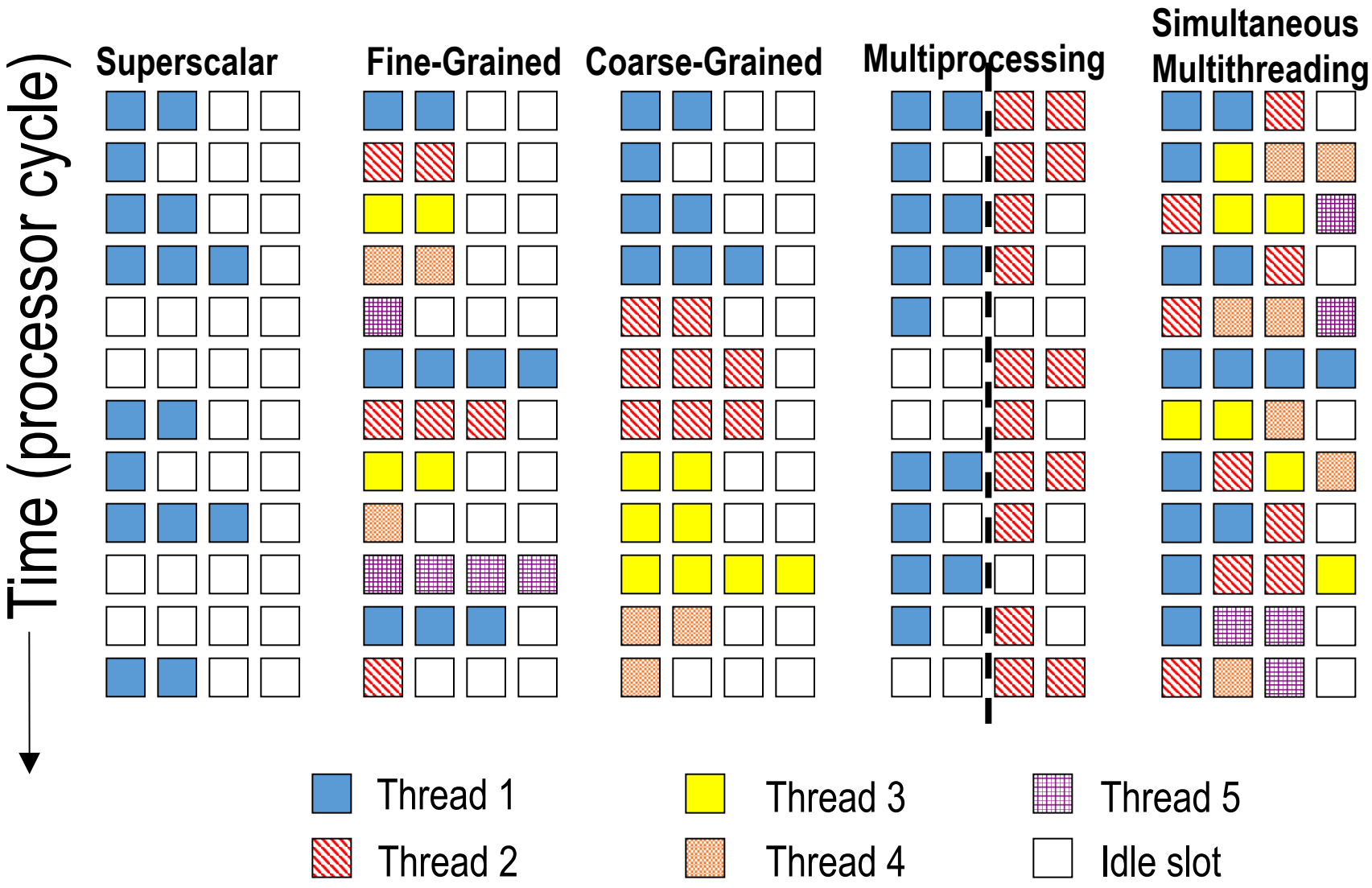
计算机体系结构

Topic: Data Level Parallelism

https://blog.csdn.net/bpssy/article/details/16965377

Review: Multithreaded Categories



Data-Level Parallelism in Vector, SIMD, and GPU Architectures

- **数据级并行的研究动机**
  - 传统指令级并行技术的问题
  - SIMD结构的优势
  - 数据级并行的种类
- **向量体系结构**
  - 向量处理模型
  - 起源-超级计算机
  - 基本特性及结构
  - 性能评估及优化
- **面向多媒体应用的SIMD指令集扩展**
- **GPU**
  - GPU简介
  - GPU的编程模型
  - GPU的存储系统

