MIPS Arithmetic Instructions

- MIPS assembly language **arithmetic statement**

```
add  $t0, $s1, $s2
sub  $t0, $s1, $s2
```

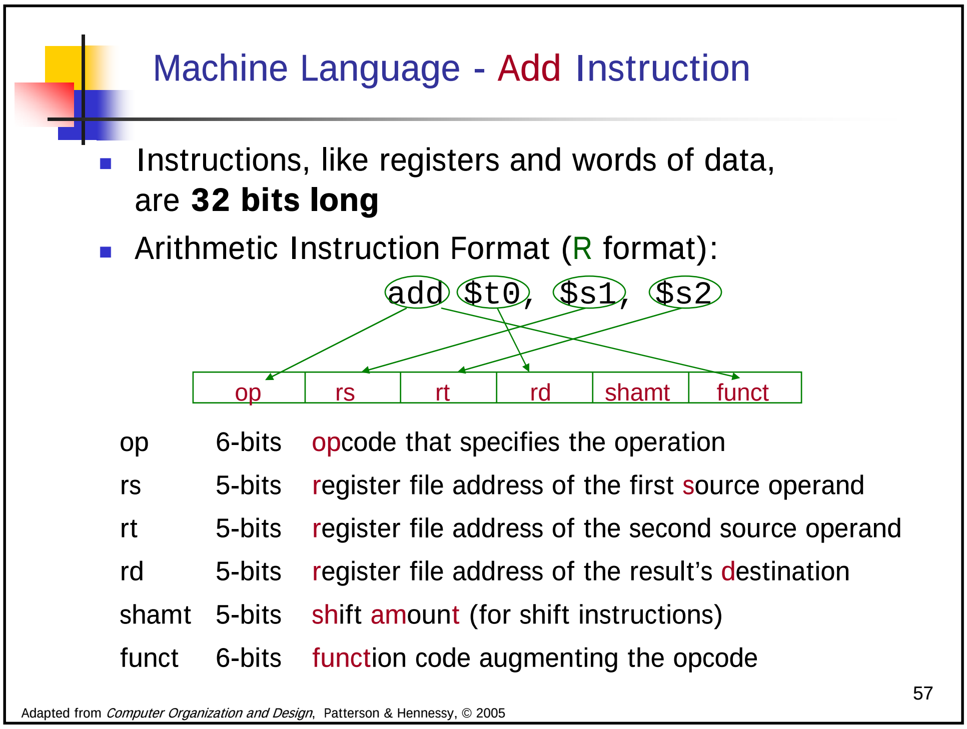
- Each arithmetic instruction performs only one operation
- Each arithmetic instruction fits in 32 bits and specifies exactly three operands

  destination ← source1 op source2

- Operand order is fixed (destination first)
- Those operands are contained in the register file (`$t0,$s1,$s2`) – **indicated by $**

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

56



## Machine Language - Add Instruction

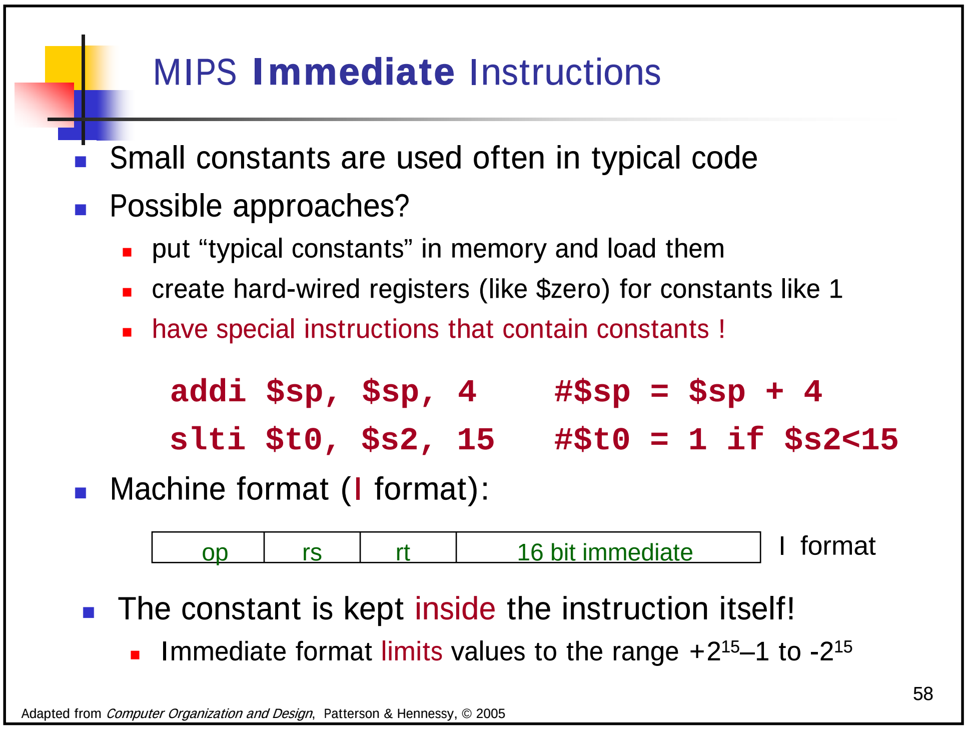
- Instructions, like registers and words of data, are **32 bits long**
- Arithmetic Instruction Format (R format):

```
add $t0, $s1, $s2
```

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|

| | | |
|---|---|---|
| op | 6-bits | opcode that specifies the operation |
| rs | 5-bits | register file address of the first source operand |
| rt | 5-bits | register file address of the second source operand |
| rd | 5-bits | register file address of the result's destination |
| shamt | 5-bits | shift amount (for shift instructions) |
| funct | 6-bits | function code augmenting the opcode |

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## MIPS **Immediate** Instructions

- Small constants are used often in typical code
- Possible approaches?
  - put "typical constants" in memory and load them
  - create hard-wired registers (like $zero) for constants like 1
  - have special instructions that contain constants !

```
addi $sp, $sp, 4     #$sp = $sp + 4
slti $t0, $s2, 15    #$t0 = 1 if $s2<15
```

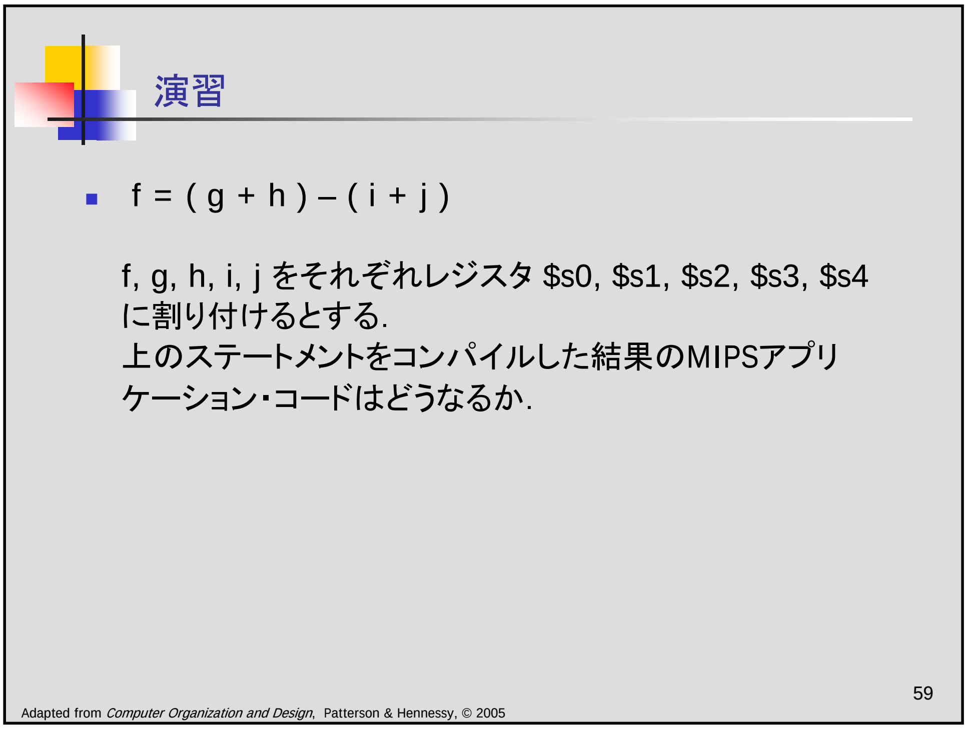
- Machine format (I format):

| op | rs | rt | 16 bit immediate |
|---|---|---|---|

I format

- The constant is kept inside the instruction itself!
  - Immediate format limits values to the range $+2^{15}–1$ to $-2^{15}$

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



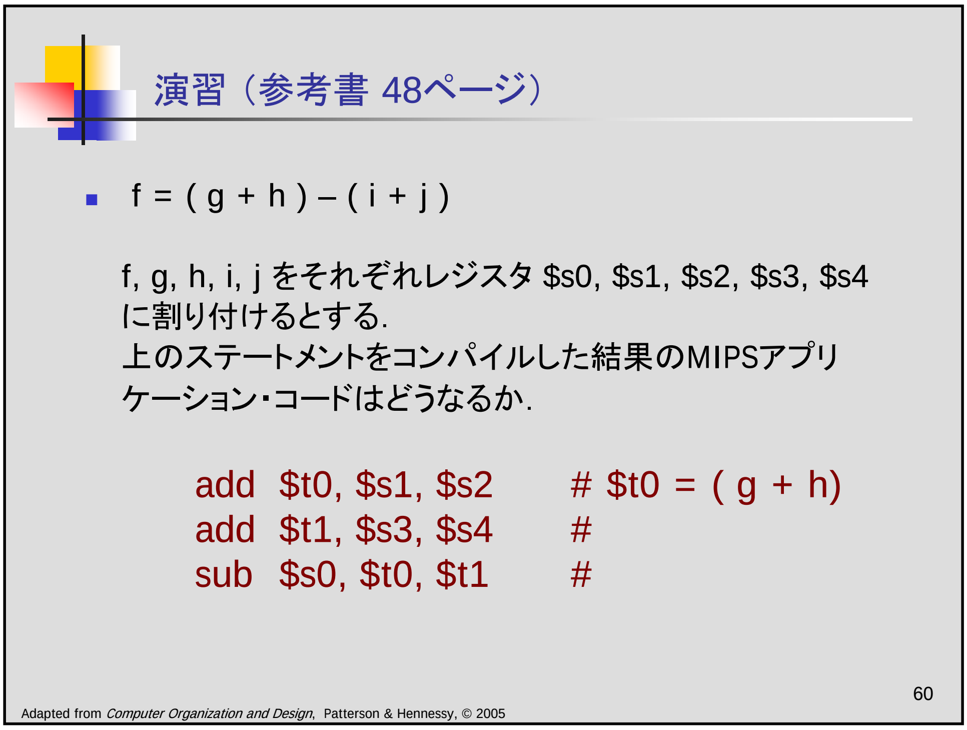
## 演習

- f = ( g + h ) – ( i + j )

f, g, h, i, j をそれぞれレジスタ $s0, $s1, $s2, $s3, $s4 に割り付けるとする.
上のステートメントをコンパイルした結果のMIPSアプリケーション・コードはどうなるか.

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

59



## 演習（参考書 48ページ）

- f = ( g + h ) – ( i + j )

f, g, h, i, j をそれぞれレジスタ $s0, $s1, $s2, $s3, $s4 に割り付けるとする.
上のステートメントをコンパイルした結果のMIPSアプリケーション・コードはどうなるか.

```
add  $t0, $s1, $s2      # $t0 = ( g + h)
add  $t1, $s3, $s4      #
sub  $s0, $t0, $t1      #
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

60

## MIPS **Memory Access** Instructions

- MIPS has two basic data transfer instructions for accessing memory

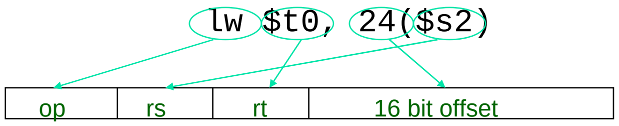
```
lw  $t0, 4($s3)  # load word from memory
sw  $t0, 8($s3)  # store word to memory
```

- The data is loaded into (lw) or stored from (sw) a register in the register file
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value
  - A 16-bit field is limited to memory locations within a region of ±$2^{13}$ or 8,192 words (±$2^{15}$ or 32,768 bytes) of the address in the base register

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

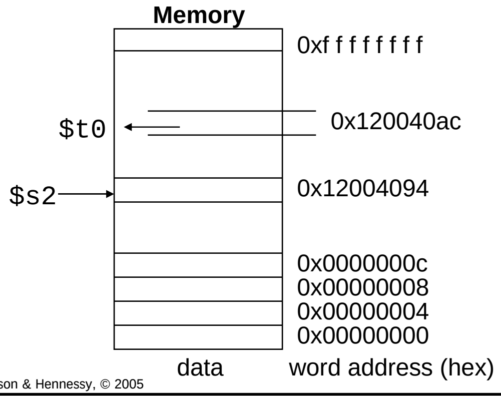
## Machine Language - Load Instruction

- Load / Store Instruction Format (I format):



$24_{10}$ + $s2 =

. . . 0001 1000
+ . . . 1001 0100
. . . 1010 1100 =
0x120040ac



Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## 演習

- g = h + A[8]
100語から成る配列Aがあるとする．また，コンパイラは変数g, h にレジスタ $s1, $s2 を割り付ける．さらに配列の開始アドレスは $s3 に納められているとする．
上のステートメントをコンパイルせよ．

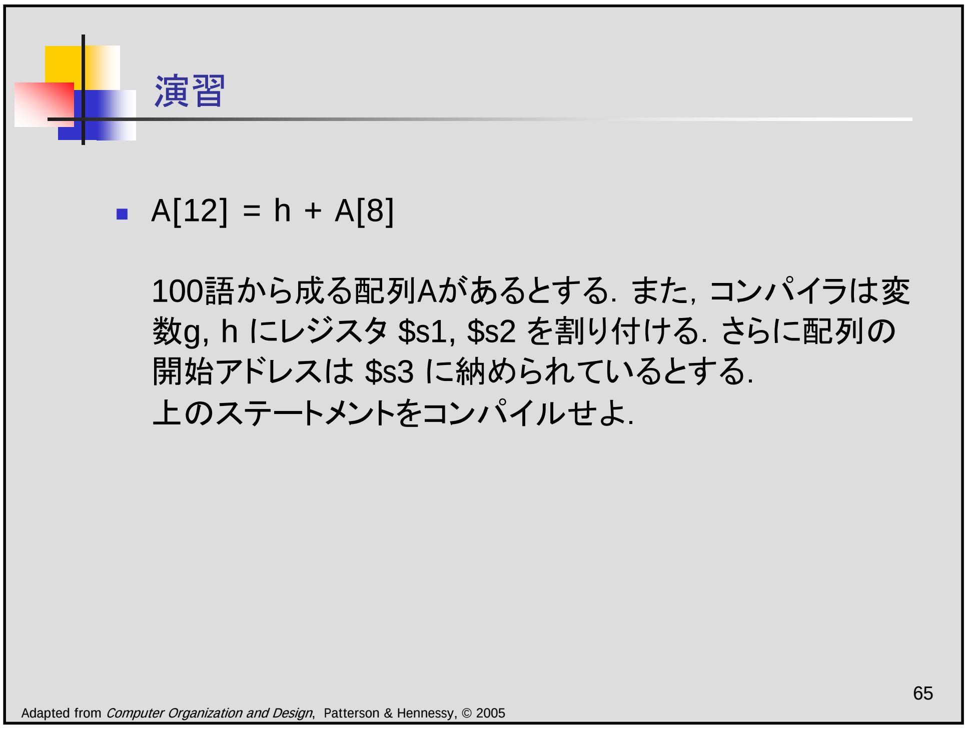
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## 演習（参考書 50ページ）

- g = h + A[8]
100語から成る配列Aがあるとする．また，コンパイラは変数g, h にレジスタ $s1, $s2 を割り付ける．さらに配列の開始アドレスは $s3 に納められているとする．
上のステートメントをコンパイルせよ．

```
lw    $t0, 32($s3)      # $t0 = A[8]
add   $s1, $s2, $t0     # g = h + $t0
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

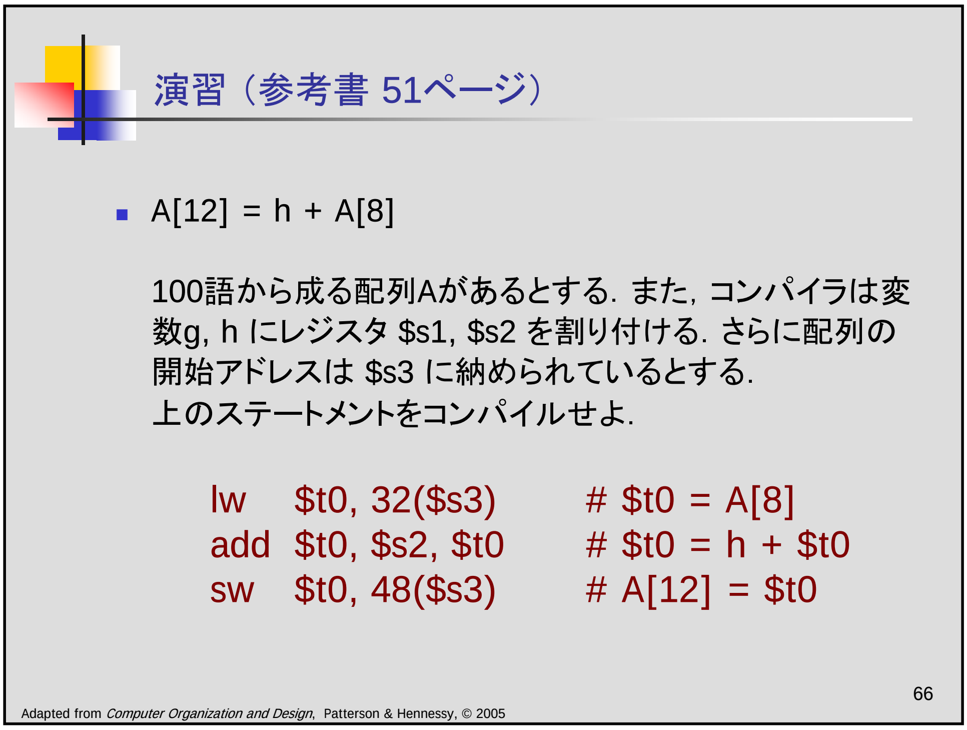


## 演習

- A[12] = h + A[8]

100語から成る配列Aがあるとする．また，コンパイラは変数g, h にレジスタ $s1, $s2 を割り付ける．さらに配列の開始アドレスは $s3 に納められているとする．
上のステートメントをコンパイルせよ．

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## 演習（参考書 51ページ）

- A[12] = h + A[8]

100語から成る配列Aがあるとする．また，コンパイラは変数g, h にレジスタ $s1, $s2 を割り付ける．さらに配列の開始アドレスは $s3 に納められているとする．
上のステートメントをコンパイルせよ．

```
lw   $t0, 32($s3)      # $t0 = A[8]
add  $t0, $s2, $t0     # $t0 = h + $t0
sw   $t0, 48($s3)      # A[12] = $t0
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## MIPS **Control Flow** Instructions

- MIPS conditional branch instructions:

```
bne $s0, $s1, Lbl #go to Lbl if $s0≠$s1
beq $s0, $s1, Lbl #go to Lbl if $s0=$s1
```

```
  ▪ Ex:     if (i==j) h = i + j;

            bne $s0, $s1, Lbl1
            add $s3, $s0, $s1
Lbl1:       ...
```

- Instruction Format (I format):

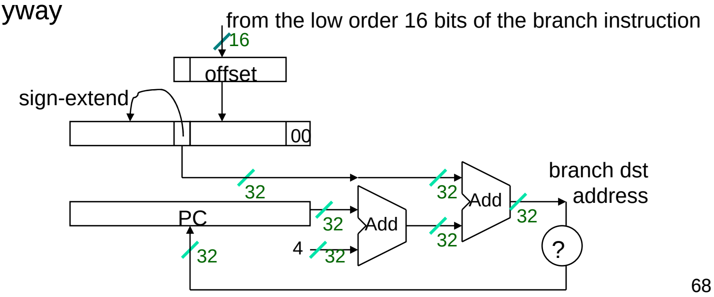
| op | rs | rt | 16 bit offset |
|---|---|---|---|

- How is the branch destination address specified?

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Specifying Branch Destinations

- Use a register (like in lw and sw) added to the 16-bit offset
  - which register? Instruction Address Register (the PC)
    - its use is automatically implied by instruction
    - PC gets updated (PC+4) during the fetch cycle so that it holds the address of the next instruction
  - limits the branch distance to $-2^{15}$ to $+2^{15}-1$ instructions from the (instruction after the) branch instruction, but most branches are local anyway



Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## More Branch Instructions

- We have beq, bne, but what about other kinds of brances (e.g., branch-if-less-than)? For this, we need yet another instruction, slt
- Set on less than instruction:

```
slt $t0, $s0, $s1     # if $s0 < $s1    then
                      # $t0 = 1         else
                      # $t0 = 0
```

- Instruction format (R format):

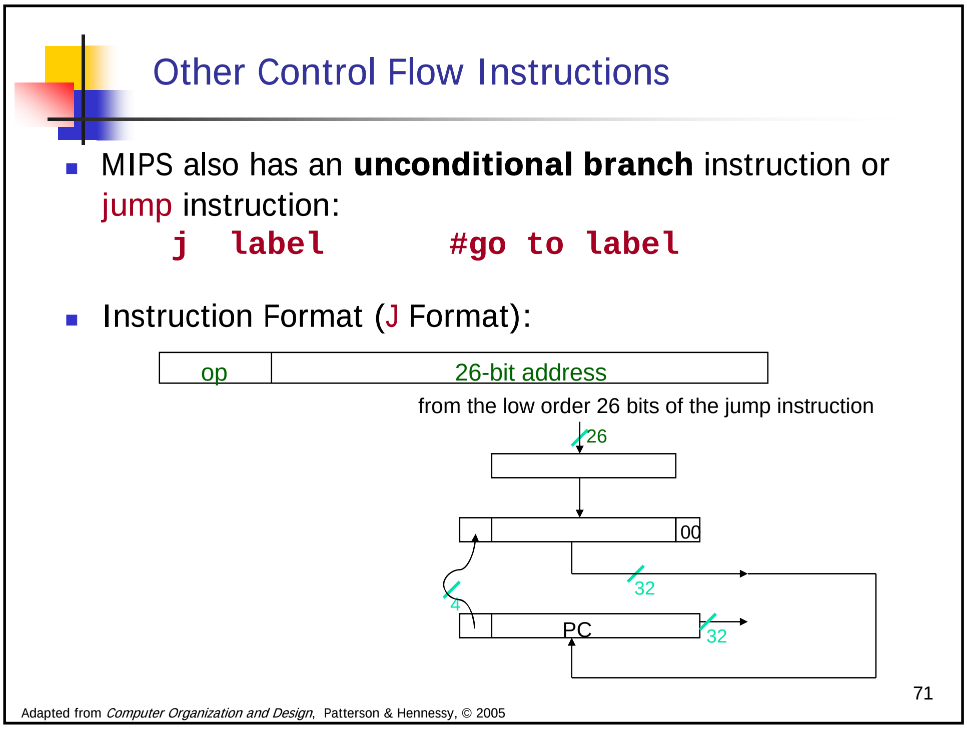
| op | rs | rt | rd |  | funct |
|---|---|---|---|---|---|

## More Branch Instructions, Con't

- Can use slt, beq, bne, and the fixed value of 0 in register $zero to create other conditions
  - less than blt $s1, $s2, Label

```
slt  $at, $s1, $s2       # $at set to 1 if
bne  $at, $zero, Label   #  $s1 < $s2
```

  - less than or equal to **ble $s1, $s2, Label**
  - greater than **bgt $s1, $s2, Label**
  - great than or equal to **bge $s1, $s2, Label**
- Such branches are included in the instruction set as pseudo instructions - recognized (and expanded) by the assembler
  - Its why the assembler needs a reserved register ($at)



## Other Control Flow Instructions

- MIPS also has an **unconditional branch** instruction or jump instruction:

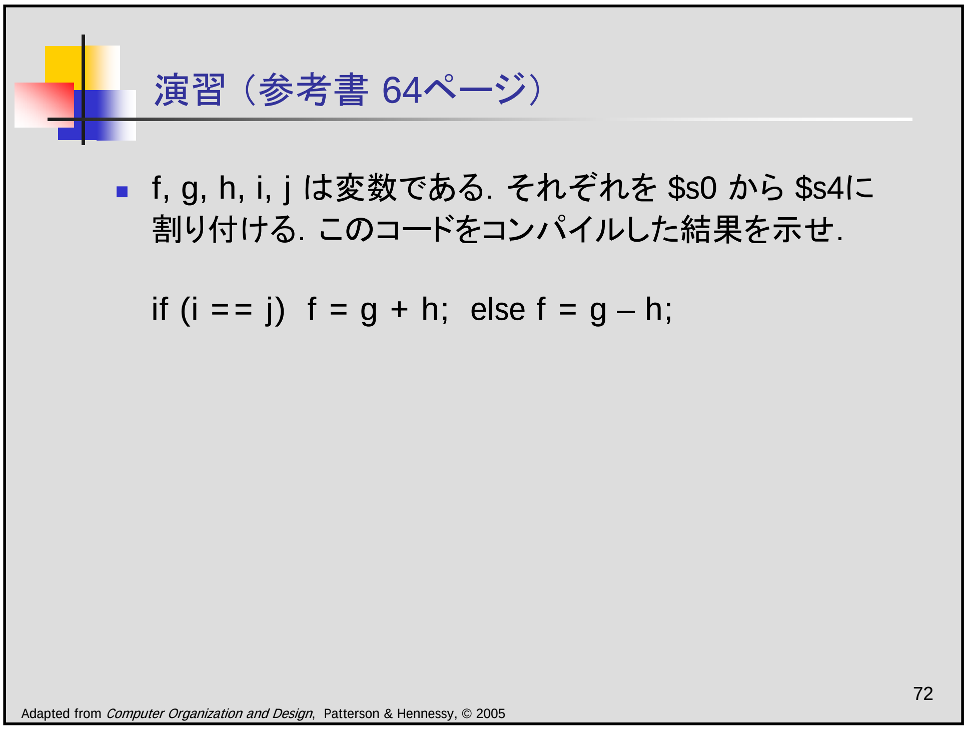
```
      j   label       #go to label
```

- Instruction Format (J Format):

| op | 26-bit address |
|---|---|

from the low order 26 bits of the jump instruction

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## 演習（参考書 64ページ）

- f, g, h, i, j は変数である．それぞれを $s0 から $s4に割り付ける．このコードをコンパイルした結果を示せ．

  if (i == j)  f = g + h;  else f = g – h;

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## 演習（参考書 64ページ）

- f, g, h, i, j は変数である．それぞれを $s0 から $s4に割り付ける．このコードをコンパイルした結果を示せ．

  if (i == j)  f = g + h;  else f = g – h;

```
      bne   $s3, $s4, Else      # if (i!=j) goto Else
      add   $s0, $s1, $s2       # f = g + h
      j     Exit                # goto Exit
Else:
      sub   $s0, $s1, $s2       # f = g - h
Exit:
```

73

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

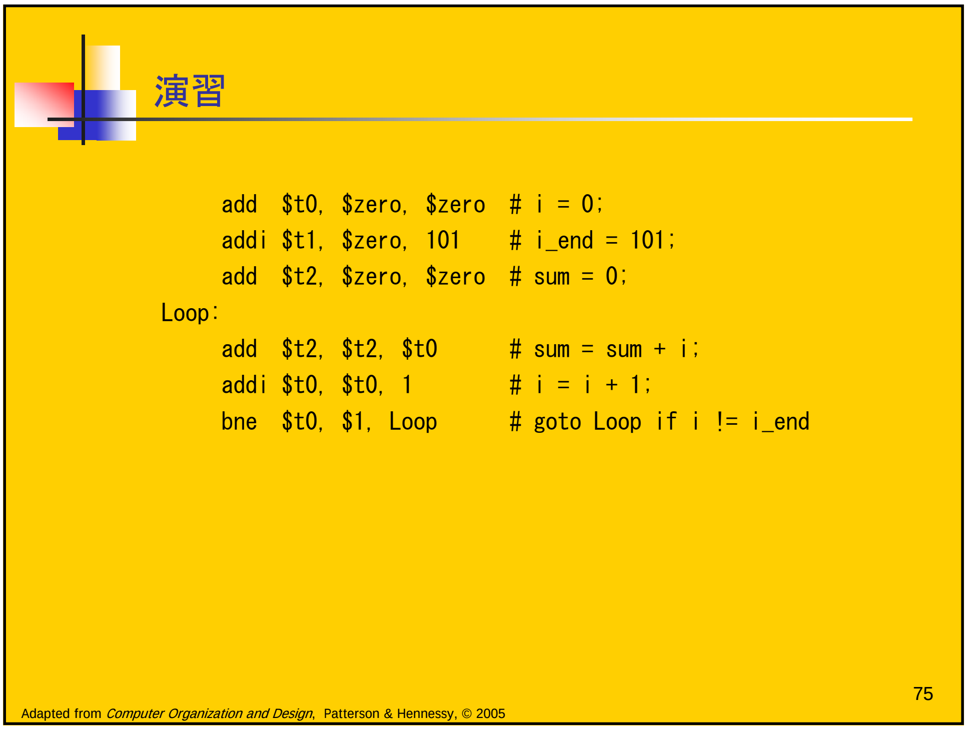
## 演習

- ループを利用して1から100までの合計値を求めるアセンブラを示せ．

氏名，学籍番号，
学籍番号マーク欄(右詰で)

74

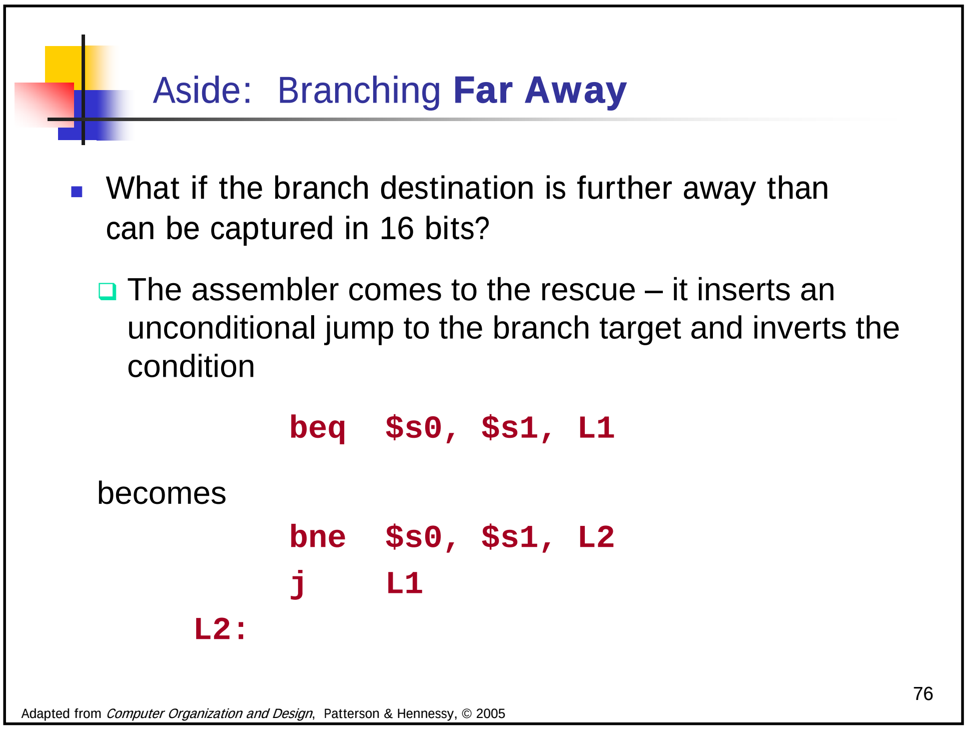
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## 演習

```
          add  $t0, $zero, $zero  # i = 0;
          addi $t1, $zero, 101    # i_end = 101;
          add  $t2, $zero, $zero  # sum = 0;
     Loop:
          add  $t2, $t2, $t0      # sum = sum + i;
          addi $t0, $t0, 1        # i = i + 1;
          bne  $t0, $1, Loop      # goto Loop if i != i_end
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Aside: Branching **Far Away**

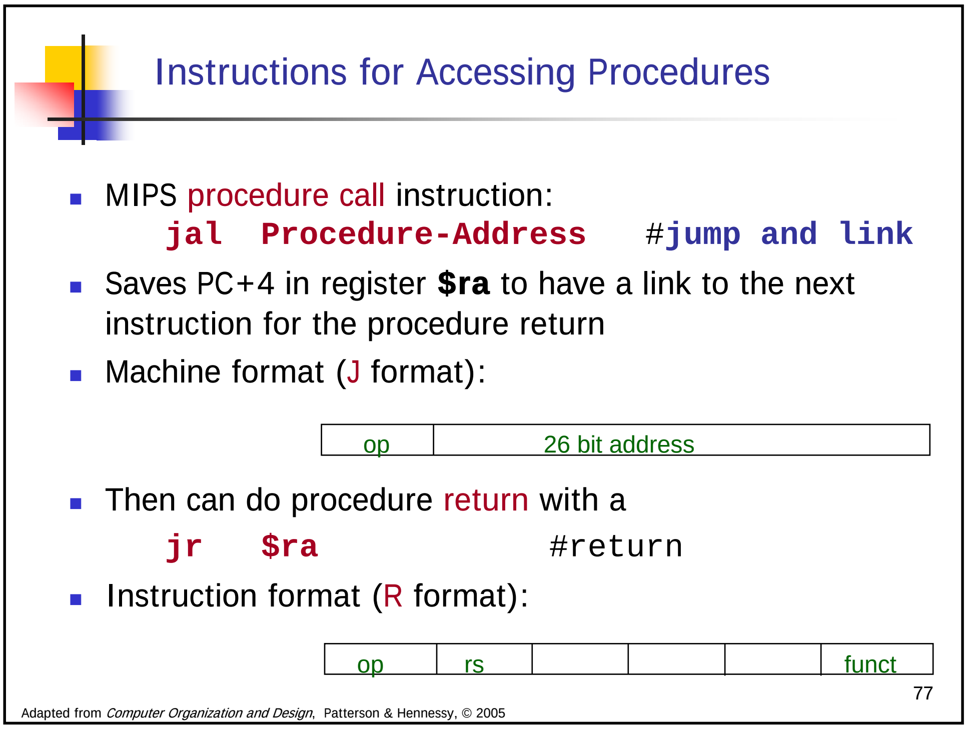
- What if the branch destination is further away than can be captured in 16 bits?
  - The assembler comes to the rescue – it inserts an unconditional jump to the branch target and inverts the condition

```
                beq  $s0, $s1, L1
```

becomes

```
                bne  $s0, $s1, L2
                j    L1
        L2:
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Instructions for Accessing Procedures

- MIPS procedure call instruction:

```
    jal   Procedure-Address    #jump and link
```

- Saves PC+4 in register **$ra** to have a link to the next instruction for the procedure return
- Machine format (J format):

| op | 26 bit address |
|---|---|

- Then can do procedure return with a

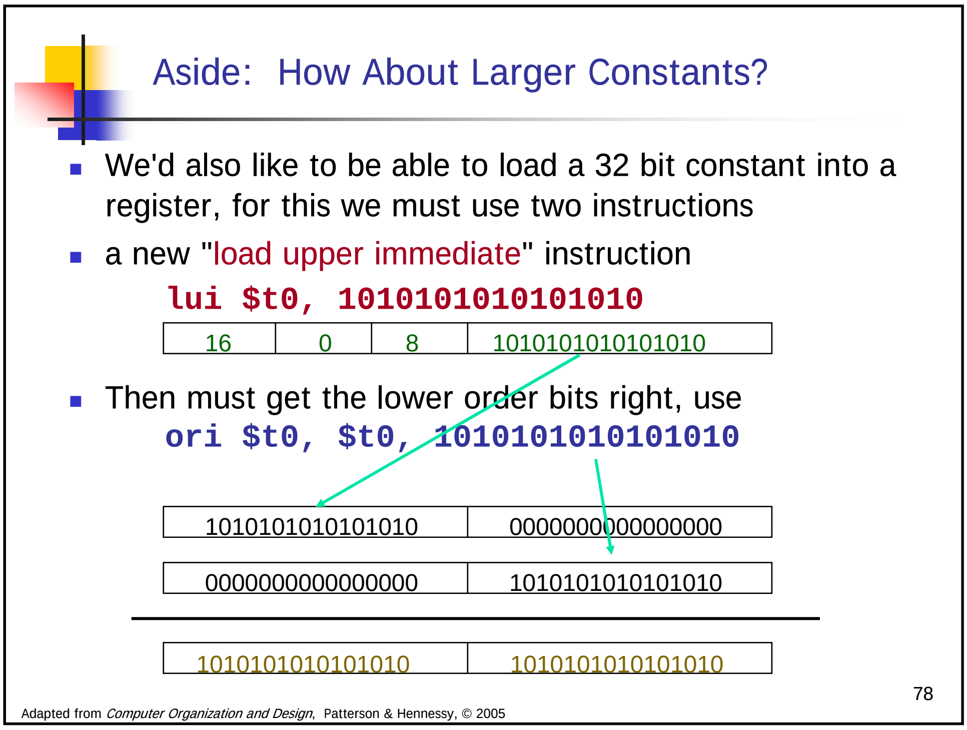
```
    jr    $ra                 #return
```

- Instruction format (R format):

| op | rs | | | | funct |
|---|---|---|---|---|---|

77

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Aside: How About Larger Constants?

- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

```
    lui $t0, 1010101010101010
```

| 16 | 0 | 8 | 1010101010101010 |
|---|---|---|---|

- Then must get the lower order bits right, use

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

| | |
|---|---|
| 1010101010101010 | 0000000000000000 |
| 0000000000000000 | 1010101010101010 |
| 1010101010101010 | 1010101010101010 |

78

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

# MIPS ISA So Far

| Category | Instr | Op Code | Example | Meaning |
|---|---|---|---|---|
| Arithmetic<br>(R & I format) | add | 0 and 32 | add $s1, $s2, $s3 | $s1 = $s2 + $s3 |
| | subtract | 0 and 34 | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 |
| | add immediate | 8 | addi $s1, $s2, 6 | $s1 = $s2 + 6 |
| | or immediate | 13 | ori $s1, $s2, 6 | $s1 = $s2 v 6 |
| Data Transfer<br>(I format) | load word | 35 | lw $s1, 24($s2) | $s1 = Memory($s2+24) |
| | store word | 43 | sw $s1, 24($s2) | Memory($s2+24) = $s1 |
| | load byte | 32 | lb $s1, 25($s2) | $s1 = Memory($s2+25) |
| | store byte | 40 | sb $s1, 25($s2) | Memory($s2+25) = $s1 |
| | load upper imm | 15 | lui $s1, 6 | $s1 = 6 * $2^{16}$ |
| Cond. Branch<br>(I & R format) | br on equal | 4 | beq $s1, $s2, L | if ($s1==$s2) go to L |
| | br on not equal | 5 | bne $s1, $s2, L | if ($s1 !=$s2) go to L |
| | set on less than | 0 and 42 | slt $s1, $s2, $s3 | if ($s2<$s3) $s1=1 else $s1=0 |
| | set on less than immediate | 10 | slti $s1, $s2, 6 | if ($s2<6) $s1=1 else $s1=0 |
| Uncond. Jump (J & R format) | jump | 2 | j 2500 | go to 10000 |
| | jump register | 0 and 8 | jr $t1 | go to $t1 |
| | jump and link | 3 | jal 2500 | go to 10000; $ra=PC+4 |

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

# 今日のまとめ, MIPS R3000 ISA

- Instruction Categories
  - Computational
  - Load / Store
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special

**Registers**

**R0 - R31**

**PC**

**HI**

**LO**

**3 Instruction Formats: all 32 bits wide**

| | | | | | | |
|---|---|---|---|---|---|---|
| **OP** | **rs** | **rt** | **rd** | **sa** | **funct** | R format |
| **OP** | **rs** | **rt** | **Immediate (16bit)** | | | I format |
| **OP** | **jump target (26bit)** | | | | | J format |

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

2009-05-07

2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 3. 命令形式，アドレス指定形式

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

# Acknowledgement

- Lecture slides for Computer Organization and Design, Third Edition, courtesy of **Professor Mary Jane Irwin**, Penn State University
- Lecture slides for Computer Organization and Design, third edition, Chapters 1-9, courtesy of **Professor Tod Amon,** Southern Utah University.

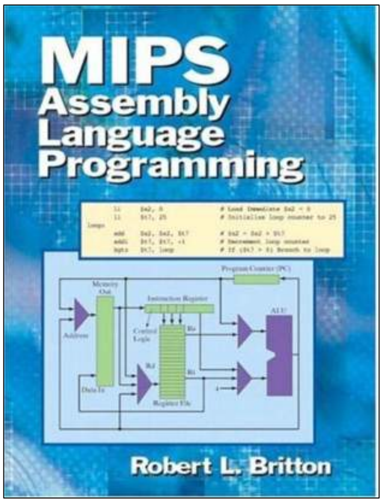
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## 参考書（読んでください）

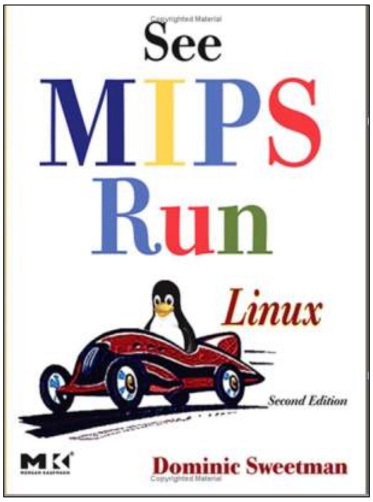
- **コンピュータの構成と設計 第3版**、パターソン＆ヘネシー（成田光彰　訳）、日経BP社、2006
- コンピュータアーキテクチャ 定量的アプローチ 第4版
  翔泳社, 2008
- コンピュータアーキテクチャ,
  村岡 洋一 著，近代科学社，1989
- 計算機システム工学,
  富田 真治，村上 和彰 著，昭晃堂，1988
- コンピュータハードウヱア,
  富田 真治，中島 浩 著，昭晃堂，1995
- 計算機アーキテクチャ,
  橋本 昭洋 著，昭晃堂，1995

パターソン&ヘネシー
コンピュータの構成と設計
ハードウエアとソフトウエアのインタフェース
David A.Patterson/John L.Hennessy
成田 光彰 訳
第3版
上
下
日経BP社

## 参考書（アセンブラに興味があれば）



MIPSのアセンブラがよくわかります．面白いです．



MIPSとLinuxの間がわかります．お勧め．

## Aside: How About Larger Constants?

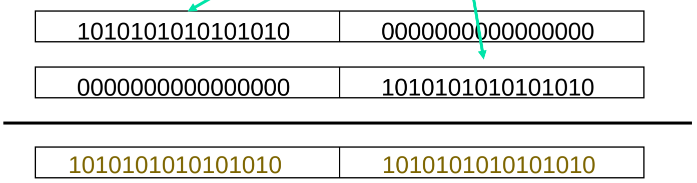
- We'd also like to be able to load a 32 bit constant into a register, for this we must use two instructions
- a new "load upper immediate" instruction

  **`lui $t0, 1010101010101010`**

| | | | |
|---|---|---|---|
| 16 | 0 | 8 | 1010101010101010 |

- Then must get the lower order bits right, use

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



Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## MIPS ISA So Far

| Category | Instr | Op Code | Example | Meaning |
|---|---|---|---|---|
| Arithmetic<br>(R & I format) | add | 0 and 32 | add $s1, $s2, $s3 | $s1 = $s2 + $s3 |
| | subtract | 0 and 34 | sub $s1, $s2, $s3 | $s1 = $s2 - $s3 |
| | add immediate | 8 | addi $s1, $s2, 6 | $s1 = $s2 + 6 |
| | or immediate | 13 | ori $s1, $s2, 6 | $s1 = $s2 v 6 |
| Data Transfer<br>(I format) | load word | 35 | lw $s1, 24($s2) | $s1 = Memory($s2+24) |
| | store word | 43 | sw $s1, 24($s2) | Memory($s2+24) = $s1 |
| | load byte | 32 | lb $s1, 25($s2) | $s1 = Memory($s2+25) |
| | store byte | 40 | sb $s1, 25($s2) | Memory($s2+25) = $s1 |
| | load upper imm | 15 | lui $s1, 6 | $s1 = 6 * $2^{16}$ |
| Cond. Branch<br>(I & R format) | br on equal | 4 | beq $s1, $s2, L | if ($s1==$s2) go to L |
| | br on not equal | 5 | bne $s1, $s2, L | if ($s1 !=$s2) go to L |
| | set on less than | 0 and 42 | slt $s1, $s2, $s3 | if ($s2<$s3) $s1=1 else $s1=0 |
| | set on less than immediate | 10 | slti $s1, $s2, 6 | if ($s2<6) $s1=1 else $s1=0 |
| Uncond. Jump (J & R format) | jump | 2 | j 2500 | go to 10000 |
| | jump register | 0 and 8 | jr $t1 | go to $t1 |
| | jump and link | 3 | jal 2500 | go to 10000; $ra=PC+4 |

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Aside: Loading and Storing Bytes

- MIPS provides special instructions to move bytes

```
lb   $t0, 1($s3)  #load byte from memory
sb   $t0, 6($s3)  #store byte to  memory
```

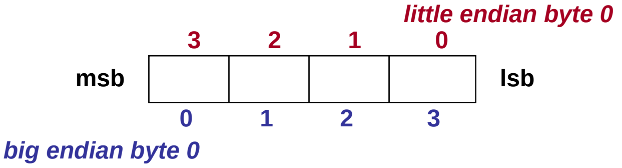
| op | rs | rt | 16 bit offset |
|---|---|---|---|

- What 8 bits get loaded and stored?
  - load byte places the byte from memory in the rightmost 8 bits of the destination register
    - what happens to the other bits in the register?
  - store byte takes the byte from the rightmost 8 bits of a register and writes it to a byte in memory
    - what happens to the other bits in the memory word?

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Byte Addresses

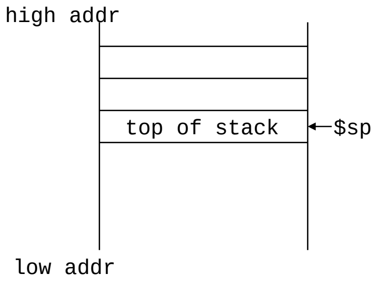
- Since 8-bit bytes are so useful, most architectures address individual bytes in memory
  - The memory address of a word must be a multiple of 4 (alignment restriction)
- **Big Endian:**
  - leftmost byte is word address
    IBM 360/370, Motorola 68k, **MIPS**, SPARC, HP PA
- **Little Endian:**
  - rightmost byte is word address
    Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Aside: **Spilling Registers**

- What if the **callee** needs more registers? What if the procedure is **recursive**?
  - uses a **stack** – a last-in-first-out queue – in memory for passing additional values or saving (recursive) return address(es)



- One of the general registers, **$sp**, is used to address the stack (which "grows" from high address to low address)
  - add data onto the stack – **push**

    $sp = $sp – 4
    data on stack at new $sp
  - remove data from the stack – **pop**

    data from stack at $sp
    $sp = $sp + 4

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## MIPS R3000 ISA

- Instruction Categories
  - Computational
  - Load / Store
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special

**Registers**

| **R0 - R31** |
|---|
| **PC** |
| **HI** |
| **LO** |

**3 Instruction Formats: all 32 bits wide**

| | | | | | | |
|---|---|---|---|---|---|---|
| **OP** | **rs** | **rt** | **rd** | **sa** | **funct** | R format |
| **OP** | **rs** | **rt** | **Immediate (16bit)** | | | I format |
| **OP** | **jump target (26bit)** | | | | | J format |

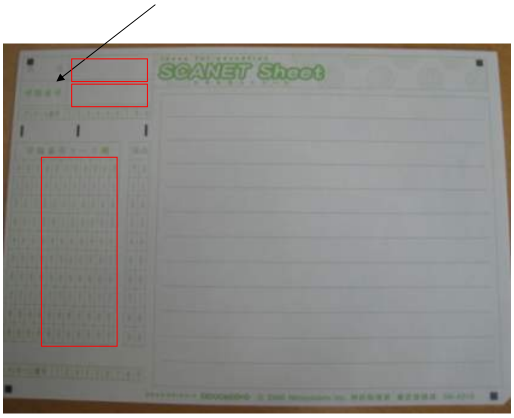
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

# 演習

- アセンブラを示せ.

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

氏名, 学籍番号,
学籍番号マーク欄(右詰で)



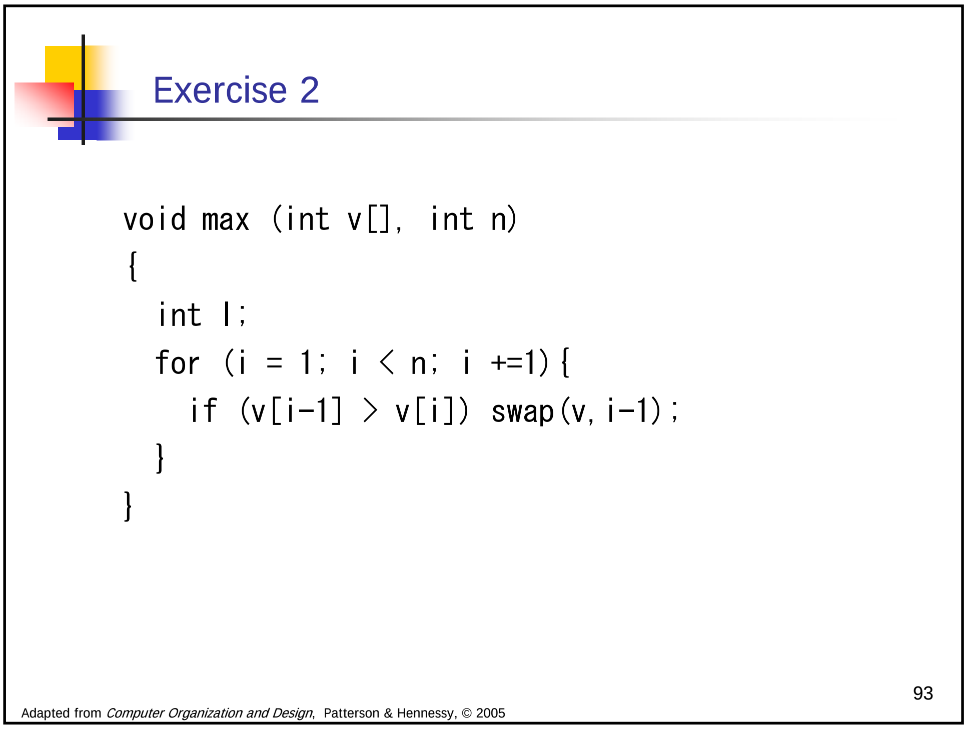
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

# Exercise 1

```
swap:
     add $t1, $a1, $a1  #
     add $t1, $t1, $t1  # $t1 = k * 4;
     add $t1, $a0, $t1  # $t1 = &v[k];

     lw  $t0, 0($t1)    # $t0 = v[k];
     lw  $t2, 4($t1)    # $t2 = v[k+1];

     sw  $t0, 4($t1)    # v[k+1] = $t0;
     sw  $t2, 0($t1)    # v[k]   = $t2;

     jr $ra             # return
```

sll (shift left logical) $t1, $a1, 2

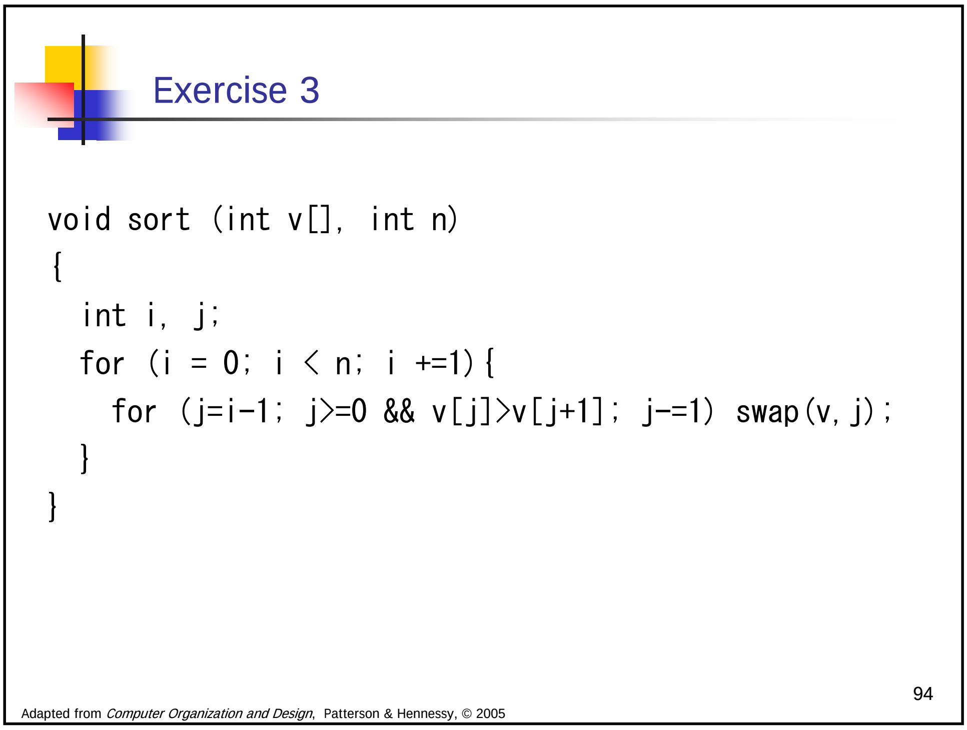
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Exercise 2

```
void max (int v[], int n)
{
  int I;
  for (i = 1; i < n; i +=1){
    if (v[i-1] > v[i]) swap(v,i-1);
  }
}
```

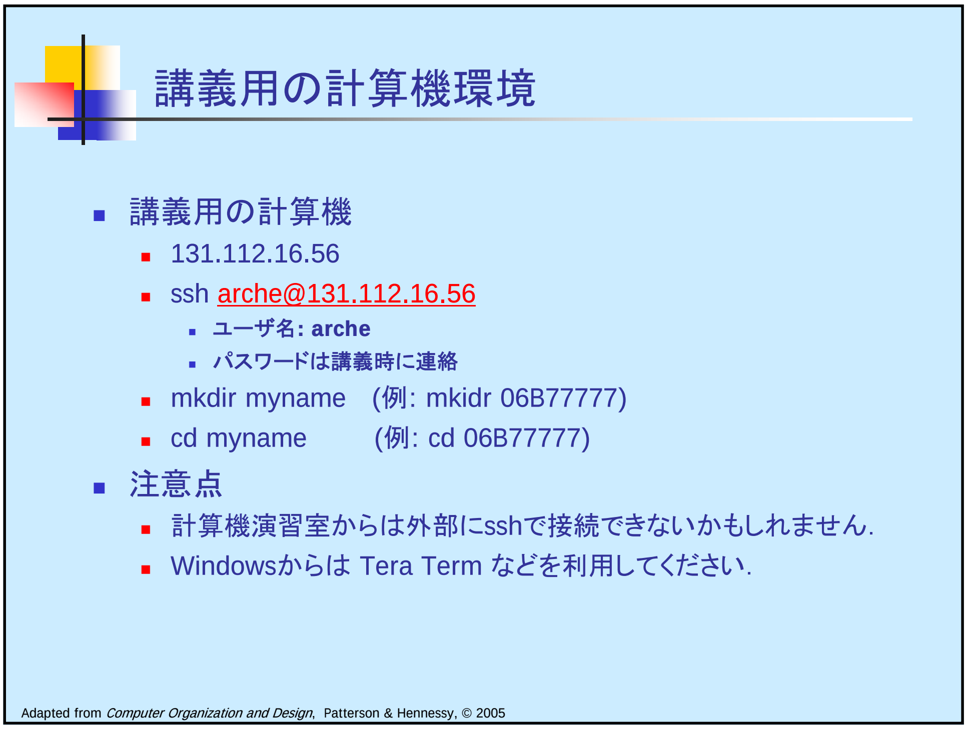
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Exercise 3

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i +=1){
    for (j=i-1; j>=0 && v[j]>v[j+1]; j-=1) swap(v,j);
  }
}
```

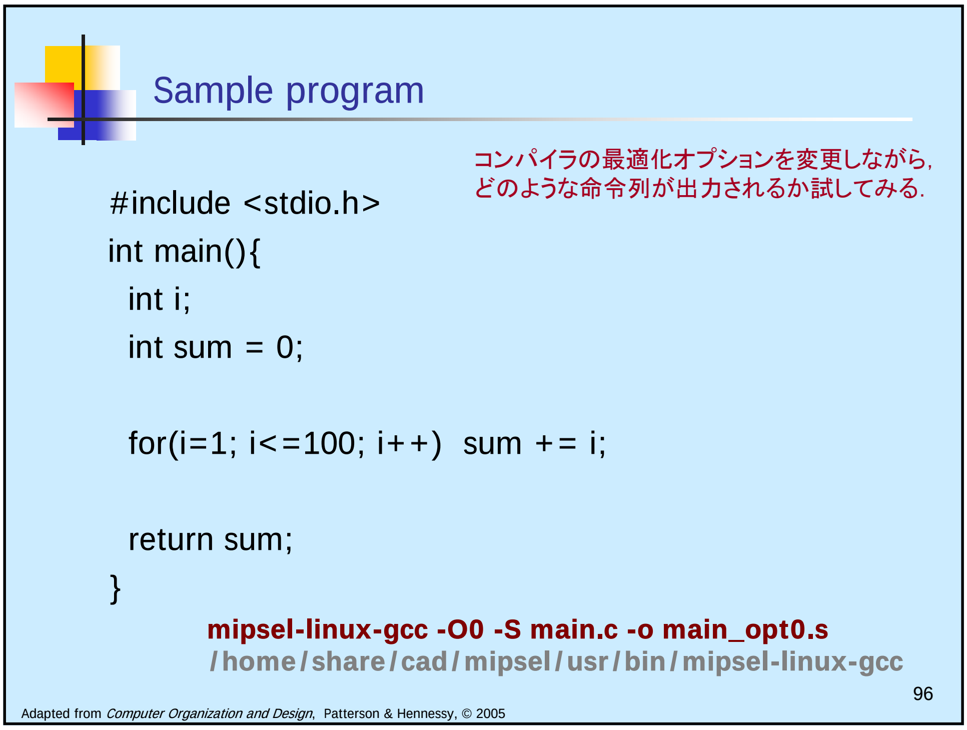
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## 講義用の計算機環境

- 講義用の計算機
  - 131.112.16.56
  - ssh arche@131.112.16.56
    - **ユーザ名: arche**
    - **パスワードは講義時に連絡**
  - mkdir myname　(例: mkidr 06B77777)
  - cd myname　　(例: cd 06B77777)
- 注意点
  - 計算機演習室からは外部にsshで接続できないかもしれません.
  - Windowsからは Tera Term などを利用してください.

