高级计算机体系结构

陈文智　浙江大学计算机学院

chenwz@zju.edu.cn

2014年9月

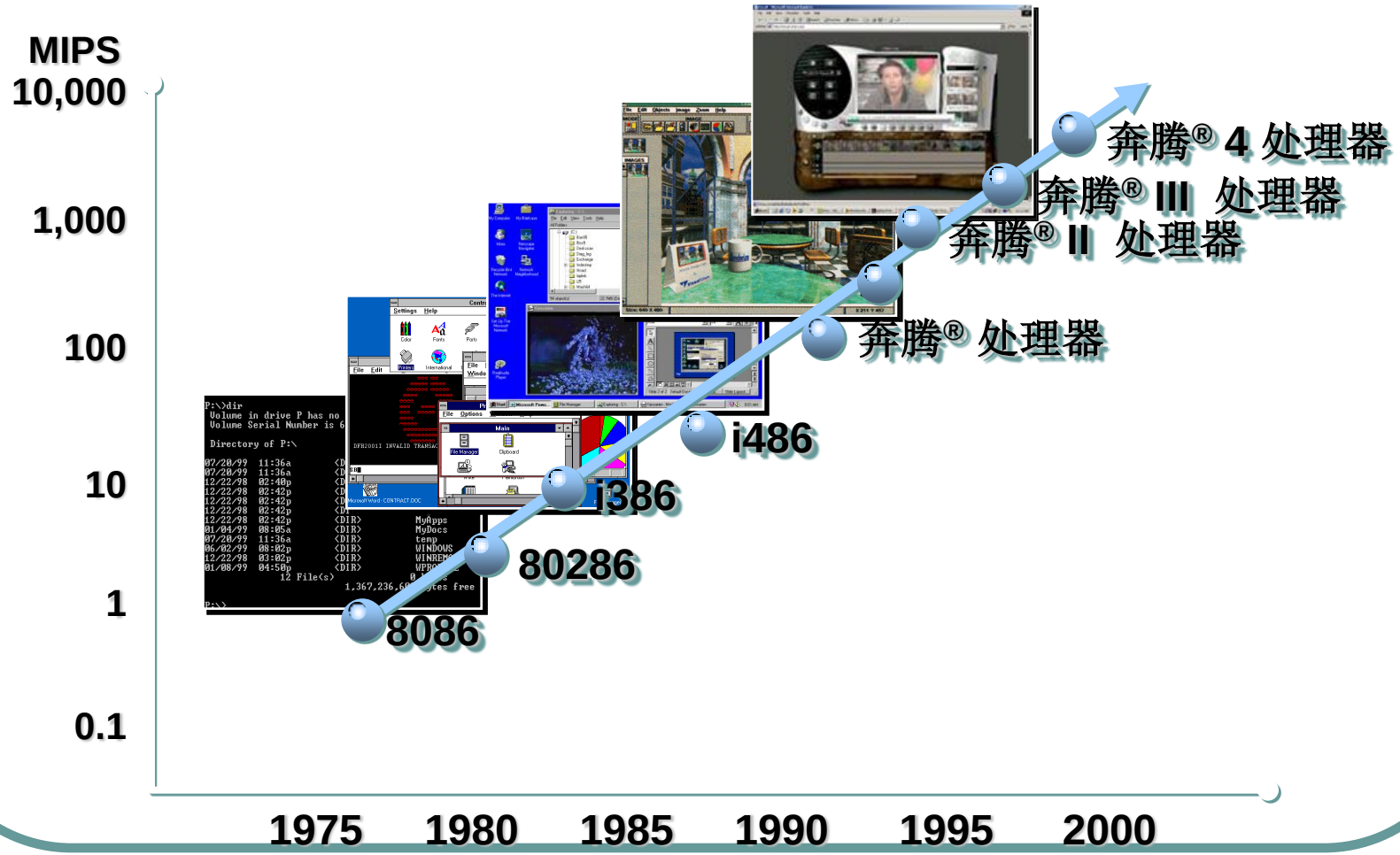
1.1 计算机技术发展综述(1)

- **1946年：在二次世界大战期间研制成功的世界上第一台电子计算机ENIAC (Electronic Numerical Intergrator and Calculator)正式对外宣布。**
  - 用途：军用；
  - 体积：100英尺长×8.5英尺高×n英尺宽；
    组成：18000真空管；
  - 指令数：不足10条，加法，数据传输与转移

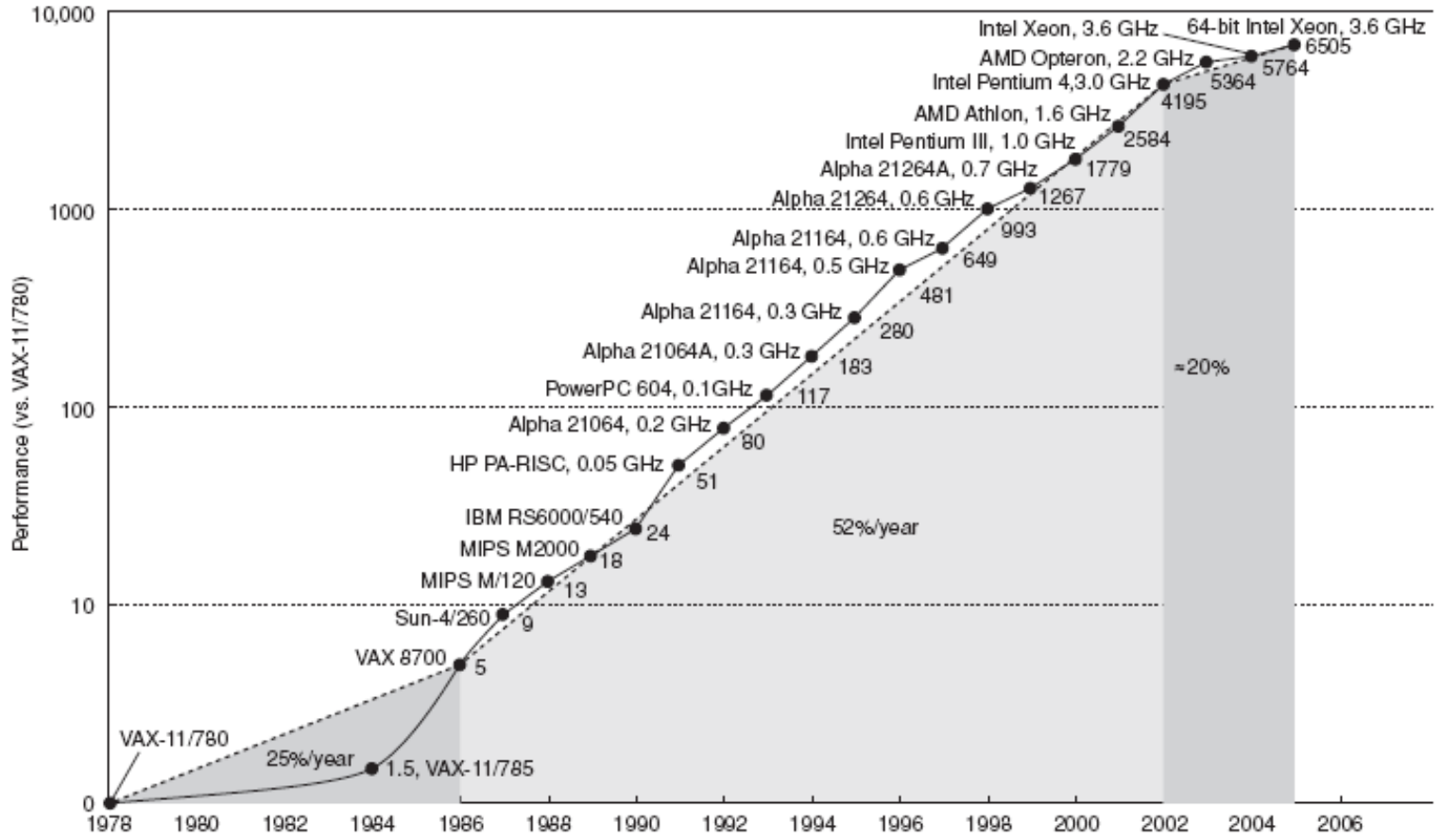
计算机技术发展综述(2)