动机：传统指令级并行技术的问题

- **提高性能的传统方法（挖掘ILP）的主要缺陷:**
  - 程序内在的并行性
  - 提高流水线的时钟频率: 提高时钟频率，有时导致CPI随着增加 (branches, other hazards)
  - 指令预取和译码: 有时在每个时钟周期很难预取和译码多条指令
  - 提高Cache命中率 : 在有些计算量较大的应用中（科学计算）需要大量的数据，其局部性较差，有些程序处理的是连续的媒体流(multimedia),其局部性也较差。

动机：DLP的兴起

- 应用需求和技术发展推动着体系结构的发展
- 图形、机器视觉、语音识别、机器学习等新的应用均需要大量的数值计算，其算法通常具有数据并行特征
- SIMD-based 结构 (vector-SIMD, subword-SIMD, SIMT/GPUs) 是执行这些算法的最有效途径

系统结构的Flynn分类 (1966)

- **SISD: Single instruction stream, single data stream**
  - 单处理器模式
- **SIMD: Single instruction stream, multiple data streams**
  - 相同的指令作用在不同的数据
  - 可用来挖掘数据级并行(Data Level Parallelism)
  - 如：Vector processors, SIMD instructions, and Graphics processing units
- **MISD: Multiple instruction streams, single data stream**
  - No commercial implementation
- **MIMD: Multiple instruction streams, multiple data streams**
  - 通用性最强的一种结构，可用来挖掘线程级并行、数据级并行.....
  - 组织方式可以是松耦合方式也可以是紧耦合方式

动机：SIMD结构的优势

- **SIMD 结构可有效地挖掘数据级并行:**
  - 基于矩阵运算的科学计算
  - 图像和声音处理
- **SIMD比MIMD更节能**
  - 针对每组数据操作仅需要取指一次
  - SIMD对PMD(personal mobile devices)更具吸引力
- **SIMD 允许程序员继续以串行模式思维**

SIMD 结构的种类

- **向量体系结构**
- **多媒体SIMD指令集 扩展**
- **Graphics Processor Units (GPUs)**

- **For x86 processors:**
  - 每年增加2cores/chip
  - SIMD 宽度每4年翻一番
  - SIMD潜在加速比是MIMD的2倍