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Sample program

コンパイラの最適化オプションを変更しながら,
どのような命令列が出力されるか試してみる.

```
#include <stdio.h>
int main(){
  int i;
  int sum = 0;

  for(i=1; i<=100; i++)  sum += i;

  return sum;
}
```

**mipsel-linux-gcc -O0 -S main.c -o main_opt0.s**

**/home/share/cad/mipsel/usr/bin/mipsel-linux-gcc**

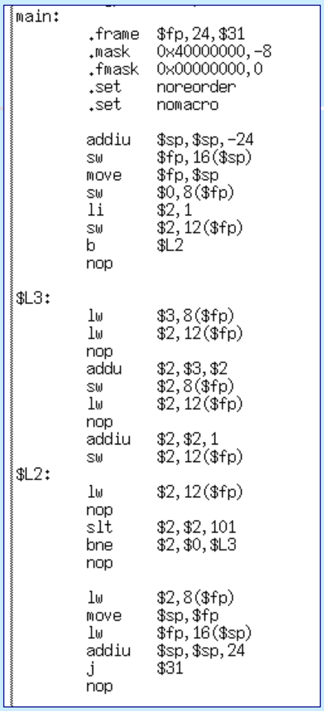
96

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Sample program

```
#include <stdio.h>
int main(){
  int i;
  int sum = 0;

  for(i=1; i<=100; i++)
    sum += i;

  return sum;
}
```

**mipsel-linux-gcc -O0 -S main.c -o main_opt0.s**



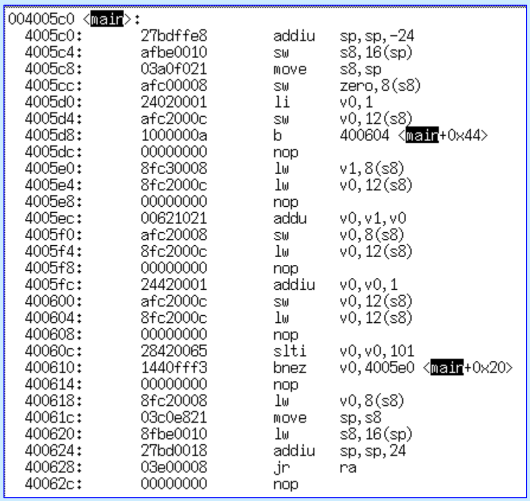
```
main:
        .frame  $fp,24,$31
        .mask   0x40000000,-8
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro

        addiu   $sp,$sp,-24
        sw      $fp,16($sp)
        move    $fp,$sp
        sw      $0,8($fp)
        li      $2,1
        sw      $2,12($fp)
        b       $L2
        nop

$L3:
        lw      $3,8($fp)
        lw      $2,12($fp)
        nop
        addu    $2,$3,$2
        sw      $2,8($fp)
        lw      $2,12($fp)
        nop
        addiu   $2,$2,1
        sw      $2,12($fp)
$L2:
        lw      $2,12($fp)
        nop
        slt     $2,$2,101
        bne     $2,$0,$L3
        nop

        lw      $2,8($fp)
        move    $sp,$fp
        lw      $fp,16($sp)
        addiu   $sp,$sp,24
        j       $31
        nop
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Sample program

```
#include <stdio.h>

int main(){
  int i;
  int sum = 0;

  for(i=1; i<=100; i++)
    sum += i;

  return sum;
}
```

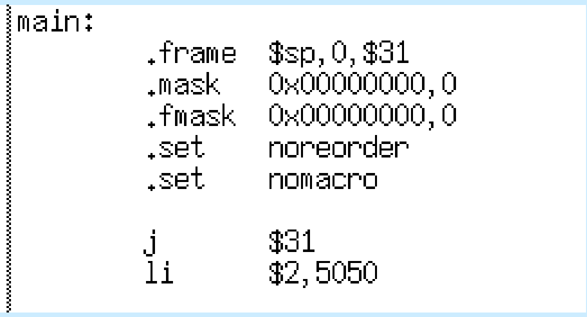
**mipsel-linux-objdump** -d ./a.out



```
004005c0 <main>:
  4005c0:       27bdffe8        addiu   sp,sp,-24
  4005c4:       afbe0010        sw      s8,16(sp)
  4005c8:       03a0f021        move    s8,sp
  4005cc:       afc00008        sw      zero,8(s8)
  4005d0:       24020001        li      v0,1
  4005d4:       afc2000c        sw      v0,12(s8)
  4005d8:       1000000a        b       400604 <main+0x44>
  4005dc:       00000000        nop
  4005e0:       8fc30008        lw      v1,8(s8)
  4005e4:       8fc2000c        lw      v0,12(s8)
  4005e8:       00000000        nop
  4005ec:       00621021        addu    v0,v1,v0
  4005f0:       afc20008        sw      v0,8(s8)
  4005f4:       8fc2000c        lw      v0,12(s8)
  4005f8:       00000000        nop
  4005fc:       24420001        addiu   v0,v0,1
  400600:       afc2000c        sw      v0,12(s8)
  400604:       8fc2000c        lw      v0,12(s8)
  400608:       00000000        nop
  40060c:       28420065        slti    v0,v0,101
  400610:       1440fff3        bnez    v0,4005e0 <main+0x20>
  400614:       00000000        nop
  400618:       8fc20008        lw      v0,8(s8)
  40061c:       03c0e821        move    sp,s8
  400620:       8fbe0010        lw      s8,16(sp)
  400624:       27bd0018        addiu   sp,sp,24
  400628:       03e00008        jr      ra
  40062c:       00000000        nop
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Sample program



```
main:
        .frame  $sp,0,$31
        .mask   0x00000000,0
        .fmask  0x00000000,0
        .set    noreorder
        .set    nomacro

        j       $31
        li      $2,5050
```

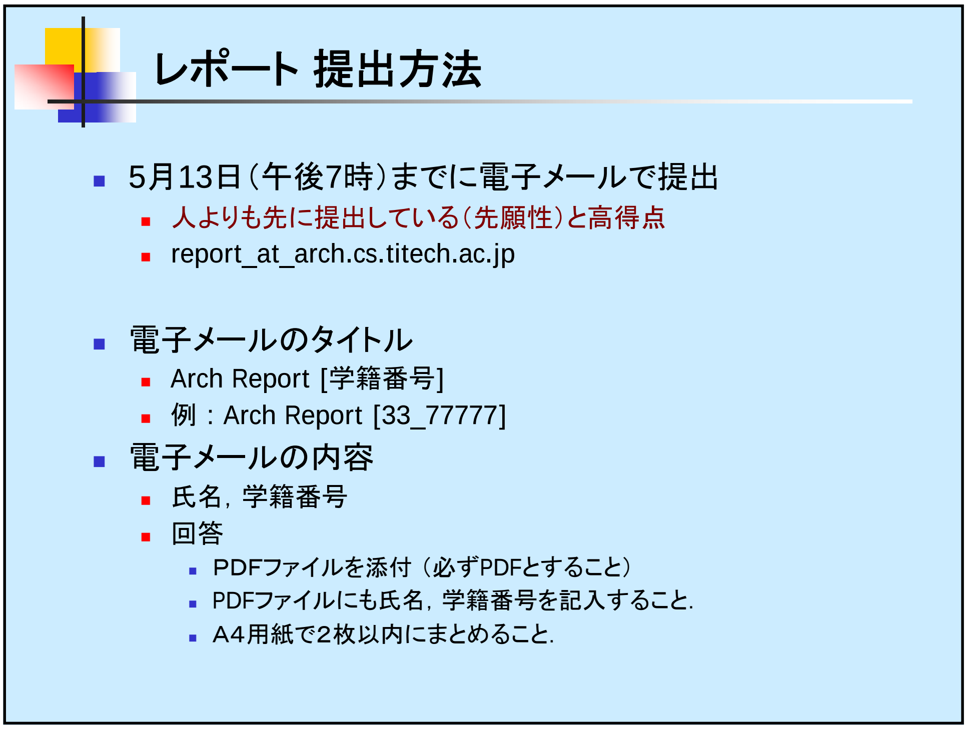
遅延分岐に注意

```
# Makefile
all:
        mipsel-linux-gcc -O0 -S main.c -o main_opt0.s
        mipsel-linux-gcc -O1 -S main.c -o main_opt1.s
        mipsel-linux-gcc -O2 -S main.c -o main_opt2.s
        mipsel-linux-gcc -O3 -S main.c -o main_opt3.s
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

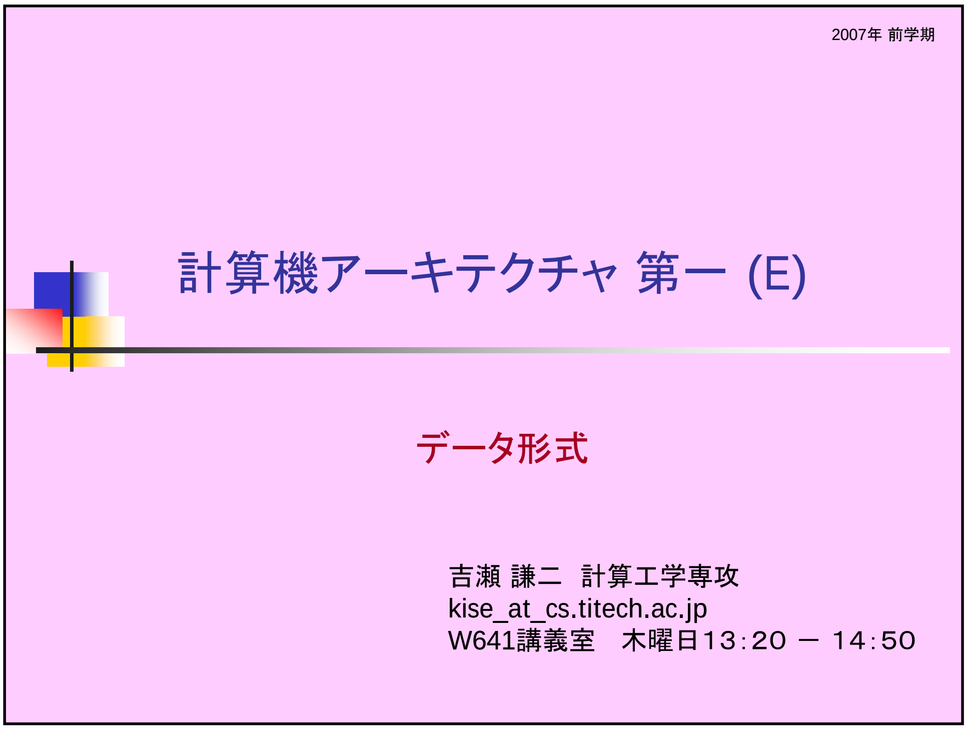
## レポート 問題

1. void max (int v[], int n)
   をクロスコンパイラにてMIPS命令セットにコンパイルし，コンパイルオプションによってどのように変化するかをまとめよ．
2. void sort (int v[], int n)
   をクロスコンパイラにてMIPS命令セットにコンパイルし，コンパイルオプションによってどのように変化するかをまとめよ．
3. 同様に，サンプルアプリケーションを作成し，それをクロスコンパイラにてMIPS命令セットにコンパイルし，コンパイルオプションによってどのように変化するかをまとめよ．
4. この課題の感想をまとめること．
5. レポートはA4用紙2枚以内にまとめること．（必ずPDFとすること）
   （2段組，コードは小さい文字でもかまわない．）



## レポート 提出方法

- 5月13日（午後7時）までに電子メールで提出
  - 人よりも先に提出している（先願性）と高得点
  - report_at_arch.cs.titech.ac.jp
- 電子メールのタイトル
  - Arch Report [学籍番号]
  - 例：Arch Report [33_77777]
- 電子メールの内容
  - 氏名，学籍番号
  - 回答
    - PDFファイルを添付（必ずPDFとすること）
    - PDFファイルにも氏名，学籍番号を記入すること.
    - A4用紙で2枚以内にまとめること.

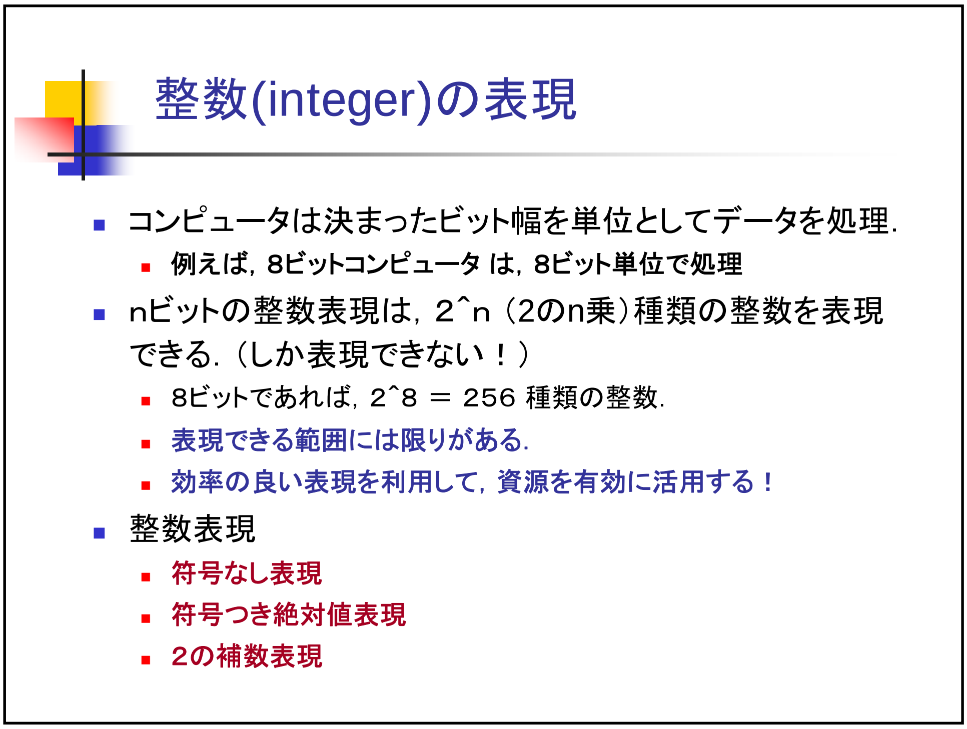


2007年 前学期

# 計算機アーキテクチャ 第一 (E)

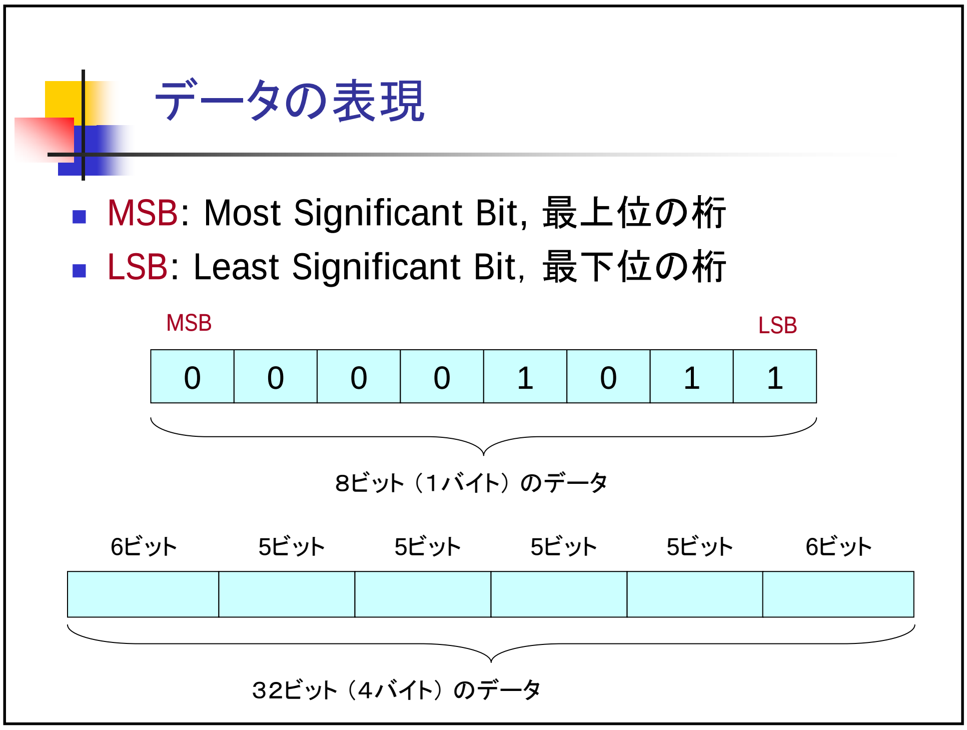
## データ形式

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50



## 整数(integer)の表現

- コンピュータは決まったビット幅を単位としてデータを処理.
  - 例えば，8ビットコンピュータ は，8ビット単位で処理
- nビットの整数表現は，2^n（2のn乗）種類の整数を表現できる.（しか表現できない！）
  - 8ビットであれば，2^8 ＝ 256 種類の整数.
  - 表現できる範囲には限りがある.
  - 効率の良い表現を利用して，資源を有効に活用する！
- 整数表現
  - 符号なし表現
  - 符号つき絶対値表現
  - 2の補数表現



## データの表現

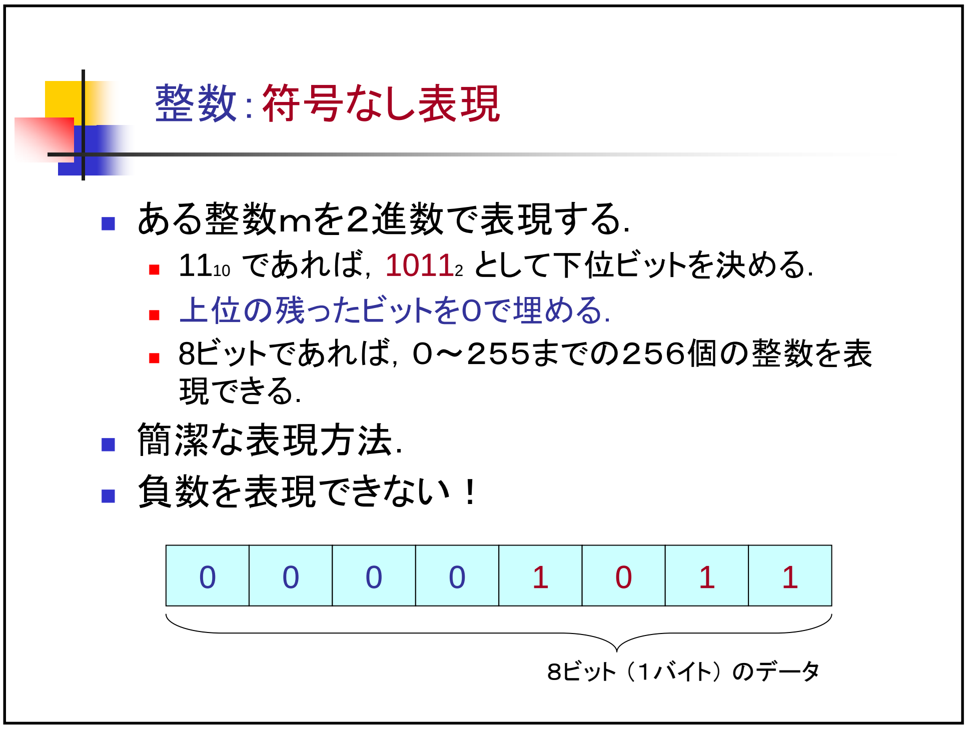
- MSB: Most Significant Bit, 最上位の桁
- LSB: Least Significant Bit, 最下位の桁

| MSB | | | | | | | LSB |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

8ビット（1バイト）のデータ

| 6ビット | 5ビット | 5ビット | 5ビット | 5ビット | 6ビット |
|---|---|---|---|---|---|
| | | | | | |

32ビット（4バイト）のデータ

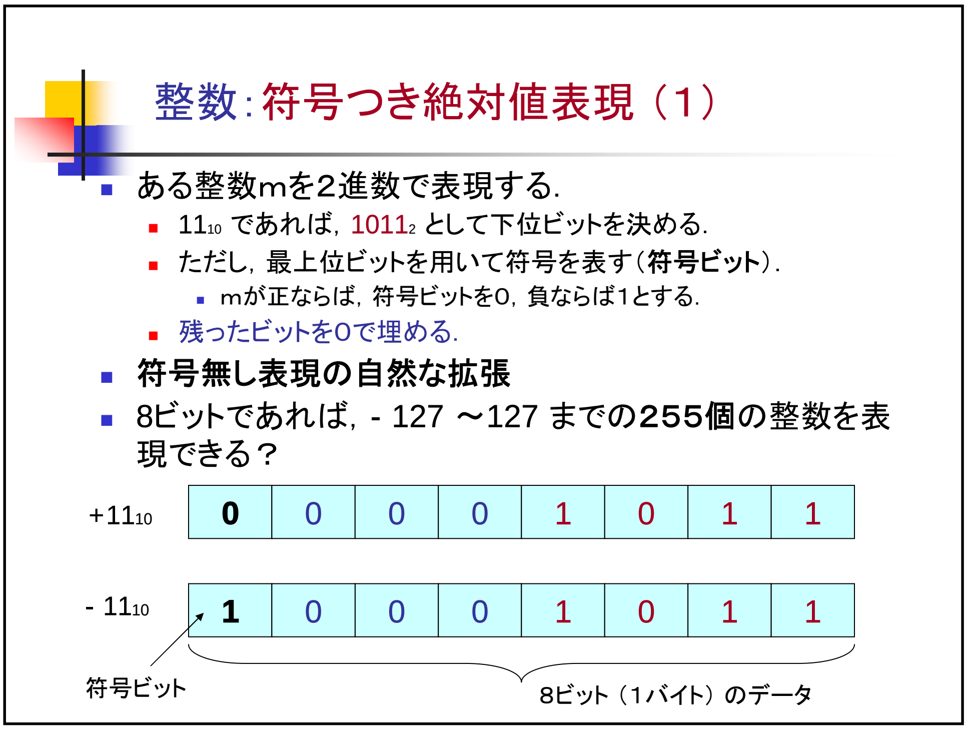


## 整数：符号なし表現

- ある整数mを2進数で表現する.
  - $11_{10}$ であれば，$1011_2$ として下位ビットを決める.
  - 上位の残ったビットを0で埋める.
  - 8ビットであれば，０～255までの256個の整数を表現できる.
- 簡潔な表現方法.
- 負数を表現できない！

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

8ビット（1バイト）のデータ



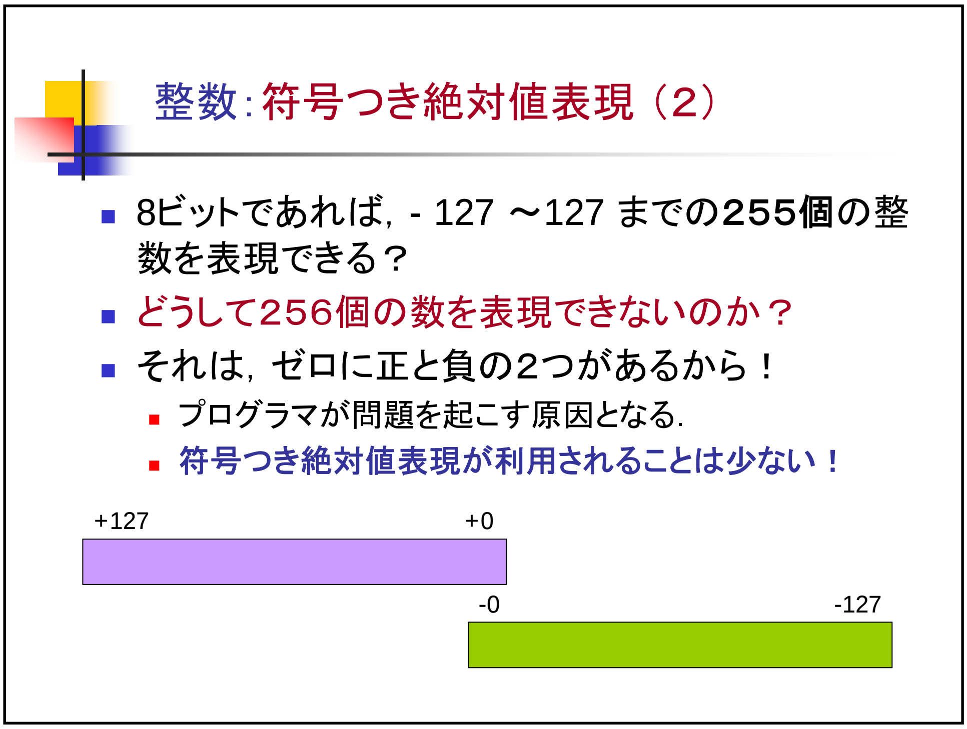
## 整数：符号つき絶対値表現（1）

- ある整数mを2進数で表現する.
  - $11_{10}$ であれば，$1011_2$ として下位ビットを決める.
  - ただし，最上位ビットを用いて符号を表す（**符号ビット**）.
    - mが正ならば，符号ビットを0，負ならば1とする.
  - 残ったビットを0で埋める.
- **符号無し表現の自然な拡張**
- 8ビットであれば，- 127 ～127 までの**255個**の整数を表現できる？

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $+11_{10}$ | **0** | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| $-11_{10}$ | **1** | 0 | 0 | 0 | 1 | 0 | 1 | 1 |

符号ビット

8ビット（1バイト）のデータ

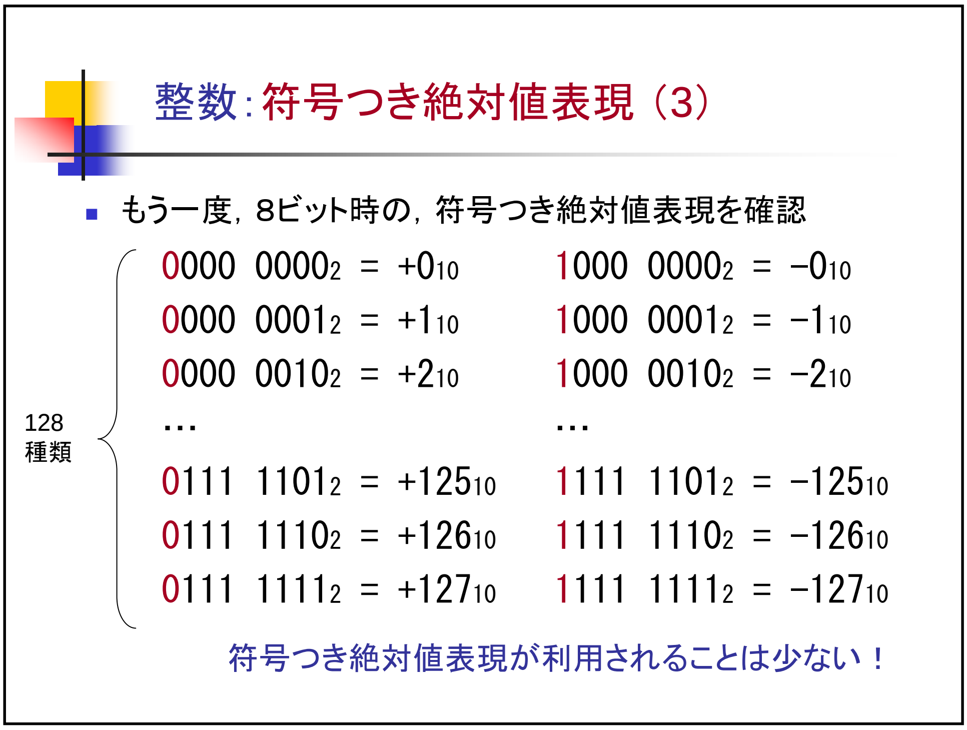


## 整数：符号つき絶対値表現（2）

- 8ビットであれば，- 127 ～127 までの255個の整数を表現できる？
- どうして256個の数を表現できないのか？
- それは，ゼロに正と負の2つがあるから！
  - プログラマが問題を起こす原因となる.
  - 符号つき絶対値表現が利用されることは少ない！

+127 +0

-0 -127



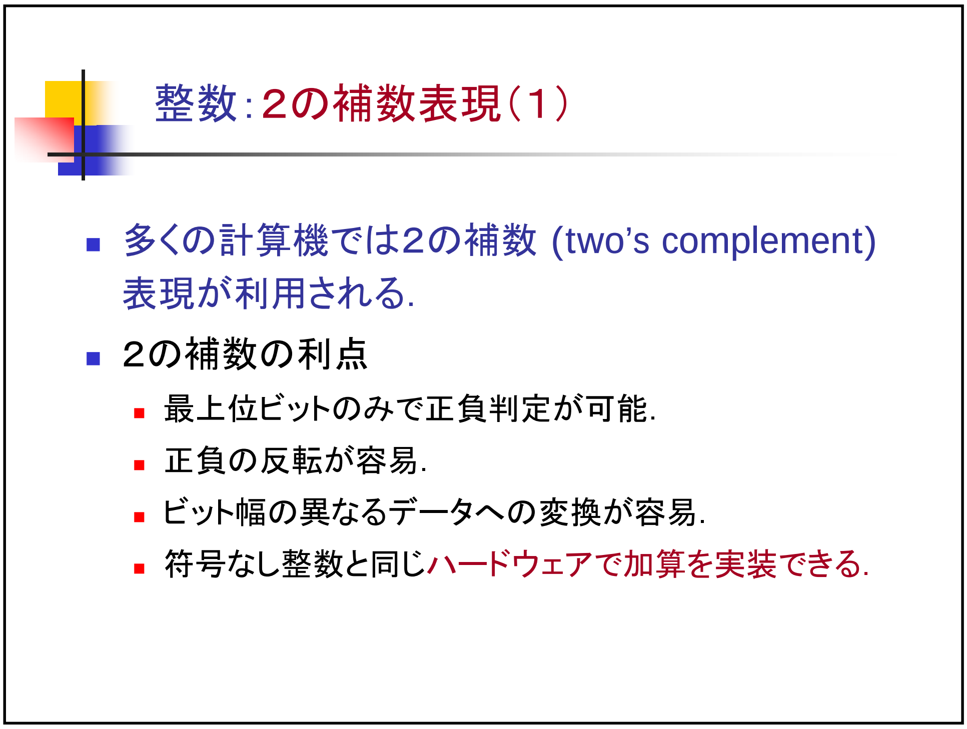
## 整数：符号つき絶対値表現（3）

- もう一度，8ビット時の，符号つき絶対値表現を確認

128種類

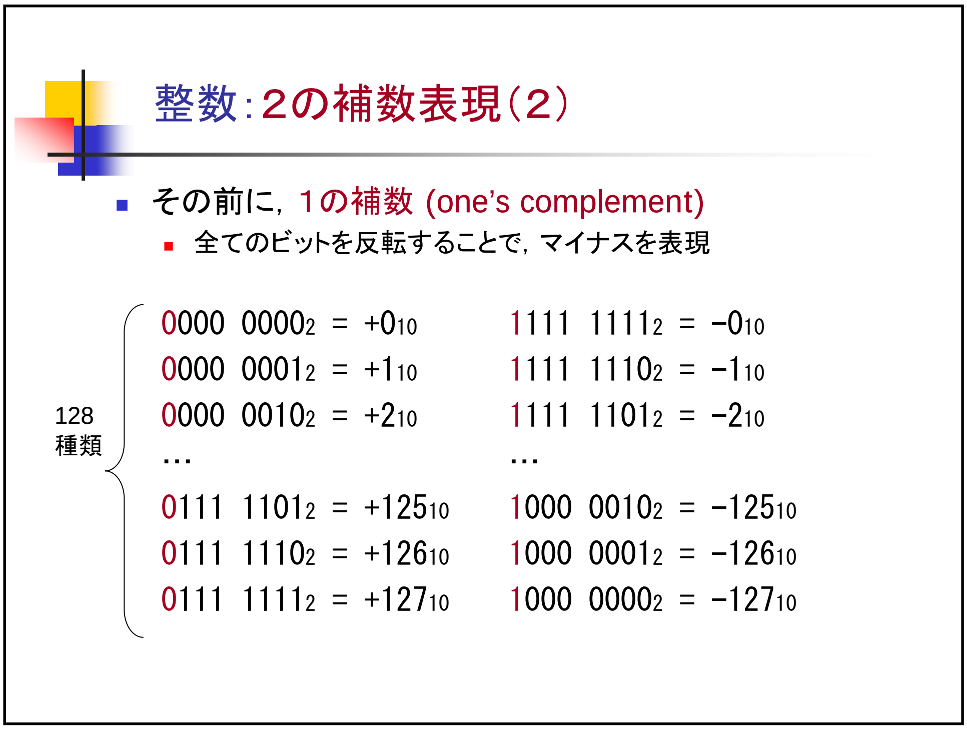
| | |
|---|---|
| $0000\ 0000_2 = +0_{10}$ | $1000\ 0000_2 = -0_{10}$ |
| $0000\ 0001_2 = +1_{10}$ | $1000\ 0001_2 = -1_{10}$ |
| $0000\ 0010_2 = +2_{10}$ | $1000\ 0010_2 = -2_{10}$ |
| … | … |
| $0111\ 1101_2 = +125_{10}$ | $1111\ 1101_2 = -125_{10}$ |
| $0111\ 1110_2 = +126_{10}$ | $1111\ 1110_2 = -126_{10}$ |
| $0111\ 1111_2 = +127_{10}$ | $1111\ 1111_2 = -127_{10}$ |

符号つき絶対値表現が利用されることは少ない！



## 整数：2の補数表現（1）

- 多くの計算機では2の補数 (two’s complement) 表現が利用される.
- 2の補数の利点
  - 最上位ビットのみで正負判定が可能.
  - 正負の反転が容易.
  - ビット幅の異なるデータへの変換が容易.
  - 符号なし整数と同じハードウェアで加算を実装できる.

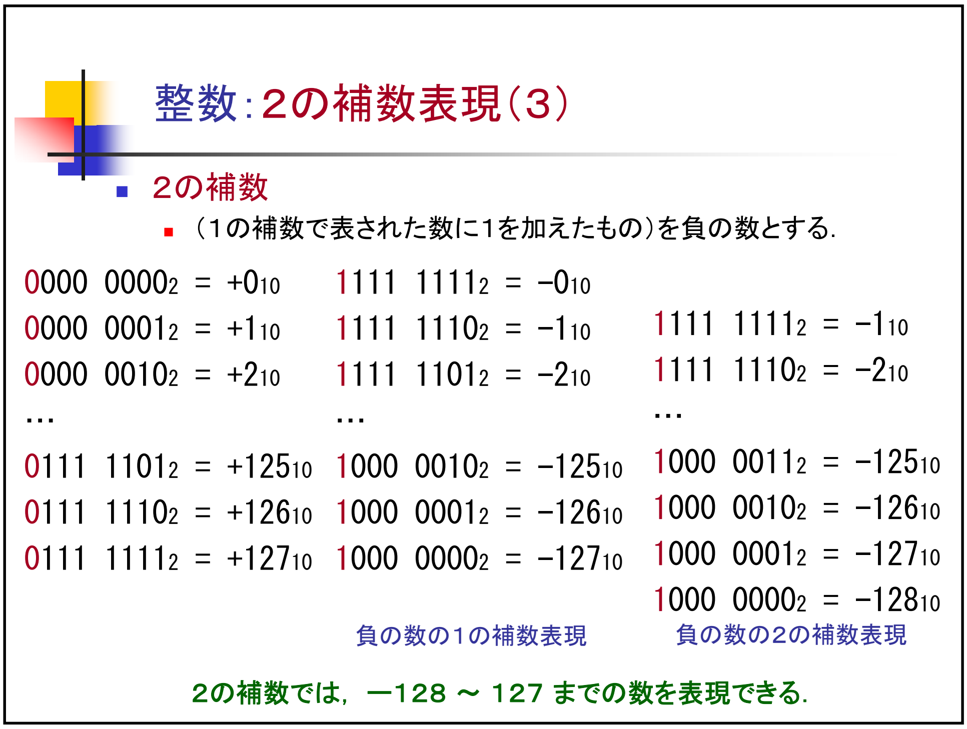


## 整数：2の補数表現（2）

- その前に，1の補数 (one’s complement)
  - 全てのビットを反転することで，マイナスを表現

128
種類

| | |
|---|---|
| $0000\ 0000_2 = +0_{10}$ | $1111\ 1111_2 = -0_{10}$ |
| $0000\ 0001_2 = +1_{10}$ | $1111\ 1110_2 = -1_{10}$ |
| $0000\ 0010_2 = +2_{10}$ | $1111\ 1101_2 = -2_{10}$ |
| … | … |
| $0111\ 1101_2 = +125_{10}$ | $1000\ 0010_2 = -125_{10}$ |
| $0111\ 1110_2 = +126_{10}$ | $1000\ 0001_2 = -126_{10}$ |
| $0111\ 1111_2 = +127_{10}$ | $1000\ 0000_2 = -127_{10}$ |

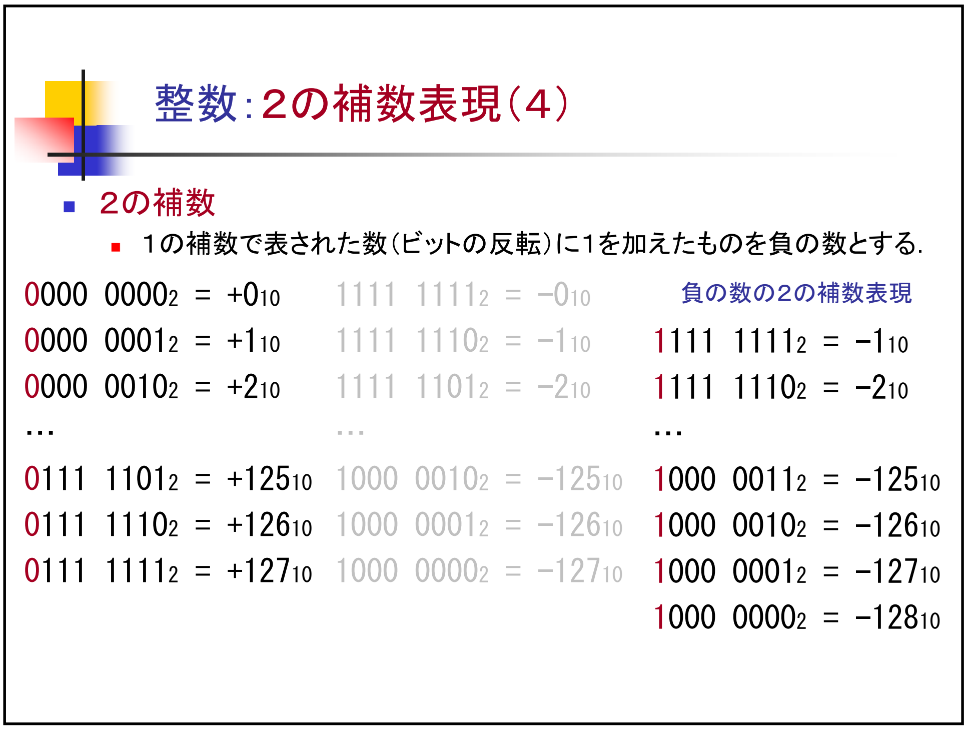


## 整数：2の補数表現（3）

- 2の補数
  - （1の補数で表された数に1を加えたもの）を負の数とする．

| | | |
|---|---|---|
| $0000\ 0000_2 = +0_{10}$ | $1111\ 1111_2 = -0_{10}$ | |
| $0000\ 0001_2 = +1_{10}$ | $1111\ 1110_2 = -1_{10}$ | $1111\ 1111_2 = -1_{10}$ |
| $0000\ 0010_2 = +2_{10}$ | $1111\ 1101_2 = -2_{10}$ | $1111\ 1110_2 = -2_{10}$ |
| … | … | … |
| $0111\ 1101_2 = +125_{10}$ | $1000\ 0010_2 = -125_{10}$ | $1000\ 0011_2 = -125_{10}$ |
| $0111\ 1110_2 = +126_{10}$ | $1000\ 0001_2 = -126_{10}$ | $1000\ 0010_2 = -126_{10}$ |
| $0111\ 1111_2 = +127_{10}$ | $1000\ 0000_2 = -127_{10}$ | $1000\ 0001_2 = -127_{10}$ |
| | | $1000\ 0000_2 = -128_{10}$ |
| | 負の数の１の補数表現 | 負の数の2の補数表現 |

**2の補数では，―128 ～ 127 までの数を表現できる．**



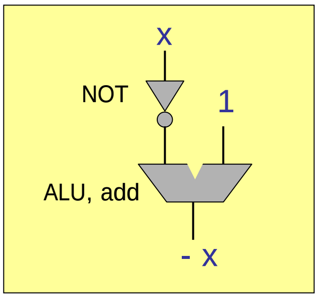
## 整数：2の補数表現（4）

- 2の補数
  - 1の補数で表された数（ビットの反転）に1を加えたものを負の数とする．

| | | 負の数の2の補数表現 |
|---|---|---|
| $0000\ 0000_2 = +0_{10}$ | $1111\ 1111_2 = -0_{10}$ | |
| $0000\ 0001_2 = +1_{10}$ | $1111\ 1110_2 = -1_{10}$ | $1111\ 1111_2 = -1_{10}$ |
| $0000\ 0010_2 = +2_{10}$ | $1111\ 1101_2 = -2_{10}$ | $1111\ 1110_2 = -2_{10}$ |
| … | … | … |
| $0111\ 1101_2 = +125_{10}$ | $1000\ 0010_2 = -125_{10}$ | $1000\ 0011_2 = -125_{10}$ |
| $0111\ 1110_2 = +126_{10}$ | $1000\ 0001_2 = -126_{10}$ | $1000\ 0010_2 = -126_{10}$ |
| $0111\ 1111_2 = +127_{10}$ | $1000\ 0000_2 = -127_{10}$ | $1000\ 0001_2 = -127_{10}$ |
| | | $1000\ 0000_2 = -128_{10}$ |

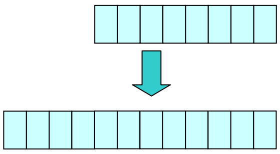
## 整数：2の補数表現（5）

- 2の補数
  - 1の補数で表された数（ビットの反転）に1を加えたものを負の数とする.
- 2の補数表現では，正負の反転を簡潔に実現できる！
  - 正数から負数への変換
    - 2進数表現の1と0を反転する.
    - 得られたデータに1を加える.
  - 負数から正数への変換
    - 2進数表現の1と0を反転する.
    - 得られたデータに1を加える.



## 整数：2の補数表現（6）

- 符号拡張
  - ビット幅の異なるデータへの変換
  - 例：8ビットから12ビットのデータへの変換
- 符号拡張の処理
  - ビット幅を増やすときには，最上位ビットの値で補填すればよい.



$1111\ 1111_2 = -1_{10}$

$1111\ 1110_2 = -2_{10}$

…

$1000\ 0011_2 = -125_{10}$

$1000\ 0010_2 = -126_{10}$

$1000\ 0001_2 = -127_{10}$

$1000\ 0000_2 = -128_{10}$

符号拡張

$1111\ 1111\ 1111_2 = -1_{10}$

$1111\ 1111\ 1110_2 = -2_{10}$

…

$1111\ 1000\ 0011_2 = -125_{10}$

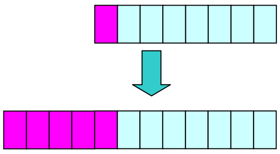
$1111\ 1000\ 0010_2 = -126_{10}$

$1111\ 1000\ 0001_2 = -127_{10}$

$1111\ 1000\ 0000_2 = -128_{10}$

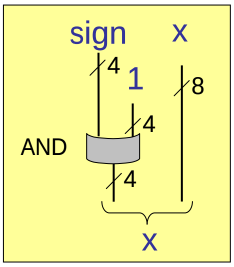
## 整数：2の補数表現（7）

- 符号拡張
  - ビット幅の異なるデータへの変換
  - 例：8ビットから12ビットのデータへの変換
- 符号拡張の処理
  - ビット幅を増やすときには，最上位ビットの値で補填すればよい．



証明

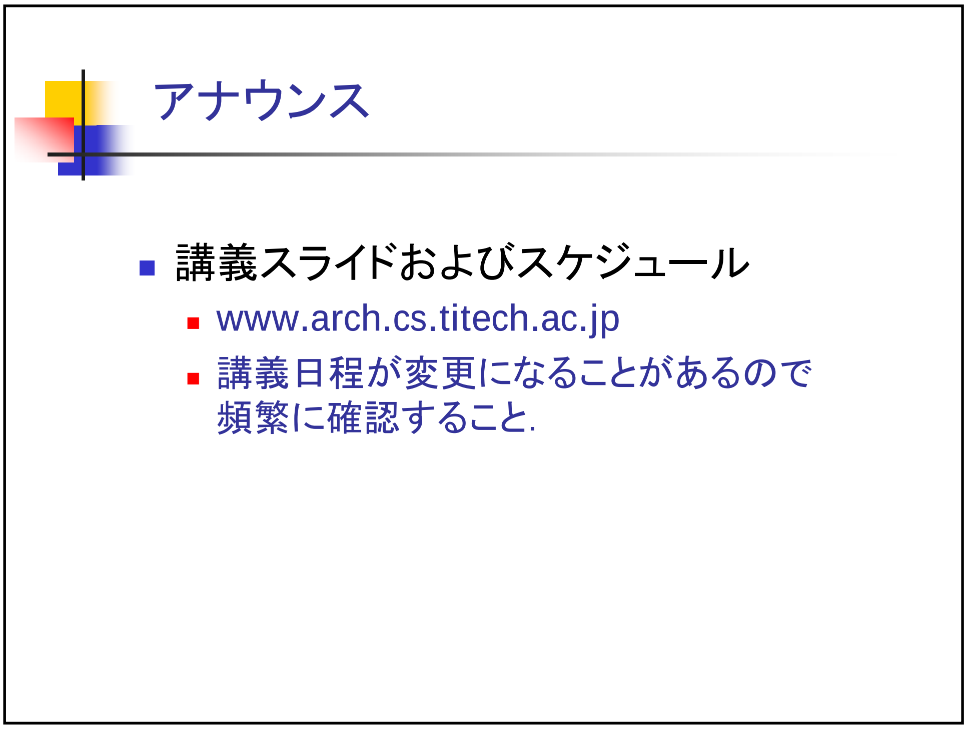
ある数 X が正の数の場合には自明．
それから．．．



## 講義項目

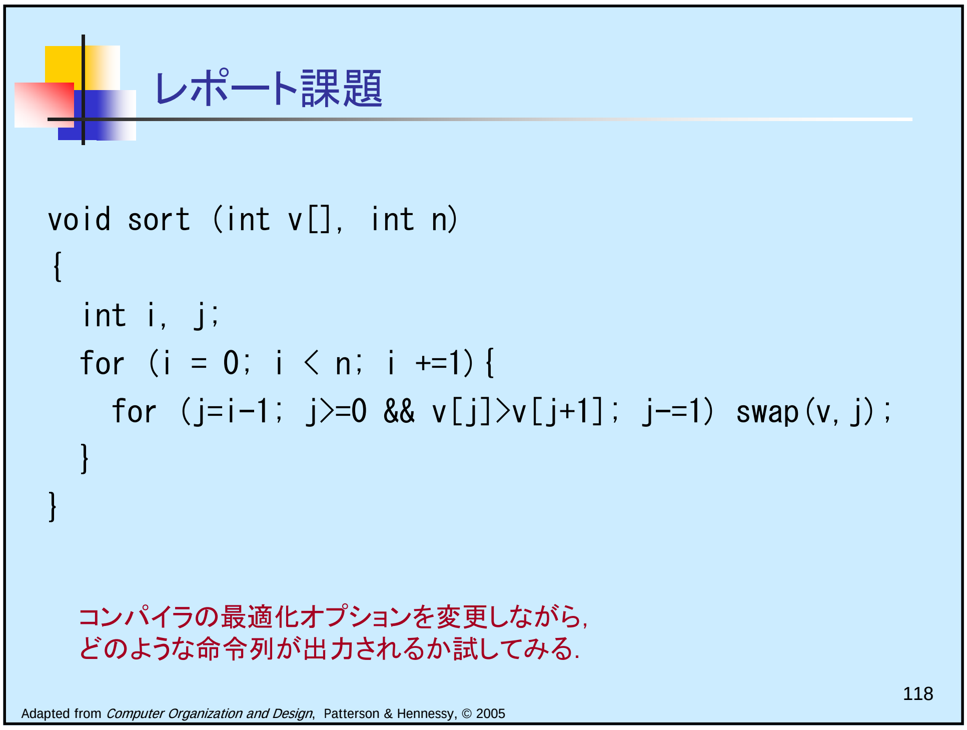
- 計算機システムの基本構成と動作原理
- (1) 命令形式，アドレス指定形式
- (2) 命令形式，データ形式
- メモリ1：半導体メモリシステム，ファイルメモリシステム
- メモリ2：記憶階層，キャッシュシステム
- メモリ3：仮想記憶システム（セグメンテーション，ページング，等）
- メモリ4：主記憶とファイルメモリの管理，多重仮想記憶，記憶保護
- 割り込み1：割り込みの必要性，割り込みの種類
- 割り込み2：割り込み処理の流れ
- 入出力制御1：チャネル，チャネルプログラム方式
- 入出力制御2：入出力動作の流れ，チャネル動作の効率化
- 入出力制御3：チャネルの種類，通信制御

レポートと期末試験により評価



## アナウンス

- 講義スライドおよびスケジュール
  - www.arch.cs.titech.ac.jp
  - 講義日程が変更になることがあるので頻繁に確認すること.



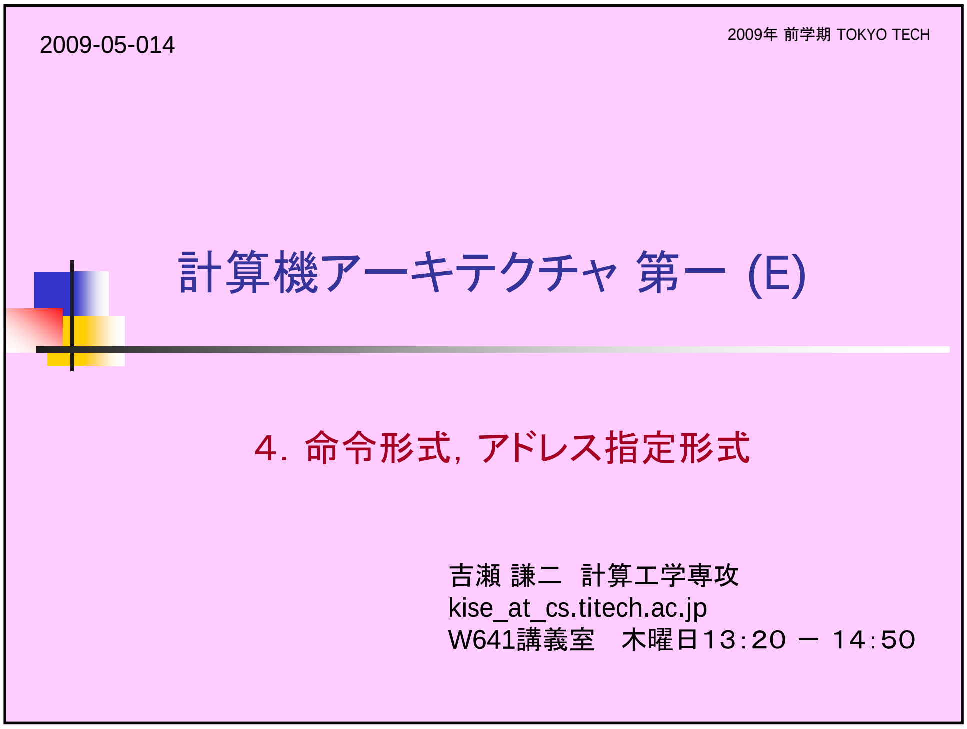
## レポート課題

