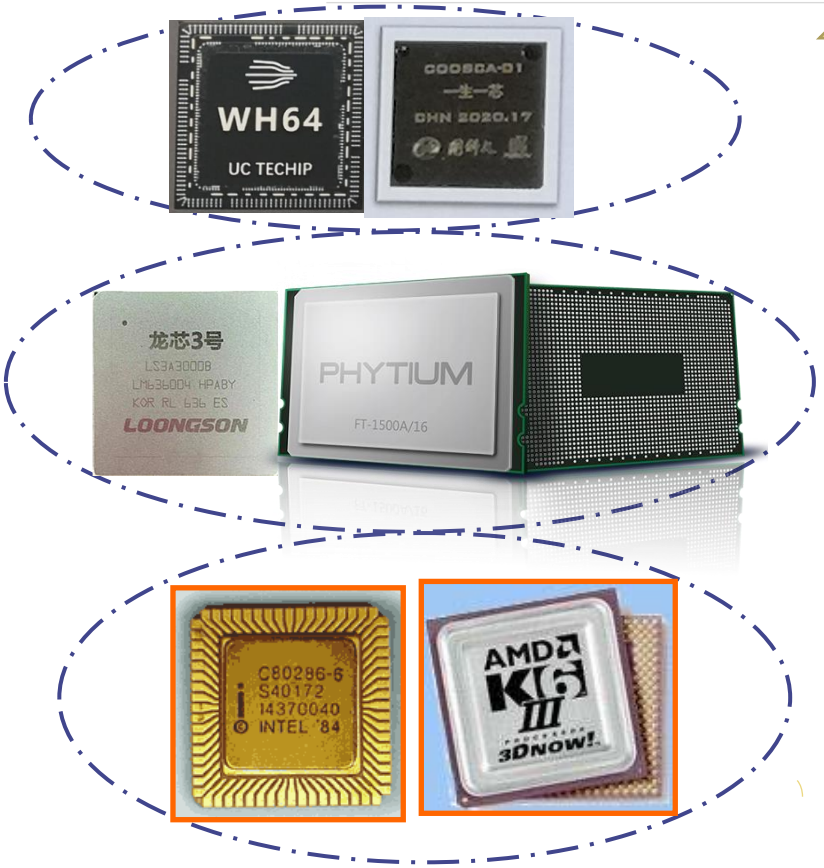
行大道看大势



RISC-V指令系统架构：

- 真正开源时代来临，技术壁垒的鸿沟将会消除
- 专注生态，开发者计划让人人成为开发者
- 合作共赢成为正常现象
- 垄断与霸权会逐步消退

MIPS与ARM指令系统架构：

- 转让架构与版权，标志半开源时代的来临
- 开始专注生态
- 形成短时间合作共利的少数案例
- 但技术壁垒仍难消除

Wintel主体的X86架构：

- 形成封闭体系，构建技术壁垒
- 垄断市场资源和经济利益
- 主要集中在应用层面利用接口进行开发
- 能实现技术屏蔽与封锁

计算机系统结构及课程实验

- 同济大学立项的重点建设课程
- 多项教育部产学协同育人项目支持的课程

同济大学本科生院

同济本内〔2020〕6 号

关于公布 2020 年度同济大学重点课程建设项目立项名单的通知

各学院：

根据《关于开展 2020 年度同济大学重点课程建设项目申报工作的通知》，本科生院协同资产与实验室管理处开展了 2020 年校级重点课程建设项目申报工作。经学院推荐、学校评审等环节，确定了 99 门课程为 2020 年度同济大学重点课程建设项目，现将名单予以公布（见附件）。

本次立项课程的建设周期为 2 年，建设起讫日期为 2020 年 7 月到 2022 年 6 月。各课程团队应按照教育部一流本科课程建设要求和上海高等学校一流本科建设引领计划等文件精神，结合同济大学“培养引领未来的社会栋梁和专业精英”本科人才培养总目标，加强课程综合设计，有序推进适应新时代要求的一流本科课程建设。希望各建设学院夯实教学基层组织，激励教师主动创新、充分投入，进一步提升我校教学创新氛围与教育教学质量。

附件：2020 年同济大学重点课程建设项目立项名单

本科生院

资产与实验室管理处

2020 年 7 月 15 日

计算机系统结构及课程实验

计划安排与成绩考核评定方式

1.本学期根据系的教学改革任务安排, 开设系统结构课程实验.

2.实验任务:两个基本实验.

3.实验成果的认定方式:每个实验的成果资料都要在时间节点内递交上传,且通过比对查重才算有效;如有相同的(代码连续20行相同,报告连续23字相同),均视为抄袭,不合格,且无补交机会.

4.成绩评定:每个同学必须完成两个基本实验，参加期终考试。第一个基本实验占总评10分，第二个基本实验占总评20分，共占30分;考勤占总评10分、平时作业占总评10分；期终考试占总评50分。

5.基本实验：

1)简单的7条MIPS指令的流水线CPU设计，定量分析CPU的性能指标。

2)31条MIPS指令的动态流水线CPU，定量分析CPU的性能指标。

所有实验的报告和源程序都递交到www.univercom.cn:8080。

6.平时考勤与期终考试:占总评成绩70分;其中出勤20分(包括课堂测试点名和作业),期终考试卷面成绩占50分.

7.实验报告是实验环节的重要书面报告，必须包括以下内容:

1、实验名称

2、实验目的

3、实验仪器设备

4、实验内容

5、结果分析和问题讨论与体会

7.时间节点安排:

周次	课次	教学/课程实验内容	实验材料与报告时间节点
1	1(3)	讲座与实验安排课	熟悉已完成的CPU及工具
2	2(1)	计算机系统结构概论	
2	3(3)	计算机系统结构概论	
3	4(3)	存储层次结构.降低CACHE失效率	
4	5(1)	减少CACHE失效的开销,减少命中时间	
4	6(3)	主存	
5	7(3)	主存	
6	8(1)	主存	
6	9(3)	虚拟存储器	

7.时间节点安排:

周次	课次	教学/课程实验内容	实验材料与报告时间节点
7	10(3)	计算机指令集结构,分类与寻址方式	开始第1个基本实验
8	11(1)	指令集结构的功能设计,指令格式设计	第1个基本实验
8	12(3)	流水线技术,重叠执行与先行控制	第1个基本实验
9	13(3)	第1个实验教学辅导(教室现场辅导)	第1个基本实验
10	14(1)	流水线基本概念,性能指标	第1个基本实验
10	15(3)	流水线的实现	本周日24点前递交资料
11	16(3)	向量处理机,指令级并行概念	开始第2个基本实验
12	17(1)	指令的动态调度	第2个基本实验
12	18(3)	动态分支预测	第2个基本实验

7.时间节点安排:

周次	课次	教学/课程实验内容	实验材料与报告时间节点
13	19(3)	第2个实验教学辅导(教室现场辅导)	第2个基本实验
14	20(1)	多指令流出技术,循环展开和指令调度	本周日24点前递交资料
14	21(3)	互联网络基本概念,互联网络结构	
15	22(3)	多处理机	
16	23(1)	复习课	
16	24(3)	自己复习	
17	25(3)	考试	

8.辅导与支持:

秦国锋,陆有军，周涛，冯煊，李文涛等

微信群:

9.参考资料

第1章 计算机系统结构的基本概念

1.1 引 言

1. 第一台通用电子计算机诞生于1946年
2. 计算机技术的飞速发展得益于两个方面
   - 计算机制造技术的发展
   - 计算机系统结构的创新
3. 经历了4个发展过程

时 间	原 因	每年的性能增长
1946年起的25年	两种因素都起着主要的作用	25%
20世纪70年代末—80年代初	大规模集成电路和微处理器出现，以集成电路为代表的制造技术的发展	约35%
20世纪80年代中开始	RISC结构的出现，系统结构不断更新和变革，制造技术不断发展	50%以上 维持了约16年
2002年以来	3个（见下页）	约20%