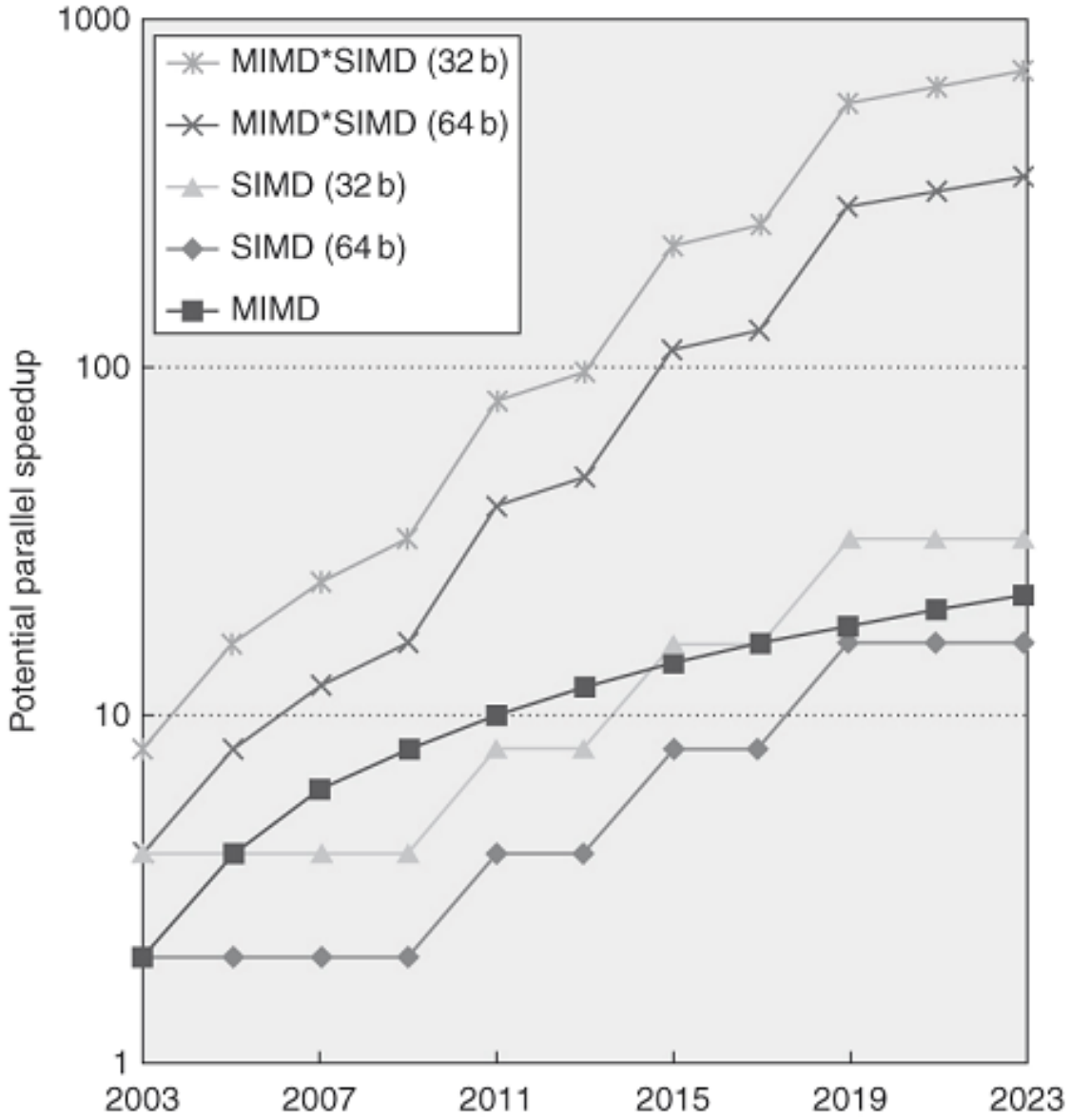
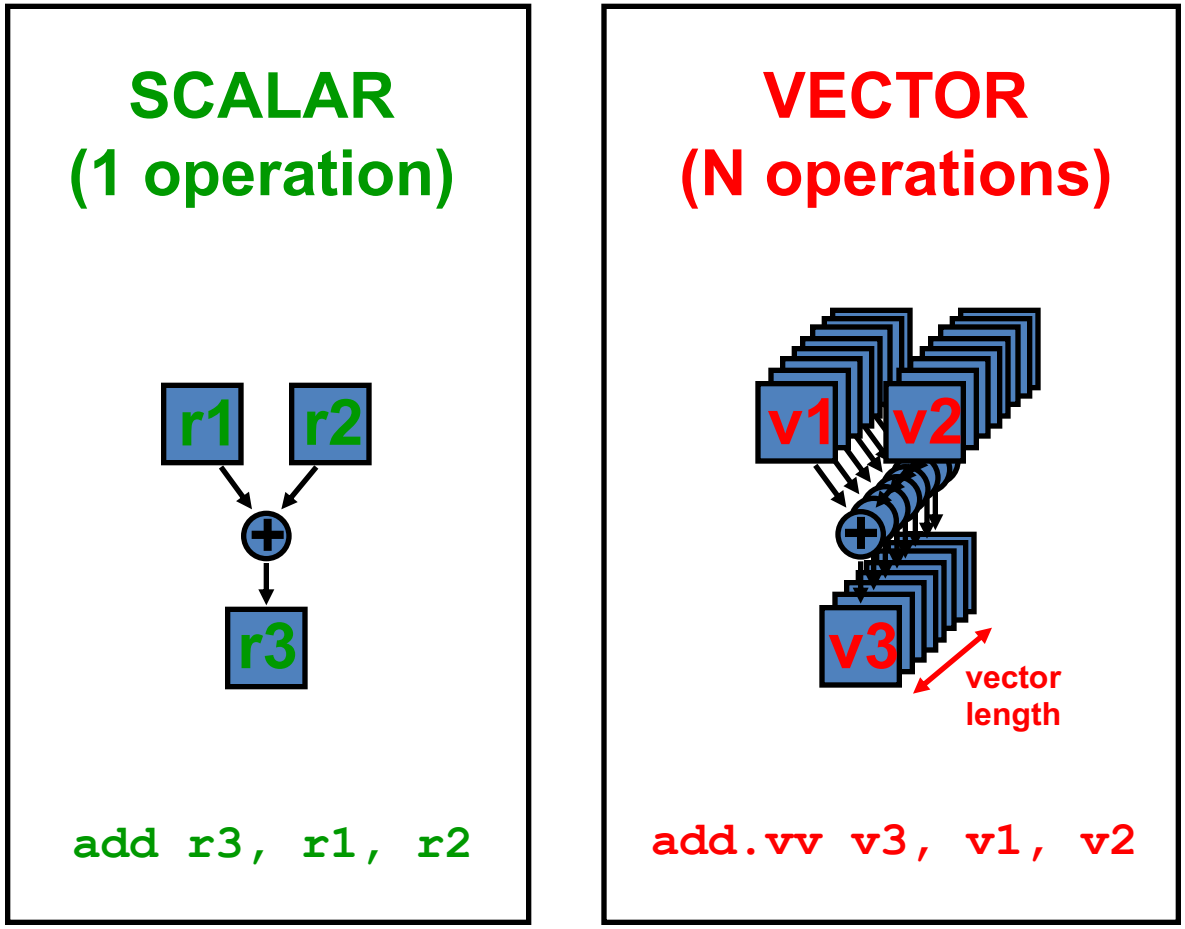


Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

向量处理模型

- **向量处理机具有更高层次的操作，一条向量指令可以处理N个或N对操作数（处理对象是向量）**



起源：Supercomputers

- **Supercomputer的定义:**
  - 对于给定任务而言世界上最快的机器
  - 任何造价超过3千万美元的机器
  - 计算能力达到每秒万亿次的机器
- **由Seymour Cray设计的机器**
- **CDC6600 (Cray, 1964) 被认为是第一台超级计算机**

CDC 6600 *Seymour Cray, 1963*

- **A fast pipelined machine with 60-bit words**
  - 128 Kword main memory capacity, 32 banks
- **Ten functional units (parallel, unpipelined)**
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, …
- **Hardwired control (no microcoding)**
- ***Scoreboard* for dynamic scheduling of instructions**
- **Ten Peripheral Processors for Input/Output**
  - a fast multi-threaded 12-bit integer ALU
- **Very fast clock, 10 MHz (FP add in 4 clocks)**
- **>400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling**
- **Fastest machine in world for 5 years (until 7600)**
  - over 100 sold ($7-10M each)

IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

> "Last week, Control Data … announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers… Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."

To which Cray replied: "It seems like Mr. Watson has answered his own question."

Supercomputer Applications

- **典型应用领域**
  - 军事研究领域（核武器研制、密码学）
  - 科学研究
  - 天气预报
  - 石油勘探
  - 工业设计 (car crash simulation)
  - 生物信息学
  - 密码学
- **均涉及大量的数据集处理**
- **70-80年代Supercomputer = Vector Machine**

向量处理机的基本特性

- **基本思想：两个向量的对应分量进行运算，产生一个结果向量。**
- **简单的一条向量指令包含了多个操作=> fewer instruction fetches**
- **每一结果独立于前面的结果**
  - 长流水线，编译器保证操作间没有相关性
  - 硬件仅需检测两条向量指令间的相关性
  - 较高的时钟频率
- **向量指令以已知的模式访问存储器**
  - 可有效发挥多体交叉存储器的优势
  - 可通过重叠减少存储器操作的延时 64 elements
  - 不需要数据Cache! (仅使用指令cache)
- **在流水线控制中减少了控制相关**