```
void sort (int v[], int n)
{
  int i, j;
  for (i = 0; i < n; i +=1){
    for (j=i-1; j>=0 && v[j]>v[j+1]; j-=1) swap(v,j);
  }
}
```

コンパイラの最適化オプションを変更しながら,
どのような命令列が出力されるか試してみる.

118

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



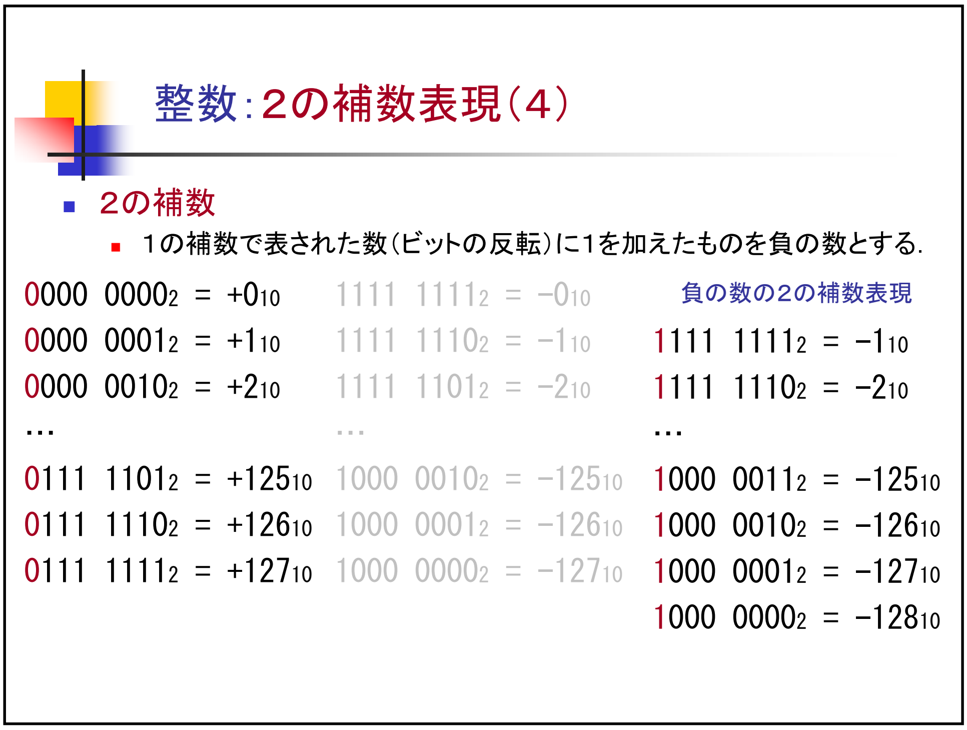
2009-05-014

2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 4. 命令形式，アドレス指定形式

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 ― 14:50



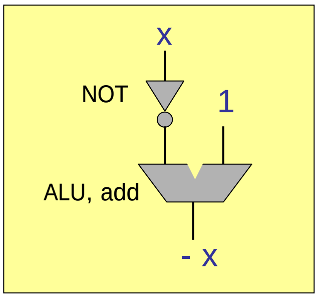
## 整数：2の補数表現（4）

- 2の補数
  - 1の補数で表された数（ビットの反転）に1を加えたものを負の数とする.

| | | 負の数の2の補数表現 |
|---|---|---|
| 0000 0000$_2$ = +0$_{10}$ | 1111 1111$_2$ = −0$_{10}$ | |
| 0000 0001$_2$ = +1$_{10}$ | 1111 1110$_2$ = −1$_{10}$ | 1111 1111$_2$ = −1$_{10}$ |
| 0000 0010$_2$ = +2$_{10}$ | 1111 1101$_2$ = −2$_{10}$ | 1111 1110$_2$ = −2$_{10}$ |
| … | … | … |
| 0111 1101$_2$ = +125$_{10}$ | 1000 0010$_2$ = −125$_{10}$ | 1000 0011$_2$ = −125$_{10}$ |
| 0111 1110$_2$ = +126$_{10}$ | 1000 0001$_2$ = −126$_{10}$ | 1000 0010$_2$ = −126$_{10}$ |
| 0111 1111$_2$ = +127$_{10}$ | 1000 0000$_2$ = −127$_{10}$ | 1000 0001$_2$ = −127$_{10}$ |
| | | 1000 0000$_2$ = −128$_{10}$ |

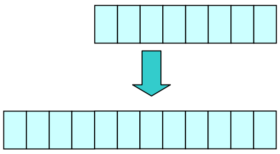
## 整数：2の補数表現(5)

- 2の補数
  - 1の補数で表された数(ビットの反転)に1を加えたものを負の数とする.
- 2の補数表現では，正負の反転を簡潔に実現できる！
  - 正数から負数への変換
    - 2進数表現の1と0を反転する.
    - 得られたデータに1を加える.
  - 負数から正数への変換
    - 2進数表現の1と0を反転する.
    - 得られたデータに1を加える.



## 整数：2の補数表現(6)

- 符号拡張
  - ビット幅の異なるデータへの変換
  - 例：8ビットから12ビットのデータへの変換
- 符号拡張の処理
  - ビット幅を増やすときには，最上位ビットの値で補填すればよい.



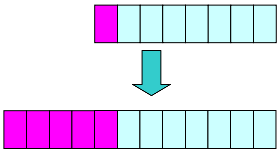
$1111\ 1111_2 = -1_{10}$
$1111\ 1110_2 = -2_{10}$
…
$1000\ 0011_2 = -125_{10}$
$1000\ 0010_2 = -126_{10}$
$1000\ 0001_2 = -127_{10}$
$1000\ 0000_2 = -128_{10}$

符号拡張 →

$1111\ 1111\ 1111_2 = -1_{10}$
$1111\ 1111\ 1110_2 = -2_{10}$
…
$1111\ 1000\ 0011_2 = -125_{10}$
$1111\ 1000\ 0010_2 = -126_{10}$
$1111\ 1000\ 0001_2 = -127_{10}$
$1111\ 1000\ 0000_2 = -128_{10}$

## 整数：2の補数表現（7）

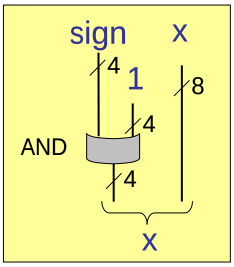
- 符号拡張
  - ビット幅の異なるデータへの変換
  - 例：8ビットから12ビットのデータへの変換



- 符号拡張の処理
  - ビット幅を増やすときには，最上位ビットの値で補填すればよい．

証明

ある数 X が正の数の場合には自明．
それから．．．



## 2の補数の加算（1）

- 符号を意識することなく，符号なし整数の加算と同様に計算できる．

桁上げ → 0 0 0 0 1 1 0

$$
\begin{array}{rl}
 & 0000\ 0111_2 = 7_{10} \\
+ & 0000\ 0110_2 = 6_{10} \\
\hline
 & 0000\ 1101_2 = 13_{10}
\end{array}
$$

## 2の補数の加算（2）

- 符号を意識することなく，符号なし整数の加算と同様に計算できる.

桁上げ →

$$
\begin{array}{rrrl}
 & 1111 & 110 & \\
 & 0000 & 0111_{2} & = \quad 7_{10} \\
+ & 1111 & 1010_{2} & = \ -6_{10} \\
\hline
 & 0000 & 0001_{2} & = \quad 1_{10}
\end{array}
$$

**減算：X – Y ＝ X + (- Y)**

## 整数の表現のまとめ

- 符号なし表現
- 符号つき絶対値表現
- 1の補数表現
- 2の補数表現
  - 最上位ビットのみで正負判定が可能.
  - 正負の反転が容易.
  - ビット幅の異なるデータへの変換が容易.
  - 符号なし整数と同じハードウェアで符号付き加算を実装できる.

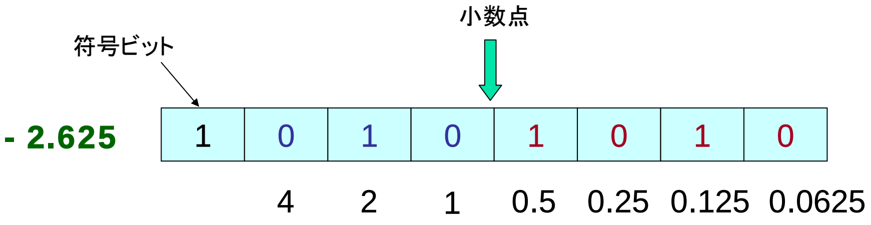
# 実数

- 少数を含む数値を取り扱う.
- 実数の例
  - 3.1419926… (π)
  - 0.000000001, **$1.0 \times 10^{-9}$**
  - 3,155,760,000, **$3.1556 \times 10^{9}$**

科学記数法：小数点の左側には数字を一つしか書かない.
科学記数法で書いた数値で先頭に0がこないものを正規化数と呼ぶ.

# 固定小数点表現

- あまり利用されない！
  - 小数点の位置を固定する.

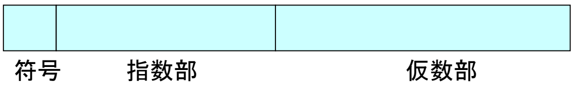


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **- 2.625** | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| | | 4 | 2 | 1 | 0.5 | 0.25 | 0.125 | 0.0625 |

## 浮動小数点表現（1）

- 小数点位置が変動
- 科学記数法で数値で先頭に0がこない正規化数を利用.

$$\underbrace{1.xxxxxxxxx}_{\text{仮数部}} \times 2^{\overbrace{yyyy}^{\text{指数部}}}$$

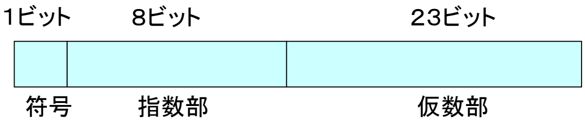


| 符号 | 指数部 | 仮数部 |
|---|---|---|

## 浮動小数点表現（2）

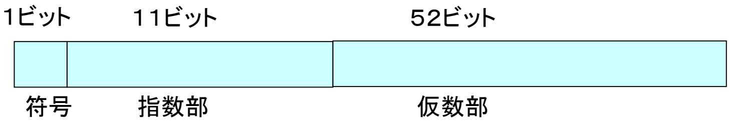
- IEEE754

単精度
（32ビット）

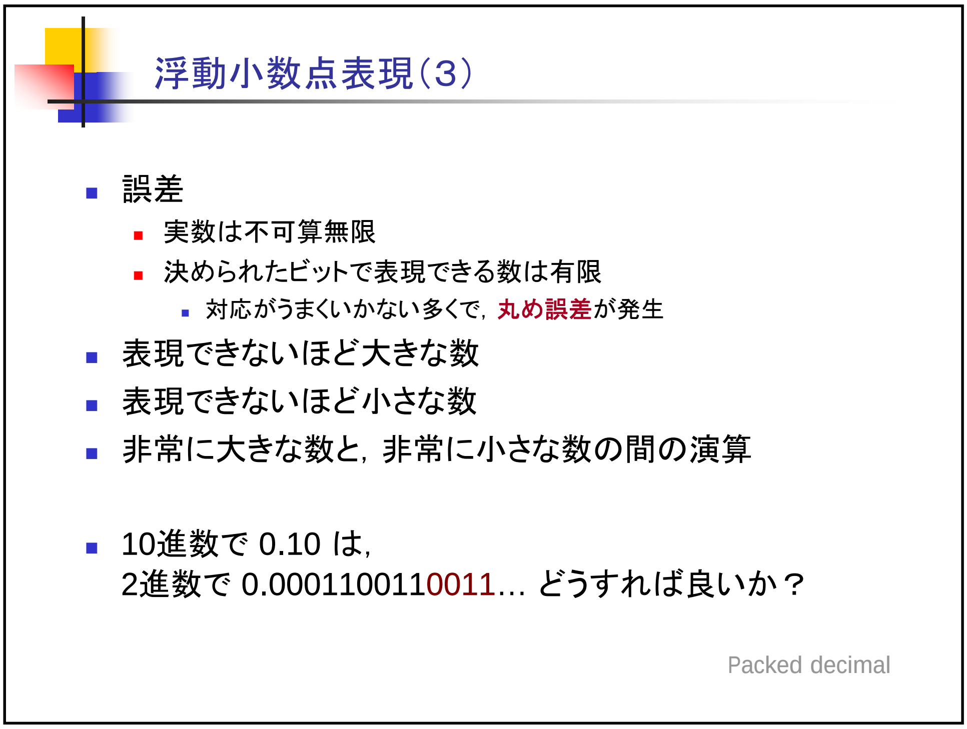


| 1ビット | 8ビット | 23ビット |
|---|---|---|
| 符号 | 指数部 | 仮数部 |

倍精度
（64ビット）



| 1ビット | 11ビット | 52ビット |
|---|---|---|
| 符号 | 指数部 | 仮数部 |

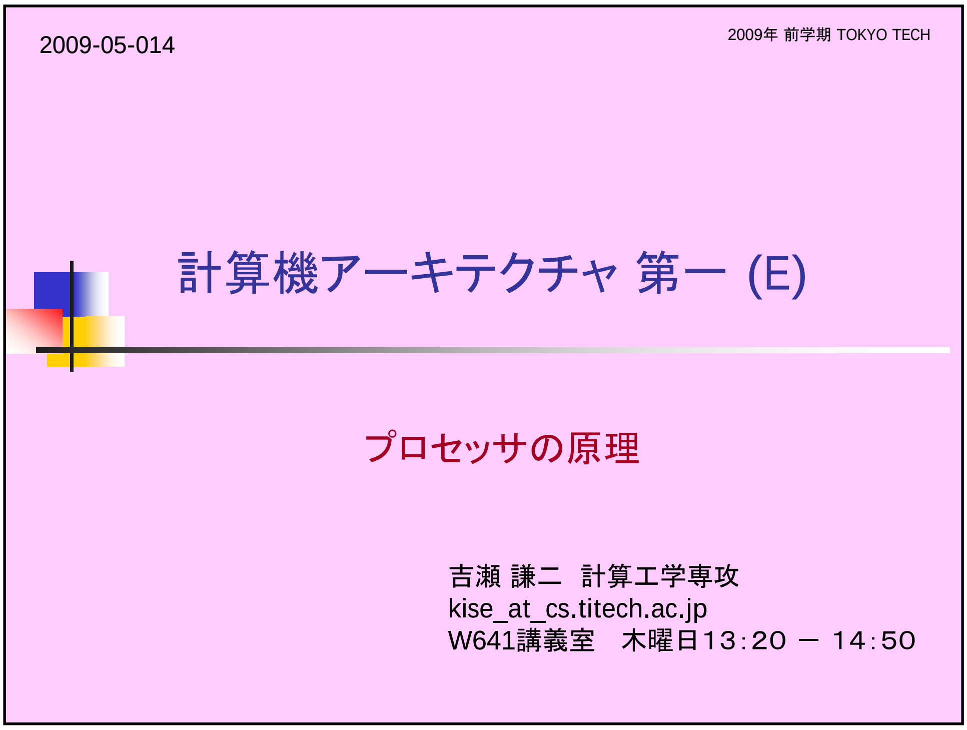


## 浮動小数点表現（3）

- 誤差
  - 実数は不可算無限
  - 決められたビットで表現できる数は有限
    - 対応がうまくいかない多くで，丸め誤差が発生
- 表現できないほど大きな数
- 表現できないほど小さな数
- 非常に大きな数と，非常に小さな数の間の演算

- 10進数で 0.10 は，
  2進数で 0.0001100110011… どうすれば良いか？

Packed decimal



2009-05-014

2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

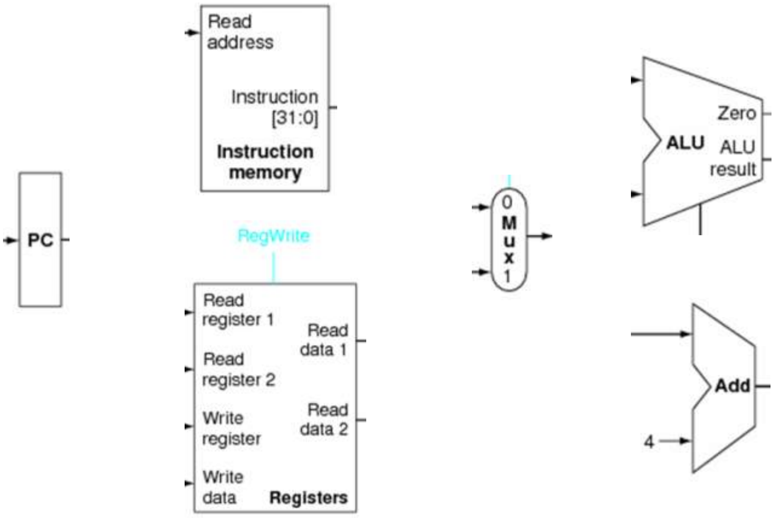
## プロセッサの原理

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

## MIPSの基本的な5つのステップ（ステージ）

- **IFステージ**
  メモリから命令をフェッチする.
- **IDステージ**
  命令をデコードしながら, レジスタを読み出す.
- **EXステージ**
  命令操作の実行またはアドレスの生成を行う.
- **MEMステージ**
  データ・メモリ中のオペランドにアクセスする.
- **WBステージ**
  結果をレジスタに書き込む.

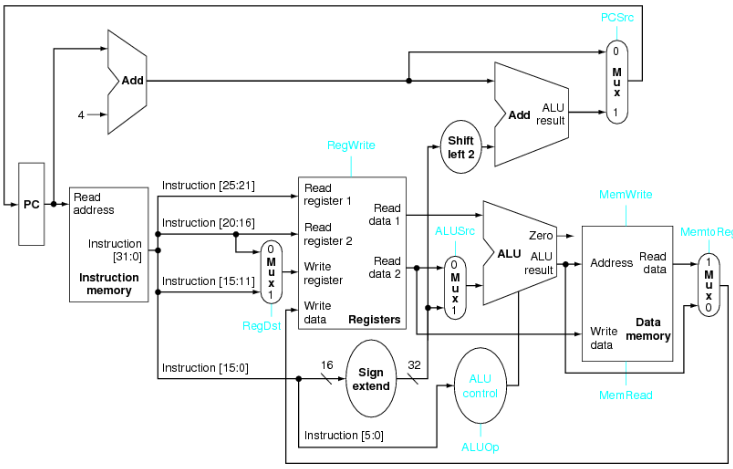
## プロセッサの主な構成要素



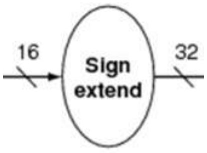
## プロセッサのデータパス（シングル・サイクル）

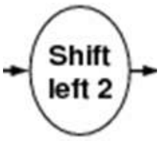
| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|

**add $t0, $s1, $s2 [ add $8, $17, $18 ]**



## プロセッサの構成要素（1）



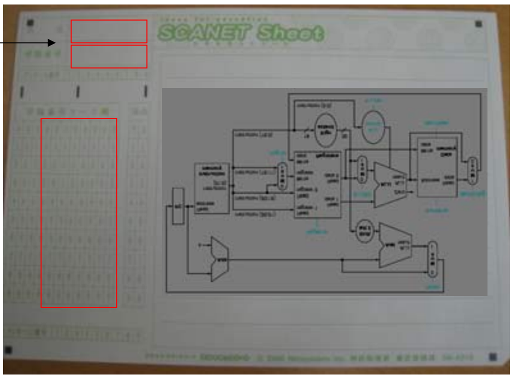


## Exercise

| | | | | |
|---|---|---|---|---|
| op | rs | rt | 16 bit immediate | I format |

**addi $t0, $t1, -1** **[ addi $8, $9, -1 ]**

氏名，学籍番号，
学籍番号マーク欄(右詰で)

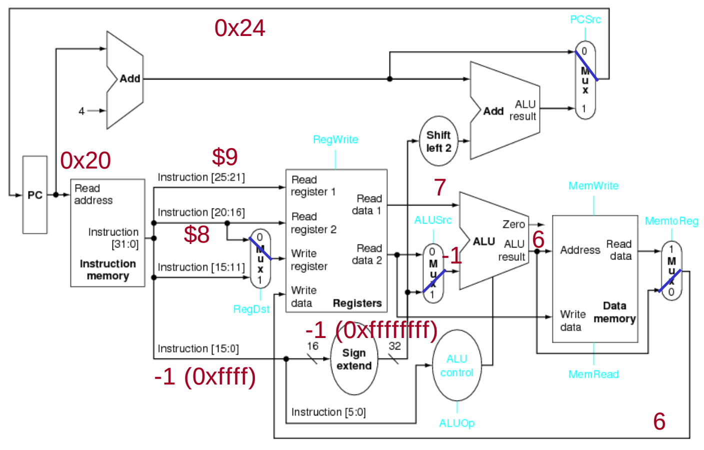


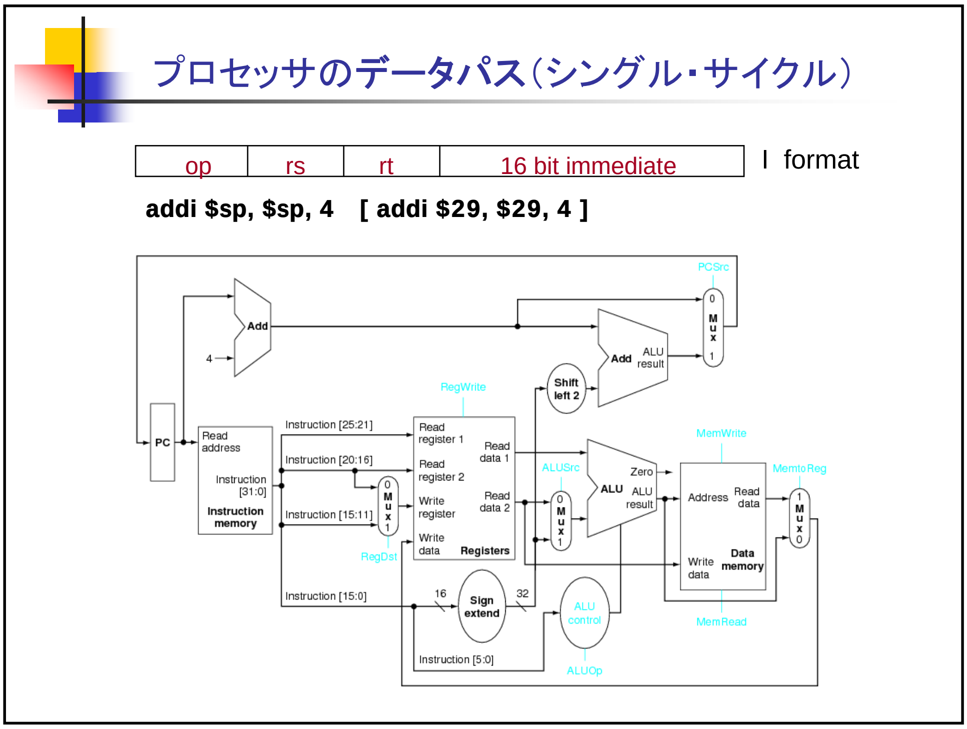
## Exercise

| | | | | |
|---|---|---|---|---|
| op | rs | rt | 16 bit immediate | I format |

**addi $t0, $t1, -1** **[ addi $8, $9, -1 ]**

**PC = 0x20**
**$9 = 7**



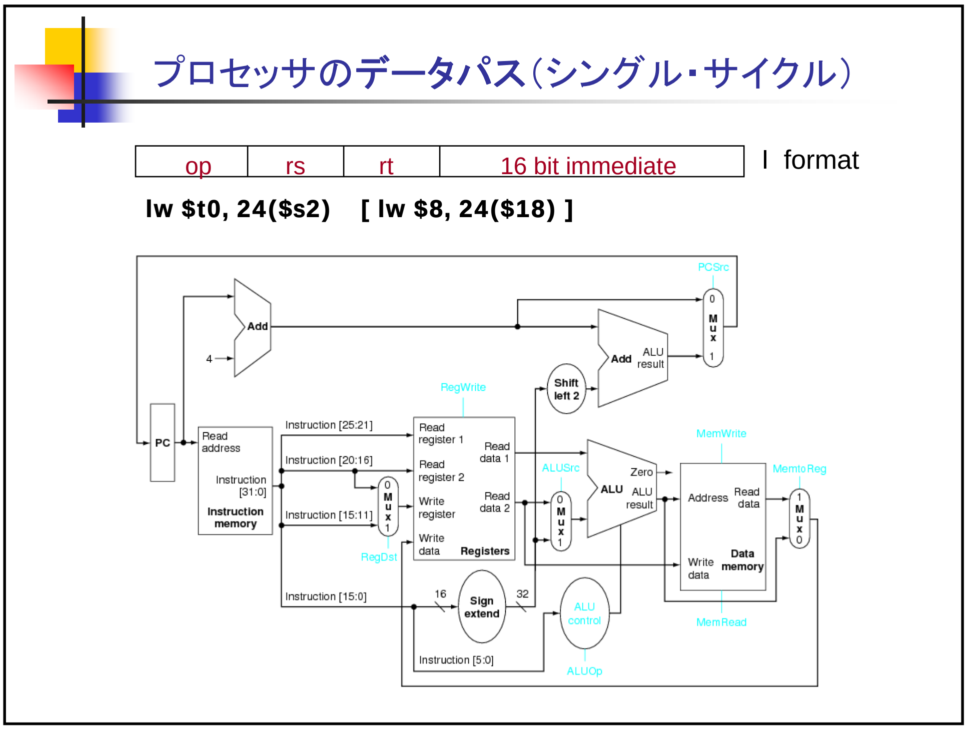


## プロセッサのデータパス（シングル・サイクル）

| op | rs | rt | 16 bit immediate |
|---|---|---|---|

I format

**addi $sp, $sp, 4   [ addi $29, $29, 4 ]**



## プロセッサのデータパス（シングル・サイクル）

| op | rs | rt | 16 bit immediate |
|---|---|---|---|

I format

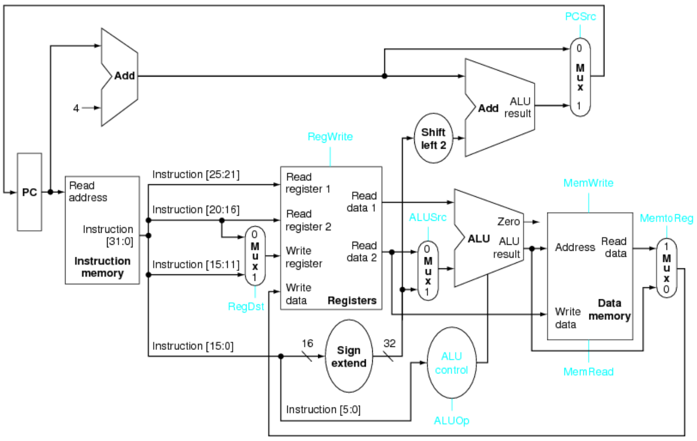
**lw $t0, 24($s2)   [ lw $8, 24($18) ]**

## プロセッサのデータパス（シングル・サイクル）

|  |  |  |  |
|---|---|---|---|
| op | rs | rt | 16 bit immediate |

I format

**sw $t0, 24($s2)  [ sw $8, 24($18) ]**

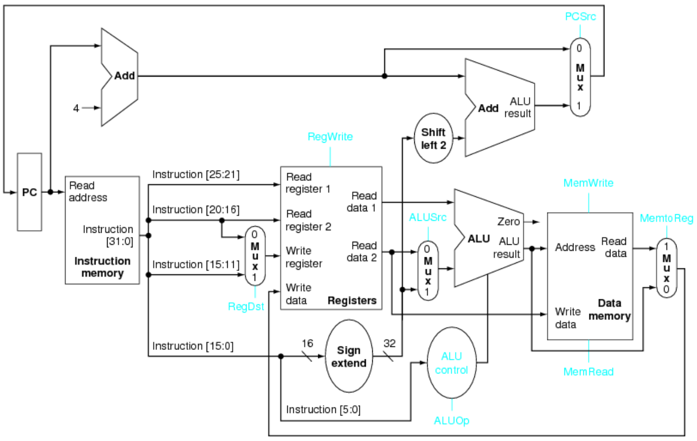


## プロセッサのデータパス（シングル・サイクル）

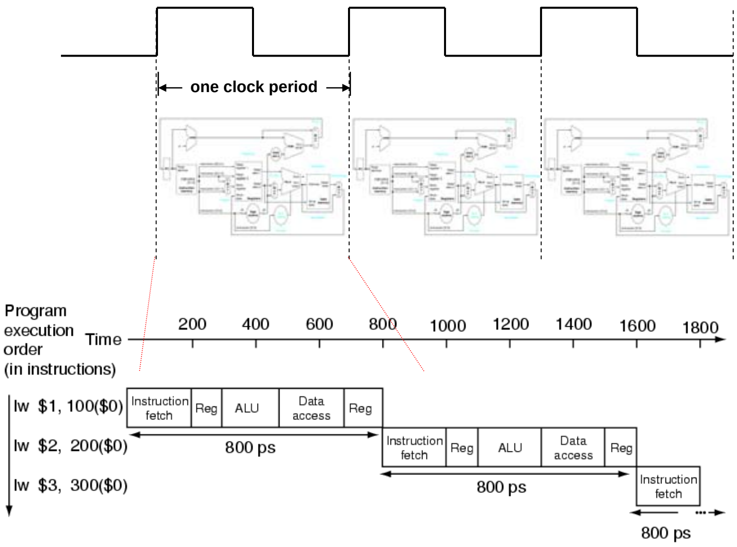
|  |  |  |  |
|---|---|---|---|
| op | rs | rt | 16 bit immediate |

I format

**beq $s0, $s1, Label  [beq $16, $17, Label ]**



## プロセッサのデータパス（シングル・サイクル）



## Sample program

コンパイラの最適化オプションを変更しながら，SimMipsで実行し，その実行サイクル数をみる．

```
#include <stdio.h>
int main(){
  int i;
  int sum = 0;

  for(i=1; i<=100; i++)  sum += i;

