计算机系统结构

第十课：复习

王韬
http://ceca.pku.edu.cn/wangtao

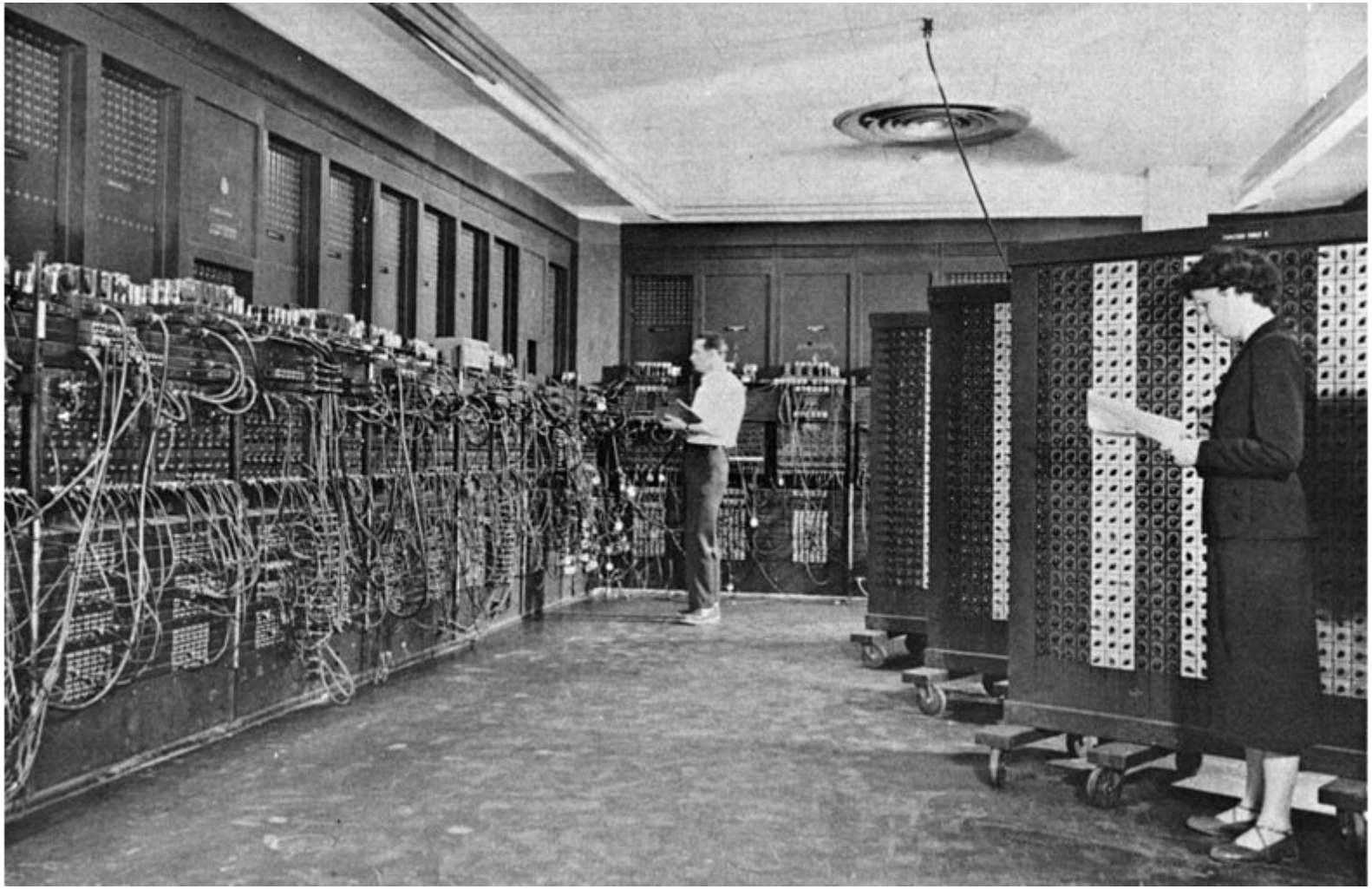
2013

基础知识

四代计算机系统

- 第一代：电子管计算机（**1945-55**）
- 第二代：晶体管计算机（**1955-65**）
- 第三代：集成电路计算机（**1965-1980**）
- 第四代：个人计算机（**1980-**现在）

第一代：电子管计算机



Glen Beck (background) and Betty Snyder (foreground) program ENIAC in BRL building 328. (U.S. Army photo)

第二代：晶体管计算机 1/2

- IBM 7090

第二代：晶体管计算机 2/2

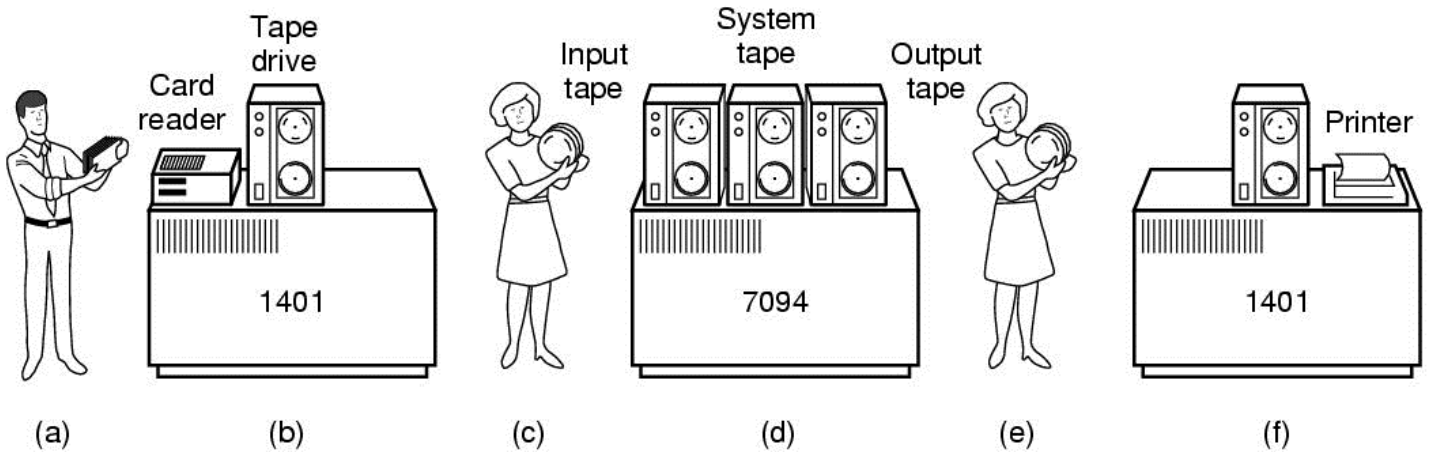


Figure 1-3. An early batch system.

- (a) Programmers bring cards to 1401
- (b) 1401 reads batch of jobs onto tape
- (c) Operator carries input tape to 7094
- (d) 7094 does computing.
- (e) Operator carries output tape to 1401
- (f) 1401 prints output

第三代：集成电路计算机 1/2

IBM System/360

SPOOLing

Time-sharing

MULTICS

Mini-computer

PDP-1, 7, 11,…

Ken Thompson

UNIX

……

- The PDP-1 (Programmed Data Processor-1, DEC) was first produced in 1960. It was also the original hardware for playing history's first game on a minicomputer, Steve Russell's Spacewar!. – Source: Wikipedia

第三代：集成电路计算机 2/2

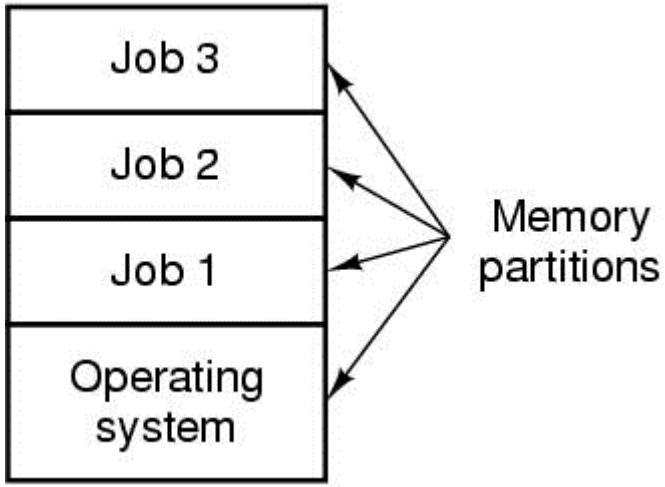


Figure 1-5. A multiprogramming system with three jobs in memory.

第四代：个人计算机 1/2

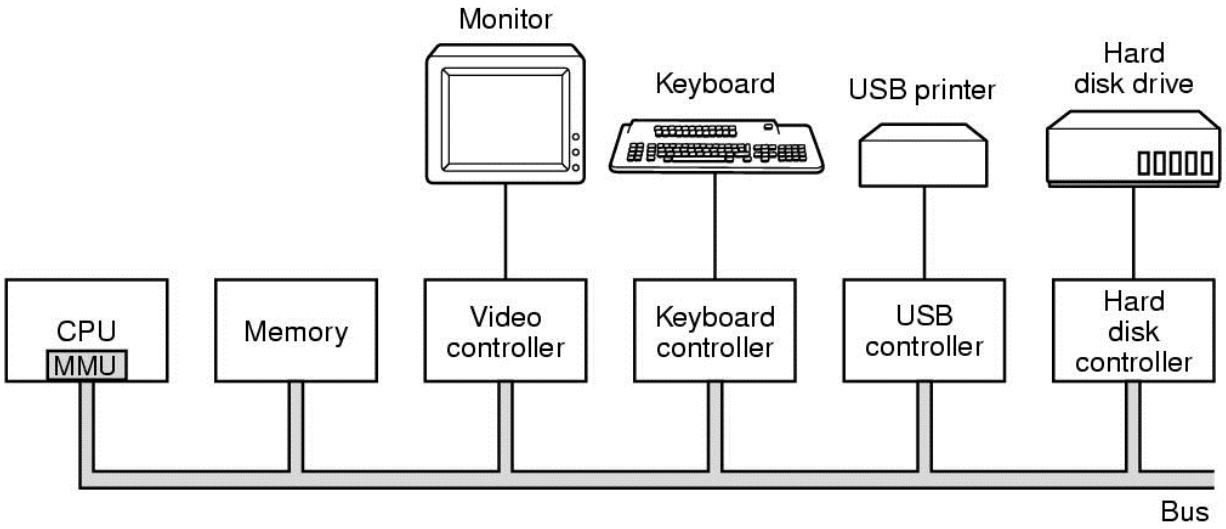


Figure 1-6. Some of the components of a simple personal computer.

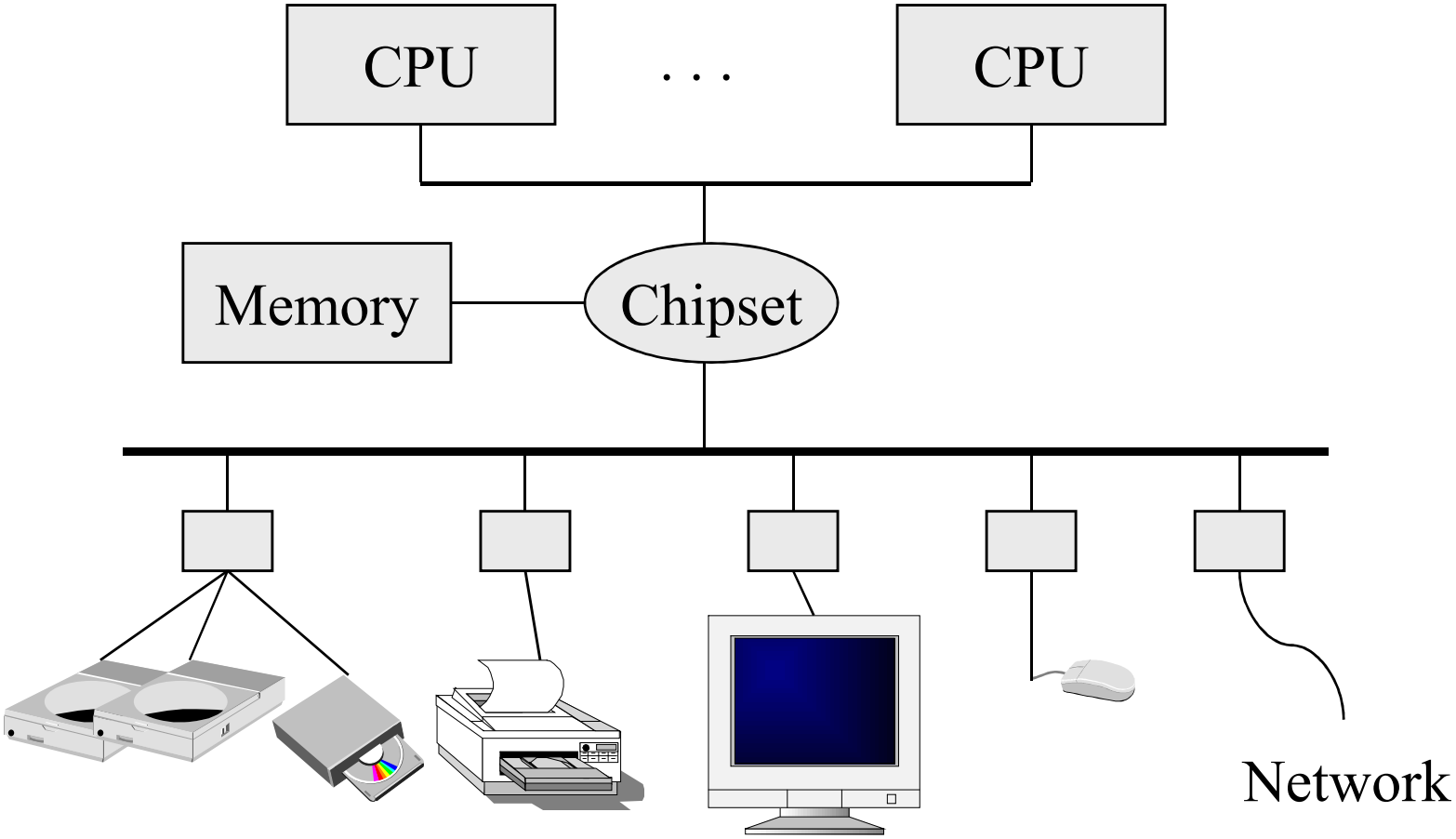
第四代：个人计算机 2/2

- Intel 8080, 8086, 8088, 80286, 80386, 80486, Pentium, Core, …
- Motorola 6502, 68000, …
- ARM, …
- IBM PC, Apple II, smart phone, …
- CP/M, DOS, Windows, Unix/Linux, Mac OS, iOS, Android, …

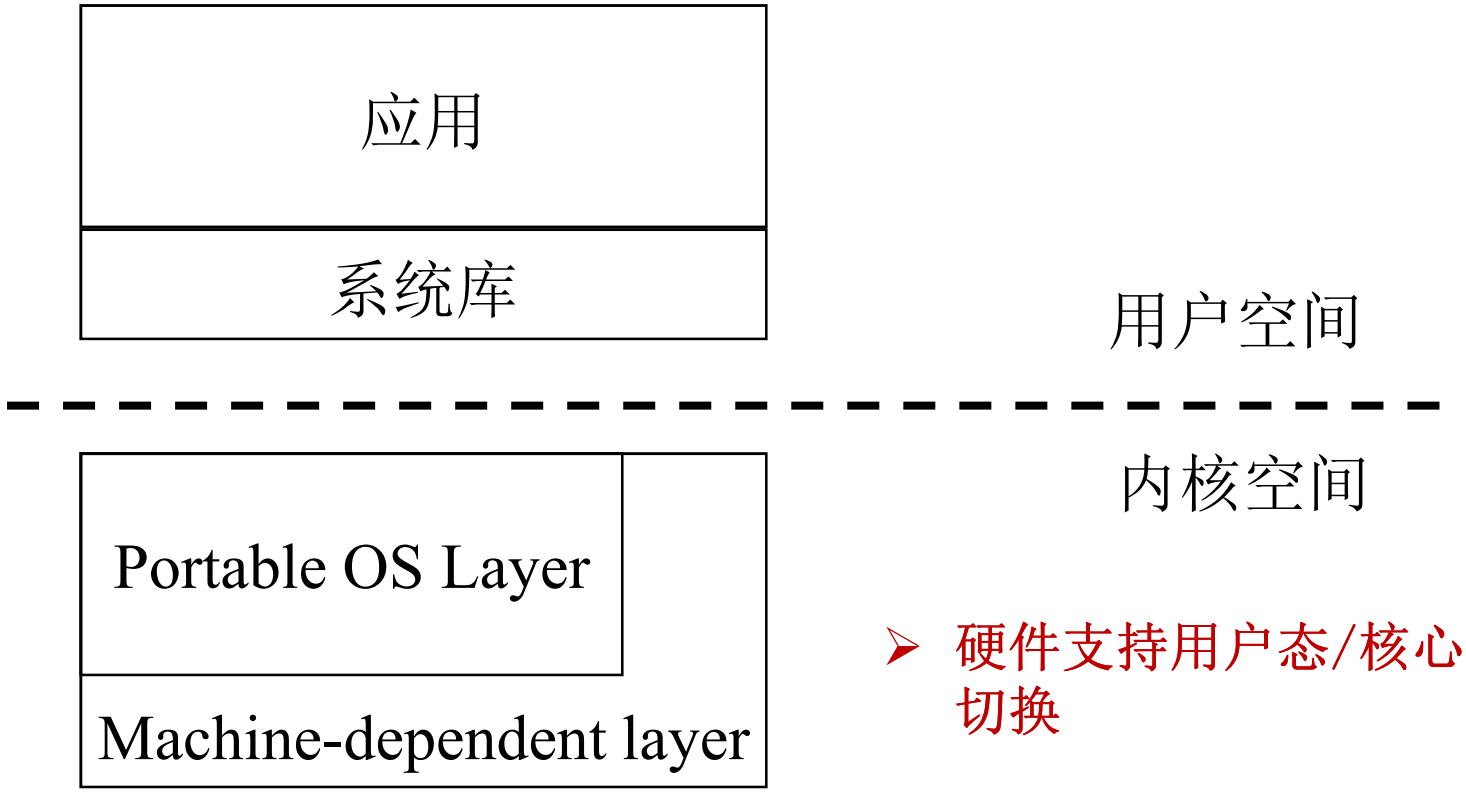
历史比较

	大型机	小型机	微机/移动计算机
系统价格/工作者年薪	10:1 – 100:1	10:1 – 1:1	1:10-1:100
主要考虑	系统利用率	总体造价	生产率
设计目标	系统容量	系统功能	易用性

一个典型的计算机系统



Unix/Linux操作系统一览



- 硬件支持用户态/核心态的切换

操作系统所需要的硬件支持

- 特权指令
- 操作系统保护（核心/用户模式）
- 内存保护
- 事件：异常与中断
- 时钟
- 输入输出控制

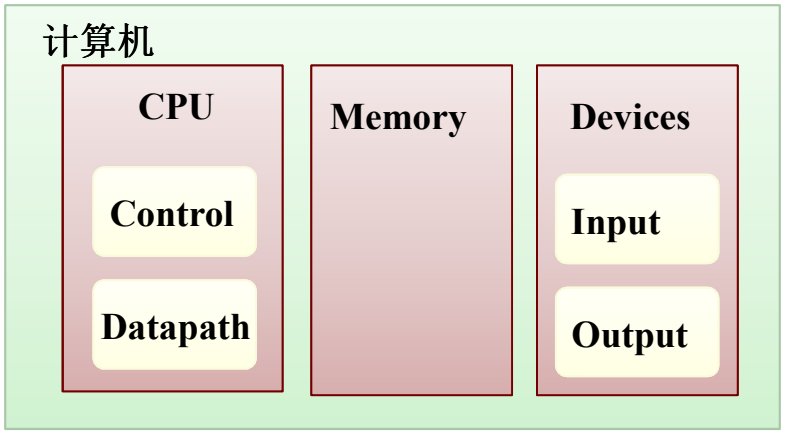
冯·诺依曼体系结构 1/2

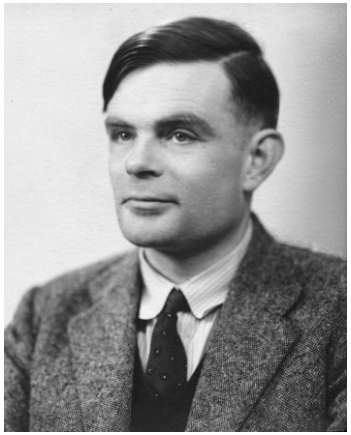
- 一种较为流行的观点：冯·诺依曼体系结构，也叫普林斯顿体系结构，描述了含有如下部件的电子数字计算机
  - 一个处理单元，包含了数学逻辑单元和处理器寄存器
  - 一个控制单元，包含了一个指令寄存器和一个**PC**
  - 一个内存来同时存储数据和指令
  - 外部大容量存储
  - 输入输出机制
- 这个概念演化为
  - 一种存储程序的计算机，在这样的计算机上，由于指令的读取和数据的操作共享同一总线，两者不能同时进行
  - 这个问题被称作冯·诺依曼瓶颈

From Wikipedia: http://en.wikipedia.org/wiki/Von_neumann_architecture

冯·诺依曼体系结构 2/2

- 冯·诺依曼体系结构有哪些优势、哪些劣势？
- 能不能想出一些非冯·诺依曼体系结构的计算系统？





Alan Turing

On Computable Numbers, with an Application to the Entscheidungsproblem, 1936

J. Presper Eckert

John William Mauchly

Progress report for the first period of the ENIAC's development, 1943

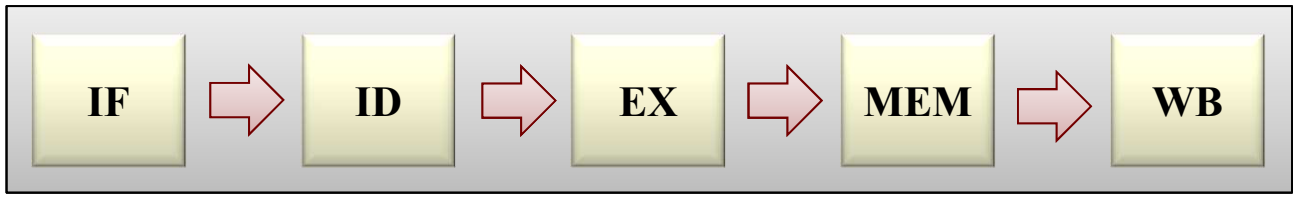
John Von Neumann

First Draft of a Report on the EDVAC, 1945

EDSAC, May 1949

冯·诺依曼体系结构处理器的内部

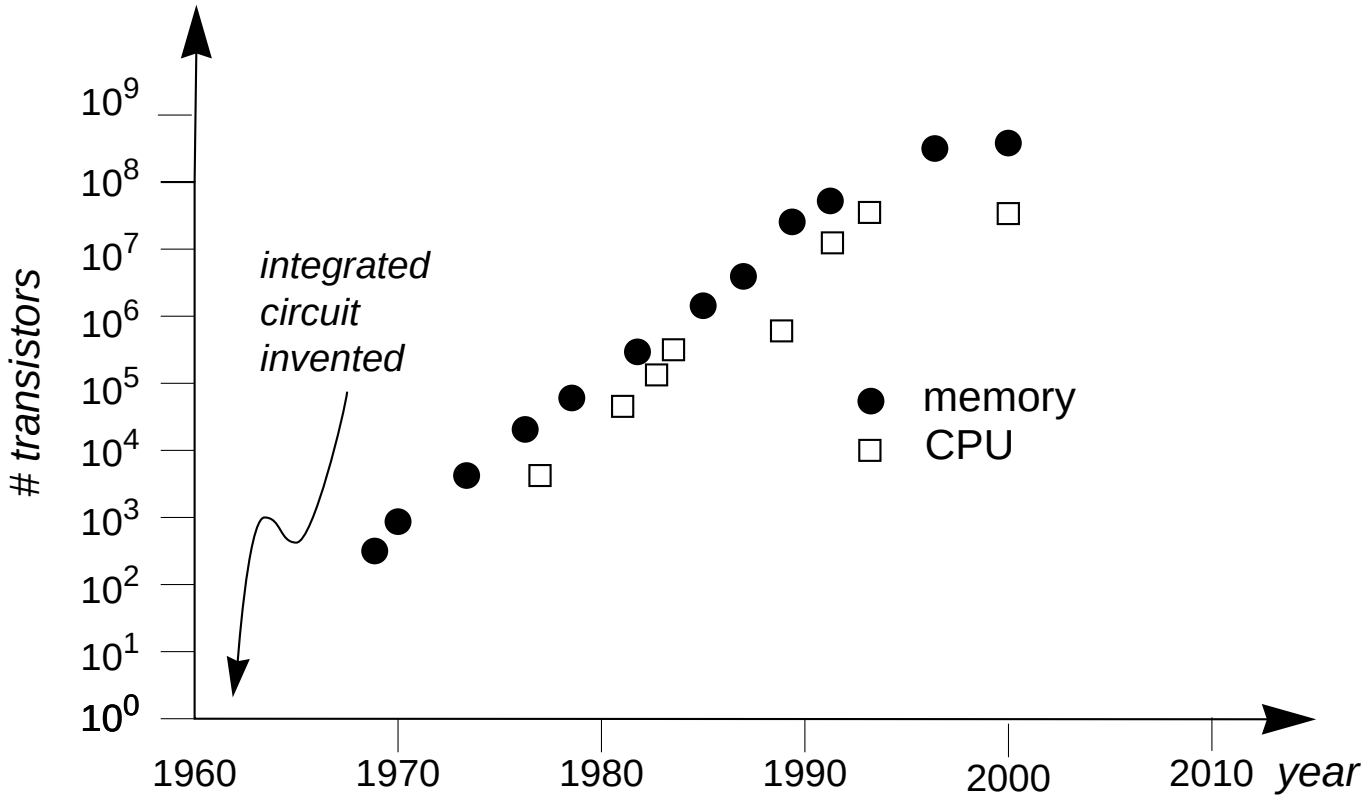
- 指令集体系结构（ISA）
- 通用寄存器
- **Program counter（PC）**
- **Program status word（PSW）**
- 流水线级



- 进程切换

摩尔定律（Moore's Law）1/2

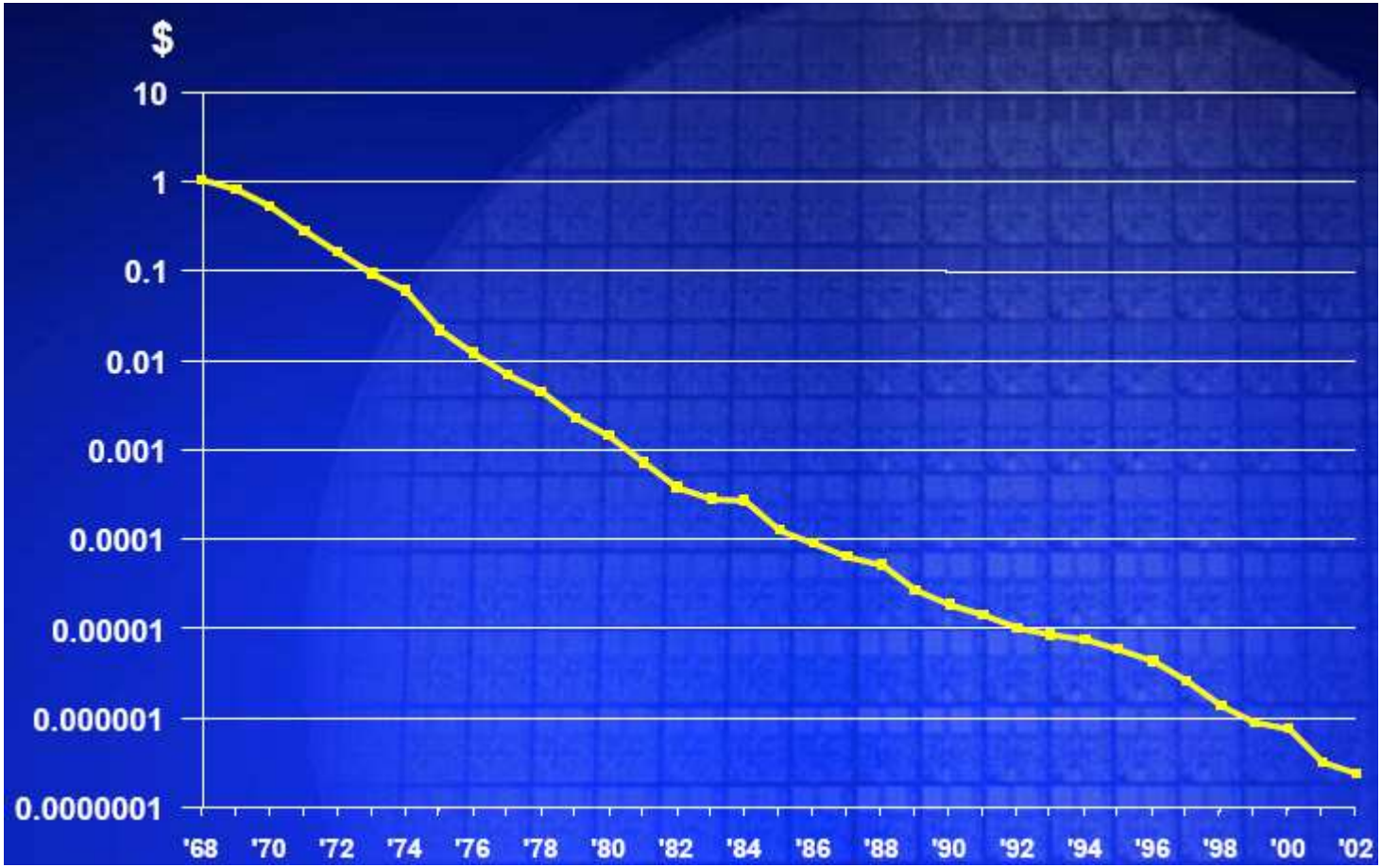
- 1965年，Gordon Moore预期，芯片上晶体管的数目每18个月翻一番