向量处理机的基本结构

- ***memory-memory vector processors:*** **所有的向量操作是存储器到存储器**
- ***vector-register processors:*** **除了load 和store操作外，所有的操作是向量寄存器与向量寄存器间的操作**
  - 向量机的Load/Store结构
  - 1980年以后的所有的向量处理机都是这种结构: Cray, Convex, Fujitsu, Hitachi, NEC
  - 我们也主要针对这种结构

Vector Memory-Memory versus Vector Register Machines

- **存储器-存储器型向量机所有指令操作的操作数来源于存储器**
- **第一台向量机 CDC Star-100 ('73) and TI ASC ('71), 是存储器-存储器型机器**
- **Cray-1 (' 76) 是第一台寄存器型向量机**

Example Source Code

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

Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

Vector Memory-Memory vs. Vector Register Machines

- **存储器-存储器型向量机 (VMMA) 需要更高的存储器带宽**
  - All operands must be read in and out of memory
- **VMMA结构使得多个向量操作重叠执行较困难**
  - Must check dependencies on memory addresses
- **VMMA启动时间更长**
  - CDC Star-100 在向量元素小于100时，标量代码的性能高于向量化代码
- **CDC Cray-1后续的机器 (Cyber-205, ETA-10) 都是寄存器型向量机**

Vector Supercomputers

- **Cray-1的变体（1976）：**
- **Scalar Unit：Load/Store Architecture**
- **Vector Extension**
  - Vector Registers
  - Vector Instructions
- **Implementation**
  - 硬布线逻辑控制
  - 高效流水化的功能部件
  - 多体交叉存储系统
  - 无Data Cache
  - 不支持 Virtual Memory

Vector Instruction Set Advantages

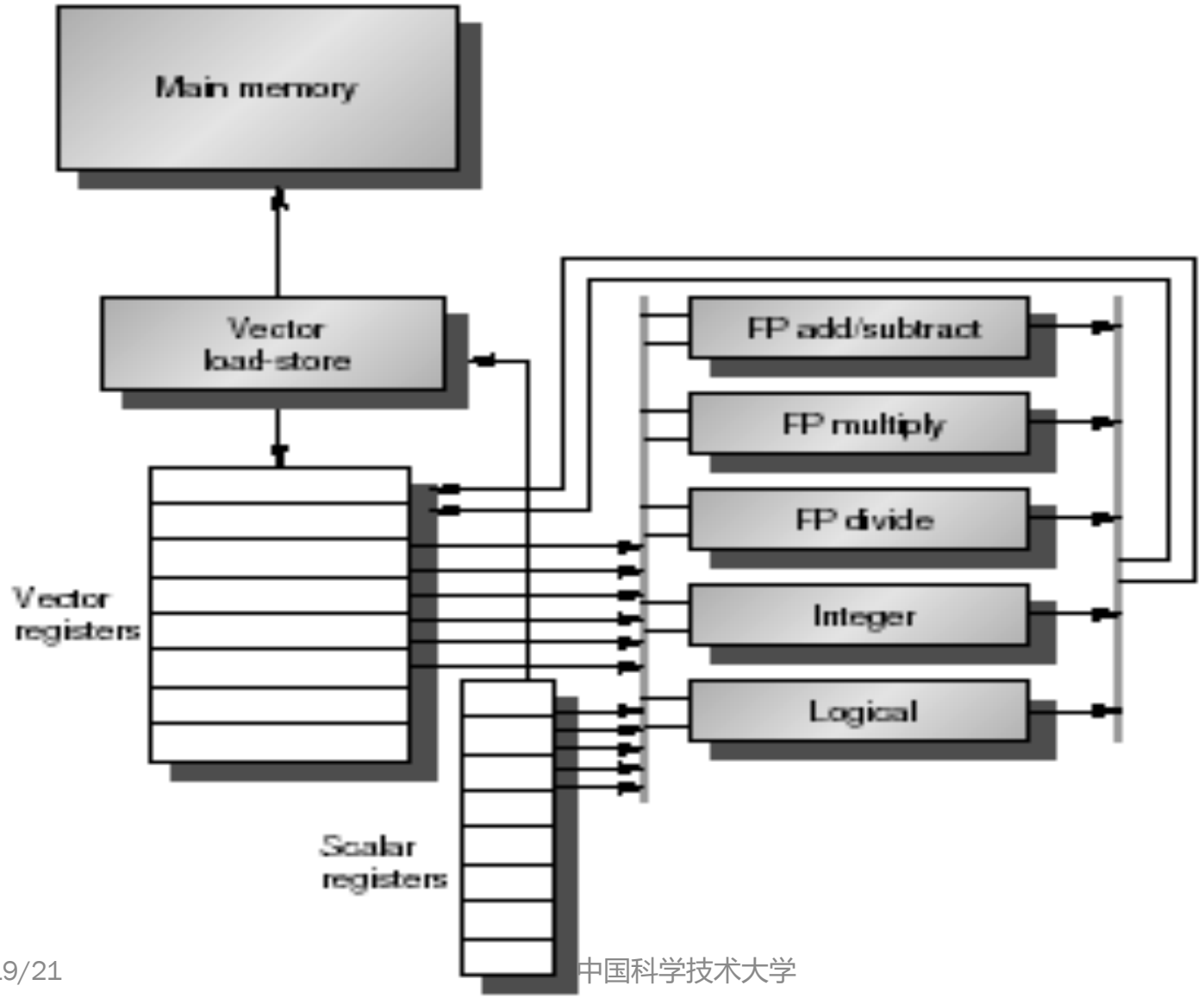
- **格式紧凑**
  - 一条指令包含N个操作
- **表达能力强，一条指令能告诉硬件:**
  - N个操作之间无相关性访
  - 使用同样的功能部件
  - 访问不相交的寄存器
  - 与前面的操作以相同模式访问寄存器
  - 问存储器中的连续块 (unit-stride load/store)
  - 以已知的模式访问存储器 (strided load/store)
- **可扩展性好**
  - 可以在多个并行的流水线上运行同样的代码 (lanes)