  return sum;
}
```

```
mipsel-linux-gcc –static –O0 main.c –o a.out
SimMips a.out
/home/share/cad/mipsel/usr/bin/mipsel-linux-gcc
```

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

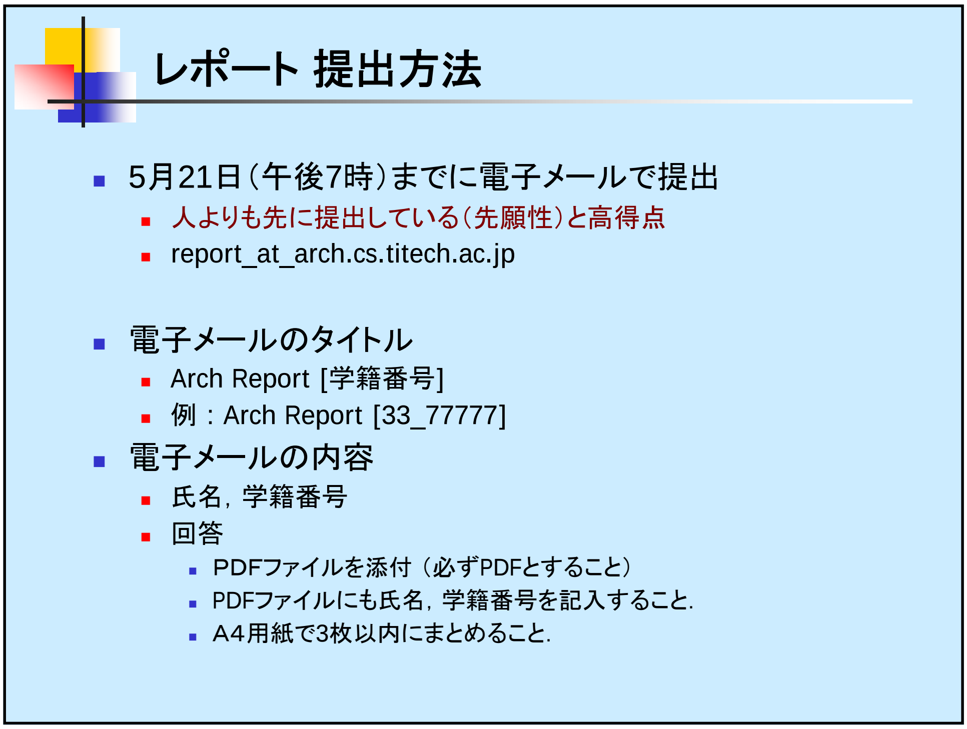
## レポート 問題

1. void max (int v[], int n)
   をクロスコンパイラにてMIPS命令セットにコンパイルし，コンパイルオプションによってどのように変化するかをまとめよ．また，SimMipsで実行し，実行サイクル数を比較せよ．
2. void sort (int v[], int n)
   をクロスコンパイラにてMIPS命令セットにコンパイルし，コンパイルオプションによってどのように変化するかをまとめよ．また，SimMipsで実行し，実行サイクル数を比較せよ．
3. 同様に，**複雑なアプリケーション**を作成し，それをクロスコンパイラにてMIPS命令セットにコンパイルし，コンパイルオプションによってどのように変化するかをまとめよ．また，SimMipsで実行し，実行サイクル数を比較せよ．
4. この課題の感想をまとめること.
5. レポートはA4用紙3枚以内にまとめること．（必ずPDFとすること）
   （2段組，コードは小さい文字でもかまわない．）

## 講義用の計算機環境

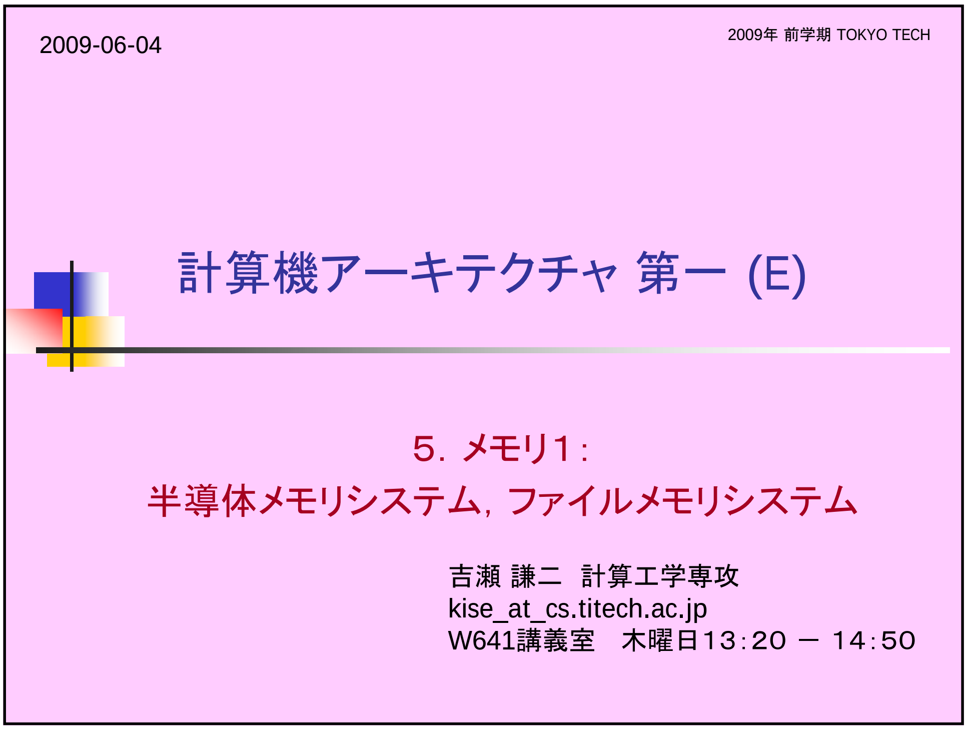
- 講義用の計算機
  - 131.112.16.56
  - ssh arche@131.112.16.56
    - **ユーザ名: arche**
    - **パスワードは講義時に連絡**
  - cd myname　　　(例: cd 06B77777)
  - cp –r /home/arche/v0.5.5 .
  - cd v0.5.5
  - memory.cc などを修正してコンパイル，実行
- 注意点
  - 計算機演習室からは外部にsshで接続できないかもしれません.
  - Windowsからは Tera Term などを利用してください.

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## レポート 提出方法

- 5月21日（午後7時）までに電子メールで提出
  - 人よりも先に提出している（先願性）と高得点
  - report_at_arch.cs.titech.ac.jp
- 電子メールのタイトル
  - Arch Report [学籍番号]
  - 例：Arch Report [33_77777]
- 電子メールの内容
  - 氏名，学籍番号
  - 回答
    - PDFファイルを添付（必ずPDFとすること）
    - PDFファイルにも氏名，学籍番号を記入すること.
    - A4用紙で3枚以内にまとめること.



2009-06-04

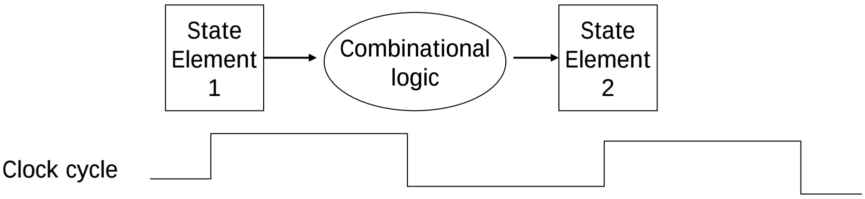
2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 5. メモリ1：
## 半導体メモリシステム，ファイルメモリシステム

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

# エッジトリガ方式による設計

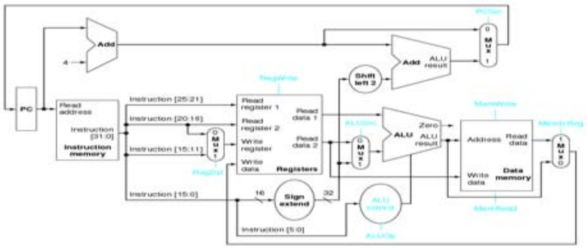


State Element 1

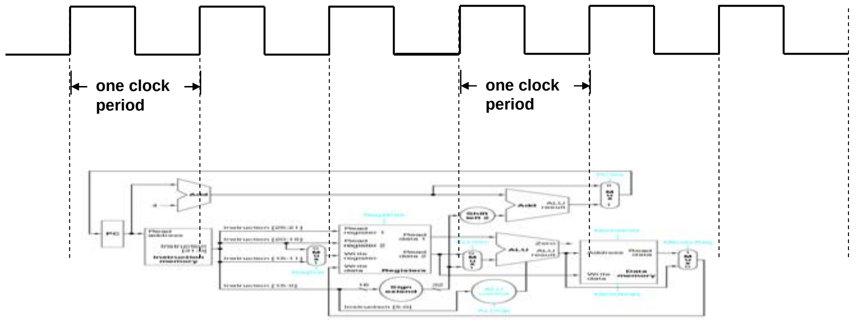
Combinational logic

State Element 2

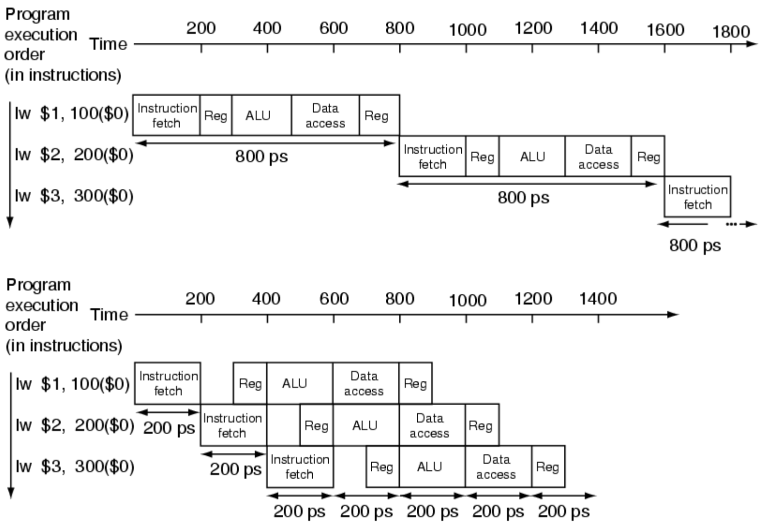
Clock cycle



# プロセッサのデータパス（マルチ・サイクル）



# パイプライン処理 (pipelining)



Program execution order (in instructions)

Time: 200 400 600 800 1000 1200 1400 1600 1800

lw $1, 100($0): Instruction fetch | Reg | ALU | Data access | Reg — 800 ps

lw $2, 200($0): Instruction fetch | Reg | ALU | Data access | Reg — 800 ps

lw $3, 300($0): Instruction fetch — 800 ps ...

Program execution order (in instructions)

Time: 200 400 600 800 1000 1200 1400

lw $1, 100($0): Instruction fetch | Reg | ALU | Data access | Reg

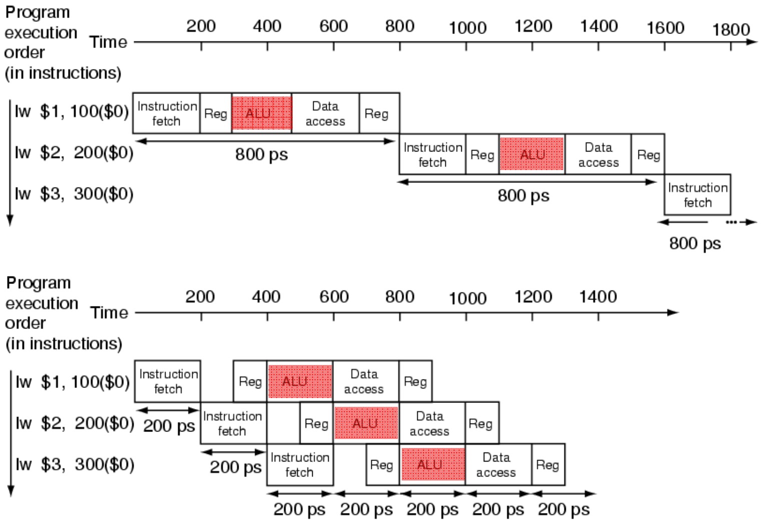
lw $2, 200($0): 200 ps | Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0): 200 ps | Instruction fetch | Reg | ALU | Data access | Reg

200 ps 200 ps 200 ps 200 ps 200 ps

151

# パイプライン処理 (pipelining)



Program execution order (in instructions)

Time: 200 400 600 800 1000 1200 1400 1600 1800

lw $1, 100($0): Instruction fetch | Reg | ALU | Data access | Reg — 800 ps

lw $2, 200($0): Instruction fetch | Reg | ALU | Data access | Reg — 800 ps

lw $3, 300($0): Instruction fetch — 800 ps ...

Program execution order (in instructions)

Time: 200 400 600 800 1000 1200 1400

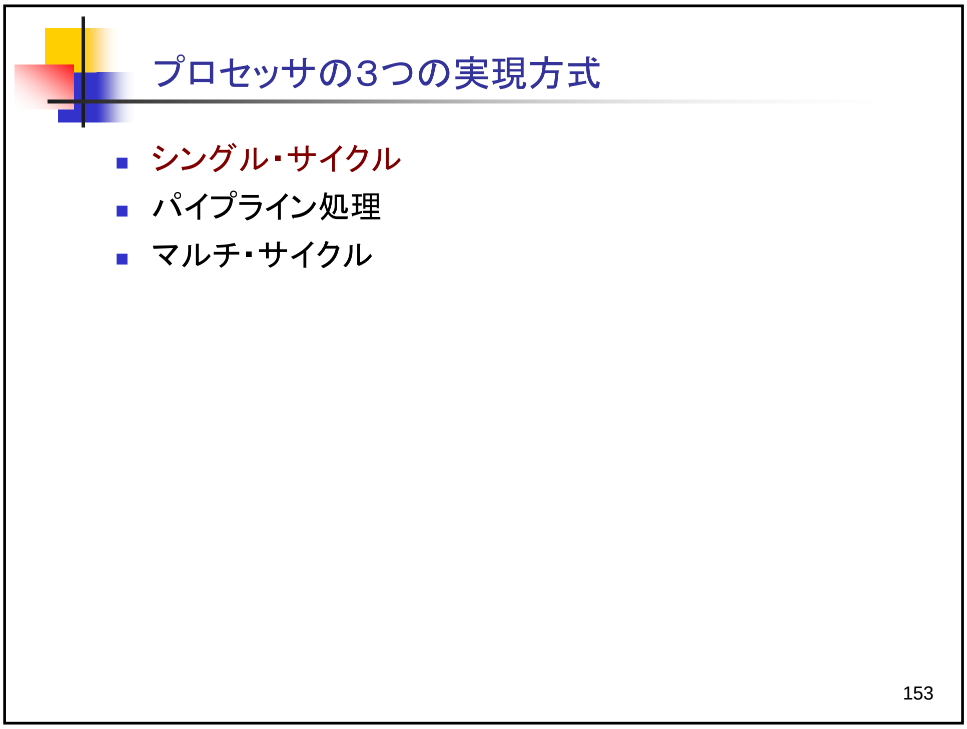
lw $1, 100($0): Instruction fetch | Reg | ALU | Data access | Reg

lw $2, 200($0): 200 ps | Instruction fetch | Reg | ALU | Data access | Reg

lw $3, 300($0): 200 ps | Instruction fetch | Reg | ALU | Data access | Reg

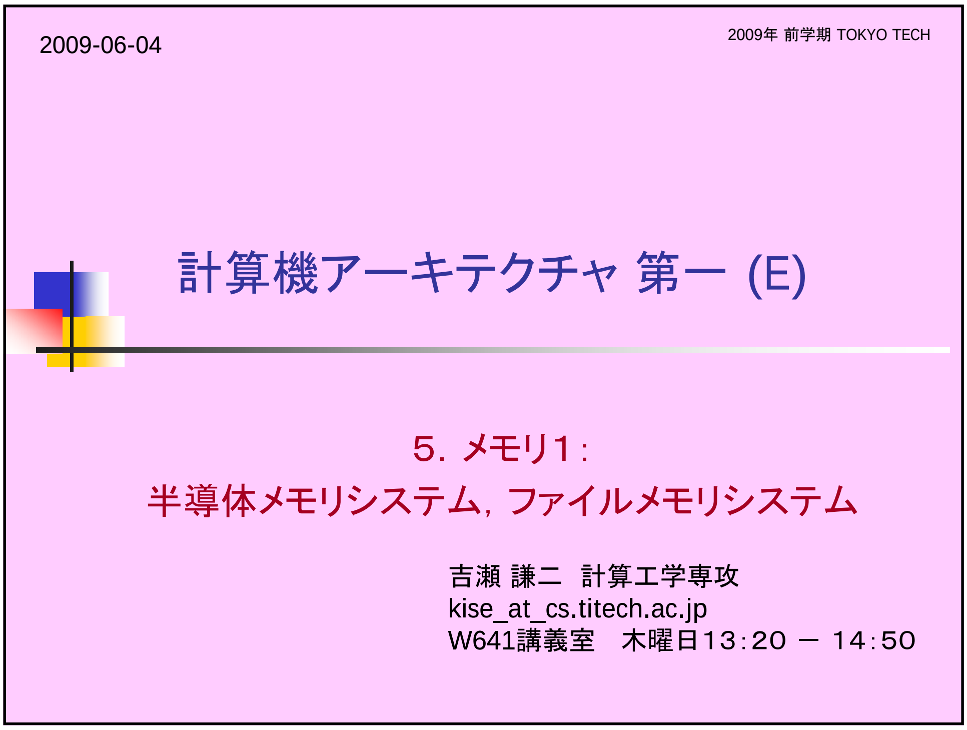
200 ps 200 ps 200 ps 200 ps 200 ps

152



## プロセッサの3つの実現方式

- シングル・サイクル
- パイプライン処理
- マルチ・サイクル



2009-06-04

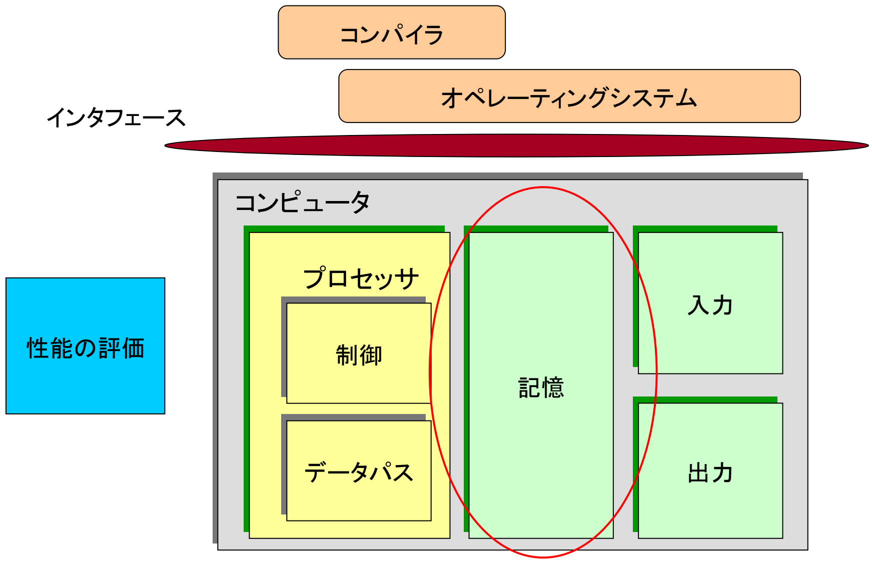
2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 5. メモリ1：
## 半導体メモリシステム，ファイルメモリシステム

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

## コンピュータ（ハードウェア）の古典的な要素



コンパイラ

オペレーティングシステム

インタフェース

性能の評価

コンピュータ

プロセッサ

制御

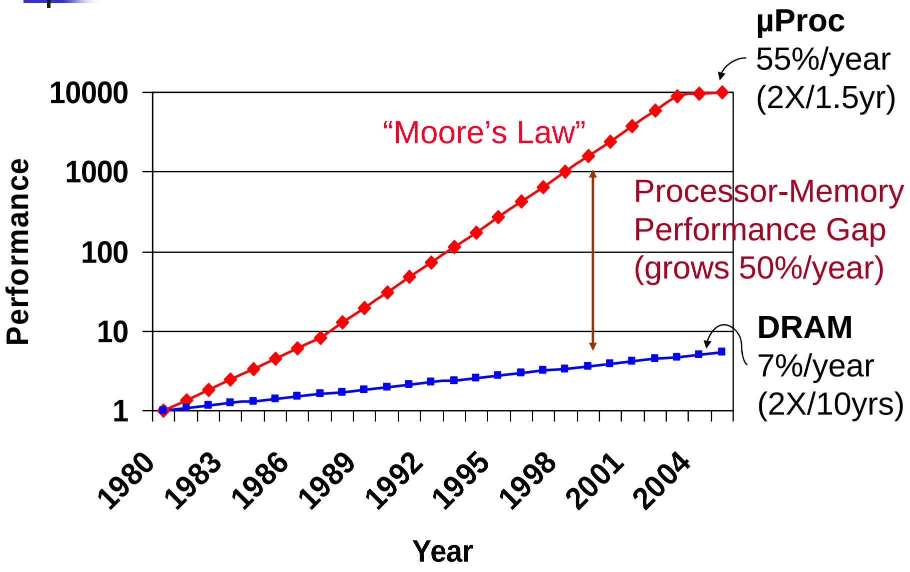
データパス

記憶

入力

出力

## Processor-Memory Performance Gap



µProc
55%/year
(2X/1.5yr)

“Moore’s Law”

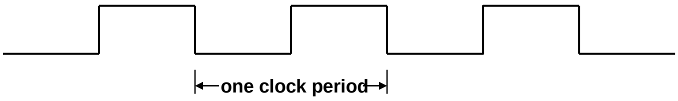
Processor-Memory
Performance Gap
(grows 50%/year)

DRAM
7%/year
(2X/10yrs)

## Machine Clock Rate

- **Clock rate** (MHz, GHz) is inverse of clock cycle time (**clock period**)

**Clock period** = 1 / (**clock rate**)



| | | |
|---|---|---|
| 10 nsec clock cycle | => | 100 MHz clock rate |
| 5 nsec clock cycle | => | 200 MHz clock rate |
| 2 nsec clock cycle | => | 500 MHz clock rate |
| 1 nsec clock cycle | => | 1 GHz clock rate |
| 500 psec clock cycle | => | 2 GHz clock rate |
| 250 psec clock cycle | => | 4 GHz clock rate |
| 200 psec clock cycle | => | 5 GHz clock rate |

## Clock Cycles per Instruction, **CPI**

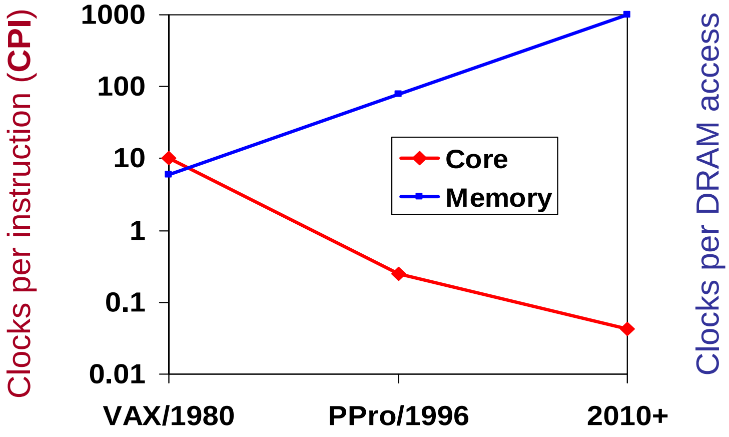
- Not all instructions take the same amount of time to execute

$$\text{\# CPU clock cycles for a program} = \text{\# Instructions for a program} \times \text{Average clock cycles per instruction}$$

- **Clock cycles per instruction** (**CPI**) – the average number of clock cycles each instruction takes to execute
  - CPI = 10.0
  - CPI = 1.0
  - CPI = 0.5
  - CPI = 0.1

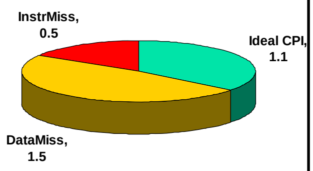
## The “**Memory Wall**”

- **Arithmetic** vs **DRAM** speed gap continues to grow



## Memory Performance Impact

- A processor executes at
  - ideal CPI = 1.1
  - 50% arith/logic, 20% control, **30% ld/st**
  - **10%** of data memory operations miss with a **50 cycle miss penalty**



- CPI = ideal CPI + average stalls per instruction
  = 1.1(cycle) + ( 0.30 x 0.10 x 50 (cycle/miss) )
  = **1.1 cycle** + **1.5 cycle** = **2.6**
- **58% of the time** the processor is stalled waiting for memory!
- A 1% instruction miss rate would add an additional **?** to the CPI!
- Answer 0.5

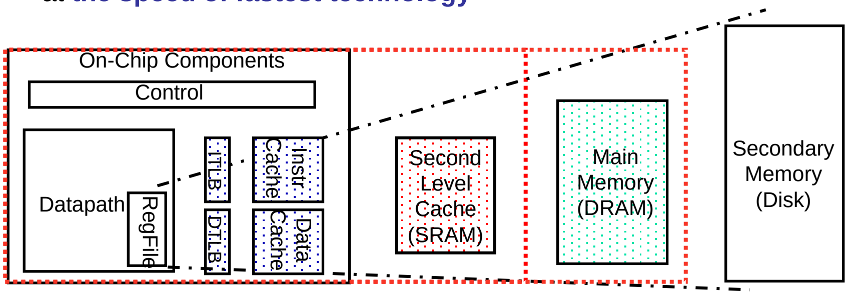
## The Memory Hierarchy **Goal**

- **Fact**:
  **Large memories are slow** and
  **fast memories are small**

- How do we create a memory that gives the illusion of being large, cheap and fast ?
  - With **hierarchy**（階層）
  - With **parallelism**（並列性）

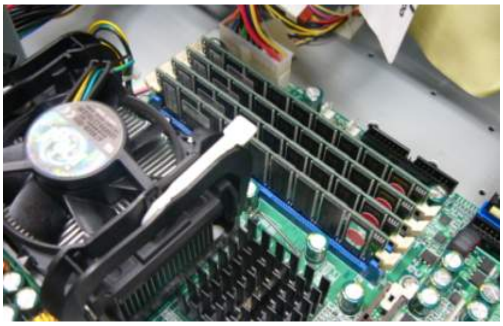
## A Typical Memory Hierarchy

□ By taking advantage of **the principle of locality**（局所性）
- Present **much memory** in **the cheapest technology**
- at **the speed of fastest technology**

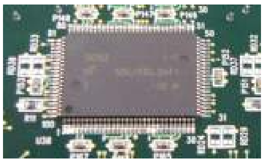


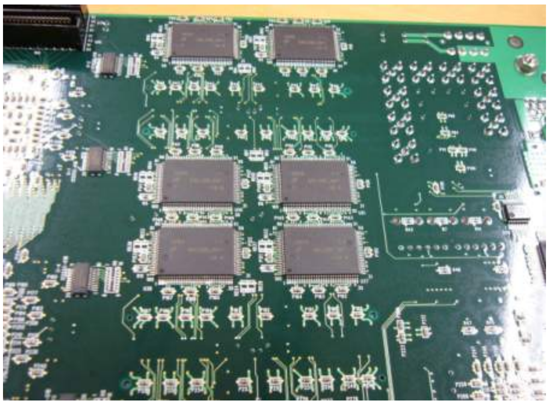
| | | | | | |
|---|---|---|---|---|---|
| **Speed (%cycles):** | ½'s | 1's | 10's | 100's | 1,000's |
| **Size (bytes):** | 100's | K's | 10K's | M's | G's to T's |
| **Cost:** | highest | | | | lowest |

## DRAM (dynamic random access memory)

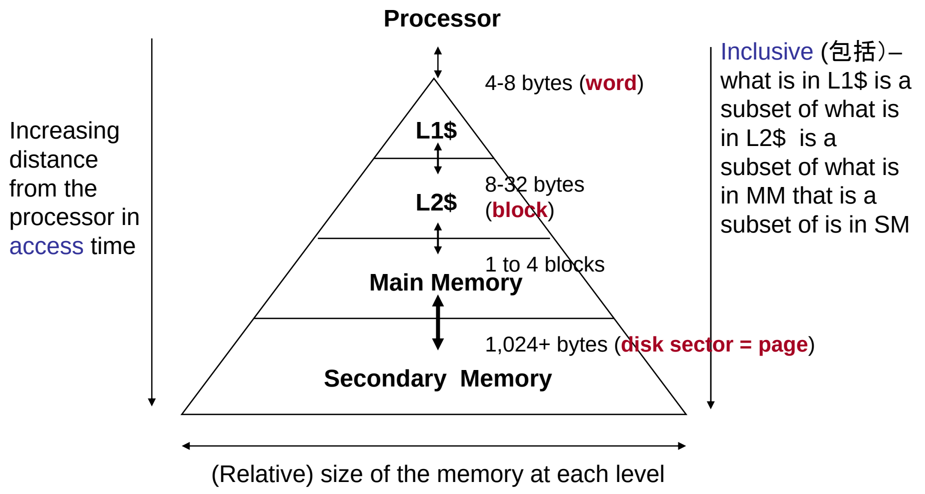


## SRAM (static random access memory)





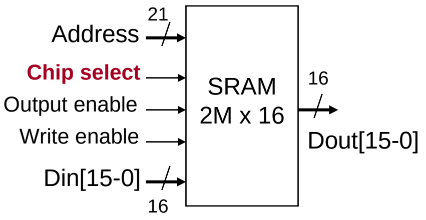
## Characteristics of the Memory Hierarchy


Processor
4-8 bytes (word)
L1$
8-32 bytes (block)
L2$
1 to 4 blocks
Main Memory
1,024+ bytes (disk sector = page)
Secondary Memory
Increasing distance from the processor in access time
(Relative) size of the memory at each level

Inclusive (包括)– what is in L1$ is a subset of what is in L2$ is a subset of what is in MM that is a subset of is in SM

## Memory Hierarchy Technologies

- Caches use SRAM for speed and technology compatibility
  - **Low density** (**6 transistor cells**), high power, expensive, fast
  - **Static**: content will last "forever" (until power turned off)

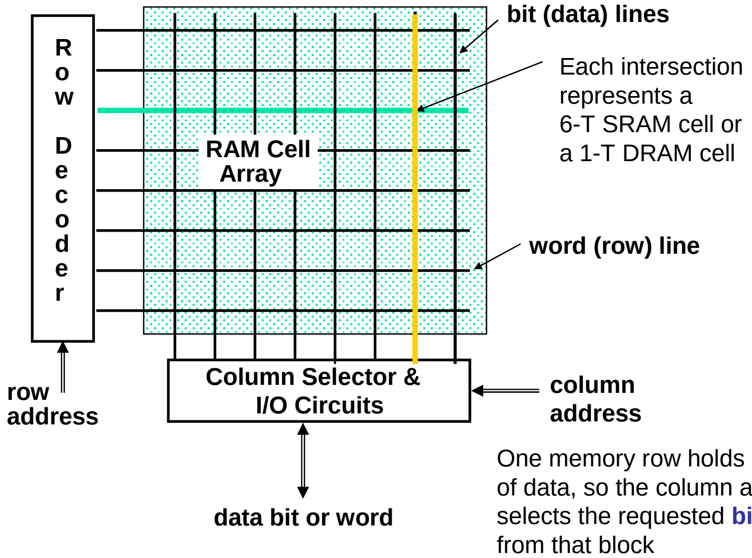


- Main Memory uses DRAM for size (density)
  - **High density** (**1 transistor cells**), low power, cheap, slow
  - **Dynamic**: needs to be "refreshed" regularly (~ every 8 ms)
    - 1% to 2% of the active cycles of the DRAM
  - Addresses divided into 2 halves (row and column)
    - **RAS** or Row Access Strobe triggering row decoder
    - **CAS** or Column Access Strobe triggering column selector

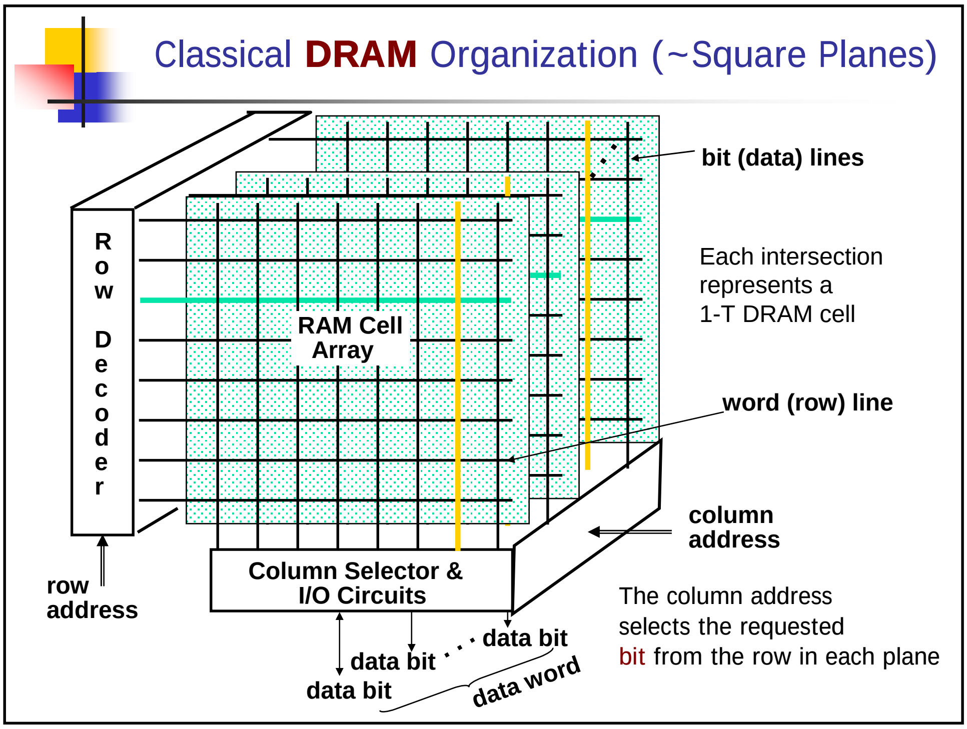
## Memory Performance **Metrics**

- **Latency**(レイテンシ，応答時間)：
  Time to access one word
  - **Cycle time**: time between requests
  - **Access time**: time between the request and when the data is available (or written)
  - Usually **cycle time** > **access time**
- **Bandwidth**(バンド幅，スループット)：
  How much data from the memory can be supplied to the processor per unit time
  - width of the data channel * the rate at which it can be used

## Classical RAM Organization (~Square)



One memory row holds a block of data, so the column address selects the requested **bit** or **word** from that block



## Classical **DRAM** Organization (~Square Planes)

bit (data) lines

Each intersection represents a 1-T DRAM cell

word (row) line

Row Decoder

RAM Cell Array

Column Selector & I/O Circuits

row address

column address

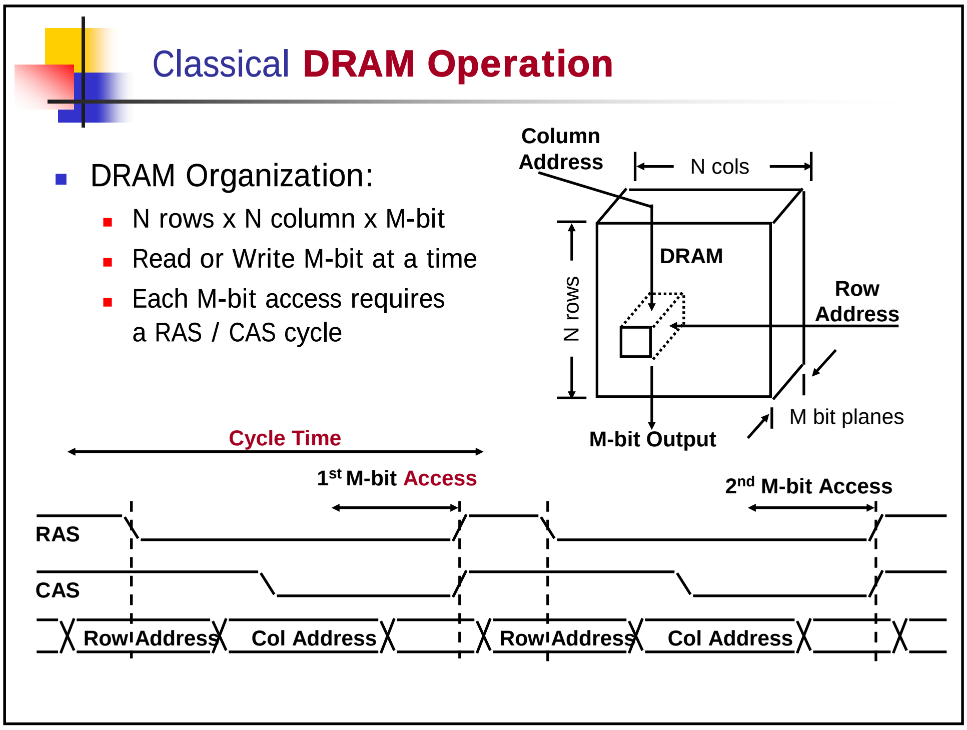
data bit

data bit

data bit

data word

The column address selects the requested bit from the row in each plane



## Classical **DRAM Operation**

- DRAM Organization:
  - N rows x N column x M-bit
  - Read or Write M-bit at a time
  - Each M-bit access requires a RAS / CAS cycle

Column Address

N cols

N rows

DRAM

Row Address

M bit planes

M-bit Output

Cycle Time

1st M-bit Access

2nd M-bit Access

RAS

CAS

Row Address

Col Address

Row Address

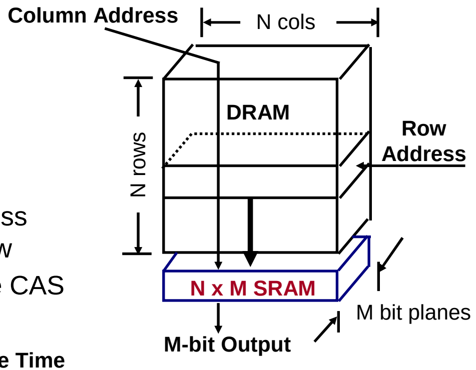
Col Address

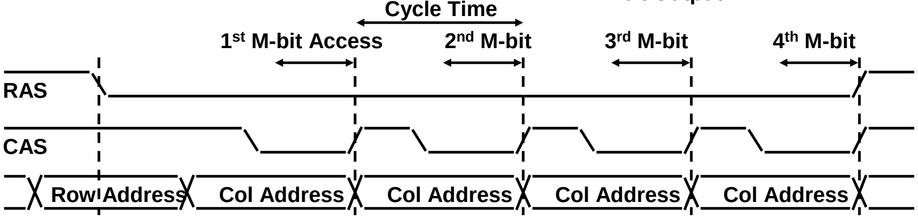
## **Page Mode** DRAM Operation

- Page Mode DRAM
  - N x M SRAM to save a row

❑ After a row is read into the SRAM “register”

- Only CAS is needed to access other M-bit words on that row
- RAS remains asserted while CAS is toggled





2009-06-11

2009年 前学期 TOKYO TECH

## 計算機アーキテクチャ 第一 (E)

### 6. メモリ2:
### 半導体メモリシステム, ファイルメモリシステム

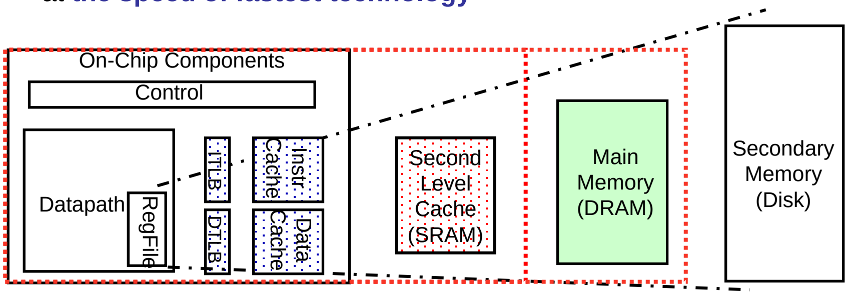
吉瀬 謙二 計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室 木曜日13:20 ー 14:50

## The Memory Hierarchy **Goal**

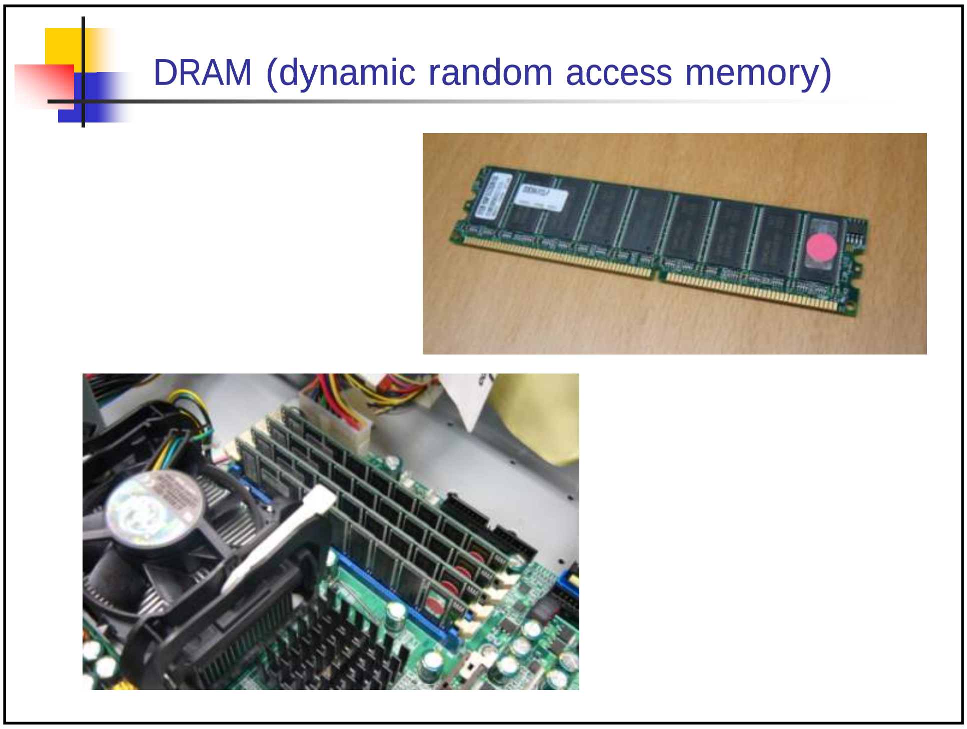
- **Fact**:
  **Large memories are slow** and
  **fast memories are small**
- How do we create a memory that gives the illusion of being large, cheap and fast ?
  - With **hierarchy**（階層）
  - With **parallelism**（並列性）

## A Typical Memory Hierarchy

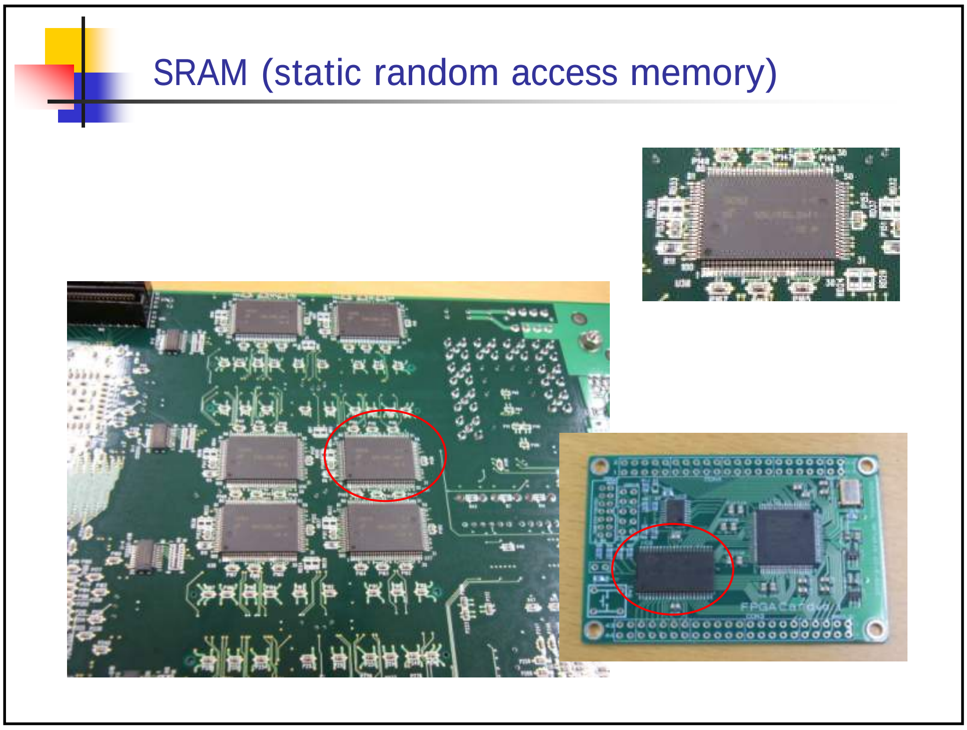
- By taking advantage of **the principle of locality**（局所性）
  - Present **much memory** in **the cheapest technology**
  - at **the speed of fastest technology**



| | | | | | |
|---|---|---|---|---|---|
| **Speed (%cycles):** | ½'s | 1's | 10's | 100's | 1,000's |
| **Size (bytes):** | 100's | K's | 10K's | M's | G's to T's |
| **Cost:** | highest | | | | lowest |

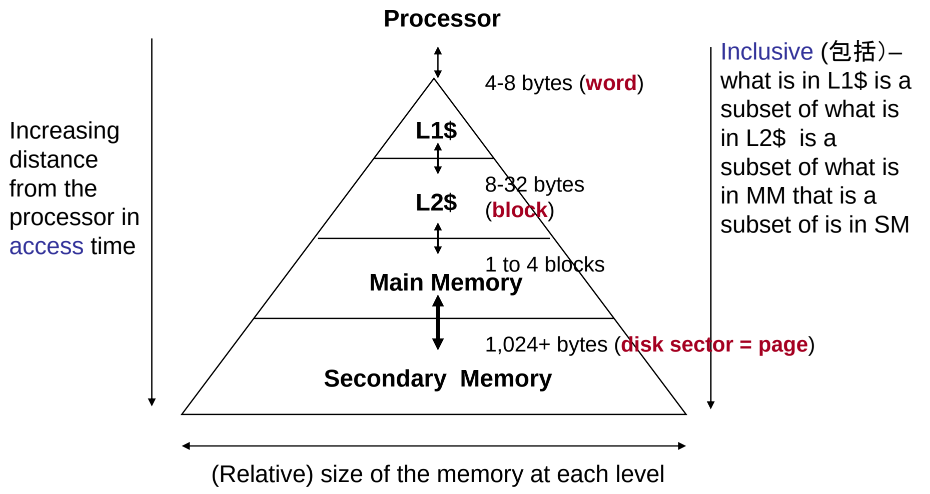


## DRAM (dynamic random access memory)



## SRAM (static random access memory)

## Characteristics of the Memory Hierarchy



Processor

4-8 bytes (**word**)

L1$

8-32 bytes (**block**)

L2$

1 to 4 blocks

Main Memory

1,024+ bytes (**disk sector = page**)

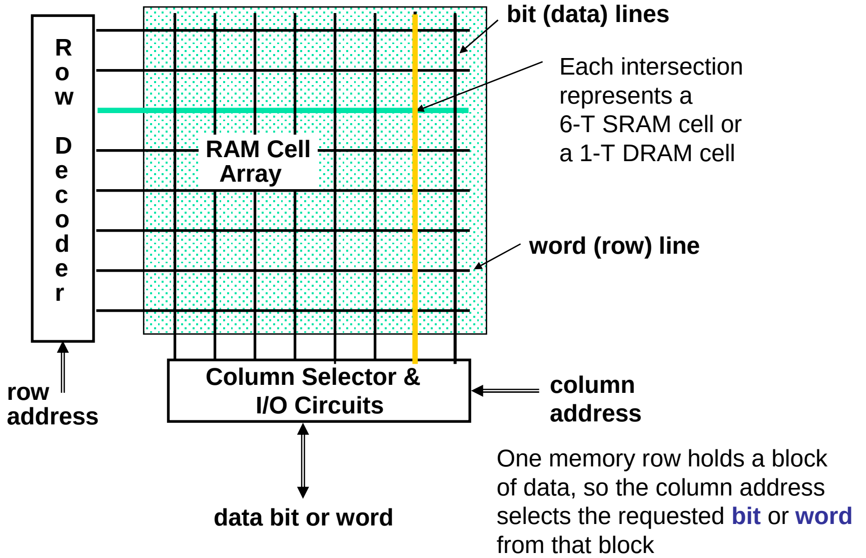
Secondary Memory

Increasing distance from the processor in access time

Inclusive (包括)– what is in L1$ is a subset of what is in L2$ is a subset of what is in MM that is a subset of is in SM

(Relative) size of the memory at each level

## Classical RAM Organization (~Square)



bit (data) lines

Each intersection represents a 6-T SRAM cell or a 1-T DRAM cell

R o w D e c o d e r

RAM Cell Array

word (row) line

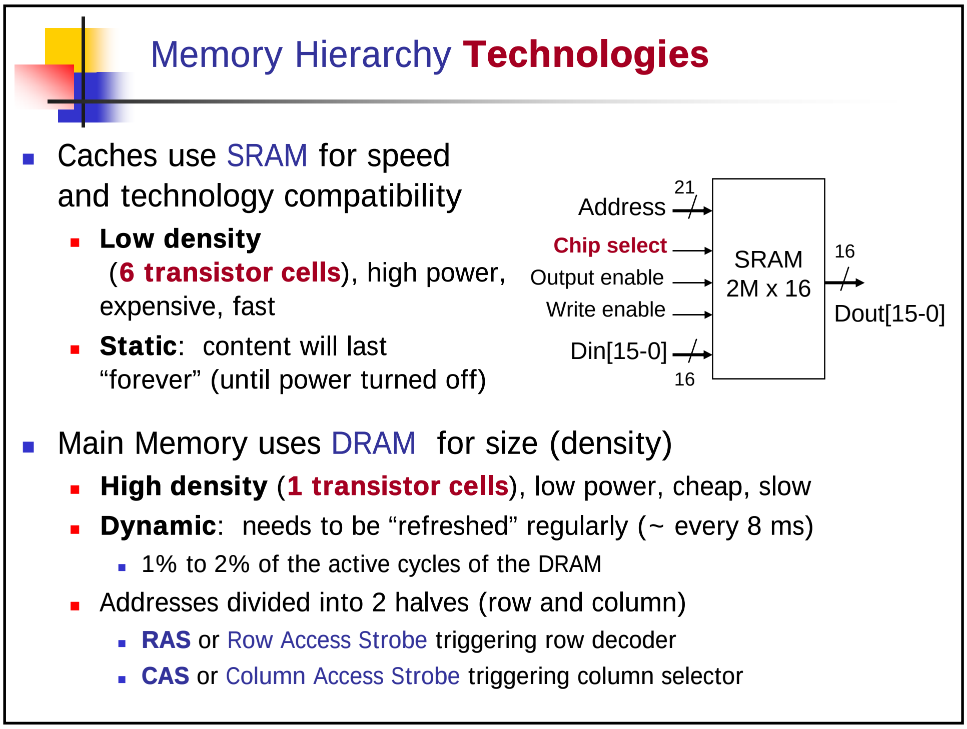
row address

Column Selector & I/O Circuits

column address

data bit or word

One memory row holds a block of data, so the column address selects the requested **bit** or **word** from that block

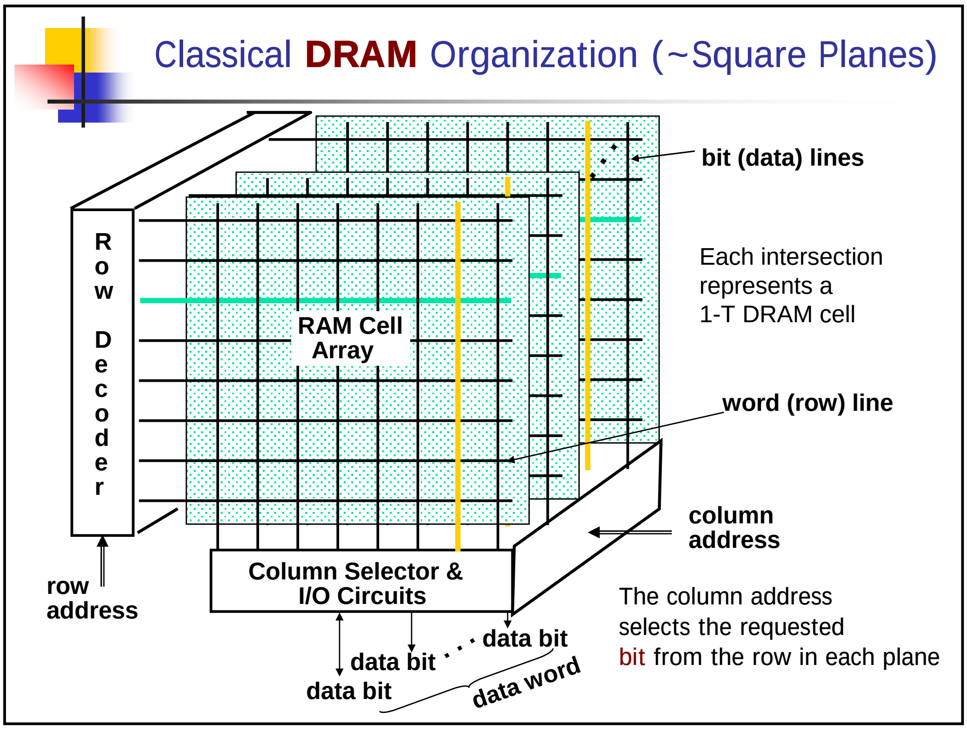


## Memory Hierarchy **Technologies**

- Caches use SRAM for speed and technology compatibility
  - **Low density** (**6 transistor cells**), high power, expensive, fast
  - **Static**: content will last "forever" (until power turned off)

Address
21
Chip select
Output enable
Write enable
Din[15-0]
16
SRAM
2M x 16
16
Dout[15-0]

- Main Memory uses DRAM for size (density)
  - **High density** (**1 transistor cells**), low power, cheap, slow
  - **Dynamic**: needs to be "refreshed" regularly (~ every 8 ms)
    - 1% to 2% of the active cycles of the DRAM
  - Addresses divided into 2 halves (row and column)
    - **RAS** or Row Access Strobe triggering row decoder
    - **CAS** or Column Access Strobe triggering column selector



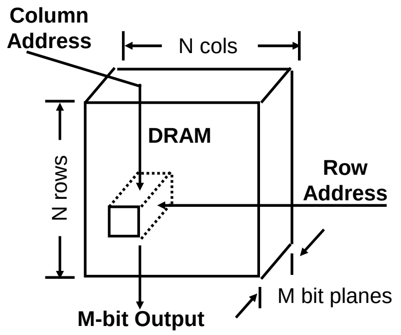
## Classical **DRAM** Organization (~Square Planes)

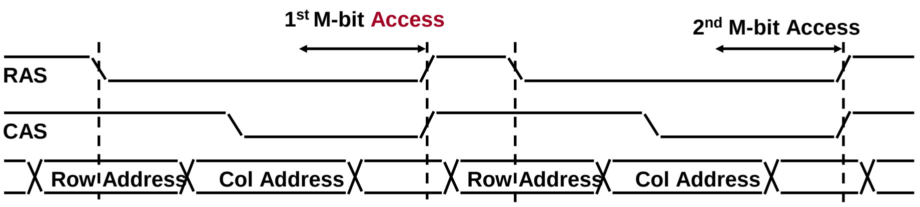
Each intersection represents a 1-T DRAM cell

The column address selects the requested bit from the row in each plane

## Classical **DRAM Operation**

- DRAM Organization:
  - N rows x N column x M-bit
  - Read or Write M-bit at a time
  - Each M-bit access requires a RAS / CAS cycle



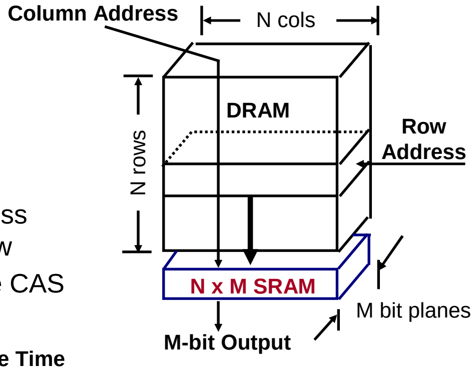


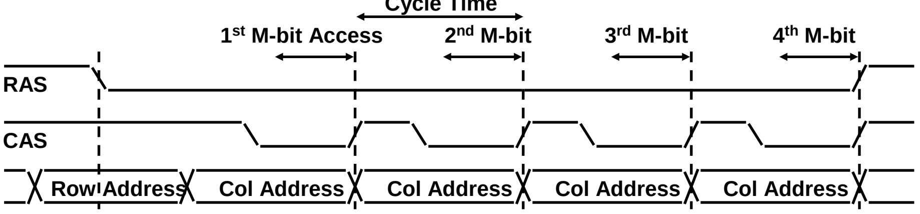
## **Page Mode** DRAM Operation

- Page Mode DRAM
  - N x M SRAM to save a row

❑ After a row is read into the SRAM “register”

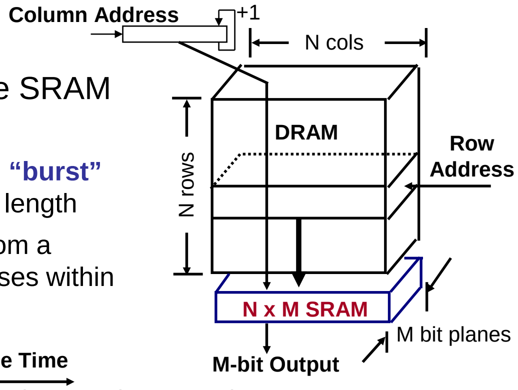
- Only CAS is needed to access other M-bit words on that row
- RAS remains asserted while CAS is toggled

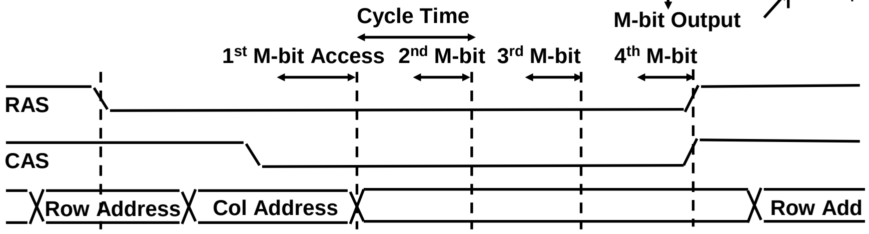




## Synchronous DRAM (SDRAM) Operation

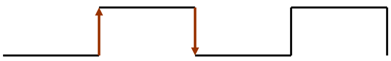
- After a **row** is read into the SRAM register
  - Inputs CAS as **the starting "burst" address** along with a burst length
  - Transfers a burst of data from a series of sequential addresses within that row





## Other DRAM Architectures

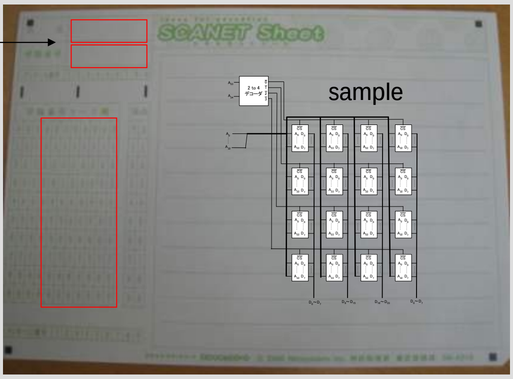
- Double Data Rate SDRAMs – **DDR-SDRAMs** (and DDR-SRAMs)
  - Double data rate because they transfer data on both the rising and falling edge of the clock
  - Are the most widely used form of SDRAMs
- **DDR2-SDRAMs**
- **DDR3-SDRAMs**



## 演習

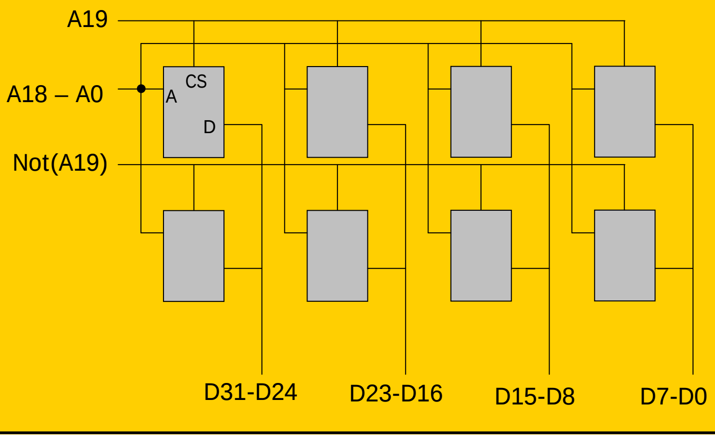
- 512K x 8ビット（512KB) のSRAMを用いて、32ビットデータ幅の4MB のメモリを実現したい.
- 8個のメモリチップ，チップ選択信号CS， データ信号，アドレス信号の接続を示せ.

氏名，学籍番号，
学籍番号マーク欄(右詰で)



## 演習

- 512K x 8ビット（512KB) のSRAMを用いて、32ビットデータ幅の4MB のメモリを実現したい.
- 8個のメモリチップ，チップ選択信号CS， データ信号(D)，アドレス信号(A)の接続を示せ.



## DRAM Memory Latency & Bandwidth Milestones

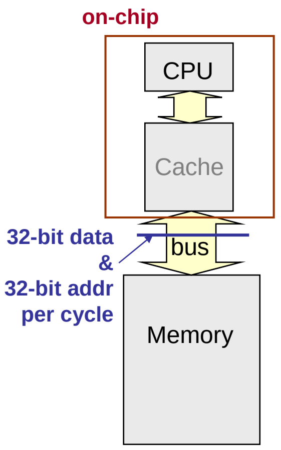
| | DRAM | Page DRAM | FastPage DRAM | FastPage DRAM | Synch DRAM | DDR SDRAM |
|---|---|---|---|---|---|---|
| Module Width | 16b | 16b | 32b | 64b | 64b | 64b |
| Year | 1980 | 1983 | 1986 | 1993 | 1997 | 2000 |
| Mb/chip | 0.06 | 0.25 | 1 | 16 | 64 | 256 |
| Die size (mm$^2$) | 35 | 45 | 70 | 130 | 170 | 204 |
| Pins/chip | 16 | 16 | 18 | 20 | 54 | 66 |
| **BWidth (MB/s)** | **13** | **40** | **160** | **267** | **640** | **1600** |
| **Latency (nsec)** | **225** | **170** | **125** | **75** | **62** | **52** |

Patterson, CACM Vol 47, #10, 2004

- In the time that the memory to processor bandwidth doubles the memory latency improves by a factor of only 1.2 to 1.4
- To deliver such high bandwidth, the internal DRAM has to be organized as **interleaved memory banks**

## Memory Systems that Support Caches

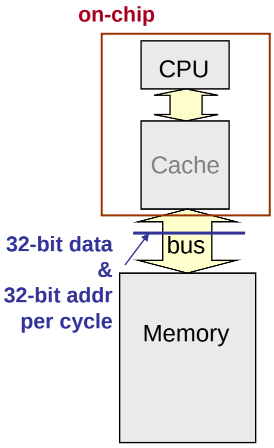
- The off-chip interconnect and memory architecture can affect overall system performance **in dramatic ways**



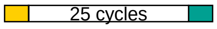
One word wide organization (one word wide bus and one word wide memory)

- **Assume** （前提）
  1. **1** clock cycle to send the address
  2. **25** clock cycles for DRAM **cycle time**, **8** clock cycles **access time**
  3. **1** clock cycle to return a word of data
- Memory-Bus to Cache **bandwidth**
  - **number of bytes transferred from memory to cache per clock cycle**

## One Word Wide Memory Organization

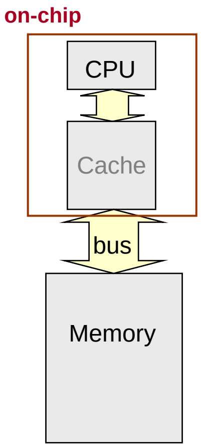


- The pipeline stalls the number of cycles for **one word** (32bit) from memory
  - **1** cycle to send address
  - **25** cycles to read DRAM
  - **1** cycle to return data
  - **27 total clock cycles** miss penalty

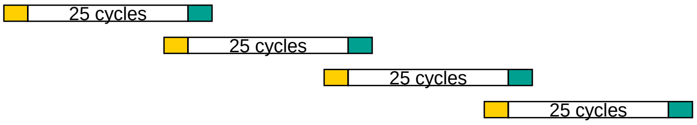


- Number of bytes transferred per clock cycle (**bandwidth**) for a single miss
  - **4 / 27 = 0.148** bytes per clock

## One Word Wide Memory Organization, con't

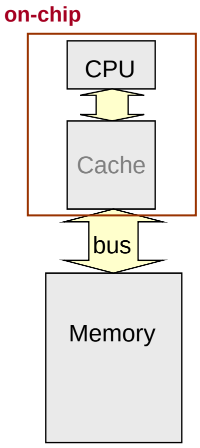


- What if the block size is **four words**?
  - **1** cycle to send 1st address
  - **4 * 25 = 100** cycles to read DRAM
  - **1** cycle to return last data word
  - **102 total clock cycles** miss penalty

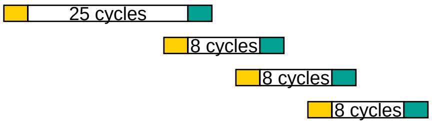


- Number of bytes transferred per clock cycle (**bandwidth**) for a single miss
  - **(4 x 4) / 102 = 0.157** bytes per clock

## One Word Wide Memory Organization, con’t

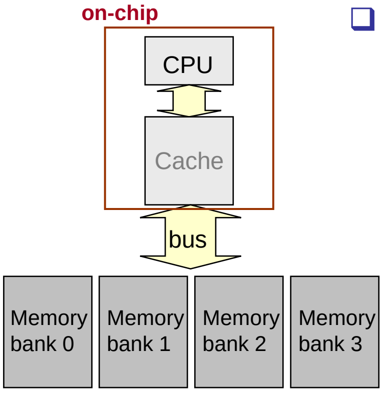


- What if the block size is **four words** and if a **fast page mode DRAM** is used?
  - **1** cycle to send 1st address
  - **25 + (3 * 8) = 49** cycles to read DRAM
  - **1** cycle to return last data word
  - **51 total clock cycles** miss penalty

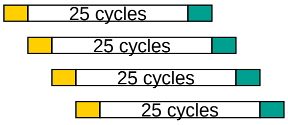


- Number of bytes transferred per clock cycle (**bandwidth**) for a single miss
  - **(4 x 4) / 51 = 0.314** bytes per clock

## Interleaved（インターリーブ）Memory Organization



❑ For a block size of **four words with interleaved memory (4 banks)**