- 功耗问题（已经很大）。
- 可以进一步有效开发的指令级并行性已经很少。
- 存储器访问速度的提高缓慢。

转向多核的研究和开发,通过单个芯片上实现多个处理器提高系统性能。

系统结构的重大转折：

从单纯依靠指令级并行转向开发线程级并行和数据级并行。

计算机系统结构在计算机的发展中有着极其重要的作用。

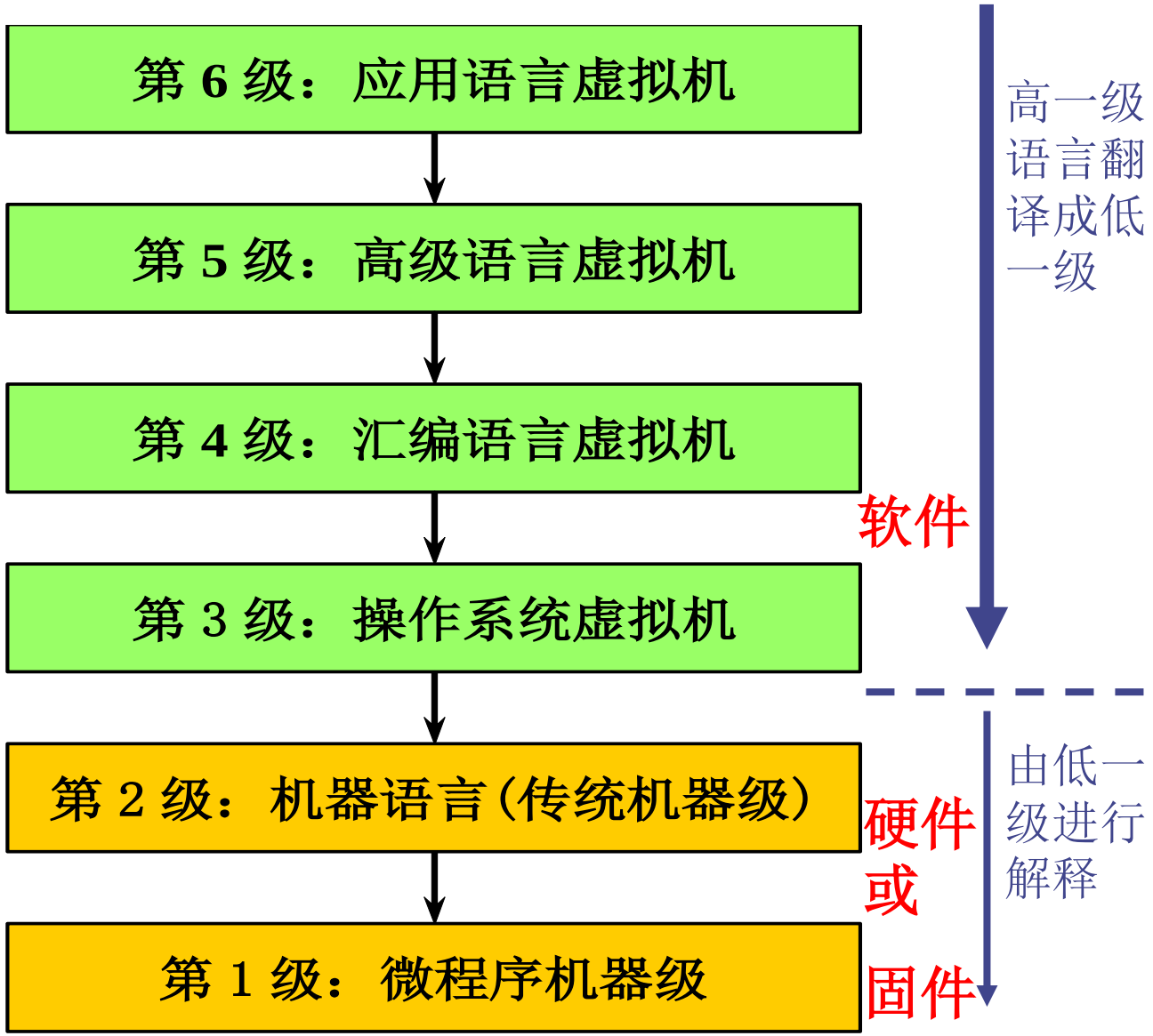
1.2 计算机系统结构的概念

1.2.1 计算机系统的层次结构

1. 计算机系统＝硬件/固件＋软件
2. 计算机语言从低级向高级发展

   高一级语言的语句相对于低一级语言来说功能更强，更便于应用，但又都以低级语言为基础。

3. 从计算机语言的角度，把计算机系统按功能划分成多级层次结构。
   - 每一层以一种语言为特征



某层编程语言的程序员，不关心其程序在计算机中具体执行细节，仅关注逻辑、数据、结果正确。

- 虚拟机：由软件实现的机器
- 语言实现的两种基本技术
  - 翻译：先把$N+1$级程序全部转换成N级程序后，再去执行新产生的N级程序，在执行过程中$N+1$级程序不再被访问。
  - 解释：每当一条$N+1$级指令被译码后，就直接去执行一串等效的N级指令，然后再去取下一条$N+1$级的指令，依此重复进行。

L1~L3用解释方法，L4~L6用翻译方法；解释执行比编译后再执行所花的时间多，但占用的存储空间较少。

1.2.2 计算机系统结构的定义

1. 计算机系统结构的经典定义

   程序员所看到的计算机属性，即概念性结构与功能特性。

2. 按照计算机系统的多级层次结构，不同级程序员所看到的计算机具有不同的属性。

3. 透明性

   在计算机技术中，把这种本来存在的事物或属性，但从某种角度看又好像不存在的概念称为透明性。

4. Amdahl提出的系统结构

   传统机器语言级程序员所看到的计算机属性。

5. 广义的系统结构定义：指令集结构、组成、硬件

   （计算机设计的3个方面）

6. 对于通用寄存器型机器来说，这些属性主要是指：

   - 指令系统

     包括机器指令的操作类型和格式、指令间的排序和控制机构等。

   - 数据表示

     硬件能直接识别和处理的数据类型。

   - 寻址规则

     包括最小寻址单元、寻址方式及其表示。

- 寄存器定义

  （包括各种寄存器的定义、数量和使用方式）

- 中断系统

  （中断的类型和中断响应硬件的功能等）

- 机器工作状态的定义和切换

  （如管态和目态等）

- 存储系统

  （主存容量、程序员可用的最大存储容量等）

- 信息保护

  （包括信息保护方式和硬件对信息保护的支持）

- I/O结构

  包括I/O连结方式、处理机/存储器与I/O设备之间数据传送的方式和格式以及I/O操作的状态等

计算机系统结构概念的实质：

确定计算机系统中软、硬件的界面，界面之上是软件实现的功能，界面之下是硬件和固件实现的功能。

1.2.3 计算机组成和计算机实现

1. 计算机系统结构：计算机系统的软、硬件的界面

   即机器语言程序员所看到的传统机器级所具有的属性。

2. 计算机组成：计算机系统结构的逻辑实现
   - 包含物理机器级中的数据流和控制流的组成以及逻辑设计等。
   - 着眼于：物理机器级内各事件的排序方式与控制方式、各部件的功能以及各部件之间的联系。
3. 计算机实现：计算机组成的物理实现
   - 包括处理机、主存等部件的物理结构，器件的集成度和速度，模块、插件、底板的划分与连接，信号传输，电源、冷却及整机装配技术等。
   - 着眼于：器件技术（起主导作用）、微组装技术。

一种体系结构可以有多种组成。

一种组成可以有多种物理实现。

4. 系列机

由同一厂家生产的具有相同系统结构、但具有不同组成和实现的一系列不同型号的计算机。

例如，IBM公司的IBM 370系列，Intel公司的x86系列等。

1.2.4 计算机系统结构的分类

常见的计算机系统结构分类法有两种：

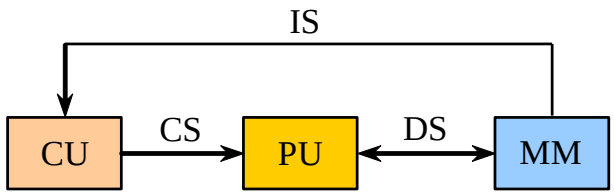
Flynn分类法、冯氏分类法

1. 冯氏分类法
   - 用系统的最大并行度对计算机进行分类。
   - 最大并行度：计算机系统在单位时间内能够处理的最大的二进制位数。

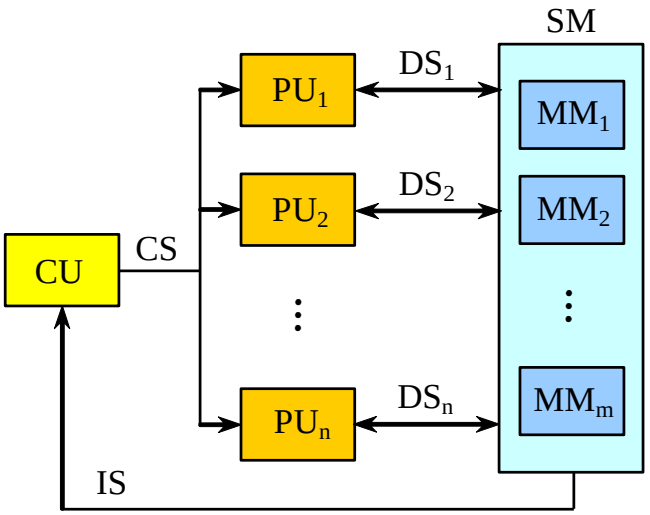
     用平面直角坐标系中的一个点代表一个计算机系统，其横坐标表示字宽（*n*位），纵坐标表示一次能同时处理的字数（*m*字）。*m*×*n*就表示了其最大并行度。
2. Flynn分类法
   - 按照指令流和数据流的多倍性进行分类。
   - 指令流：计算机执行的指令序列。

- 数据流：由指令流调用的数据序列。
- 多倍性：在系统受限的部件上，同时处于同一执行阶段的指令或数据的最大数目。
- Flynn分类法把计算机系统的结构分为4类：
  - 单指令流单数据流(SISD)
  - 单指令流多数据流(SIMD)
  - 多指令流单数据流(MISD)
  - 多指令流多数据流(MIMD)
- 4类计算机的基本结构

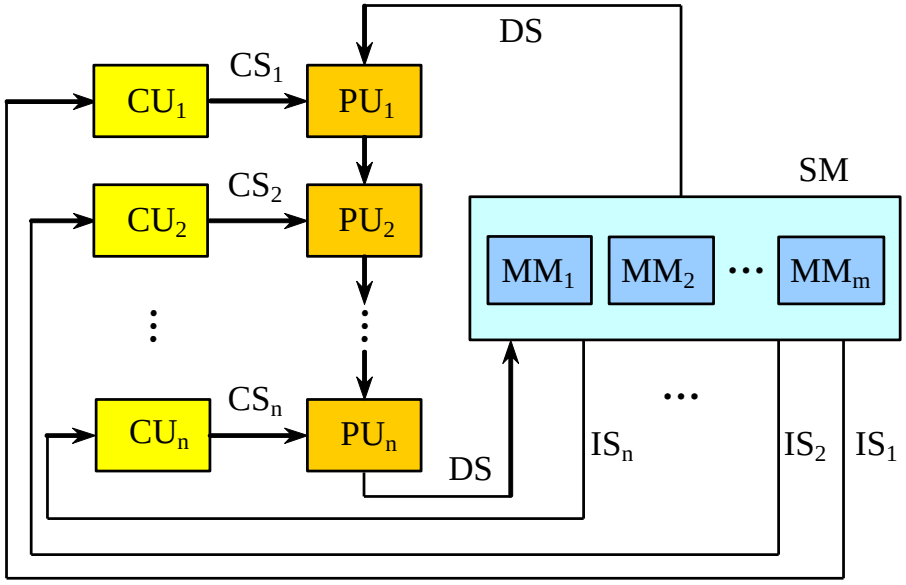
IS：指令流，DS：数据流，CS：控制流，
CU：控制部件，PU：处理部件，MM和SM：存储器。