Vector Instructions

	Instr.	Operands	Operation Comment
ADDV	V1,V2,V3	V1=V2+V3	vector + vector
ADDSV	V1,F0,V2	V1=F0+V2	scalar + vector
MULTV	V1,V2,V3	V1=V2xV3	vector x vector
MULSV	V1,F0,V2	V1=F0xV2	scalar x vector
LV	V1,R1	V1=M[R1..R1+63]	load, stride=1
LVWS	V1,R1,R2	V1=M[R1..R1+63*R2]	load, stride=R2
LVI	V1,R1,V2	V1=M[R1+V2i,i=0..63]	indir.("gather")
CeqV	VM,V1,V2	VMASKi = (V1i=V2i)?	comp. setmask
MOV	VLR,R1	Vec. Len. Reg. = R1	set vector length
MOV	VM,R1	Vec. Mask = R1	set vector mask

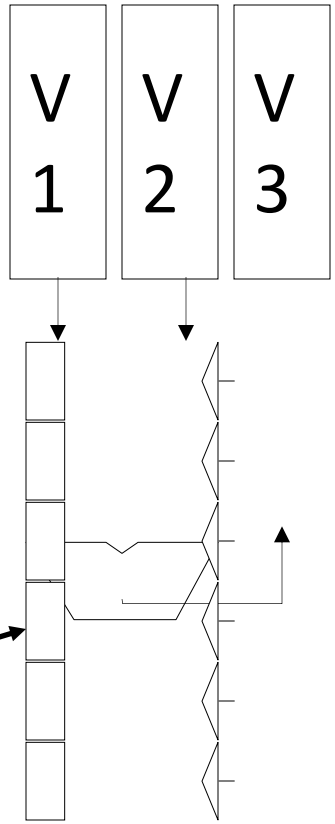
向量处理机的基本组成单元

- ***Vector Register*: 固定长度的一块区域，存放单个向量**
  - 至少2个读端口和一个写端口（一般最少16个读端口，8个写端口）
  - 典型的有8-32 向量寄存器，每个寄存器存放64到128个64位元素
- ***Vector Functional Units (FUs)*: 全流水化的，每一个clock启动一个新的操作**
  - 一般4到8个FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; 可能有些重复设置的部件
- ***Vector Load-Store Units (LSUs)*: 全流水化地load 或store一个向量，可能会配置多个LSU部件**
- ***Scalar registers*: 存放单个元素用于标量处理或存储地址**
- **用交叉开关连接(Cross-bar) FUs , LSUs, registers**