- **1** cycle to send 1st address
- **25 + 3 = 28** cycles to read DRAM
- **1** cycle to return last data word
- **30 total clock cycles** miss penalty



- Number of bytes transferred per clock cycle (**bandwidth**) for a single miss
  - **(4 x 4) / 30 = 0.533** bytes per clock

2009-06-18

2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## 7. メモリ3：<br>キャッシュシステム

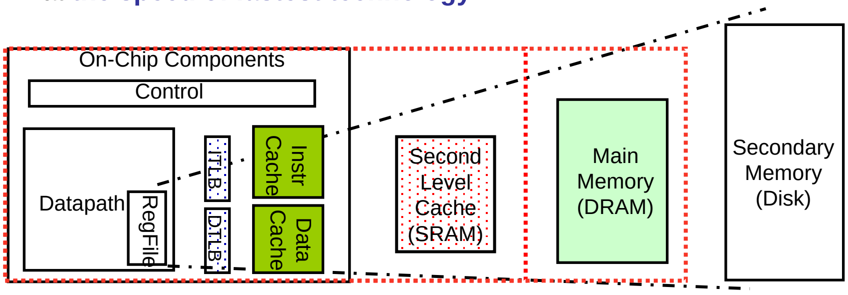
吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

## The Memory Hierarchy **Goal**

- **Fact**:
  **Large memories are slow** and
  **fast memories are small**
- How do we create a memory that gives the illusion of being large, cheap and fast ?
  - With **hierarchy**（階層）
  - With **parallelism**（並列性）

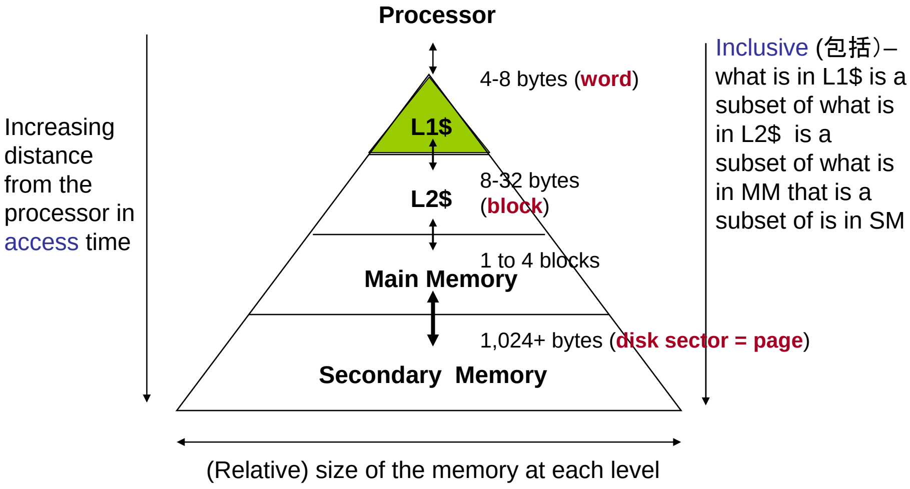
## A Typical Memory Hierarchy

- By taking advantage of **the principle of locality**（局所性）
  - Present **much memory** in **the cheapest technology**
  - at **the speed of fastest technology**



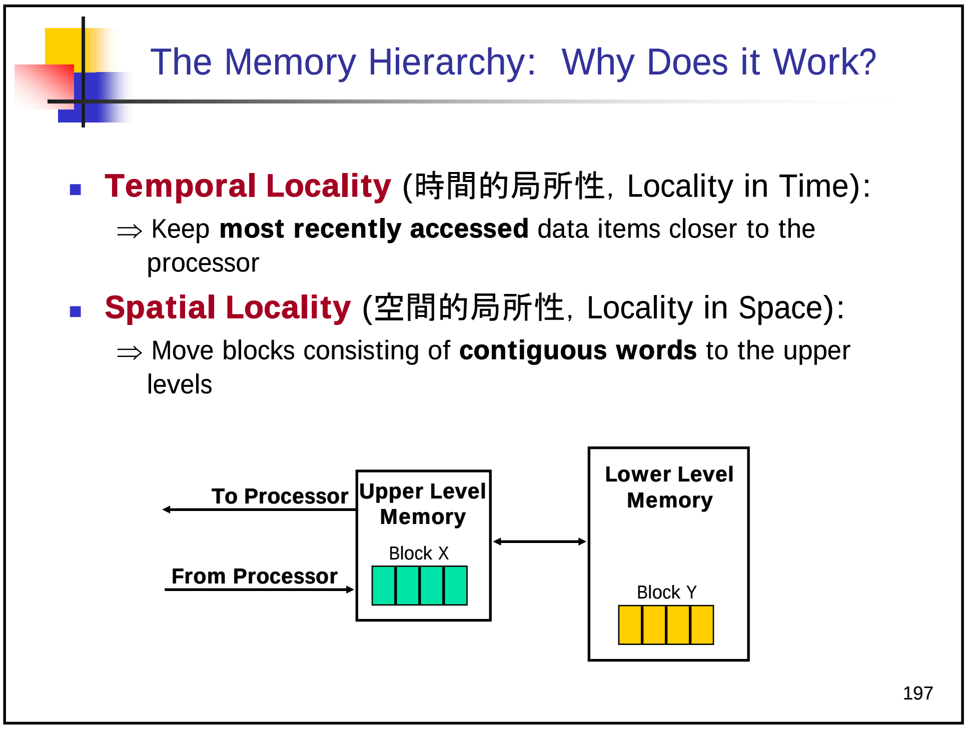
| **Speed (%cycles):** | ½'s | 1's | 10's | 100's | 1,000's |
|---|---|---|---|---|---|
| **Size (bytes):** | 100's | K's | 10K's | M's | G's to T's |
| **Cost:** | highest | | | | lowest |

## Characteristics of the Memory Hierarchy



Increasing distance from the processor in access time

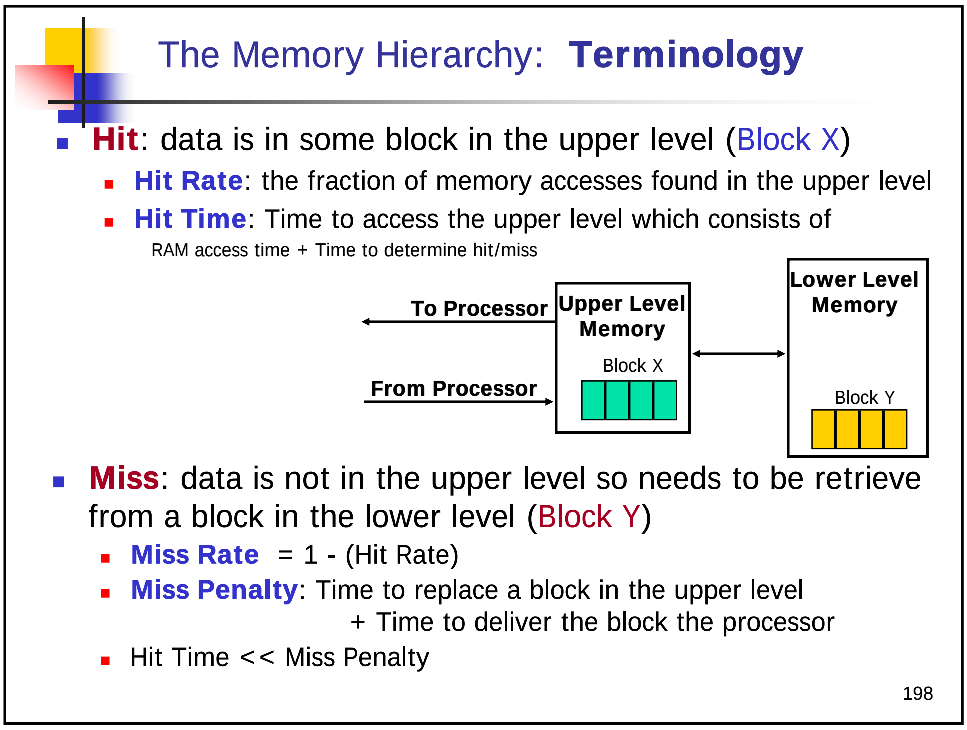
Inclusive (包括)– what is in L1$ is a subset of what is in L2$ is a subset of what is in MM that is a subset of is in SM



## The Memory Hierarchy: Why Does it Work?

- **Temporal Locality** (時間的局所性, Locality in Time):
  ⇒ Keep **most recently accessed** data items closer to the processor
- **Spatial Locality** (空間的局所性, Locality in Space):
  ⇒ Move blocks consisting of **contiguous words** to the upper levels

To Processor
From Processor
Upper Level Memory
Block X
Lower Level Memory
Block Y



## The Memory Hierarchy: **Terminology**

- **Hit**: data is in some block in the upper level (Block X)
  - **Hit Rate**: the fraction of memory accesses found in the upper level
  - **Hit Time**: Time to access the upper level which consists of
    RAM access time + Time to determine hit/miss

To Processor
From Processor
Upper Level Memory
Block X
Lower Level Memory
Block Y

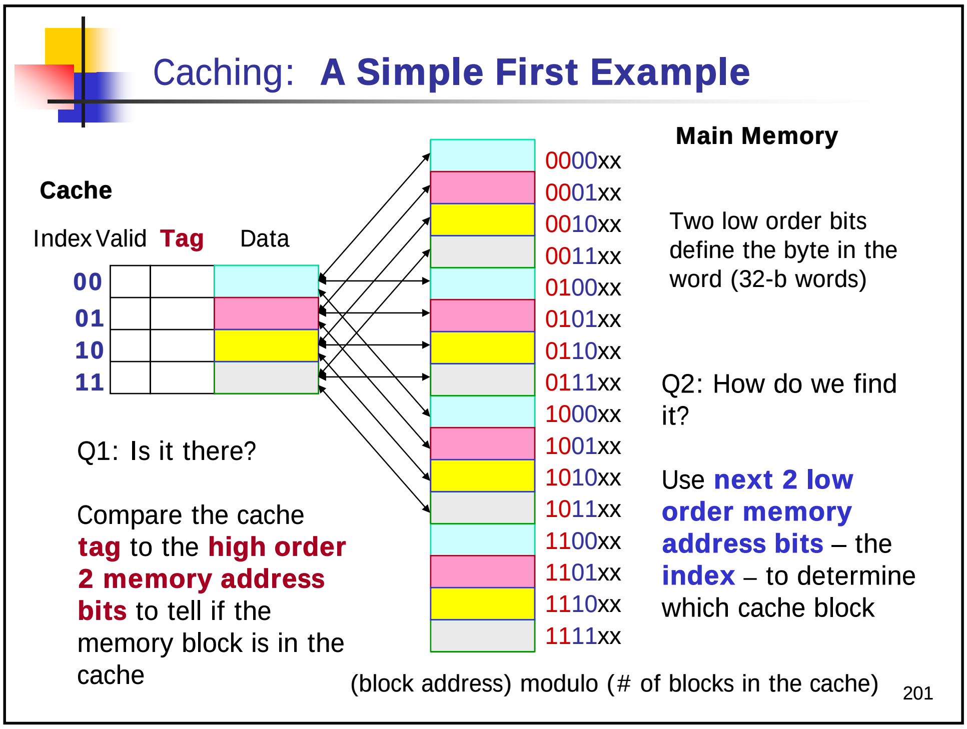
- **Miss**: data is not in the upper level so needs to be retrieve from a block in the lower level (Block Y)
  - **Miss Rate** = 1 - (Hit Rate)
  - **Miss Penalty**: Time to replace a block in the upper level
    + Time to deliver the block the processor
  - Hit Time << Miss Penalty

## How is the Hierarchy Managed?

- registers ↔ memory
  - by compiler (programmer?)
- cache ↔ main memory
  - by the cache controller hardware
- main memory ↔ disks
  - by the operating system (**virtual memory**)
    - virtual to physical address mapping assisted by the hardware (TLB, Translation Look-aside Buffer)
  - by the programmer (**files**)

## Cache

- Two questions to answer (in hardware):
  - Q1: **How do we know if a data item is in the cache?**
  - Q2: **If it is, how do we find it?**
- **Direct mapped**
  - For each item of data at the lower level, there is exactly one location in the cache where it might be - so lots of items at the lower level must **share** locations in the upper level
  - Address mapping:
    **(block address) modulo (# of blocks in the cache)**
  - First, consider block sizes of **one word**



## Caching: A Simple First Example

**Cache**

Index Valid **Tag** Data

| Index | Valid | Tag | Data |
|---|---|---|---|
| 00 | | | |
| 01 | | | |
| 10 | | | |
| 11 | | | |

**Main Memory**

0000xx<br>0001xx<br>0010xx<br>0011xx<br>0100xx<br>0101xx<br>0110xx<br>0111xx<br>1000xx<br>1001xx<br>1010xx<br>1011xx<br>1100xx<br>1101xx<br>1110xx<br>1111xx

Two low order bits define the byte in the word (32-b words)

Q2: How do we find it?

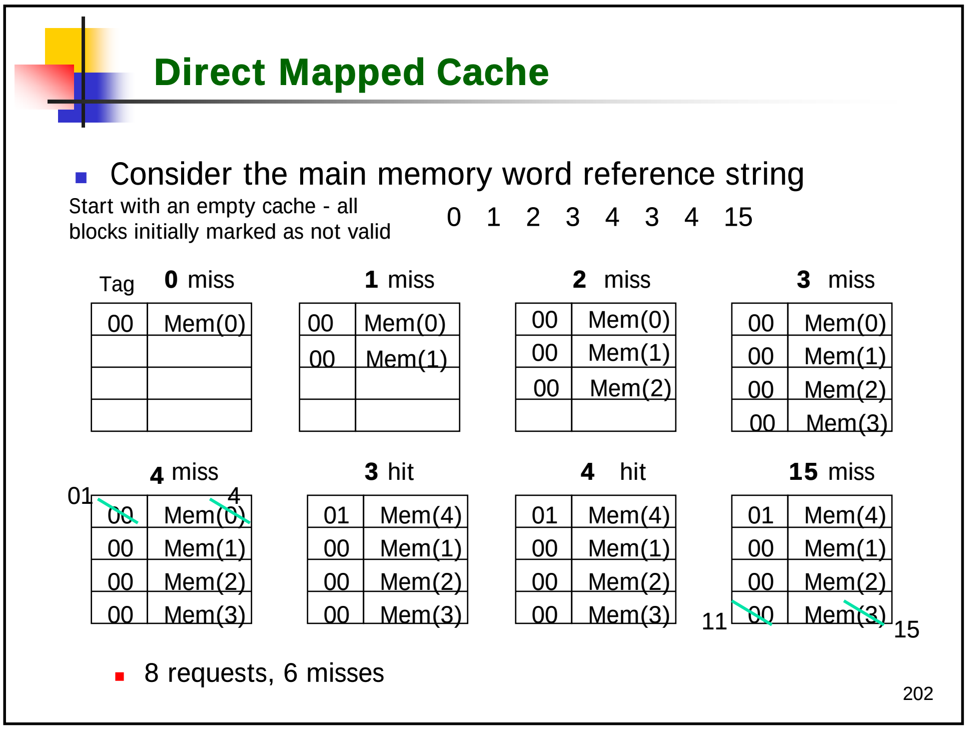
Use **next 2 low order memory address bits** – the **index** – to determine which cache block

Q1: Is it there?

Compare the cache **tag** to the **high order 2 memory address bits** to tell if the memory block is in the cache

(block address) modulo (# of blocks in the cache)

201



## Direct Mapped Cache

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 1 2 3 4 3 4 15

Tag **0** miss

| | |
|---|---|
| 00 | Mem(0) |
| | |
| | |
| | |

**1** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| | |
| | |

**2** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| | |

**3** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**4** miss

| | |
|---|---|
| 01 ~~00~~ | ~~Mem(0)~~ 4 |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**3** hit

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**4** hit

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**15** miss

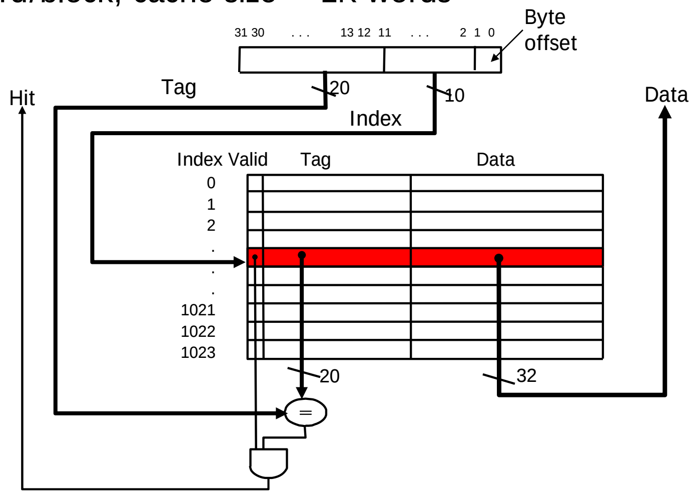
| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 11 ~~00~~ | ~~Mem(3)~~ 15 |

- 8 requests, 6 misses

202

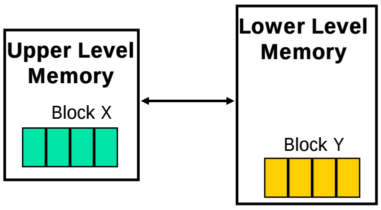
## MIPS Direct Mapped Cache Example

- One word/block, cache size = 1K words



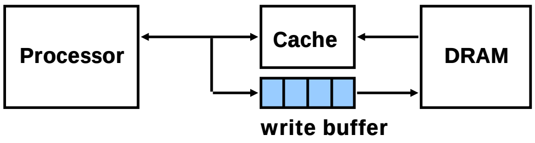
*What kind of locality are we taking advantage of?*

## Handling Cache **Hits**



- **Read hits (I$ and D$)**
  - this is what we want!
- **Write hits (D$ only)**
  - allow cache and memory to be **inconsistent**
    - write the data only into the cache block (**write-back**)
    - need a **dirty** bit for each data cache block to tell if it needs to be written back to memory when it is evicted
  - require the cache and memory to be **consistent**
    - always write the data into both the cache block and the next level in the memory hierarchy (**write-through**) so don't need a dirty bit
    - writes run at the speed of the next level in the memory hierarchy – **so slow**! – or can use a **write buffer**, so only have to stall if the write buffer is full

## Write Buffer for Write-Through Caching



- **Write buffer** between the cache and main memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller:  writes contents of the write buffer to memory
- The write buffer is just a **FIFO**
  - Typical number of entries: 4
  - Works fine if store frequency is low
- Memory system designer's nightmare, Write buffer **saturation**（飽和）
  - One solution is to use a write-back cache; another is to use an L2 cache

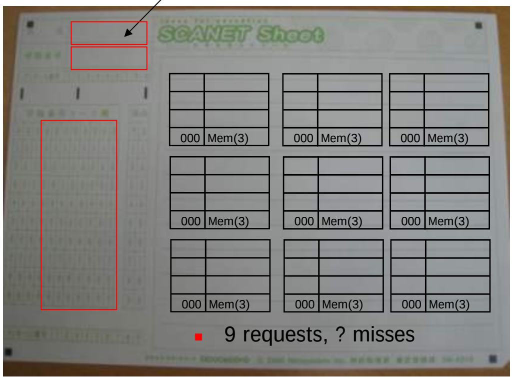
## Exercise

- Consider the main memory word reference string
  - 3, 2, 18, 3, 16, 2, 3, 18, 3

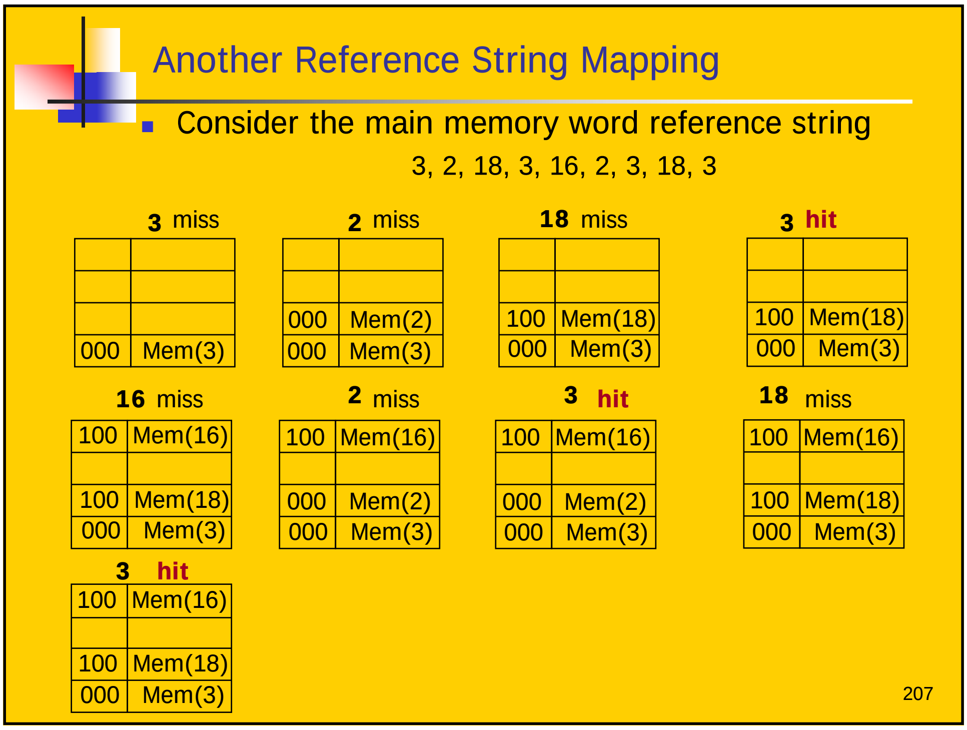
| Tag | **3** miss |
|---|---|
| | |
| | |
| | |
| 000 | Mem(3) |

- 9 requests, ? misses

氏名，学籍番号，
学籍番号マーク欄(右詰で)



- 9 requests, ? misses



## Another Reference String Mapping

- Consider the main memory word reference string
  3, 2, 18, 3, 16, 2, 3, 18, 3

**3** miss

| | |
|---|---|
| | |
| | |
| | |
| 000 | Mem(3) |

**2** miss

| | |
|---|---|
| | |
| | |
| 000 | Mem(2) |
| 000 | Mem(3) |

**18** miss

| | |
|---|---|
| | |
| | |
| 100 | Mem(18) |
| 000 | Mem(3) |

**3** hit

| | |
|---|---|
| | |
| | |
| 100 | Mem(18) |
| 000 | Mem(3) |

**16** miss

| | |
|---|---|
| 100 | Mem(16) |
| | |
| 100 | Mem(18) |
| 000 | Mem(3) |

**2** miss

| | |
|---|---|
| 100 | Mem(16) |
| | |
| 000 | Mem(2) |
| 000 | Mem(3) |

**3** hit

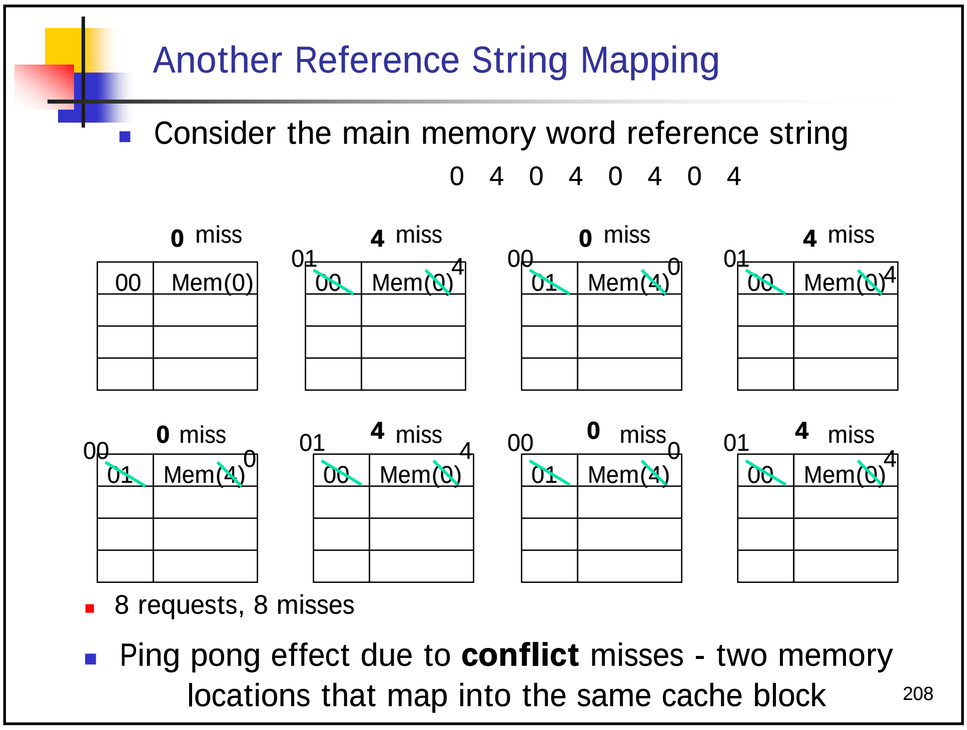
| | |
|---|---|
| 100 | Mem(16) |
| | |
| 000 | Mem(2) |
| 000 | Mem(3) |

**18** miss

| | |
|---|---|
| 100 | Mem(16) |
| | |
| 100 | Mem(18) |
| 000 | Mem(3) |

**3** hit

| | |
|---|---|
| 100 | Mem(16) |
| | |
| 100 | Mem(18) |
| 000 | Mem(3) |



## Another Reference String Mapping

- Consider the main memory word reference string
  0 4 0 4 0 4 0 4

**0** miss

| | |
|---|---|
| 00 | Mem(0) |
| | |
| | |
| | |

**4** miss

| | |
|---|---|
| 01 ~~00~~ | Mem(~~0~~) 4 |
| | |
| | |
| | |

**0** miss

| | |
|---|---|
| 00 ~~01~~ | Mem(~~4~~) 0 |
| | |
| | |
| | |

**4** miss

| | |
|---|---|
| 01 ~~00~~ | Mem(~~0~~) 4 |
| | |
| | |
| | |

**0** miss

| | |
|---|---|
| 00 ~~01~~ | Mem(~~4~~) 0 |
| | |
| | |
| | |

**4** miss

| | |
|---|---|
| 01 ~~00~~ | Mem(~~0~~) 4 |
| | |
| | |
| | |

**0** miss

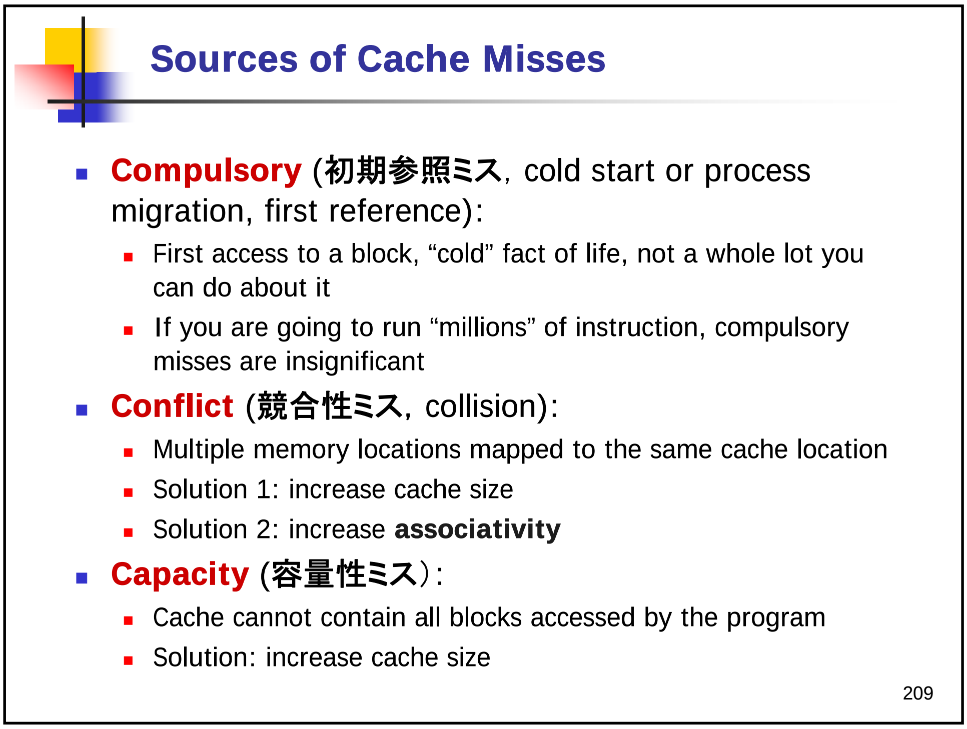
| | |
|---|---|
| 00 ~~01~~ | Mem(~~4~~) 0 |
| | |
| | |
| | |

**4** miss

| | |
|---|---|
| 01 ~~00~~ | Mem(~~0~~) 4 |
| | |
| | |
| | |

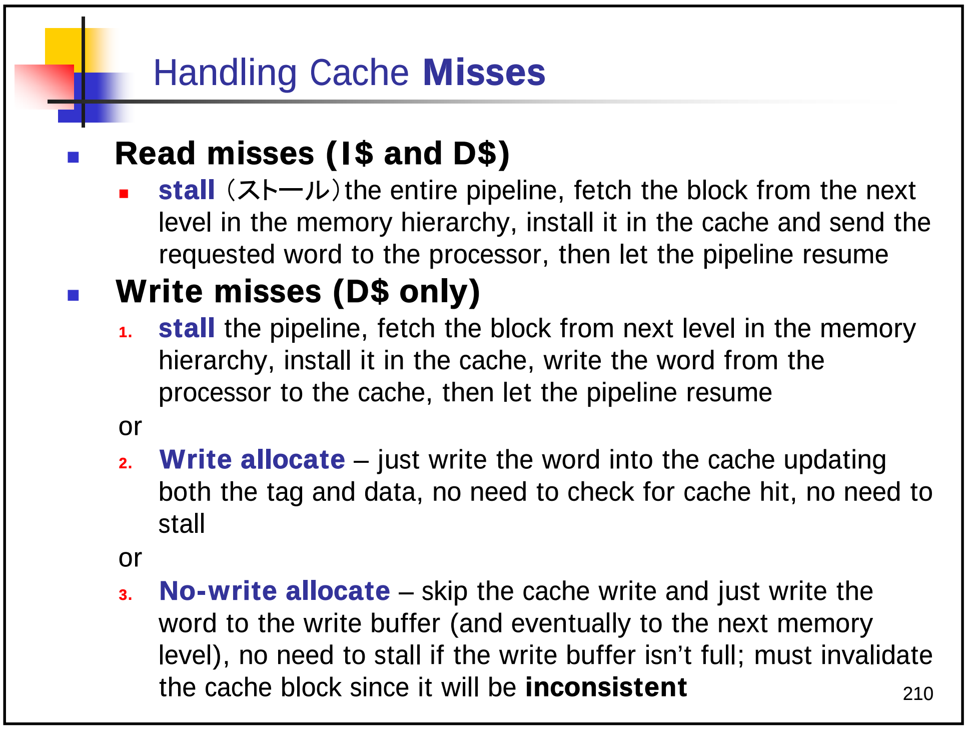
- 8 requests, 8 misses

- Ping pong effect due to **conflict** misses - two memory locations that map into the same cache block



# Sources of Cache Misses

- **Compulsory** (初期参照ミス, cold start or process migration, first reference):
  - First access to a block, “cold” fact of life, not a whole lot you can do about it
  - If you are going to run “millions” of instruction, compulsory misses are insignificant
- **Conflict** (競合性ミス, collision):
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase **associativity**
- **Capacity** (容量性ミス):
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size



# Handling Cache **Misses**

- **Read misses (I$ and D$)**
  - **stall** （ストール）the entire pipeline, fetch the block from the next level in the memory hierarchy, install it in the cache and send the requested word to the processor, then let the pipeline resume
- **Write misses (D$ only)**
  1. **stall** the pipeline, fetch the block from next level in the memory hierarchy, install it in the cache, write the word from the processor to the cache, then let the pipeline resume

  or

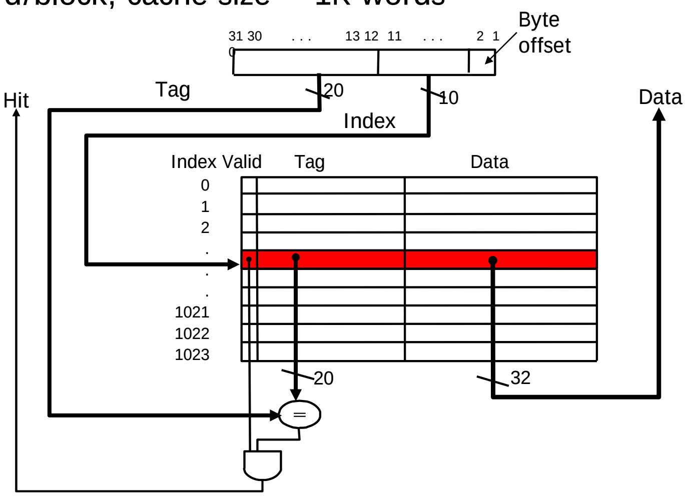
  2. **Write allocate** – just write the word into the cache updating both the tag and data, no need to check for cache hit, no need to stall

  or

  3. **No-write allocate** – skip the cache write and just write the word to the write buffer (and eventually to the next memory level), no need to stall if the write buffer isn’t full; must invalidate the cache block since it will be **inconsistent**

## MIPS Direct Mapped Cache Example

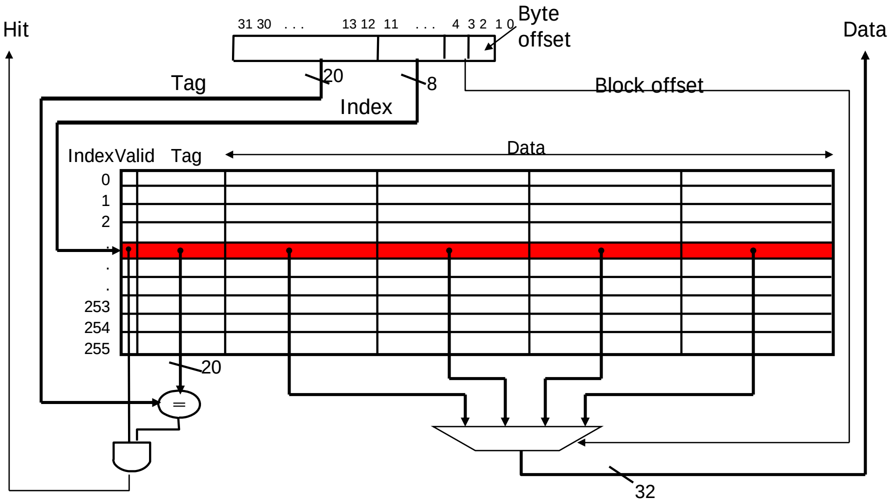
- One word/block, cache size = 1K words



*What kind of locality are we taking advantage of?*

## **Multiword** Block Direct Mapped Cache

- Four words/block, cache size = 1K words



*What kind of locality are we taking advantage of?*

# **Direct Mapped Cache** again!

- Consider the main memory word reference string

0 1 2 3 4 3 4 15

**0** miss

| | |
|---|---|
| 00 | Mem(0) |
| | |
| | |
| | |

**1** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| | |
| | |

**2** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| | |

**3** miss

| | |
|---|---|
| 00 | Mem(0) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**4** miss

| | |
|---|---|
| 01 ~~00~~ | ~~Mem(0)~~ 4 |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**3** hit

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**4** hit

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 00 | Mem(3) |

**15** miss

| | |
|---|---|
| 01 | Mem(4) |
| 00 | Mem(1) |
| 00 | Mem(2) |
| 11 ~~00~~ | ~~Mem(3)~~ 15 |

- 8 requests, 6 misses

# Taking Advantage of **Spatial Locality**

- Let cache block hold more than one word

0 1 2 3 4 3 4 15

**0** miss

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| | | |

**1** hit

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| | | |

**2** miss

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

**3** hit

| | | |
|---|---|---|
| 00 | Mem(1) | Mem(0) |
| 00 | Mem(3) | Mem(2) |

**4** miss

| | | |
|---|---|---|
| 01 ~~00~~ | ~~Mem(1)~~ 5 | ~~Mem(0)~~ 4 |
| 00 | Mem(3) | Mem(2) |

**3**hit

| | | |
|---|---|---|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

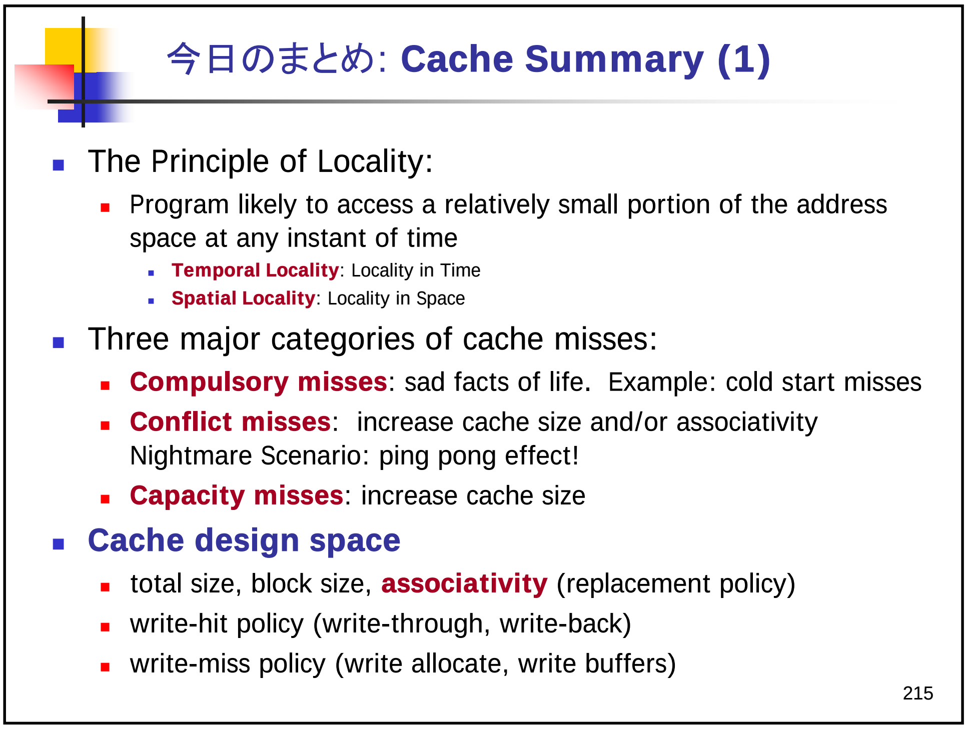
**4** hit

| | | |
|---|---|---|
| 01 | Mem(5) | Mem(4) |
| 00 | Mem(3) | Mem(2) |

**15**miss

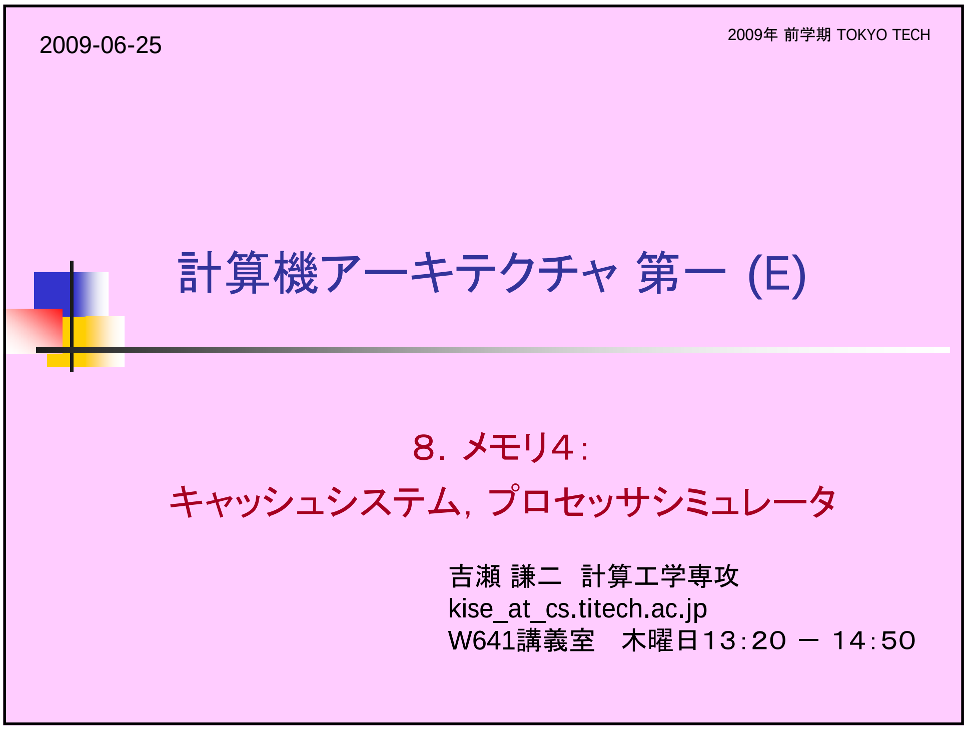
| | | |
|---|---|---|
| 01 | Mem(5) | Mem(4) |
| 11 ~~00~~ | ~~Mem(3)~~ 15 | ~~Mem(2)~~ 14 |

- 8 requests, 4 misses



## 今日のまとめ: **Cache Summary (1)**

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time
    - **Temporal Locality**: Locality in Time
    - **Spatial Locality**: Locality in Space
- Three major categories of cache misses:
  - **Compulsory misses**: sad facts of life.  Example: cold start misses
  - **Conflict misses**:  increase cache size and/or associativity
    Nightmare Scenario: ping pong effect!
  - **Capacity misses**: increase cache size
- **Cache design space**
  - total size, block size, **associativity** (replacement policy)
  - write-hit policy (write-through, write-back)
  - write-miss policy (write allocate, write buffers)



2009-06-25

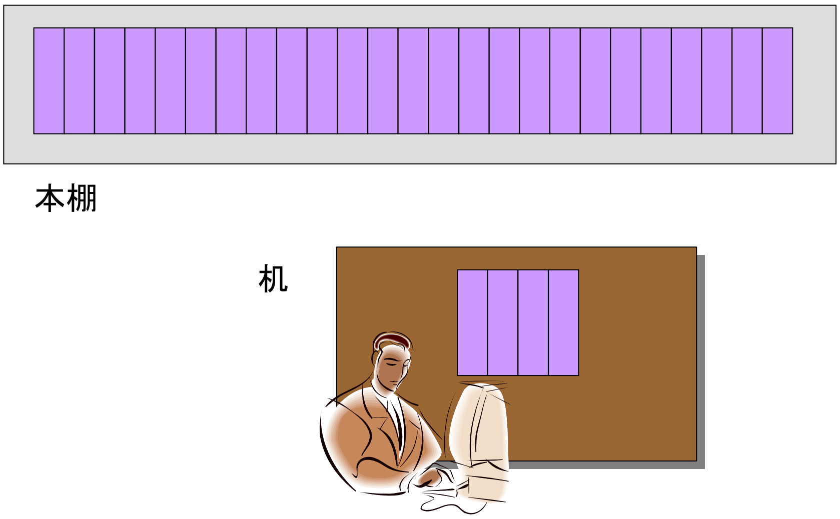
2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

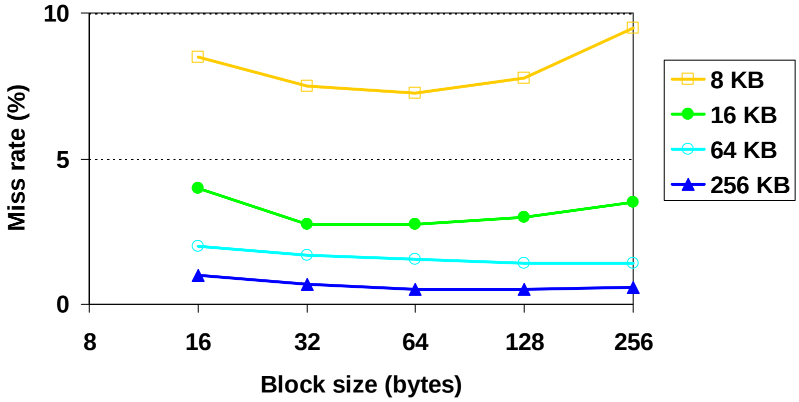
## 8. メモリ4:
## キャッシュシステム, プロセッサシミュレータ

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 － 14:50

## Cache



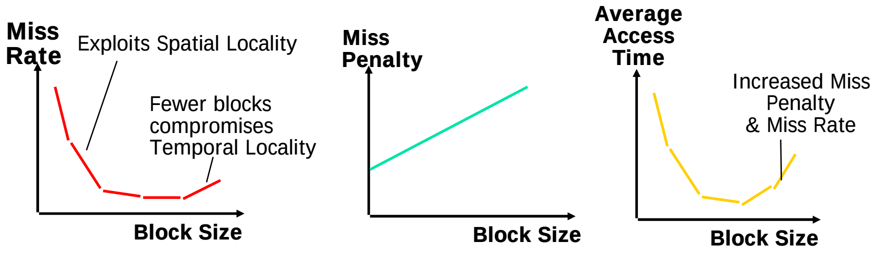
## Miss Rate vs Block Size vs Cache Size



- Miss rate goes up if the block size becomes a significant fraction of the cache size
because the number of blocks that can be held in the same size cache is smaller

## Block Size **Tradeoff**

- Larger block sizes take advantage of spatial locality **but**
  - If the block size is too big relative to the cache size, the miss rate will go up
  - Larger block size means larger miss penalty
    - Latency to first word in block + transfer time for remaining words



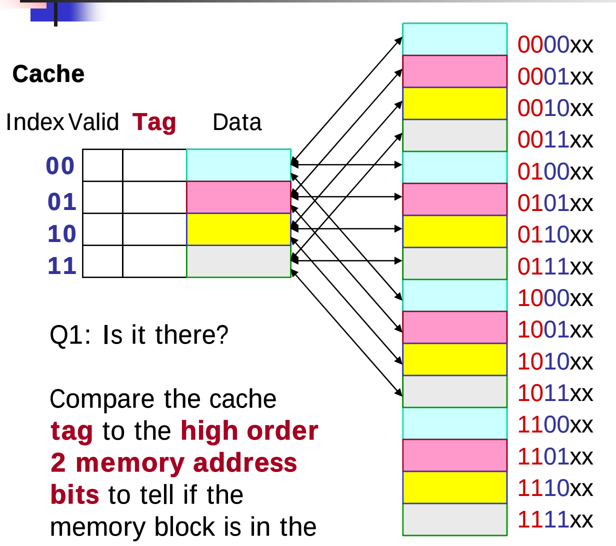
❑ In general, **Average Memory Access Time**

= Hit Time + **Miss Penalty x Miss Rate**

## Reducing Cache Miss Rates, **associativity**

- **Allow more flexible block placement**
  - In a **direct mapped cache** a memory block maps to exactly **one cache block**
  - At the other extreme, could allow a memory block to be mapped to any cache block – **fully associative cache**
  - A compromise is to divide the cache into **sets** each of which consists of **n "ways"** (**n-way set associative**). A memory block maps to a unique set and can be placed in any way of that set (so there are **n choices**)

## Caching: **A Simple First Example**



Two low order bits define the byte in the word (32-b words)

Q1: Is it there?

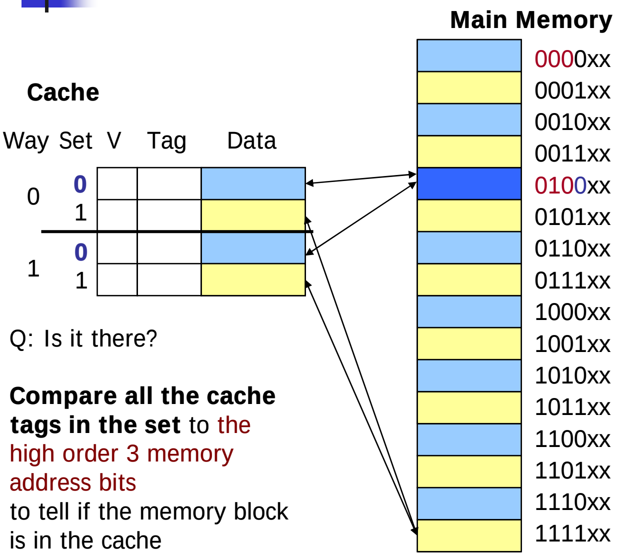
Compare the cache **tag** to the **high order 2 memory address bits** to tell if the memory block is in the cache

Q2: How do we find it?

Use **next 2 low order memory address bits** – the **index** – to determine which cache block

(block address) modulo (# of blocks in the cache)

## **Set Associative** Cache Example



Two low order bits define the byte in the word (32-b words)
**One word blocks**

Q: Is it there?

**Compare all the cache tags in the set** to the high order 3 memory address bits to tell if the memory block is in the cache

Q: How do we find it?

Use next 1 low order memory address bit to determine which cache set

# Another Reference String Mapping

- Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

0 4 0 4 0 4 0 4

**0** miss

| | |
|---|---|
| 000 | Mem(0) |
| | |
| | |
| | |

**4** miss

| | |
|---|---|
| 000 | Mem(0) |
| | |
| 010 | Mem(4) |
| | |

**0** hit

| | |
|---|---|
| 000 | Mem(0) |
| | |
| 010 | Mem(4) |
| | |

**4** hit

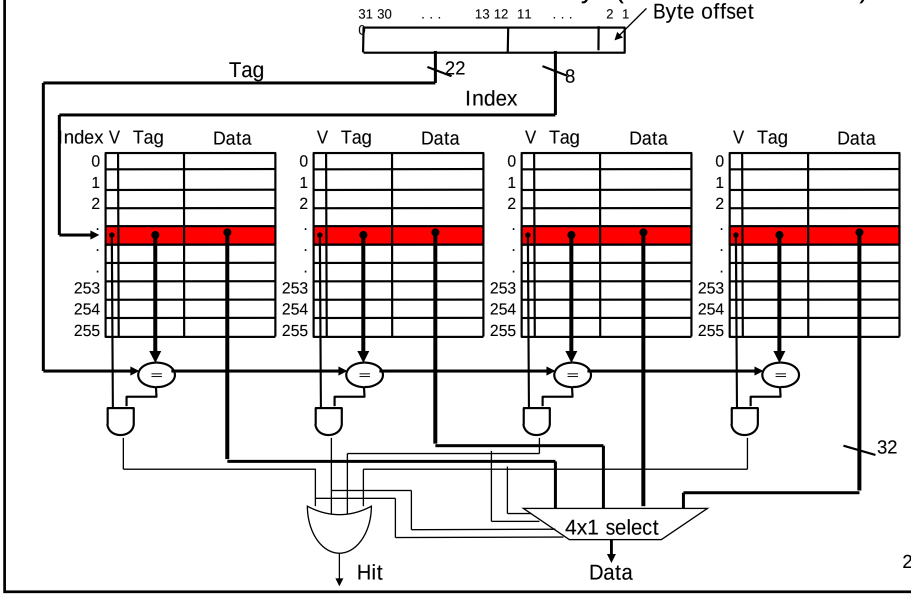
| | |
|---|---|
| 000 | Mem(0) |
| | |
| 010 | Mem(4) |
| | |

- 8 requests, 2 misses

- Solves the **ping pong effect** in a direct mapped cache due to conflict misses

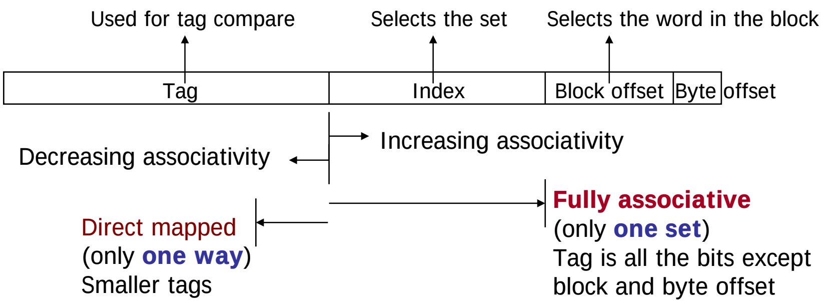
# Four-Way Set Associative Cache

- $2^8$ = **256 sets** each with four ways (each with one block)



## Range of Set Associative Caches

- For a fixed size cache



Used for tag compare

Selects the set

Selects the word in the block

| Tag | Index | Block offset | Byte offset |
|---|---|---|---|

Decreasing associativity

Increasing associativity

**Fully associative**
(only **one set**)
Tag is all the bits except
block and byte offset

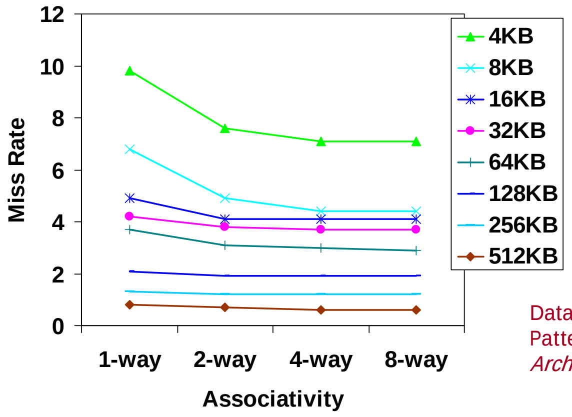
Direct mapped
(only **one way**)
Smaller tags

## **Costs** of Set Associative Caches

- **N-way set associative** cache costs
  - N comparators (delay and area)
  - MUX delay (set selection) before data is available
  - Data available **after** set selection and Hit/Miss decision.
- **When a miss occurs**,
  which way's block do we pick **for replacement** ?
  - **Least Recently Used (LRU):**
    the block replaced is the one that has been unused for the longest time
    - Must have hardware to keep track of when each way's block was used
    - For **2-way set associative**, takes **one bit per set** →
      set the bit when a block is referenced
      (and reset the other way's bit)
  - **Random**

## **Benefits** of Set Associative Caches

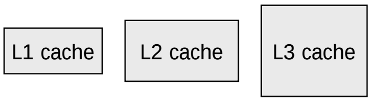