摩尔定律（Moore's Law）2/2

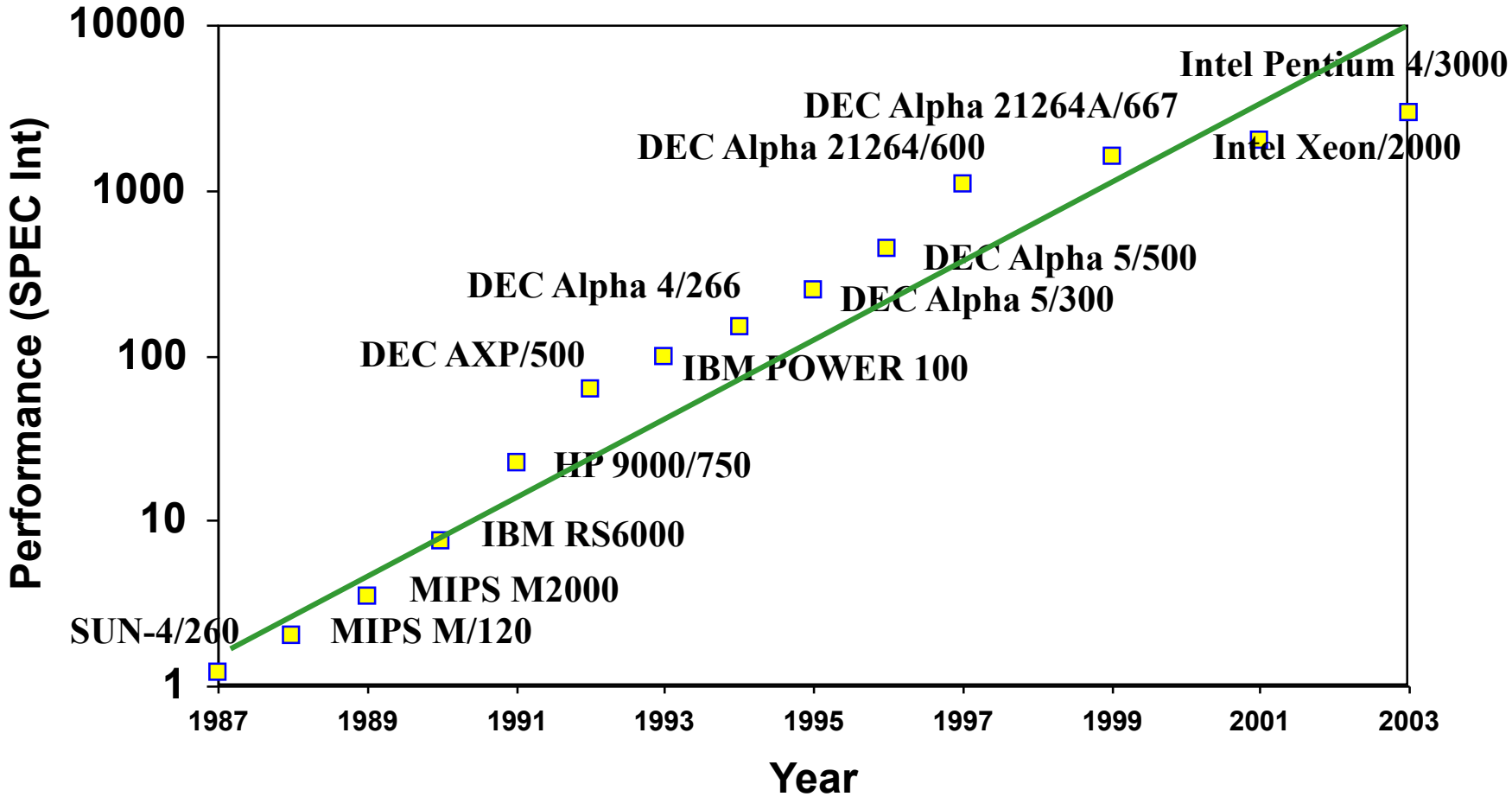
- 40余年，芯片上晶体管数目从2300突破了50亿
  - 1971，2300晶体管，Intel 4004（1 MHz），10 μm
  - 1979，68000晶体管，Motorola 68000，4 μm
  - 1989，100万晶体管，Intel 80486，1 μm
  - 1999，950万晶体管，Intel Pentium III，250nm
  - 2003，1亿晶体管， AMD K8，130 nm
  - 2010，10亿晶体管，SUN 16-Core SPARC T3，40nm
  - 2012，50亿晶体管，Intel 62-Core Xeon Phi,，22nm

每年每晶体管的平均价格

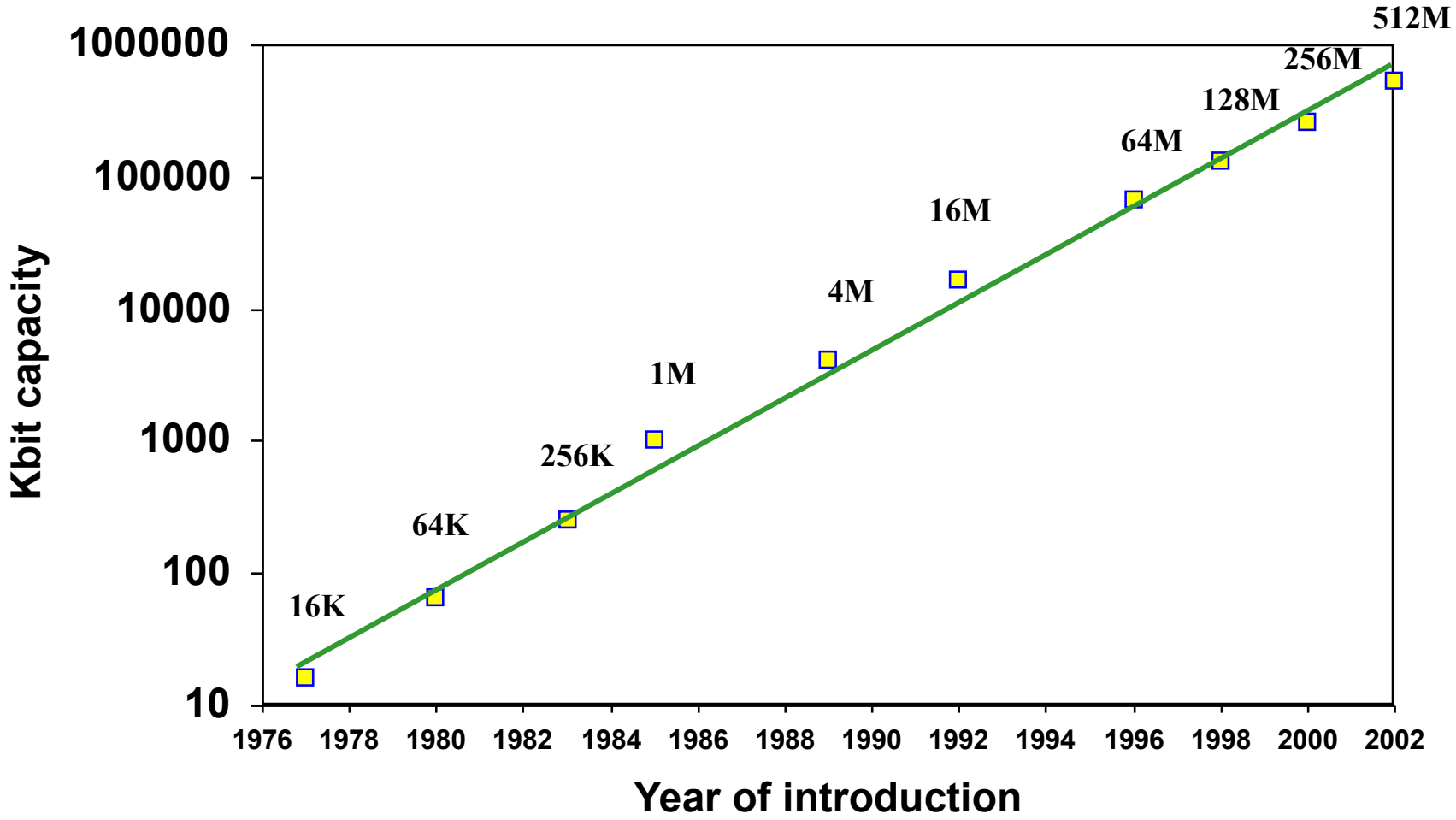


Source: G. Moore, ISSCC 2003

处理器性能的不断提升


10000
1000
100
10
1
Performance (SPEC Int)
1987
1989
1991
1993
1995
1997
1999
2001
2003
Year
Intel Pentium 4/3000
DEC Alpha 21264A/667
DEC Alpha 21264/600
Intel Xeon/2000
DEC Alpha 5/500
DEC Alpha 4/266
DEC Alpha 5/300
DEC AXP/500
IBM POWER 100
HP 9000/750
IBM RS6000
MIPS M2000
SUN-4/260
MIPS M/120

DRAM容量的变化


512M
256M
128M
64M
16M
4M
1M
256K
64K
16K
1000000
100000
10000
1000
100
10
Kbit capacity
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
Year of introduction

新技术的影响

- 处理器
  - 逻辑容量：每年增长30%左右
  - 性能：每1.5年提高一倍
- 内存
  - DRAM容量：原本每3年4倍；现在每两年2倍
  - 内存速度：每10年1.5倍
  - 每bit价格：大致每年减少25%
- 硬盘
  - 容量：每年提高60%

CPU性能公式

- 程序/进程运行的CPU时间
  = 指令数 × CPI × 时钟周期时间
  = 指令数 × CPI ÷ 时钟频率

- CPI：平均执行每条指令所需的时钟周期数

计算机性能与绝对指标

- **MIPS**：每秒百万次整数运算
  - **Intel 4004，0.092 MIPS at 740 KHz**
  - **Motorola 68000，0.700 MIPS at 8 MHz**
  - **Intel 80486DX2，54 MIPS at 66 MHz**
  - **Intel Pentium III，2,054 MIPS at 600 MHz**
  - **Intel Core i7 2600K（4-Core），128,300 MIPS at 3.4 GHz**
  - 上述处理器的**CPI**各是多少？
- **MFLOPS**：每秒百万次浮点运算，通常用于超级计算机评测
  - 当前世界最快的超级计算机**Titan - Cray XK7**，速度**17.59 PFlops**（**17,590,000,000 MFLOPS**）
  - 中国最快的超级计算机天河一号（目前世界第**8**），速度**2.57 PFlops**

Amdahl定律

- 一段程序，由W_S和W_P两部分组成（$W_S + W_P = 1$），W_S部分是串行运行的，W_P部分是并行运行的（或可以加速的），假设通过某种方法可以将W_P部分加速N倍，最后会使程序快多少倍？
  - 设原运行时间为T_S，加速后的时间T_T为
    $T_T = (T_S * W_S) + (T_S * W_P)/N = T_S * (W_S + W_P/N)$
  - 加速比Speedup为$T_S / T_T =$

$$\frac{1}{W_S + \frac{W_P}{N}}$$

- 如果N=5，则Wp = 90%时会怎样？Wp = 10%时会怎样？
- 如果N=∞呢？

计算机系统结构、组成、实现

- 计算机系统结构
  - 也称计算机体系结构，是软硬件的交界面，是程序员所看到的计算机的属性，即概念性结构与功能特性
  - 例：指令系统中是否含有乘法指令？
- 计算机组成
  - 计算机系统结构的逻辑实现，包括机器级内的数据通路和控制的组成及逻辑设计等
  - 例：乘法指令是通过加法器实现的，还是通过专门的高速乘法器（例如**booth**乘法器）实现的？
- 计算机实现
  - 计算机组成的物理实现，如器件、微组装技术等
  - 例：加法器、高速乘法器的具体实现是什么？

指令集体系结构（ISA）

- 一台机器硬件与最底层软件之间的抽象界面，包含了编写机器语言程序所必需的所有信息，包括指令集、寄存器、存储访问方法、I/O方法等

  "... the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation."
  – Amdahl, Blaauw, and Brooks, 1964

- 可以使相同软件运行在同一指令集体系结构的不同型号计算机/处理器上（它们具有不同的价格和性能）

结构 v.s. 实现

By the *architecture* of a system, I mean the complete and detailed specification of the user interface. … As Blaauw has said, "Where architecture tells *what* happens, implementation tells *how* it is made to happen."

***The Mythical Man-Month*, Brooks, pg 45**

指令系统

字节、字、双字

- 字（Word）的原始含义
  - 一个8位的系统，一个字原始含义是多少位？
  - 64位的系统呢？
- 通常含义
  - 字节（Byte）：8位
  - 字（Word）：16位
  - 双字（DWord）：32位

有符号数

- 32位补码

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
...

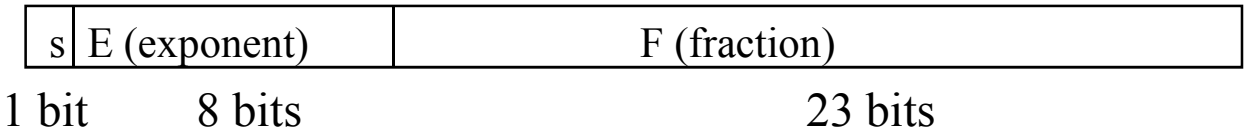
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

- 如何快速生成负数的二进制表示？
- 如何将非32位数扩展到32位数？
  - 0010如何扩展？
  - 1010如何扩展？

超大和超小的数 → 浮点数

- 32位定点数的极限
  - Q0: 0 – 4,294,967,295，或者 -2,147,483,648 – 2,147,483,647
  - Q32: 非0的数最小到$1/2^{32} = 0.00000000023283064365 3\ldots$
- 如何表示地球的年龄？
  - 4,600,000,000 or 4.6×10^{9}
- 如何表示原子质量单位？
  - 0.00000000000000000000000166 or 1.6×10^{-27}
- 浮点数（课后自行参考IEEE 754）
  - $(-1)^{sign} \times F \times 2^{E}$
  - F被标准化为1.xxx…
  - 单精度32位



  - 双精度64位

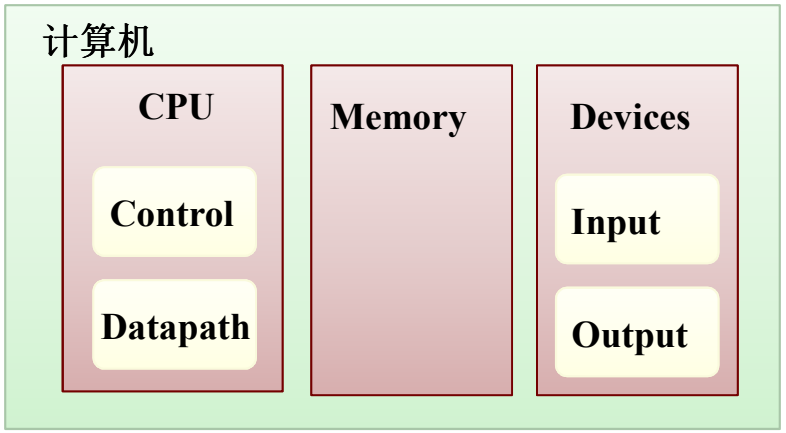
字符的表示方法

- **American Standard Code for Information Interchange (ASCII)**

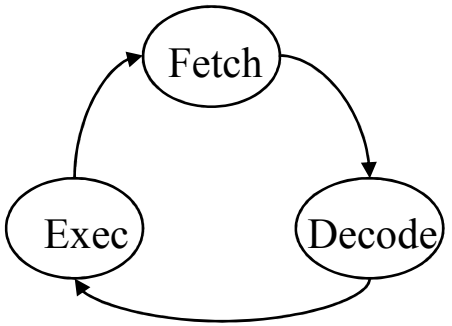
ASCII	Char		ASCII	Char	ASCII	Char	ASCII	Char		ASCII	Char	ASCII	Char
0	Null		32	space	48	0	64	@		96	`	112	p
1			33	!	49	1	65	A		97	a	113	q
2			34	“	50	2	66	B		98	b	114	r
3			35	#	51	3	67	C		99	c	115	s
4	EOT		36	$	52	4	68	D		100	d	116	t
5			37	%	53	5	69	E		101	e	117	u
6	ACK		38	&	54	6	70	F		102	f	118	v
7			39	‘	55	7	71	G		103	g	119	w
8	bksp		40	(	56	8	72	H		104	h	120	x
9	tab		41	)	57	9	73	I		105	i	121	y
10	LF		42	*	58	:	74	J		106	j	122	z
11			43	+	59	;	75	K		107	k	123	{
12	FF		44	,	60	<	76	L		108	l	124	|
15			47	/	63	?	79	O		111	o	127	DEL

冯·诺依曼体系结构组织

- 数据通路 – **Datapath**，包含
  - 各功能模块以及执行指令所需的存储（例如寄存器）
  - 各模块之间的互联（包含支持从内存中访问数据的部分）



- 控制 - **Control**
  - 从内存中取指令
  - 发射信号，控制在数据通路不同模块之间的信息传递和它们的操作类型
  - 控制指令序列

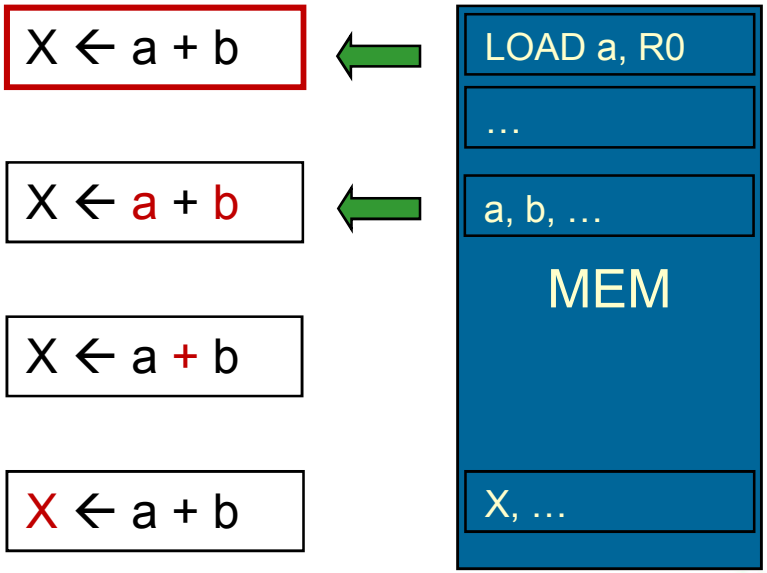


如何实现/执行指令

- 取值 (IF)
- 译码 (ID)
- 执行 (EX)
- 访问内存 (MEM)
- 回写寄存器 (WB)

X ← a + b

- LOAD a, R0
- LOAD b, R1
- ADD R0, R1, R2
- STORE R2, X

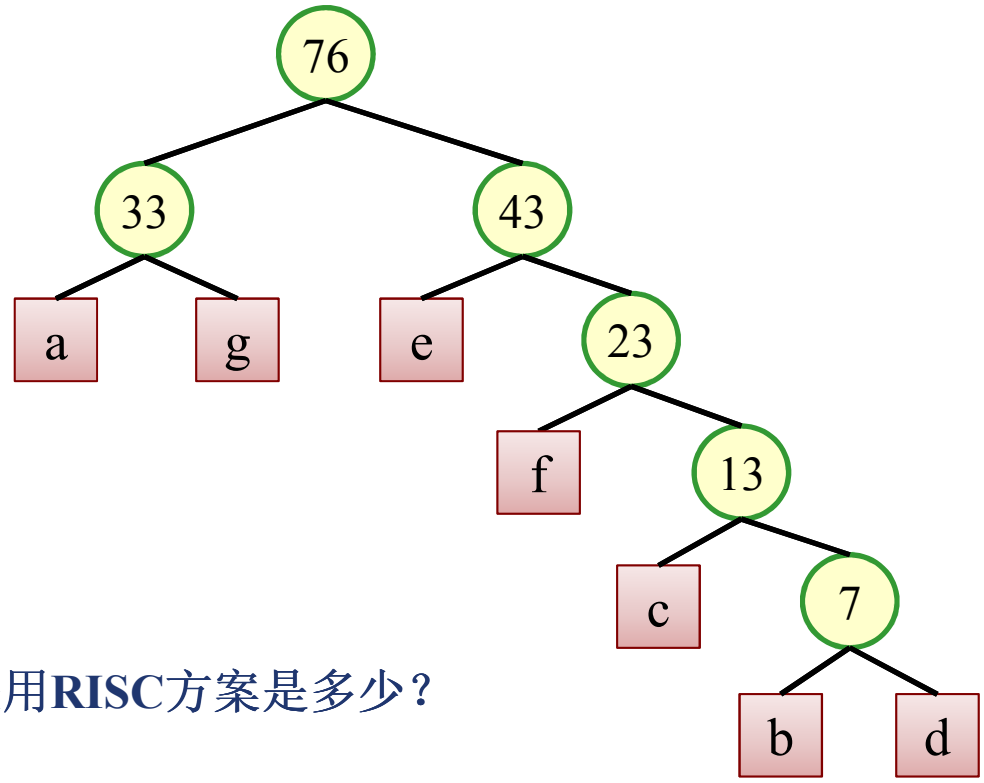


RISC

- 精简指令集计算机
  - MIPS、PowerPC，……
  - 与其相对：CISC复杂指令集计算机 – x86
- RISC的特点
  - 固定的指令长度
  - 采用LOAD、STORE方式访问内存
  - 有限的寻址方式
  - 有限的操作集
- 目标
  - 优化最常用的情景
  - 简化处理器的实现
- 优缺点？

CISC处理器的指令编码

- CISC处理器的指令长度是可变的，指令操作码部分的长度也是可变的
  - 有何好处？
- 如果一个CISC处理器各指令使用的频率不同，应当如何对操作码编码？
  - 设共有7条指令a, b, c, d, e, f, g，使用频率比值为：15:2:6:5:20:10:18。
  - Huffman树
  - a: 00
  - b: 11110
  - c: 1110
  - d: 11111
  - e: 10
  - f: 110
  - g: 01
  - 平均码长是多少？如果采用RISC方案是多少？

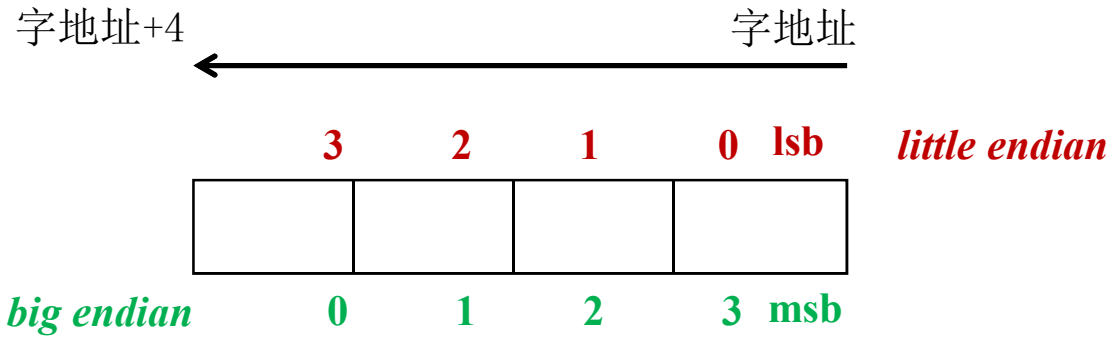


内存：按字节寻址

- 8位的字节很常用，因此大多数系统都在内存中按照字节寻址
  - 注意，于是，对于一个32位系统而言，一个原始含义的字，在内存中的地址均为4的倍数（对齐）
- 对于原始含义的字，如何在内存中存放其各个字节？
  - 大端Big Endian: 最左边的字节（最高位msb）放在字地址上

    IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
  - 小端Little Endian: 最右面的字节（最低位lsb）放到字地址上

    Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

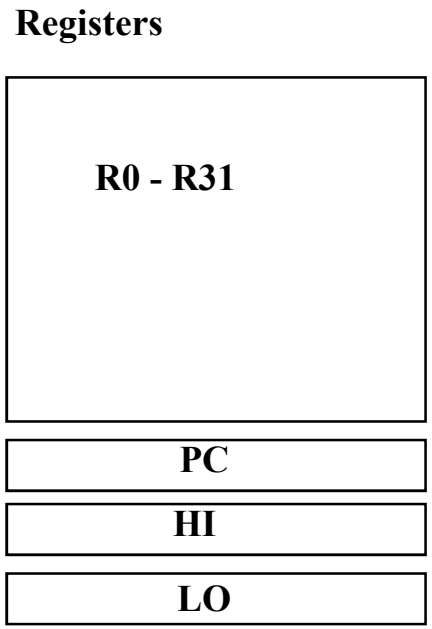


主要寻址方式

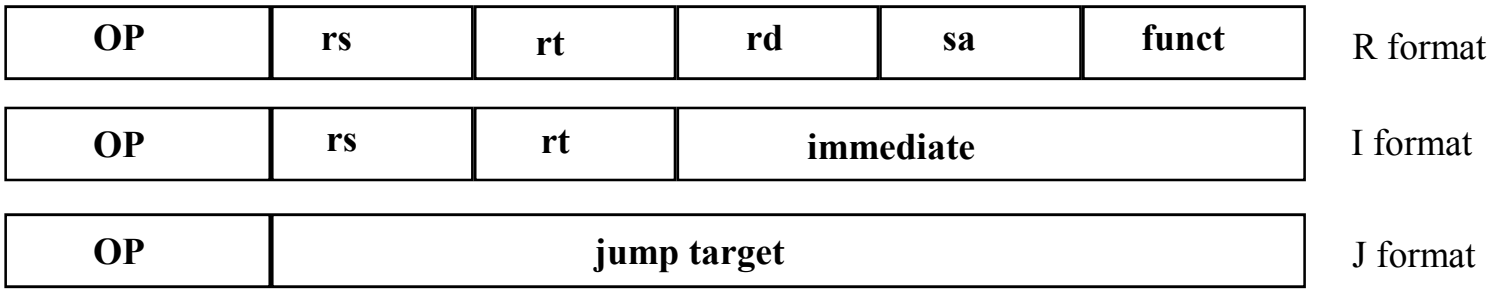
- 立即数寻址方式
  - 所需数据作为立即数编码在指令中
- 寄存器寻址方式
  - 所需数据在某一寄存器中，指令指定这一寄存器
- 直接寻址方式
  - 地址编码在指令之中
- 寄存器间接寻址方式
  - 地址在某一寄存器中，指令中指定这一寄存器
- 基址间接寻址方式（变址寻址方式）
  - 地址为：某一在指令中指定的寄存器中的值+一个在指令中指定的立即数
  - 某些时候，寄存器也可以是PC，或者PC中的前几位
  - 为何需要这样的寻址方式？
    - 转移、数组
- 基址变址寻址方式
- 各自优缺点？

MIPS R3000指令集体系结构

- RISC处理器
- 指令类型
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide

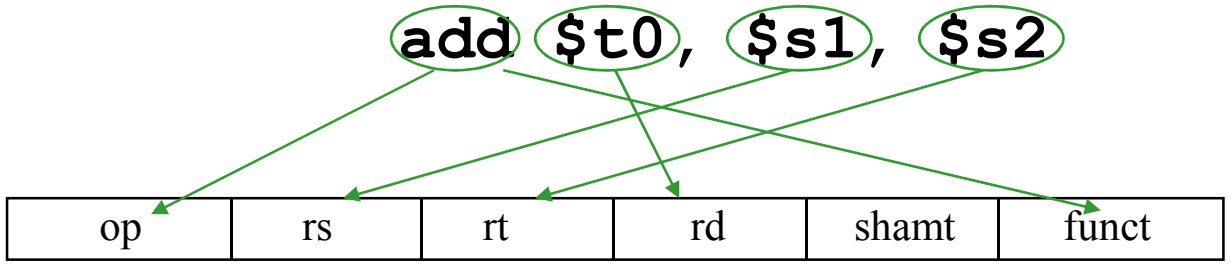


MIPS的寄存器

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

MIPS指令格式 1/3

- 所有指令均为32位长
- R格式指令



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

MIPS指令格式 2/3

- I格式指令

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

I format

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 result's destination
immediate	16-bit	$+2^{15}-1$ to -2^{15}

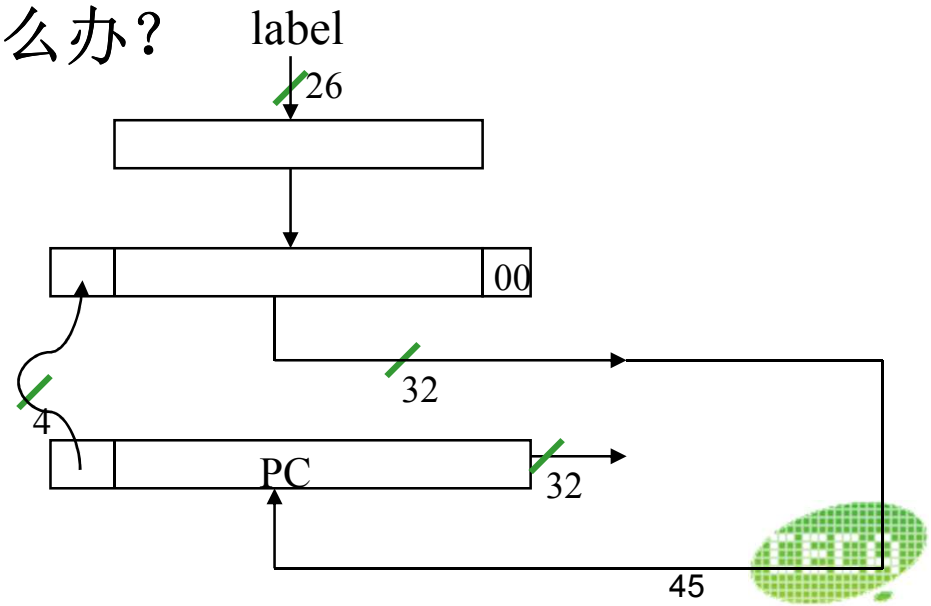
MIPS指令格式 3/3

- J指令格式

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

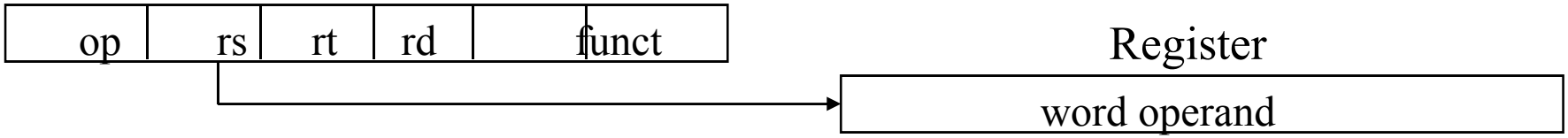
op	6-bits	opcode that specifies the operation
lable	26-bits	$+2^{25}-1$ to -2^{25}