Main memory
Vector
load-store
FP add/subtract
FP multiply
FP divide
Integer
Logical
Vector
registers
Scalar
registers

Vector Arithmetic Execution

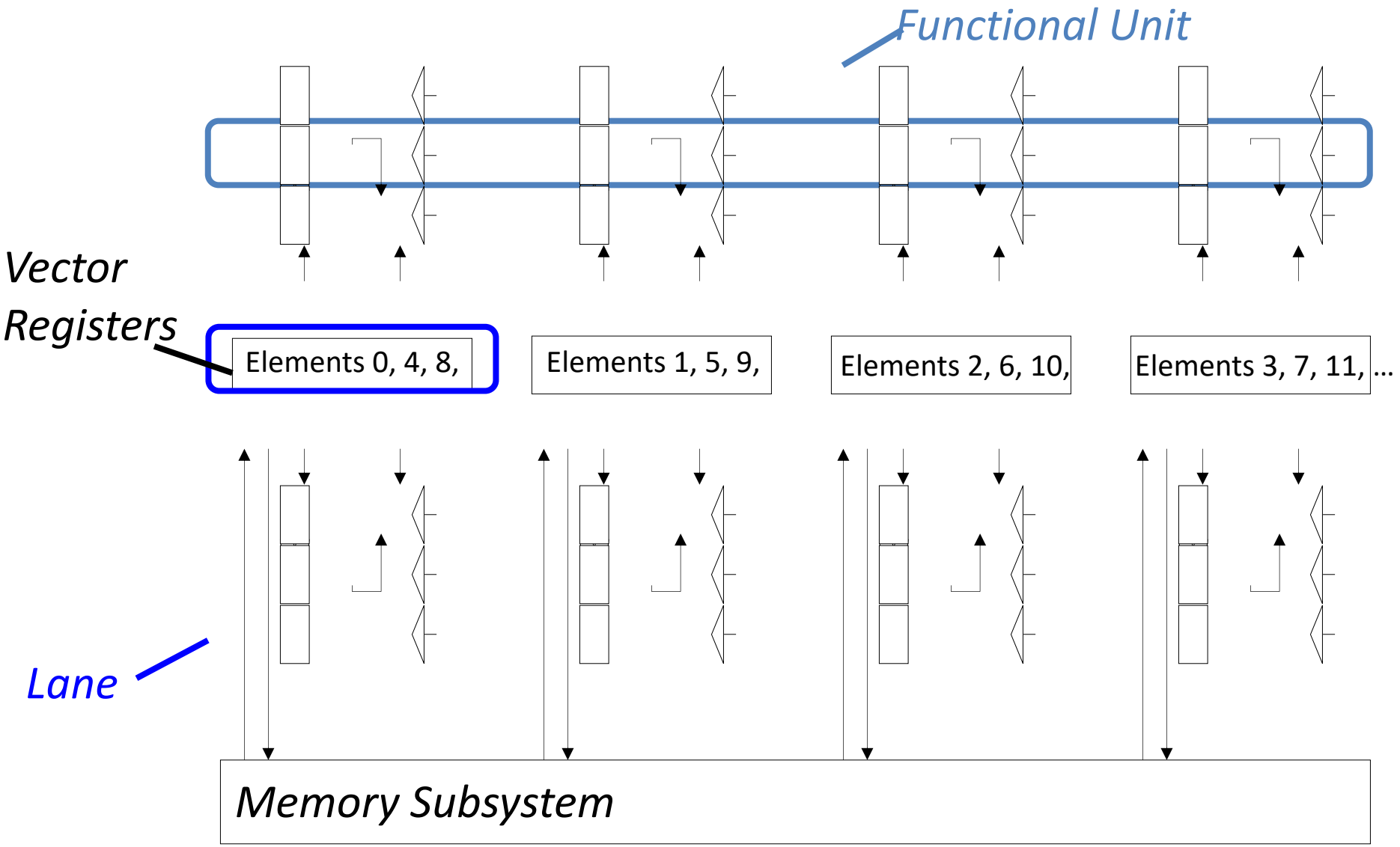
- **使用较深的流水线(=> fast clock) 执行向量元素的操作**
- **由于向量元素相互独立，简化了深度流水线的控制 (=> no hazards!)**