- The choice of direct mapped or set associative depends on **the cost of a miss** versus **the cost of implementation**



Data from Hennessy & Patterson, *Computer Architecture*, 2003

- Largest gains are in going from **direct mapped** to **2-way**

## Reducing Cache Miss Rates by **multiple levels**



- Enough room on the die for **bigger L1 caches** *or* for a **second level of caches** – normally a **unified** L2 cache (i.e., it holds both instructions and data) and in some cases even a **unified L3 cache**
- For our example,
  - $CPI_{ideal}$ of 2,
  - 100 cycle miss penalty (to main memory),
  - 36% load/stores,
  - a 2% (4%) L1I$ (D$) miss rate,
  - **add a UL2$ that has a 25 cycle miss penalty and a 0.5% miss rate**

$$CPI_{stalls} = 2 + .02 \times 25 + .36 \times .04 \times 25 + \mathbf{.005 \times 100} + \mathbf{.36 \times .005 \times 100} = 3.54$$

(as compared to **5.44** with no L2$)

## Multilevel Cache Design Considerations

- Design considerations for L1 and L2 caches are very different
  - Primary cache should focus on **minimizing hit time** in support of a shorter clock cycle
  - Secondary cache should focus on **reducing miss rate** to reduce the penalty of long main memory access times
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache – so it can be smaller (i.e., faster) but have a higher miss rate
- For the L2 cache, hit time is less important than miss rate
  - The L2$ hit time determines L1$'s miss penalty

## Key Cache Design Parameters

| | L1 typical | L2 typical |
|---|---|---|
| Total size (blocks) | 250 to 2000 | 4000 to 250,000 |
| Total size (KB) | 16 to 64 | 500 to 8000 |
| Block size (B) | 32 to 64 | 32 to 128 |
| Miss penalty (clocks) | 10 to 25 | 100 to 1000 |
| Miss rates | 2% to 5% | 0.1% to 2% |

## Two Machines' Cache Parameters

| | Intel P4 | AMD Opteron |
|---|---|---|
| L1 organization | Split I$ and D$ | Split I$ and D$ |
| L1 cache size | 8KB for D$, 96KB for trace cache (~I$) | 64KB for each of I$ and D$ |
| L1 block size | 64 bytes | 64 bytes |
| L1 associativity | 4-way set assoc. | 2-way set assoc. |
| L1 replacement | ~ LRU | LRU |
| L1 write policy | write-through | write-back |
| L2 organization | Unified | Unified |
| L2 cache size | 512KB | 1024KB (1MB) |
| L2 block size | 128 bytes | 64 bytes |
| L2 associativity | 8-way set assoc. | 16-way set assoc. |
| L2 replacement | ~LRU | ~LRU |
| L2 write policy | write-back | write-back |

231

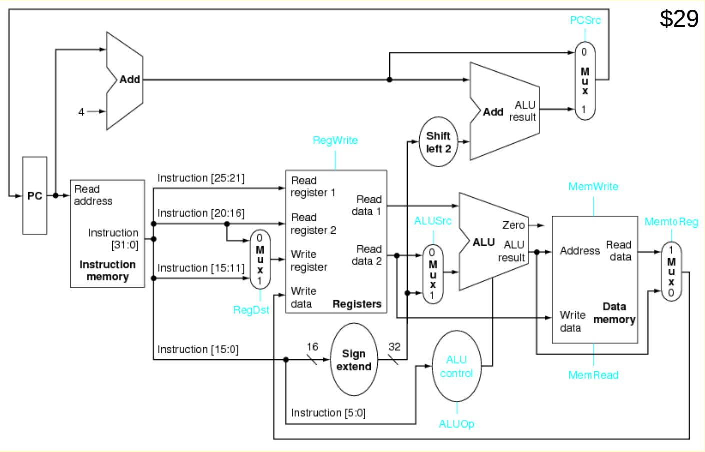
## プロセッサのデータパス（シングル・サイクル）

| op | rs | rt | 16 bit immediate |
|---|---|---|---|

I format

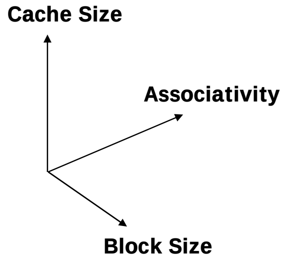
**addi $sp, $sp, 4 [ addi $29, $29, 4 ]**

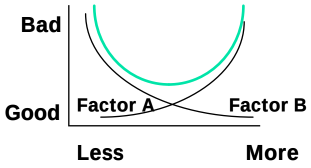
PC = 36
$29 = 8000



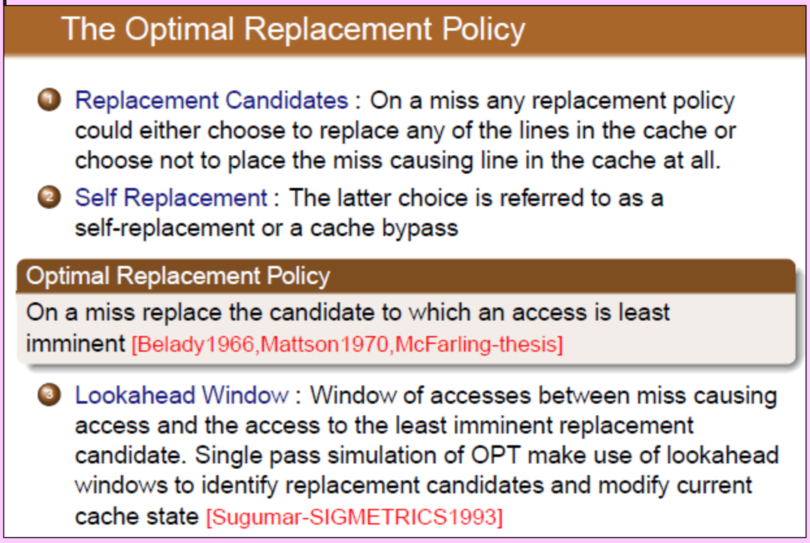
## Summary: The Cache Design Space

- **Several interacting dimensions**
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- **The optimal choice is a compromise**
  - depends on access characteristics
    - workload
    - I-cache, D-cache
  - depends on technology / cost
- **Simplicity** often wins





## OPT: Optimal Replacement Policy



### The Optimal Replacement Policy

1. Replacement Candidates : On a miss any replacement policy could either choose to replace any of the lines in the cache or choose not to place the miss causing line in the cache at all.
2. Self Replacement : The latter choice is referred to as a self-replacement or a cache bypass

**Optimal Replacement Policy**

On a miss replace the candidate to which an access is least imminent [Belady1966,Mattson1970,McFarling-thesis]

3. Lookahead Window : Window of accesses between miss causing access and the access to the least imminent replacement candidate. Single pass simulation of OPT make use of lookahead windows to identify replacement candidates and modify current cache state [Sugumar-SIGMETRICS1993]

OPT: あまり切迫していないものを置き換える.

MICRO-40 Emulating Optimal Replacement with a Shepherd Cache

## Optimal Replacement Policy の例

### Understanding OPT

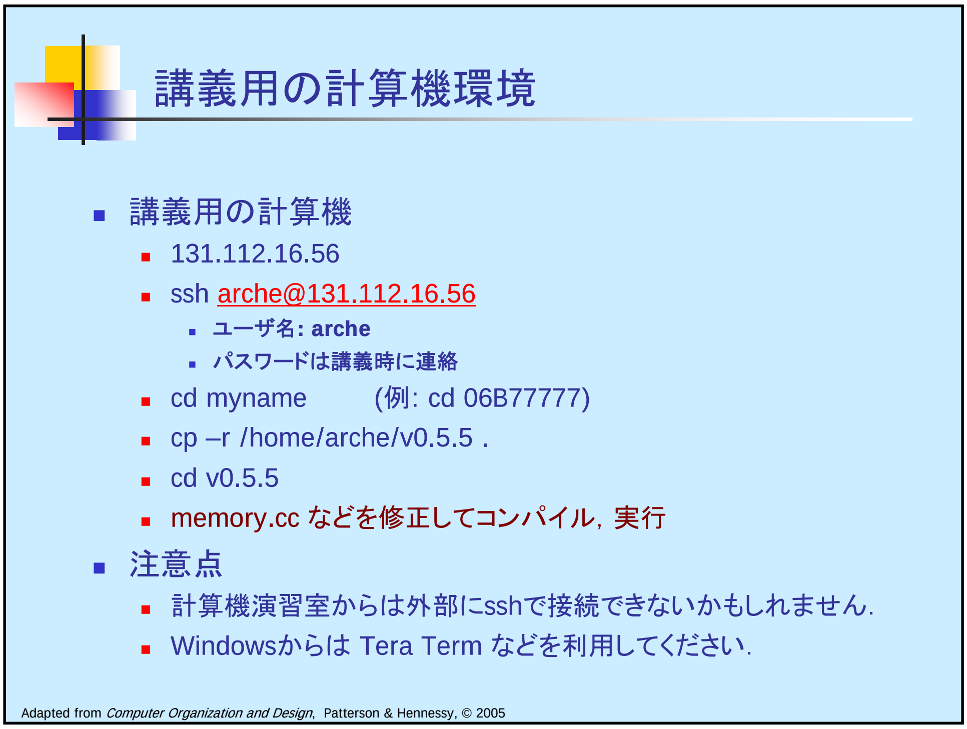
| Access Sequence | $A_5$ | $A_1$ | $A_6$ | $A_3$ | $A_1$ | $A_4$ | $A_5$ | $A_2$ | $A_5$ | $A_7$ | $A_6$ | $A_8$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| OPT order for $A_5$ | | 0 | | 1 | | 2 | 3 | 4 | | | | |
| OPT order for $A_6$ | | | | 0 | 1 | 2 | 3 | | | | 4 | |

- Consider 4 way associative cache with one set initially containing lines ($A_1$,$A_2$,$A_3$,$A_4$), consider the access stream shown in table
- Access $A_5$ misses, replacement decision proceeds as follows
  1. Identify replacement candidates : ($A_1$,$A_2$,$A_3$,$A_4$,$A_5$)
  2. Lookahead and gather imminence order : shown in table, lookahead window circled
  3. Make replacement decision : $A_5$ replaces $A_2$
- $A_6$ self-replaces, lookahead window and imminence order in table

MICRO-40 Emulating Optimal Replacement with a Shepherd Cache

## レポート 問題

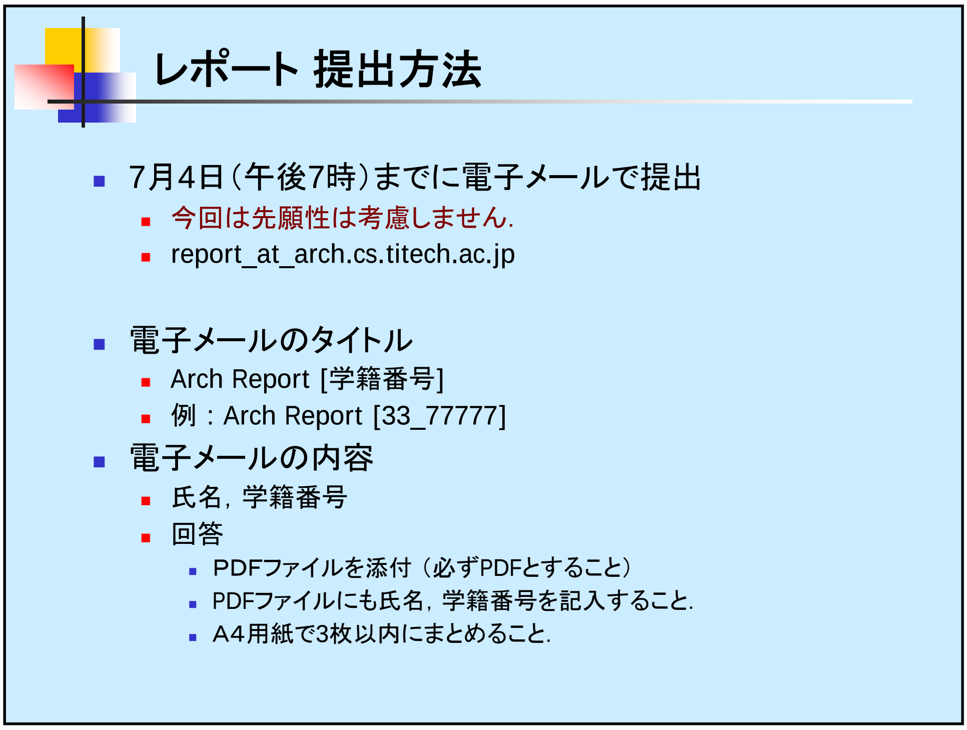
1. SimMipsにデータキャッシュのヒット率を測定する仕組みを追加し，ヒット率を測定せよ．
   1. ダイレクトマップ方式，ラインサイズは４ワードとする．
   2. セット数を８，１６，３２，６４，１２８，２５６，５１２に変更した場合のヒット率を示せ．
   3. 以前作成した sort（1000要素のランダムデータ）を含む３つのアプリケーションを作成し，そのヒット率を示すこと．
2. キャッシュのヒット率を改善する方式を実装し，その効果を示せ．
   1. 例えば，ラインサイズの変更
   2. 例えば，セットアソシアティブ方式
   3. 例えば，フルアソシアティブ方式
3. レポートはA4用紙3枚以内にまとめること．（必ずPDFとすること）
   （2段組，コードは小さい文字でもかまわない．）



## 講義用の計算機環境

- 講義用の計算機
  - 131.112.16.56
  - ssh arche@131.112.16.56
    - **ユーザ名: arche**
    - **パスワードは講義時に連絡**
  - cd myname　　(例: cd 06B77777)
  - cp –r /home/arche/v0.5.5 .
  - cd v0.5.5
  - memory.cc などを修正してコンパイル，実行
- 注意点
  - 計算機演習室からは外部にsshで接続できないかもしれません.
  - Windowsからは Tera Term などを利用してください.

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## レポート 提出方法

- 7月4日（午後7時）までに電子メールで提出
  - 今回は先願性は考慮しません.
  - report_at_arch.cs.titech.ac.jp
- 電子メールのタイトル
  - Arch Report [学籍番号]
  - 例：Arch Report [33_77777]
- 電子メールの内容
  - 氏名，学籍番号
  - 回答
    - PDFファイルを添付（必ずPDFとすること）
    - PDFファイルにも氏名，学籍番号を記入すること.
    - A4用紙で3枚以内にまとめること.

2009-07-02

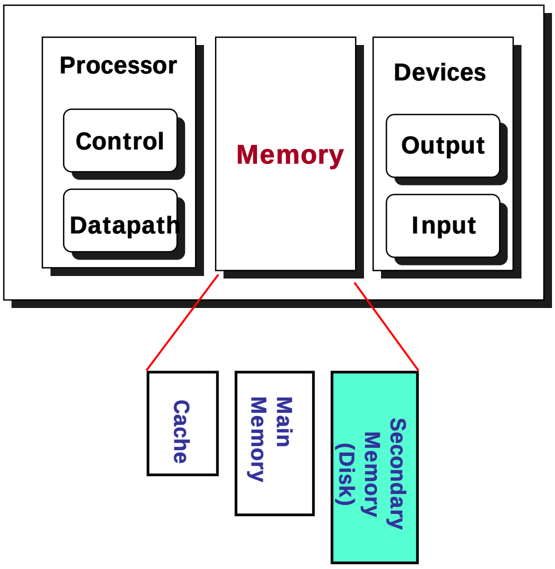
2009年 前学期 TOKYO TECH

# 計算機アーキテクチャ 第一 (E)

## ９. 磁気ディスク, RAID

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日１３:２０ － １４:５０

## Major Components of a Computer


Processor

Control

Datapath

Memory

Devices

Output

Input

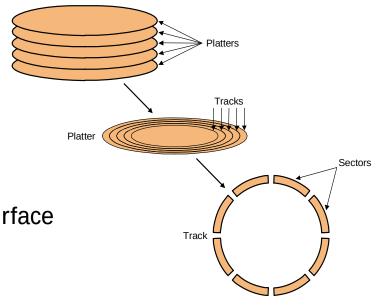
Cache

Main Memory

Secondary Memory (Disk)

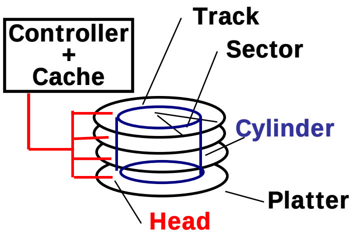
# Magnetic Disk（磁気ディスク）

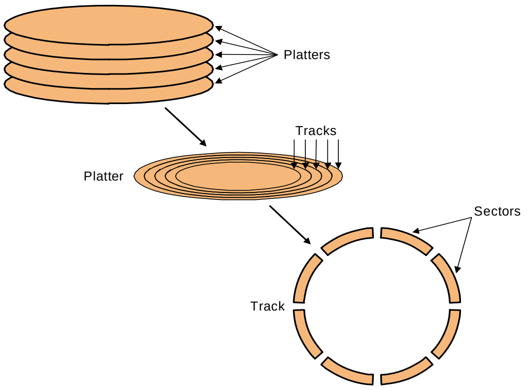
- Purpose
  - Long term, **nonvolatile**（不揮発性）storage
  - Lowest level in the memory hierarchy
    - slow, large, inexpensive



- General structure
  - A rotating **platter** coated with a magnetic surface
  - A moveable read/write **head** to access the information on the disk
- Typical numbers
  - 1 to 4 platters per disk of 1” to 5.25” in diameter (3.5” dominate in 2004)
  - Rotational speeds of 5,400 to 15,000 RPM (rotation per minute)
  - 10,000 to 50,000 **tracks** per surface
    - **cylinder** - all the tracks under the head at a given point on all surfaces
  - 100 to 500 **sectors** per track
    - the smallest unit that can be read/written (typically 512B)

# Disk Drives



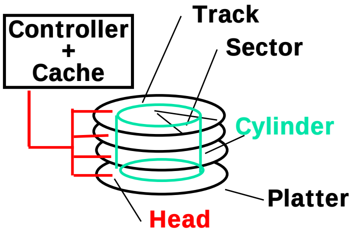


**To access data:**

- **seek time**（シーク時間）: position head over the proper track
- **rotational latency**（回転待ち時間）: wait for desired sector
- **transfer time**（転送時間）: grab the data (one or more sectors)
- **Controller time**（制御時間）: the overhead the disk controller imposes in performing a disk I/O access

## Magnetic Disk Characteristic

- **Disk read/write components**



1. **Seek time**: position the head over the proper track (**3 to 14 ms avg**)
    - due to locality of disk references the actual average seek time may be only 25% to 33% of the advertised number
2. **Rotational latency**: wait for the desired sector to rotate under the head (½ of 1/RPM converted to ms)
    - 0.5/5400RPM = **5.6ms** to 0.5/15000RPM = **2.0ms**
3. **Transfer time**: transfer a block of bits (one or more sectors) under the head to the disk controller's cache (**30 to 80 MB/s** are typical disk transfer rates)
4. **Controller time**: the overhead the disk controller imposes in performing a disk I/O access (**typically < .2 ms**)

## Typical Disk Access Time

- The average time to read or write a 512B sector for a disk rotating at 10,000RPM with average seek time of 6ms, a 50MB/sec transfer rate, and a 0.2ms controller overhead

**Avg disk read/write time**
**= 6.0ms + 0.5/(10000RPM/(60sec/minute) )+ 0.5KB/(50MB/sec) + 0.2ms**
**= 6.0 + 3.0 + 0.01 + 0.2**
**= 9.21ms**

If the measured average seek time is **25%** of the advertised average seek time, then

**Avg disk read/write = 1.5 + 3.0 + 0.01 + 0.2 = 4.71ms**

- The **rotational latency** is usually the largest component of the access time

## Disk Latency & Bandwidth Milestones

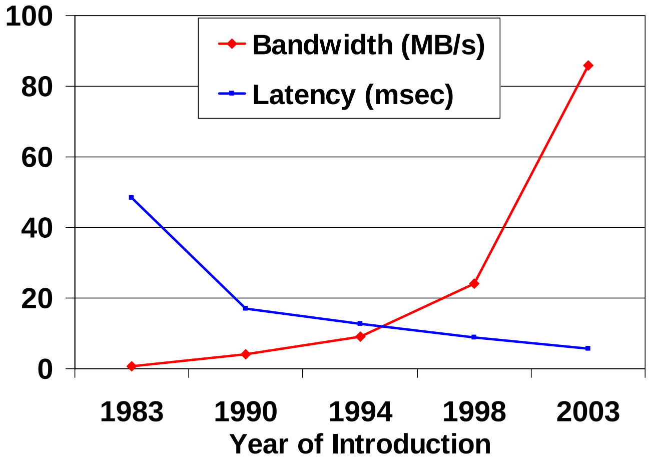
- Disk **latency** is one average seek time plus the rotational latency.
- Disk **bandwidth** is the peak transfer time of formatted data from the media (not from the cache).

| | CDC Wren | SG ST41 | SG ST15 | SG ST39 | SG ST37 |
|---|---|---|---|---|---|
| Speed (RPM) | 3600 | 5400 | 7200 | 10000 | 15000 |
| **Year** | **1983** | **1990** | **1994** | **1998** | **2003** |
| Capacity (Gbytes) | 0.03 | 1.4 | 4.3 | 9.1 | 73.4 |
| Diameter (inches) | 5.25 | 5.25 | 3.5 | 3.0 | 2.5 |
| Interface | ST-412 | SCSI | SCSI | SCSI | SCSI |
| **Bandwidth** (MB/s) | 0.6 | 4 | 9 | 24 | 86 |
| **Latency** (msec) | 48.3 | 17.1 | 12.7 | 8.8 | 5.7 |

**Patterson, CACM Vol 47, #10, 2004**

## Latency & Bandwidth Improvements

- In the time that the disk **bandwidth doubles** the **latency** improves by a factor of only **1.2** to **1.4**



## Reliability（信頼性）, Availability

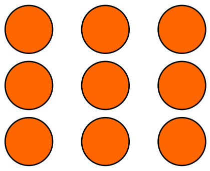
- Reliability – measured by the mean time to failure (平均故障寿命, MTTF). Service interruption is measured by mean time to repair (平均修復時間, MTTR)
- Availability（アベイラビリティ）

  Availability = MTTF / (MTTF + MTTR)

- To increase MTTF, either improve the quality of the components or design the system to continue operating in the presence of faulty components
  1. Fault avoidance: preventing fault occurrence by construction
  2. Fault tolerance: using redundancy to correct or bypass faulty components (hardware)

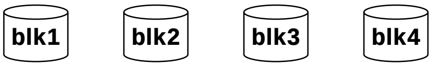
## **RAID**: Disk Arrays

**Redundant Array of Inexpensive Disks**



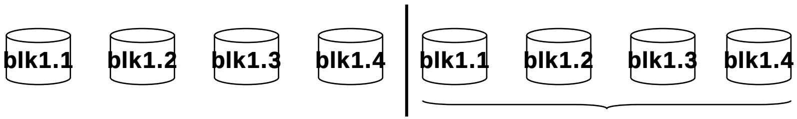
- Arrays of small and inexpensive disks
  - Increase potential **throughput** by having many disk drives
    - Data is spread over multiple disk
    - Multiple accesses are made to several disks at a time
- **Reliability** is lower than a single disk
- But **availability** can be improved by adding redundant disks (RAID)

## RAID: **Level 0** (冗長性なし; Striping ストライピング)



- Multiple smaller disks as opposed to one big disk
  - Spreading the blocks over multiple disks – **striping** – means that multiple blocks can be accessed in parallel increasing the performance
    - A 4 disk system gives four times the throughput of a 1 disk system
  - Same cost as one *big* disk – assuming 4 small disks cost the same as one big disk
- No redundancy, so what if one disk fails?

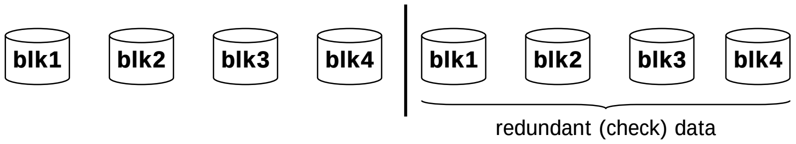
## RAID: **Level 1** (Redundancy via Mirroring)



redundant (check) data

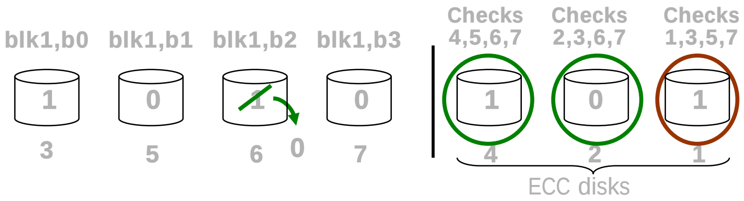
- Uses twice as many disks for redundancy so there are always two copies of the data
  - The number of redundant disks = the number of data disks so twice the cost of one big disk
    - writes have to be made to both sets of disks, so writes would be only 1/2 the performance of RAID 0
- What if one disk fails?
  - If a disk fails, the system just goes to the "**mirror**" for the data

## RAID: **Level 0+1** (Striping with Mirroring)



- Combines the best of RAID 0 and RAID 1,
  data is striped across four disks and mirrored to four disks
  - Four times the throughput (due to striping)
  - # redundant disks = # of data disks
    so twice the cost of one big disk
    - writes have to be made to both sets of disks,
      so writes would be only 1/2 the performance of RAID 0
- What if one disk fails?
  - If a disk fails, the system just goes to the “**mirror**” for the data

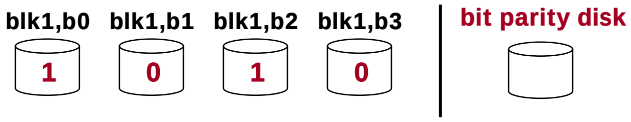
## RAID: Level 2 (Redundancy via ECC)



誤り訂正コード **(ECC, error-correcting code)** disks 4 and 2 point to either data disk 6 or 7, but ECC disk 1 says disk 7 is okay, so disk 6 must be in error

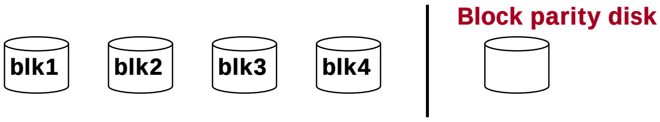
- ECC disks contain the parity of data on a set of distinct overlapping disks
  - # redundant disks = log (total # of data disks)
    so almost twice the cost of one big disk
    - writes require computing parity to write to the ECC disks
    - reads require reading ECC disk and confirming parity

## RAID: Level 3 (Bit-Interleaved **Parity**)



- Cost of higher availability is reduced to 1/N where N is the number of disks in a protection group（保護グループ）
  - # redundant disks = 1 × # of protection groups
    - **writes** require writing the new data to the data disk as well as computing the parity, meaning reading the other disks, so that the parity disk can be updated
    - **reads** require reading all the operational data disks as well as the parity disk to calculate the missing data that was stored on the **failed disk**

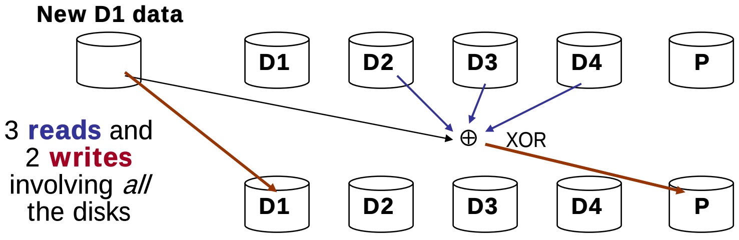
## RAID: Level 4 (Block-Interleaved Parity)



- Cost of higher availability still only 1/N but the parity is stored as **blocks** associated with sets of data blocks
  - Four times the throughput (striping)
  - # redundant disks = 1 × # of protection groups
  - Supports “small reads” and “small writes” (reads and writes that go to just one (or a few) data disk in a protection group)

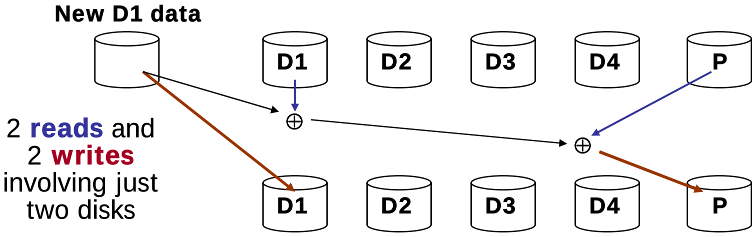
## Small Writes

- RAID 3


**New D1 data**

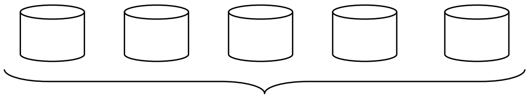
3 **reads** and
2 **writes**
involving *all*
the disks

- RAID 4 small writes


**New D1 data**

2 **reads** and
2 **writes**
involving just
two disks

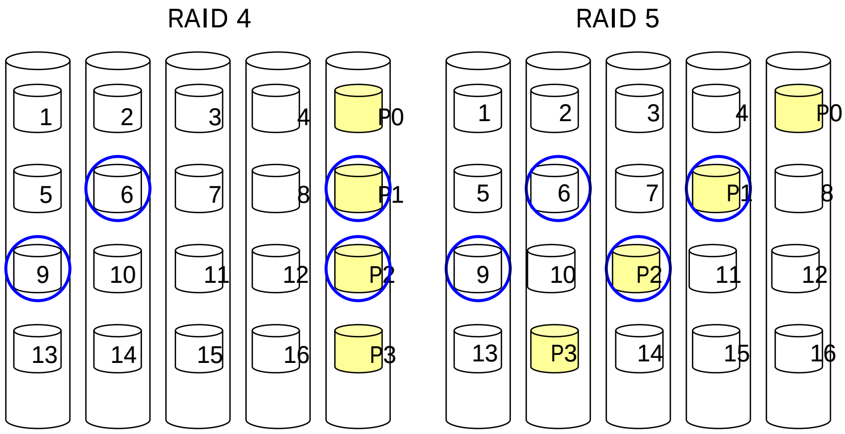
## **RAID: Level 5** (Distributed Block-Interleaved Parity)



one of these assigned as the block parity disk

- Cost of higher availability still only 1/N but the parity block can be located on any of the disks
  so there is no single bottleneck for writes
  - Still four times the throughput (striping)
  - # redundant disks = 1 × # of protection groups
  - Supports "**small reads**" and "**small writes**" (reads and writes that go to just one (or a few) data disk in a protection group)
  - Allows **multiple simultaneous writes**

## Distributing Parity Blocks



RAID 4

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | P0 |
| 5 | 6 | 7 | 8 | P1 |
| 9 | 10 | 11 | 12 | P2 |
| 13 | 14 | 15 | 16 | P3 |

RAID 5

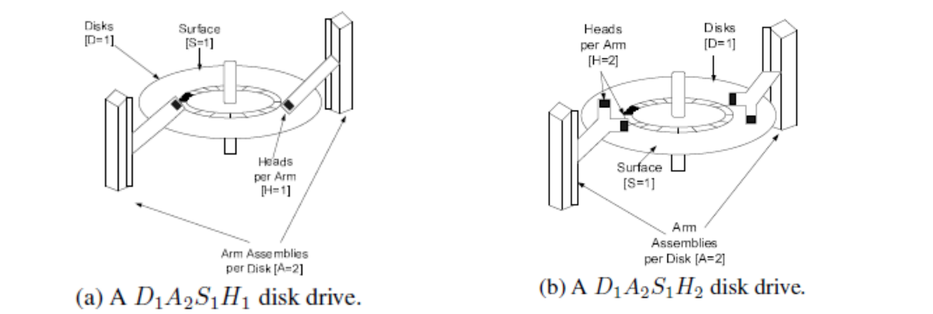
| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | P0 |
| 5 | 6 | 7 | P1 | 8 |
| 9 | 10 | P2 | 11 | 12 |
| 13 | P3 | 14 | 15 | 16 |

- By distributing parity blocks to all disks, some small writes can be performed **in parallel**

## Disk and RAID **Summary**

- Four components of disk access time:
  - Seek Time: advertised to be 3 to 14 ms but lower in real systems
  - Rotational Latency: 5.6 ms at 5400 RPM and 2.0 ms at 15000 RPM
  - Transfer Time: 30 to 80 MB/s
  - Controller Time: typically less than .2 ms
- RAIDs can be used to improve availability
  - RAID 0 and RAID 5 – widely used in servers, one estimate is that 80% of disks in servers are RAIDs
  - RAID 1 (mirroring) – EMC, Tandem, IBM
  - RAID 3 – Storage Concepts
  - RAID 4 – Network Appliance
- RAIDs have enough redundancy to allow continuous operation

## Intra-Disk Parallelism: An Idea Whose Time Has Come, ISCA2008



(a) A $D_1A_2S_1H_1$ disk drive.

(b) A $D_1A_2S_1H_2$ disk drive.

**Figure 1. Example design points within the $DASH$ intra-disk parallelism taxonomy.**

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

2009-07-09

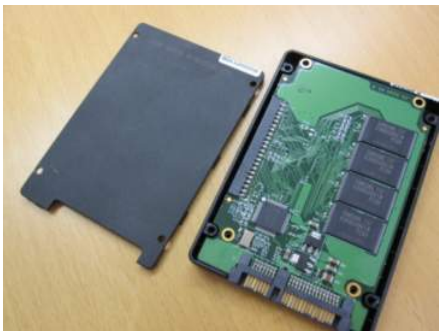
# 計算機アーキテクチャ 第一 (E)

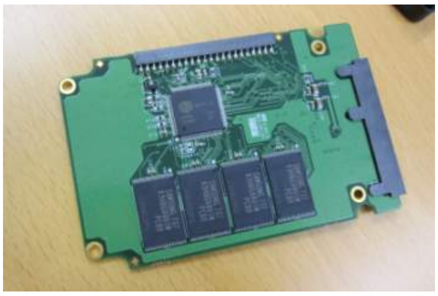
## 10. 主記憶とファイルメモリの管理, 多重仮想記憶, 記憶保護

吉瀬 謙二 計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日13:20 ー 14:50

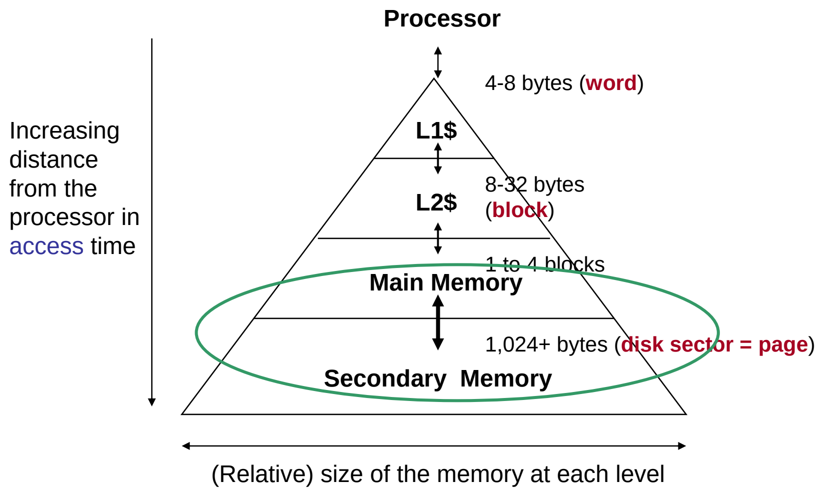
## SSD (Solid State Drive)

## SSD (Solid State Drive)





## Memory Hierarchy


Processor
4-8 bytes (word)
L1$
8-32 bytes (block)
L2$
1 to 4 blocks
Main Memory
1,024+ bytes (disk sector = page)
Secondary Memory
Increasing distance from the processor in access time
(Relative) size of the memory at each level

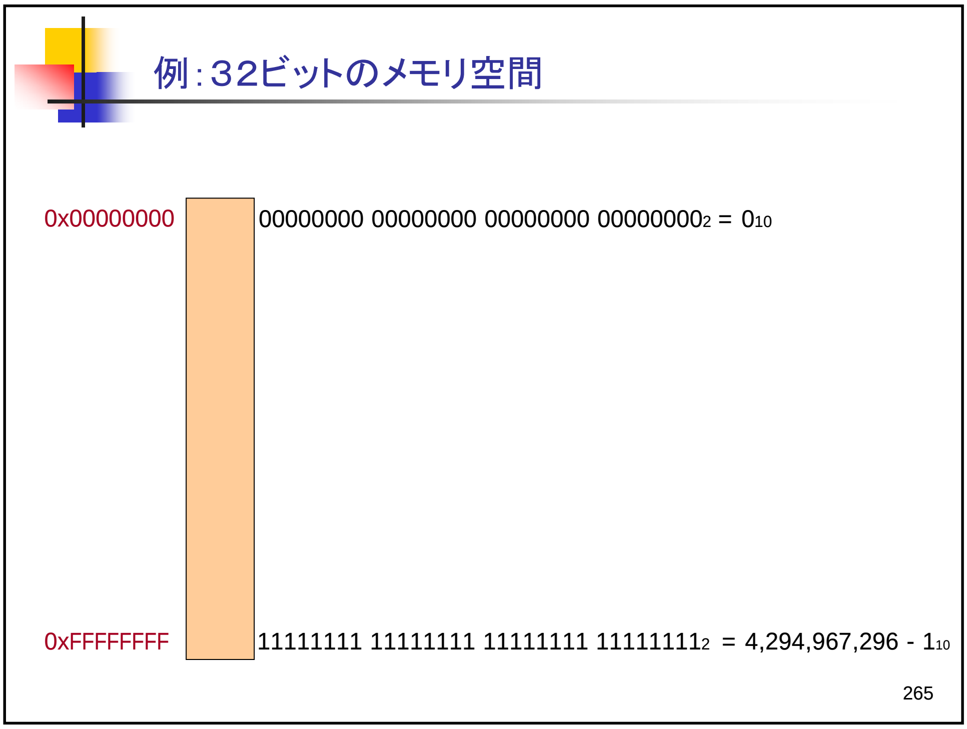
## Loading and Storing Bytes

- MIPS has two basic data transfer instructions for accessing memory