- 思考：如果要跳转到更远处怎么办？
  - 用R指令格式
    - `jr $rs`

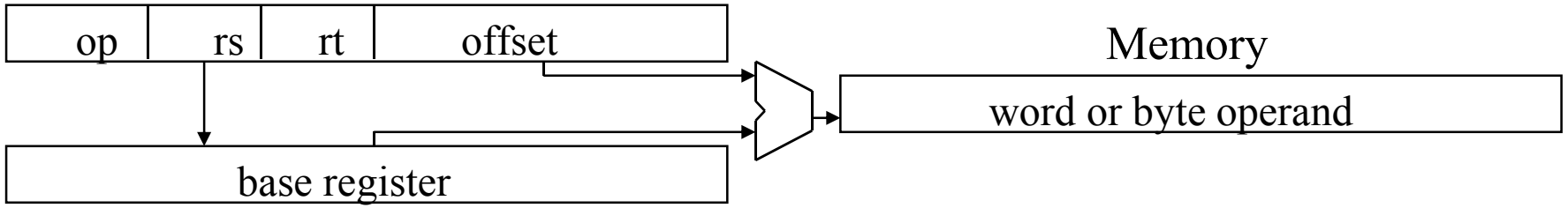


MIPS寻址方式 1/2

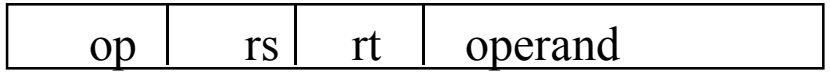
- **Register addressing**（寄存器寻址） – 操作数在寄存器中



- **Base (displacement) addressing**（基址间接寻址） – 操作数在内存中，其地址的指定方式：某一寄存器中的值与指令中16位立即数之和

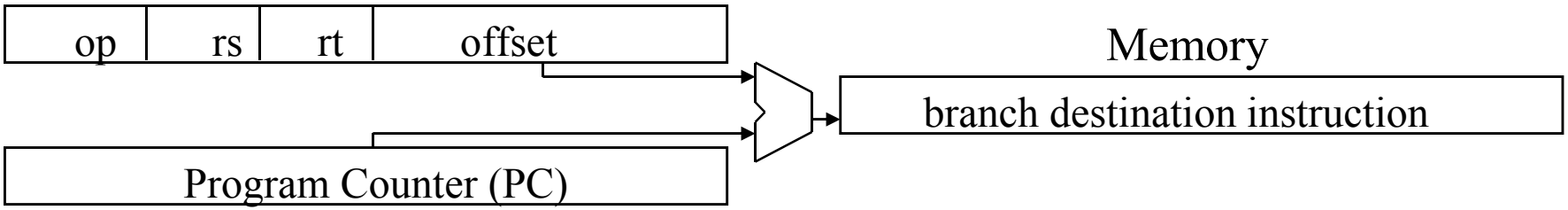


- **Immediate addressing**（立即数寻址） – 操作数是指令中16位立即数

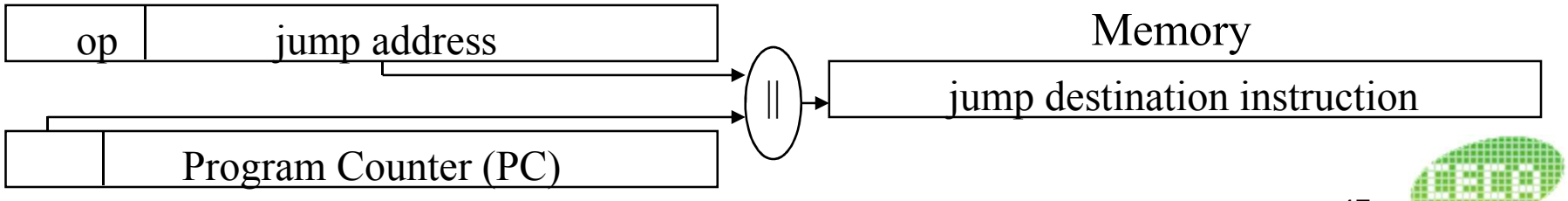


MIPS寻址方式 2/2

- **PC-relative addressing** （基址间接寻址之一） – 指令地址是PC与指令中16位立即数之和



- **Pseudo-direct addressing** （基址间接寻址之二） – 指令地址为PC的最高4位，接上指令中26位立即数



MIPS ISA (部分)

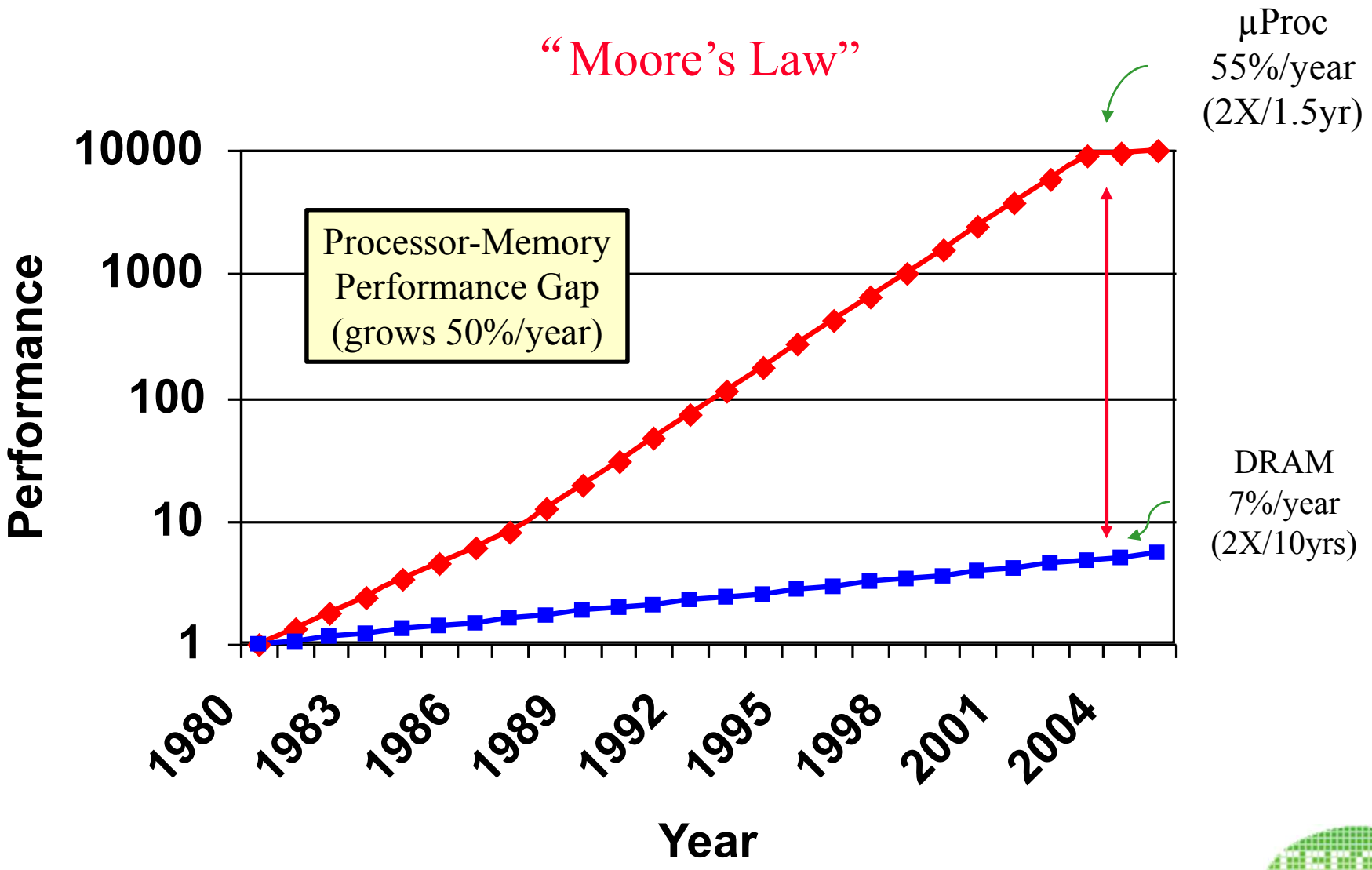
Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	or immediate	13	ori $s1, $s2, 6	$s1 = $s2 v 6
Data Transfer (I format)	load word	35	lw $s1, 24($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 (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

MIPS (RISC) 设计原则

- 简单、规整
  - fixed size instructions – 32-bits
  - small number of instruction formats
  - opcode always the first 6 bits
- 好的设计需要好的折中
  - three instruction formats
- 更小，则更快
  - limited instruction set
  - limited number of registers in register file
  - limited number of addressing modes
- 让通常情况变快
  - arithmetic operands from the register file (load-store machine)
  - allow instructions to contain immediate operands

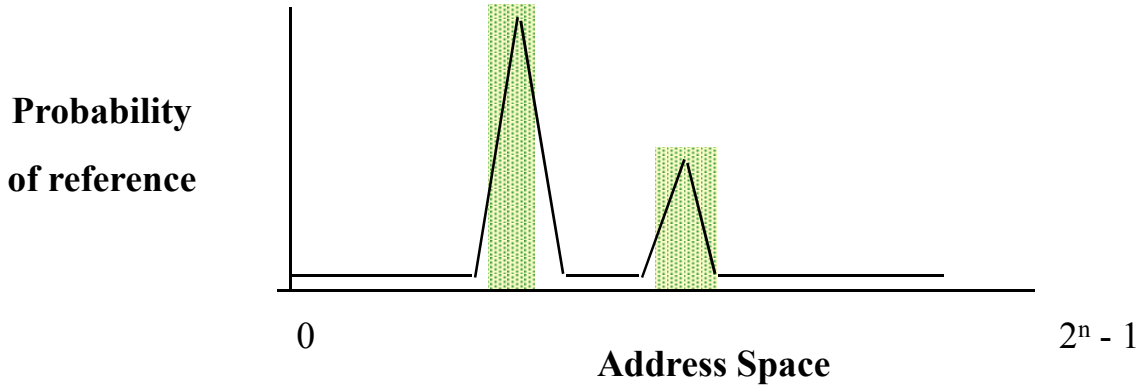
存储层次

处理器与内存之间的性能差距



存储层次的目标

- 期望：又大，又快
- 事实：大的存储慢，快的存储小
- 我们如何能够提供一种存储，让人们感觉在大多数情况下足够大、便宜，但是又快呢？
- 局部性原理的背后 – 在任何一个时刻，程序只访问一小块地址空间



局部性原理

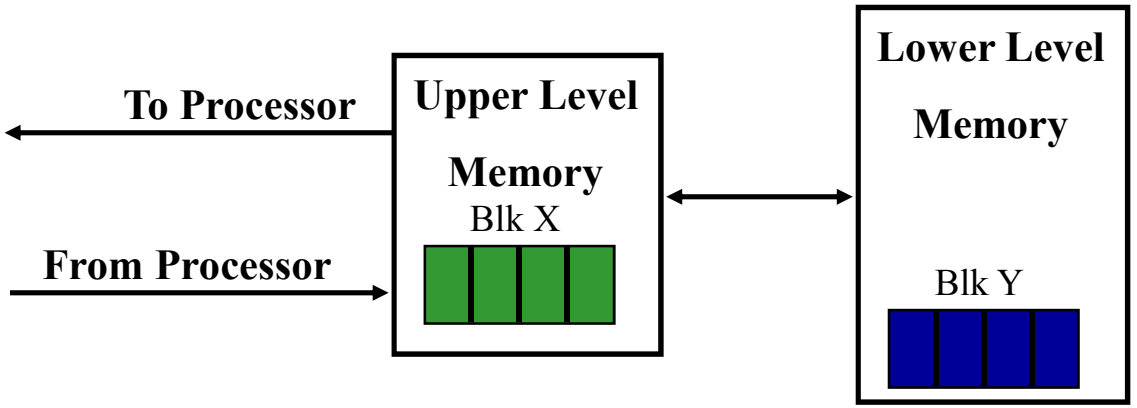
- 时间局部性
  - 近期被访问的位置，很可能会再次被访问
- 空间局部性
  - 近期被访问的位置的“邻居位置”，也很有可能会被访问

分层次处理问题

- 较上层存储（离处理器较近）
  - 更快、更小，更贵，离处理器更近
- 时间局部性

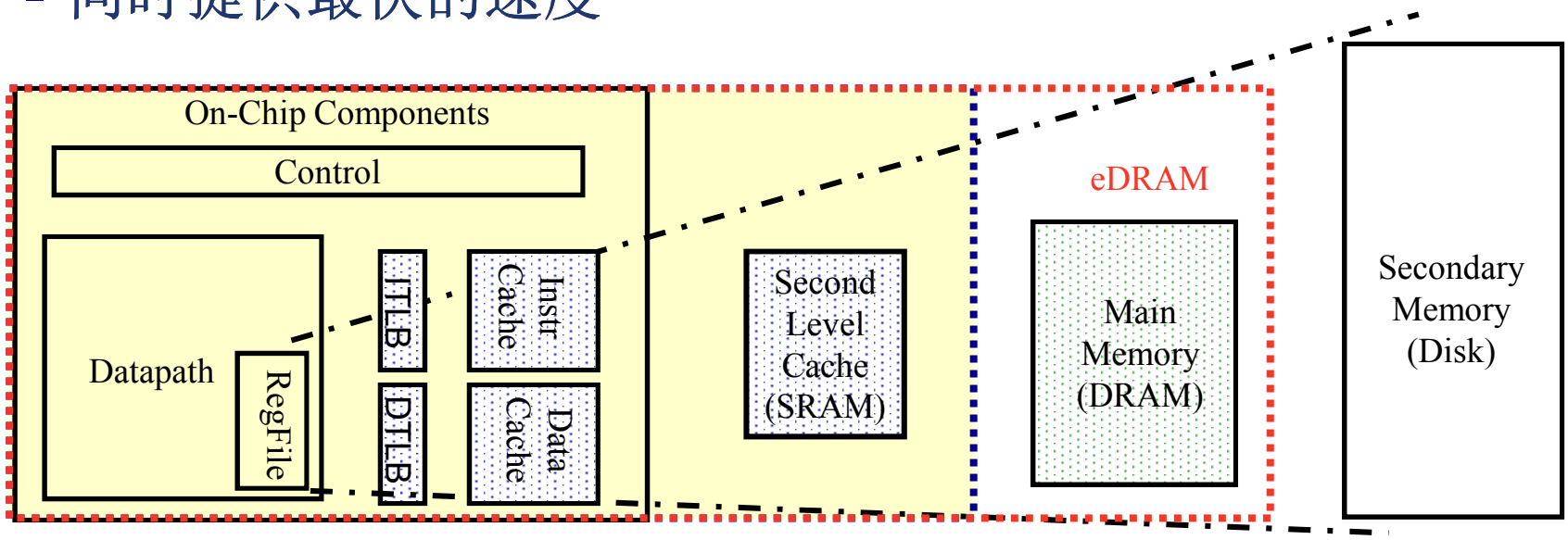
  => 将最近访问过的数据保持在离处理器较近的位置
- 空间局部性

  => 将含有连续数据的块同时搬移到上一层



一个典型的存储结构

- 通过利用局部性原理
  - 对用户展现尽可能大的存储空间（利用最便宜的技术）
  - 同时提供最快的速度



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

存储层次所用的技术

- 用*SRAM*实现缓存以提高速度
  - 低密度、高功耗、昂贵、快速
  - 静态：只要不断电，内容就会一直存在
- 用*DRAM*实现内存来提高容量
  - 高密度、低功耗、便宜、慢
  - 动态：必须定时“刷新”（~ 8 ms）
    - 1% to 2% of the active cycles of the DRAM
  - 分两部分输入地址（行和列）
    - RAS or Row Access Strobe：行地址解码
    - CAS or Column Access Strobe：列地址解码

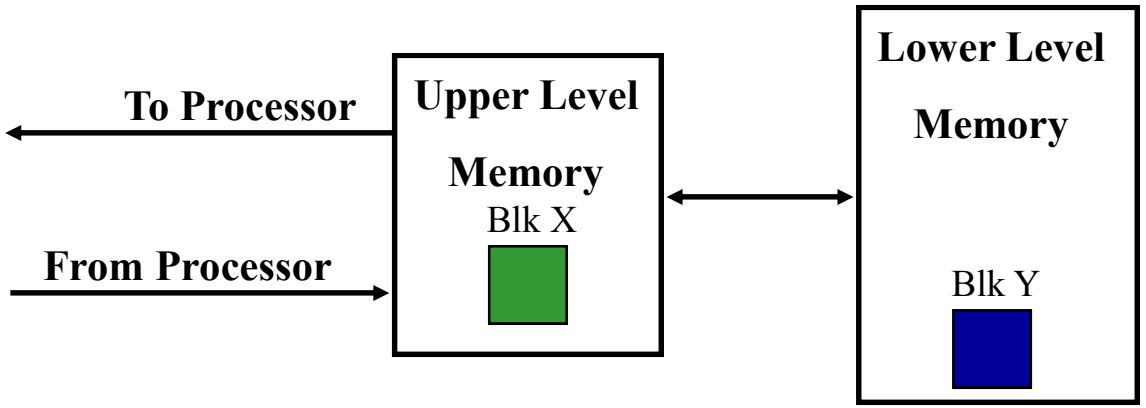
存储性能衡量维度

- 延迟：访问一个**word**的时间
  - *访问时间 Access time*：从请求发出，到请求完成之间的时间
  - *周期时间 Cycle time*：两个请求之间间隔的最短时间
  - 通常周期时间 > 访问时间
- 带宽：单位时间内可以访问多大的数据量
  - 数据通道的宽度 * 每秒能够访问的次数

- *容量：* **DRAM to SRAM** ***4 to 8***
- *造价/周期时间：* **SRAM to DRAM** ***8 to 16***

命中与失效

- 命中 **Hit:** 数据在上层存储的某块中能够被找到 **(Block X)**
  - 命中率: 能够命中的比率
  - 命中时间: 访问上层存储的时间－**RAM**访问时间 +判断是否命中的时间

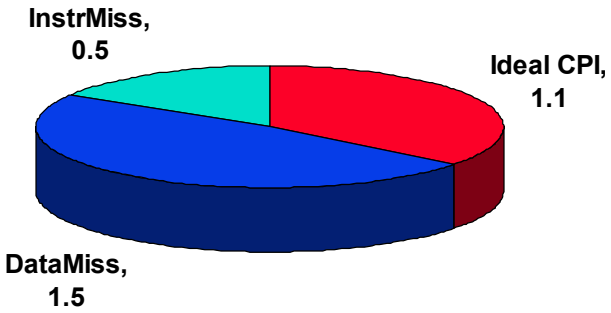


- 失效 **Miss:** 数据需要从下层存储的某块中取出 **(Block Y)**
  - 失效率 = 1 – 命中率
  - 失效损失 **Miss Penalty:** 替换上层存储某块的时间 + 访问此块的时间
  - 命中时间 << 失效损失

存储性能对于系统性能的影响

- 假设一个处理器有如下特性
  - ideal CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control

  并且有10%的数据失效率，失效损失是50个周期



- CPI = ideal CPI + average stalls per instruction
  = 1.1 (cycle) + (0.30 (datamemops/instr)
  x 0.10 (miss/datamemop)
  x 50 (cycle/miss))
  = 1.1 (cycle) + 1.5 (cycle) = 2.6
- 处理器58%的时间用来等待存储（失效）！
- 1%的指令失效会再给CPI增加0.5！

又快、空间又大

- **DRAM慢，但是便宜，且密度高**
  - 为用户提供很好的“大内存”
- **SRAM快，但是贵、密度低**
  - 为用户提供快速的访问
- 两种不同的局部性
  - 时间局部性：近期被访问的位置，很可能会再次被访问
  - 空间局部性：近期被访问的位置的“邻居位置”，也很有可能会被访问
- 通过利用局部性原理
  - 对用户展现尽可能大的存储空间
  - 同时提供最快的速度

高速缓存 Cache

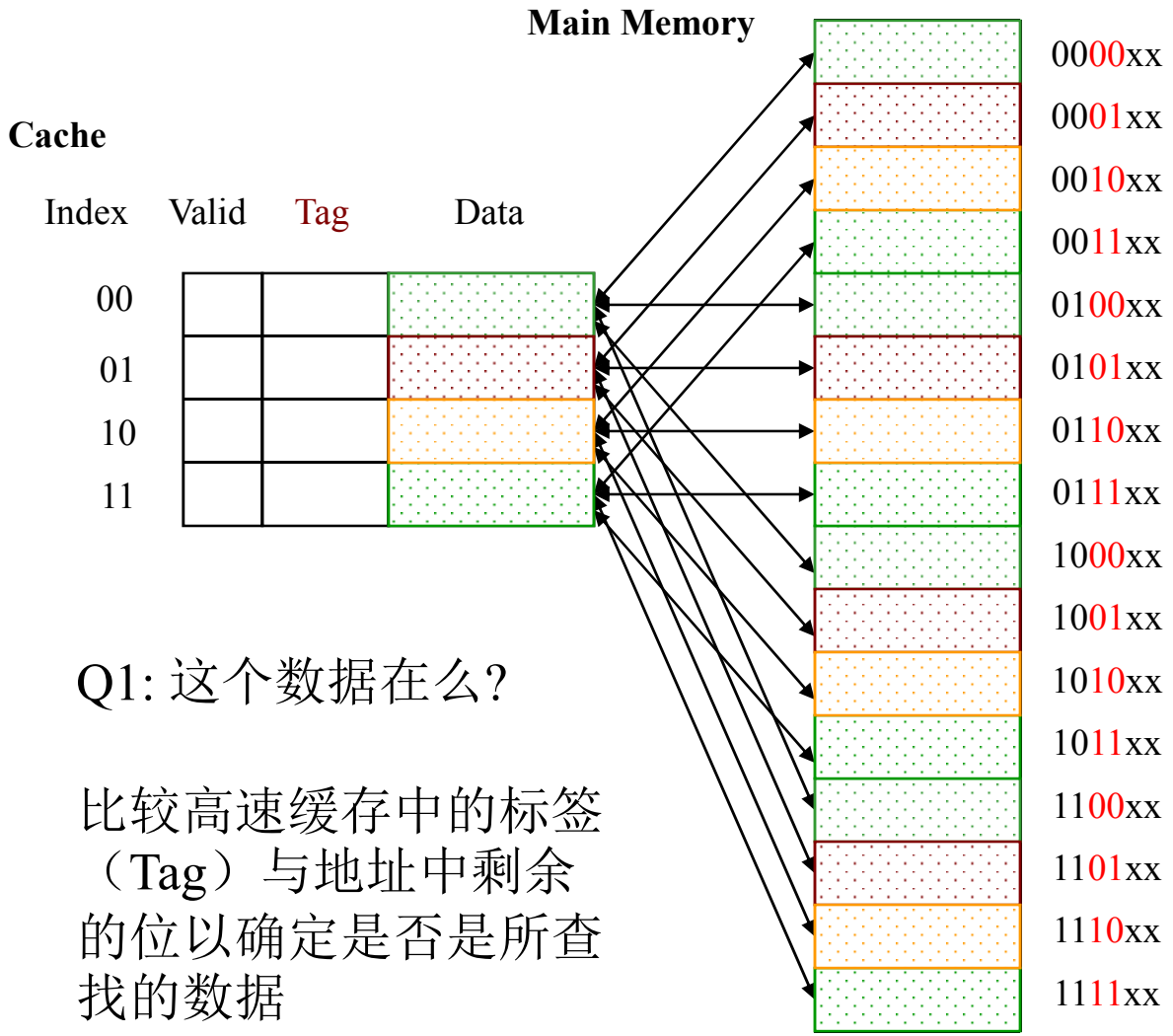
- 快而小（**SRAM**），离处理器近

- 两个问题 (硬件中):
  - **Q1:** 如何能够知道一份数据是否在高速缓存中？
  - **Q2:** 如果在，怎么能够找到它（寻址）？

- 直接映射
  - 对于任一个在较低层的数据，在上一层中都能找到其唯一可能的位置
    – 于是较低层很多的数据必须共享上一层的位置（为什么？）
  - 地址映射方式:

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

直接映射举例：假设一个缓存块为一个Word



地址最低两位没有被使用：定义一个32位word中的字节位置

Q1: 这个数据在么？

比较高速缓存中的标签（Tag）与地址中剩余的位以确定是否是所查找的数据

Q2: 如何找到数据？

用地址中接下来的两个低位作为索引值来查找高速缓存块 (i.e., modulo the number of blocks in the cache)

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

高速缓存命中时的处理

- 读命中 **(I$ and D$)**
  - 完成任务！
- 写命中 **(D$ only)**
  - 允许高速缓存与内存存在不一致的情况（**write-back**）
    - 仅将数据写入到高速缓存块中 (当这个缓存块下次被“弹出”的时候，写回（**write-back**）到下一层次中
    - 需要为每个缓存块标志一个脏（**dirty**）位，表明这个块是否与下一层次中的内容不一致
  - 要求高速缓存与内存一致的情况（**write-through**）
    - 总是将数据同时写入到高速缓存块与下一层次中
    - 注意：下一层次的写速度比较慢，所以性能较低，可以采用写缓冲**buffer**来缓解此问题，只有在缓冲满的时候才需要暂停（**stall**）

高速缓存失效时的处理

- 读失效 (I$ and D$)
  - 暂停整个处理器流水线，从存储层次下一层取出相应数据块，将其装入缓存（可能会涉及到块替换），并把需要读取的word发给处理器，最后继续流水线的执行
- 写失效 (D$ only)
  - **Write allocate:** 暂停整个处理器流水线，从存储层次下一层取出相应数据块，将其装入缓存（可能会涉及到块替换），将处理器所需要写的word写到缓存中，最后继续流水线的执行
  - **No-write allocate**方法:不写入缓存，直接写入内存

高速缓存失效的三种原因

- 义务失效/冷失效（冷启动或者进程切换时产生）
  - 第一次访问缓存块时产生，没有什么办法避免
  - 如果要运行几百万条指令，义务失效可以忽略
- 冲突失效
  - 多个内存地址被映射到同一个缓存块中
  - 解决方法1：增大缓存容量（随即增加了缓存块数目）
  - 解决方法2：增加相联度（每个缓存位置存在多个缓存块）
- 容量失效
  - 高速缓存不能容纳程序所需访问的所有内容
  - 解决方法：增加缓存容量

提高高速缓存性能的方法 1/2

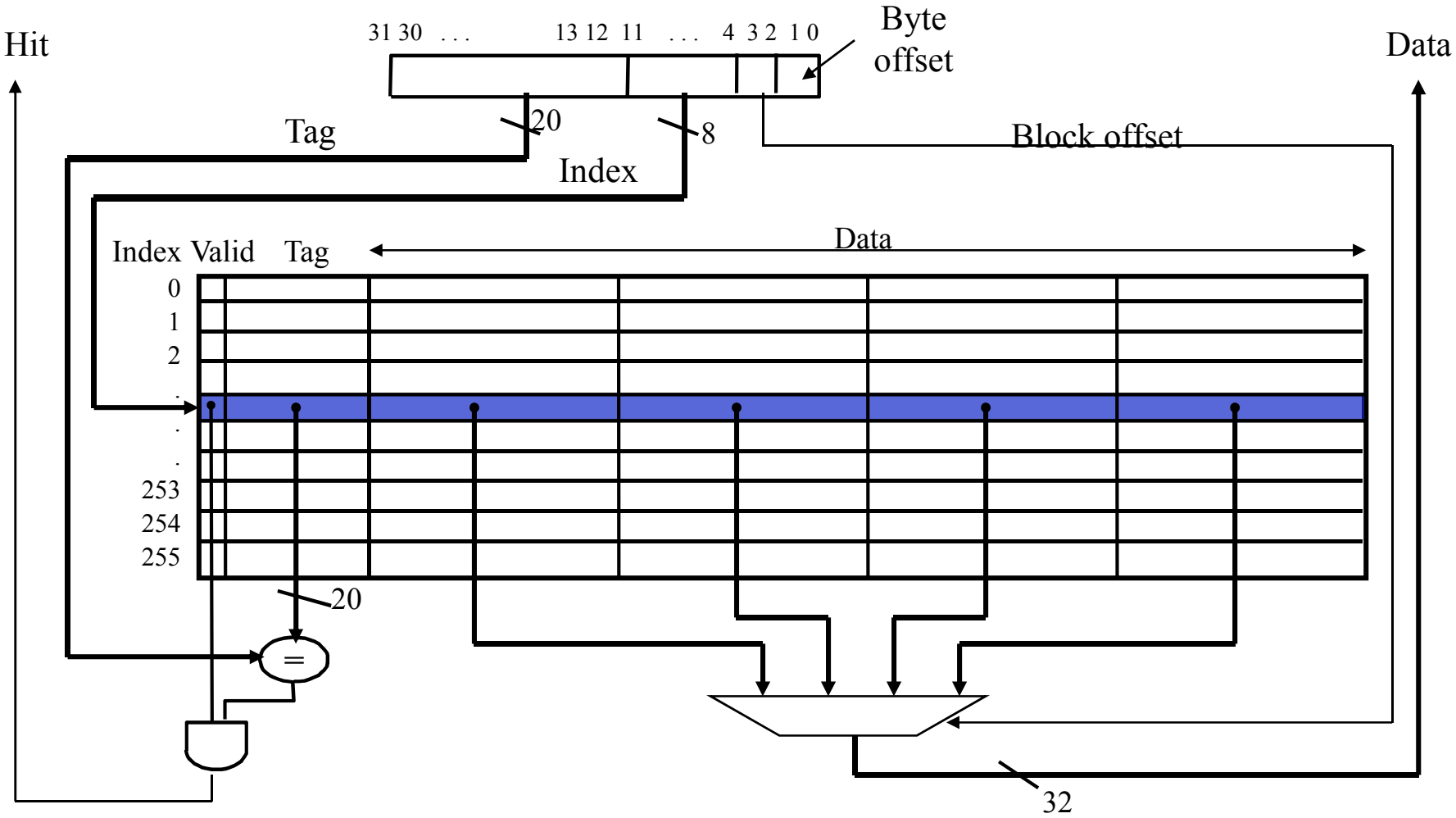
- 减小高速缓存命中时所需的时间
  - 更小的缓存
  - 采用直接映射
  - 更小的缓存块（更快取出所需word）
- 减小失效率
  - 更大的缓存
  - 允许更高的相联度
  - 更大的缓存块（典型值为16到64个字节）
  - victim cache – 小的缓冲区，保存刚刚被弹出的若干块数据