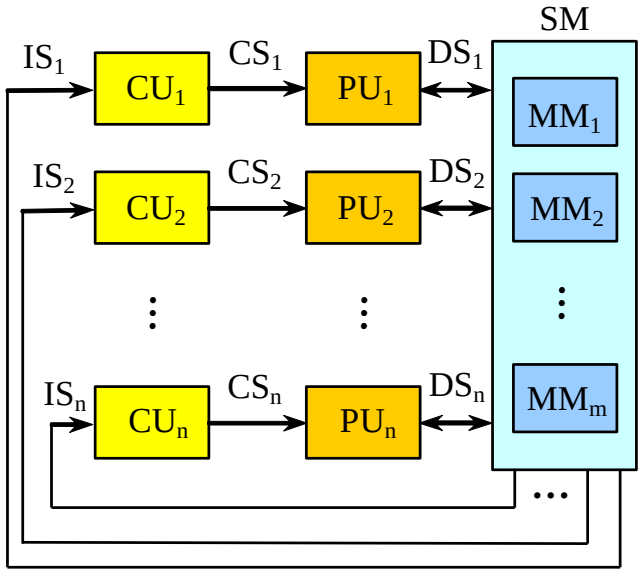
（a）SISD 计算机



（b）SIMD 计算机

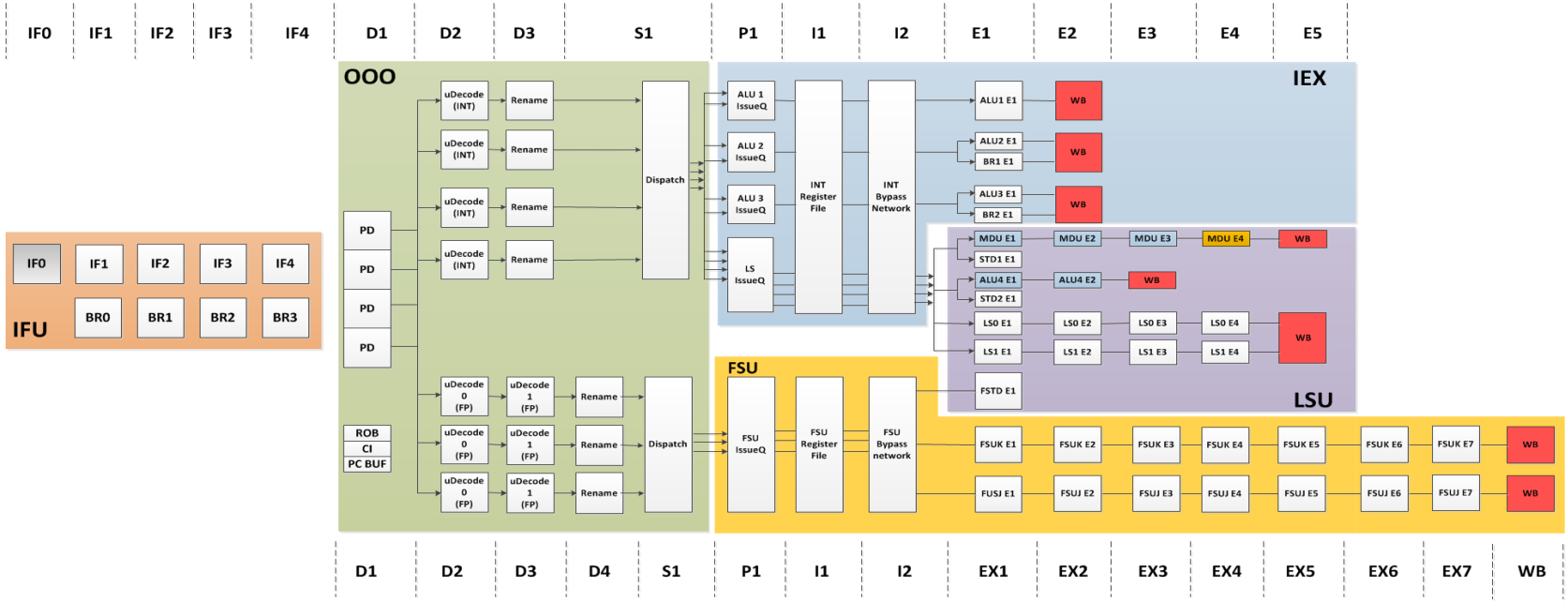


（c）MISD 计算机



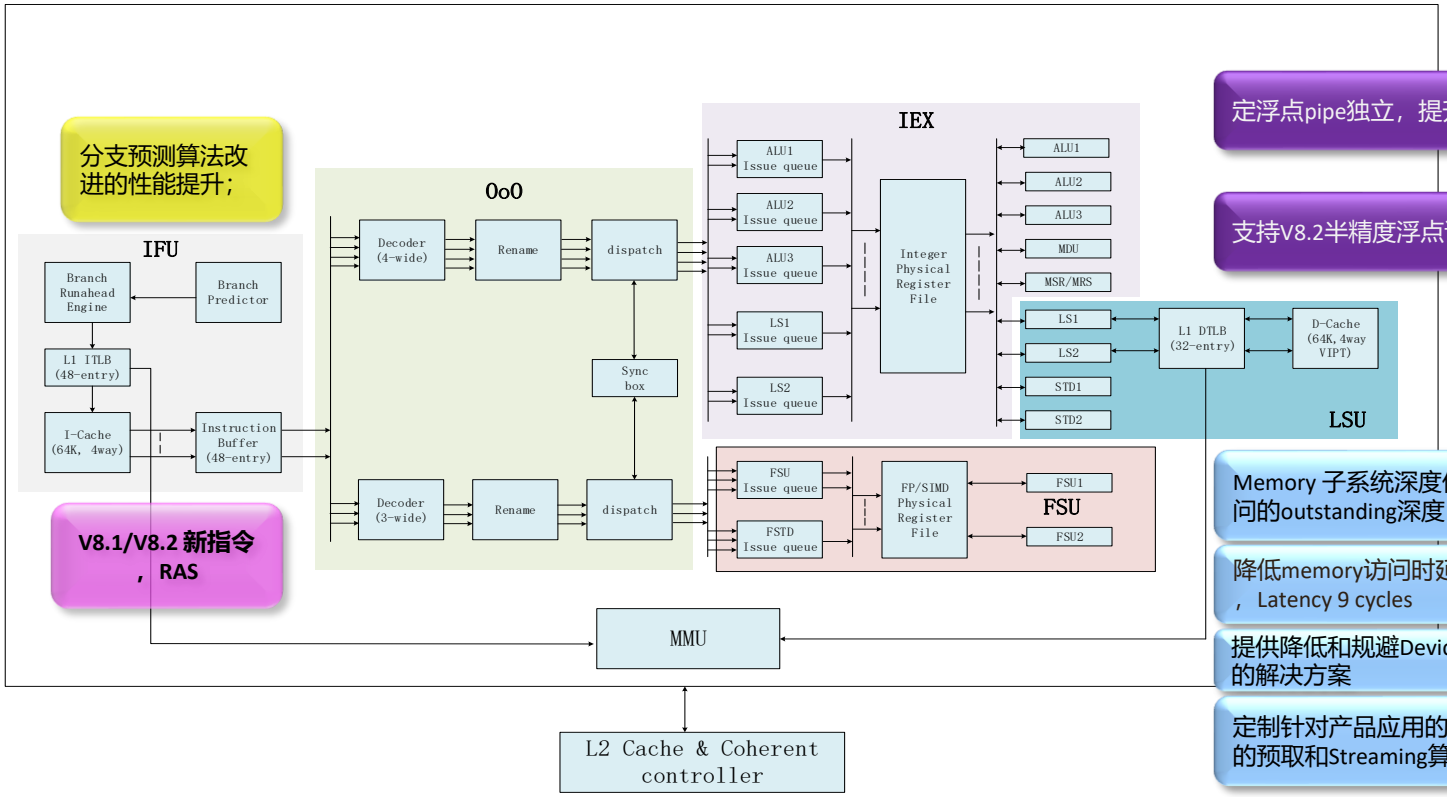
（d）MIMD 计算机

基于ARMv8的鲲鹏流水线技术特点



Taishan coreV110 Pipeline Architecture

基于ARMv8的鲲鹏处理器微架构



定浮点pipe独立，提升dispatch带宽

支持V8.2半精度浮点计算

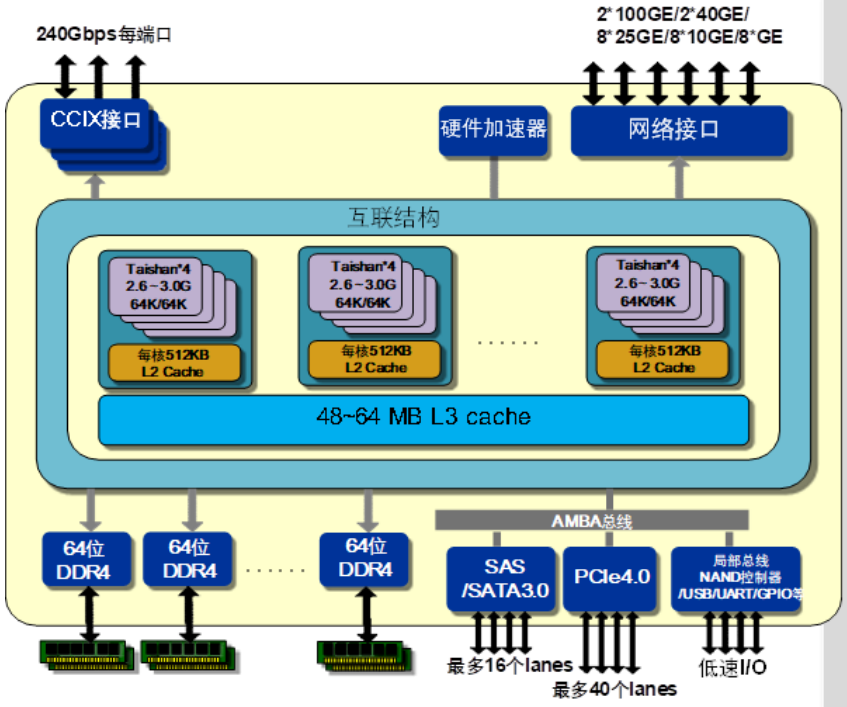
Memory 子系统深度优化，提升内存访问的outstanding深度

降低memory访问时延，每个核私有L2，Latency 9 cycles

提供降低和规避Device访问对性能影响的解决方案

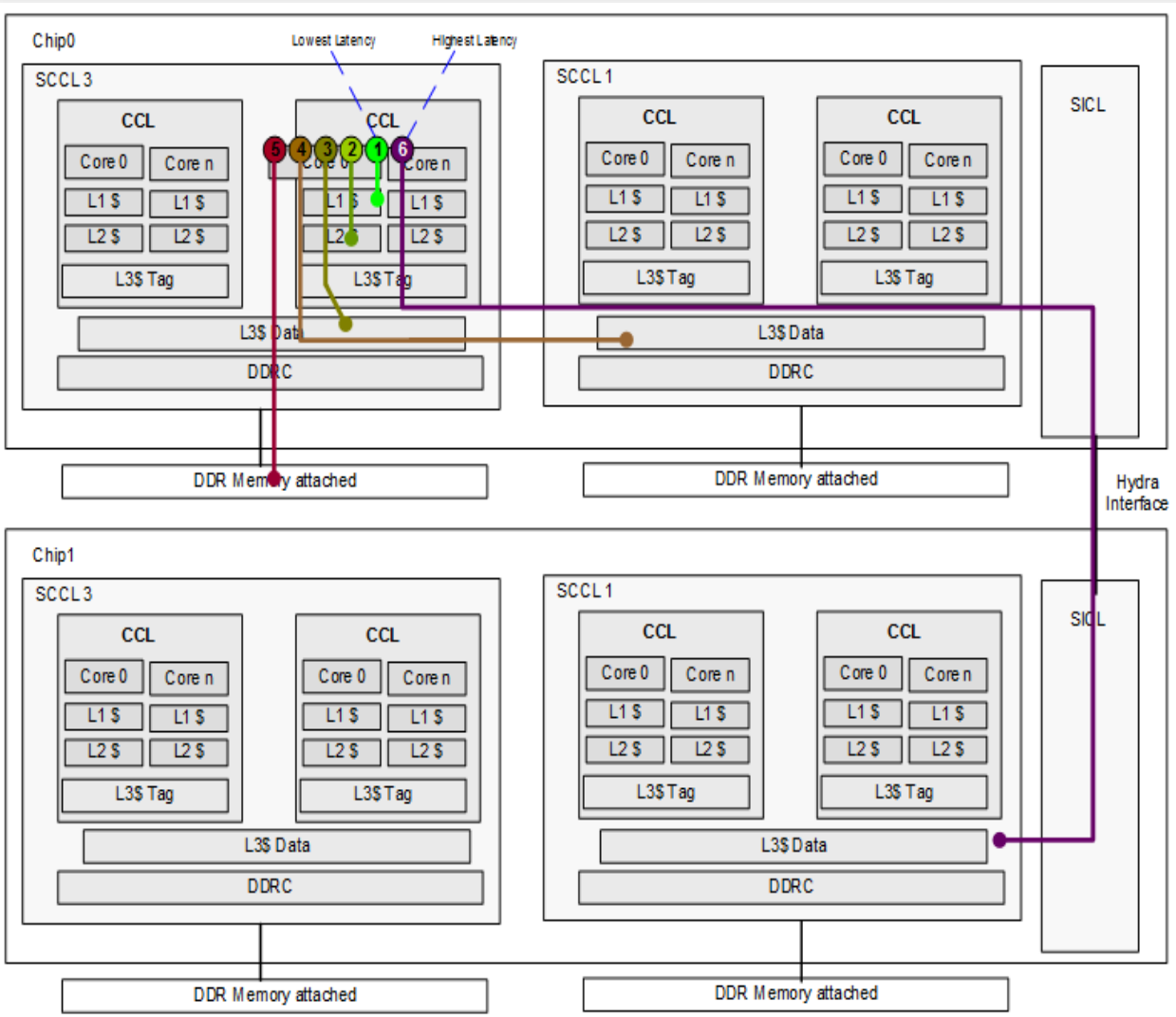
定制针对产品应用的指令和数据Cache的预取和Streaming算法

鲲鹏920系列芯片规格



- 集成最多64×自研核
  - 指令集兼容ARMv8.2, 最高主频达3.0GHz
  - 每核集成64KB L1 I/D缓存
  - 每核独享512KB L2缓存，单芯片共享48-64MB L3缓存
- 8×DDR4控制器@2933MT/s
- 集成PCI-e/SAS接口
  - 支持PCI-e 4.0，向下兼容PCI-e 3.0/2.0/1.0
  - 支持x16,x8,x4,x2,x1 PCI-e 4.0, 集成20 PCI-e控制器
  - 支持16×SAS/SATA 3.0控制器
- 支持CCIX接口，支持加速器的缓存一致性
- 支持2×100G RoCE v2, 支持25GE/50GE/100GE标准NIC
- 支持2P/4P扩展
- 封装大小: 60mm×75mm

鲲鹏920系列芯片架构——内存子系统 (2)



- 一个CPU Die包含4个DDR Channel
- 一个Socket包含2个CPU Die，8个DDR Channel
- 每个控制器支持2DPC 2933
- 本地内存访问均在本地进行，不走片间互联总线，因此访存时延最小，总体性能最好。

时延	ARM CPU Cycles	Skylake Cycles
Register	1	1
L1 cache	4	4
L2 cache	8	14
L3 cache	40	55
DRAM	71 -221(ns)	83-143(ns)

1.3 定量分析技术

1.3.1 计算机系统设计的定量原理

4个定量原理：

1. 以经常性事件为重点

- 对经常发生的情况采用优化方法的原则进行选择，以得到更多的总体上的改进。
- 优化是指分配更多的资源、达到更高的性能或者分配更多的电能等。

2. Amdahl定律

加快某部件执行速度所能获得的系统性能加速比，受限于该部件的执行时间占系统中总执行时间的百分比。

系统性能加速比：

$$\text{加速比} = \frac{\text{系统性能}_{\text{改进后}}}{\text{系统性能}_{\text{改进前}}} = \frac{\text{总执行时间}_{\text{改进前}}}{\text{总执行时间}_{\text{改进后}}}$$

- 加速比依赖于两个因素
  - 可改进比例：在改进前的系统中，可改进部分的执行时间在总的执行时间中所占的比例。

    它总是小于等于1。

    例如：一个需运行60秒的程序中有20秒的运算可以加速，那么这个比例就是20/60。
  - 部件加速比：可改进部分改进以后性能提高的倍数。

    它是改进前所需的执行时间与改进后执行时间的比。

    一般情况下部件加速比是大于1的。

    例如：若系统改进后，可改进部分的执行时间是2秒，而改进前其执行时间为5秒，则部件加速比为5/2。

- 改进后程序的总执行时间

$$\text{总执行时间}_{\text{改进后}} = \text{不可改进部分的执行时间} + \text{可改进部分改进后的执行时间}$$

$$\text{总执行时间}_{\text{改进后}} = (1-\text{可改进比例}) \times \text{总执行时间}_{\text{改进前}} + \frac{\text{可改进比例} \times \text{总执行时间}_{\text{改进前}}}{\text{部件加速比}}$$

$$= \left[(1-\text{可改进比例}) + \frac{\text{可改进比例}}{\text{部件加速比}}\right] \times \text{总执行时间}_{\text{改进前}}$$

系统加速比为改进前与改进后总执行时间之比

$$\text{加速比} = \frac{\text{总执行时间}_{\text{改进前}}}{\text{总执行时间}_{\text{改进后}}}$$

$$= \frac{1}{(1-\text{可改进比例}) + \dfrac{\text{可改进比例}}{\text{部件加速比}}}$$