```
lw  $t0, 4($s3)  # load word from memory
sw  $t0, 8($s3)  # store word to memory
```

- The data is loaded into (**lw**) or stored from (**sw**) a register in the register file
- The memory address – a 32 bit address – is formed by adding the contents of the base address register to the offset value

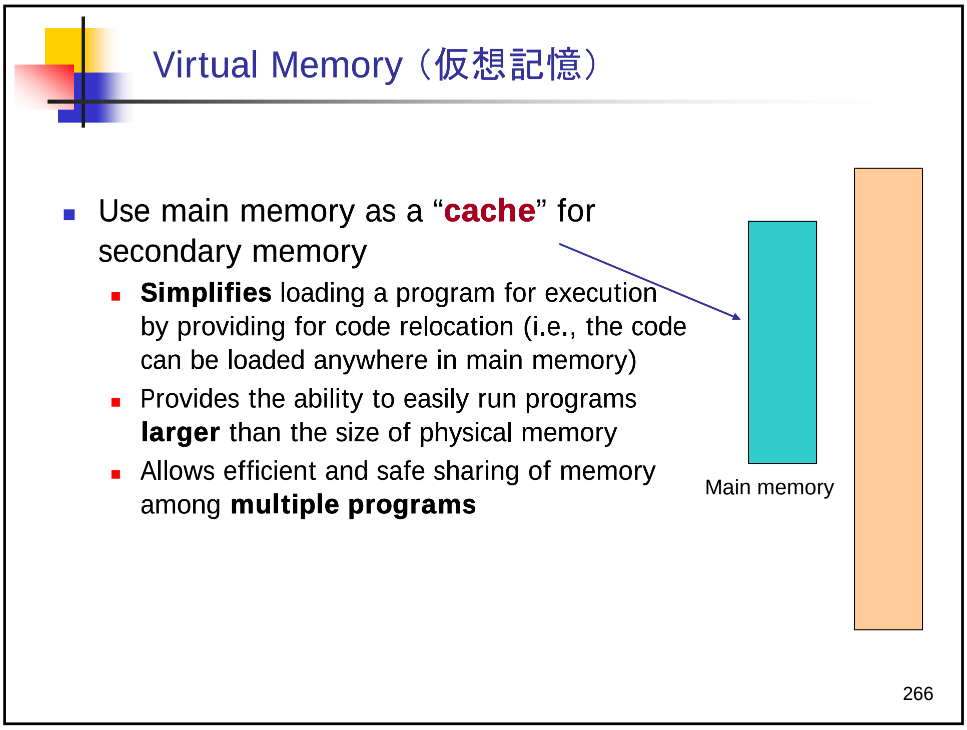
| op | rs | rt | 16 bit offset |
|---|---|---|---|



## 例:32ビットのメモリ空間

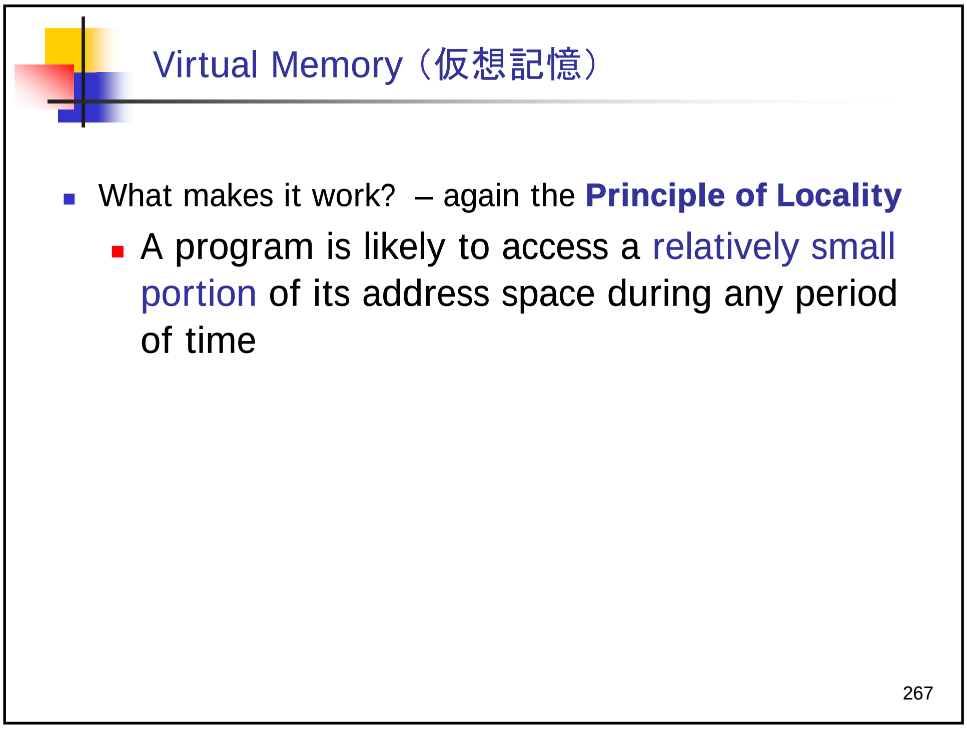
0x00000000 $00000000\ 00000000\ 00000000\ 00000000_2 = 0_{10}$

0xFFFFFFFF $11111111\ 11111111\ 11111111\ 11111111_2 = 4{,}294{,}967{,}296 - 1_{10}$



## Virtual Memory（仮想記憶）

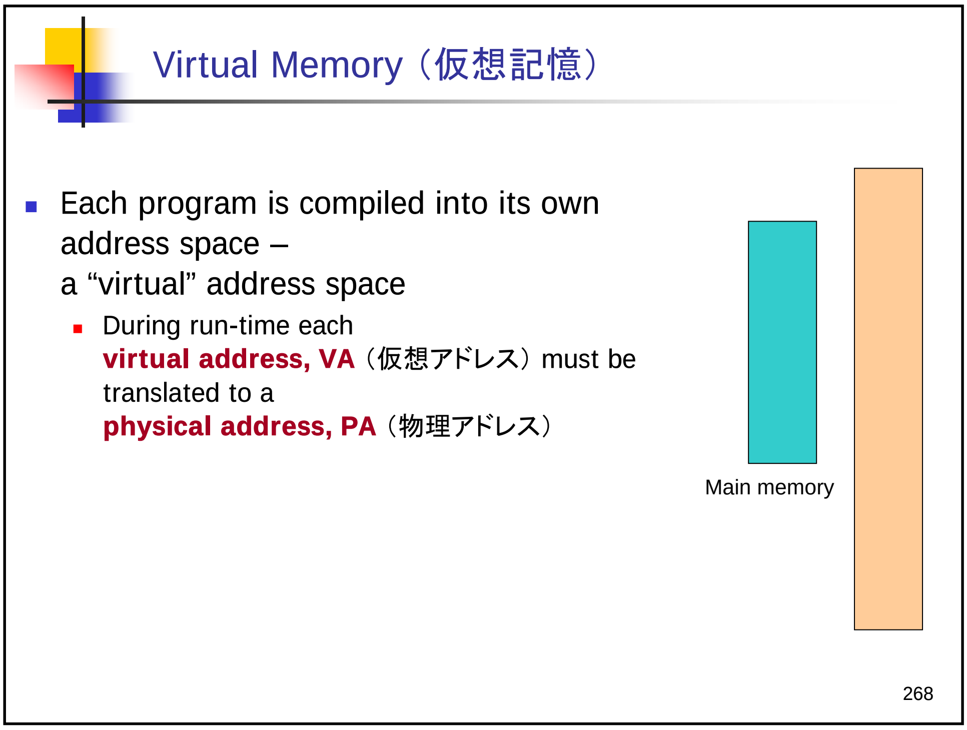
- Use main memory as a “**cache**” for secondary memory
  - **Simplifies** loading a program for execution by providing for code relocation (i.e., the code can be loaded anywhere in main memory)
  - Provides the ability to easily run programs **larger** than the size of physical memory
  - Allows efficient and safe sharing of memory among **multiple programs**



## Virtual Memory（仮想記憶）

- What makes it work?  – again the **Principle of Locality**
  - A program is likely to access a relatively small portion of its address space during any period of time

267



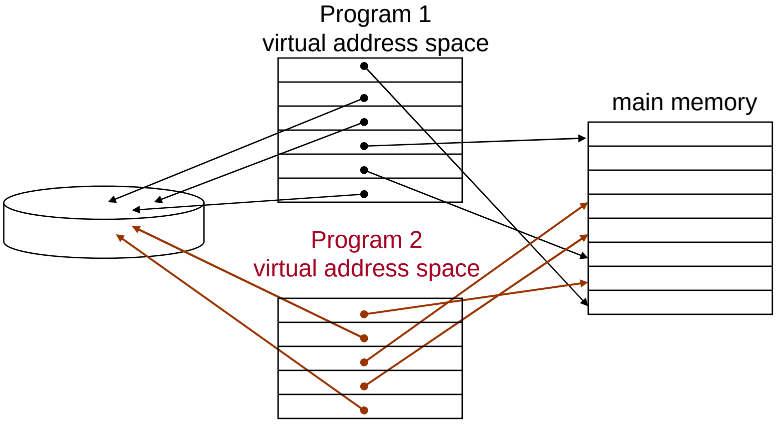
## Virtual Memory（仮想記憶）

- Each program is compiled into its own address space –
  a “virtual” address space
  - During run-time each **virtual address, VA**（仮想アドレス）must be translated to a **physical address, PA**（物理アドレス）

268

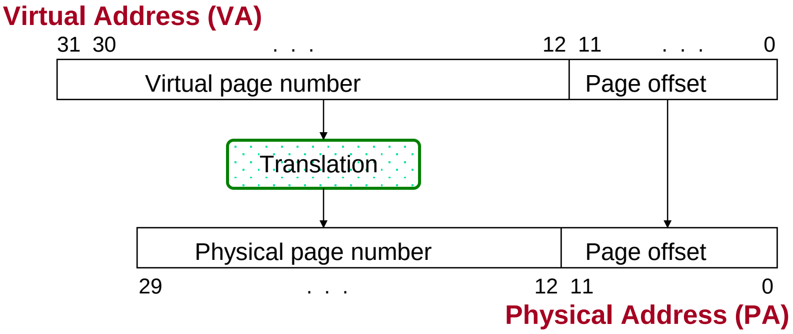
## Two Programs Sharing Physical Memory

- A program's address space is divided into **pages** (all one fixed size) or **segments** (variable sizes)
  - The starting location of each page (either in **main memory** or in **secondary memory**) is contained in the program's **page table**



## Address Translation

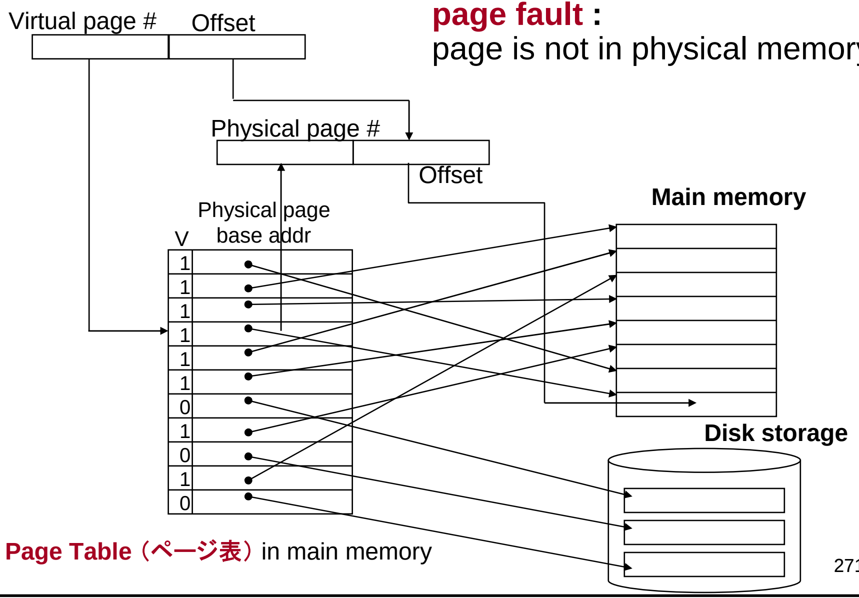
- A virtual address is translated to a physical address by a combination of hardware and software



- So each memory request **first** requires an **address translation** from the virtual space to the physical space

## Address Translation Mechanisms

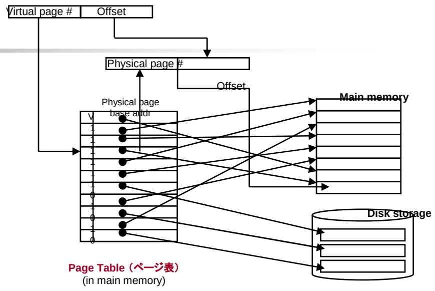
**page fault :**
page is not in physical memory



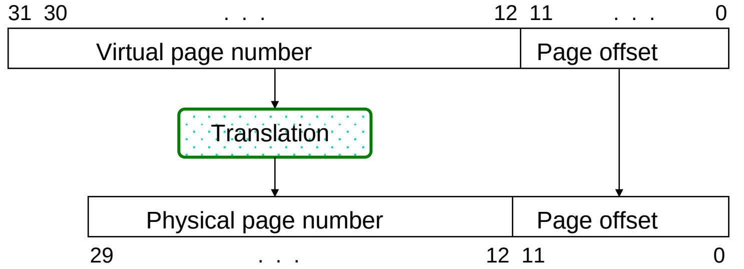
**Page Table（ページ表）**in main memory

271

## Address Translation



**Virtual Address (VA)**



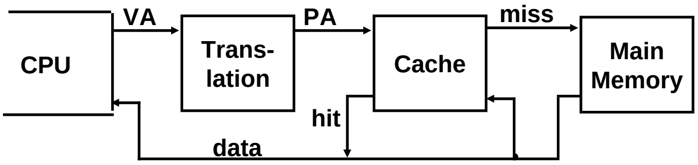
**Physical Address (PA)**

- ページサイズは？ ページ表のメモリサイズは？

272

## Virtual Addressing with a **Cache**

- Thus it takes an ***extra*** memory access to translate a virtual address to a physical address



CPU

VA

Trans-lation

PA

Cache

miss

Main Memory

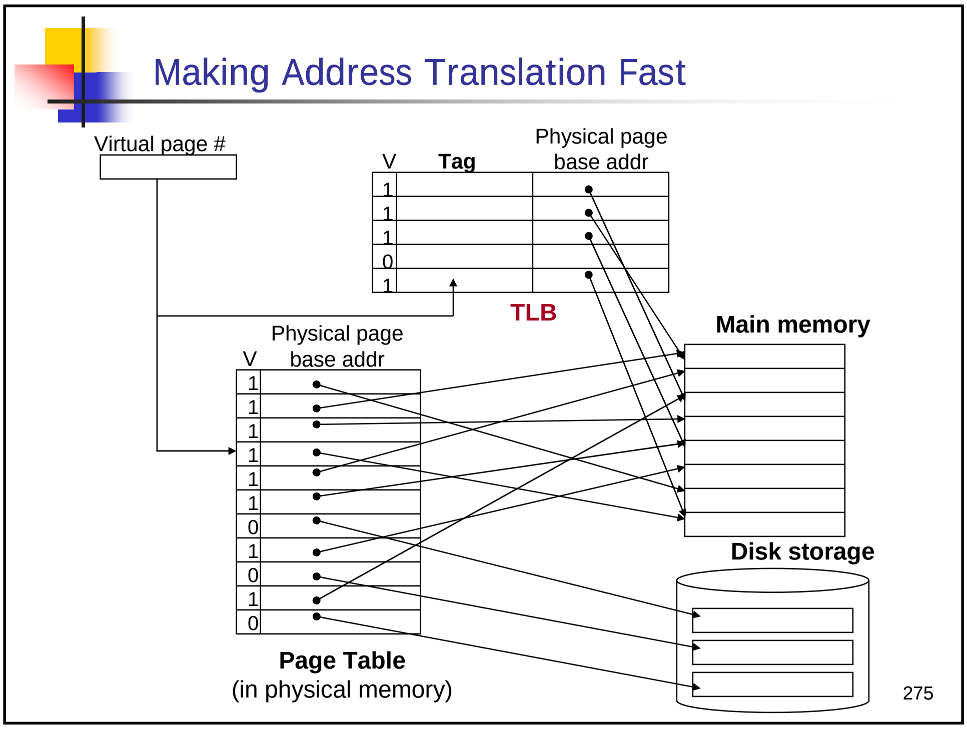
hit

data

- This makes memory (cache) accesses **very expensive** (if every access was really ***two*** accesses)
- The hardware fix is to use a **Translation Lookaside Buffer (TLB)** – a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup

## Virtual Addressing, the hardware fix

- The hardware fix is to use a **Translation Lookaside Buffer (TLB)**（アドレス変換バッファ）
  - a small cache that keeps track of recently used address mappings to avoid having to do a page table lookup



## Making Address Translation Fast

Virtual page #

V | Tag | Physical page base addr

1
1
1
0
1

TLB

Physical page base addr

V

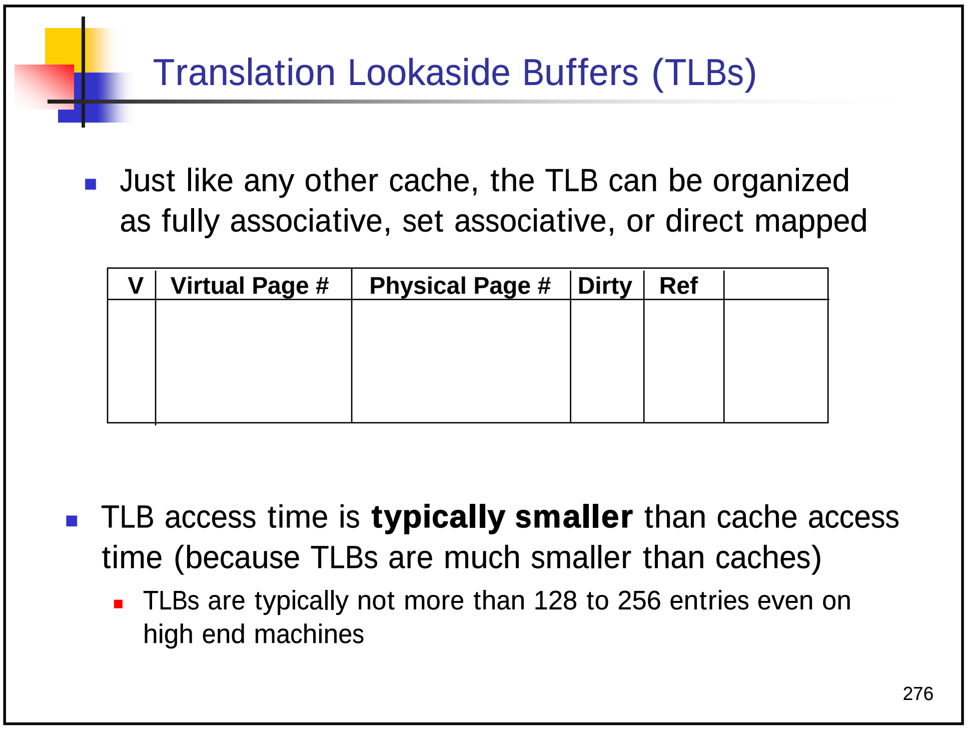
1
1
1
1
1
1
0
1
0
1
0

Page Table
(in physical memory)

Main memory

Disk storage

275



## Translation Lookaside Buffers (TLBs)

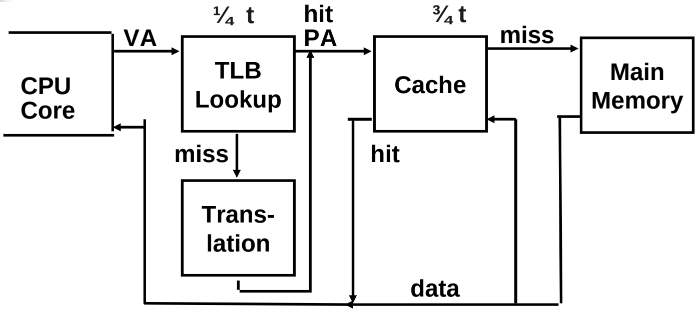
- Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

| V | Virtual Page # | Physical Page # | Dirty | Ref | |
|---|---|---|---|---|---|
| | | | | | |

- TLB access time is **typically smaller** than cache access time (because TLBs are much smaller than caches)
  - TLBs are typically not more than 128 to 256 entries even on high end machines

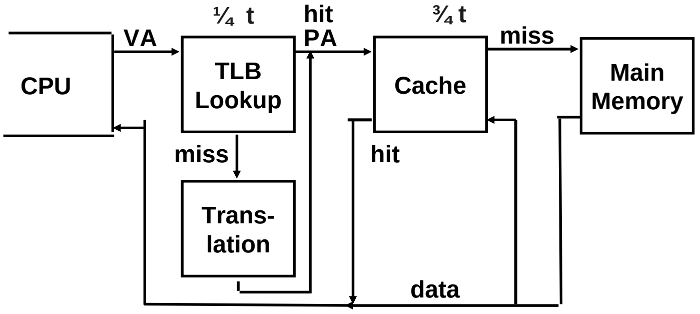
276

## A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
    - Takes 10's of cycles to find and load the translation info into the TLB
  - If the page is not in main memory, then it's a true page fault
    - Takes 1,000,000's of cycles to service a page fault

## A TLB in the Memory Hierarchy



- **page fault** : page is not in physical memory
- **TLB misses** are much more frequent than true page faults

## Two Machines' TLB Parameters

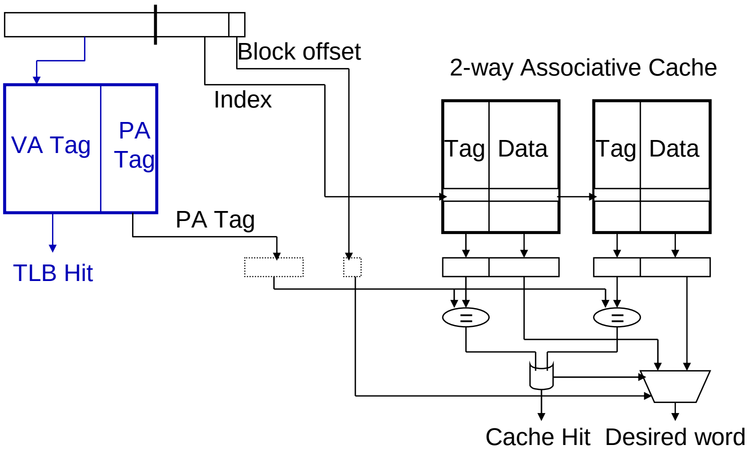
| | Intel P4 | AMD Opteron |
|---|---|---|
| TLB organization | 1 TLB for instructions and 1TLB for data<br>Both 4-way set associative<br>Both use ~LRU replacement<br>Both have 128 entries<br>TLB misses handled in hardware | 2 TLBs for instructions and 2 TLBs for data<br>Both L1 TLBs fully associative with ~LRU replacement<br>Both L2 TLBs are 4-way set associative with round-robin LRU<br>Both L1 TLBs have 40 entries<br>Both L2 TLBs have 512 entries<br>TBL misses handled in hardware |

## TLB Event Combinations

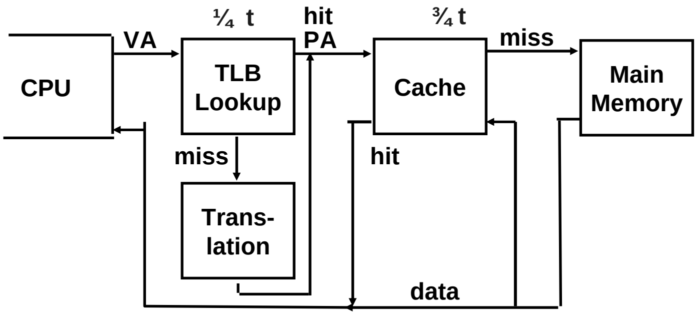
| TLB | Page Table | Cache | Possible? Under what circumstances? |
|---|---|---|---|
| Hit | Hit | Hit | Yes – what we want! |
| Hit | Hit | Miss | Yes – although the page table is not checked if the TLB hits |
| Miss | Hit | Hit | Yes – TLB miss, PA in page table |
| Miss | Hit | Miss | Yes – TLB miss, PA in page table, but data not in cache |
| Miss | Miss | Miss | Yes – page fault |
| Hit | Miss | Miss/ Hit | Impossible – TLB translation not possible if page is not present in memory |
| Miss | Miss | Hit | Impossible – data not allowed in cache if page is not in memory |

## Reducing Translation Time

- Can **overlap** the cache access with the TLB access
  - Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache



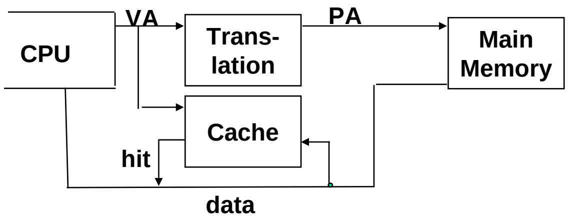
## A TLB in the Memory Hierarchy



- **page fault** : page is not in physical memory
- **TLB misses** are much more frequent than true page faults

## Why Not a **Virtually Addressed Cache?**

- **A virtually addressed cache** would only require address translation on cache misses



but

- Two different virtual addresses can map to the same physical address (when processes are sharing data),
- Two different cache entries hold data for the same physical address – **synonyms**（别名）
  - Must update all cache entries with the same physical address or the memory becomes inconsistent

## The Hardware/Software Boundary

- What parts of the virtual to physical address translation is done by or assisted by the hardware?
  - **Translation Lookaside Buffer (TLB)** that caches the recent translations
    - TLB access time is part of the cache hit time
    - May cause an extra stage in the pipeline for TLB access
  - Page table storage, fault detection and updating
    - **Page faults** result in **interrupts (precise)** that are then handled by the **OS**
    - Hardware must support (i.e., update appropriately) **Dirty** and **Reference bits (e.g., ~LRU)** in the Page Tables

## Summary

- The Principle of Locality:
  - Program likely to access a relatively small portion of the address space at any instant of time.
    - **Temporal Locality**: Locality in Time
    - **Spatial Locality**: Locality in Space
- Caches, TLBs, Virtual Memory all understood by examining how they deal with the four questions
  1. Where can block be placed?
  2. How is block found?
  3. What block is replaced on miss?
  4. How are writes handled?
- **Page tables** map virtual address to physical address
  - **TLBs** are important for fast translation

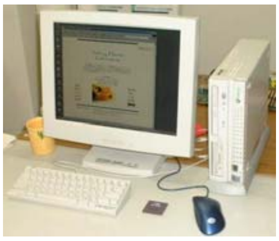
2009-07-16

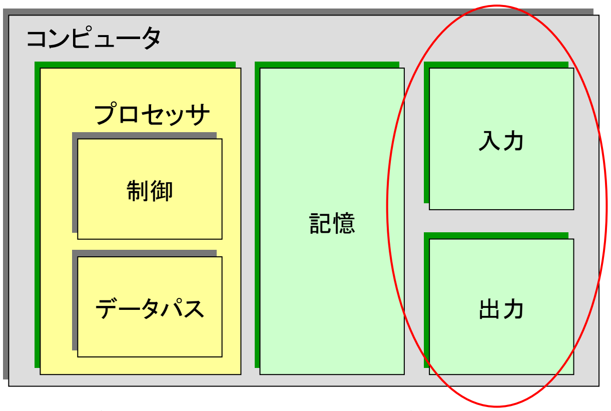
# 計算機アーキテクチャ 第一 (E)

## 11. 入出力制御, 割り込み

吉瀬 謙二 計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室 木曜日13:20 − 14:50

## コンピュータ（ハードウェア）の古典的な要素

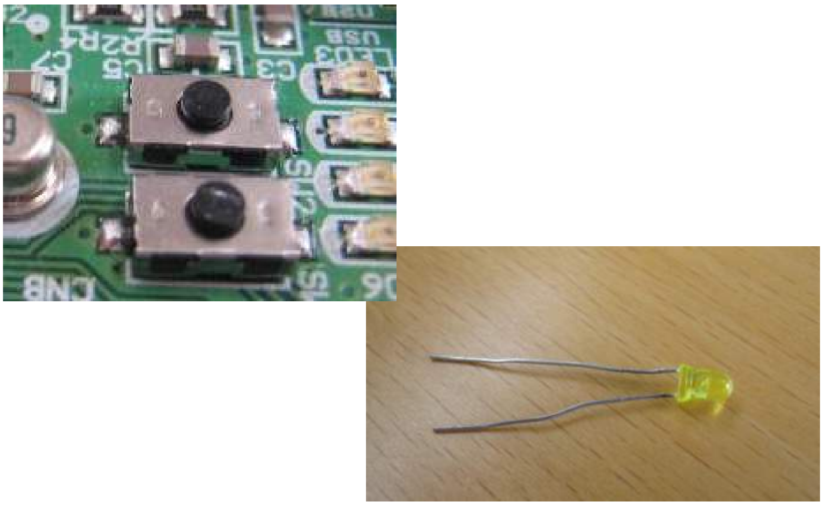




プロセッサは記憶装置から命令とデータを取り出す。入力装置はデータを記憶装置に書き込む。出力装置は記憶装置からデータを読みだす。制御装置は、データパス、記憶装置、入力装置、そして出力装置の動作を指定する信号を送る。

出典：パターソン ＆ ヘネシー、コンピュータの構成と設計

## Input and Output Devices（入出力装置）



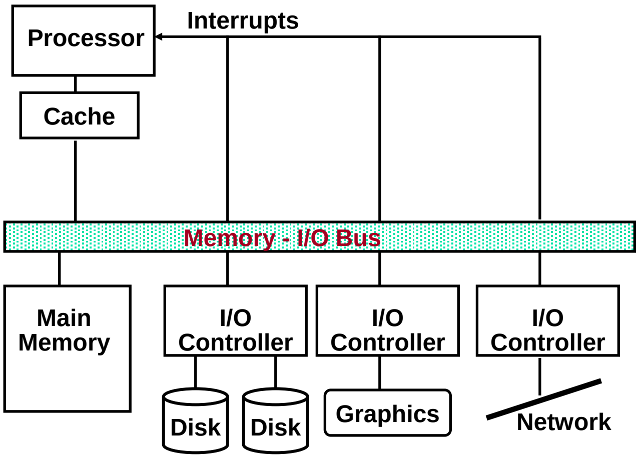
## Input and Output Devices（入出力装置）

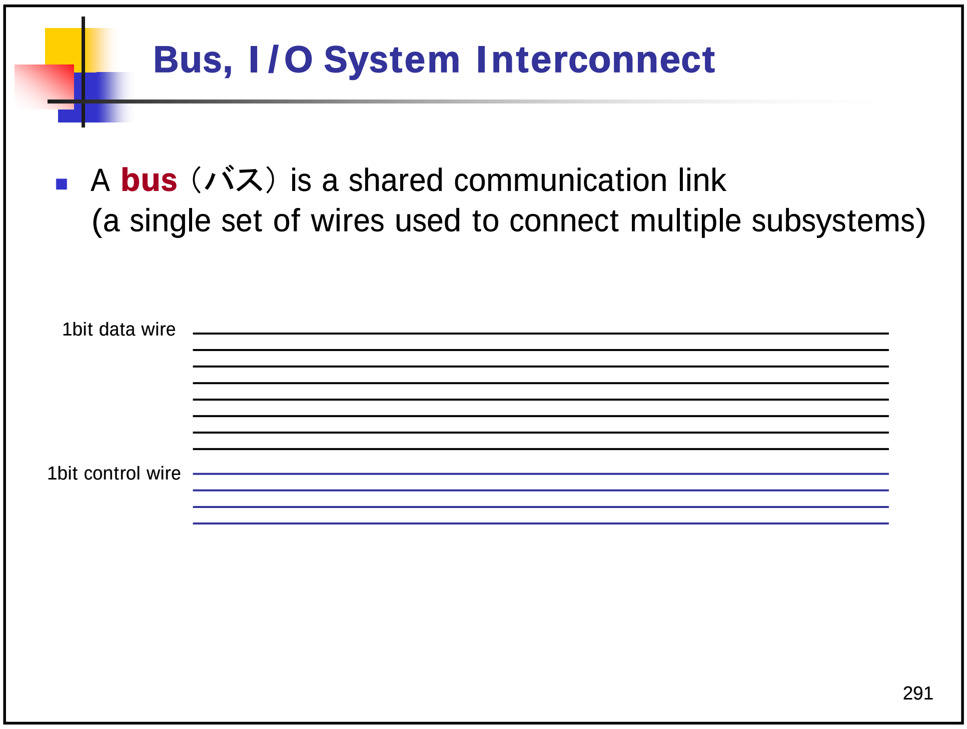
- I/O devices are **diverse** with respect to
  - **Behavior**（動作）– input, output or storage
  - **Partner**（相手）– human or machine
  - **Data rate**（転送速度）– the peak rate at which data can be transferred between the I/O device and the main memory or CPU

| Device | Behavior | Partner | Data rate (Mb/s) |
|---|---|---|---|
| Keyboard | input | human | 0.0001 |
| Mouse | input | human | 0.0038 |
| Laser printer | output | human | 3.2000 |
| Graphics display | output | human | 800.0000-8000.0000 |
| Network/LAN | input or output | machine | 100.0000-1000.0000 |
| Magnetic disk | storage | machine | 240.0000-2560.0000 |

8 orders of magnitude range

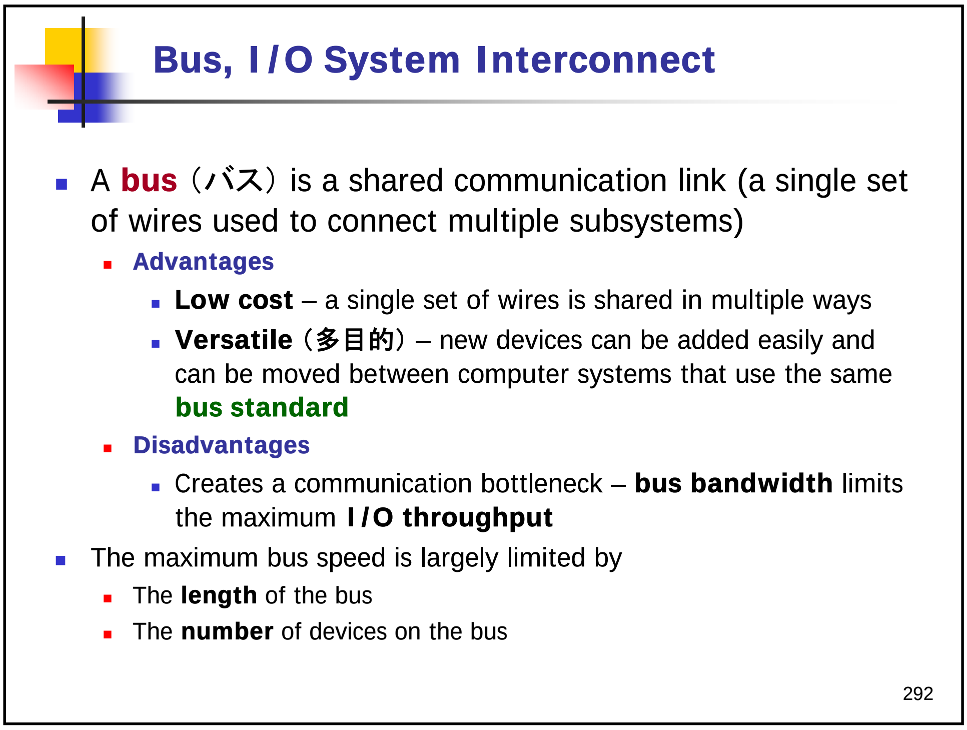
## A Typical I/O System（代表的な入出力装置）





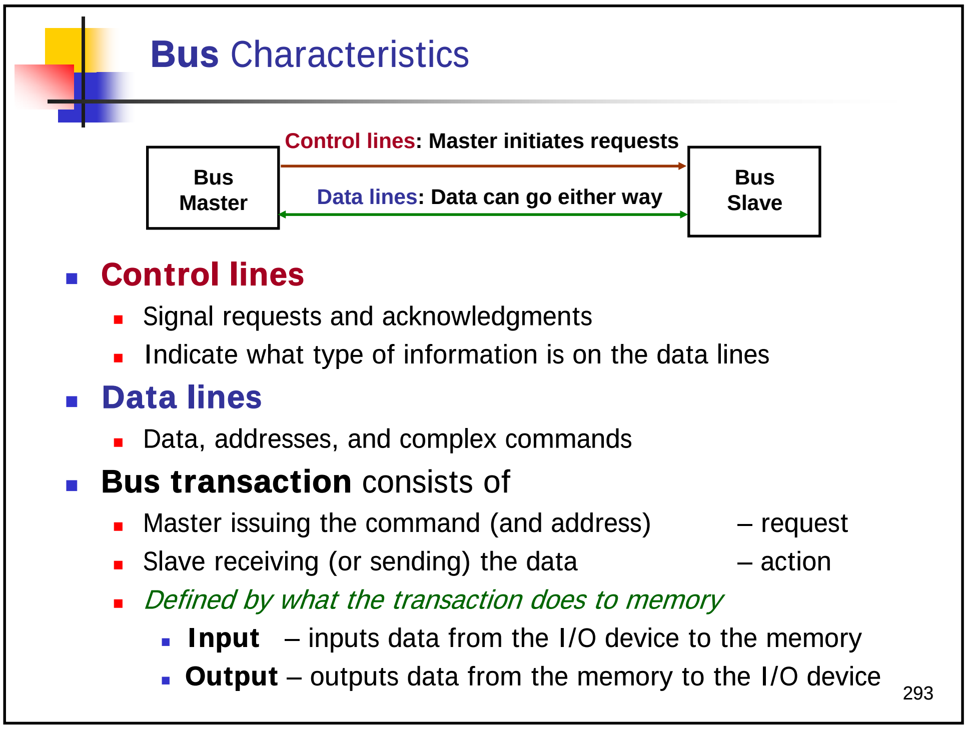
## Bus, I/O System Interconnect

- A **bus**（バス）is a shared communication link
  (a single set of wires used to connect multiple subsystems)



## Bus, I/O System Interconnect

- A **bus**（バス）is a shared communication link (a single set of wires used to connect multiple subsystems)
  - **Advantages**
    - **Low cost** – a single set of wires is shared in multiple ways
    - **Versatile**（多目的）– new devices can be added easily and can be moved between computer systems that use the same **bus standard**
  - **Disadvantages**
    - Creates a communication bottleneck – **bus bandwidth** limits the maximum **I/O throughput**
- The maximum bus speed is largely limited by
  - The **length** of the bus
  - The **number** of devices on the bus



## Bus Characteristics

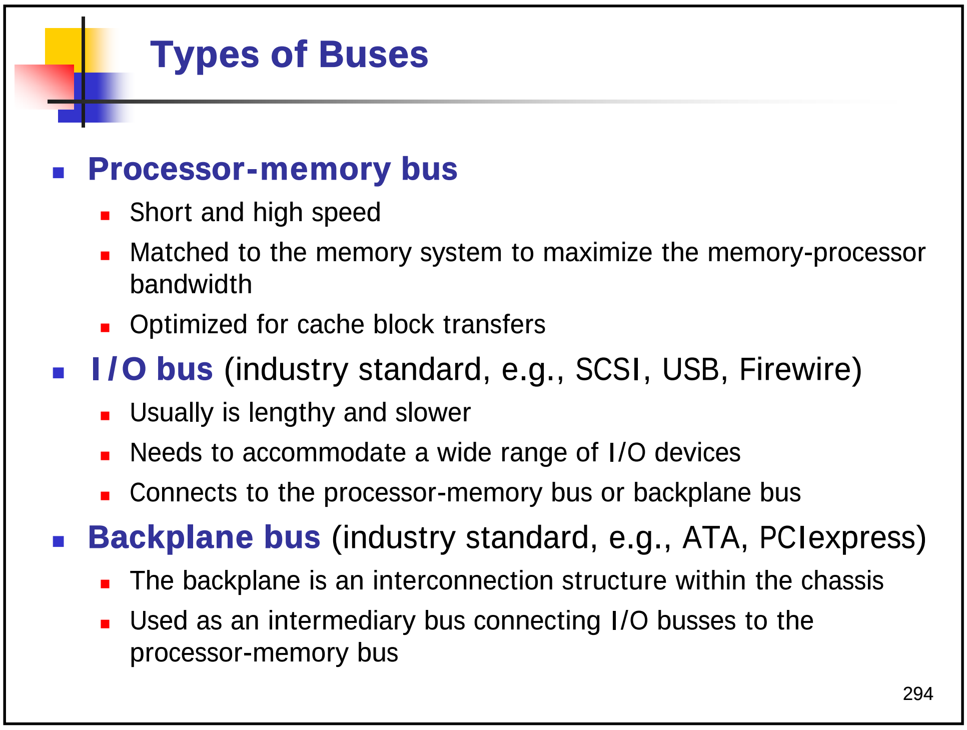
Bus Master

Control lines: Master initiates requests

Data lines: Data can go either way

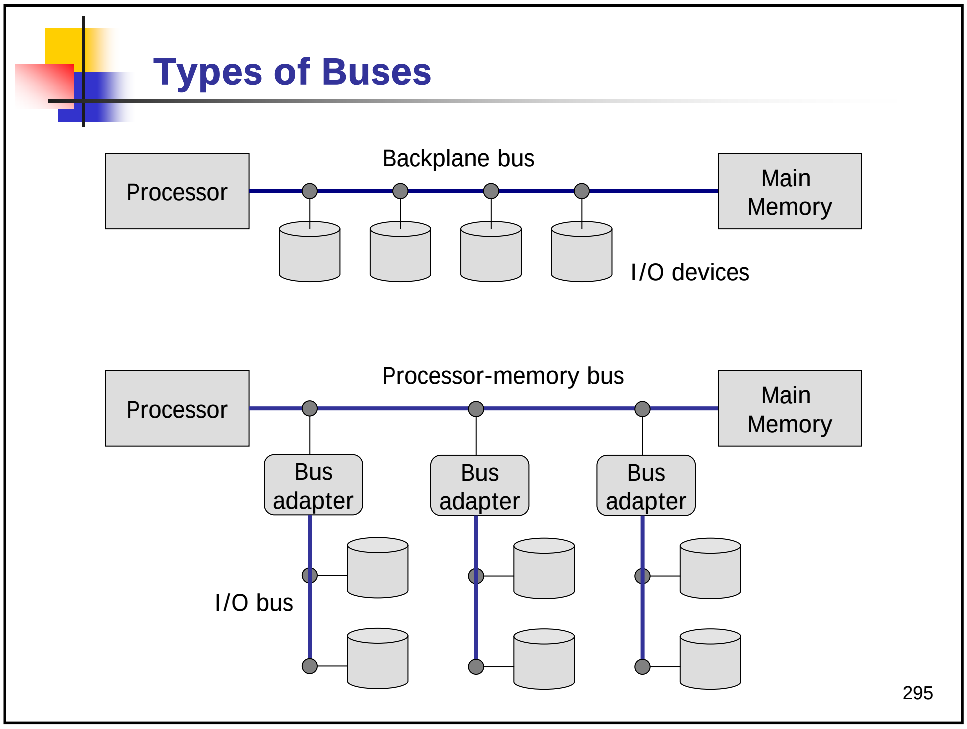
Bus Slave

- **Control lines**
  - Signal requests and acknowledgments
  - Indicate what type of information is on the data lines
- **Data lines**
  - Data, addresses, and complex commands
- **Bus transaction** consists of
  - Master issuing the command (and address) – request
  - Slave receiving (or sending) the data – action
  - *Defined by what the transaction does to memory*
    - **Input** – inputs data from the I/O device to the memory
    - **Output** – outputs data from the memory to the I/O device



## Types of Buses

- **Processor-memory bus**
  - Short and high speed
  - Matched to the memory system to maximize the memory-processor bandwidth
  - Optimized for cache block transfers
- **I/O bus** (industry standard, e.g., SCSI, USB, Firewire)
  - Usually is lengthy and slower
  - Needs to accommodate a wide range of I/O devices
  - Connects to the processor-memory bus or backplane bus
- **Backplane bus** (industry standard, e.g., ATA, PCIexpress)
  - The backplane is an interconnection structure within the chassis
  - Used as an intermediary bus connecting I/O busses to the processor-memory bus



# Types of Buses

Processor

Backplane bus

Main Memory

I/O devices

Processor

Processor-memory bus

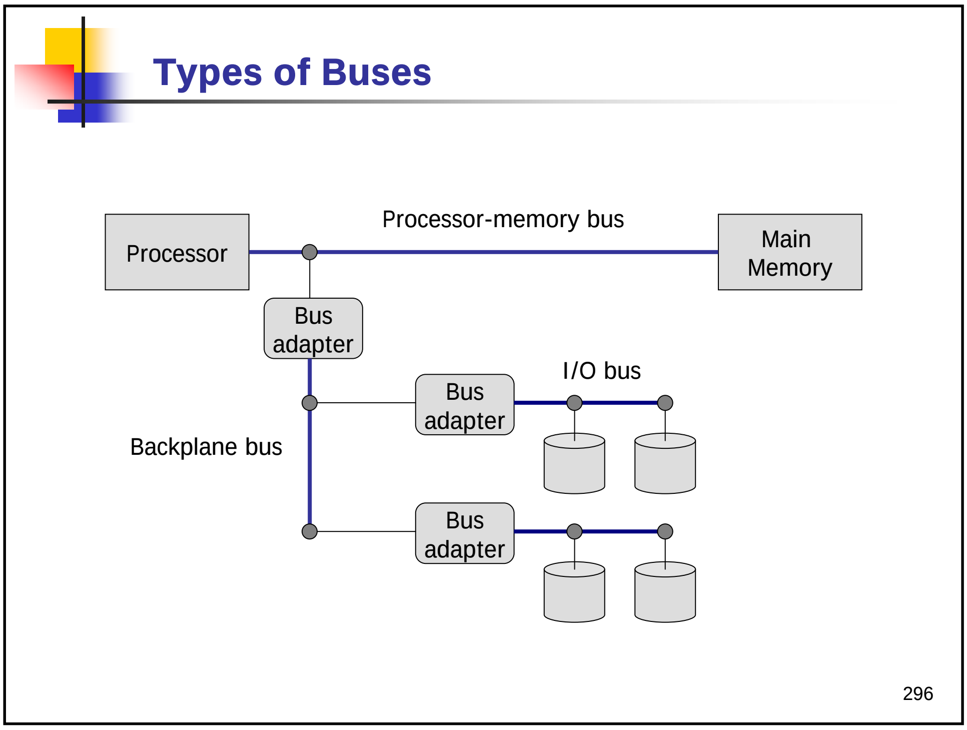
Main Memory

Bus adapter

Bus adapter

Bus adapter

I/O bus



# Types of Buses

Processor

Processor-memory bus

Main Memory

Bus adapter

I/O bus

Bus adapter

Backplane bus

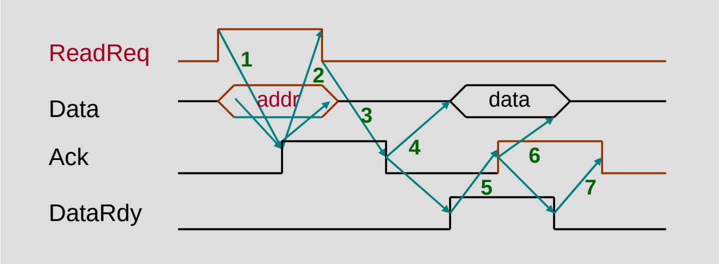
Bus adapter

## Synchronous(同期式), Asynchronous(非同期式) Buses

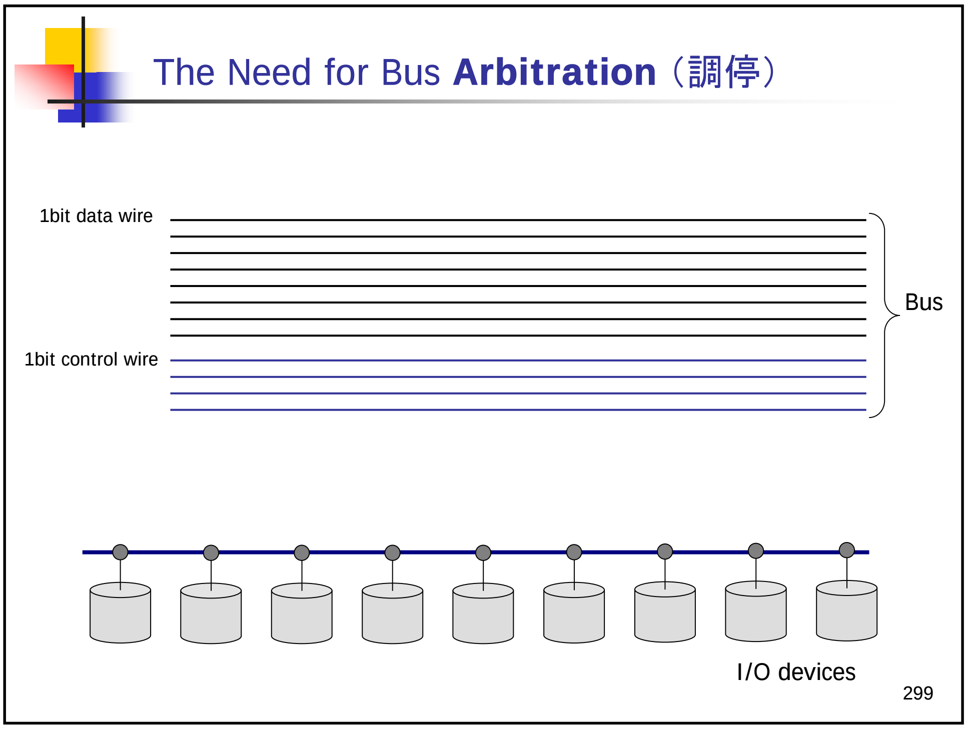
- **Synchronous bus** (e.g., processor-memory buses)
  - Includes a clock in the control lines and has a fixed protocol for communication that is **relative** to the clock
  - **Advantage**: involves very little logic and can run very fast
  - **Disadvantages**:
    - Every device communicating on the bus must use same clock rate
    - To avoid **clock skew**, they cannot be long if they are fast
- **Asynchronous bus** (e.g., I/O buses)
  - It is not clocked, so requires a **handshaking protocol** and additional control lines (**ReadReq, Ack, DataRdy**)
  - **Advantages**:
    - Can accommodate a wide range of devices and device speeds
    - Can be lengthened without worrying about clock skew or synchronization problems
  - **Disadvantage**: slow

## Asynchronous Bus **Handshaking Protocol**

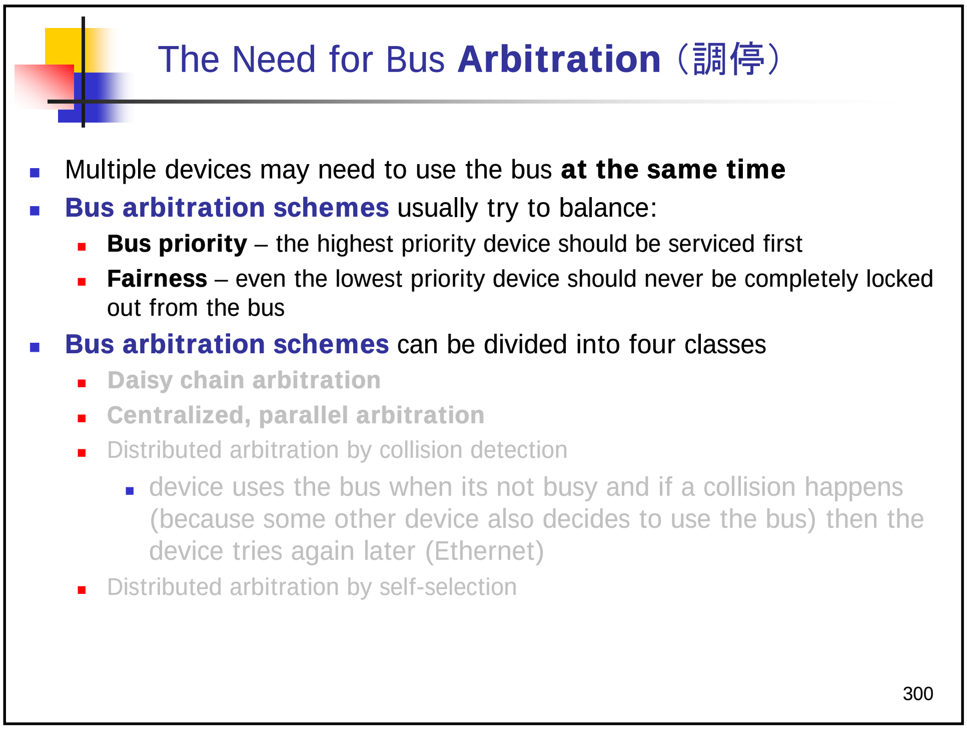
Output (read) data from memory to an I/O device



1. Memory sees **ReadReq**, reads **addr** from data lines, and raises **Ack**
2. I/O device sees **Ack** and releases the **ReadReq** and data lines
3. Memory sees **ReadReq** go low and drops Ack
4. When memory has data ready, it places it on data lines and raises **DataRdy**
5. I/O device sees **DataRdy**, reads the data from data lines, and raises **Ack**
6. Memory sees **Ack**, releases the data lines, and drops **DataRdy**
7. I/O device sees **DataRdy** go low and drops **Ack**



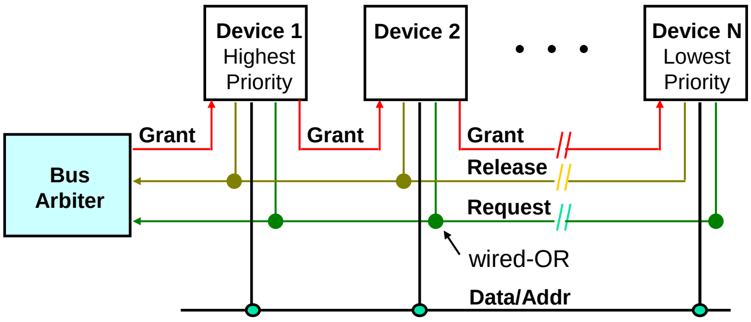
## The Need for Bus **Arbitration**（調停）



## The Need for Bus **Arbitration**（調停）

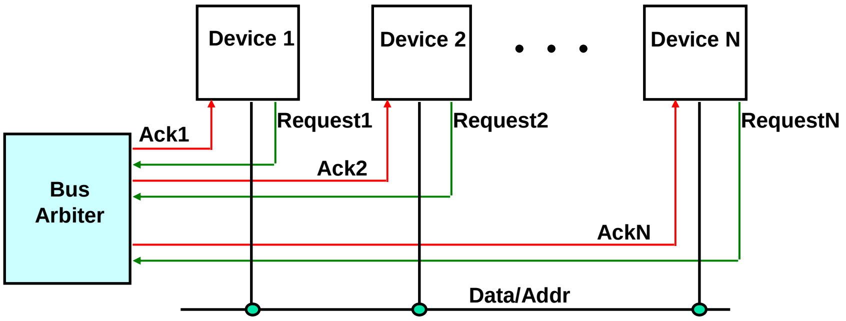
- Multiple devices may need to use the bus **at the same time**
- **Bus arbitration schemes** usually try to balance:
  - **Bus priority** – the highest priority device should be serviced first
  - **Fairness** – even the lowest priority device should never be completely locked out from the bus
- **Bus arbitration schemes** can be divided into four classes
  - **Daisy chain arbitration**
  - **Centralized, parallel arbitration**
  - Distributed arbitration by collision detection
    - device uses the bus when its not busy and if a collision happens (because some other device also decides to use the bus) then the device tries again later (Ethernet)
  - Distributed arbitration by self-selection

## Daisy Chain Bus Arbitration（デイジーチェイン方式）

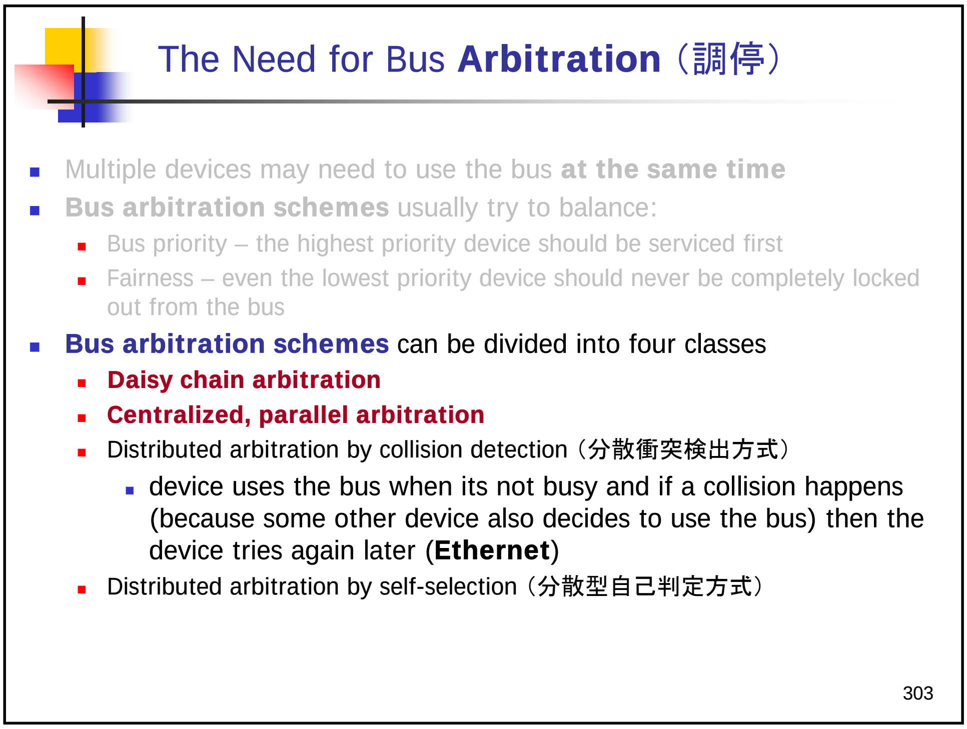

Device 1
Highest
Priority
Device 2
• • •
Device N
Lowest
Priority
Bus
Arbiter
Grant
Grant
Grant
Release
Request
wired-OR
Data/Addr

- **Advantage:** simple
- **Disadvantages**:
  - Cannot assure fairness – a low-priority device may be locked out
  - Slower – the daisy chain grant signal limits the bus speed

## Centralized Parallel Arbitration（集中並列方式）

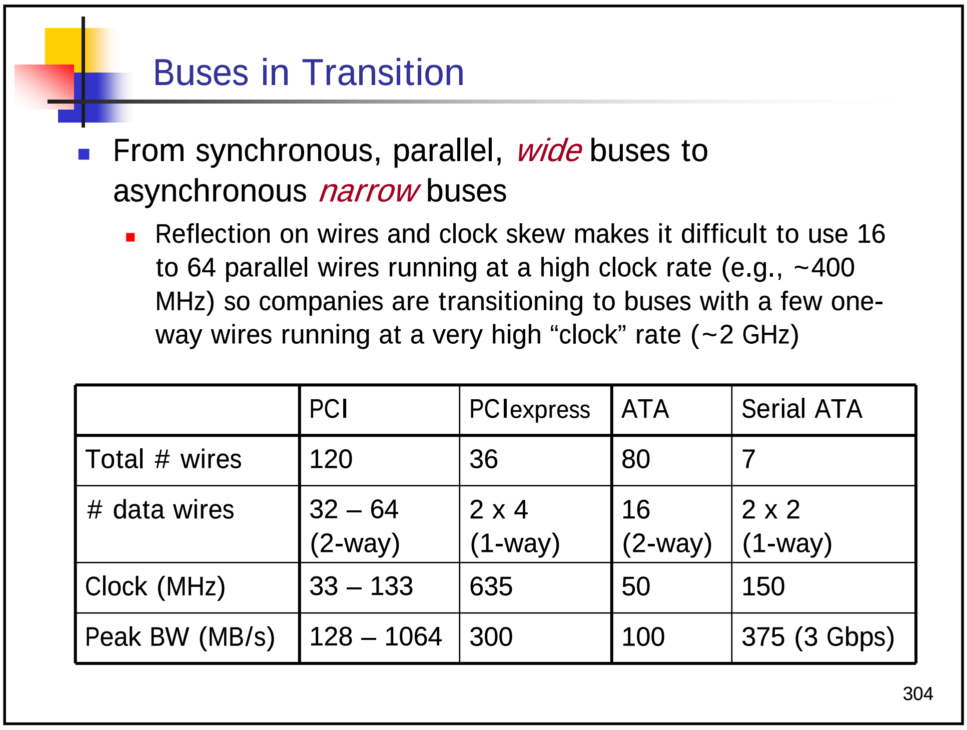

Device 1
Device 2
• • •
Device N
Bus
Arbiter
Ack1
Request1
Ack2
Request2
RequestN
AckN
Data/Addr

- **Advantages**: flexible, can assure fairness
- **Disadvantages**: more complicated arbiter hardware
- Used in essentially all processor-memory buses and in high-speed I/O buses



## The Need for Bus **Arbitration**（調停）

- Multiple devices may need to use the bus **at the same time**
- **Bus arbitration schemes** usually try to balance:
  - Bus priority – the highest priority device should be serviced first
  - Fairness – even the lowest priority device should never be completely locked out from the bus
- **Bus arbitration schemes** can be divided into four classes
  - **Daisy chain arbitration**
  - **Centralized, parallel arbitration**
  - Distributed arbitration by collision detection（分散衝突検出方式）
    - device uses the bus when its not busy and if a collision happens (because some other device also decides to use the bus) then the device tries again later (**Ethernet**)
  - Distributed arbitration by self-selection（分散型自己判定方式）



## Buses in Transition

- From synchronous, parallel, *wide* buses to asynchronous *narrow* buses
  - Reflection on wires and clock skew makes it difficult to use 16 to 64 parallel wires running at a high clock rate (e.g., ~400 MHz) so companies are transitioning to buses with a few one-way wires running at a very high “clock” rate (~2 GHz)

| | PCI | PCIexpress | ATA | Serial ATA |
|---|---|---|---|---|
| Total # wires | 120 | 36 | 80 | 7 |
| # data wires | 32 – 64<br>(2-way) | 2 x 4<br>(1-way) | 16<br>(2-way) | 2 x 2<br>(1-way) |
| Clock (MHz) | 33 – 133 | 635 | 50 | 150 |
| Peak BW (MB/s) | 128 – 1064 | 300 | 100 | 375 (3 Gbps) |

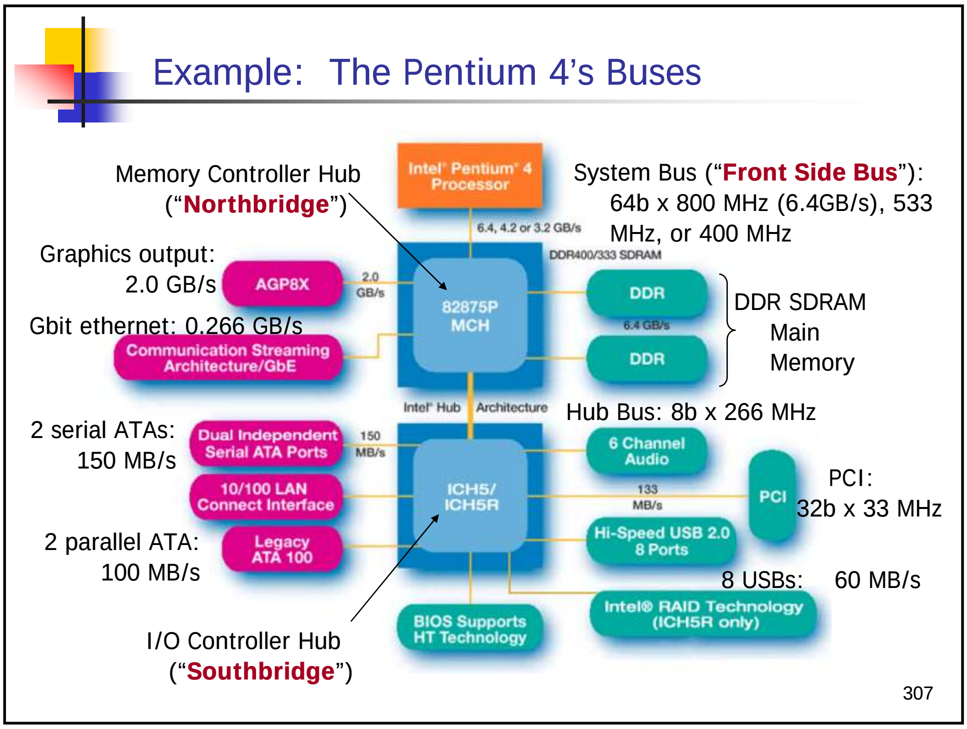
## ATA Cable Sizes

- **Serial ATA** cables (**red**) are much thinner than **parallel ATA** cables (**green**)

## Bus Bandwidth Determinates

- The bandwidth of a bus is determined by
  - Whether its is synchronous or asynchronous and the timing characteristics of the protocol used
  - The data bus width
  - Whether the bus supports block transfers or only word transfers

| | Firewire | USB 2.0 |
|---|---|---|
| Type | I/O | I/O |
| Data lines | 4 | 2 |
| Clocking | Asynchronous | Synchronous |
| Max # devices | 63 | 127 |
| Max length | 4.5 meters | 5 meters |
| Peak bandwidth | 50 MB/s (400 Mbps)<br>100 MB/s (800 Mbps) | 0.2 MB/s (low)<br>1.5 MB/s (full)<br>60 MB/s (high) |