提高高速缓存性能的方法 2/2

- 减少失效损失
  - 更小的缓存块
  - 采用写缓冲，这样下一个读操作就不用等被替换的块真正完成被写操作
  - 当读失效的时候，检测**victim cache**或者写缓冲 – 看运气怎样
  - 对于大的缓存块，先取回关键**word**（需要操作的**word**）
  - 用多级缓存 – **L2**缓存的时钟周期可以比处理器时钟周期慢很多
  - 更快的下一级存储、更高的内存带宽
    - 更宽的内存总线
    - 交叉访问内存

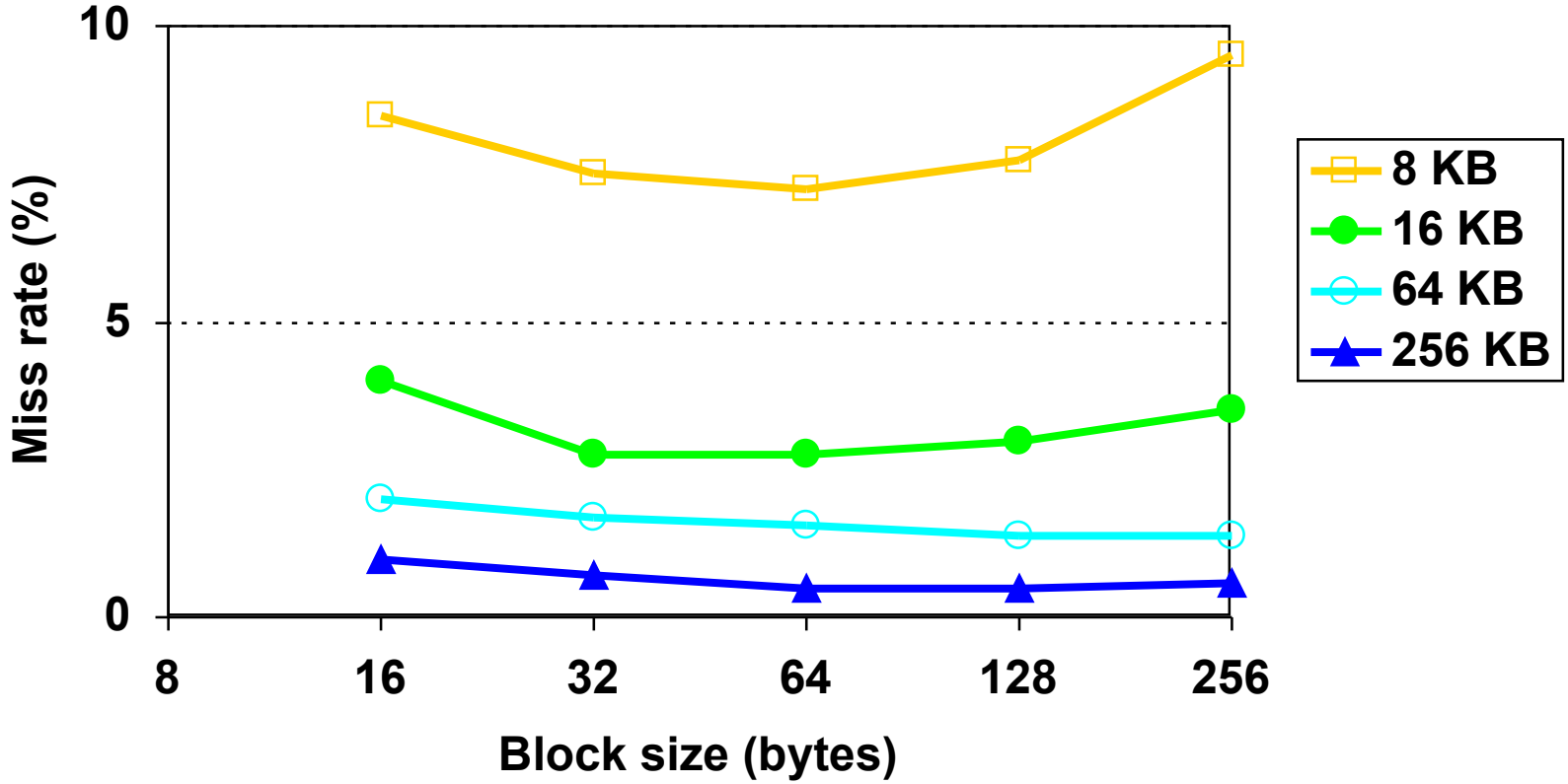
更大的缓存块：示例

- 每个块有4个word，缓存容量 = 1K words



这种情况下，我们能够利用到哪种局部性？

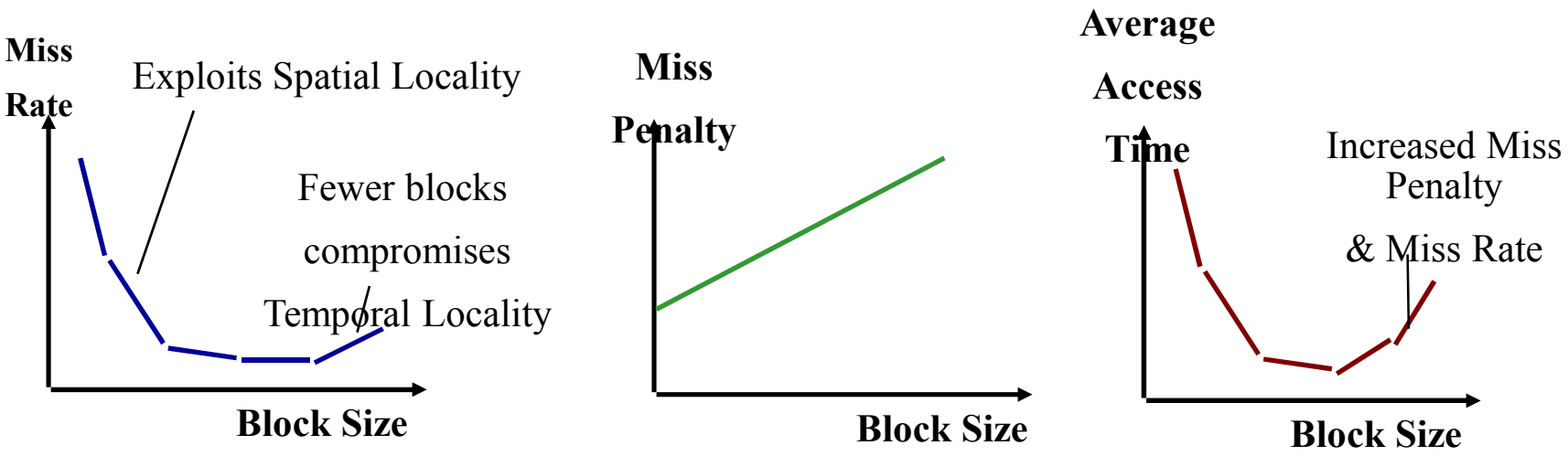
失效率 v. s. 缓存块大小 v. s. 缓存容量



- 当缓存块大小相对整个缓存容量太大时，由于块数目减少，增大了冲突失效，整体失效率会提高

缓存块大小的权衡

- 大的缓存块，意味着大的失效损失
  - 读取块中第一个word的延时+传输整个块剩下部分的时间
- 大的缓存块，能够更好地利用空间局部性，但是
  - 当缓存块大小相对整个缓存容量太大时，整体失效率会提高



- 通常情况下，平均内存访问时间

  = Hit Time + Miss Penalty x Miss Rate

每个块含有多个word时的一些考虑事项

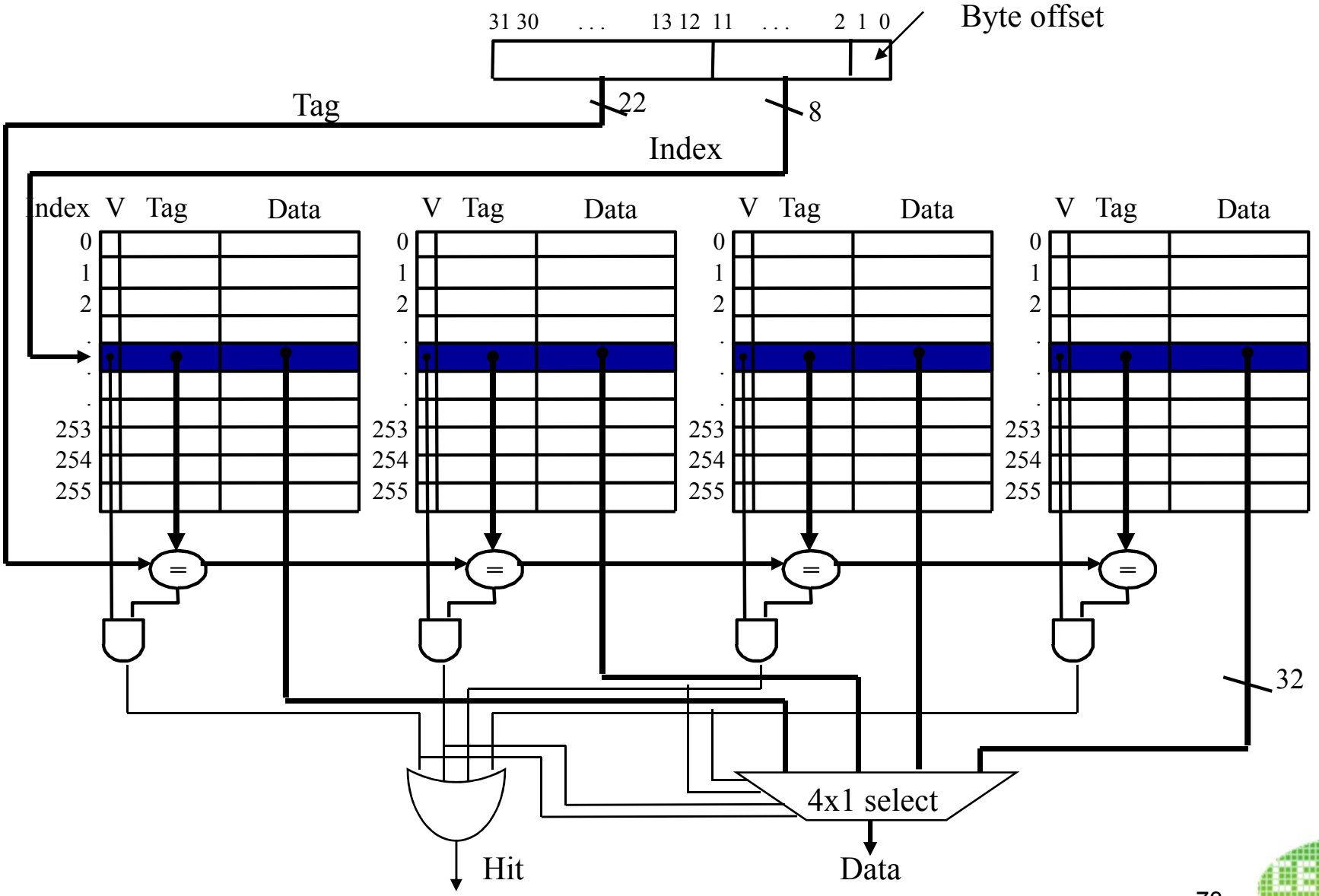
- 与单word块处理相同，一次失效时，要将整个块都从内存中读入高速缓存
- 当块大小增加时，失效损失增大；一组优化方法：
  - **Requested word first** – 先传输一个块中需要访问的word
  - **Early restart** – 一个块中需要访问的word一准备好，就恢复处理器执行
- 非阻塞型高速缓存 – 高速缓存处理前期失效时，允许处理器继续访问高速缓存的其它块

多路组相联高速缓存

- 在直接映射缓存中，一个内存块只能映射到唯一一个缓存块中
- 处于另外一个极端，全相联缓存中，一个内存块可以映射到任意一个缓存块中

- 一个权衡是，将缓存分成若干**sets**，每个**set**包含**N**路（**N**路组相联）
  - 每个内存块被映射到唯一的组中（通过索引），而在这个组中，可以放到任意一个块中

4路组相联高速缓存

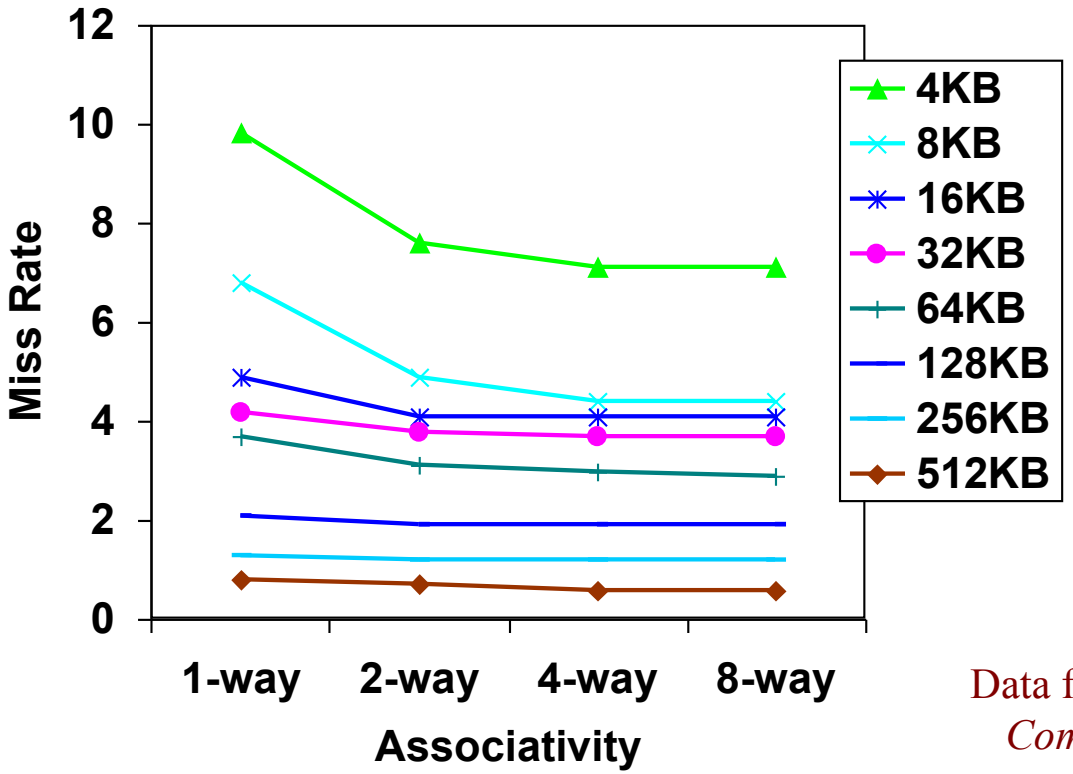

31 30 . . . 13 12 11 . . . 2 1 0
Byte offset
Tag
22
8
Index
Index V Tag Data
V Tag Data
V Tag Data
V Tag Data
0
1
2
253
254
255
=
32
4x1 select
Hit
Data

组相联高速缓存的开销

- 产生失效时，如何选择被替换的缓存块？
  - **FIFO**
  - 随机
  - **Least Recently Used (LRU):** 选择离现在最长时间没有被用到的缓存块
    - 必须有硬件来记录使用信息
- **N路组相联高速缓存的开销**
  - **N个比较器（延迟、面积）**
  - **N-to-1选择器MUX**
  - 无法像直接映射高速缓存那样假设命中猜测执行（不知道取哪一路）

组相联高速缓存的优势

- 做抉择时，需要综合考虑失效损失的代价和实现开销



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

最大的提升，在于从直接映射到2路组相联
（失效率减少20%左右）

多级高速缓存

- 随着工艺进步，同样芯片面积可容纳更多的缓存
  - 分级处理，在下一级中采用更大的缓存容量
- 考虑例子：
  - **CPI_{ideal} = 2, 到内存中有100个cycle的失效损失，36%的内存读写指令，2%的L1指令缓存失效率，4%的L1数据缓存失效率**

CPIstalls = 2 + .02x100 + .36*.04*100 = 5.44

  - **再加上一个统一的L2缓存，如果它命中，就为L1缓存带来25个cycle的失效损失；它本身有0.5%的失效率**

CPIstalls = 2 + .02×25 + .36×.04×25 + .005×100 + .36×.005×100 = 3.54

多级高速缓存的考虑因素

- 对于L1和L2高速缓存的设计考虑因素是非常不同的
  - 第一级高速缓存，应当专注于尽量减小命中时间，以支持更短的时钟周期
    - 小容量、小缓存块
  - 第二级高速缓存，应当专注于减少失效率，以减小从很慢的内存中访问数据带来的高失效损失
    - 大容量、大缓存块
- L2高速缓存的存在可以显著降低L1高速缓冲的失效损失，所以L1高速缓存可以容忍较高的失效率，而专注于减少命中时间

对于存储层次的四个问题

- **Q1:** 在更上层次中，一个数据块应到被放到哪？
- **Q2:** 如果数据块在更上层次，应当如何被找到？
- **Q3:** 发生失效时，哪个数据块应当被替换？
- **Q4:** 写数据的时候会发生什么？

问题1、2：数据放在哪、如何找到？

	组的数目	每组中的块数目
直接映射	缓存中总的块数	1
组相联	缓存中总的块数 / 相联度	相联度 (通常2到16)
全相联	1	缓存中总的块数

	定位方法	比较次数
直接映射	索引	1
组相联	对组进行索引，组内比较标签	相联度
全相联	比较所有块的标签	块的数目

问题3:发生失效时，哪个数据块应当被替换？

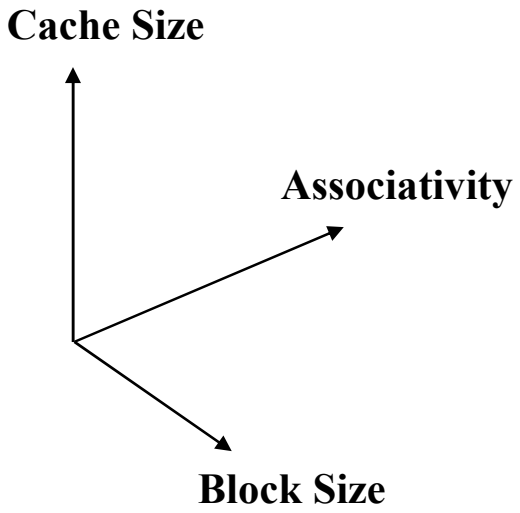
- 对于直接映射缓存，最为简单 – 只有一个选择
- 对于全相联或者多路组相联缓存
  - **FIFO**
  - 随机
  - **LRU（Least Recently Used）**
  - **LFU（Least Frequently Used）**
  - 在后面页面替换算法部分有更详细介绍
- 对于一个**2**路组相联缓存，随机策略比**LRU**策略的失效率高**1.1**倍
- 但是对于更高相联度的缓存（例如，超过**4**路），**LRU**的实现代价过高

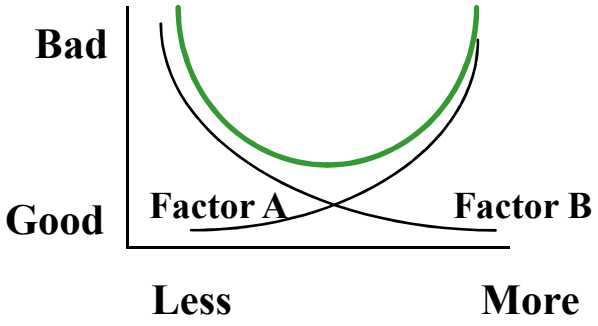
问题4:写数据的时候会发生什么？

- **Write-through** – 数据会被同时写到缓存块以及下一级存储层次中
  - **Write-through**总是带有一个写缓冲，用来优化
- **Write-back** – 数据只写到缓存块中，只有当缓存块被替换的时候，数据才会被写回下一级存储
  - 需要一个“脏位”来标识是否需要写回
- 两种方法的优劣？
  - **Write-through:** 读失效的时候，不用回写被替换的缓存块（更简单、便宜）
  - **Write-back:** 反复的写一个位置的时候，最终只会引发一次对下一级存储的写操作

高速缓存设计空间

- 若干相互影响的维度
  - 缓存容量
  - 块大小
  - 相联度
  - 替换策略
  - **write-through vs write-back**
  - **write allocation**
- 需要权衡
  - 取决于数据访问特性
    - 不同应用、不同系统
  - 还取决于造价
- 简单的设计经常会更好

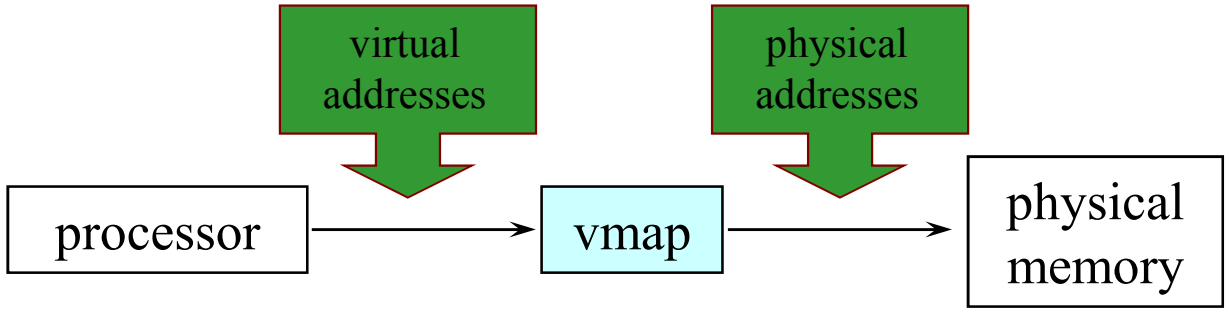




虚拟地址

- 为了更容易地管理进程所需要的内存，我们让进程使用虚拟地址 (逻辑地址)
  - **Virtual addresses are independent of the actual physical location of the data referenced**
  - **OS determines location of data in physical memory**
  - **Virtual addresses are translated by hardware into physical addresses (with help from OS)**
  - **The set of virtual addresses that can be used by a process comprises its virtual address space**

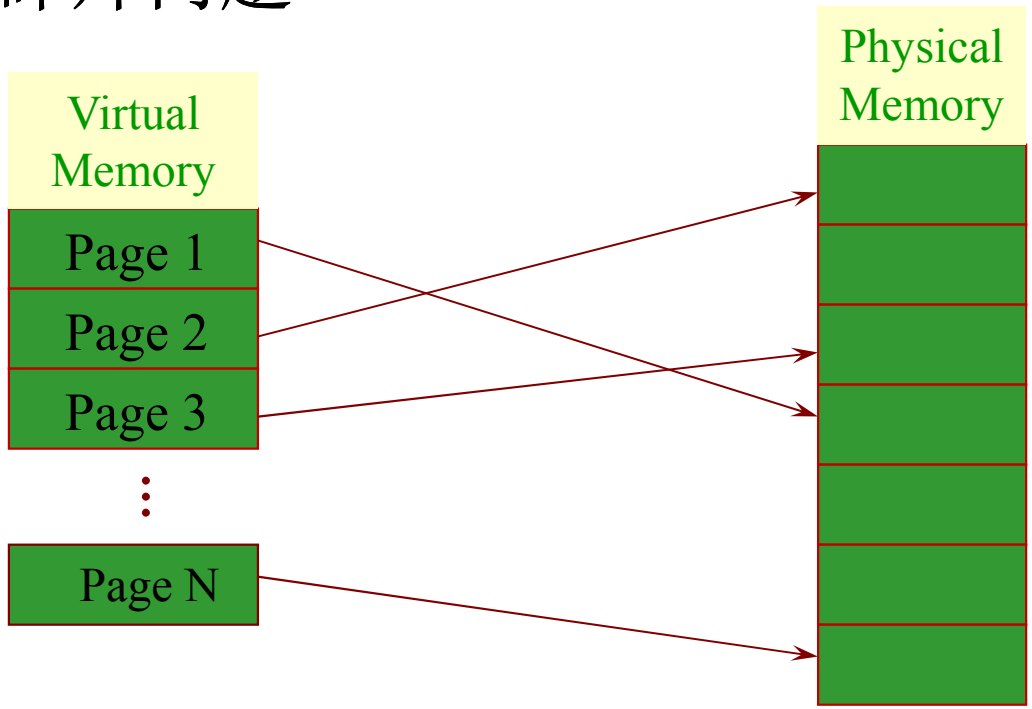
虚拟地址转换



- 有很多种方法做这种转换...

第三种虚拟地址转换的方式：分页

- 分页机制，通过同时在物理内存和虚拟内存中都使用固定大小的单元（但可以灵活映射），来解决外部碎片问题



从用户角度看

- 用户和进程看到的内存还是一片连续的地址空间
  - 虚拟地址空间
- 但事实上，这些页面分布在物理内存（不同位置）
  - 与不同大小分区方案不同
- 用户和程序看不到这种映射

分页机制

- 地址转换
  - Virtual address has two parts: virtual page number and offset
  - Virtual page number (VPN) is an index into a page table
  - Page table determines page frame number (PFN)
  - Physical address is PFN::offset
- 页表
  - Map virtual page number (VPN) to page frame number (PFN)
    - VPN is the index into the table that determines PFN
  - One page table entry (PTE) per page in virtual address space
    - Or, one PTE per VPN

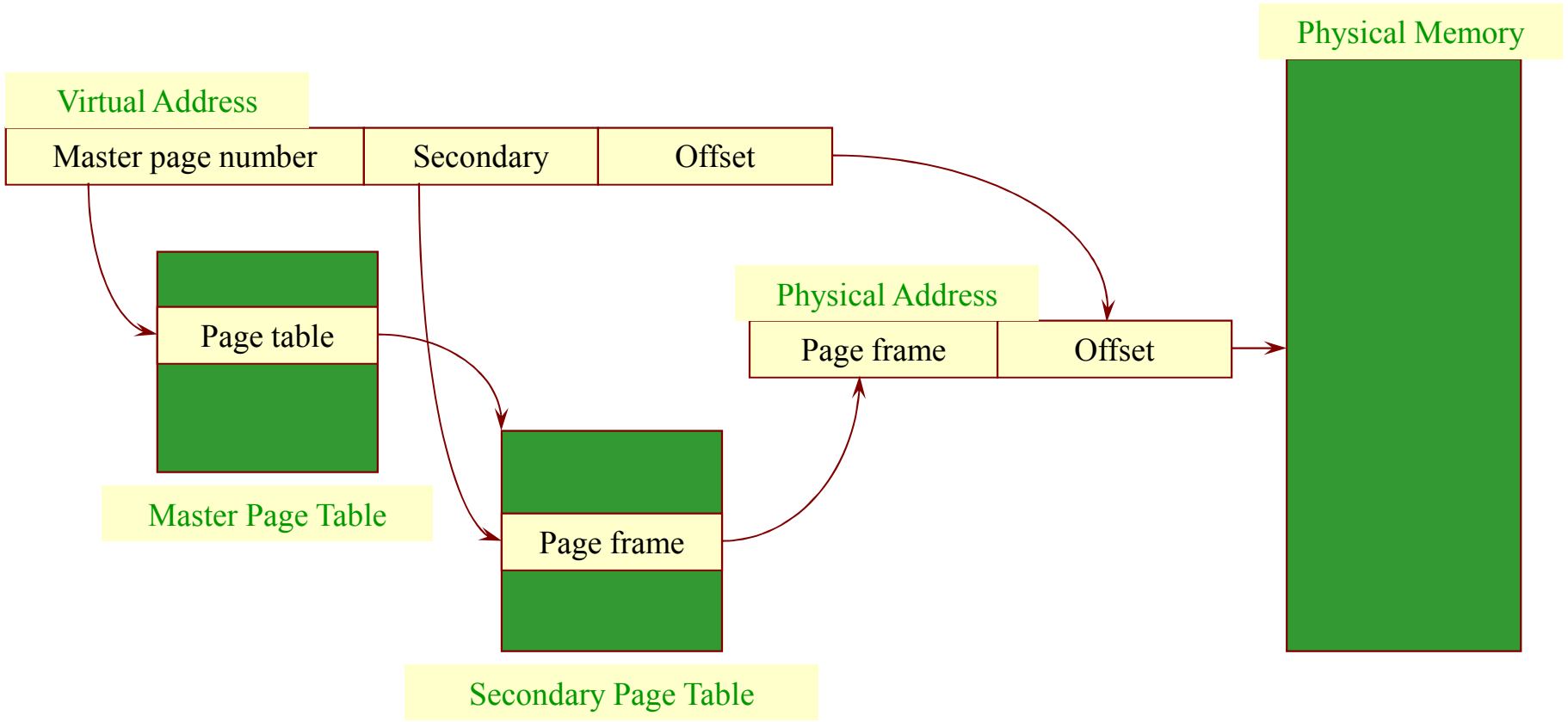
分页机制的问题

- 在最后一个页面中仍然会有内部碎片
  - **But not a big deal**
- 给内存的访问带来额外开销
  - **2 references per address lookup (page table, then memory)**
  - **Solution – use a hardware cache of lookups**
- 存储页表所用的内存空间就很大
  - **Need one PTE per page**
  - **32 bit address space w/ 4KB pages = 2^{20} PTEs**
  - **4 bytes/PTE = 4MB/page table**
  - **25 processes = 100MB just for page tables!**
  - **Solution – page the page tables**

两级页表方法

- 两级页表
  - **Virtual addresses (VAs) have three parts:**
    - **Master page number, secondary page number, and offset**
  - **Master page table maps VAs to secondary page table**
  - **Secondary page table maps page number to physical page**
  - **Offset indicates where in physical page address is located**
- 举例
  - **4K pages, 4 bytes/PTE**
  - **How many bits in offset? 4K = 12 bits**
  - **Want master page table in one page: 4K/4 bytes = 1K entries**
  - **Hence, 1024 secondary page tables. How many bits?**
  - **Master (1K) = 10, offset = 12, secondary = 32 – 10 – 12 = 10 bits**