例1.1 将计算机系统中某一功能的处理速度提高到原来的20倍，但该功能的处理时间仅占整个系统运行时间的40%，则采用此提高性能的方法后，能使整个系统的性能提高多少？

解 由题可知，可改进比例 = 40% = 0.4，

部件加速比 = 20

根据Amdahl定律可知：

$$\text{总加速比} = \frac{1}{0.6 + \frac{0.4}{20}} = 1.613$$

采用此提高性能的方法后，能使整个系统的性能提高到原来的1.613倍。

例1.2 某计算机系统采用浮点运算部件后，使浮点运算速度提高到原来的20倍，而系统运行某一程序的整体性能提高到原来的5倍，试计算该程序中浮点操作所占的比例。

解 由题可知，部件加速比 = 20，系统加速比 = 5

根据Amdahl定律可知

$$5 = \frac{1}{(1 - \text{可改进比例}) + \frac{\text{可改进比例}}{20}}$$

由此可得：可改进比例 = 84.2%

即程序中浮点操作所占的比例为84.2%。

- Amdahl定律：一种性能改进的递减规则
  - 如果仅仅对计算任务中的一部分做性能改进，则改进得越多，所得到的总体性能的提升就越有限。
- 重要推论：如果只针对整个任务的一部分进行改进和优化，那么所获得的加速比不超过

  1/（1－可改进比例）

3. CPU性能公式

- 执行一个程序所需的CPU时间

CPU时间 = 执行程序所需的时钟周期数×时钟周期时间

其中，时钟周期时间是系统时钟频率的倒数。

- 每条指令执行的平均时钟周期数CPI

（Cycles Per Instruction）

CPI = 执行程序所需的时钟周期数 / IC

IC：所执行的指令条数

- 程序执行的CPU时间可以写成

CPU时间 = IC ×CPI ×时钟周期时间

- CPU的性能取决于3个参数
  - 时钟周期时间：取决于硬件实现技术和计算机组成。
  - CPI：取决于计算机组成和指令集结构。
  - IC：取决于指令集结构和编译技术。

```
void insertion_sort(long a[], size_t n)
{
  for (size_t i = 1, j; i < n; i++) {
    long x = a[i];
    for (j = i; j > 0 && a[j-1] > x; j--) {
      a[j] = a[j-1];
    }
    a[j] = x;
  }
}
```

```
# RV32I (19 instructions, 76 bytes, or 52 bytes with RVC)
# a1 is n, a3 points to a[0], a4 is i, a5 is j, a6 is x
   0: 00450693  addi  a3,a0,4    # a3 is pointer to a[i]
   4: 00100713  addi  a4,x0,1    # i = 1
Outer Loop:
   8: 00b76463  bltu  a4,a1,10   # if i < n, jump to Continue Outer loop
Exit Outer Loop:
   c: 00008067  jalr  x0,x1,0    # return from function
Continue Outer Loop:
  10: 0006a803  lw    a6,0(a3)   # x = a[i]
  14: 00068613  addi  a2,a3,0    # a2 is pointer to a[j]
  18: 00070793  addi  a5,a4,0    # j = i
Inner Loop:
  1c: ffc62883  lw    a7,-4(a2)  # a7 = a[j-1]
  20: 01185a63  bge   a6,a7,34   # if a[j-1] <= a[i], jump to Exit Inner Loop
  24: 01162023  sw    a7,0(a2)   # a[j] = a[j-1]
  28: fff78793  addi  a5,a5,-1   # j--
  2c: ffc60613  addi  a2,a2,-4   # decrement a2 to point to a[j]
  30: fe0796e3  bne   a5,x0,1c   # if j != 0, jump to Inner Loop
Exit Inner Loop:
  34: 00279793  slli  a5,a5,0x2  # multiply a5 by 4
  38: 00f507b3  add   a5,a0,a5   # a5 is now byte address oi a[j]
  3c: 0107a023  sw    a6,0(a5)   # a[j] = x
  40: 00170713  addi  a4,a4,1    # i++
  44: 00468693  addi  a3,a3,4    # increment a3 to point to a[i]
  48: fc1ff06f  jal   x0,8       # jump to Outer Loop
```