- 60多年来计算机技术有了惊人的发展
  - 性能：（加法）速度提高了**>5个数量级**
  - 价格：**今天$500的机器相当于80年代中$$10^7$$的机器，这里同性能计算机的价格比，改善了近5个数量级。**

处理能力 → 新的应用


MIPS
10,000
1,000
100
10
1
0.1
1975
1980
1985
1990
1995
2000
8086
80286
i386
i486
奔腾® 处理器
奔腾® II 处理器
奔腾® III 处理器
奔腾® 4 处理器

Incredible performance improvement



计算机技术发展综述(3)

- 计算机技术快速发展进步的原因之一

  技术进步——集成电路技术的进步，还有存储器（包括内外存）和各类外设的进步。

  特点：稳定快速发展，即按Moore定律发展，即微处理器性能（按芯片上晶体管数定义）每18个月翻一番，即每年提高58%。

$$D(T) = D(T_0)2^{(T-T_0)/1.5}$$

计算机技术发展综述(4)

- 计算机技术快速发展进步的原因之二
  - 计算机设计创新，即计算机体系结构的不断创新。
    - 经历了由简单→复杂→极其复杂→简单→复杂→极其复杂的经历
    - 有时快，有时慢(1977年的VAX/780为 1 MIPS机器，1985年VAS/785仅为1.5 MIPS，几乎停止不前。
    - 有很多技术，经不起时间考验，已退出历史舞台。

计算机技术发展综述(5)

- 今天计算机体系结构的研究内容
  - 进一步提高单个微处理器的性能。(光速极限问题)
  - 基于微处理器的多处理器体系结构。
  - 全面提高计算机的系统性能：可用性，可维护性，可缩放性。
  - 新型器件的处理器：如光计算机；新原理的计算机（生物，分子，又提出了DNA计算机)。

1.2 计算机体系结构定义(1)

- 定义1：

  “计算机体系结构设计就是指令集设计”

  早期：指令集设计具有至关重要的作用

  今天：指令集趋向于相同（差别很小）。

  甚至：指令集几乎相同的不同机器，其性能差异很大。

  定义存在问题：与计算机的功能设计，逻辑设计以及实现技术分割开来，是不科学的。

计算机体系结构定义(2)

- 定义2：

  计算机体系结构应包括指令集设计，计算机组成设计与硬件（硬件与逻辑设计）实现。

  体系结构不同的例子：

  指令集相同，组成不同：

  VAX11/780-- SPARC2;

  VAX8600--SPARC20;

  指令集相同，组成也相同，实现不同：

  VAX11/780--VAX11/785（IC工艺不同）；

  不同型号的**Indy(**时钟和**Cache**不同）

1.3 计算机系统结构的分类

一、当前计算机的种类

1989IEEE电气与电子工程师委员会提出的计算机分类：

- 个人计算机 Personal Computer (PC)
- 工作站 WorkStation (WS)
- 小型机 Mini Computer
- 中型机 Mainframe
- 小巨型机 Minisupercomputer
- 巨型机 Supercomputer

*服务器

*桌面计算

*嵌入式系统

*网络并行计算机(Cluster、Grid Computer)

*云计算、框计算、智慧地球

界线模糊

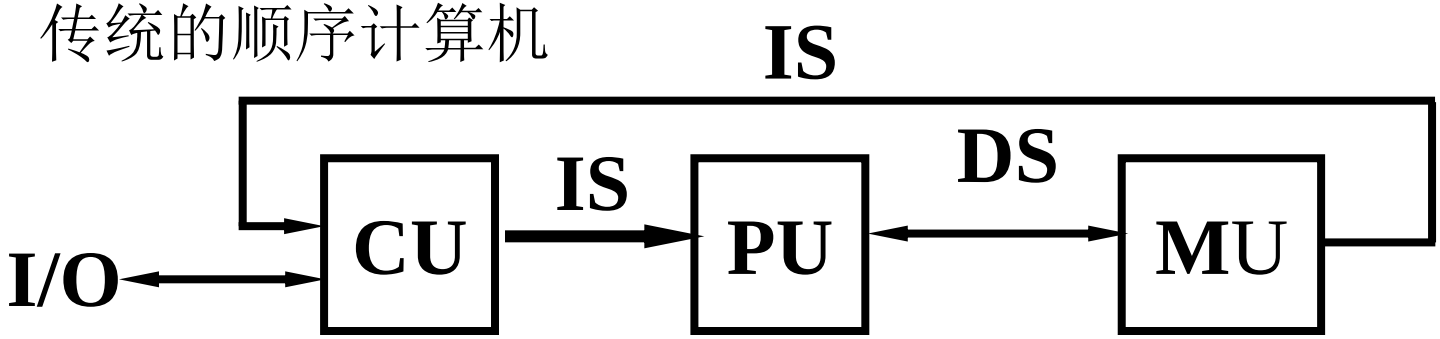
二、Flynn分类法

- 根据指令流和数据流的多倍性对计算机系统结构进行分类，
- 基本思想是计算机工作过程是指令流的执行和数据流的处理。
- **指令流：**机器执行的指令序列
- **数据流：**由指令流调用的数据序列（包括输入数据和中间结果）
- **多倍性：**在系统性能的瓶颈部件上处于同一执行阶段的指令或数据的最大个数。