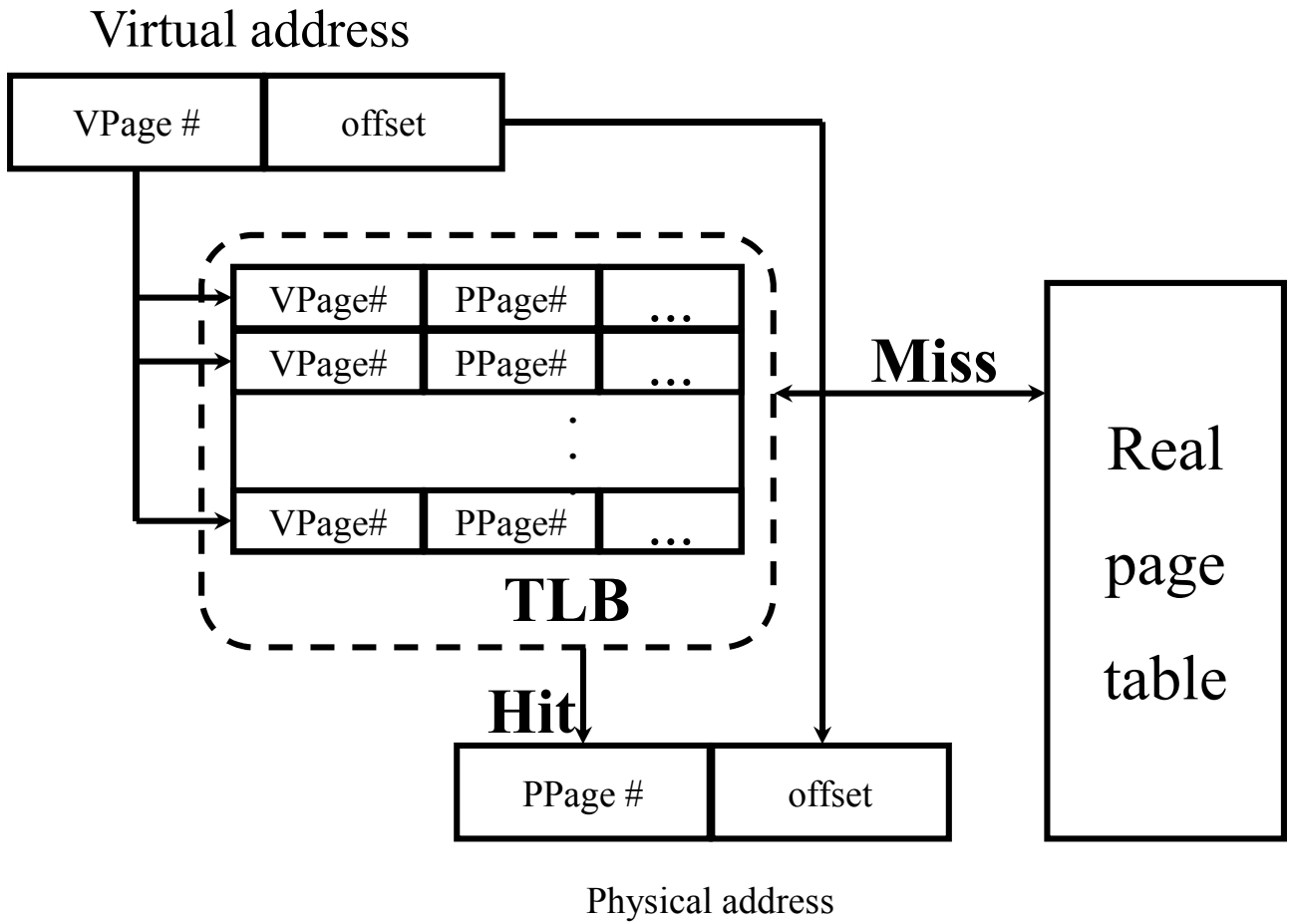
两级页表示意


Physical Memory
Virtual Address
Master page number
Secondary
Offset
Page table
Physical Address
Page frame
Offset
Master Page Table
Page frame
Secondary Page Table

高效地址转换

- 访问一个内存数据，一级页表方法就需要访问两次内存
  - **One lookup into the page table, another to fetch the data**
- 二级页表需要访问三次！
  - **Two lookups into the page tables, a third to fetch the data**
  - **And this assumes the page table is in memory**
- 如何能够减少页面查找的开销？
  - **Cache translations in hardware**
  - **Translation Lookaside Buffer (TLB)**
  - **TLB managed by Memory Management Unit (MMU)**

快表 TLB，Translation Look-aside Buffer



快表

- **Translation Lookaside Buffers**
  - **Translate virtual page #s into PTEs (not physical addrs)**
  - **Can be done in a single machine cycle**
- **TLB由硬件实现**
  - **Fully associative cache (all entries looked up in parallel)**
  - **Cache tags are virtual page numbers**
  - **Cache values are PTEs (entries from page tables)**
  - **With PTE + offset, can directly calculate physical address**
- 局部性原理
  - **Processes only use a handful of pages at a time**
    - **16-48 entries/pages (64-192K)**
    - **Only need those pages to be "mapped"**
  - **Hit rates are therefore very important**

快表的功能

- 当接受一个虚拟地址的时候，硬件首先将它与**TLB**中的所有条目进行对比 (并行)
- 如果找到，就直接使用**TLB**中的地址转换
- 如果没有找到，就采用正常页面查找流程
  - 将**TLB**中的一项替换成刚刚完成的地址转换

输入输出系统

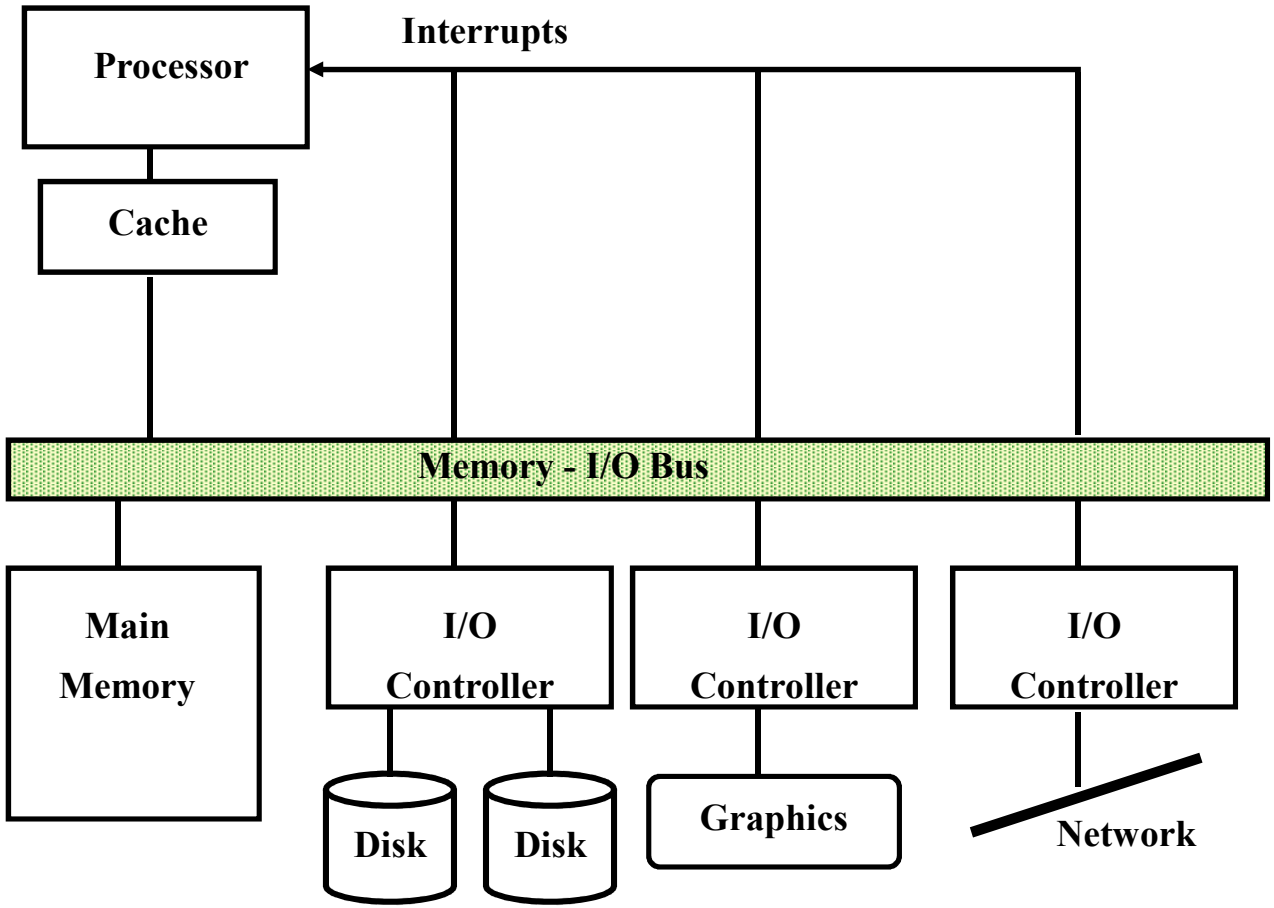
I/O设备的类别

- 三种类别
  - 与人交互
  - 与机器交互
  - 通信

I/O设备之间的差异

- 设备之间在很多方面都存在差异
  - 数据传输率
  - 应用场景
  - 控制复杂度
  - 传输数据的单位
  - …

典型的I/O系统


Processor
Interrupts
Cache
Memory - I/O Bus
Main Memory
I/O Controller
I/O Controller
I/O Controller
Disk
Disk
Graphics
Network

I/O性能的度量方法

- I/O带宽 (吞吐率) – 单位时间内能够通过I/O连接通道的信息量
  1. How much data can we move through the system in a certain time?
  2. How many I/O operations can we do per unit time?
- I/O响应时间 (延迟) – 为了完成一个I/O操作，需要的总时间
  - An especially important performance metric in real-time systems
- 许多应用既需要高吞吐率，又需要短的响应时间

I/O系统性能

设计一个I/O系统来满足一系列带宽/延迟的要求：

1. 找到I/O system的最薄弱部分 – 哪个部分会出问题
   - 处理器和内存系统？
   - 底层互联系统 (e.g., 总线) ？
   - I/O控制器?
   - I/O设备本身?
2. 重新配置最薄弱部分，以满足带宽/延迟需求
3. 检查其它部分，从（1）重新开始

I/O与操作系统

- 操作系统处于I/O硬件与请求I/O操作的程序之间
  - To protect the shared I/O resources, the user program is not allowed to communicate directly with the I/O device
- 操作系统必须能够对I/O设备下指令、处理I/O设备产生的中断、提供可以配置的对共享I/O系统的访问，以及对I/O请求进行合理调度以增强系统整体吞吐率
  - I/O interrupts result in a transfer of processor control to the supervisor (OS) process

I/O系统连接：总线

- 总线：一个共享的通信通路，需要支持很多设备，而这些设备的数据传输率和延迟可能有很大的不同
  - **Advantages**
    - **Versatile – new devices can be added easily and can be moved between computer systems that use the same bus standard**
    - **Low cost – a single set of wires is shared in multiple ways**
  - **Disadvantages**
    - **Creates a communication bottleneck – bus bandwidth limits the maximum I/O throughput**
- 总线的最高速度，还被如下两个因素所限制
  - **The length of the bus**
  - **The number of devices on the bus**

总线类型

- 处理器-内存总线
  - Short and high speed
  - Matched to the memory system to maximize the memory-processor bandwidth
  - Optimized for cache block transfers
- I/O总线 (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
- 底板总线 (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

同步总线与异步总线

- 同步总线 (e.g.,处理器-内存总线)
  - 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
- 异步总线 (e.g., I/O总线)
  - 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(er)

总线仲裁

- 一个总线上的多个设备可能在同时都竞争总线使用权
- 总线仲裁通常试图在以下两个方面取得平衡：
  - **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**
- 四种不同的总线仲裁类型
  - **Daisy chain arbitration**
  - **Centralized arbitration - used in essentially all processor-memory buses and in high-speed I/O buses**
  - **Distributed arbitration by self-selection – each device wanting the bus places a code indicating its identity on the bus**
  - **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)**

总线类型的变化趋势

- 从同步、并行、宽的总线向异步窄总线演进
  - **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
Clock (MHz)	33 – 133	635	50	150
Peak BW (MB/s)	128 – 1064	300	100	375 (3 Gbps)

处理器进行I/O操作的方法

- 特殊的I/O指令
  - Must specify both the device and the command
- 内存映射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

I/O设备与处理器之间的数据传输

- 可编程I/O
- 中断驱动I/O
- 直接内存访问 (DMA)

Table 11.1 I/O Techniques

	No Interrupts	Use of Interrupts
I/O-to-memory transfer through processor	Programmed I/O	Interrupt-driven I/O
Direct I/O-to-memory transfer		Direct memory access (DMA)

直接内存访问（Direct Memory Access）

- 对于高带宽的设备（例如硬盘），仅仅使用中断驱动I/O会浪费很多处理器时间
- DMA –I/O控制器可以不通过处理器，直接传输数据
  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
- 一个系统中可能有多个DMA设备
  - Processor and I/O controllers contend for bus cycles and for memory

DMA数据不一致问题

- 带有高速缓存的系统中，缓存中和内存中都有数据
  - **For a DMA read (from disk to memory) – the processor will be using stale data if that location is also in the cache**
  - **For a DMA write (from memory to disk) and a write-back cache – the I/O device will receive stale data if the data is in the cache and has not yet been written back to the memory**
- 通过如下方法解决不一致问题
  1. **Routing all I/O activity through the cache – expensive and a large negative performance impact**
  2. **Having the OS selectively invalidate the cache for an I/O read or force write-backs for an I/O write (flushing)**
  3. **Providing hardware to selectively invalidate or flush the cache – need a hardware snooper**

硬盘

- 柱面, 磁头, 扇区 (CHS), 磁道, …

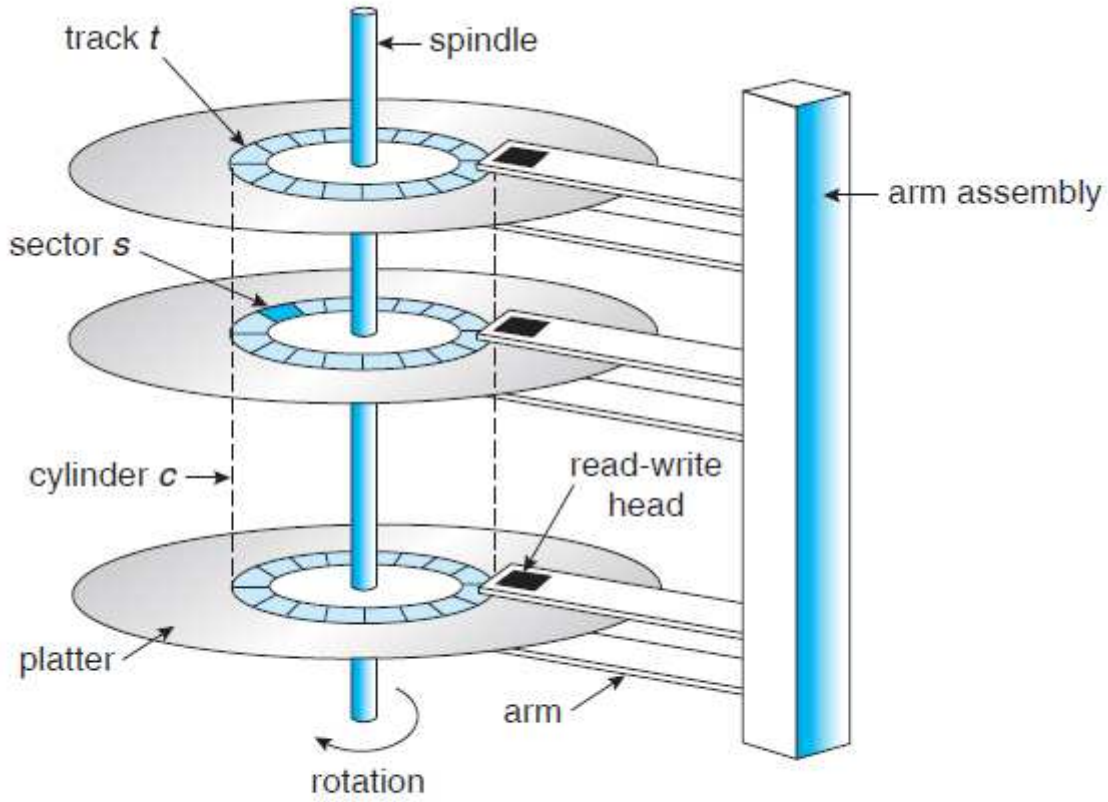


Figure 11.1 Moving-head disk mechanism.

Figure from "Operating System Concepts Essentials", A. Silberschatz et at., John Wiley & Sons. Inc

硬盘性能指标

- *访问时间*，为以下时间之和：
  - ***Seek time:* The time it takes to position the head at the desired track**
  - ***Rotational delay* or *rotational latency:* The time its takes for the beginning of the sector to reach the head**
- *传输时间*，传输数据的时间

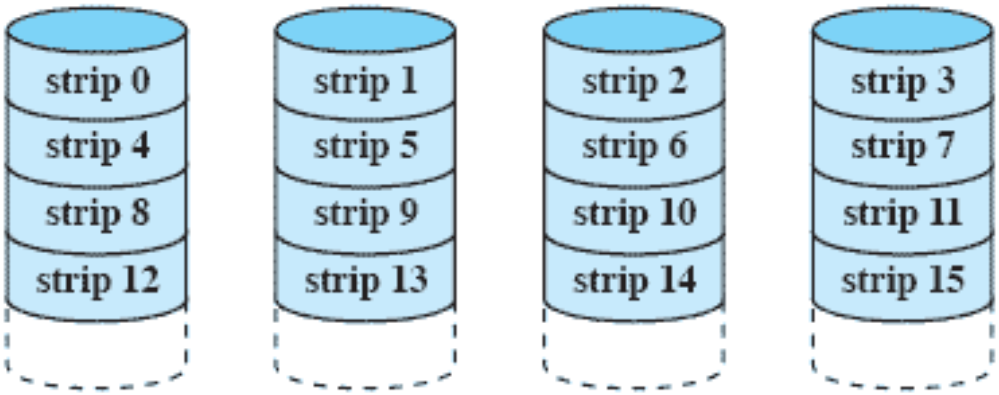
多硬盘系统

- 硬盘I/O的总体性能可以靠将操作分布在多个读写磁头上完成
- 或者采用多个磁盘并行工作
- 如果存在校验信息，就可以纠错

廉价冗余磁盘阵列RAID

- 廉价冗余磁盘阵列
  - **Redundant Array of Independent Disks**
- 在操作系统及上层看来，多个物理硬盘组成了一个逻辑磁盘
- 数据分布在阵列中的多块物理硬盘中
- 冗余的磁盘容量，用来存储校验信息，以备出错时进行数据恢复

RAID 0 – 交织

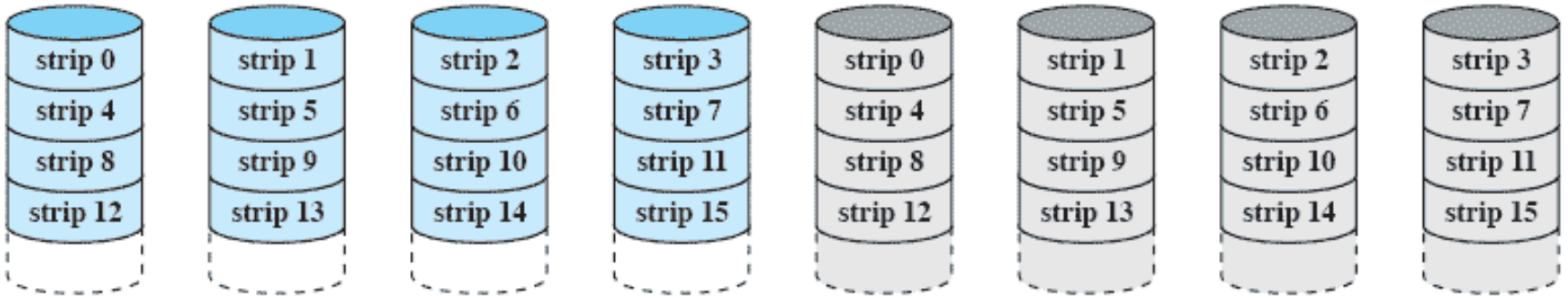


(a) RAID 0 (non-redundant)

- 不是真的**RAID** – 没有冗余
- 出错时不可恢复
- 由于并行读写，速度非常快

RAID 1 – 镜像

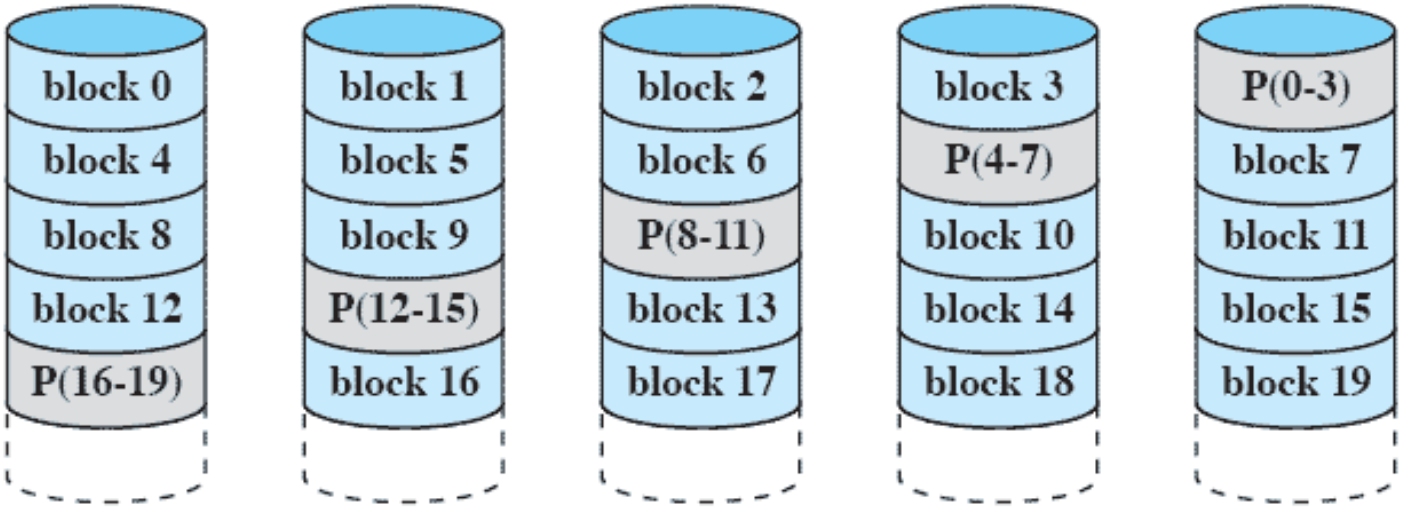
- 通过复制，而不是校验码，来实现冗余
- 可以进行并行读（两个请求各向一组发出）
- 数据恢复非常容易



(b) RAID 1 (mirrored)

RAID 5 块分布校验

- 将块校验码分布在各个磁盘中
- 对于用N个相同硬盘组成的系统，可用容量为N-1

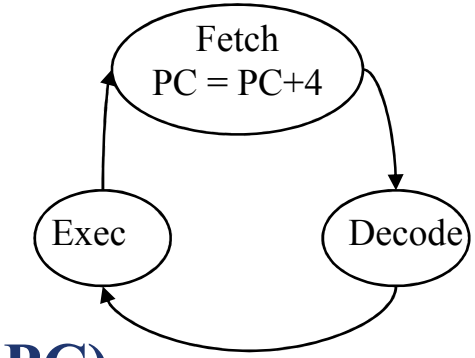


(f) RAID 5 (block-level distributed parity)

处理器结构

处理器：数据通路与控制

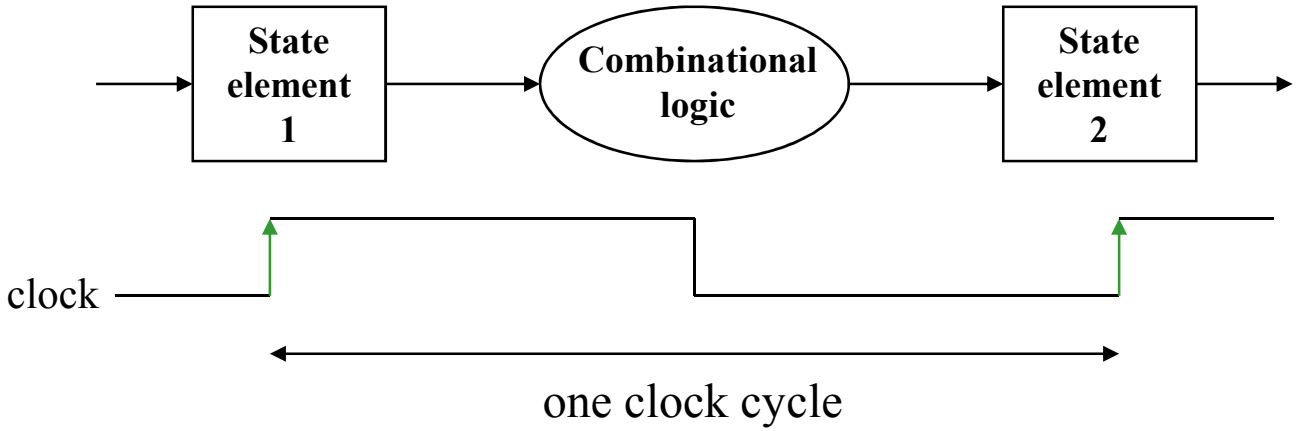
- 简化的MIPS
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- 通用实现
  - use the program counter (PC) to supply the instruction address and fetch the instruction from memory (and update the PC)
  - decode the instruction (and read registers)
  - execute the instruction



- 除了j以外的所有指令，都会在读寄存器后用到ALU
  - How? memory-reference? arithmetic? control flow?

时钟方式

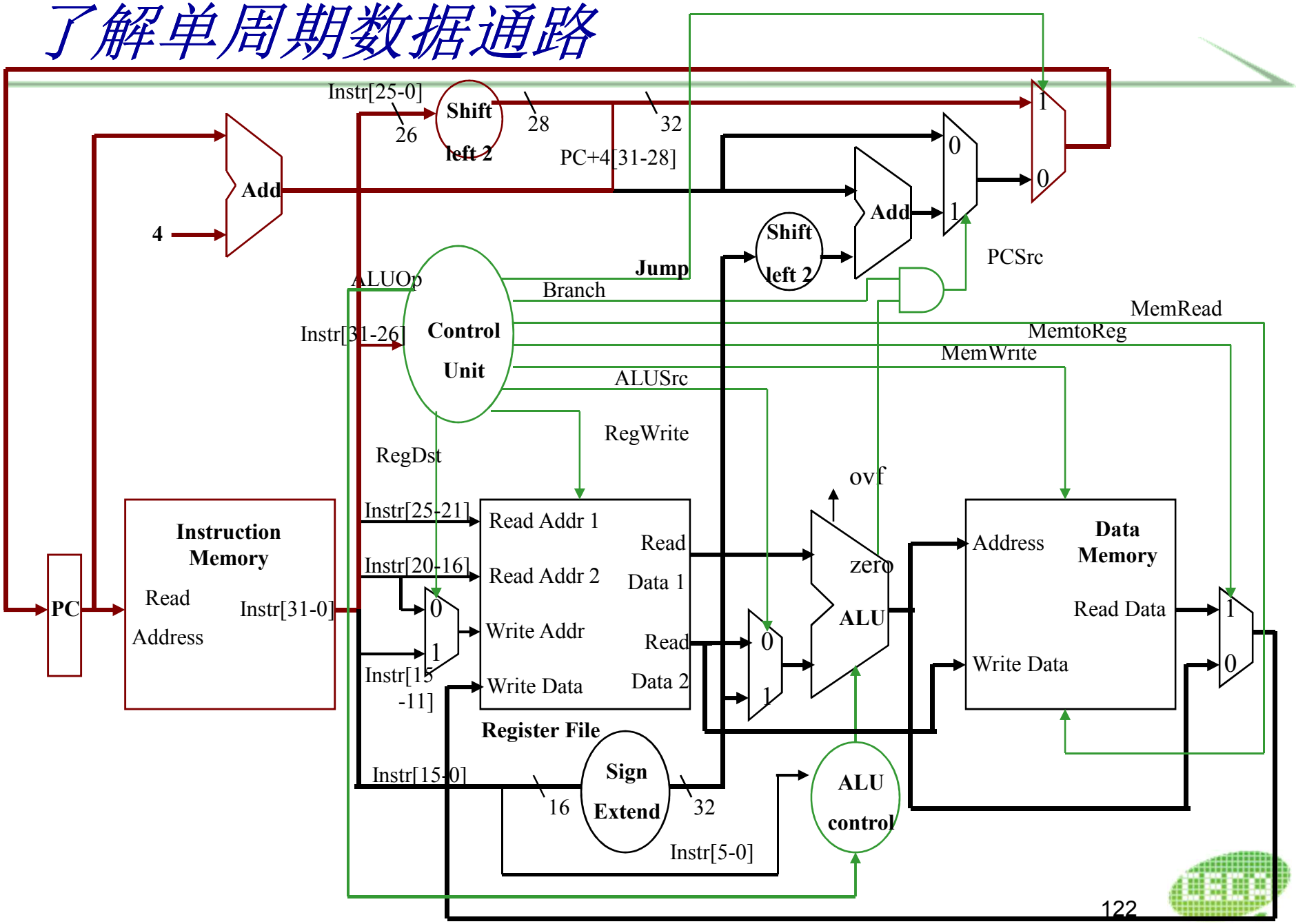
- 时钟方式定义何时读写信号
  - An edge-triggered methodology
- 通常的执行流程
  - read contents of state elements
  - send values through combinational logic
  - write results to one or more state elements



建立一个简单的数据通路

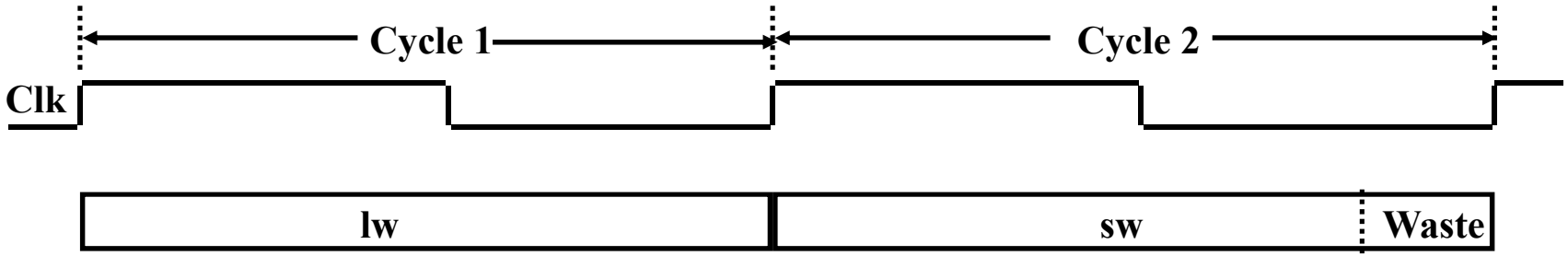
- 将数据通路的各个模块组装起来，加入控制线，并添加适当的选择器（**multiplexors**）
- 单周期**Single cycle** 设计 – 每个指令的取指、译码和执行在一个时钟周期完成
  - **no datapath resource can be used more than once per instruction, so some must be duplicated (e.g., separate Instruction Memory and Data Memory, several adders)**
  - **multiplexors needed at the input of shared elements with control lines to do the selection**
  - **write signals to control writing to the Register File and Data Memory**
- 时钟时间由最长路径决定