1.3 定量分析技术

```
# ARM-32 (19 instructions, 76 bytes; or 18 insns/46 bytes with Thumb-2)
# r0 points to a[0], r1 is n, r2 is j, r3 is i, r4 is x
  0: e3a03001 mov  r3, #1              # i = 1
  4: e1530001 cmp  r3, r1              # i vs. n (unnecessary?)
  8: e1a0c000 mov  ip, r0              # ip = a[0]
  c: 212fff1e bxcs lr                  # don't let return address change ISAs
 10: e92d4030 push {r4, r5, lr}        # save r4, r5, return address
Outer Loop:
 14: e5bc4004 ldr  r4, [ip, #4]!       # x = a[i] ; increment ip
 18: e1a02003 mov  r2, r3             # j = i
 1c: e1a0e00c mov  lr, ip             # lr = a[0] (using lr as scratch reg)
Inner Loop:
 20: e51e5004 ldr  r5, [lr, #-4]       # r5 = a[j-1]
 24: e1550004 cmp  r5, r4             # compare a[j-1] vs. x
 28: da000002 ble  38                 # if a[j-1]<=a[i], jump to Exit Inner Loop
 2c: e2522001 subs r2, r2, #1         # j--
 30: e40e5004 str  r5, [lr], #-4      # a[j] = a[j-1]
 34: 1afffff9 bne  20                 # if j != 0, jump to Inner Loop
Exit Inner Loop:
 38: e2833001 add  r3, r3, #1         # i++
 3c: e1530001 cmp  r3, r1             # i vs. n
 40: e7804102 str  r4, [r0, r2, lsl #2] # a[j] = x
 44: 3afffff2 bcc  14                 # if i < n, jump to Outer Loop
 48: e8bd8030 pop  {r4, r5, pc}       # restore r4, r5, and return address
```

```
# MIPS-32 (24 instructions, 96 bytes, or 56 bytes with microMIPS)
# a1 is n, a3 is pointer to a[0], v0 is j, v1 is i, t0 is x
   0: 24860004 addiu a2,a0,4    # a2 is pointer to a[i]
   4: 24030001 li    v1,1       # i = 1
Outer Loop:
   8: 0065102b sltu  v0,v1,a1   # set on i < n
   c: 14400003 bnez  v0,1c      # if i<n, jump to Continue Outer Loop
  10: 00c03825 move  a3,a2      # a3 is pointer to a[j] (slot filled)
  14: 03e00008 jr    ra         # return from function
  18: 00000000 nop              # branch delay slot unfilled
Continue Outer Loop:
  1c: 8cc80000 lw    t0,0(a2)   # x = a[i]
  20: 00601025 move  v0,v1      # j = i
Inner Loop:
  24: 8ce9fffc lw    t1,-4(a3)  # t1 = a[j-1]
  28: 00000000 nop              # load delay slot unfilled
  2c: 0109502a slt   t2,t0,t1   # set a[i] < a[j-1]
  30: 11400005 beqz  t2,48      # if a[j-1]<=a[i], jump to Exit Inner Loop
  34: 00000000 nop              # branch delay slot unfilled
  38: 2442ffff addiu v0,v0,-1   # j--
  3c: ace90000 sw    t1,0(a3)   # a[j] = a[j-1]
  40: 1440fff8 bnez  v0,24      # if j != 0, jump to Inner Loop
  44: 24e7fffc addiu a3,a3,-4   # decr. a2 to point to a[j] (slot filled)
Exit Inner Loop:
  48: 00021080 sll   v0,v0,0x2 #
  4c: 00821021 addu  v0,a0,v0   # v0 now byte address oi a[j]
  50: ac480000 sw    t0,0(v0)   # a[j] = x
  54: 24630001 addiu v1,v1,1    # i++
  58: 1000ffeb b     8          # jump to Outer Loop
  5c: 24c60004 addiu a2,a2,4    # incr. a2 to point to a[i] (slot filled)
```

```
# x86-32 (20 instructions, 45 bytes)
# eax is j, ecx is x, edx is i
# pointer to a[0] is in memory at address esp+0xc, n is in memory at esp+0x10
  0: 56             push esi                # save esi on stack (esi needed below)
  1: 53             push ebx                # save ebx on stack (ebx needed below)
  2: ba 01 00 00 00 mov  edx,0x1            # i = 1
  7: 8b 4c 24 0c    mov  ecx,[esp+0xc]      # ecx is pointer to a[0]
Outer Loop:
  b: 3b 54 24 10    cmp  edx,[esp+0x10]     # compare i vs. n
  f: 73 19          jae  2a <Exit Loop>     # if i >= n, jump to Exit Outer Loop
 11: 8b 1c 91       mov  ebx,[ecx+edx*4]    # x = a[i]
 14: 89 d0          mov  eax,edx            # j = i
Inner Loop:
 16: 8b 74 81 fc    mov  esi,[ecx+eax*4-0x4] # esi = a[j-1]
 1a: 39 de          cmp  esi,ebx            # compare a[j-1] vs. x
 1c: 7e 06          jle  24 <Exit Loop>     # if a[j-1]<=a[i],jump Exit Inner Loop
 1e: 89 34 81       mov  [ecx+eax*4],esi    # a[j] = a[j-1]
 21: 48             dec  eax                # j--
 22: 75 f2          jne  16 <Inner Loop>    # if j != 0, jump to Inner Loop
Exit Inner Loop:
 24: 89 1c 81       mov  [ecx+eax*4],ebx    # a[j] = x
 27: 42             inc  edx                # i++
 28: eb e1          jmp  b <Outer Loop>     # jump to Outer Loop
Exit Outer Loop:
 2a: 5b             pop  ebx                # restore old value of ebx from stack
 2b: 5e             pop  esi                # restore old value of esi from stack
 2c: c3             ret                     # return from function
```

ISA	ARM-32	ARM Thumb-2	MIPS-32	microMIPS	x86-32	RV32I	RV32I+RVC
Instructions	19	18	24	24	20	19	19
Bytes	76	46	96	56	45	76	52

- CPU的性能取决于3个参数
  - 时钟周期时间：取决于硬件实现技术和计算机组成。
  - CPI：取决于计算机组成和指令集结构。
  - IC：取决于指令集结构和编译技术。
- 对CPU性能公式进行进一步细化

假设：计算机系统有n种指令；

CPI_i：第i种指令的处理时间；

IC_i：在程序中第i种指令出现的次数；

则

$$\text{CPU时钟周期数} = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

CPU时间 = 执行程序所需的时钟周期数×时钟周期时间

$$= \sum_{i=1}^{n} (CPI_i \times IC_i) \times 时钟周期时间$$

CPI可以表示为

$$CPI = \frac{时钟周期数}{IC} = \frac{\sum_{i=1}^{n} (CPI_i \times IC_i)}{IC} = \sum_{i=1}^{n} (CPI_i \times \frac{IC_i}{IC})$$

其中，(IC_i/IC)反映了第i种指令在程序中所占的比例。

例1.3 考虑条件分支指令的两种不同设计方法：

（1）CPU_A：通过比较指令设置条件码，然后测试条件码进行分支。

（2）CPU_B：在分支指令中包括比较过程。

在这两种CPU中，条件分支指令都占用2个时钟周期，而所有其他指令占用1个时钟周期。对于CPU_A，执行的指令中分支指令占20%；由于每条分支指令之前都需要有比较指令，因此比较指令也占20%。由于CPU_A在分支时不需要比较，因此CPU_B的时钟周期时间是CPU_A的1.25倍。问：哪一个CPU更快？如果CPU_B的时钟周期时间只是CPU_A的1.1倍，哪一个CPU更快呢？

解 我们不考虑所有系统问题，所以可用CPU性能公式。占用2个时钟周期的分支指令占总指令的20%，剩下的指令占用1个时钟周期。所以

$$CPI_A = 0.2 \times 2 + 0.80 \times 1 = 1.2$$

则CPU_A性能为

$$总CPU时间_A = IC_A \times 1.2 \times 时钟周期_A$$

根据假设，有

$$时钟周期_B = 1.25 \times 时钟周期_A$$

在CPU_B中没有独立的比较指令，所以CPU_B的程序量为CPU_A的80%，分支指令的比例为

$$20\%/80\% = 25\%$$

这些分支指令占用2个时钟周期，而剩下的75%的指令占用1个时钟周期，因此

$$CPI_B = 0.25 \times 2 + 0.75 \times 1 = 1.25$$

因为CPU_B不执行比较，故

$$IC_B = 0.8 \times IC_A$$

因此CPU_B性能为

$$\text{总CPU时间}_B = IC_B \times CPI_B \times \text{时钟周期}_B$$
$$= 0.8 \times IC_A \times 1.25 \times (1.25 \times \text{时钟周期}_A)$$
$$= 1.25 \times IC_A \times \text{时钟周期}_A$$

在这些假设之下，尽管CPU_B执行指令条数较少，CPU_A因为有着更短的时钟周期，所以比CPU_B快。

如果CPU_B的时钟周期时间仅仅是CPU_A的1.1倍，则

$$时钟周期_B = 1.10 \times 时钟周期_A$$

CPU_B的性能为

$$总CPU时间_B = IC_B \times CPI_B \times 时钟周期_B$$
$$= 0.8 \times IC_A \times 1.25 \times (1.10 \times 时钟周期_A)$$
$$= 1.10 \times IC_A \times 时钟周期_A$$

因此CPU_B由于执行更少指令条数，比CPU_A运行更快。

4. 程序的局部性原理

程序执行时所访问的存储器地址分布不是随机的，而是相对地簇聚。

- 常用的一个经验规则

  程序执行时间的90%都是在执行程序中10%的代码。

- 程序的时间局部性

  程序即将用到的信息很可能就是目前正在使用的信息。

- 程序的空间局部性

  程序即将用到的信息很可能与目前正在使用的信息在空间上相邻或者临近。

1.3.2 计算机系统的性能评测

1. 执行时间和吞吐率

如何评测一台计算机的性能，与测试者看问题的角度有关。

- 用户关心的是：单个程序的执行时间（执行单个程序所花的时间很少）
- 数据处理中心的管理员关心的是：吞吐率（在单位时间里能够完成的任务很多 ）

假设两台计算机为X和Y，X比Y快的意思是：

对于给定任务，X的执行时间比Y的执行时间少。

X的性能是Y的*n*倍，即

$$\frac{\text{执行时间Y}}{\text{执行时间X}} = n$$

而执行时间与性能成反比，即

$$n = \frac{\text{执行时间Y}}{\text{执行时间X}} = \frac{\frac{1}{\text{性能Y}}}{\frac{1}{\text{性能X}}} = \frac{\text{性能X}}{\text{性能Y}}$$

- 执行时间可以有多种定义：
  - 计算机完成某一任务所花费的全部时间，包括磁盘访问、存储器访问、输入/输出、操作系统开销等。
  - **CPU时间**：CPU执行所给定的程序所花费的时间，不包含I/O等待时间以及运行其他程序的时间。
    - 用户CPU时间：用户程序所耗费的CPU时间。
    - 系统CPU时间：用户程序运行期间操作系统耗费的CPU时间。

2. 基准测试程序

- 用于测试和比较性能的基准测试程序的最佳选择是真实应用程序。

  （例如编译器）

- 以前常采用简化了的程序，例如：
  - 核心测试程序：从真实程序中选出的关键代码段构成的小程序。
  - 小测试程序：简单的只有几十行的小程序。
  - 合成的测试程序：人工合成出来的程序。