Flynn分类一

1．单指令流----单数据流 SISD

传统的顺序计算机

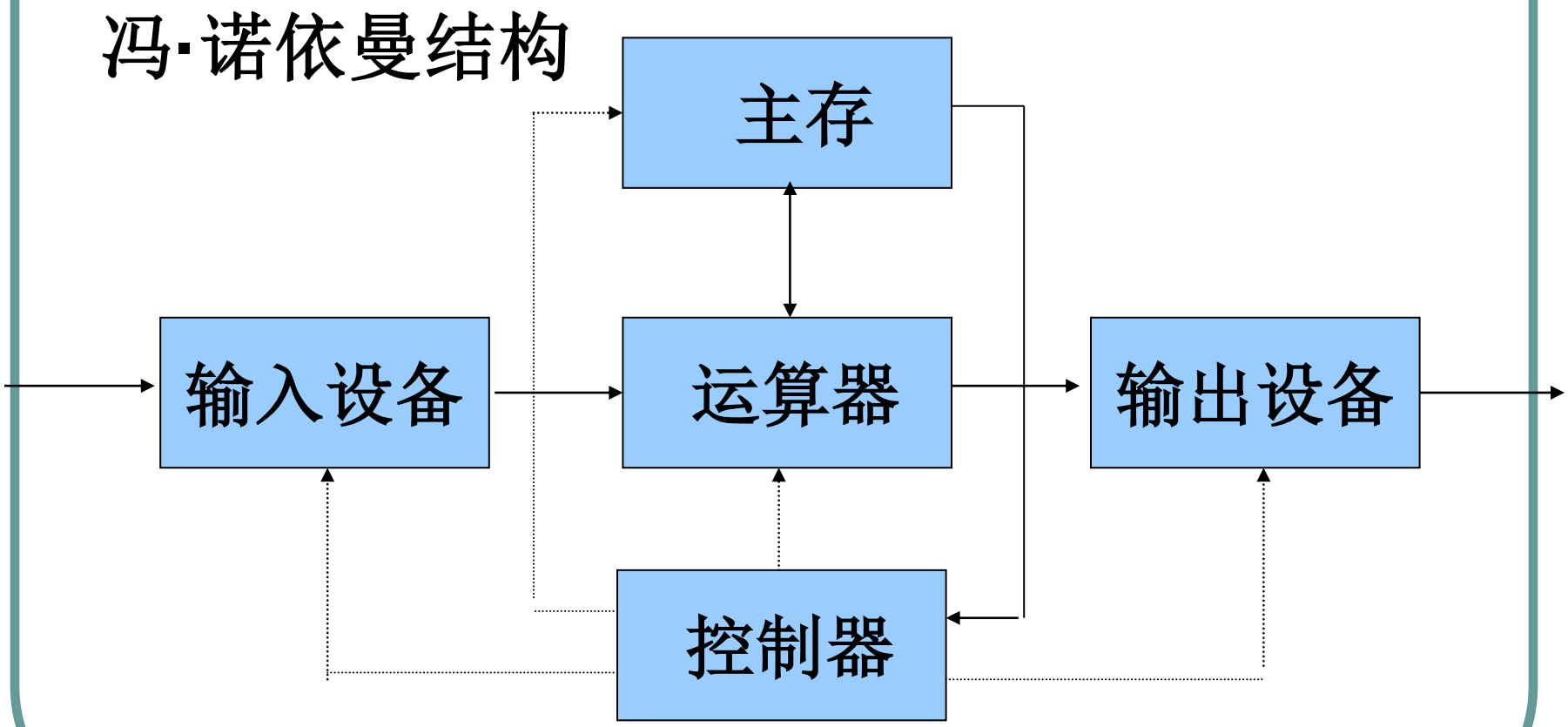


CU：控制部件　　IS：指令流

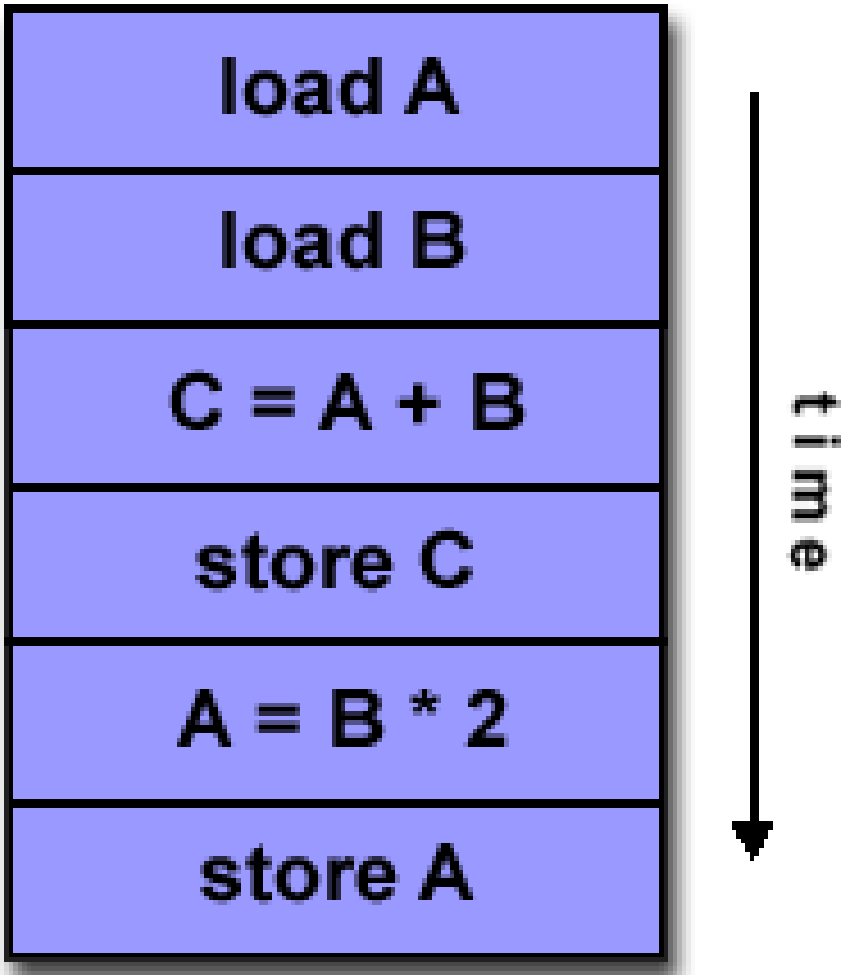
PU：处理部件　　DS：数据流

MU：存储部件

典型结构：传统的顺序计算机



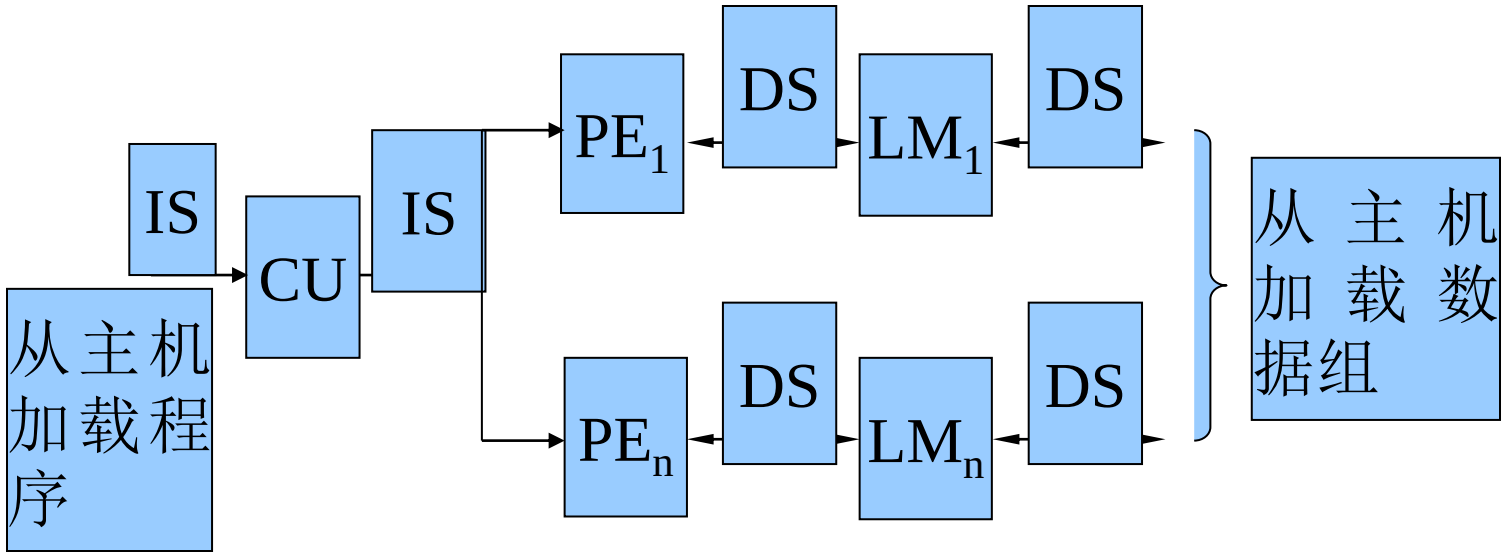
SISD



Flynn分类二

2．单指令流----多数据流 SIMD 超级计算机

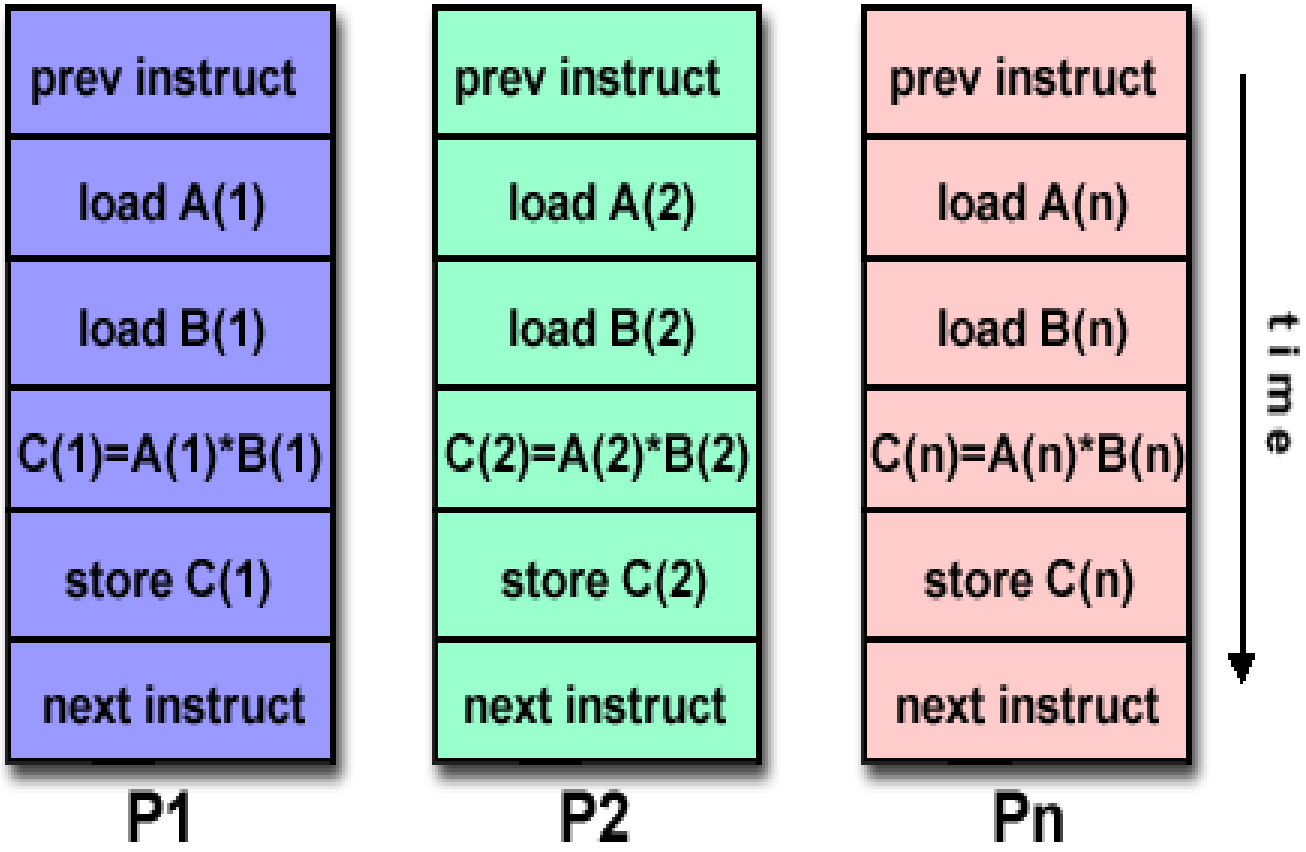
单控制器、多处理单元和多对数据进行处理



PE：处理单元

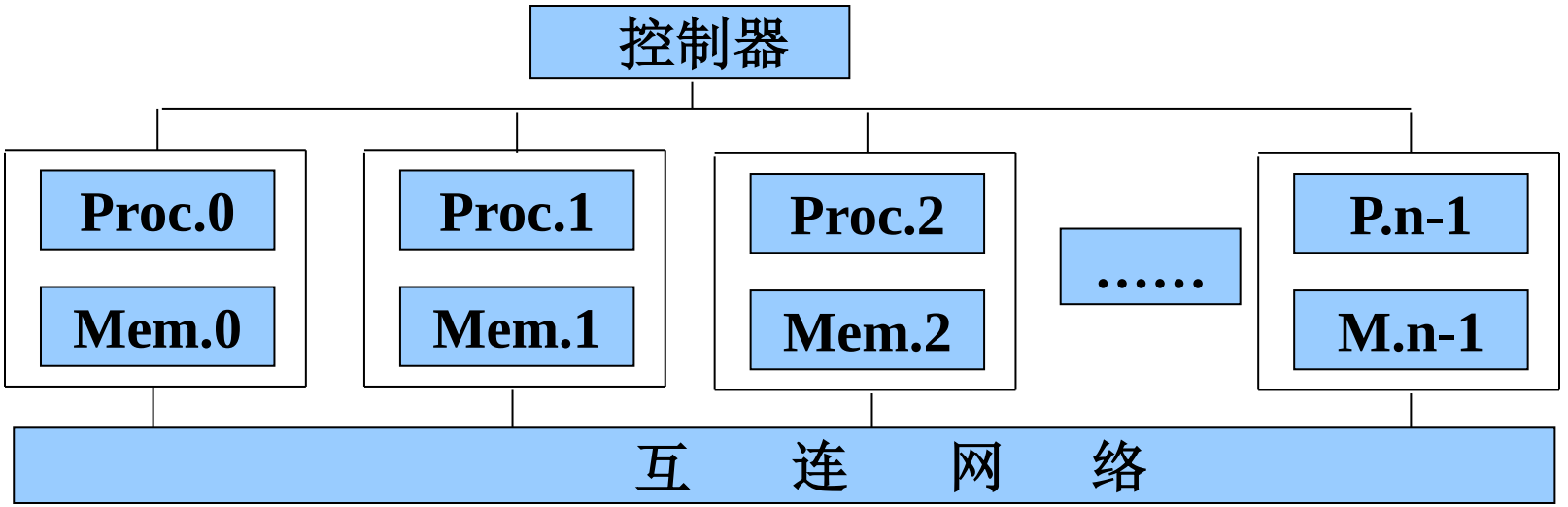
LM：本地存储器（分布存储器）

SIMD


prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