了解单周期数据通路


Instr[25-0]
Shift
left 2
26
28
32
PC+4[31-28]
Add
4
Add
Shift
left 2
PCSrc
Jump
Branch
ALUOp
Instr[31-26]
Control
Unit
MemRead
MemtoReg
MemWrite
ALUSrc
RegWrite
RegDst
ovf
Instr[25-21]
Instr[20-16]
Instr[15
-11]
Instruction
Memory
Read
Address
Instr[31-0]
PC
Read Addr 1
Read Addr 2
Write Addr
Write Data
Read
Data 1
Read
Data 2
Register File
zero
ALU
Address
Data
Memory
Read Data
Write Data
Instr[15-0]
Sign
Extend
16
32
Instr[5-0]
ALU
control
122

单周期设计的优缺点

- 并不能有效利用时钟周期 – 时钟周期必须能容纳最慢的指令
  - **especially problematic for more complex instructions like floating point multiply**



- 由于在一个时钟周期内功能单元不能被共享（例如加法器），所以必须复制多份，会造成浪费

- 最为简单、易于理解

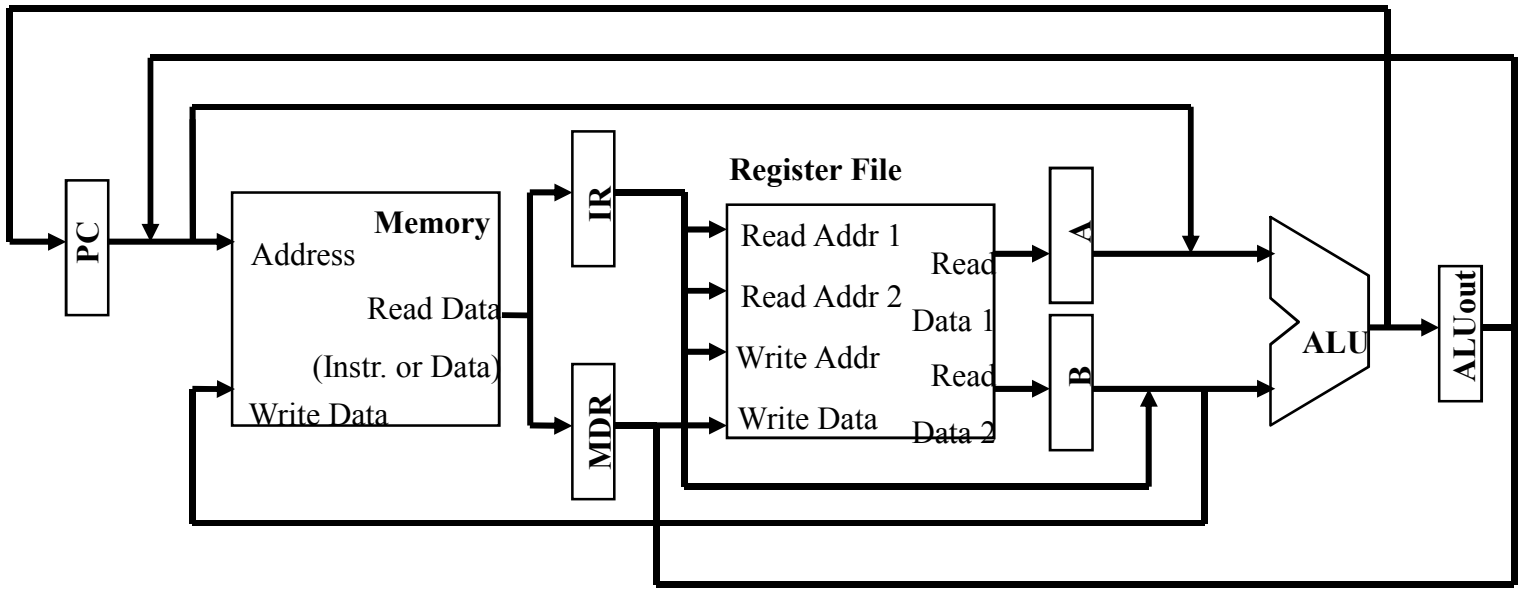
多周期数据通路

- 使一条指令花多个周期来完成
  - **Break up instructions into steps where each *step* takes a cycle while trying to**
    - **balance the amount of work to be done in each step**
    - **restrict each cycle to use only one major functional unit**
  - **Not every instruction takes the *same* number of clock cycles**

- 除了可以实现更快的时钟频率，多周期数据通路还允许一条指令在不同周期使用同一个功能单元
  - **only need one memory – but only one memory access per cycle**
  - **need only one ALU/adder – but only one ALU operation per cycle**

多周期数据通路 2

- 在每个周期结束时
  - **Store values needed in a later cycle by the current instruction in an internal register (not visible to the programmer). All (except IR) hold data only between a pair of adjacent clock cycles (no write control signal needed)**

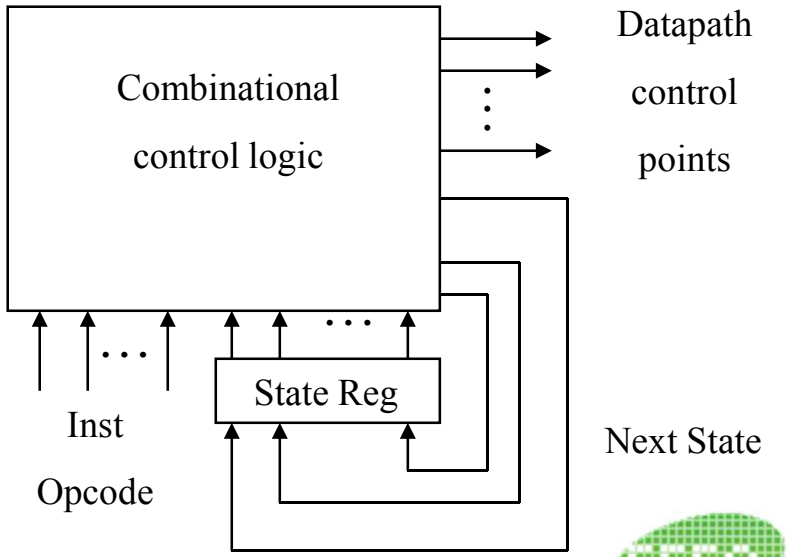


IR – Instruction Register
MDR – Memory Data Register
A, B – regfile read data registers
ALUout – ALU output register

- Data used by subsequent instructions are stored in programmer visible registers (i.e., register file, PC, or memory)

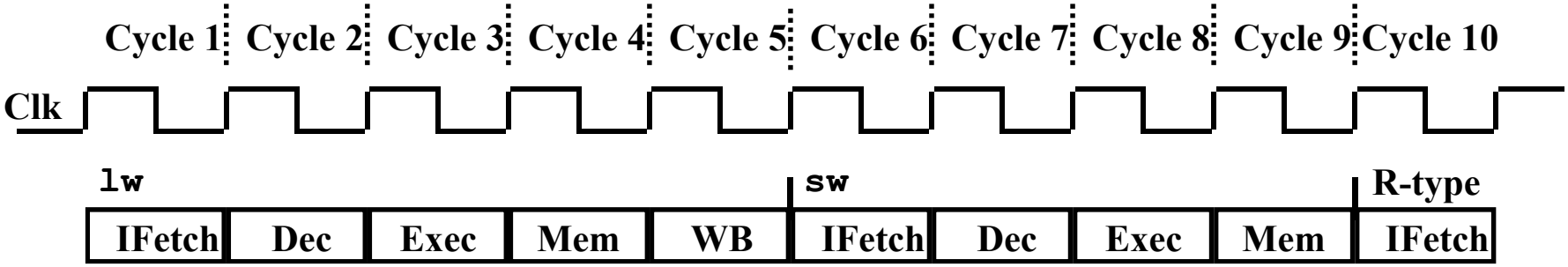
多周期数据通路的控制单元

- 多周期数据通路的控制信号并不只由指令中各个位显式决定
  - **e.g., op code bits tell what operation the ALU should be doing, but *not* what instruction cycle is to be done next**
- 必须用有限状态机（**FSM**）来控制
  - **a set of states (current state stored in State Register)**
  - **next state function (determined by current state and the input)**
  - **output function (determined by current state and the input)**



多周期设计的优缺点

- 可以有效地使用时钟周期 – 时钟周期只需容纳最慢指令多个步骤中的一步

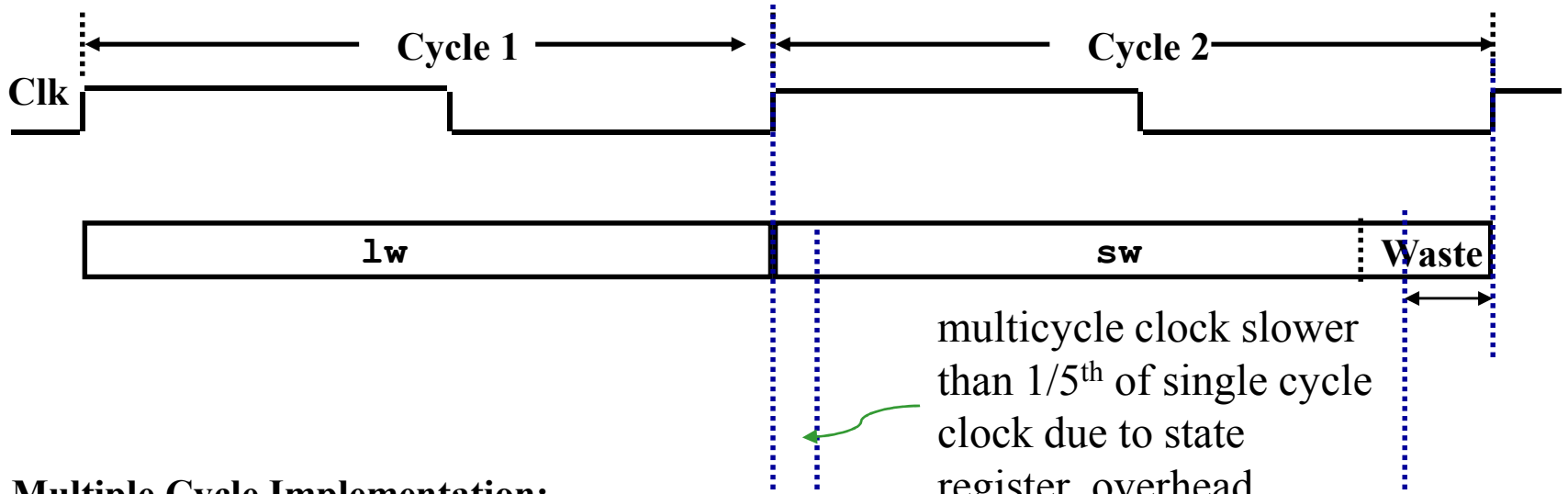


- 多周期数据通路还允许一条指令在不同周期使用同一个功能单元

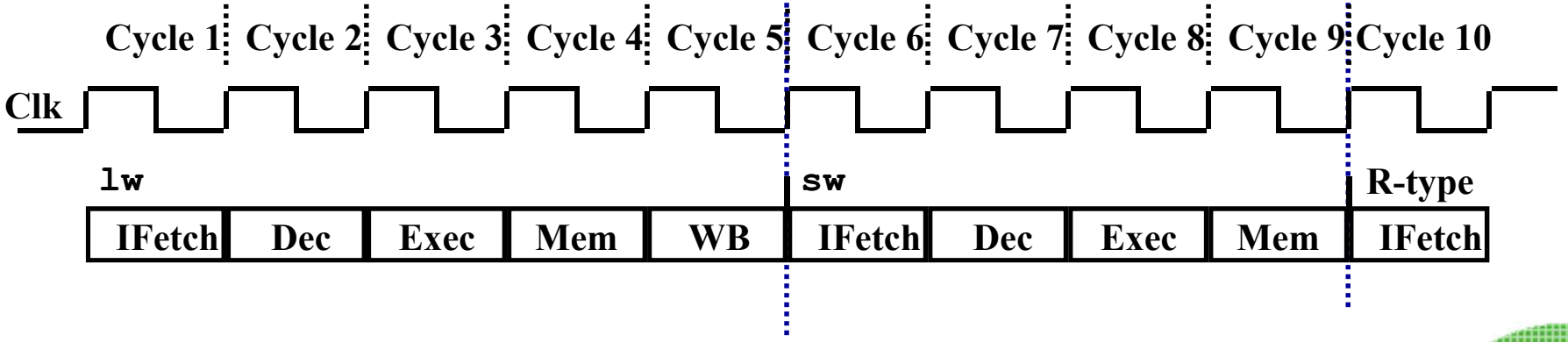
- 需要额外的内部状态寄存器、更多的选择器、更复杂的（**FSM**）控制

单周期 v.s. 多周期

Single Cycle Implementation:



Multiple Cycle Implementation:



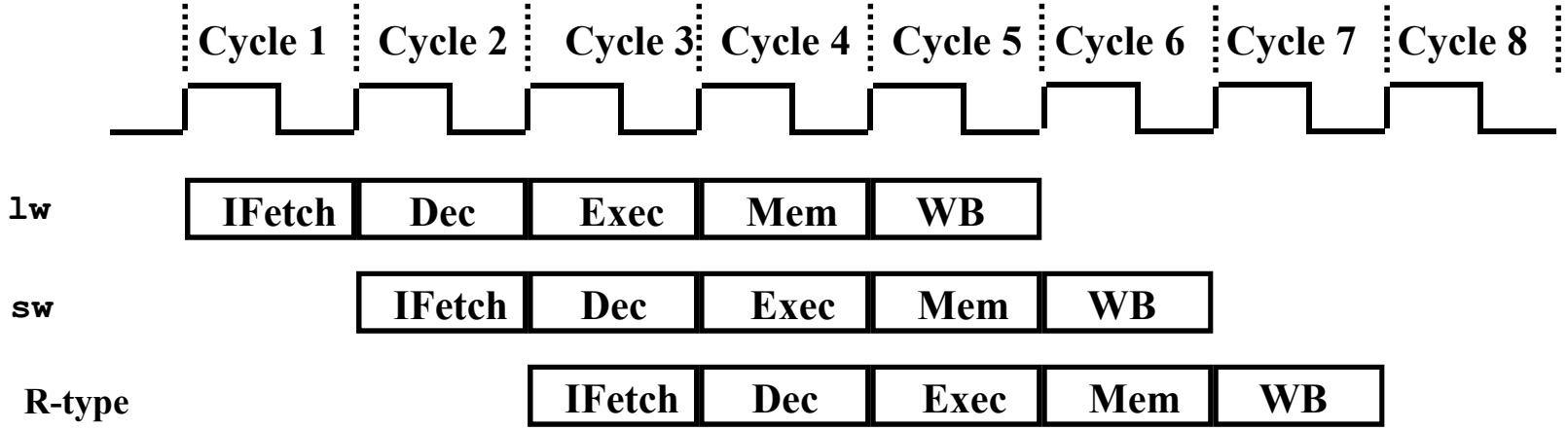
如何比多周期处理器更快？

- 将时钟周期进一步细分
  - **There is a point of diminishing returns where as much time is spent loading the state registers as doing the work**
- 在当前指令没有完成时，就开始读取和执行一下条指令
  - **Pipelining – (all?) modern processors are pipelined for performance**
  - **Remember the performance equation:**

    CPU time = CPI * Cycle Time * IC
- 一次读取并执行多条指令
  - **Superscalar processing**

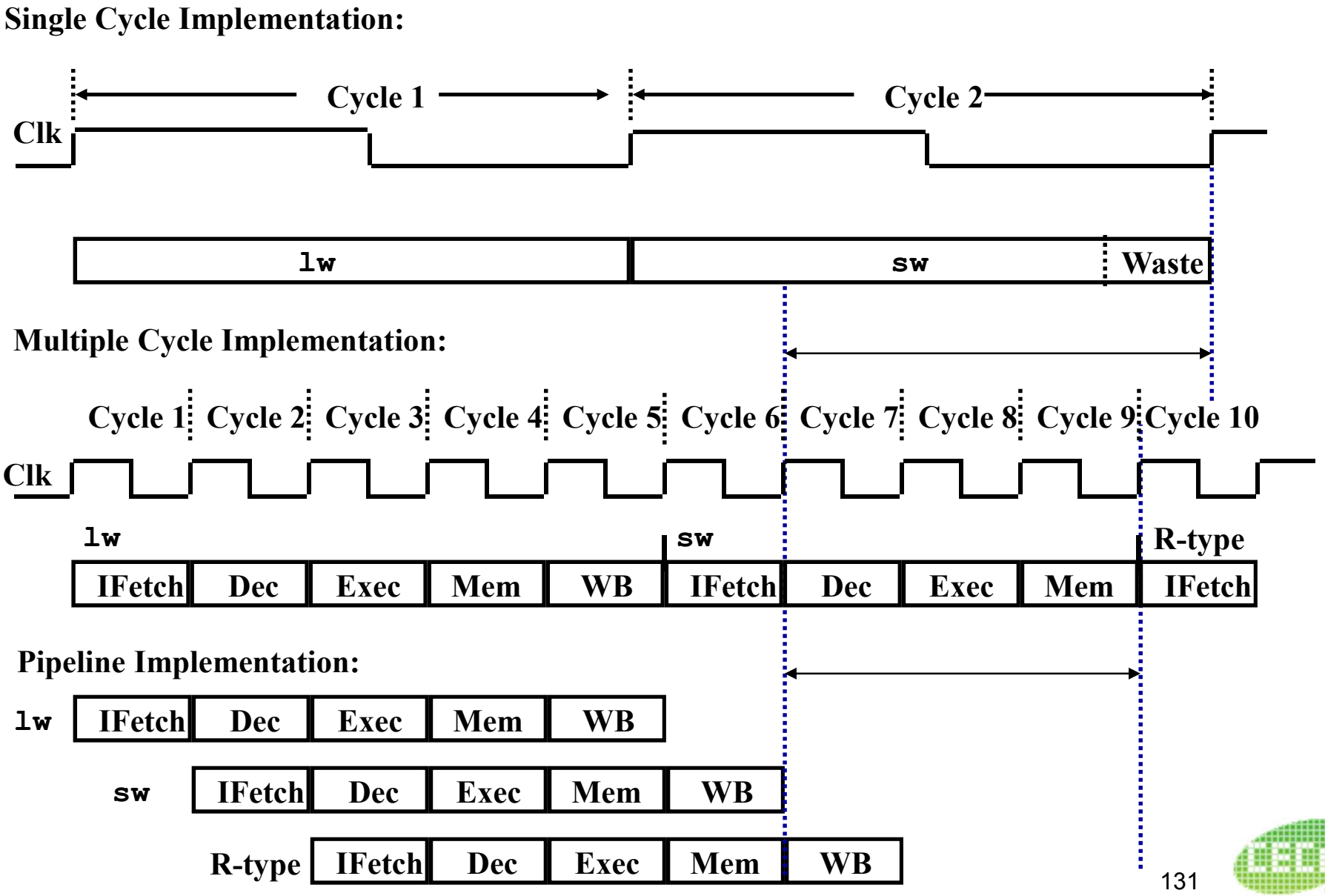
流水线结构的MIPS处理器

- 在当前指令完成之前，开始下一条指令
  - **improves throughput - total amount of work done in a given time**
  - **instruction latency (execution time, delay time, response time - time from the start of an instruction to its completion) is *not* reduced**



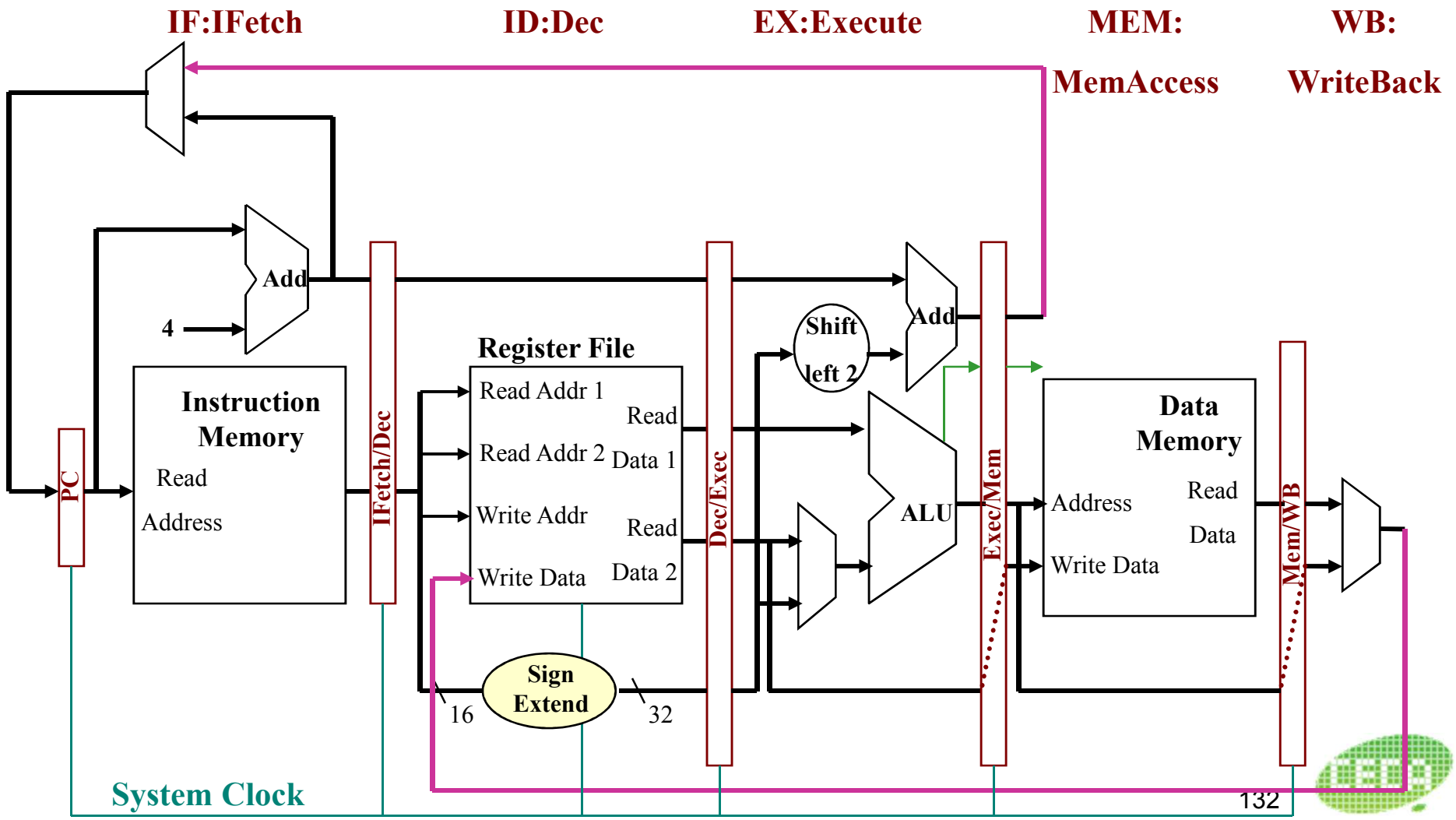
- clock cycle (pipeline stage time) is limited by the slowest stage
- for some instructions, some stages are wasted cycles

单周期 v.s. 多周期 v.s. 流水线

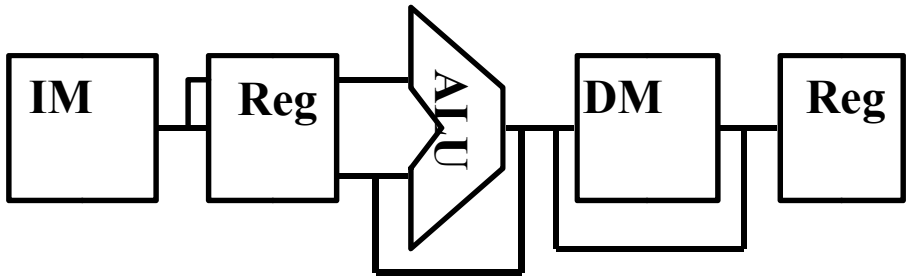


在MIPS流水线数据通路中的修改

- 在MIPS数据通路中，我们需要添加和修改哪些地方？
  - State registers between each pipeline stage to isolate them

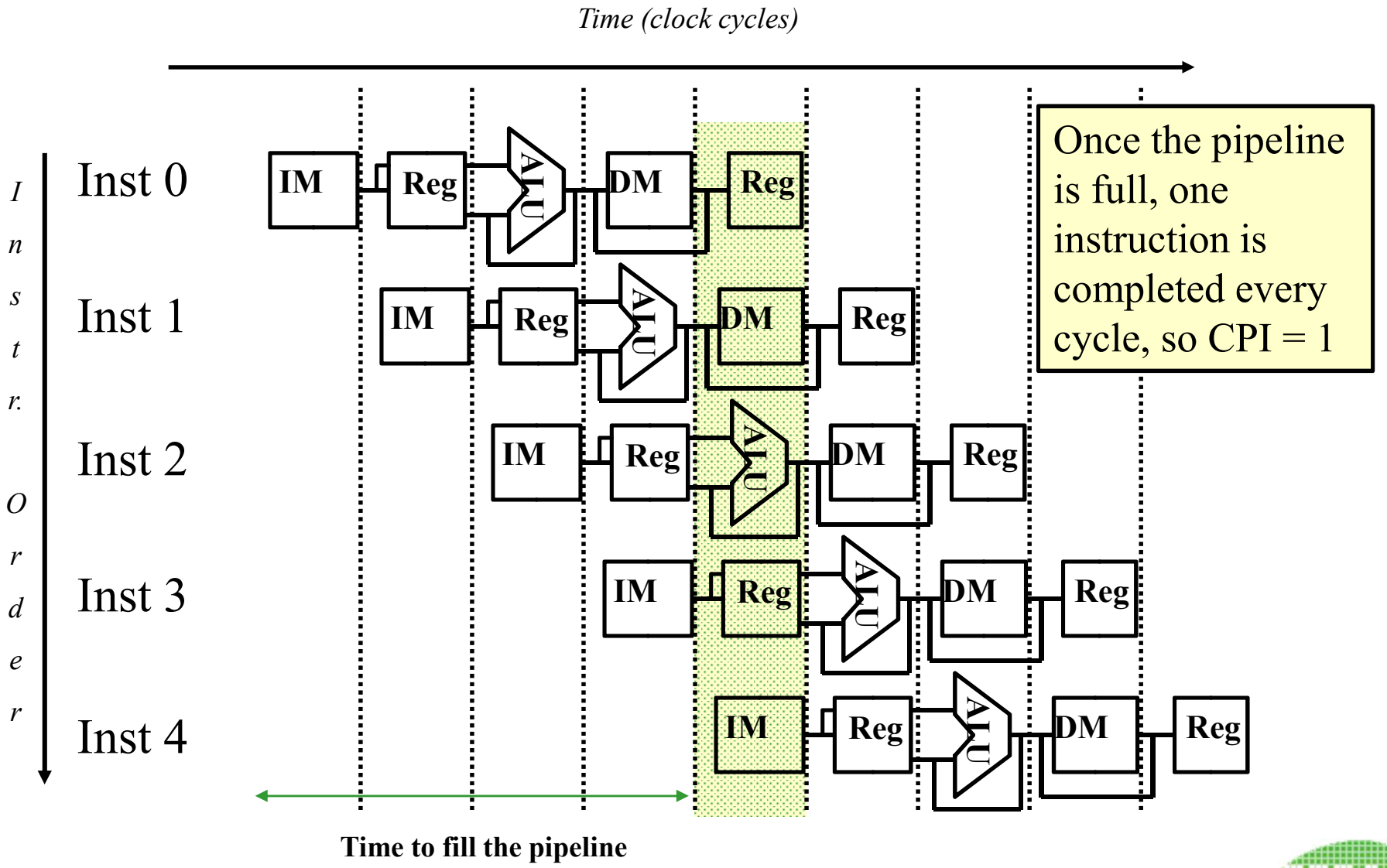


MIPS流水线的图形化表示



- 可以来帮助理解如下问题：
  - **How many cycles does it take to execute this code?**
  - **What is the ALU doing during cycle 4?**
  - **Is there a hazard, why does it occur, and how can it be fixed?**

流水线的目的：性能！


Time (clock cycles)
Instr. Order
Inst 0
Inst 1
Inst 2
Inst 3
Inst 4
IM
Reg
ALU
DM
Reg
Once the pipeline is full, one instruction is completed every cycle, so CPI = 1
Time to fill the pipeline

流水线中的问题

- 流水线冒险
  - 结构冒险: **attempt to use the same resource by two different instructions at the same time**
  - 数据冒险: **attempt to use data before it is ready**
    - **An instruction's source operand(s) are produced by a prior instruction still in the pipeline**
  - 控制冒险: **attempt to make a decision about program control flow before the condition has been evaluated and the new PC target address calculated**
    - **branch instructions**
- 总可以通过等待的方法来解决这些冒险
  - **pipeline control must detect the hazard**
  - **and take action to resolve hazards**

流水线技术

- 所有现代处理器都用到流水线技术
- 流水线并不能改进单个任务的延迟，但是可以提高整个工作的吞吐率
- 潜在的加速比: 使CPI变为1，并得到更快的时钟周期
- 流水线的速度受限于最慢的流水线级
  - **Unbalanced pipe stages makes for inefficiencies**
  - **The time to "fill" pipeline and time to "drain" it can impact speedup for deep pipelines and short code runs**
- 必须坚持和解决流水线冒险
  - **Stalling negatively affects CPI (makes CPI worse than the ideal of 1)**