  Whetstone与Dhrystone是最流行的合成测试程序。

从测试性能的角度来看，上述测试程序就不可信了。

原因：

- 这些程序比较小，具有片面性；
- 系统结构设计者和编译器的设计者可以“合谋”把他们的计算机面向这些测试程序进行优化设计，使得该计算机显得性能更高。

➢ 性能测试的结果除了和采用什么测试程序有关以外，还和在什么条件下进行测试有关。

➢ 基准测试程序设计者对制造商的要求

- 采用同一种编译器；
- 对同一种语言的程序都采用相同的一组编译标志。

- 一个问题：是否允许修改测试程序的源程序

  三种不同的处理方法：

  - 不允许修改。
  - 允许修改，但因测试程序很复杂或者很大，几乎是无法修改。
  - 允许修改，只要保证最后输出的结果相同。

- 基准测试程序套件：由各种不同的真实应用程序构成。

  （能比较全面地反映计算机在各个方面的处理性能 ）

- SPEC系列：最成功和最常见的测试程序套件

  （美国的标准性能评估公司开发）

- 台式计算机的基准测试程序套件可以分为两大类：

  处理器性能测试程序，图形性能测试程序

- SPEC89：用于测试处理器性能。10个程序（4个整数程序，6个浮点程序）
- 演化出了4个版本

  SPEC92：20个程序

  SPEC95：18个程序

  SPEC2000：26个程序

  SPEC CPU2006： 29个程序
- SPEC CPU2006

整数程序12个（CINT2006）

9个是用C写的，3个是用C++写的

浮点程序17个（CFP2006）

6个是用FORTRAN写的，4个是用C++写的，3个是用C写的，4个是用C和FORTRAN混合编写的。

- SPEC测试程序套件中的其他一系列测试程序组件
  - SPECSFS：用于NFS（网络文件系统）文件服务器的测试程序。它不仅测试处理器的性能，而且测试I/O系统的性能。它重点测试吞吐率。
  - SPECWeb：Web服务器测试程序。

- SPECviewperf：用于测试图形系统支持OpenGL库的性能。
- SPECapc：用于测试图形密集型应用的性能。

➢ 事务处理（TP）性能测试基准程序：用于测试计算机在事务处理方面的能力，包括数据库访问和更新等。

- 20世纪80年代中期，一些工程师成立了称为TPC的独立组织。目的是开发用于TP性能测试的真实而又公平的基准程序。
- 先后发布了多个版本：

  TPC-A、TPC-C、TPC-H、TPC-W、TPC-App等

  （主要是用于测试服务器的性能 ）

- 用于测试基于Microsoft公司的Windows系列操作系统平台的测试套件
  - PCMark04：中央处理器测试组、内存测试组、图形芯片测试组、硬盘测试组等。
  - Business Winstone 2004：主要用于测试计算机系统商业应用的综合性能。
  - Multimedia Content Creation Winstone 2004：主要用于测试计算机系统多媒体应用的综合性能。
  - SiSoft Sandra Pro 2004：一套功能强大的系统分析评比工具，拥有超过30种以上的分析与测试模块。

    主要包括：CPU、存储器、I/O接口、I/O设备、主板等。

3. 性能比较

两个程序在A、B、C三台计算机上的执行时间

	A机	B机	C机	W（1）	W（2）	W（3）
程序1	1.00	10.00	20.00	0.50	0.909	0.999
程序2	1000.00	10.00	20.00	0.50	0.091	0.001
加权算术平均值A_m（1）	500.50	10.00	20.00			
加权算术平均值A_m（2）	91.91	10.00	20.00			
加权算术平均值A_m（3）	2.00	10.00	20.00			

如何比较这3台计算机的性能呢？

从该表可以得出：

执行程序1：

- A机的速度是B机的10倍
- A机的速度是C机的20倍
- B机的速度是C机的2倍

执行程序2：

- B机的速度是A机的100倍
- C机的速度是A机的50倍
- B机的速度是C机的2倍

- **总执行时间**：计算机执行所有测试程序的总时间
  - B机执行程序1和程序2的速度是A机的50.05倍
  - C机执行程序1和程序2的速度是A机的25.025倍
  - B机执行程序1和程序2的速度是C机的2倍
- **平均执行时间**：各测试程序执行时间的算术平均值

$$S_m = \frac{1}{n}\sum_{i=1}^{n} T_i$$

其中，T_i：第i个测试程序的执行时间

n：测试程序组中程序的个数

- **加权执行时间**：各测试程序执行时间的加权平均值

$$A_m = \sum_{i=1}^{n} W_i \cdot T_i$$

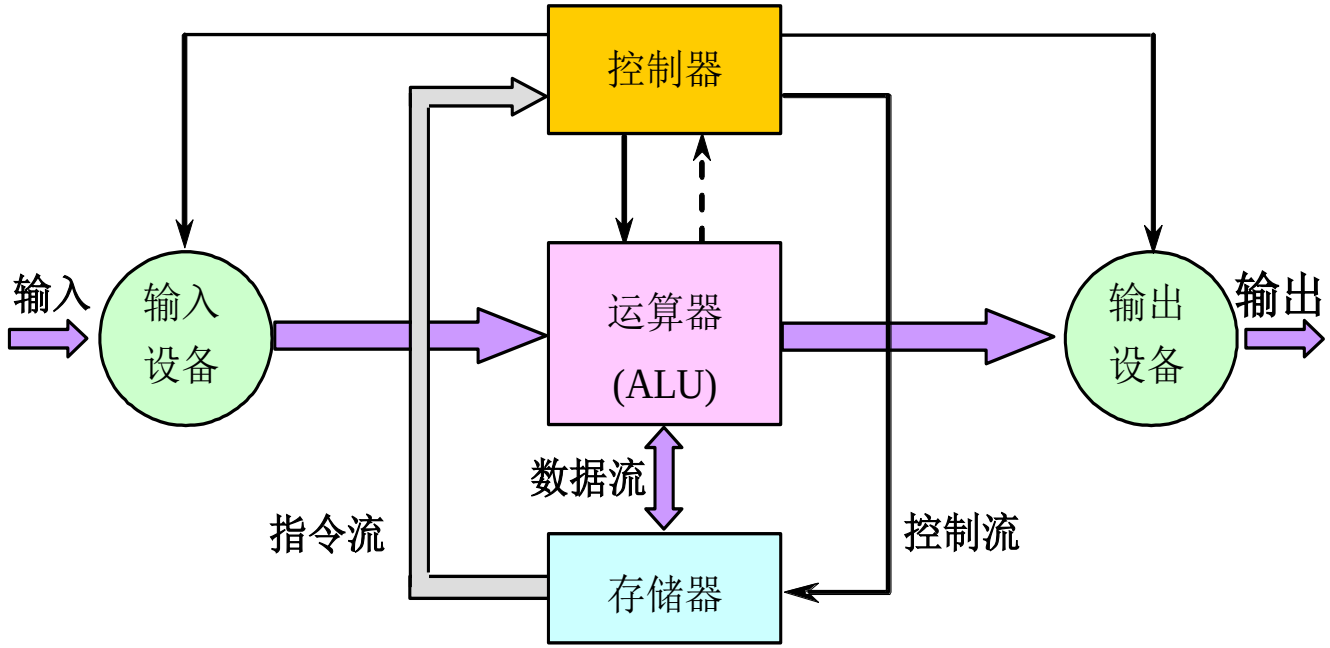
其中，W_i：第i个测试程序在测试程序组中所占的比重

$$\sum_{i=1}^{n} W_i = 1$$

T_i：该程序的执行时间

1.4 计算机系统结构的发展

1.4.1 冯·诺依曼结构



存储程序计算机的结构

1. 存储程序原理的基本点：指令驱动

程序预先存放在计算机存储器中，计算机一旦启动，就能按照程序指定的逻辑顺序执行这些程序，自动完成由程序所描述的处理工作。

2. 冯·诺依曼结构的主要特点

- 以运算器为中心。
- 在存储器中，指令和数据同等对待。

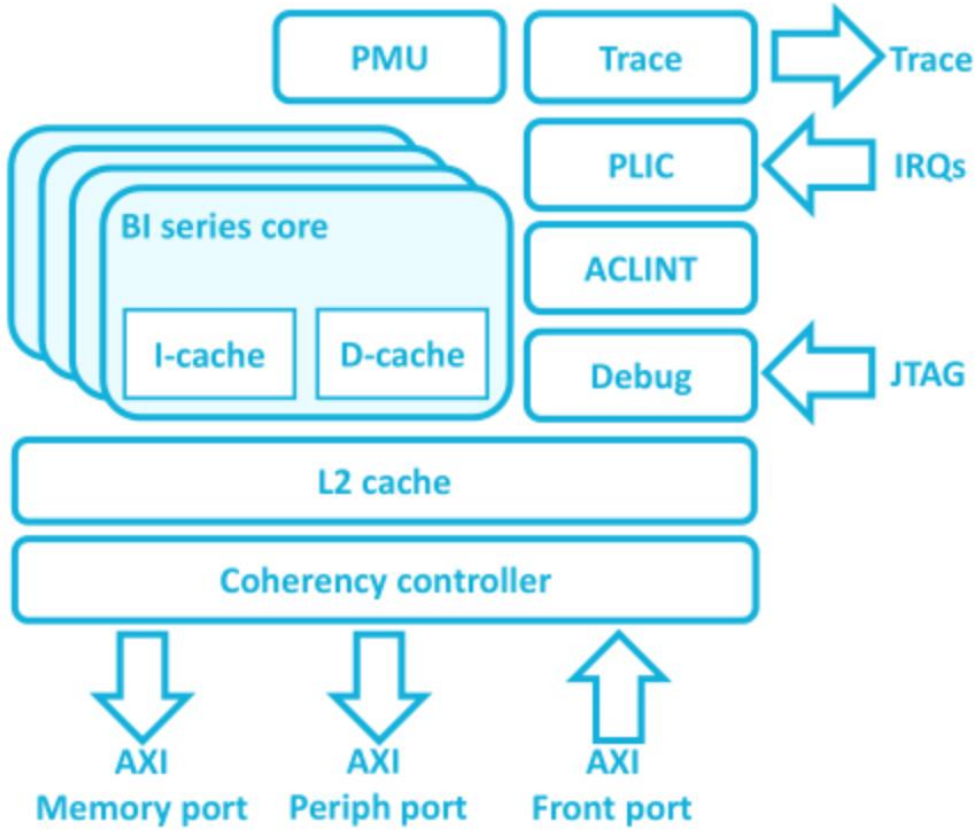
  指令和数据一样可以进行运算，即由指令组成的程序是可以修改的。
- 存储器是按地址访问、按顺序线性编址的一维结构，每个单元的位数是固定的。

- 指令的执行是顺序的。
  - 一般是按照指令在存储器中存放的顺序执行。
  - 程序的分支由转移指令实现。
  - 由指令计数器PC指明当前正在执行的指令在存储器中的地址。
- 指令由操作码和地址码组成。
- 指令和数据均以二进制编码表示，采用二进制运算。