load A(2)
load B(2)
C(2)=A(2)*B(2)
store C(2)
next instruct
P2
prev instruct
load A(n)
load B(n)
C(n)=A(n)*B(n)
store C(n)
next instruct
Pn
time

典型结构:阵列式计算机

H.J. Siegel 1979 提出的操作模型



M = < N， C， I， M， R>

- N：PE个数
- C：CU指令集
- I：CU送PE的指令集
- M：屏蔽方案
- R：数据通信功能

阵列式结构作矩阵相乘(SIMD)

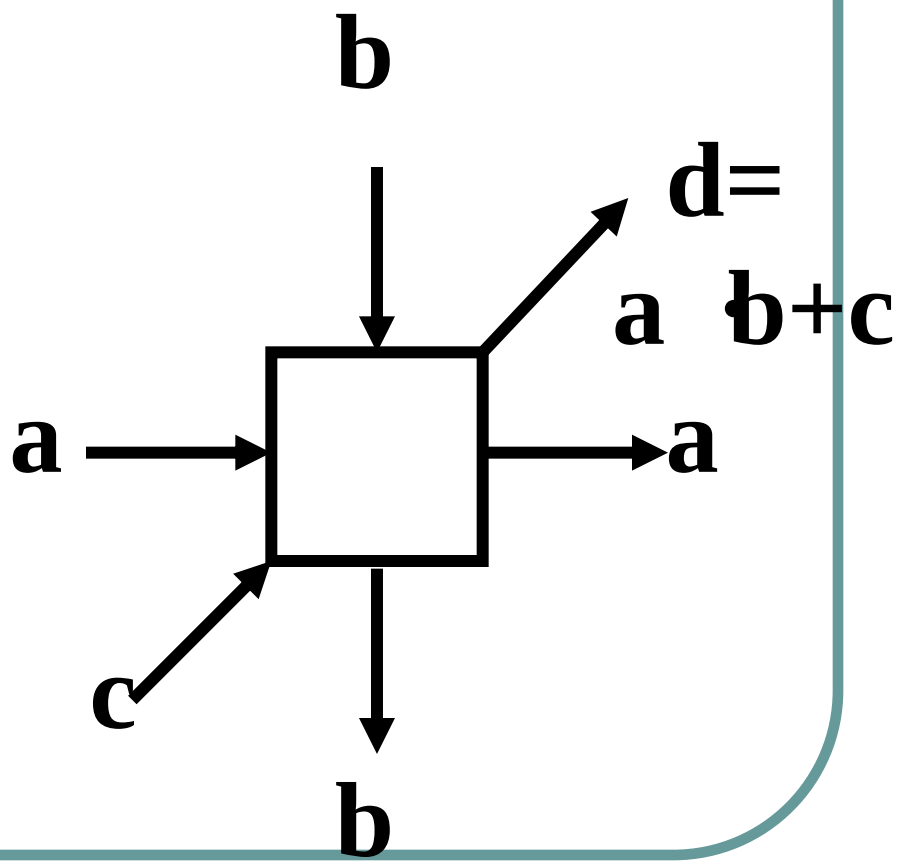
例：矩阵运算

$$A=\begin{bmatrix} a & c \\ b & d \end{bmatrix} \quad B=\begin{bmatrix} e & g \\ f & h \end{bmatrix}$$