流水线冒险

- 结构冒险
  - Design pipeline to eliminate structural hazards（复制某些关键部件）
- 数据冒险 – **read before write**
  - Use data forwarding inside the pipeline
  - For those cases that forwarding won't solve (e.g., load-use) include hazard hardware to insert stalls in the instruction stream
- 控制冒险 – **`beq, bne, j, jr, jal`**
  - Stall – hurts performance
  - Move decision point as early in the pipeline as possible – reduces number of stalls at the cost of additional hardware
  - Delay decision (requires compiler support) – next slide
  - Predict – with even more hardware, can reduce the impact of control hazard stalls even further if the branch prediction (BHT) is correct and if the branched-to instruction is cached (BTB)

解决控制冒险的延时转移技术

- 当遇到分支指令时，流水线可能暂停
  - 为了尽量保证流水线的执行效率，在分支指令之后插入一条有效的指令，而分支指令好像被延时了
  - 通常指令序列的调整由编译器自动进行
  - 调整指令序列时不能改变原有程序的数据关系
- 不适用于需要更多延迟槽来填充的更深流水线级

能不能得到更好的性能？

- **CPU Time = IC x CPI x Cycle Length**

- 到现在我们能够得到的最好**CPI**：**CPI = 1**

如果我们想要更好的性能该怎么办？

实现更好的性能

- 两个选择:
  - **Increase the depth of the pipeline to increase the clock rate – superpipelining (more details to come)**
  - **Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue**

- 流水线一级中允许多条指令，能够使CPI小于1
  - **So instead we use IPC: instructions per clock cycle**
    - **E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4**
  - **If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?**

指令与机器并行

- 程序的指令级并行（**Instruction-level parallelism，ILP**）– 描述在一个程序中，一个理想处理器有可能在同一时刻执行的平均指令数目
  - **Mostly determined by the number of true (data) dependencies and procedural (control) dependencies in relation to the number of other instructions**

- 处理器机器级别的并行度 – 衡量处理器能够在ILP中获益的能力程度
  - Determined by the number of instructions that can be fetched and executed at the same time
- 为了实现高性能，需要同时具有较高ILP和机器级别的并行度

多发射处理器类型

- 静态多发射处理器 (aka VLIW)
  - Decisions on which instructions to execute simultaneously are being made statically (at compiler time by the compiler)
  - E.g., Intel Itanium and Itanium 2 for the IA-64 ISA – EPIC (Explicit Parallel Instruction Computer), and TI TMS320C6x
- 动态多发射处理器 (aka Superscalar)
  - Decisions on which instructions to execute simultaneously are being made dynamically (at run time by the HW)
  - E.g., IBM Power 2, Pentium 4, MIPS R10K, HP PA 8500

超级流水线处理器

- 提高流水线的深度，可以导致更短的时钟周期（以及在同一时刻，流水线中更多的指令）
  - **The higher the degree of superpipelining, the more forwarding/hazard hardware needed, the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time), and the bigger the clock skew issues (i.e., because of faster and faster clocks)**

超级流水线 **vs** 超标量

- 超级流水线处理器具有更长的指令延迟，与超标量比，如果存在真数据相关时，会导致性能下降
- 超标量处理器对资源冲突更加敏感 – 不过我们可以通过添加硬件的方法来解决这个问题

指令发射与完成的策略

- 指令发射 – 引发指令的执行
  - **Instruction lookahead capability – ability of the processor to fetch, decode and issue instructions beyond the current point of execution to try to find more instructions to issue**

- 指令完成 – 指令执行的结束
  - **Processor lookahead capability – ability of the processor to examine issued instructions beyond the current point of execution to try to find more instructions to complete**

--- 按序发射、按序完成
--- 按序发射、乱序完成
--- 乱序发射、乱序完成

写相关 (output dependency)

- 在按序发射、乱序完成的处理器中，还有一种情况会暂停住指令的发射，假设

  I1 – writes to R3
  I2 – writes to R3
  I5 – reads R3

  - If the I1 write occurs after the I2 write, then I5 reads an incorrect value for R3
  - I2 has an output dependency on I1 – write before write
    - The issuing of I2 would have to be stalled if its result might later be overwritten by an previous instruction (i.e., I1) that takes longer to complete – the stall happens before instruction issue

- 虽然按序发射、乱序完成技术会提高性能，但是需要更多的依赖性检查硬件
  - Dependency checking needed to resolve both read before write *and* write before write

反相关

- 在乱序发射、乱序完成处理器中，还必须处理数据反相关 – 当后一条指令（但是先完成了）产生一个数据，将前一条指令（后提交）需要的源数据给覆盖时产生的问题

```
R3 := R3 * R5
R4 := R3 + 1
R3 := R5 + 1
```

True data dependency

Output dependency Antidependency

- 和真数据相关的限制相似，只是反过来
  - Instead of the later instruction using a value (not yet) produced by an earlier instruction (read before write), the later instruction produces a value that destroys a value that the earlier instruction (has not yet) used (write before read)

各种相关

- 下面每种数据相关
  - **True data dependencies (read before write)**
  - **Antidependencies (write before read)**
  - **Output dependencies (write before write)**

  } storage conflicts

  都操作寄存器（或者另外的存储）
- 真数据相关 显示了程序中的数据流和信息
- 反相关和写相关 由于寄存器数目有限，程序必须在不同计算中复用寄存器而产生
- 当指令被乱序发射时，寄存器与值之间的原本关系被破坏，于是产生了寄存器的值冲突

存储资源冲突与寄存器重命名

- 可以通过增加或复制相关的硬件资源，来减轻存储冲突
  - **Provide additional registers that are used to reestablish the correspondence between registers and values**
    - **Allocated dynamically by the hardware in SS processors**
- 寄存器重命名 – 处理器将指令中原本的寄存器换成另外一个新的内部寄存器 (对外不可见)

```
R3 := R3 * R5          R3b := R3a * R5a
R4 := R3 + 1     ⇨     R4a := R3b + 1
R3 := R5 + 1           R3c := R5a + 1
```

- The hardware that does renaming assigns a “replacement” register from a pool of free registers and releases it back to the pool when its value is superseded and there are no outstanding references to it

向量处理机

为什么使用向量处理器？

- 一个向量指令，就可以指定很多工作 – 相当于执行一整个循环
- 在向量中每个结果的计算都相互独立，所以硬件不用在向量内检查数据冒险
- 硬件只需要检查向量指令之间的数据冒险，于是只要每次向量整体操作时检查即可
- 访问内存的向量指令有程序已知的访问模式
- 因为一整个循环都被易于预测行为的向量指令代替，不再存在由于循环分支引发的控制冒险

基本向量体系结构

- 有两种主要的向量处理器体系结构类型：向量寄存器处理器和向量内存-内存处理器
  - **In a vector-register processor, all vector operations — except load and store — are among the vector registers.**
  - **In a memory-memory vector processor, all vector operations are memory to memory.**

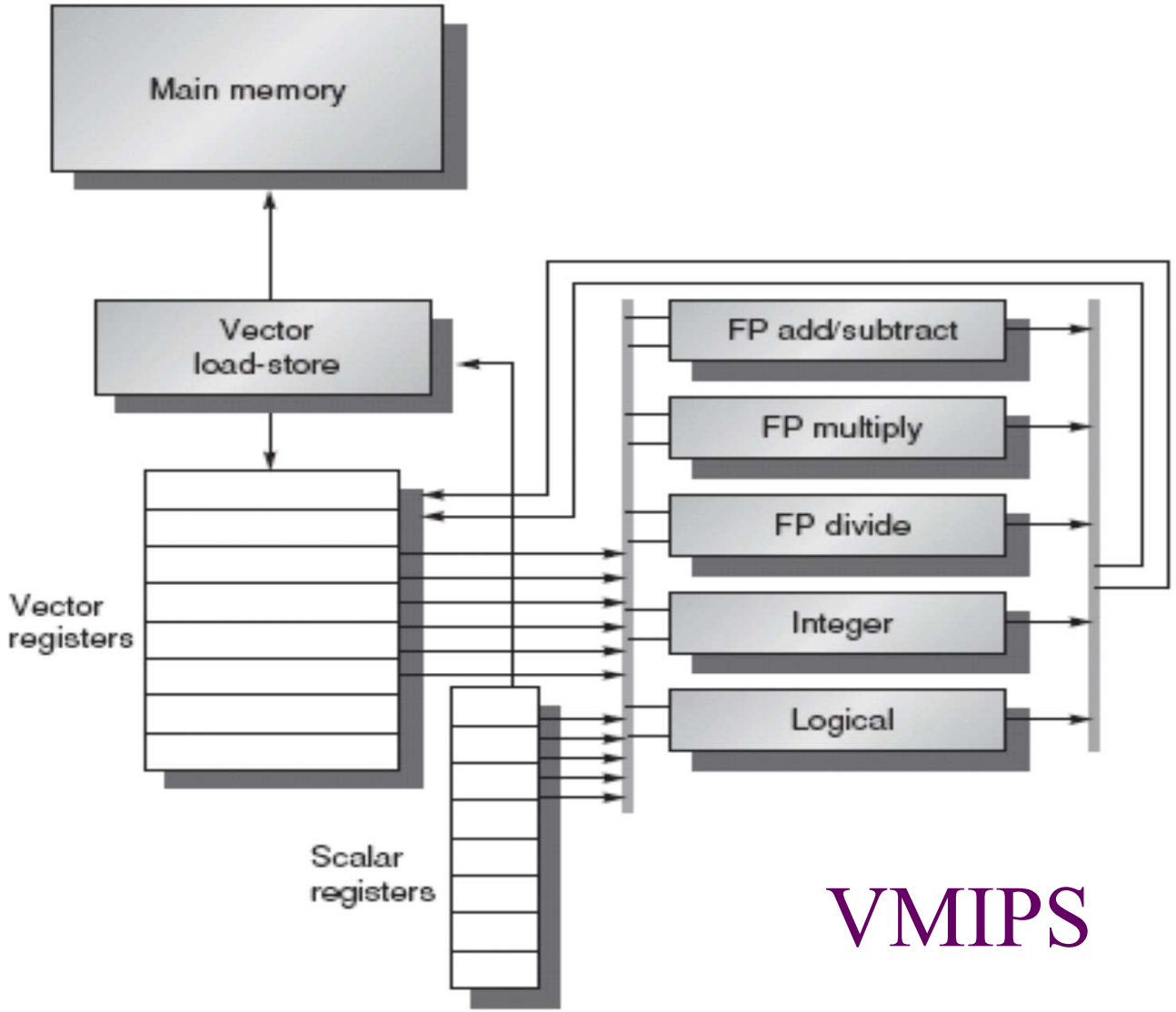
向量内存-内存结构 v.s. 向量寄存器结构

- 向量内存-内存结构 (VMMA) 需要更大的内存访问带宽，为什么？
  - All operands must be read in and out of memory
- VMMAs使多个向量操作难于相互重叠，为什么？
  - Must check dependencies on memory addresses
- VMMAs产生了更大的启动延时
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 elements

⇒ *除了CDC的后继机型 (Cyber-205, ETA-10)，从Cray-1开始所有主要的向量机都采用向量寄存器结构*

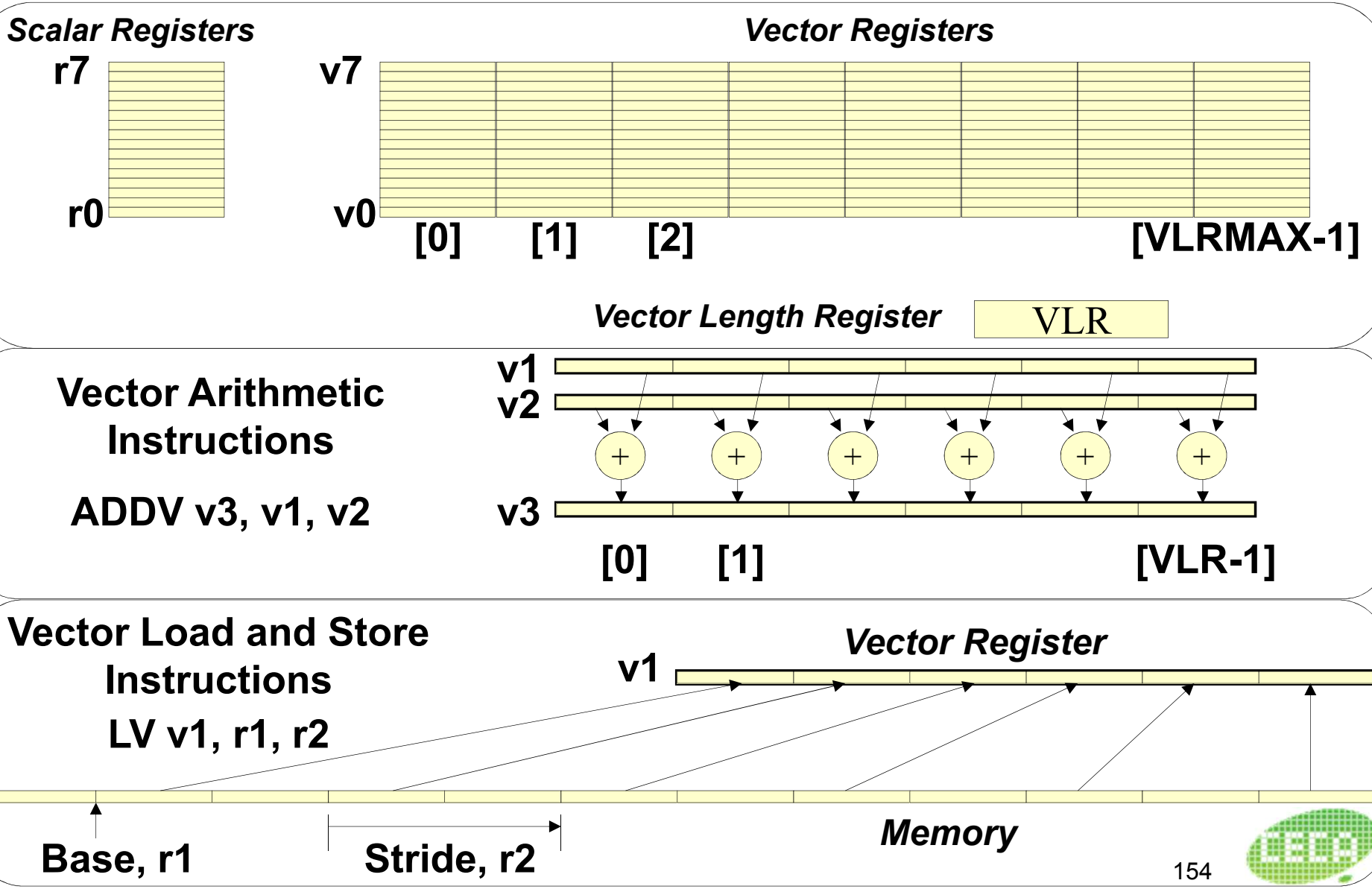
(we ignore vector memory-memory from now on)

向量寄存器体系结构的基本结构



VMIPS

向量编程模型


Scalar Registers
r7
r0
Vector Registers
v7
v0
[0]
[1]
[2]
[VLRMAX-1]
Vector Length Register
VLR
Vector Arithmetic Instructions
ADDV v3, v1, v2
v1
v2
v3
[0]
[1]
[VLR-1]
Vector Load and Store Instructions
LV v1, r1, r2
Vector Register
v1
Base, r1
Stride, r2
Memory

向量代码示例

# C code	# Scalar Code	# Vector Code
for (i=0; i<64; i++)	LI R4, 64	LI VLR, 64
C[i] = A[i] + B[i];	loop:	LV V1, R1
	L.D F0, 0(R1)	LV V2, R2
	L.D F2, 0(R2)	ADDV.D V3, V1, V2
	ADD.D F4, F2, F0	SV V3, R3
	S.D F4, 0(R3)	
	DADDIU R1, 8	
	DADDIU R2, 8	
	DADDIU R3, 8	
	DSUBIU R4, 1	
	BNEZ R4, loop	

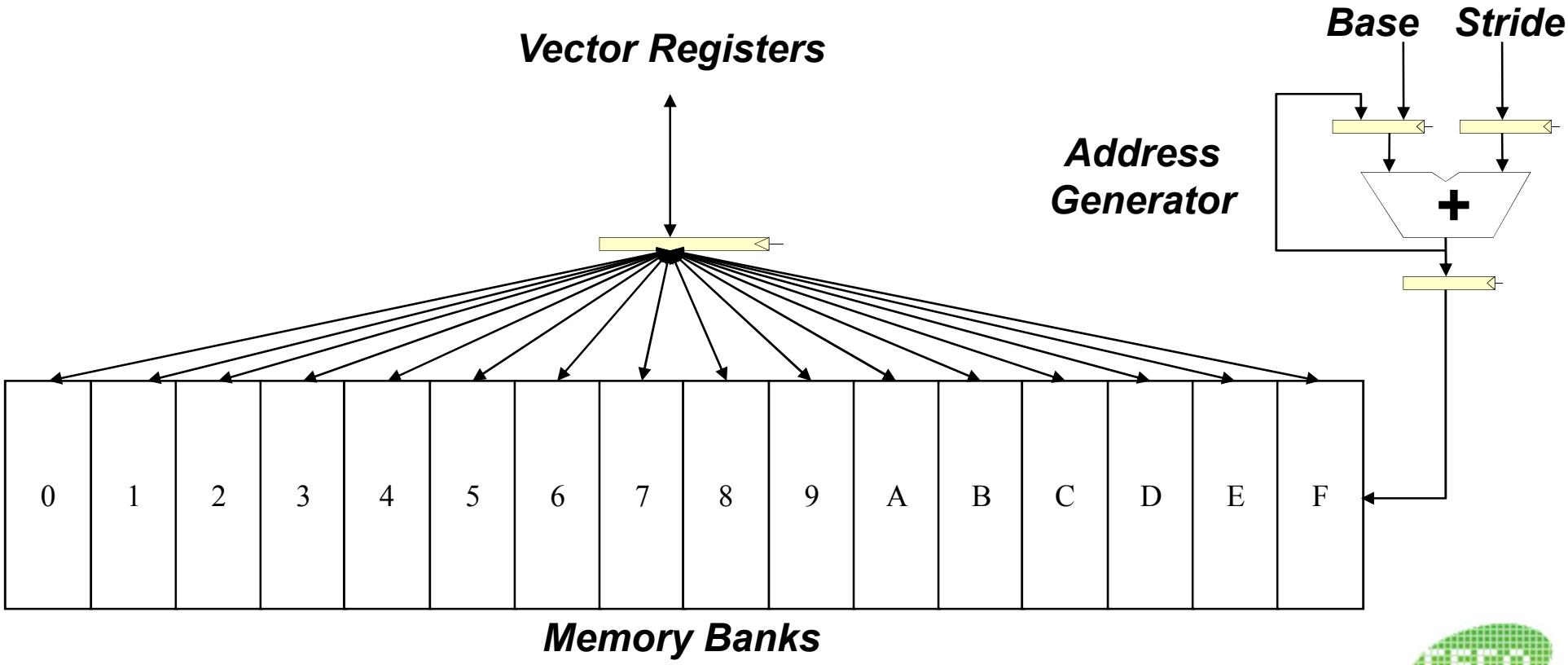
向量指令集的优势

- 紧凑
  - one short instruction encodes N operations
- 直接，告知硬件这N个操作:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in the same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- 可扩展
  - can run same object code on more parallel pipelines or *lanes*

向量内存系统

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- ***Bank busy time*: Cycles between accesses to same bank**



两个现实问题：向量长度和步幅

- 当程序中的向量长度不是正好**64**的时候该怎么办？
- 如果向量中的元素在内存中并不相邻怎么办？

Vector-Length Control

```
      do 10 i = 1,n
10    Y(i) = a * X(i) + Y(i)
```

n may not even be known until run time

向量长度

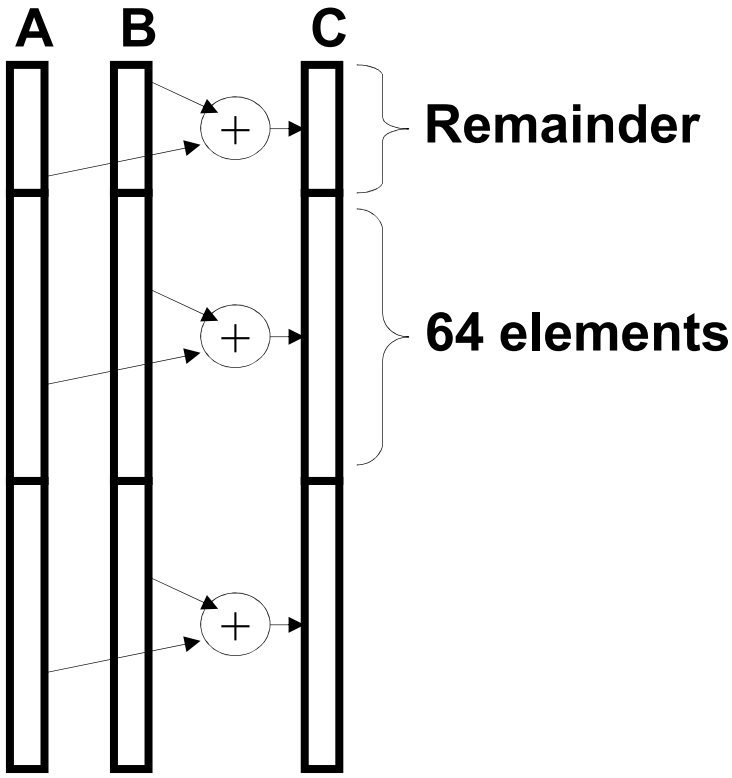
- 解决方案是：创立一个向量长度寄存器 **(VLR),** 来控制向量操作的长度
- **VLR**中的值不能超过向量寄存器的长度 **—** 最大向量长度 **(MVL).**
- 如果向量超过了最大长度，就需要是用向量条形分割的技术

向量条形分割

问题: 向量寄存器具有固定的长度

解决方案: 将循环分割成多个可以放到向量寄存器中的部分 – “条形分割”

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```




```
        ANDI R1, N, #63     ; N mod 64
        MTC1 VLR, R1        ; Do remainder
loop:
        LV V1, RA
        DSLL R2, R1, #3     ; Multiply by 8
        DADDU RA, RA, R2    ; Bump pointer
        LV V2, RB
        DADDU RB, RB, R2
        ADDV.D V3, V1, V2
        SV V3, RC
        DADDU RC, RC, R2
        DSUBU N, N, R1 ; Subtract elements
        LI R1, #64
        MTC1 VLR, R1    ; Reset full length
        BGTZ N, loop    ; Any more to do?
```

向量步幅

```
      do 10 i = 1,100
         do 10 j = 1,100
            A(i,j) = 0.0
            do 10 k = 1,100
10             A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- 第10行中，我们可以将B的每一行与C的每一列的乘积操作进行向量化
- 当一个数组被分配内存的时候，会被线性分配地址空间，要么采用行优先，要么采用列优先的方法。于是意味着要么B每一行的数据，要么C每一列的数据，在内存中是不连续的

向量步幅

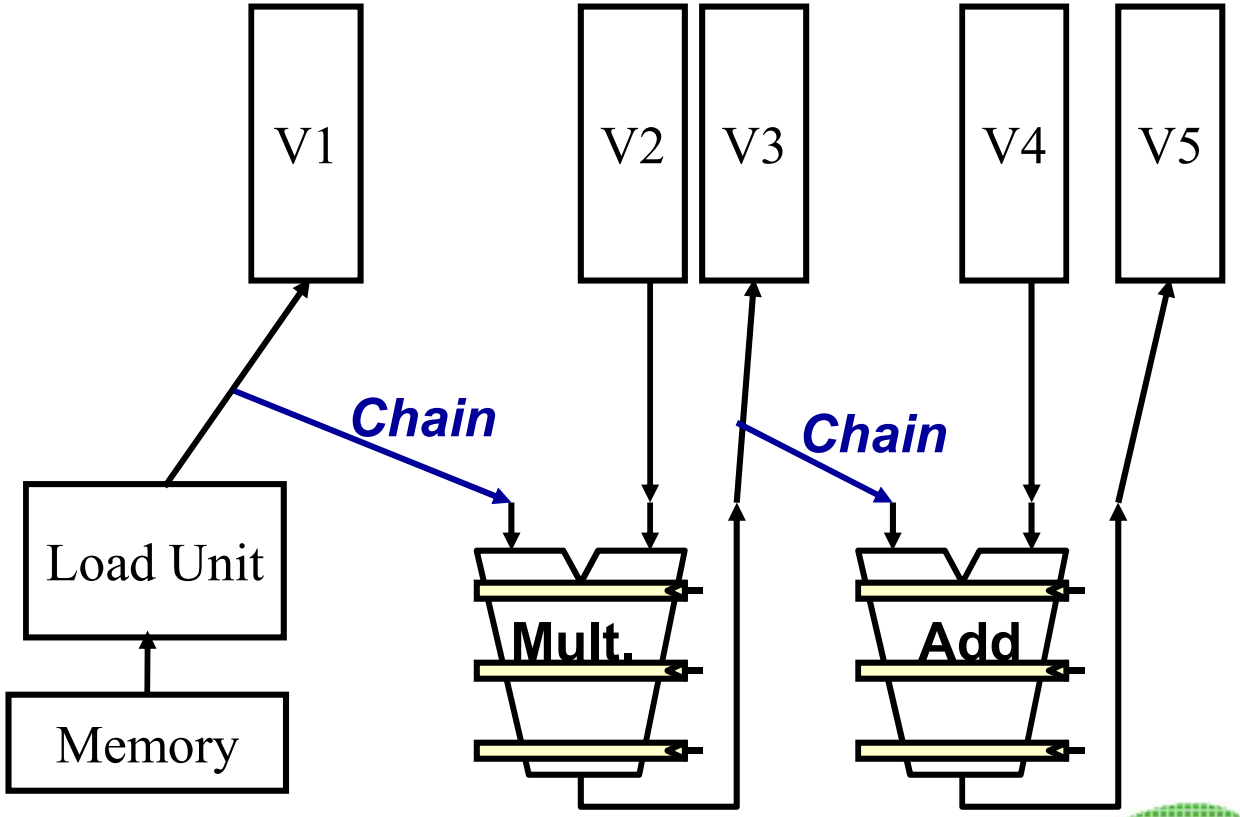
需要被放入一个向量寄存器的数据相邻的距离，叫做步幅

- 向量步幅，与向量起始地址一样，可以被放入通用寄存器中
- 于是**VMIPS**指令**LVWS** (读入带有步幅的向量) 就可以被用来把向量读入到向量寄存器中
- 与其类似，可以用**SVWS** (存储带有步幅的向量) 来存储这种向量

提高向量处理性能（1）向量链接

- 数据前递的思想可以被扩展到向量寄存器中
- 寄存器旁路的向量版
  - introduced with Cray-1

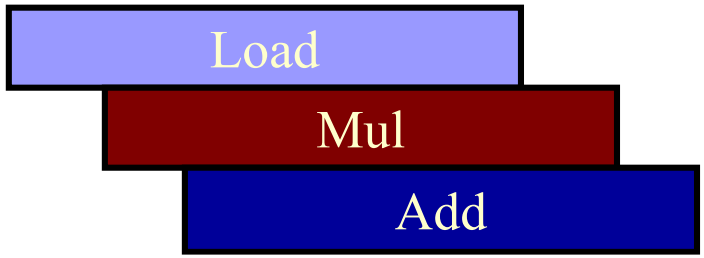
```
LV    v1
MULV  v3,v1,v2
ADDV  v5, v3, v4
```



向量链接的优点

- 如果没有链接，必须等待向量中最后一个单元的结果被写入完成，才能开始下一条有依赖关系的指令

- 有了链接，当第一个单元的结果准备好后，就可以立刻开始下一条有依赖关系的指令



提高向量处理性能（2）向量的条件执行

问题：如何对带有条件执行的循环进行向量化？

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

解决方案：加上向量屏蔽（或标志）寄存器

- vector version of predicate registers, 1 bit per element

…以及可屏蔽的向量指令

- vector operation becomes NOP at elements where mask bit is clear

代码举例：

```
CVM                 ; Turn on all elements
LV VA, RA           ; Load entire A vector
L.D F0,#0           ; Load FP zero into F0
SGTVS.D VA, F0      ; Set bits in mask register where A>0
LV VA, RB           ; Load B vector into A under mask
SV VA, RA           ; Store A back to memory under mask
```


提高向量处理性能（3）稀疏矩阵

$$A_{4\times5}=\begin{bmatrix}0 & 5 & 0 & 0 & 5\\ 1 & 0 & 3 & 0 & 0\\ 0 & -2 & 0 & 0 & 0\\ 6 & 0 & 0 & 0 & 0\end{bmatrix}$$

(a)

	i	j	v
0	0	1	5
1	0	4	5
2	1	0	1
⋮	1	2	3
⋮	2	1	-2
a→t-1	3	0	6
⋮			
MaxSize-1			

向量的分发/收集

对于存在间接引用的循环进行向量化:

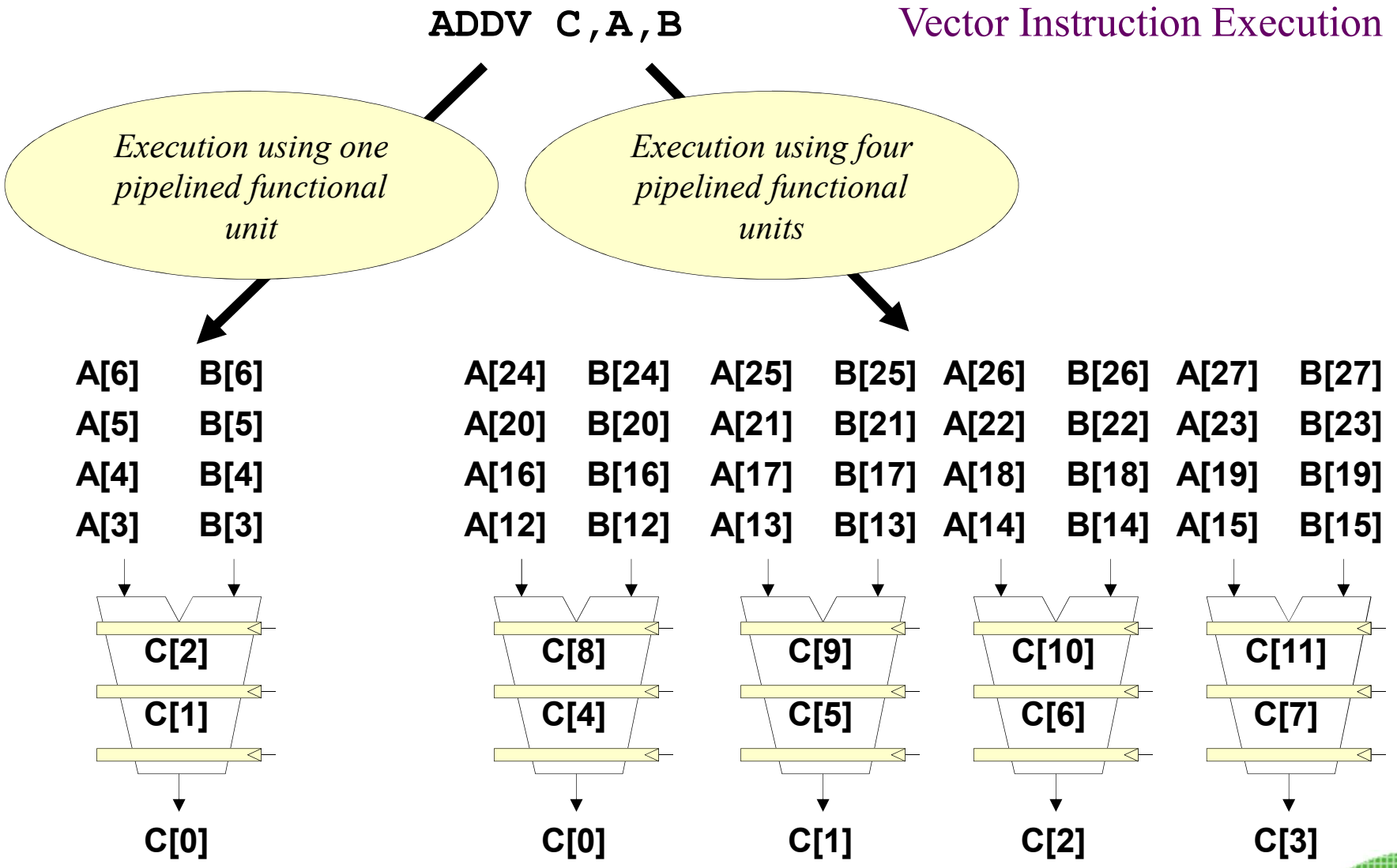
(索引向量D代表着C中的非0元素)