BI series core complex

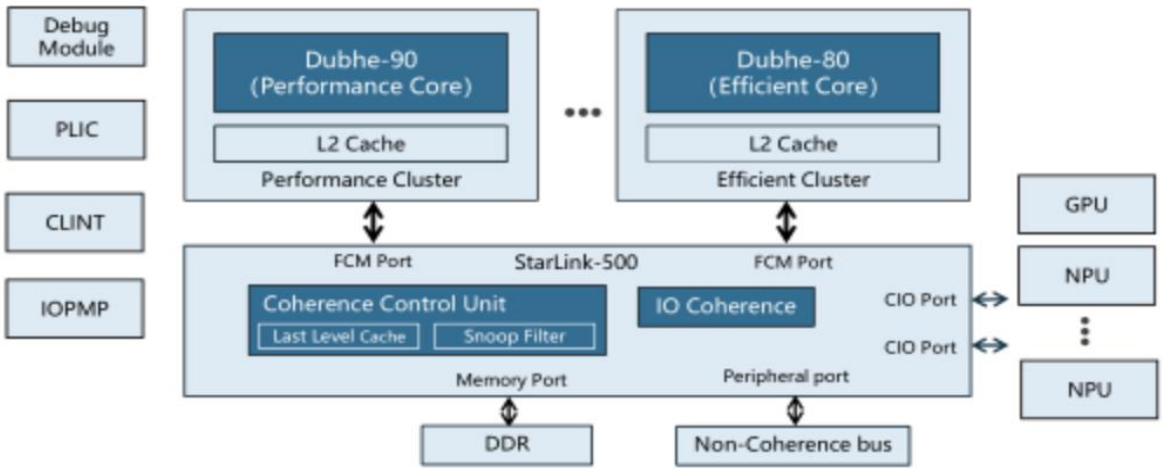
Linux capable application cores

- Multi-core fully coherent configuration
- Machine/User/Supervisor modes
- Full non-blocking caches
- L1 I/D caches
- L2 cache with stride prefetcher
- Debug module
- Power management unit
- PLIC/ACLINT interrupt controllers
- Trace interface (ingress port)
- Coherency controller for maintaining coherency with peripherals and accelerators



13

RISC-V big.LITTLE IP Sub-system



- 高度可配置的全一致性接口，支持 2-8 cores或4个clusters
- 可提供最大性能 specint2k6 75/MHz
- 支持异构系统数据一致性，支持多个IO coherent port
- 集成监听过滤器

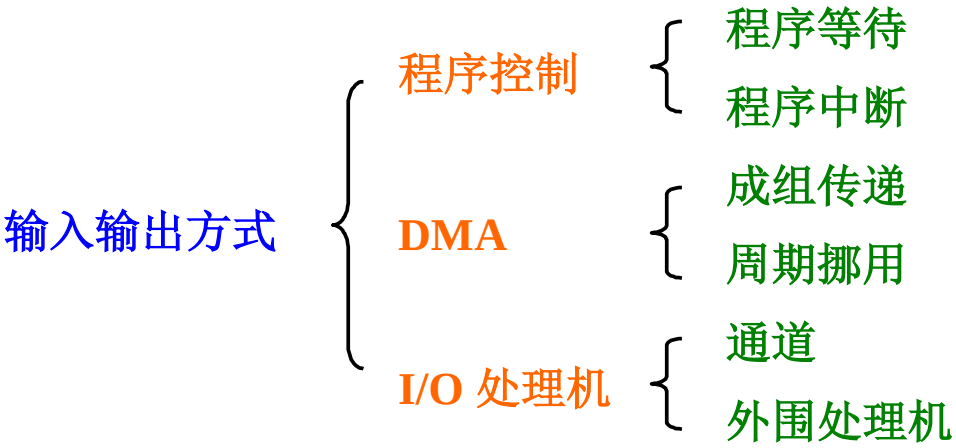
- 根据系统需求灵活配置大小核数量
  - Maximum perf，e.g. Dubhe-90*8
    应用场景：edge server，edge AI，DPU，PC
  - Balance，e.g. Dubhe-90*4+ Dubhe-80*2
    应用场景：Notebook，瘦客户机，NAS
  - Efficient perf ，e.g. Dubhe-80*4
    应用场景：高性能工控机，5G RU， BMC

Shanghai StarFive Technology Co., Ltd.

StarFive 赛昉科技

3. 对系统结构进行的改进

- 输入/输出方式的改进



- 采用并行处理技术
  - 如何挖掘传统机器中的并行性？
  - 在不同的级别采用并行技术。

    例如，微操作级、指令级、线程级、进程级、任务级等。

- 存储器组织结构的发展
  - 相联存储器与相联处理机
  - 通用寄存器组
  - 高速缓冲存储器Cache
- 指令集的发展

两个发展方向：

  - 复杂指令集计算机（CISC）
  - 精减指令集计算机（RISC）

1.4.2 软件对系统结构的影响

- 软件的可移植性：一个软件可以不经修改或者只需少量修改就可以由一台计算机移植到另一台计算机上正确地运行。差别只是执行时间的不同。

  我们称这两台计算机是软件兼容的。

- 实现可移植性的常用方法

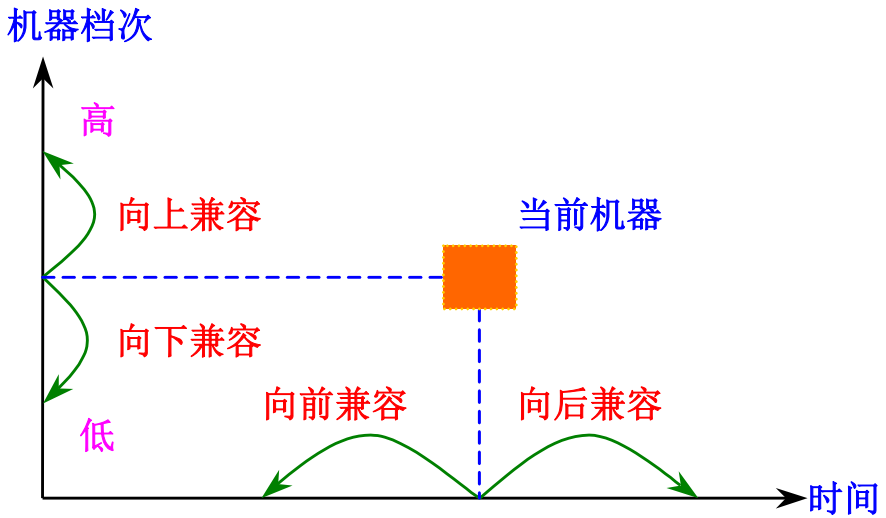
  采用系列机，模拟与仿真，统一高级语言 。

1. 系列机

由同一厂家生产的具有相同的系统结构，但具有不同组成和实现的一系列不同型号的机器。

- 较好地解决软件开发要求系统结构相对稳定与器件、硬件技术迅速发展的矛盾。

➢ 软件兼容



- 向上（下）兼容：按某档机器编制的程序，不加修改就能运行于比它高（低）档的机器。
- 向前（后）兼容：按某个时期投入市场的某种型号机器编制的程序，不加修改地就能运行于在它之前（后）投入市场的机器。

向后兼容是系列机的根本特征。

- 兼容机：由不同公司厂家生产的具有相同系统结构的计算机 。

2. 模拟和仿真

- 使软件能在具有不同系统结构的机器之间相互移植。
  - 在一种系统结构上实现另一种系统结构。
  - 从指令集的角度来看，就是要在一种机器上实现另一种机器的指令集。
- 模拟：用软件的方法在一台现有的机器（称为宿主机）上实现另一台机器（称为虚拟机）的指令集。
  - 通常用解释的方法来实现。
  - 运行速度较慢，性能较差。

- 仿真：用一台现有机器（宿主机）上的微程序去解释实现另一台机器（目标机）的指令集。
  - 运行速度比模拟方法的快
  - 仿真只能在系统结构差距不大的机器之间使用

3. 统一高级语言
   - 实现软件移植的一种理想的方法
   - 较难实现

1.4.3 器件发展对系统结构的影响

1. 摩尔定律

集成电路芯片上所集成的晶体管数目每隔18个月就翻一番。

2. 计算机的分代主要以器件作为划分标准。

- 它们在器件、系统结构和软件技术等方面都有各自的特征。
- SMP：对称式共享存储器多处理机

  MPP：大规模并行处理机

分代	器件特征	结构特征	软件特征	典型实例
第一代 （1945—1954年）	电子管和继电器	存储程序计算机 程序控制I/O	机器语言 汇编语言	普林斯顿ISA， ENIAC，IBM 701
第二代 （1955—1964年）	晶体管、磁芯 印刷电路	浮点数据表示 寻址技术 中断、I/O处理机	高级语言和编译 批处理监控系统	Univac LAPC， CDC 1604， IBM 7030
第三代 （1965—1974年）	SSI和MSI 多层印刷电路 微程序	流水线、Cache 先行处理 系列机	多道程序 分时操作系统	IBM 360/370， CDC 6600/7600， DEC PDP-8
第四代 （1975—1990年）	LSI和VLSI 半导体存储器	向量处理 分布式存储器	并行与分布处理	Cray-1，IBM 3090，DEC VAX 9000，Convax-1
第五代 （1991年—）	高性能微处理器 高密度电路	超标量、超流水 SMP、MP、MPP 机群	大规模、可扩展 并行与分布处理	SGI Cray T3E， IBM SP2，DEC AlphaServer 8400

1.4.4 应用对系统结构的影响

1. 不同的应用对计算机系统结构的设计提出了不同的要求
2. 应用需求是促使计算机系统结构发展的最根本的动力
3. 一些特殊领域：需要高性能的系统结构
   - 高结构化的数值计算