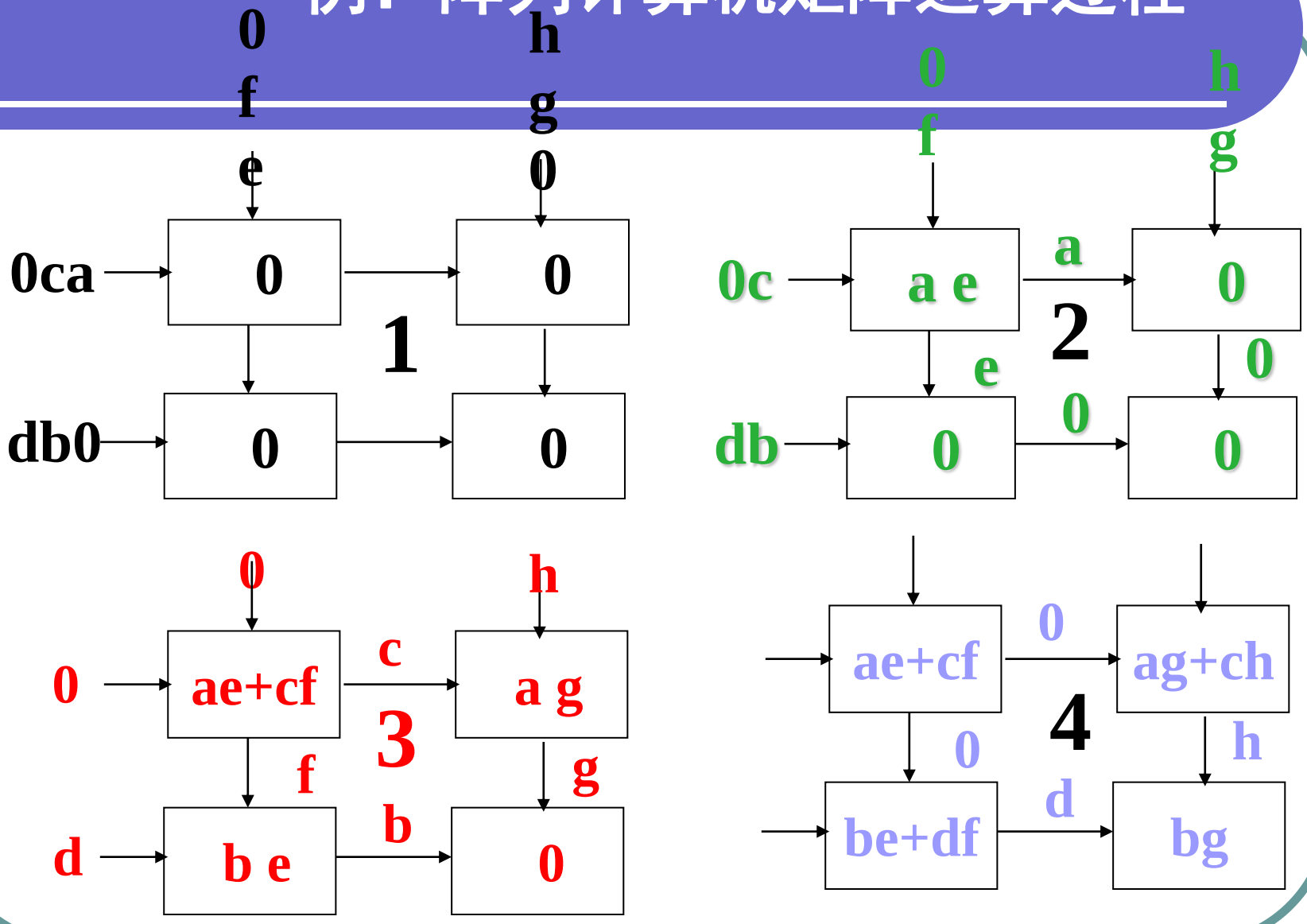
$$C = A \bullet B$$

$$C=\begin{bmatrix} ae+cf & ag+ch \\ be+df & bg+dh \end{bmatrix}$$

PE器件



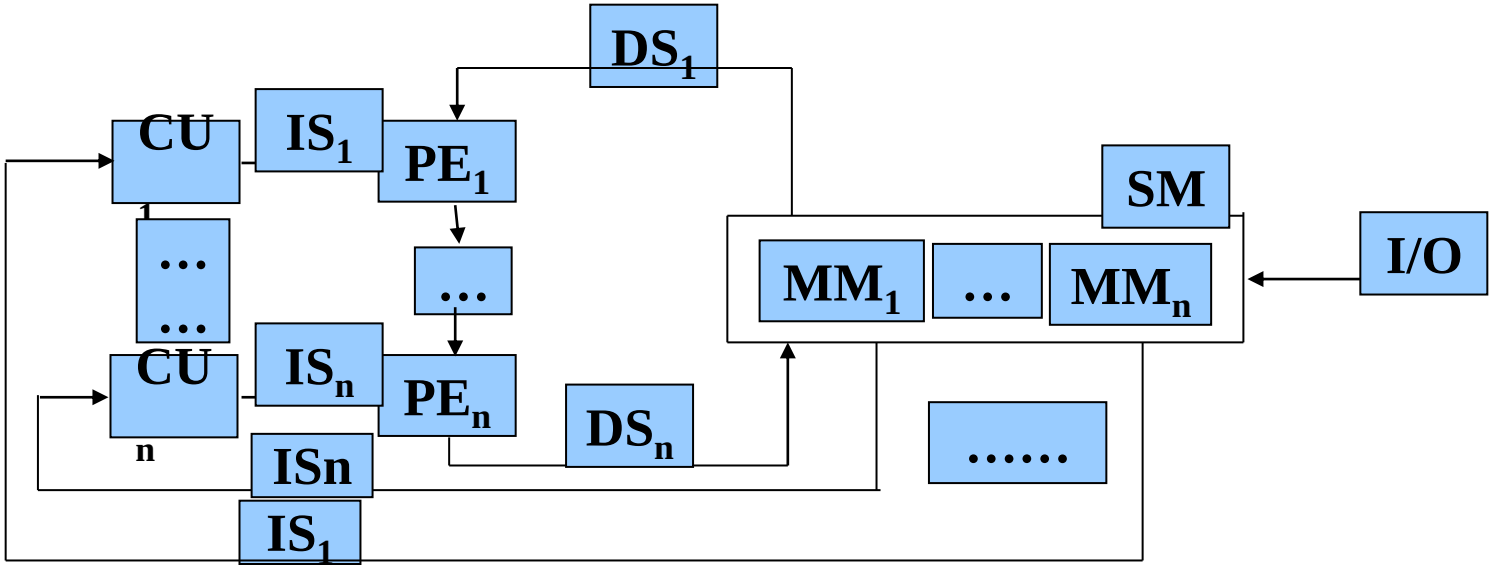
例：阵列计算机矩阵运算过程



Flynn分类三

3. 多指令流----单数据流 MISD

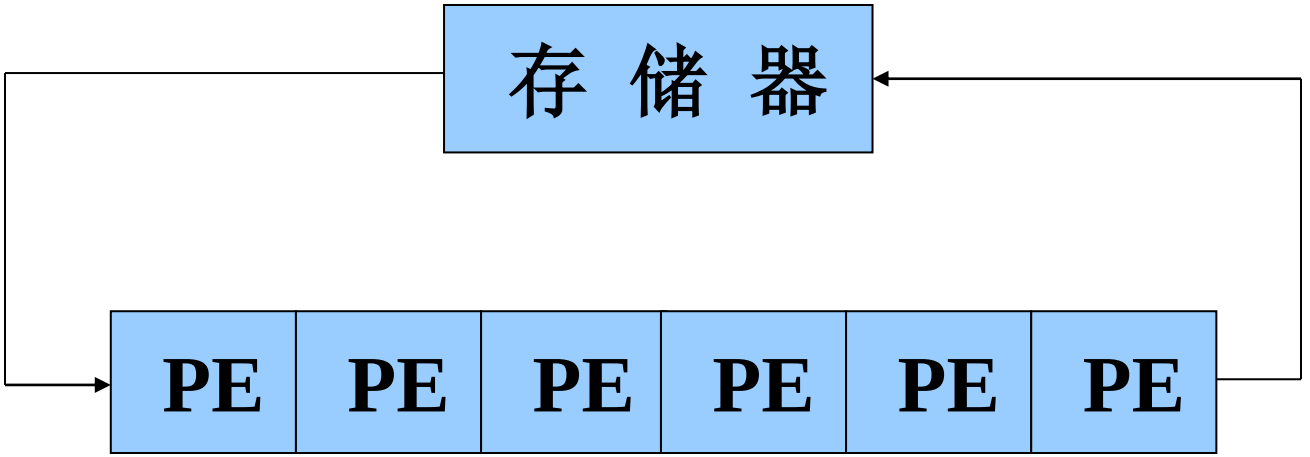
多个处理器，对同一数据流进行处理



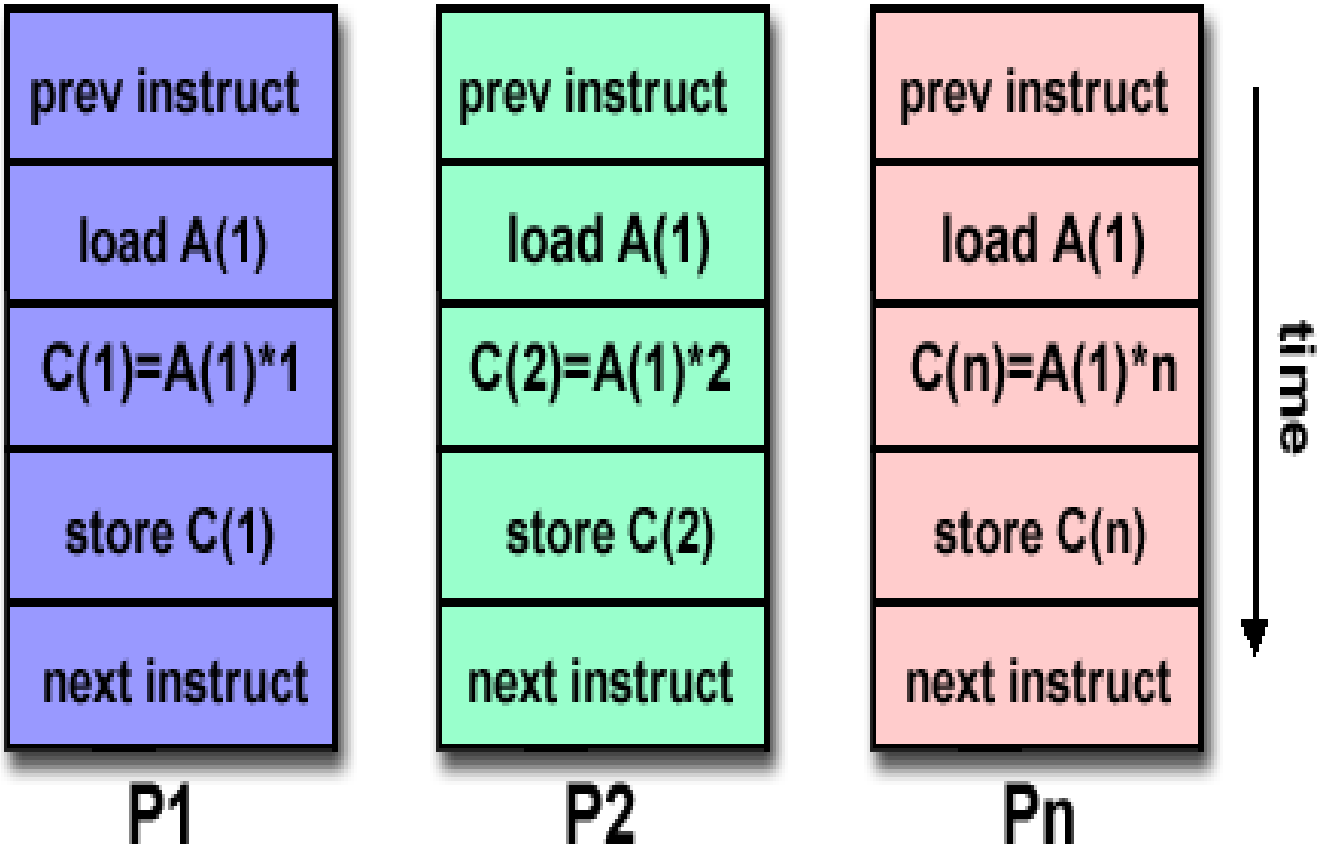
- MM主存贮模块，SM共享存储器

典型结构：脉动阵列计算机

这类计算机的实际机器并不多，一般认为超标量计算机、长指令字计算机（VLIW）和退耦（Decounted）计算机和专用脉动阵列（Systoic arrays）计算机可以作为此类计算机。



MISD


prev instruct
load A(1)
C(1)=A(1)*1
store C(1)
next instruct
P1
prev instruct
load A(1)
C(2)=A(1)*2
store C(2)
next instruct
P2
prev instruct
load A(1)
C(n)=A(1)*n
store C(n)
next instruct
Pn
time