## Example: The Pentium 4's Buses

Memory Controller Hub (“**Northbridge**”)

Graphics output: 2.0 GB/s

Gbit ethernet: 0.266 GB/s

2 serial ATAs: 150 MB/s

2 parallel ATA: 100 MB/s

I/O Controller Hub (“**Southbridge**”)

System Bus (“**Front Side Bus**”): 64b x 800 MHz (6.4GB/s), 533 MHz, or 400 MHz

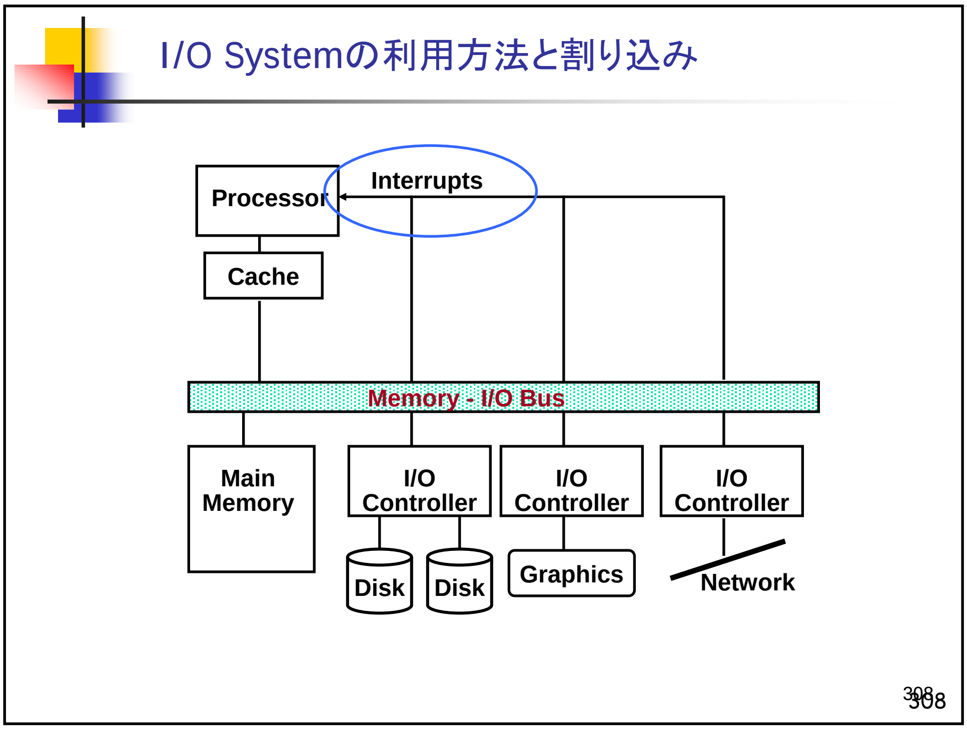
DDR SDRAM Main Memory

Hub Bus: 8b x 266 MHz

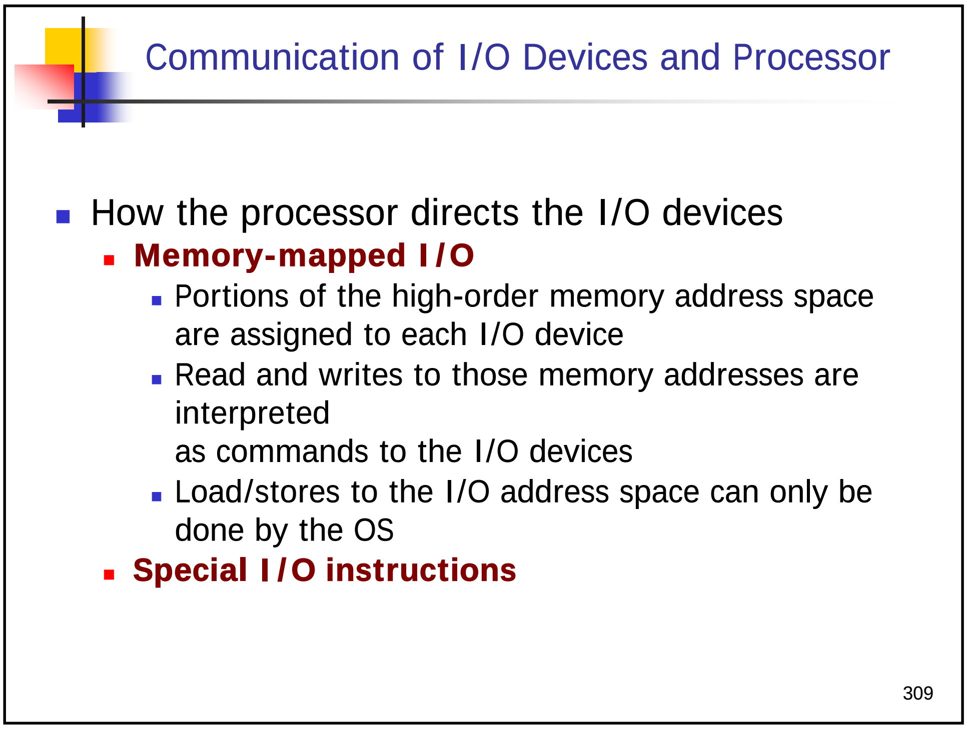
PCI: 32b x 33 MHz

8 USBs: 60 MB/s

Intel® Pentium® 4 Processor
6.4, 4.2 or 3.2 GB/s
DDR400/333 SDRAM
AGP8X
2.0 GB/s
82875P MCH
DDR
6.4 GB/s
DDR
Communication Streaming Architecture/GbE
Intel® Hub Architecture
Dual Independent Serial ATA Ports
150 MB/s
10/100 LAN Connect Interface
Legacy ATA 100
ICH5/ ICH5R
6 Channel Audio
133 MB/s
PCI
Hi-Speed USB 2.0 8 Ports
BIOS Supports HT Technology
Intel® RAID Technology (ICH5R only)

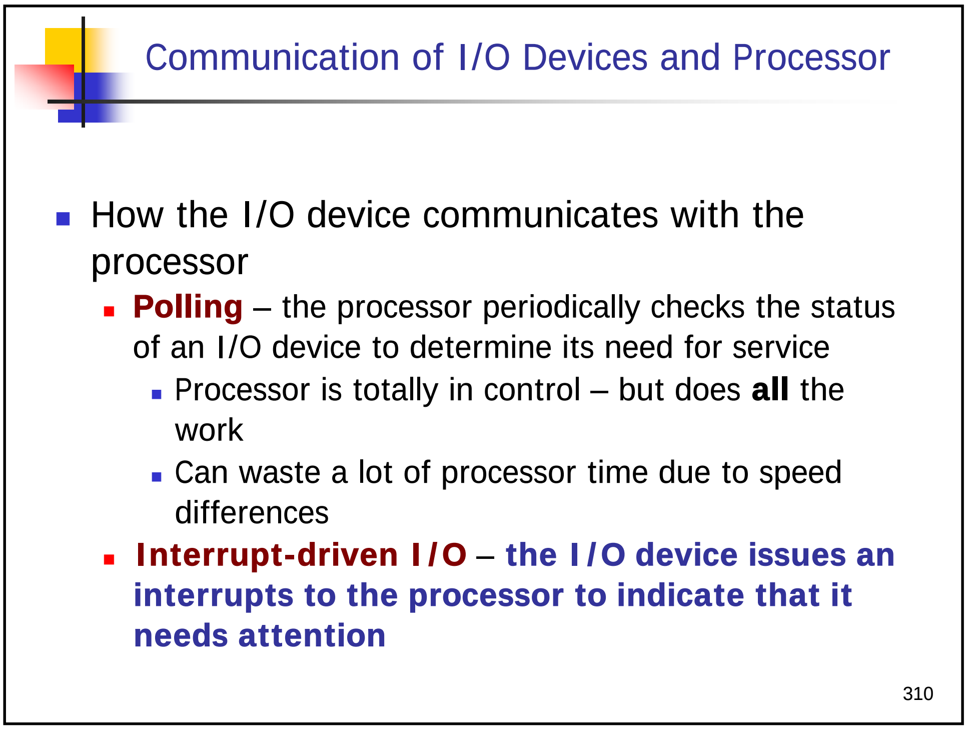


## I/O Systemの利用方法と割り込み



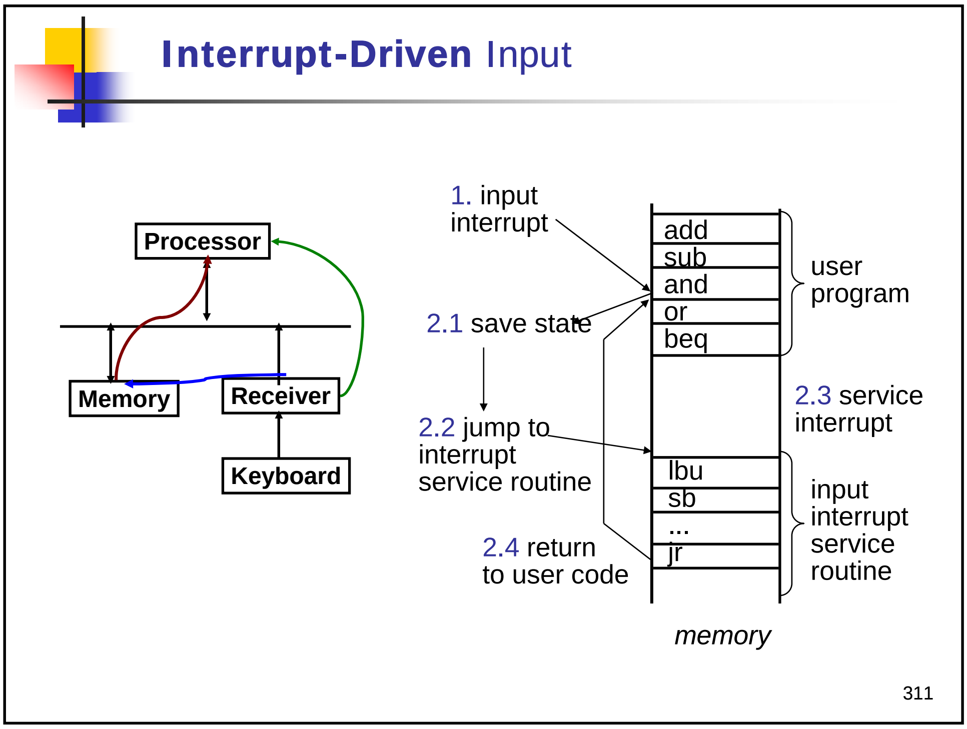
## Communication of I/O Devices and Processor

- How the processor directs the I/O devices
  - **Memory-mapped I/O**
    - Portions of the high-order memory address space are assigned to each I/O device
    - Read and writes to those memory addresses are interpreted
      as commands to the I/O devices
    - Load/stores to the I/O address space can only be done by the OS
  - **Special I/O instructions**



## Communication of I/O Devices and Processor

- How the I/O device communicates with the processor
  - **Polling** – the processor periodically checks the status of an I/O device to determine its need for service
    - Processor is totally in control – but does **all** the work
    - Can waste a lot of processor time due to speed differences
  - **Interrupt-driven I/O** – **the I/O device issues an interrupts to the processor to indicate that it needs attention**



## Interrupt-Driven Input

Processor

Memory

Receiver

Keyboard

1. input interrupt

2.1 save state

2.2 jump to interrupt service routine

2.3 service interrupt

2.4 return to user code

add

sub

and

or

beq

lbu

sb

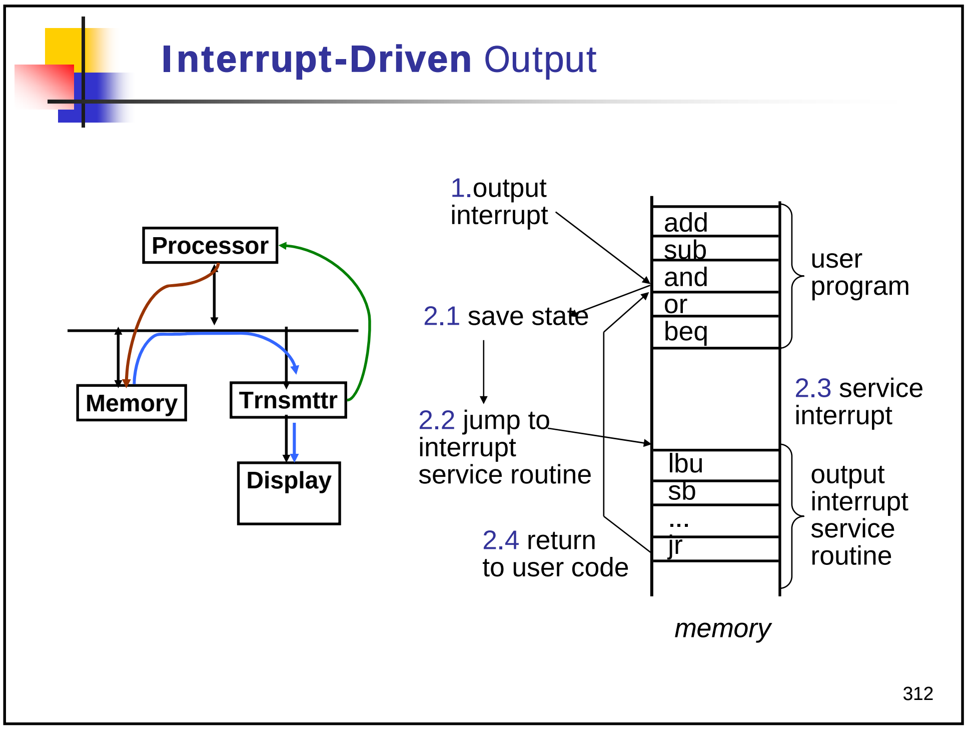
...

jr

user program

input interrupt service routine

*memory*



## Interrupt-Driven Output

Processor

Memory

Trnsmttr

Display

1.output interrupt

2.1 save state

2.2 jump to interrupt service routine

2.3 service interrupt

2.4 return to user code

add

sub

and

or

beq

lbu

sb

...

jr

user program

output interrupt service routine

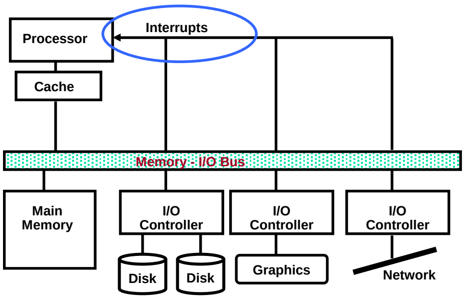
*memory*

## Interrupt-Driven I/O

- An I/O interrupt is **asynchronous**
  - Is not associated with any instruction so doesn't prevent any instruction from completing
    - You can pick your own convenient point to handle the interrupt
- With I/O interrupts
  - Need a way to identify the device generating the interrupt
  - Can have different urgencies (so may need to be prioritized)
- **Advantages** of using interrupts
  - No need to continuously poll for an I/O event; user program progress is only suspended during the actual transfer of I/O data to/from user memory space
- **Disadvantage** – special hardware is needed to
  - Cause an interrupt (I/O device) and detect an interrupt and save the necessary information to resume normal processing after servicing the interrupt (processor)

## Direct Memory Access (DMA)

- For high-bandwidth devices (like disks) **interrupt-driven I/O** would consume a *lot* of processor cycles
- **DMA** – the I/O controller has the ability to transfer data **directly** to/from the memory without involving the processor
- There may be multiple DMA devices in one system



## Direct Memory Access (DMA) how to?

1. The processor initiates the DMA transfer by supplying the I/O device address, the operation to be performed, the memory address destination/source, the number of bytes to transfer
2. The I/O DMA controller manages the entire transfer (possibly thousand of bytes in length), arbitrating for the bus
3. When the DMA transfer is complete, the I/O controller interrupts the processor to let it know that the transfer is complete

## I/O and the **Operating System**

- The operating system acts as the interface between the I/O hardware and the program requesting I/O
  - To protect the **shared I/O resources**, the user program is not allowed to communicate directly with the I/O device
- Thus **OS** must be able to give commands to I/O devices, handle interrupts generated by I/O devices, provide fair access to the shared I/O resources, and schedule I/O requests to enhance system throughput
  - I/O interrupts result in a transfer of processor control to the **supervisor (OS) process**

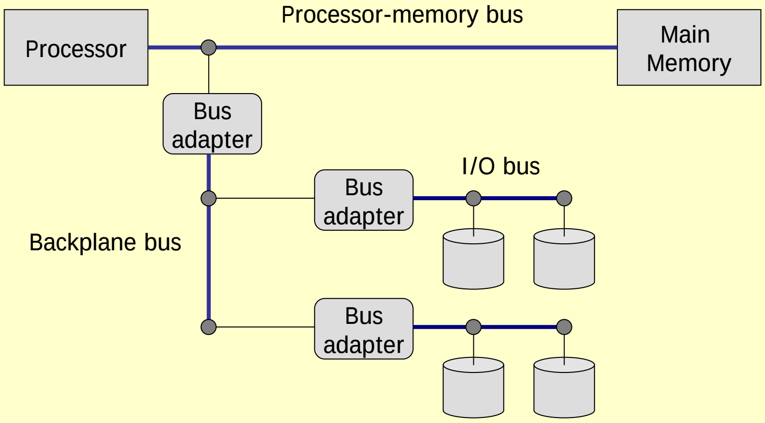
2009-07-23

# 計算機アーキテクチャ 第一 (E)

## １２. ネットワーク, マルチコアプロセッサ

吉瀬 謙二　計算工学専攻
kise_at_cs.titech.ac.jp
W641講義室　木曜日１３:２０ ― １４:５０

## Types of Buses



Processor

Processor-memory bus

Main Memory

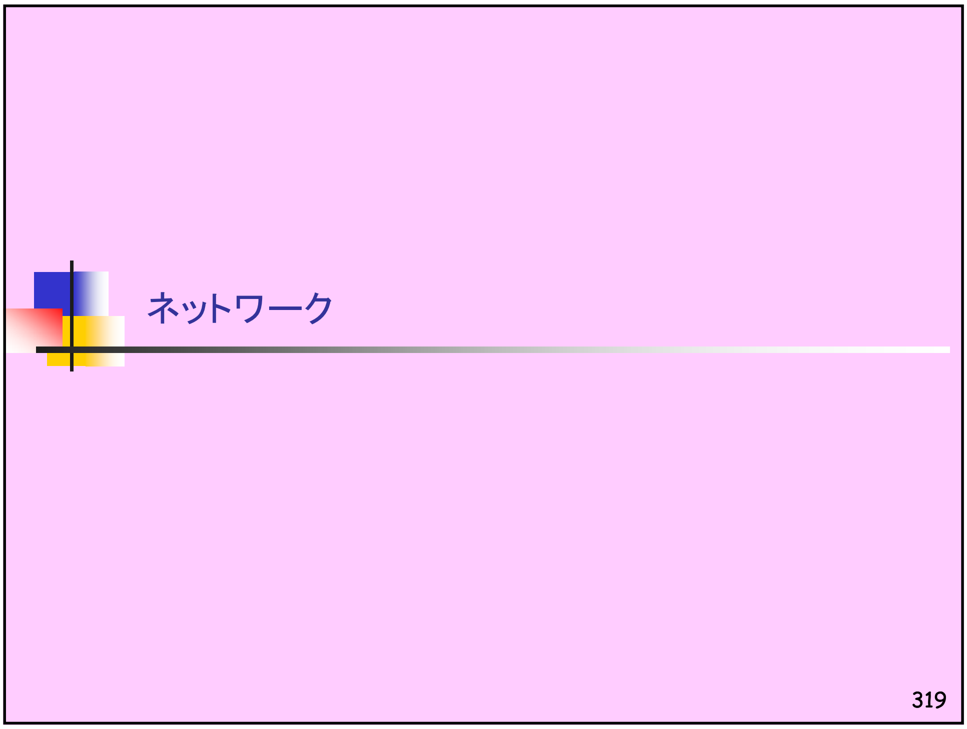
Bus adapter

I/O bus

Bus adapter

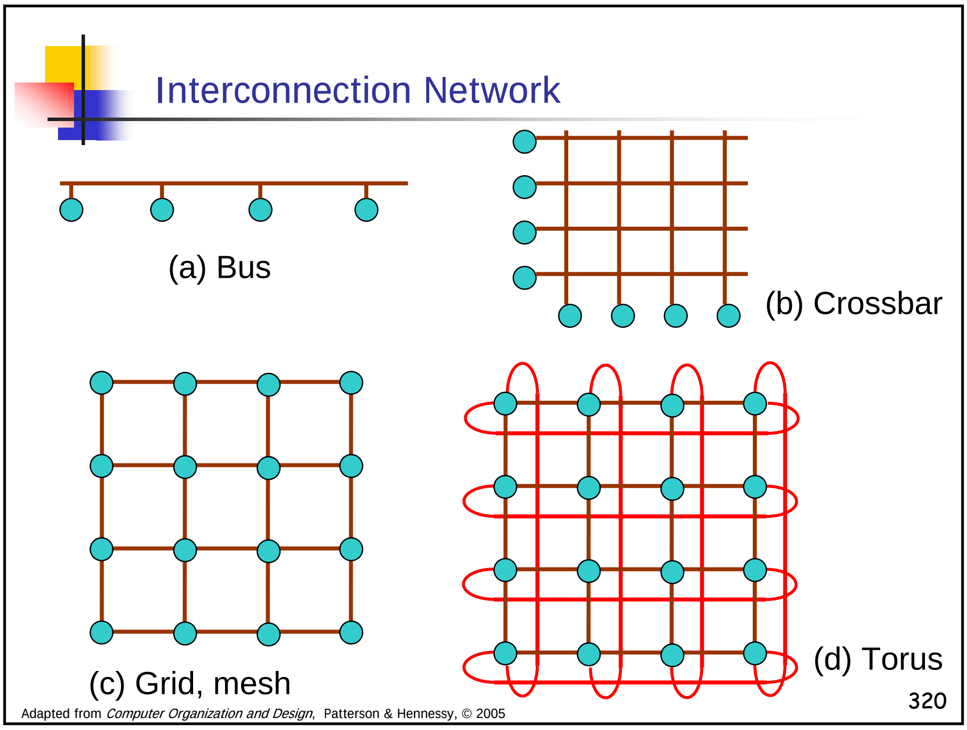
Backplane bus

Bus adapter



## ネットワーク

319



## Interconnection Network

(a) Bus

(b) Crossbar

(c) Grid, mesh

(d) Torus

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

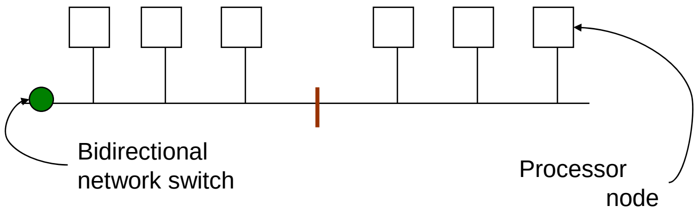
320

## Interconnection Network **Performance Metrics**

- Network cost
  - number of switches
  - number of links on a switch to connect to the network (plus one link to connect to the processor)
  - width in bits per link, length of link
- Network bandwidth (NB)
  – represents the **best** case
  - bandwidth of each link * number of links
- Bisection bandwidth (BB)バイセクションバンド幅
  – represents the **worst** case
  - divide the machine in two parts,
    each with half the nodes and
    sum the bandwidth of the links that cross the dividing line

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Bus Network



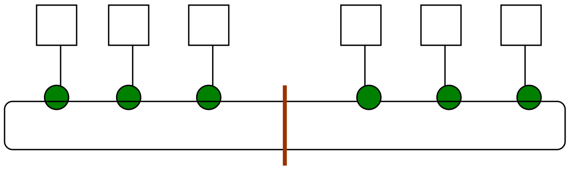
Bidirectional network switch

Processor node

- N processors, 1 switch ( ● ), 1 link (the bus)
- Only 1 simultaneous transfer at a time
  - NB (best case) = link (bus) bandwidth * 1
  - BB (worst case) = link (bus) bandwidth * 1

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

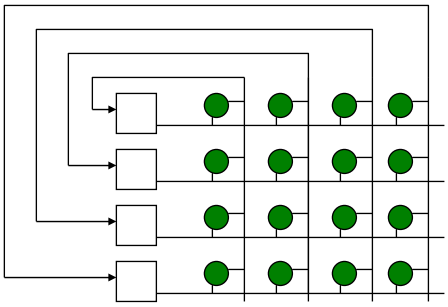
## Ring Network



- N processors, N switches, 2 links/switch, N links
- N simultaneous transfers
  - NB (best case) = link bandwidth * N
  - BB (worst case) = link bandwidth * 2
- If a link is as fast as a bus, the ring is only twice as fast as a bus in the worst case, but is N times faster in the best case

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

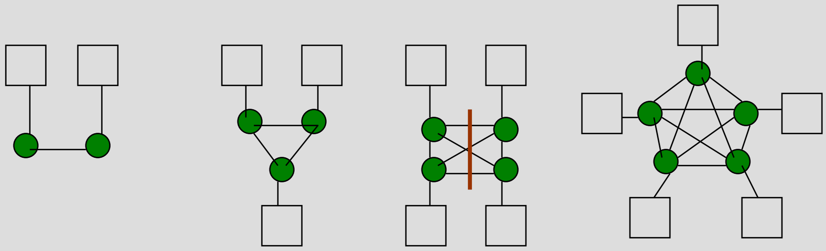
## Crossbar (Xbar) Network



- N processors, $N^2$ switches (unidirectional), 2 links/switch, $N^2$ links
- N simultaneous transfers
  - NB = link bandwidth * N
  - BB = link bandwidth * N/2

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## Fully Connected Network

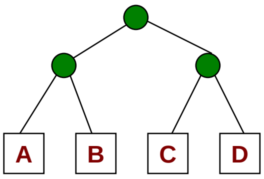


- N processors, N switches, N-1 links/switch, (N*(N-1))/2 links
- N simultaneous transfers
  - NB (best case) = link bandwidth * (N*(N-1))/2
  - BB (worst case) = link bandwidth * $(N/2)^2$

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

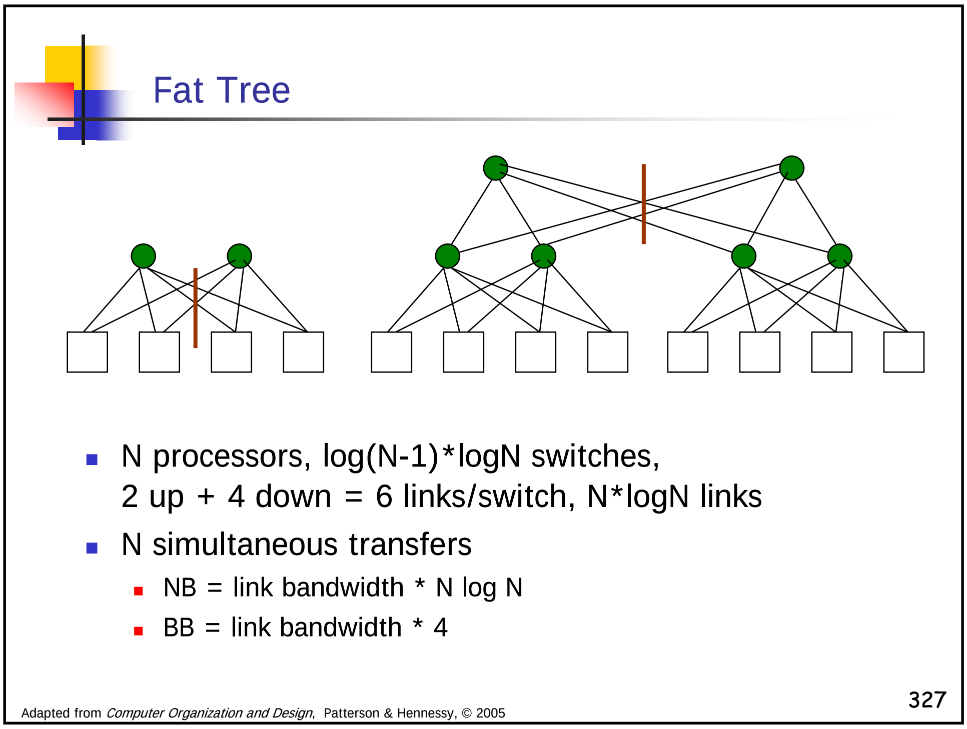
## Fat Tree

- Trees are good structures.
  People in CS (Computer Science) use them all the time.
  Suppose we wanted to make a tree network.



- Any time A wants to send to C, it ties up the upper links, so that B can't send to D.
  - The bisection bandwidth on a tree is horrible - 1 link, at all times
- The solution is to '**thicken**' the upper links.

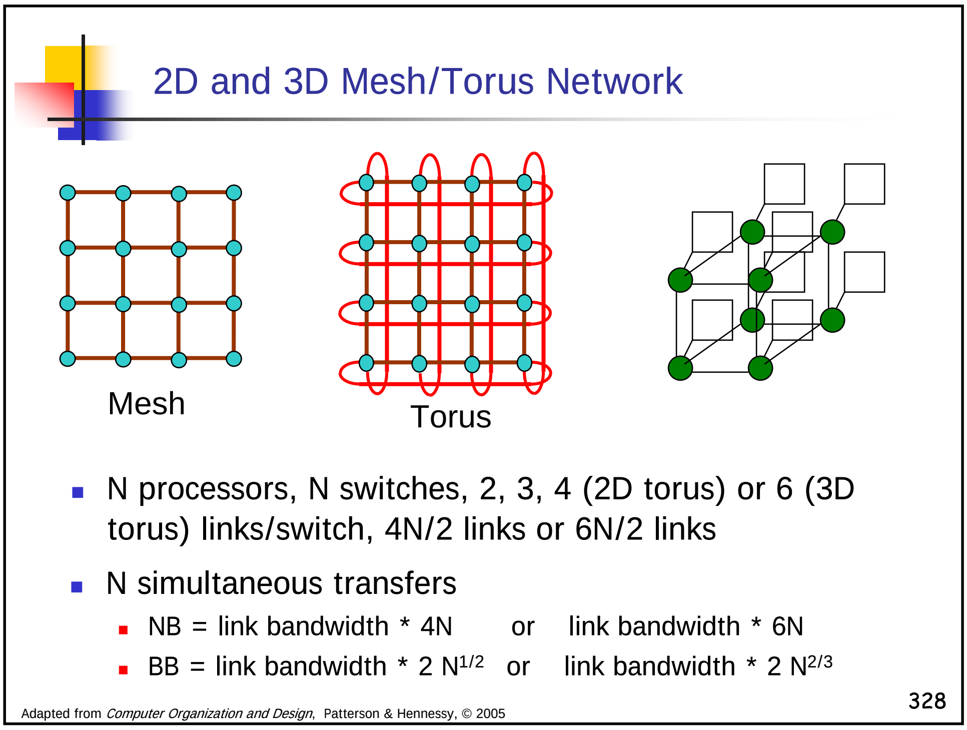
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Fat Tree

- N processors, log(N-1)*logN switches, 2 up + 4 down = 6 links/switch, N*logN links
- N simultaneous transfers
  - NB = link bandwidth * N log N
  - BB = link bandwidth * 4

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



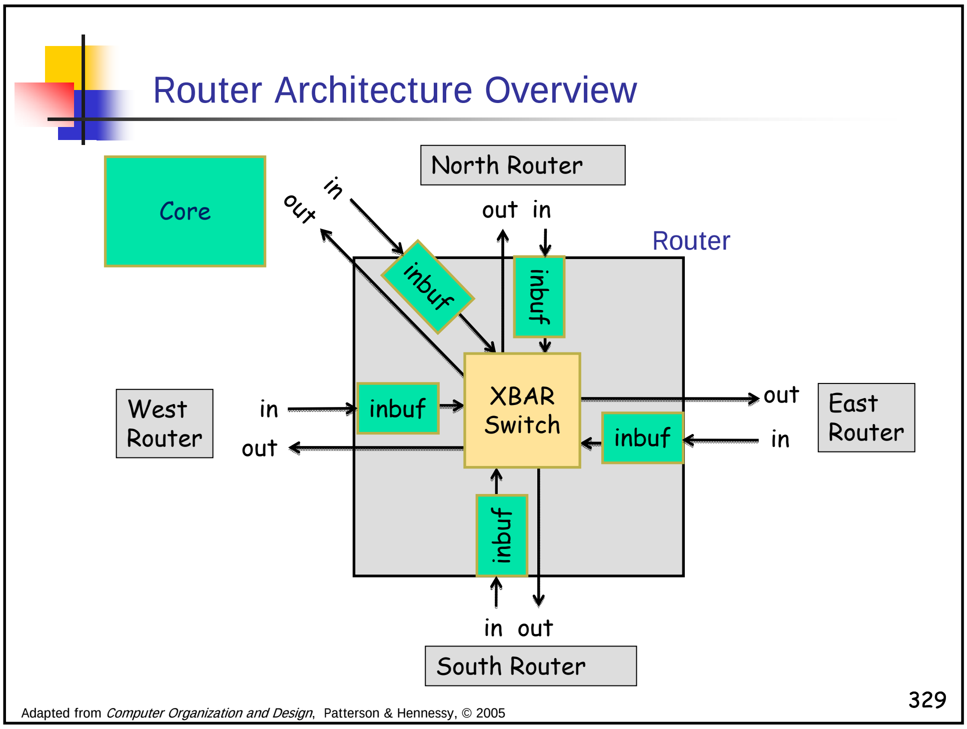
## 2D and 3D Mesh/Torus Network

Mesh

Torus

- N processors, N switches, 2, 3, 4 (2D torus) or 6 (3D torus) links/switch, 4N/2 links or 6N/2 links
- N simultaneous transfers
  - NB = link bandwidth * 4N or link bandwidth * 6N
  - BB = link bandwidth * 2 $N^{1/2}$ or link bandwidth * 2 $N^{2/3}$

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## Router Architecture Overview

Core

in

out

inbuf

North Router

out in

inbuf

Router

West Router

in

out

inbuf

XBAR Switch

inbuf

out

in

East Router

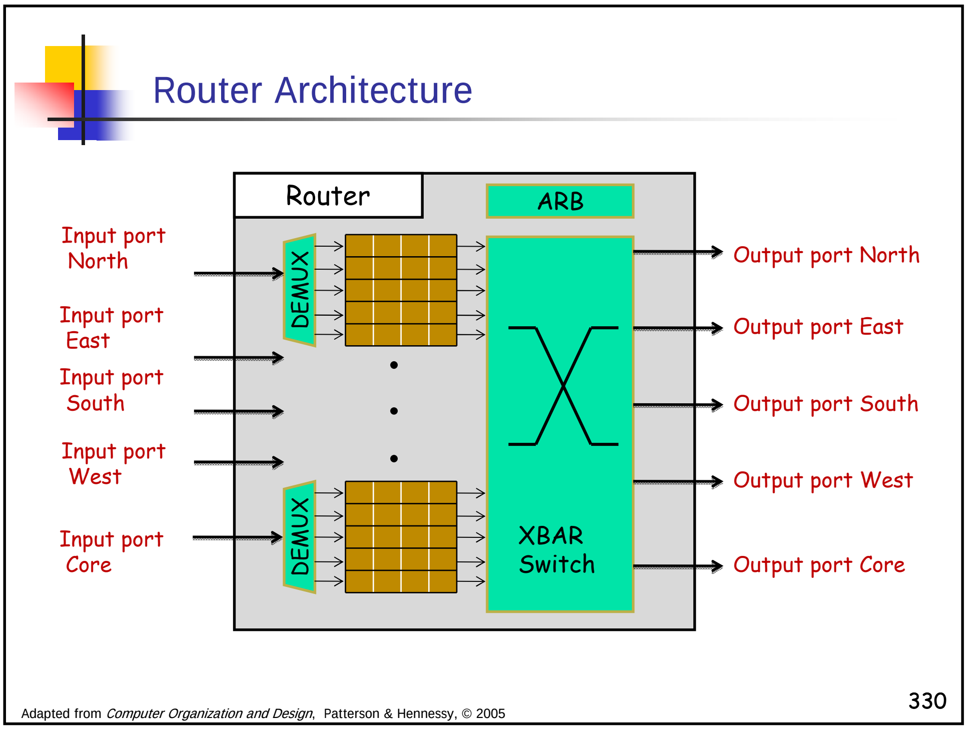
inbuf

in out

South Router

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

329



## Router Architecture

Router

ARB

Input port North

Input port East

Input port South

Input port West

Input port Core

DEMUX

DEMUX

XBAR Switch

Output port North

Output port East

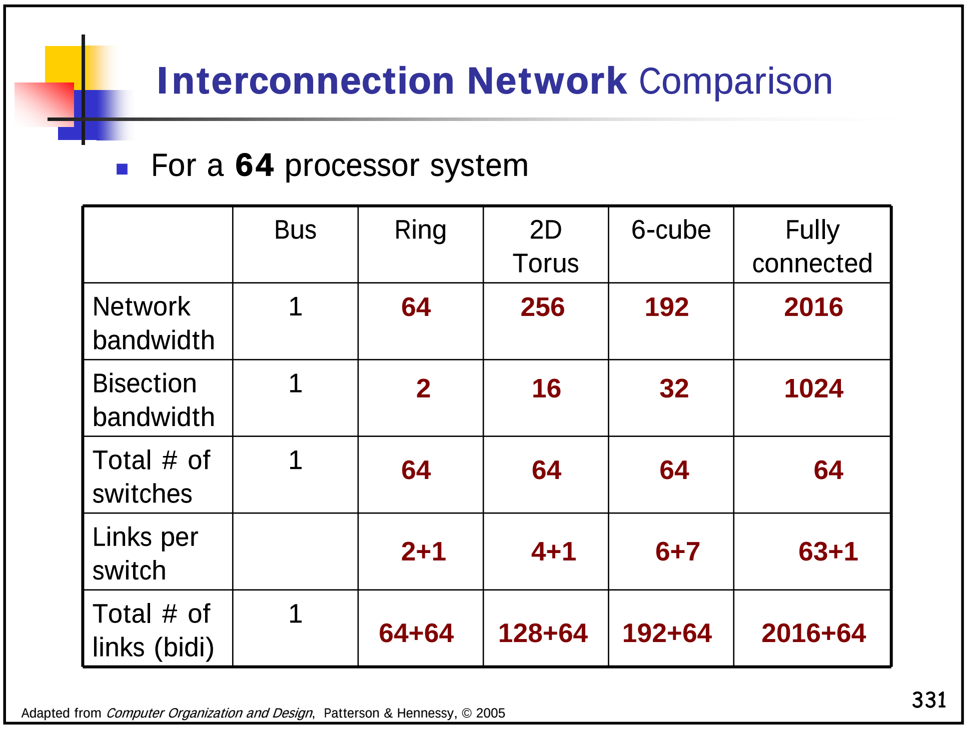
Output port South

Output port West

Output port Core

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

330



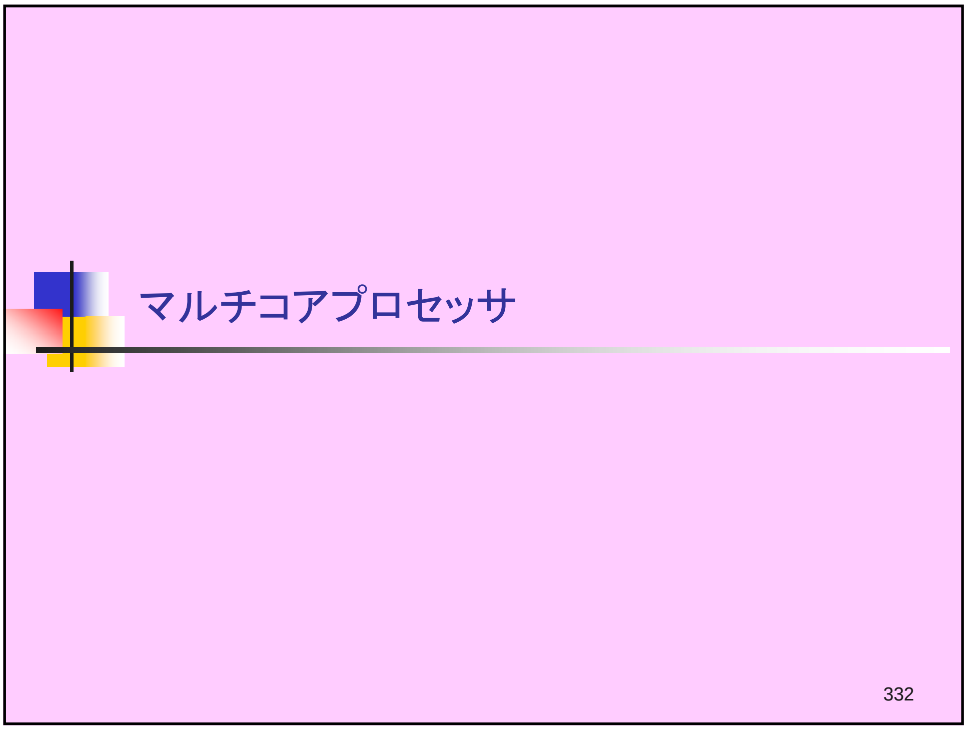
## Interconnection Network Comparison

- For a **64** processor system

| | Bus | Ring | 2D Torus | 6-cube | Fully connected |
|---|---|---|---|---|---|
| Network bandwidth | 1 | **64** | **256** | **192** | **2016** |
| Bisection bandwidth | 1 | **2** | **16** | **32** | **1024** |
| Total # of switches | 1 | **64** | **64** | **64** | **64** |
| Links per switch | | **2+1** | **4+1** | **6+7** | **63+1** |
| Total # of links (bidi) | 1 | **64+64** | **128+64** | **192+64** | **2016+64** |

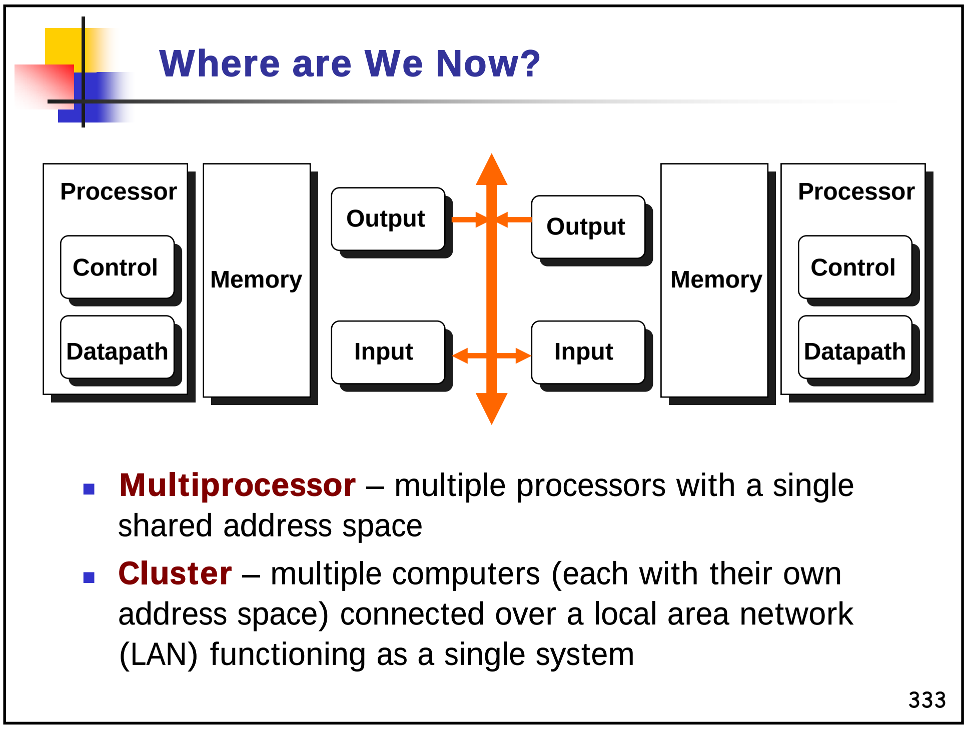
Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

331



## マルチコアプロセッサ

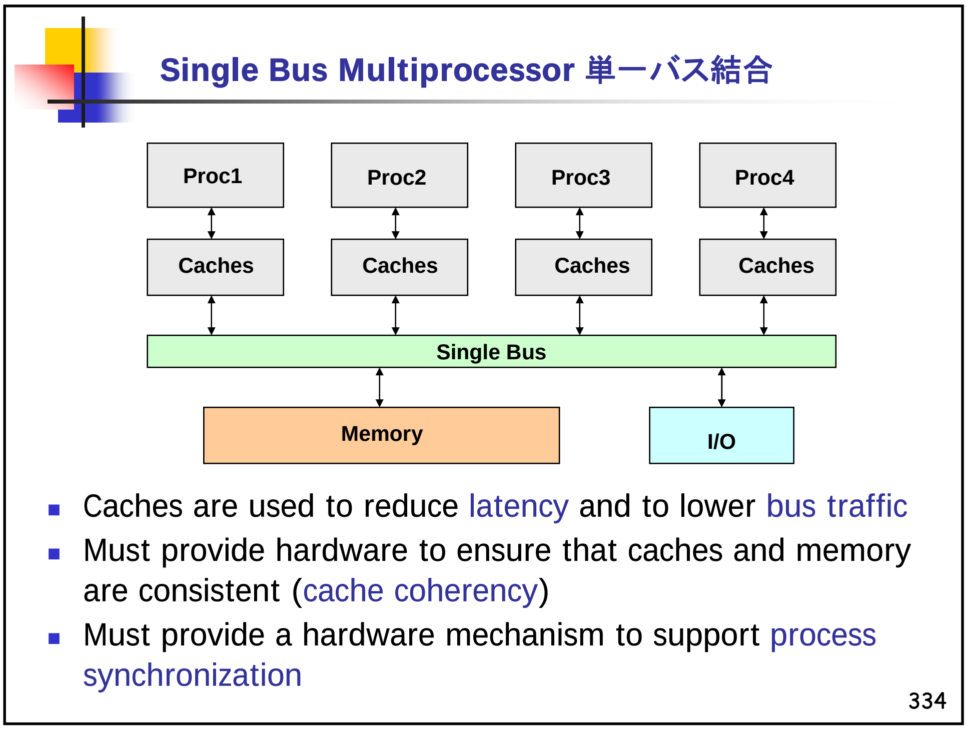
332



## Where are We Now?

- **Multiprocessor** – multiple processors with a single shared address space
- **Cluster** – multiple computers (each with their own address space) connected over a local area network (LAN) functioning as a single system

333

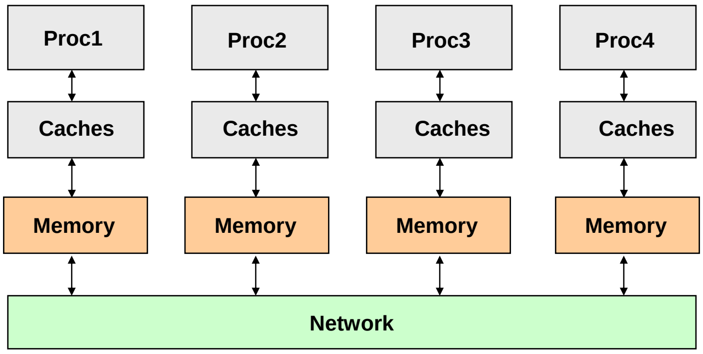


## Single Bus Multiprocessor 単一バス結合

- Caches are used to reduce latency and to lower bus traffic
- Must provide hardware to ensure that caches and memory are consistent (cache coherency)
- Must provide a hardware mechanism to support process synchronization

334

## ネットワーク結合のマルチプロセッサ



Proc1 Proc2 Proc3 Proc4

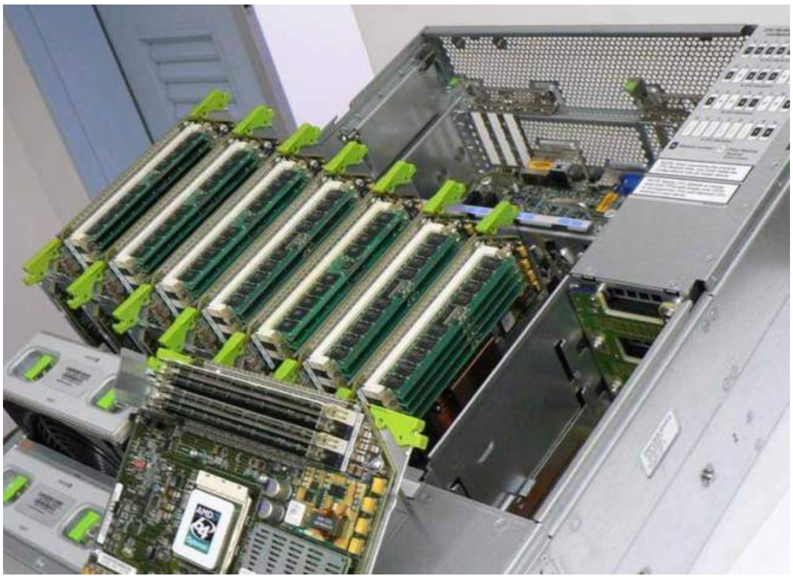
Caches Caches Caches Caches

Memory Memory Memory Memory

Network

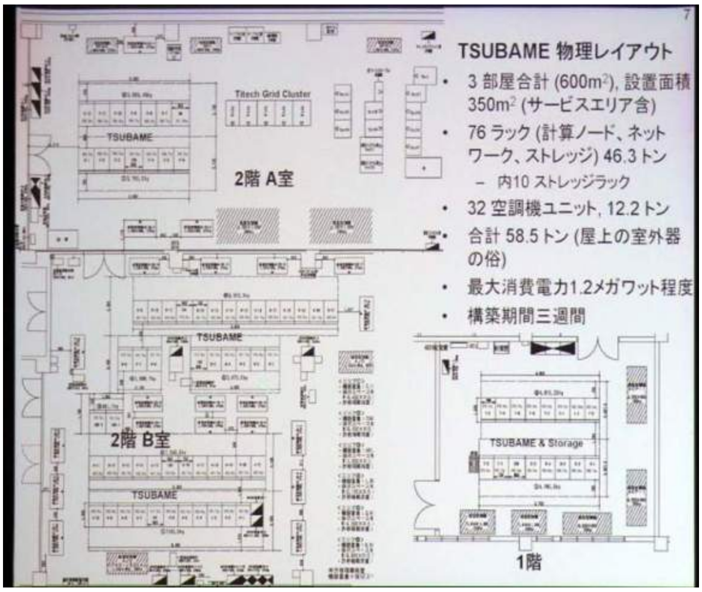
## TokyoTech TSUBAME

## TokyoTech TSUBAME



## TokyoTech TSUBAME

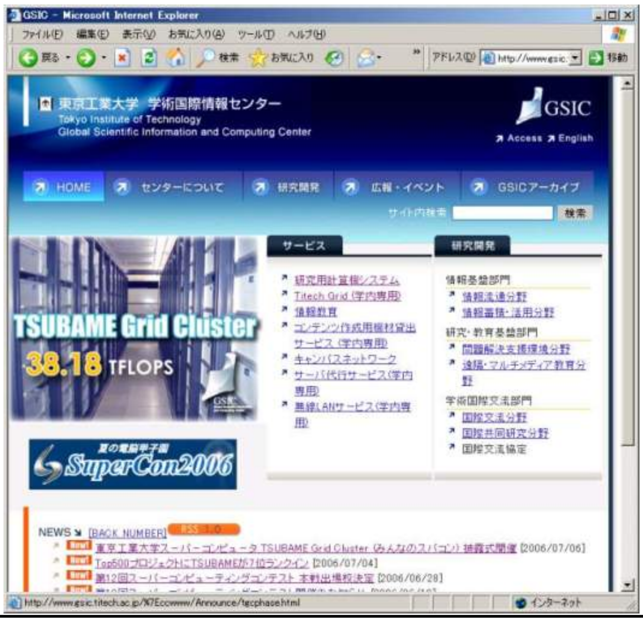
## TSUBAME 物理レイアウト



### TSUBAME 物理レイアウト

- 3 部屋合計 (600m$^2$), 設置面積 350m$^2$ (サービスエリア含)
- 76 ラック (計算ノード、ネットワーク、ストレッジ) 46.3 トン
  - 内10 ストレッジラック
- 32 空調機ユニット, 12.2 トン
- 合計 58.5 トン (屋上の室外器の俗)
- 最大消費電力1.2メガワット程度
- 構築期間三週間

## TokyoTech TSUBAME



GSIC - Microsoft Internet Explorer
ファイル(F) 編集(E) 表示(V) お気に入り(A) ツール(T) ヘルプ(H)
戻る
検索
お気に入り
アドレス(D)
http://www.gsic
移動

東京工業大学 学術国際情報センター
Tokyo Institute of Technology
Global Scientific Information and Computing Center

GSIC

Access English

HOME センターについて 研究開発 広報・イベント GSICアーカイブ

サイト内検索 検索

TSUBAME Grid Cluster
38.18 TFLOPS

夏の電脳甲子園
SuperCon2006

サービス

- 研究用計算機システム
- Titech Grid (学内専用)
- 情報教育
- コンテンツ作成用機材貸出サービス (学内専用)
- キャンパスネットワーク
- サーバ代行サービス(学内専用)
- 無線LANサービス(学内専用)

研究開発

情報基盤部門
- 情報流通分野
- 情報蓄積・活用分野

研究・教育基盤部門
- 問題解決支援環境分野
- 遠隔・マルチメディア教育分野

学術国際交流部門
- 国際交流分野
- 国際共同研究分野
- 国際交流協定

NEWS [BACK NUMBER] RSS 1.0
- New! 東京工業大学スーパーコンピュータ TSUBAME Grid Cluster (みんなのスパコン) 披露式開催 [2006/07/06]
- New! Top500プロジェクトにTSUBAMEが7位ランクイン [2006/07/04]
- New! 第12回スーパーコンピューティングコンテスト 本戦出場校決定 [2006/06/28]

http://www.gsic.titech.ac.jp/%7Eccwww/Announce/tgcphase.html

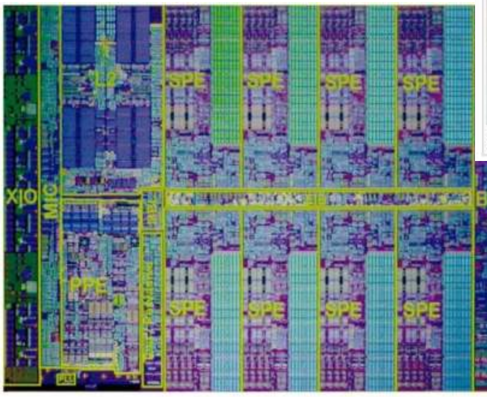
インターネット

## チップマルチプロセッサの例
## Cell Broadband Engine

341

## Cell Broadband Engine & PS3

PLAYSTATION 3試作見本（東京ゲームショウ2005バージョン）



342

## Cell BE Element Interconnect Bus

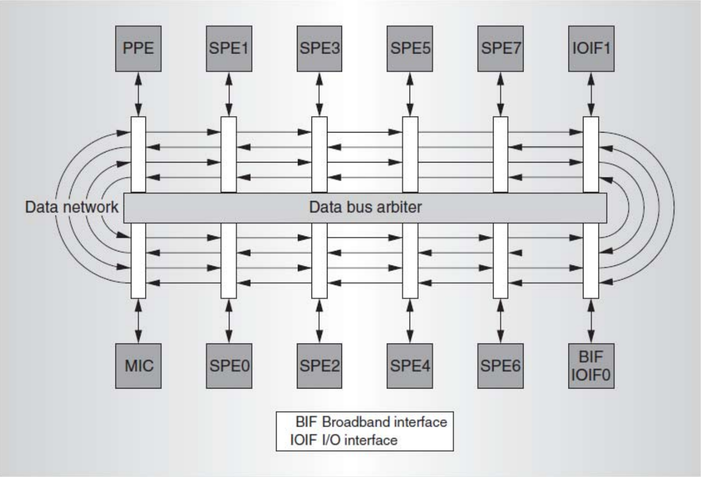
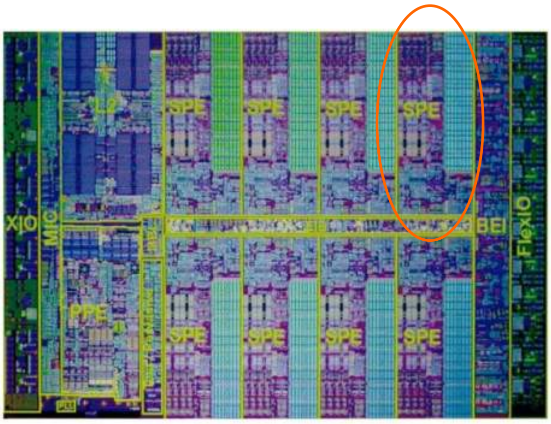


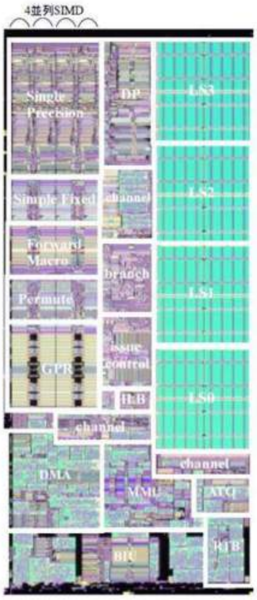
Figure 2. Element interconnect bus (EIB).

IEEE Micro, Cell Multiprocessor Communication Network: Built for Speed

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005

## SPE (Synergistic Processor Element)





## Cell/B.E. のピーク性能

- 1サイクルで積和演算を1回実行できる演算器 (2 FLOP/cycle)
- SIMD構成で，SPEあたりの並列度 4
- チップ内のSPEの数 8
- 動作周波数 4GHz

- 2 × 4 × 8×4 ＝ 256 GFLOPS
- （ペンティアムは 8GFLOPS 程度）

## プログラミング例

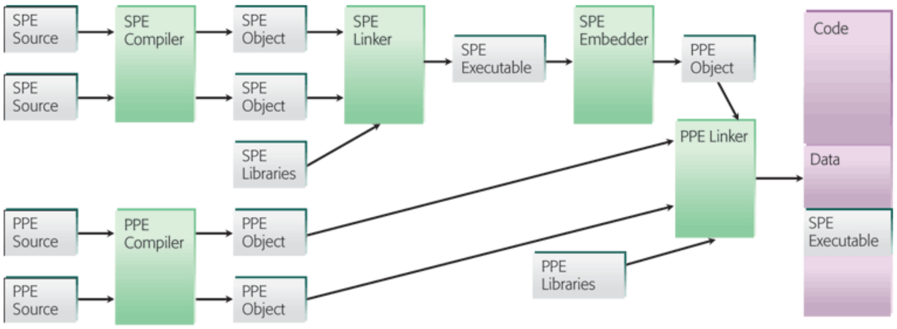
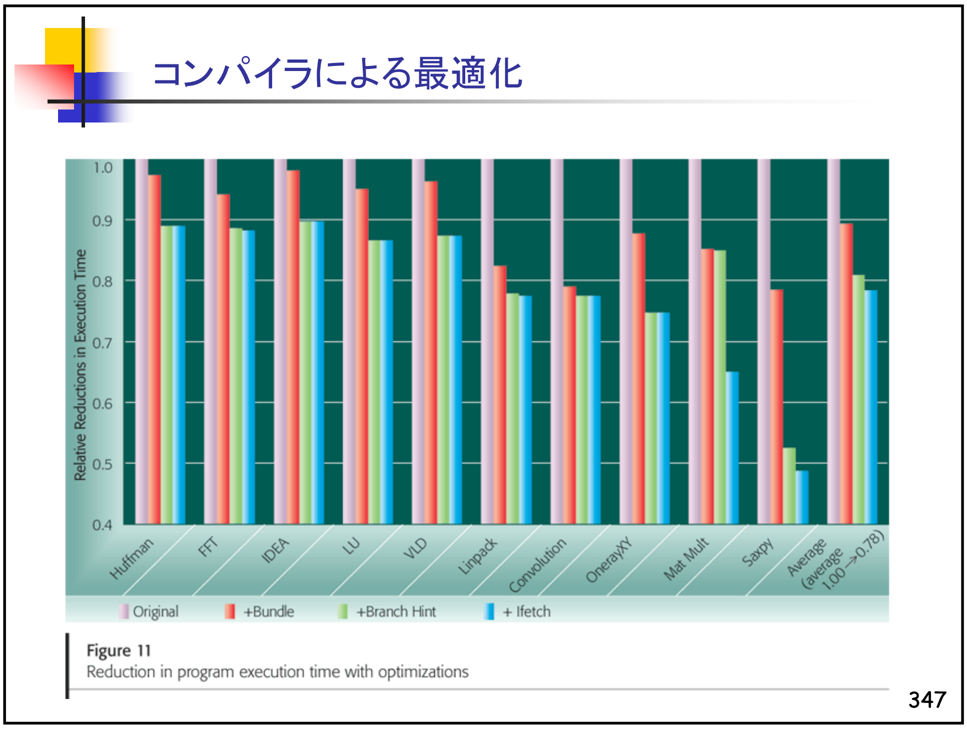


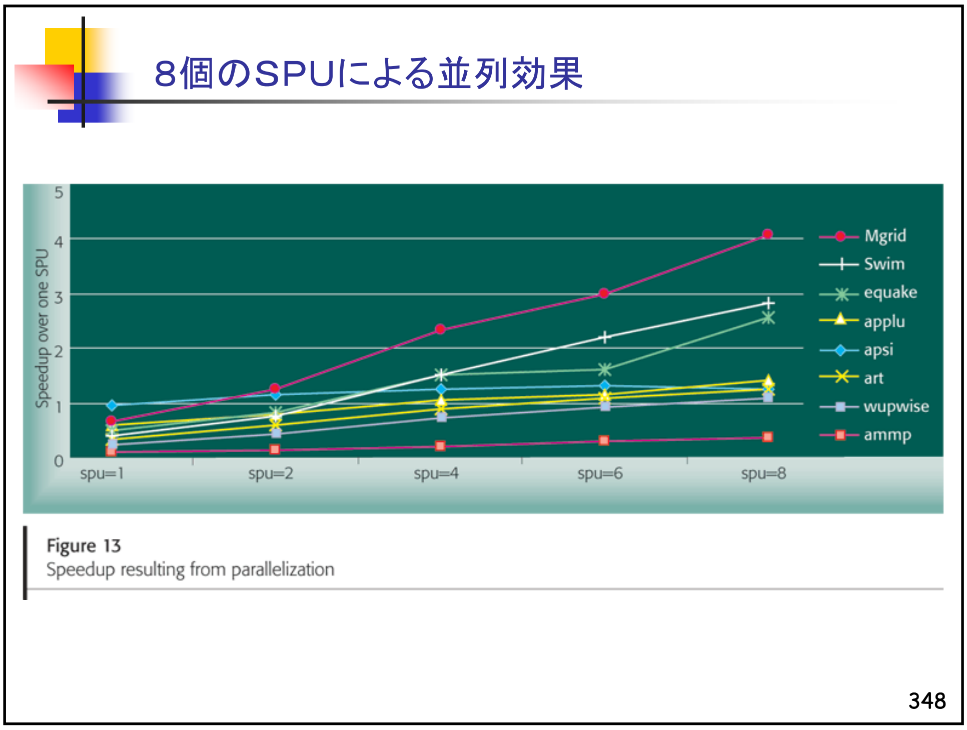
Figure 2
Manually compiling and binding a Cell BE program



## コンパイラによる最適化

Figure 11
Reduction in program execution time with optimizations

347



## ８個のSPUによる並列効果

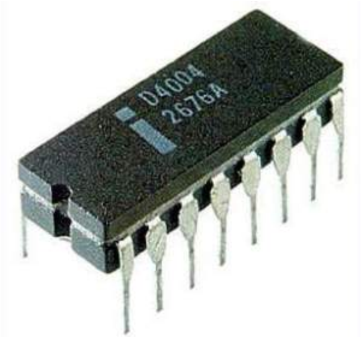
Figure 13
Speedup resulting from parallelization

348

## Cell/B.E. まとめ

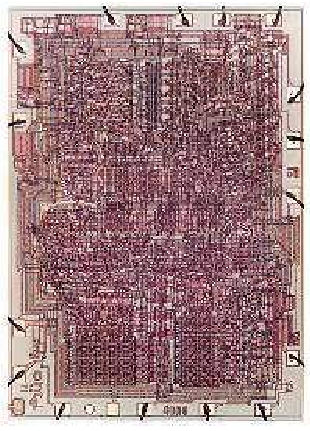
- 256GFLOPSという高いピーク性能
- SPE
  - キャッシュ無し，分岐予測無し
  - オーバヘッドの削減
  - SIMD並列化，DMA転送
  - ８個のSPEを利用した並列化
- アセンブラ技術
- ミドルウェア
- オペレーティングシステム
- コンパイラ技術

## 世界初のマイクロプロセッサ



| プロセッサ | 出荷年 | トランジスタ数 |
|---|---|---|
| 4004 | 1971 | 2,250 |

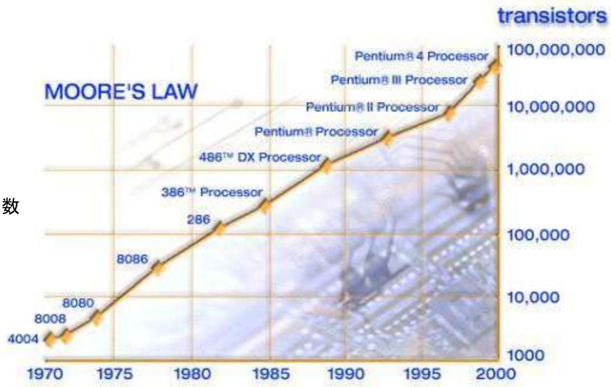
1971年: 4004 マイクロプロセッサ



出典: フリー百科事典『ウィキペディア（Wikipedia）』，　Intelミュージアム

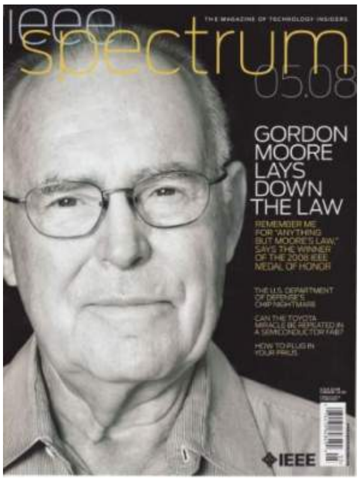
## メニーコアへの流れ，ムーアの法則

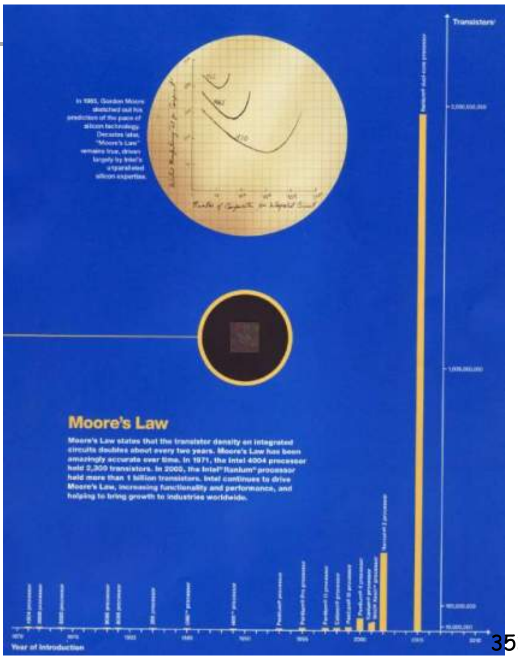
| プロセッサ | 出荷年 | トランジスタ数 |
|---|---|---|
| 4004 | 1971 | 2,250 |
| 8008 | 1972 | 2,500 |
| 8080 | 1974 | 5,000 |
| 8086 | 1978 | 29,000 |
| 286 | 1982 | 120,000 |
| 386™ processor | 1985 | 275,000 |
| 486™ DX processor | 1989 | 1,180,000 |
| Pentium® processor | 1993 | 3,100,000 |
| Pentium II processor | 1997 | 7,500,000 |
| Pentium III processor | 1999 | 24,000,000 |
| Pentium 4 processor | 2000 | 42,000,000 |



出典：Intel社, http://www.intel.com/research/silicon/mooreslaw.htm

## Moore’s Law


THE MAGAZINE OF TECHNOLOGY INSIDERS
ieee spectrum
05.08
GORDON MOORE LAYS DOWN THE LAW
REMEMBER ME FOR "ANYTHING BUT MOORE'S LAW," SAYS THE WINNER OF THE 2008 IEEE MEDAL OF HONOR
THE U.S. DEPARTMENT OF DEFENSE'S CHIP NIGHTMARE
CAN THE TOYOTA MIRACLE BE REPEATED IN A SEMICONDUCTOR FAB?
HOW TO PLUG IN YOUR PRIUS
IEEE

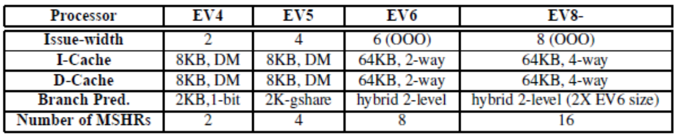


In 1965, Gordon Moore sketched out his prediction of the pace of silicon technology. Decades later, "Moore's Law" remains true, driven largely by Intel's unparalleled silicon expertise.

Moore's Law

Moore's Law states that the transistor density on integrated circuits doubles about every two years. Moore's Law has been amazingly accurate over time. In 1971, the Intel 4004 processor held 2,300 transistors. In 2000, the Intel® Itanium® processor held more than 1 billion transistors. Intel continues to drive Moore's Law, increasing functionality and performance, and helping to bring growth to industries worldwide.

## マルチコア（～10個程度）からメニーコア（多数）へ



| Processor | EV4 | EV5 | EV6 | EV8- |
|---|---|---|---|---|
| Issue-width | 2 | 4 | 6 (OOO) | 8 (OOO) |
| I-Cache | 8KB, DM | 8KB, DM | 64KB, 2-way | 64KB, 4-way |
| D-Cache | 8KB, DM | 8KB, DM | 64KB, 2-way | 64KB, 4-way |
| Branch Pred. | 2KB,1-bit | 2K-gshare | hybrid 2-level | hybrid 2-level (2X EV6 size) |
| Number of MSHRs | 2 | 4 | 8 | 16 |

Table 1. Configuration of the cores

数世代の
RISCプロセッサのサイズ

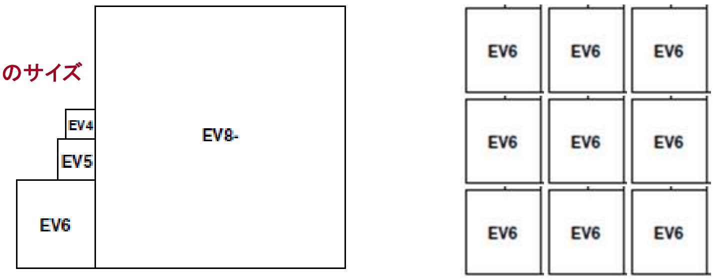


Figure 1. Relative sizes of the cores used in the study

Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction, MICRO-36

## マルチコア（～10個程度）からメニーコア（多数）へ

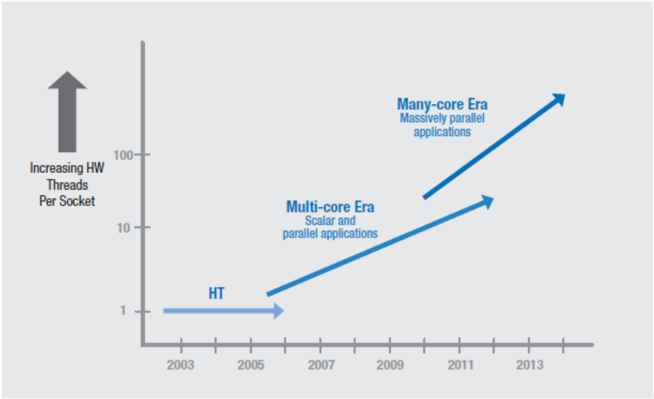
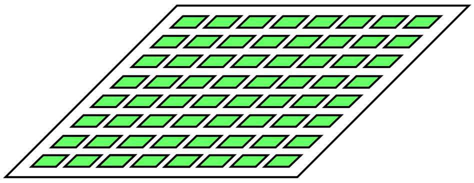


Figure 1: Current and expected eras of Intel® processor architectures

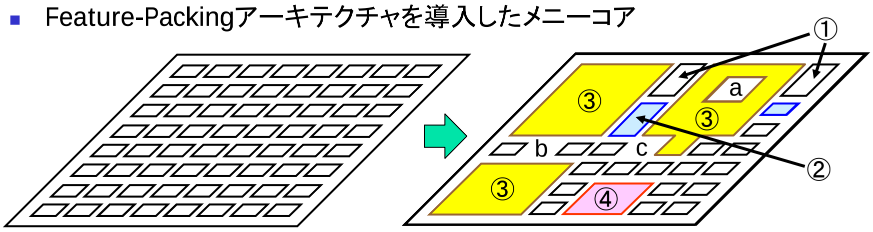
Platform 2015: Intel® Processor and Platform Evolution for the Next Decade

## Feature-Packingアーキテクチャ

- 通常のメニーコア（均一なコアの場合）



- Feature-Packingアーキテクチャを導入したメニーコア



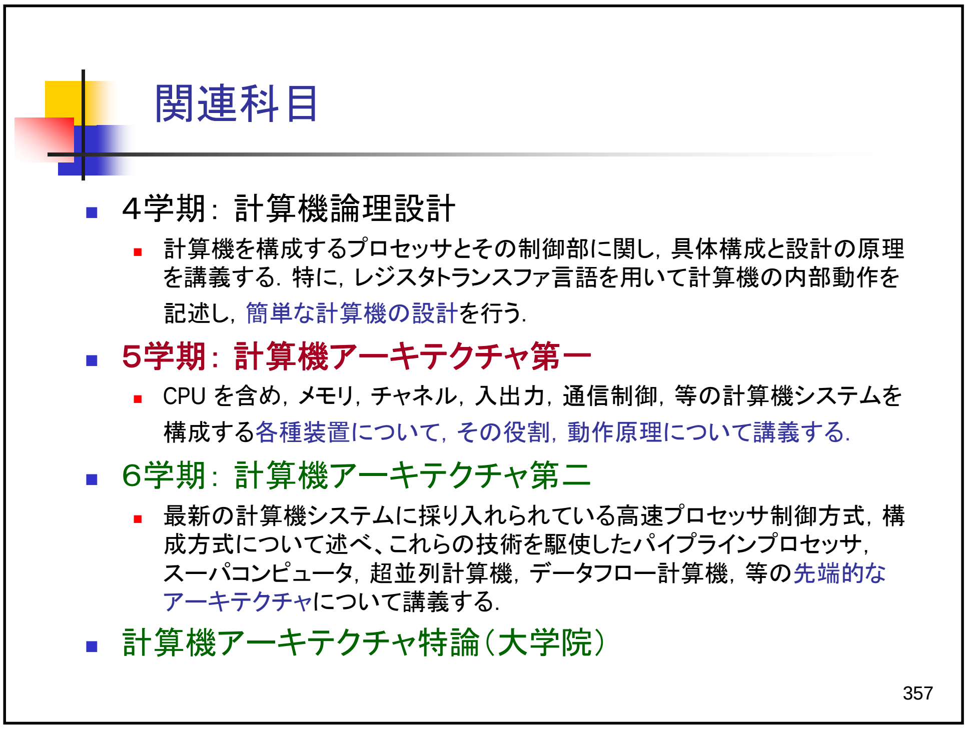
▱ 計算コア（アプリケーション実行）

▱ 命令供給支援コア　▱ データ供給支援コア

## レポート 提出方法

- ８月１１日（午後５時）までに電子メールで提出
  - report@arch.cs.titech.ac.jp
- レポートの詳細はホームページを参照
- 電子メールのタイトル
  - Computer Architecture Report
- 電子メールの内容
  - 氏名，学籍番号
  - 回答
    - PDFファイルを添付

Adapted from *Computer Organization and Design*, Patterson & Hennessy, © 2005



## 関連科目

- 4学期：計算機論理設計
  - 計算機を構成するプロセッサとその制御部に関し，具体構成と設計の原理を講義する．特に，レジスタトランスファ言語を用いて計算機の内部動作を記述し，簡単な計算機の設計を行う．
- 5学期：計算機アーキテクチャ第一
  - CPU を含め，メモリ，チャネル，入出力，通信制御，等の計算機システムを構成する各種装置について，その役割，動作原理について講義する．
- 6学期：計算機アーキテクチャ第二
  - 最新の計算機システムに採り入れられている高速プロセッサ制御方式，構成方式について述べ、これらの技術を駆使したパイプラインプロセッサ，スーパコンピュータ，超並列計算機，データフロー計算機，等の先端的なアーキテクチャについて講義する．
- 計算機アーキテクチャ特論（大学院）

357