     气象模型、流体动力学、有限元分析
   - 非结构化的数值计算

     蒙特卡洛模拟、稀疏矩阵
   - 实时多因素问题

     语音识别、图像处理、计算机视觉

- 大存储容量和输入输出密集的问题

  数据库系统、事务处理系统

- 图形学和设计问题

  计算机辅助设计

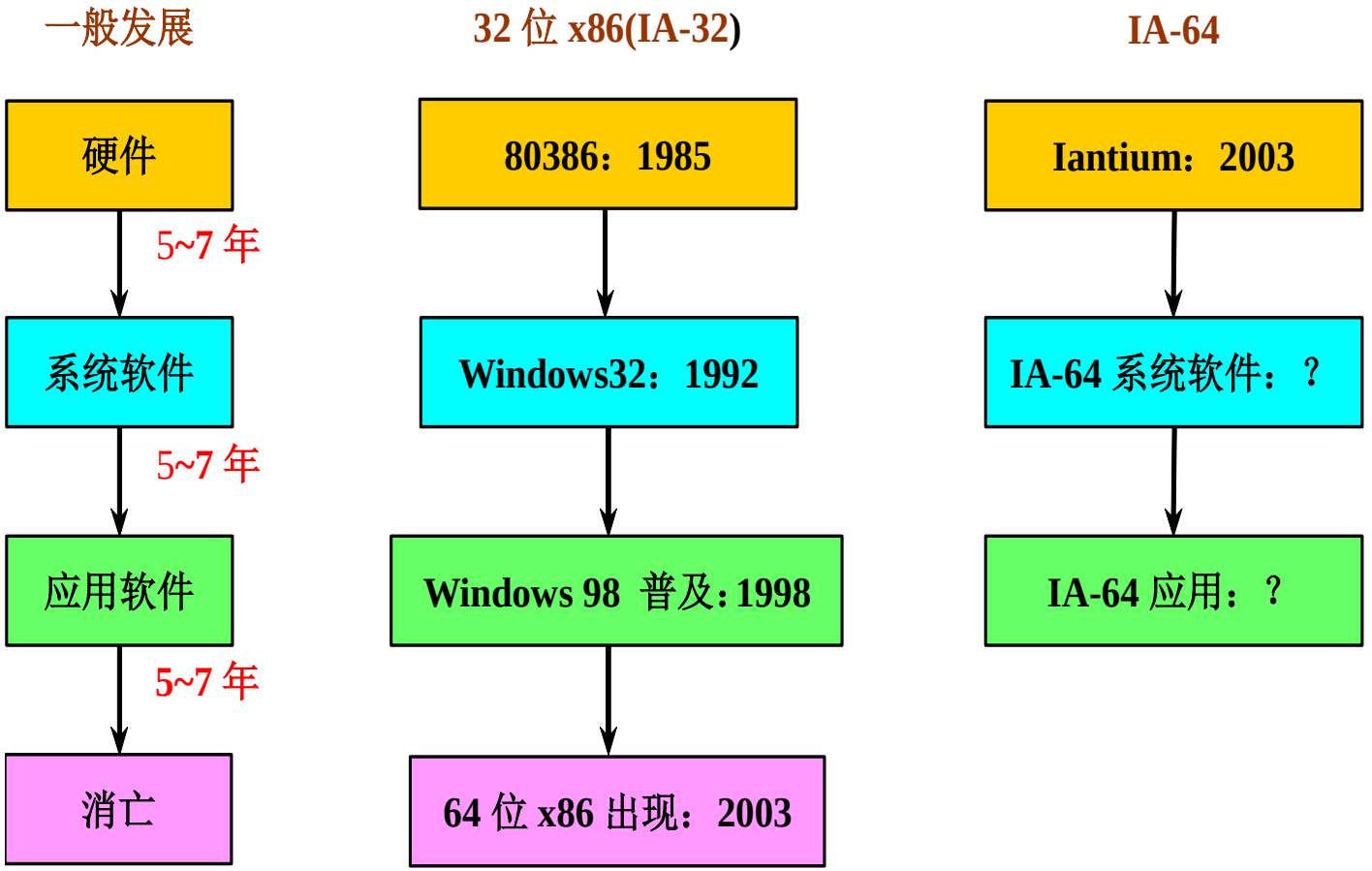
- 人工智能

  面向知识的系统、推理系统等

4. 计算机性能随时间下移到小型和微型通用计算机

1.4.5 系统结构的生命周期

1. 系统结构的生命周期：从诞生、发展、成熟到消亡
   - 从硬件成熟到系统软件成熟大约需要5～7年的时间
   - 从系统软件成熟到应用软件成熟，大约也需要5～7年时间。
   - 再过5～7年的时间，这种系统结构就不会作为主流系统结构存在了。
2. Intel的x86系列微处理器中32位系统结构的发展


一般发展
硬件
5~7 年
系统软件
5~7 年
应用软件
5~7 年
消亡
32 位 x86(IA-32)
80386：1985
Windows32：1992
Windows 98 普及：1998
64 位 x86 出现：2003
IA-64
Iantium：2003
IA-64 系统软件：？
IA-64 应用：？

1.5 计算机系统结构中并行性的发展

1.5.1 并行性的概念

1. 并行性：计算机系统在同一时刻或者同一时间间隔内进行多种运算或操作。

只要在时间上相互重叠，就存在并行性。

- 同时性：两个或两个以上的事件在同一时刻发生。
- 并发性：两个或两个以上的事件在同一时间间隔内发生。

2. 从处理数据的角度来看，并行性等级从低到高可分为：

- 字串位串：每次只对一个字的一位进行处理。

  最基本的串行处理方式，不存在并行性。

- 字串位并：同时对一个字的全部位进行处理，不同字之间是串行的。

  开始出现并行性。

- 字并位串：同时对许多字的同一位（称为位片）进行处理。

  具有较高的并行性。

- 全并行：同时对许多字的全部位或部分位进行处理。

  最高一级的并行。

3. 从执行程序的角度来看，并行性等级从低到高可分为：

- 指令内部并行：单条指令中各微操作之间的并行。
- 指令级并行：并行执行两条或两条以上的指令。
- 线程级并行：并行执行两个或两个以上的线程。

  通常是以一个进程内派生的多个线程为调度单位。
- 任务级或过程级并行：并行执行两个或两个以上的过程或任务（程序段）

  以子程序或进程为调度单元。
- 作业或程序级并行：并行执行两个或两个以上的作业或程序。

1.5.2 提高并行性的技术途径

三种途径：

1. 时间重叠

引入时间因素，让多个处理过程在时间上相互错开，轮流重叠地使用同一套硬件设备的各个部分，以加快硬件周转而赢得速度。

2. 资源重复

引入空间因素，以数量取胜。通过重复设置硬件资源，大幅度地提高计算机系统的性能。

现代计算机系统综合运用时间重叠和资源重复两个措施。

3. 资源共享

这是一种软件方法，它使多个任务按一定时间顺序轮流使用同一套硬件设备。 如:多道程序,分时系统。

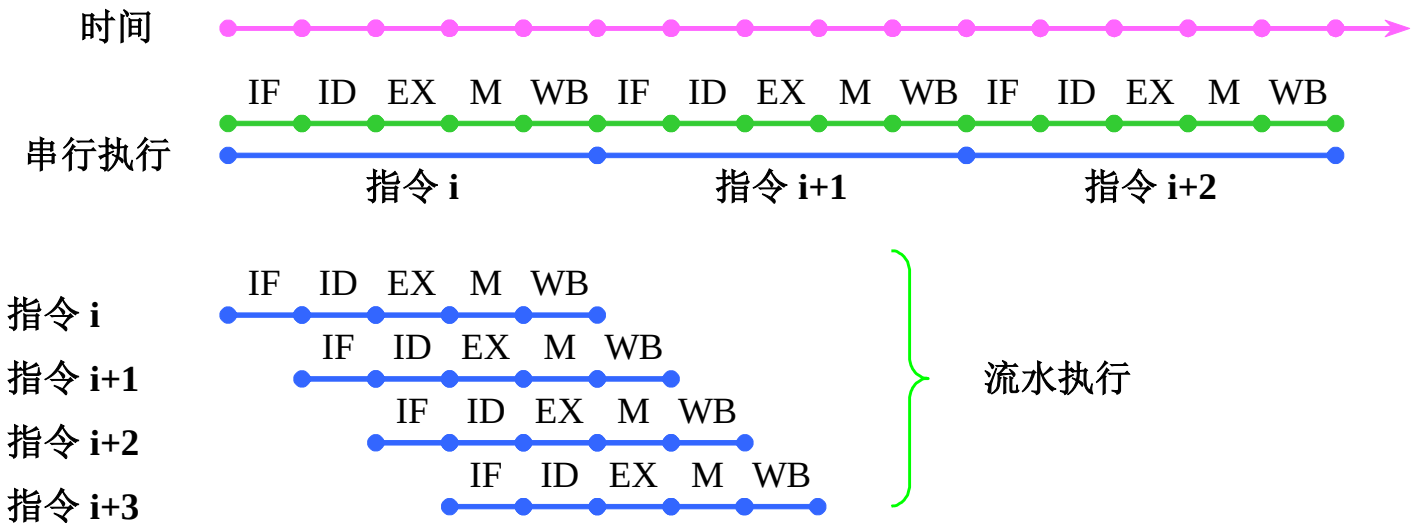
1.5.3 单机系统中并行性的发展

1. 在发展高性能单处理机过程中，起主导作用的是时间重叠原理。

实现时间重叠的基础：部件功能专用化

- 把一件工作按功能分割为若干相互联系的部分；
- 把每一部分指定给专门的部件完成；
- 按时间重叠原理把各部分的执行过程在时间上重叠起来，使所有部件依次分工完成一组同样的工作。

例如： 对于解释指令的5个过程，就分别需要5个专用的部件：取指令部件(IF)、指令译码部件(ID)、指令执行部件(EX)、访问存储器部件(M)和写结果部件(WB)。



2. 在单处理机中，资源重复原理的运用也已经十分普遍。
   - 多体存储器
   - 多操作部件
     - 通用部件被分解成若干个专用部件，如加法部件、乘法部件、除法部件、逻辑运算部件等，而且同一种部件也可以重复设置多个。
     - 只要指令所需的操作部件空闲，就可以开始执行这条指令（如果操作数已准备好的话）。
     - 这实现了指令级并行。

- 阵列处理机（并行处理机）

  更进一步，设置许多相同的处理单元，让它们在同一个控制器的指挥下，按照同一条指令的要求，对向量或数组的各元素同时进行同一操作，就形成了阵列处理机。

3. 在单处理机中，资源共享的概念实质上是用单处理机模拟多处理机的功能，形成所谓虚拟机的概念。

   - 分时系统

1.5.4 多机系统中并行性的发展

1. 多机系统遵循时间重叠、资源重复、资源共享原理，发展为3种不同的多处理机：

   同构型多处理机、异构型多处理机、分布式系统

2. 耦合度

   反映多机系统中各机器之间物理连接的紧密程度和交互作用能力的强弱。

   - 紧密耦合系统（直接耦合系统）：在这种系统中，计算机之间的物理连接的频带较高，一般是

通过总线或高速开关互连，可以共享主存。

- 松散耦合系统（间接耦合系统）：一般是通过通道或通信线路实现计算机之间的互连，可以共享外存设备（磁盘、磁带等）。机器之间的相互作用是在文件或数据集一级上进行的。

  表现为两种形式：

  - 多台计算机和共享的外存设备连接，不同机器之间实现功能上的分工（功能专用化），机器处理的结果以文件或数据集的形式送到共享外存设备，供其他机器继续处理。
  - 计算机网络，通过通信线路连接，实现更大范围的资源共享。

3. 功能专用化（实现时间重叠）

- 专用外围处理机

  例如，输入/输出功能的分离。

- 专用处理机

  如数组运算、高级语言翻译、数据库管理等，分离出来。

- 异构型多处理机系统

  由多个不同类型、至少担负不同功能的处理机组成，它们按照作业要求的顺序，利用时间重叠原理，依次对它们的多个任务进行加工，各自完成规定的功能动作。

4. 机间互连

- 容错系统
- 可重构系统

对计算机之间互连网络的性能提出了更高的要求。高带宽、低延迟、低开销的机间互连网络是高效实现程序或任务一级并行处理的前提条件。

- 同构型多处理机系统

由多个同类型或至少担负同等功能的处理机组成，它们同时处理同一作业中能并行执行的多个任务。