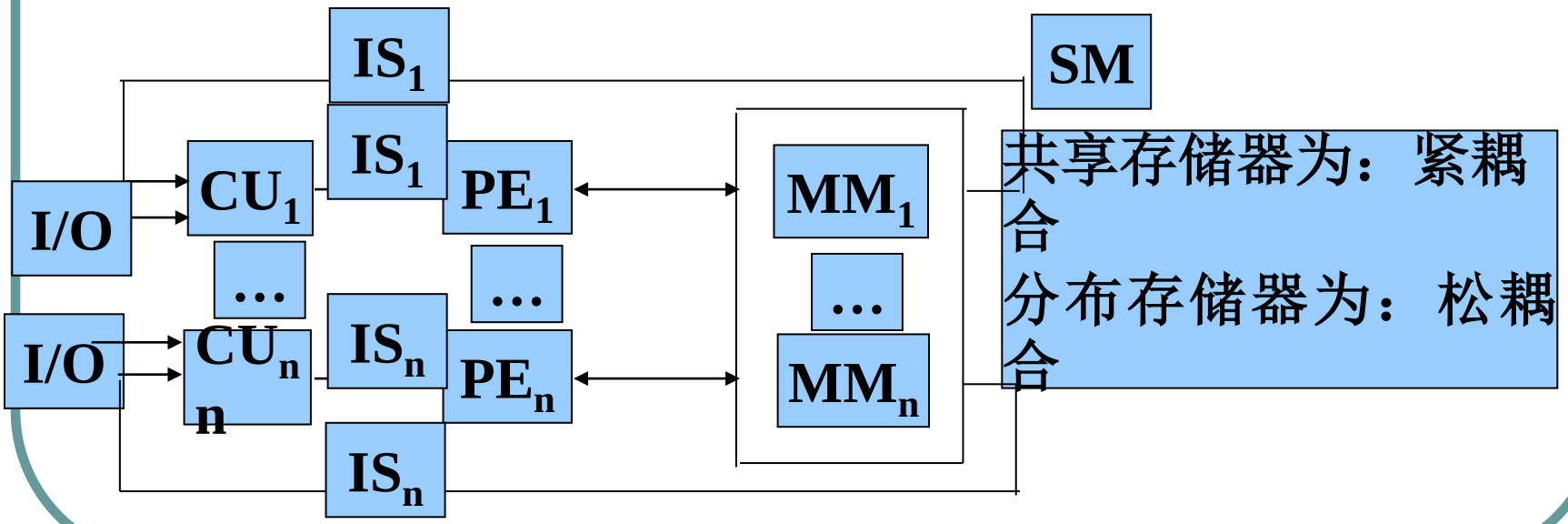
Flynn分类四

4. 多指令流—多数据流 MIMD

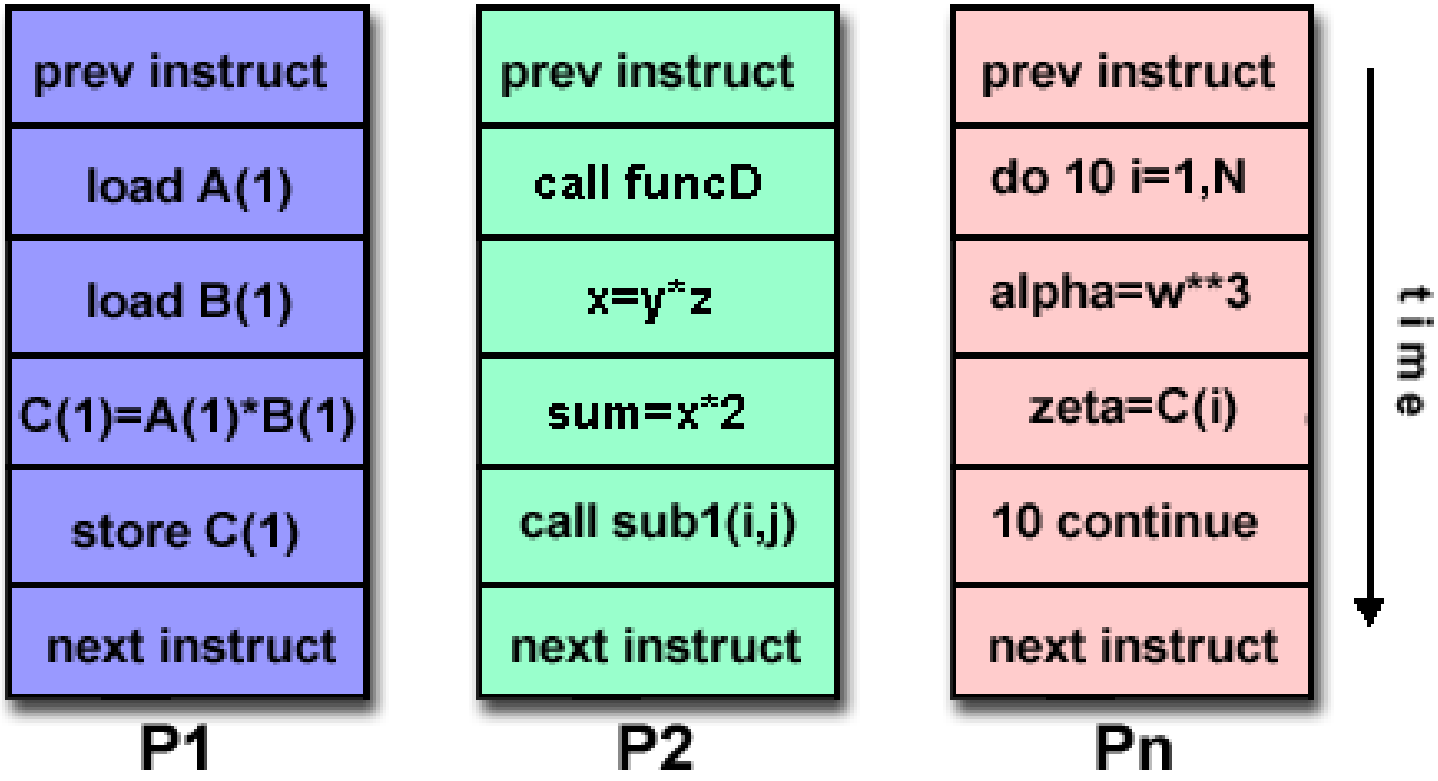
多机系统----多个处理器系统或多计算机系统

每个处理机可以独立执行指令和处理数据

一般并行计算机大多采用这种结构。



MIMD


prev instruct
load A(1)
load B(1)
C(1)=A(1)*B(1)
store C(1)
next instruct
P1
prev instruct
call funcD
x=y*z
sum=x*2
call sub1(i,j)
next instruct
P2
prev instruct
do 10 i=1,N
alpha=w**3
zeta=C(i)
10 continue
next instruct
Pn
time

1.4 Measuring Suits

- *Real applications*
  - *i.e. C compilers, TeX and Spice.*
- Modified (or scripted) applications
  - *To enhance portability or to focus on one particular aspect of system performance.*
- Kernels
  - *Small key pieces (usually small) of real programs. i.e. Livermore Loops and Linpack.*
  - *Used to isolate performance of individual features and help explain behavior of real programs.*

- Toy benchmarks
  - *Small programs (10-100) lines that produce a known result. i.e. QuickSort Sieve of Eratosthenes,Puzzle,*
- Synthetic benchmarks
  - *synthetic benchmarks try to match the average frequency of operations and operands of a large set of programs. i.e. Whetstone and Dhrystone.*
  - *Similar to kernels but are **NOT** real programs !*

1.5 Comparing and Summarizing

- *(一) Total Execution Time: A Consistent Summary Measure*

$$\frac{1}{n}\sum_{i=1}^{n}\text{Time}_i$$

  - *Arithmetic mean:*

- *(二）Weighted Execution Time*
  - The question arises: What is the proper mixture of programs for the workload?
  - *weighted arithmetic mean:*

$$\sum_{i=1}^{n} \text{Weight}_i \times \text{Time}_i$$

    - where Weight*i* is the frequency of the *i*th program in the workload and Time*i* is the execution time of that program.

- *(三）Normalized Execution Time and the Pros and Cons of Geometric Means*
  - the geometric mean

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

    - where Execution time ratio*i* is the execution time, normalized to the reference machine, for the *i*th program of a total of *n* in the workload.

$$\frac{\text{Geometric mean}(X_i)}{\text{Geometric mean}(Y_i)} = \text{Geometric mean}\left(\frac{X_i}{Y_i}\right)$$

1.6 Quantitative Principles

一、Make the Common Case Fast

- *If a design trade-off is necessary, favor the frequent case (which is often simpler) over the infrequent case.*
- *Perhaps it is the most important and pervasive principle of computer design.*
  - *For example, given that overflow in addition is infrequent, favor optimizing the case when no overflow occurs.*

Simple is fast! Small is fast!

二、Amdahl's Law

- *The performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used.*

$$\text{Speedup} = \frac{\text{Performance for entire task using the enhancement when possible}}{\text{Performance for entire task without using the enhancement}}$$

Alternatively,

$$\text{Speedup} = \frac{\text{Execution time for entire task without using the enhancement}}{\text{Execution time for entire task using the enhancement when possible}}$$

- ***Fraction*** $_{enhanced}$ ***Always <= 1.***
  - *The fraction of the computation time in the original machine that can be converted to take advantage of the enhancement*
- ***Speedup*** $_{enhanced}$ $\frac{\text{Time of original mode}}{\text{Time of enhanced mode}}$ Always > 1
  - *The improvement gained by the enhanced execution mode; that is, how much faster the task would run if the enhanced mode were used for the entire program*

$$\text{Execution time}_{\text{new}} = \text{Execution time}_{\text{old}} \times \left((1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} \right)$$

The overall speedup is the ratio of the execution times:

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution time}_{\text{old}}}{\text{Execution time}_{\text{new}}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

- **Example**
  - Suppose that we are considering an enhancement to the processor of a server system used for Web serving. The new CPU is 10 times faster on computation in the Web serving application than the original processor. Assuming that the original CPU is busy with computation 40% of the time and is waiting for I/O 60% of the time, what is the overall speedup gained by incorporating the enhancement?