```
for (i=0; i<N; i++)
    A[i] = B[i] + C[D[i]]
```

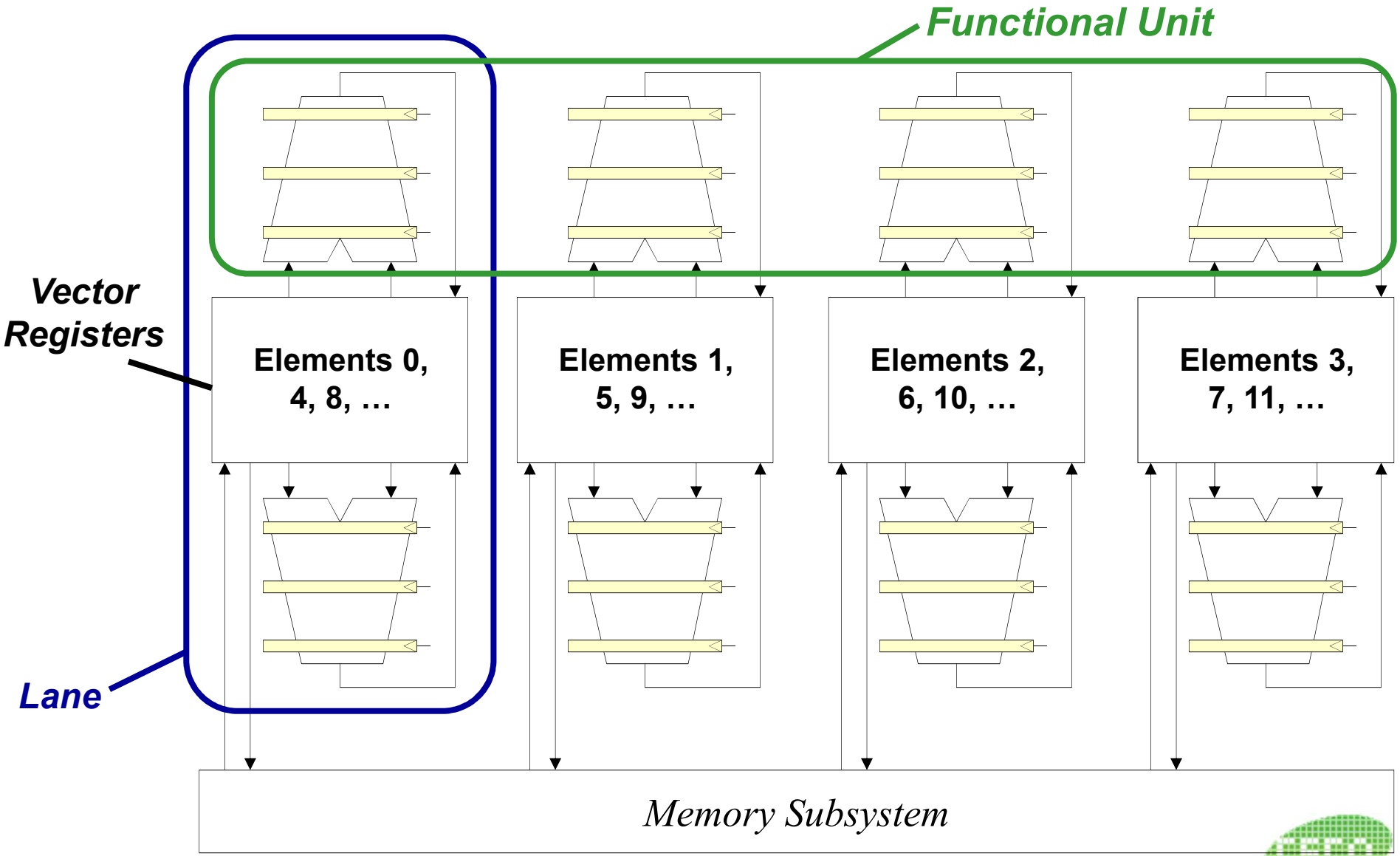
带索引的读取指令 (收集)

```
LV        VD, RD          ; Load indices in D vector
LVI       VC,(RC, VD)     ; Load indirect from RC base
LV        VB, RB          ; Load B vector
ADDV.D    VA, VB, VC      ; Do add
SV        VA, RA          ; Store result
```


提高向量处理性能（4）多路执行



向量单元的结构


Functional Unit
Vector Registers
Elements 0, 4, 8, …
Elements 1, 5, 9, …
Elements 2, 6, 10, …
Elements 3, 7, 11, …
Lane
Memory Subsystem

并行计算机结构

Amdahl定律举例

$$\text{Speedup w/ E} = \frac{1}{(1-F) + F/S}$$

- 考虑一项优化，能够提高速度到原先的**20**倍，但是只在**25%**的情况下可以被使用

  Speedup w/ E = 1/(.75 + .25/20) = 1.31

- 如果只有**15%**的情况下能够被用到会怎样？

  Speedup w/ E = 1/(.85 + .15/20) = 1.17

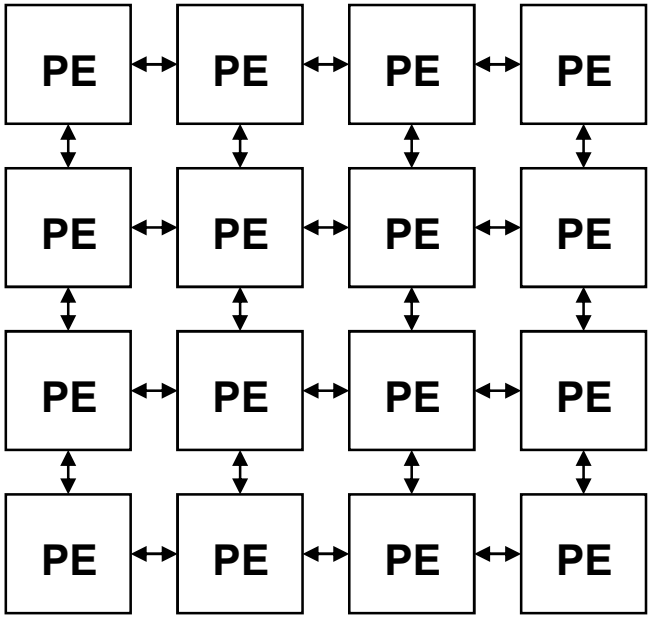
- **Amdahl**定律告诉我们，如果想用**100**个处理器来实现线性的加速比，原有的计算就不能有任何的串行部分！
- 如果用**100**个处理器达到**99**的加速比，原有的计算的串行部分必须是**0.01%**或者更少

Flynn分类法

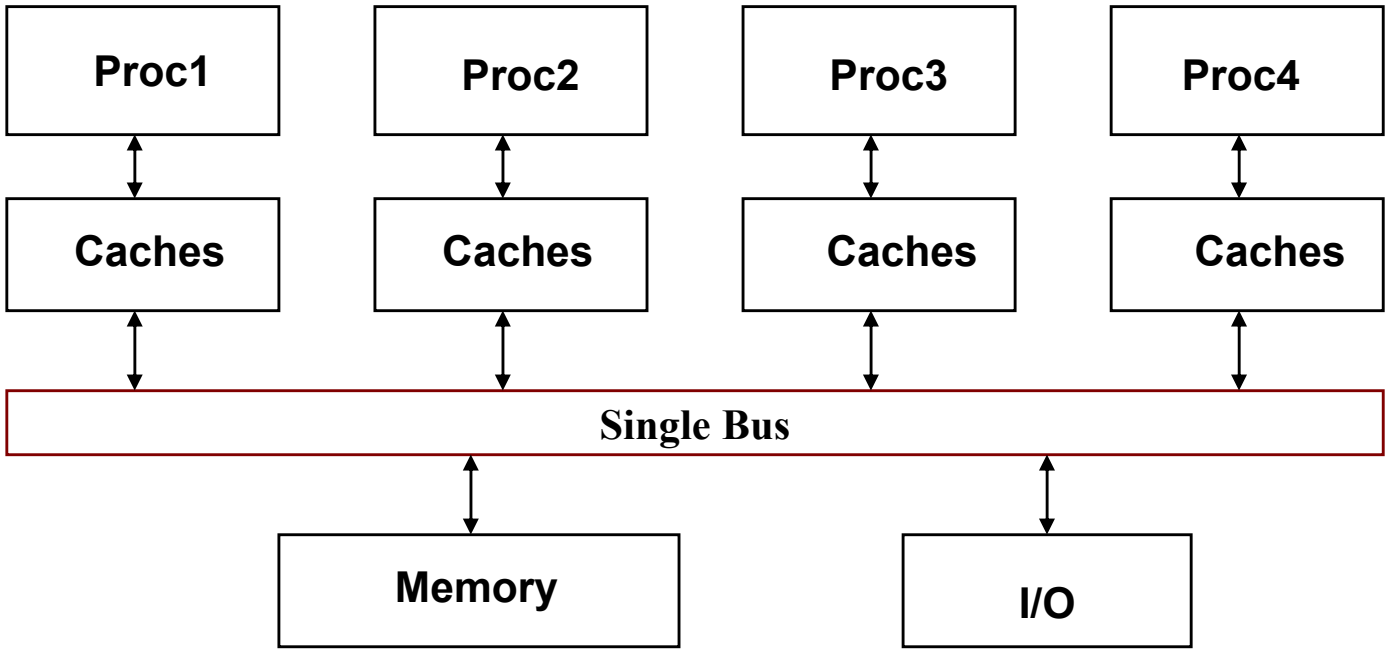
- SISD – 单指令单数据流
  - aka uniprocessor - what we have been talking about all semester
- SIMD – 单指令多数据流
  - single control unit broadcasting operations to multiple datapaths
- MISD – 多指令单数据流
  - no such machine (although some people put vector machines in this category)
- MIMD – 多指令多数据流
  - aka multiprocessors (SMPs, MPPs, clusters, NOWs)

SIMD处理器

- 单一的控制单元
- 多个数据通路 (处理单元 – **PEs**) 并行运行
  - **Q1 – PEs are interconnected (usually via a mesh or torus) and exchange/share data as directed by the control unit**
  - **Q2 – Each PE performs the same operation on its own local data**



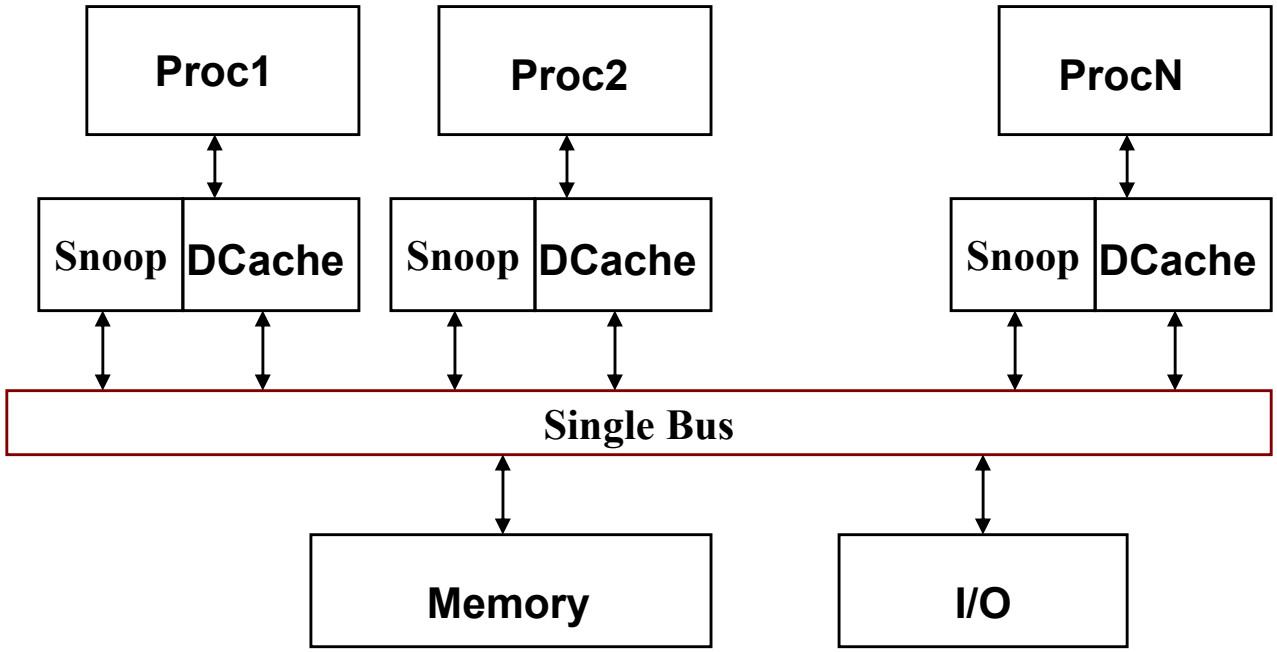
基于单一总线的对称多处理器计算机（SMP）



- 高速缓存被用来减少延迟并减少总线上的信息量
  - Write-back caches used to keep bus traffic at a minimum
- 必须提供硬件，保证告诉缓存和内存是一致的 (高速缓存一致性)
- 必须提供硬件，来支持进程的同步

缓存一致性

- 缓存一致性协议
  - **Bus snooping – cache controllers monitor shared bus traffic with duplicate address tag hardware (so they don't interfere with processor's access to the cache)**



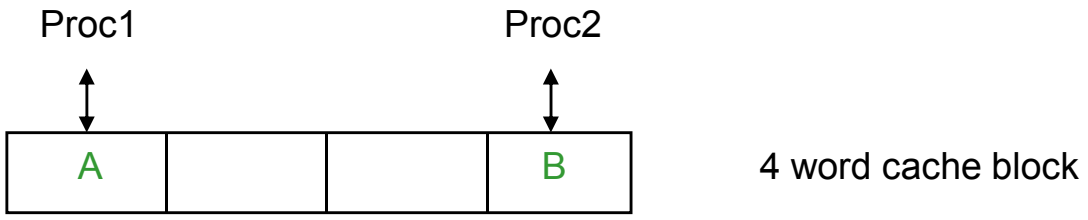
处理写操作

进行数据写操作的时候，要保证所有其它共享此数据的处理器能够被通知，可以有两种方法来处理:

1. **Write-update (write-broadcast) – writing processor broadcasts new data over the bus, all copies are updated**
   - **All writes go to the bus → higher bus traffic**
   - **Since new values appear in caches sooner, can reduce latency**
2. **Write-invalidate – writing processor issues invalidation signal on bus, cache snoops check to see if they have a copy of the data, if so they invalidate their cache block containing the word (this allows multiple readers but only one writer)**
   - **Uses the bus only on the first write → lower bus traffic, so better use of bus bandwidth**

缓存块大小的影响

- 在一个多字的缓存块中，写入一个字意味着
  - **either the full block is invalidated (write-invalidate)**
  - **or the full block is exchanged between processors (write-update)**
    - **Alternatively, could broadcast *only* the written word**
- 多字的缓存块也会导致伪共享: 当两个处理器写一个缓存块的不同位置时
  - **With write-invalidate false sharing increases cache miss rates**



- 编译器可以通过将高相关数据放入同一缓存块来缓解伪共享情况

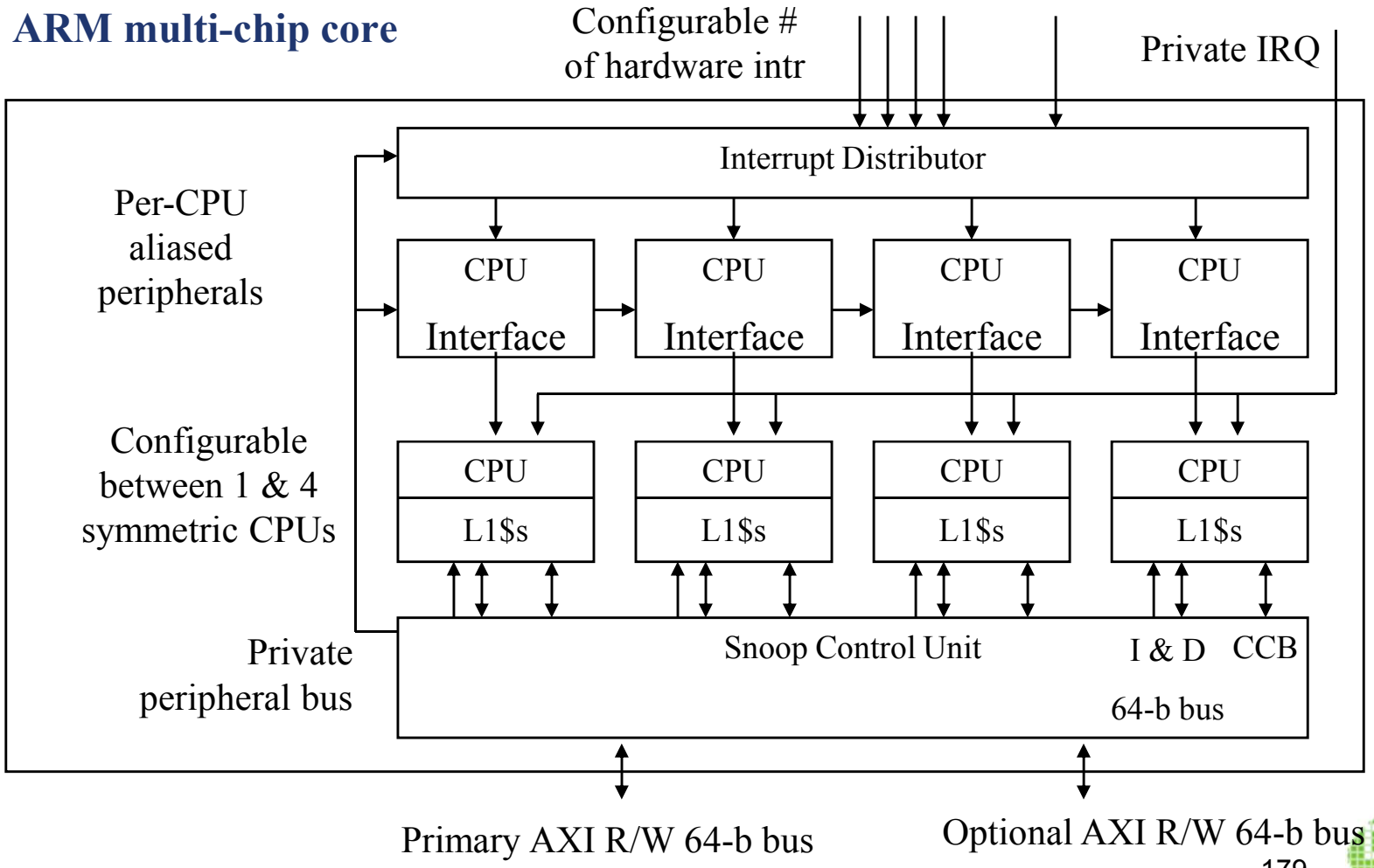
进程同步

- 需要能够协调在同一任务中工作的不同进程
- 锁变量（信号量）被用来协调或同步进程

- 需要体系结构支持的仲裁机制来决定哪个处理器能够得到锁变量的访问权
  - **Single bus provides arbitration mechanism, since the bus is the only path to memory – the processor that gets the bus wins**
- 需要一个体系结构支持的操作来对变量加锁
  - **Locking can be done via an atomic swap operation (processor can both read a location *and* set it to the locked state – test-and-set – in the same bus operation)**

CMP: 片内多处理器

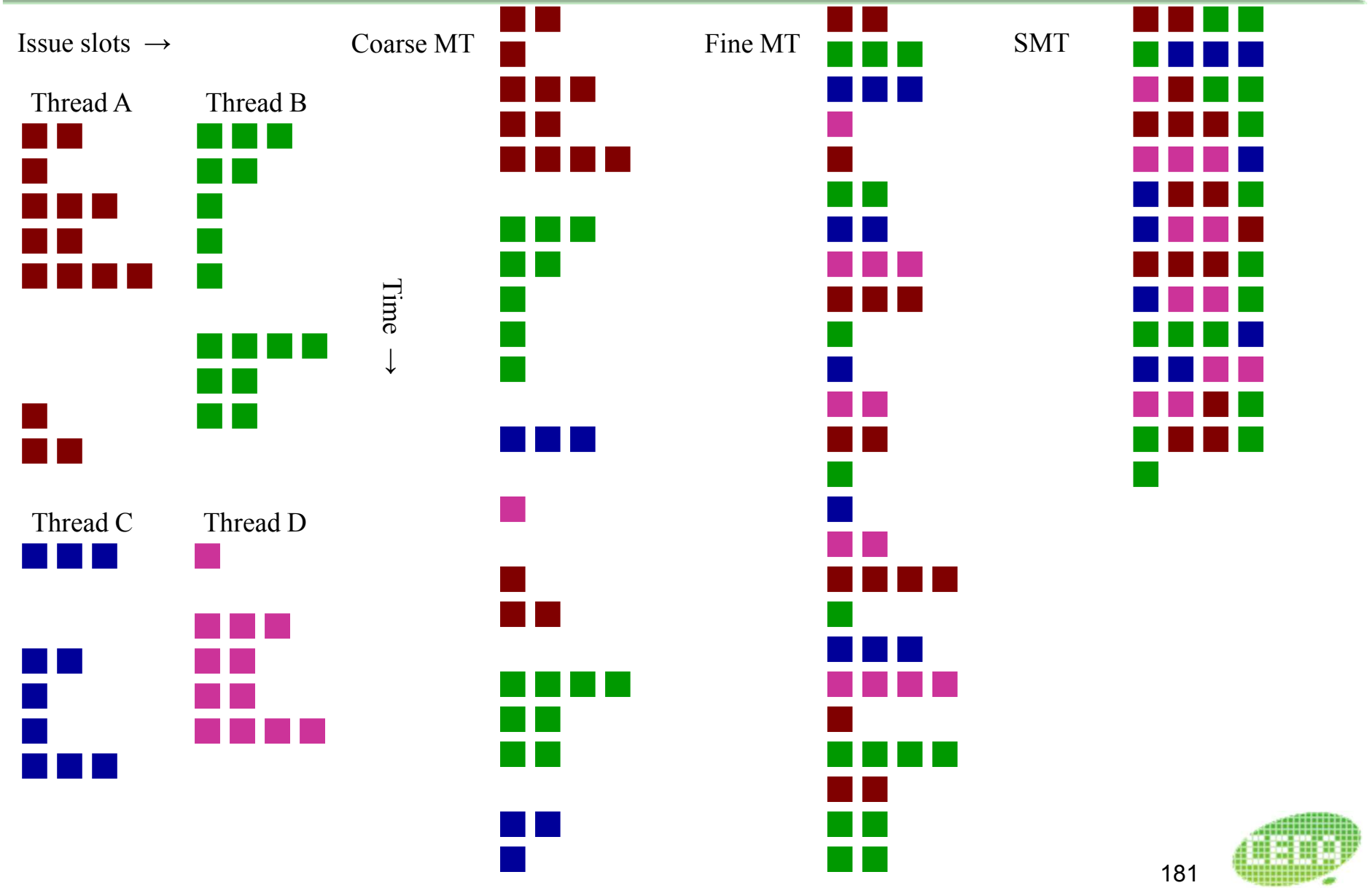
- 通过在一个芯片内放入多个处理器、存储以及互联，芯片之间的通信延迟可以被大幅度减少
  - **ARM multi-chip core**



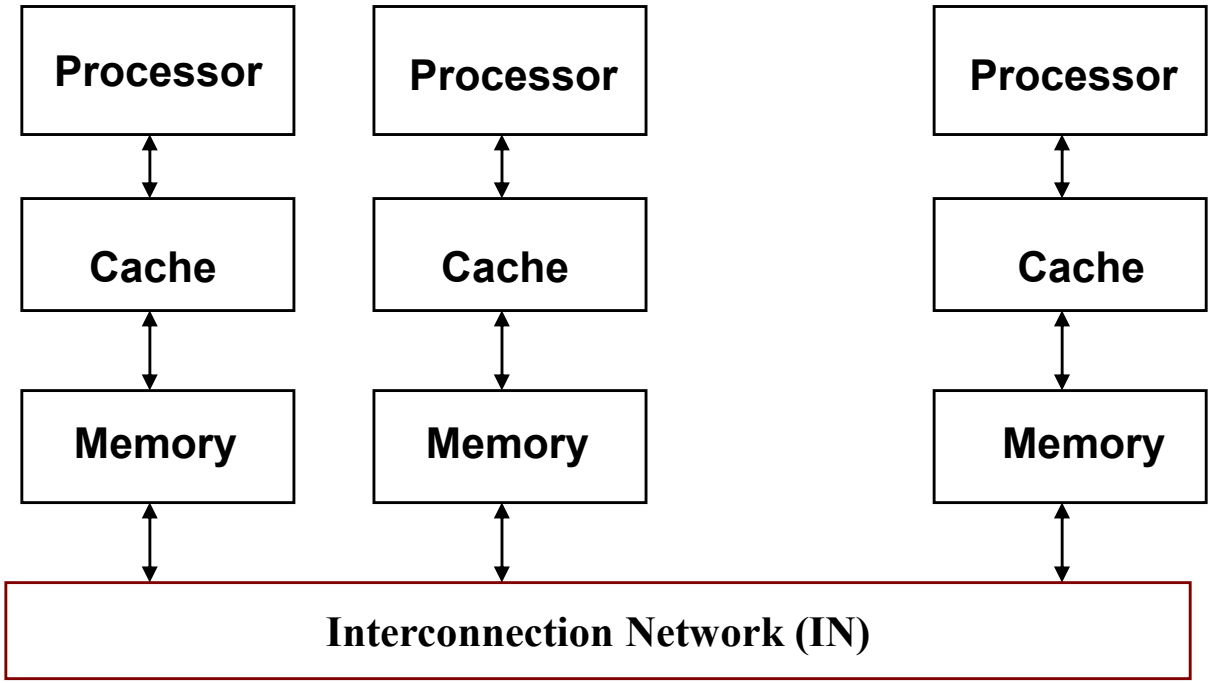
同步多线程 (SMT)

- 是一个多线程的变种，使用多发射、动态调度的处理器（超标量）中的资源来同时开发程序的指令级并行以及线程级并行
  - **Most SS processors have more machine level parallelism than most programs can effectively use (i.e., than have ILP)**
  - **With register renaming and dynamic scheduling, multiple instructions from independent threads can be issued without regard to dependencies among them**
    - **Need separate rename tables (ROBs) for each thread**
    - **Need the capability to commit from multiple threads (i.e., from multiple ROBs) in one cycle**
- **Intel's Pentium 4 SMT也被称作超线程hyperthreading**
  - **Supports just two threads (doubles the architecture state)**

在不同种类的4路超标量处理器中的多线程



网络相连的多计算机



- 或者是在单一地址空间 **(NUMA and ccNUMA)** 上用**loads**和**stores**进行隐式处理器通信，或者是在多个私有地址空间中采用消息传递机制来收发信息
  - **Interconnection network supports interprocessor communication**

网络相连的多计算机之间的通信

- 通过**loads**和**stores**进行隐式通信
  - **hardware designers have to provide coherent caches and process synchronization primitive**
  - **lower communication overhead**
  - **harder to overlap computation with communication**
  - **more efficient to use an address to remote data when *demanded* rather than to send for it in case it *might* be used (such a machine has distributed shared memory (DSM))**
- 通过**sends**和**receives**进行显式通信
  - **simplest solution for hardware designers**
  - **higher communication overhead**
  - **easier to overlap computation with communication**
  - **easier for the programmer to optimize communication**

了解互联方式

- 总线互联
- 环形互联
- 全连接互联
- 交叉开关互联
- 超立方体互联
- **2D and 3D Mesh/Torus互联**
- 胖树

工作站网络（NOWs）机群

- 采用商业计算机组成的机群系统，多个私有地址空间
- 用计算机的I/O总线进行互联
  - lower bandwidth that multiprocessor that use the memory bus
  - lower speed network links
  - more conflicts with I/O traffic
- N个计算机组成的机群系统，允许N个操作系统的拷贝，对应用的可用内存有限制
- 提高了系统的可用性和可扩展性
  - easier to replace a machine without bringing down the whole system
  - allows rapid, incremental expandability
- 从造价的角度来讲，是较为经济的，扩展起来也不贵