Answer

$$\text{Fraction}_{\text{enhanced}} = 0.4$$
$$\text{Speedup}_{\text{enhanced}} = 10$$

$$\text{Speedup}_{\text{overall}} = \frac{1}{0.6 + \frac{0.4}{10}} = \frac{1}{0.64} \approx 1.56$$

- **Example**
  - A common transformation required in graphics engines is square root. Implementations of floating-point (FP) square root vary significantly in performance, especially among processors designed for graphics. Suppose FP square root (FPSQR) is responsible for 20% of the execution time of a critical graphics benchmark.One proposal is to enhance the FPSQR hardware and speed up this operation by a factor of 10. The other alternative is just to try to make all FP instructions in the graphics processor run faster by a factor of 1.6; FP instructions are responsible for a total of 50% of the execution time for the application. The design team believes that they can make all FP instructions run 1.6 times faster with the same effort as required for the fast square root. Compare these two design alternatives.

Answer We can compare these two alternatives by comparing the speedups:

$$\text{Speedup}_{\text{FPSQR}} = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

$$\text{Speedup}_{\text{FP}} = \frac{1}{(1-0.5)+\frac{0.5}{1.6}} = \frac{1}{0.8125} = 1.23$$

Improving the performance of the FP operations overall is slightly better because of the higher frequency.

三、The CPU Performance Equation

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Clock cycle time} \times \text{Cycles per instruction}$$

CPI

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{Clock cycle time}}{\text{Clock rate}}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

- CPU performance is dependent upon three characteristics:
  - clock cycle (or rate)
  - clock cycles per instruction
  - and instruction count.
- It is difficult to change one parameter in complete isolation from others
  - Because the basic technologies involved in changing each characteristic are interdependent:

- *Clock cycle time*—Hardware technology and organization
- *CPI*—Organization and instruction set architecture
- *Instruction count*—Instruction set architecture and compiler technology

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i \right) \times \text{Clock cycle time}$$

$$\text{CPI} = \frac{\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{\text{IC}_i}{\text{Instruction count}} \times \text{CPI}_i$$

- **Example1.4:** Suppose we have made the following measurements:
  - Frequency of FP = 25%
  - Average CPI of FP = 4.0
  - Average CPI of other instructions = 1.33
  - Frequency of FPSQR= 2%
  - Average CPI of FPSQR = 20

  Assume that the two design alternatives are to decrease the CPI of FPSQR to 2 or to decrease the average CPI of all FP operations to 2.5. Compare these two design alternatives using the CPU performance equation.

Answer First, observe that only the CPI changes; the clock rate and instruction count remain identical. We start by finding the original CPI with neither enhancement:

$$\text{CPI}_{\text{original}} = \sum_{i=1}^{n} \text{CPI}_i \times \left(\frac{\text{IC}_i}{\text{Instruction count}} \right)$$
$$= (4 \times 25\%) + (1.33 \times 75\%) = 2.0$$

We can compute the CPI for the enhanced FPSQR by subtracting the cycles saved from the original CPI:

$$\text{CPI}_{\text{with new FPSQR}} = \text{CPI}_{\text{original}} - 2\% \times (\text{CPI}_{\text{old FPSQR}} - \text{CPI}_{\text{of new FPSQR only}})$$
$$= 2.0 - 2\% \times (20 - 2) = 1.64$$

We can compute the CPI for the enhancement of all FP instructions the same way or by summing the FP and non-FP CPIs. Using the latter gives us

$$\text{CPI}_{\text{new FP}} = (75\% \times 1.33) + (25\% \times 2.5) = 1.625$$

Since the CPI of the overall FP enhancement is slightly lower, its performance will be marginally better. Specifically, the speedup for the overall FP enhancement is

$$\text{Speedup}_{\text{new FP}} = \frac{\text{CPU time}_{\text{original}}}{\text{CPU time}_{\text{new FP}}} = \frac{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{original}}}{\text{IC} \times \text{Clock cycle} \times \text{CPI}_{\text{new FP}}}$$
$$= \frac{\text{CPI}_{\text{original}}}{\text{CPI}_{\text{new FP}}} = \frac{2.00}{1.625} = 1.23$$

四、Principle of Locality

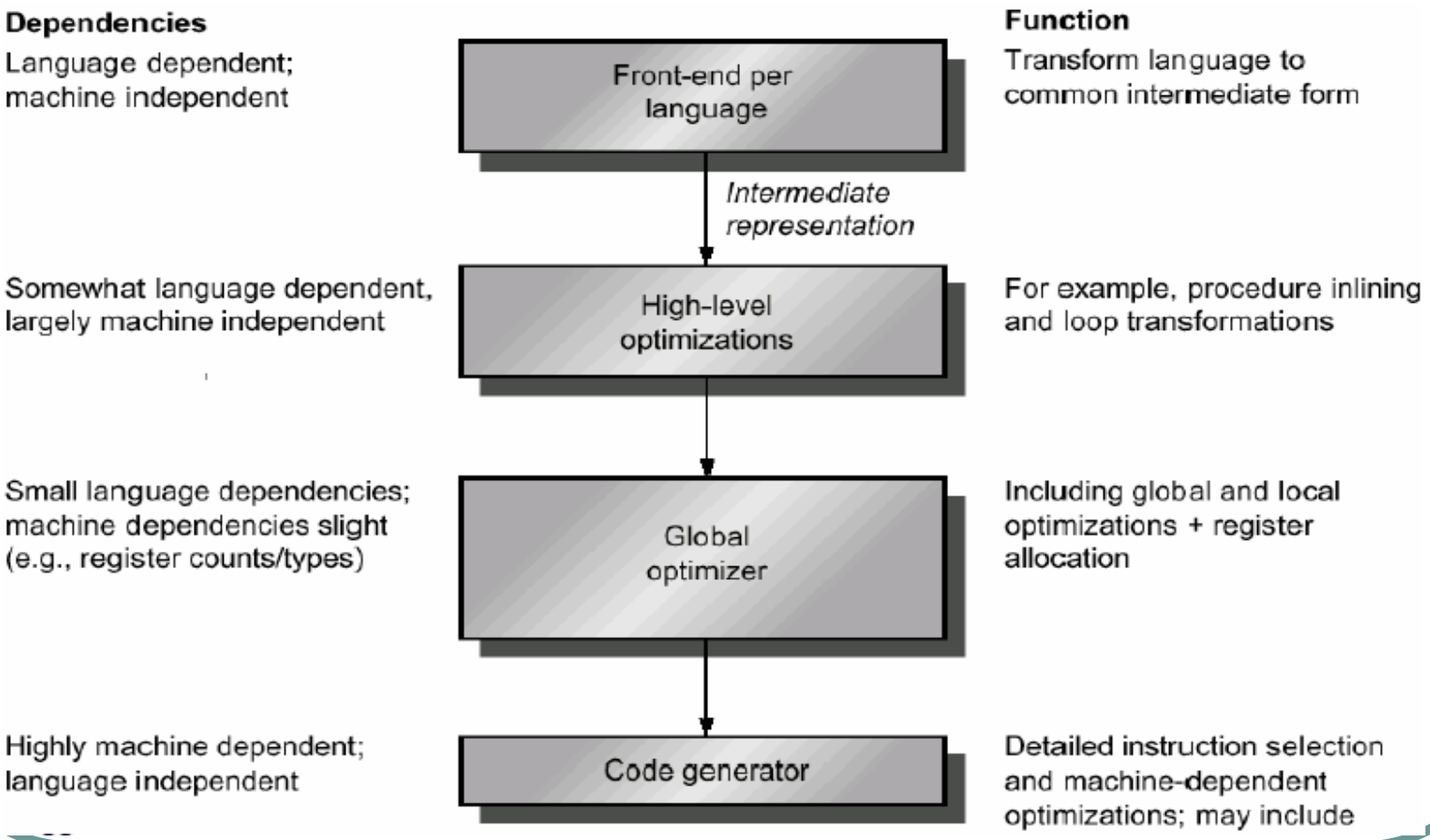
- Programs tend to reuse data and instructions they have used recently.
  - a program spends 90% of its execution time in only 10% of the code.
  - *Temporal locality*
    - states that recently accessed items are likely to be accessed in the near future.
  - *Spatial locality*
    - says that items whose addresses are near one another tend to be referenced close together in time.

五、Take Advantage of Parallelism

- Taking advantage of parallelism is one of the most important methods for improving performance.

1.7 Compilers and Architecture

Typical Compilation



再见，谢谢！