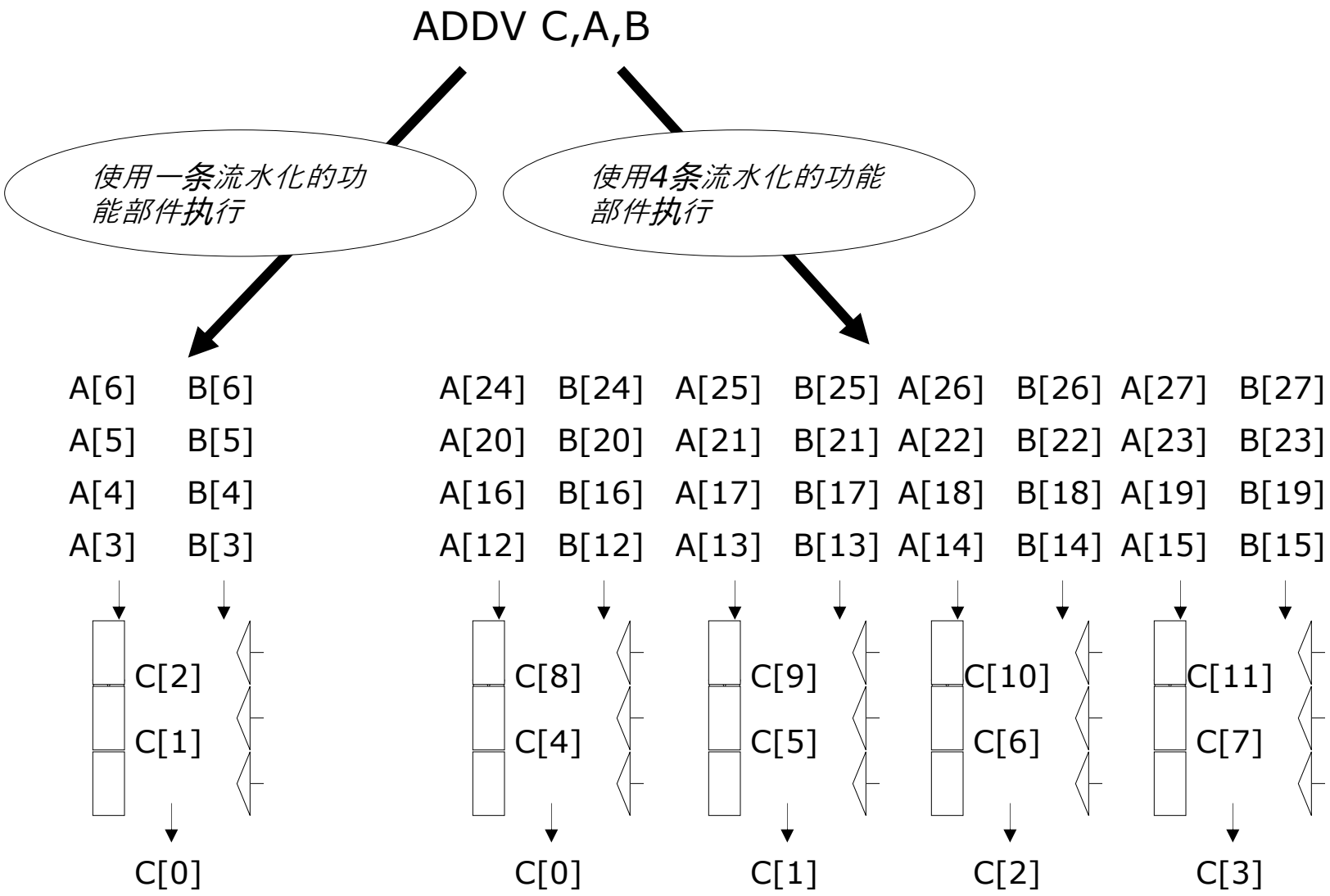
Six stage multiply pipeline

V3 <- v1 * v2

Vector Unit Structure



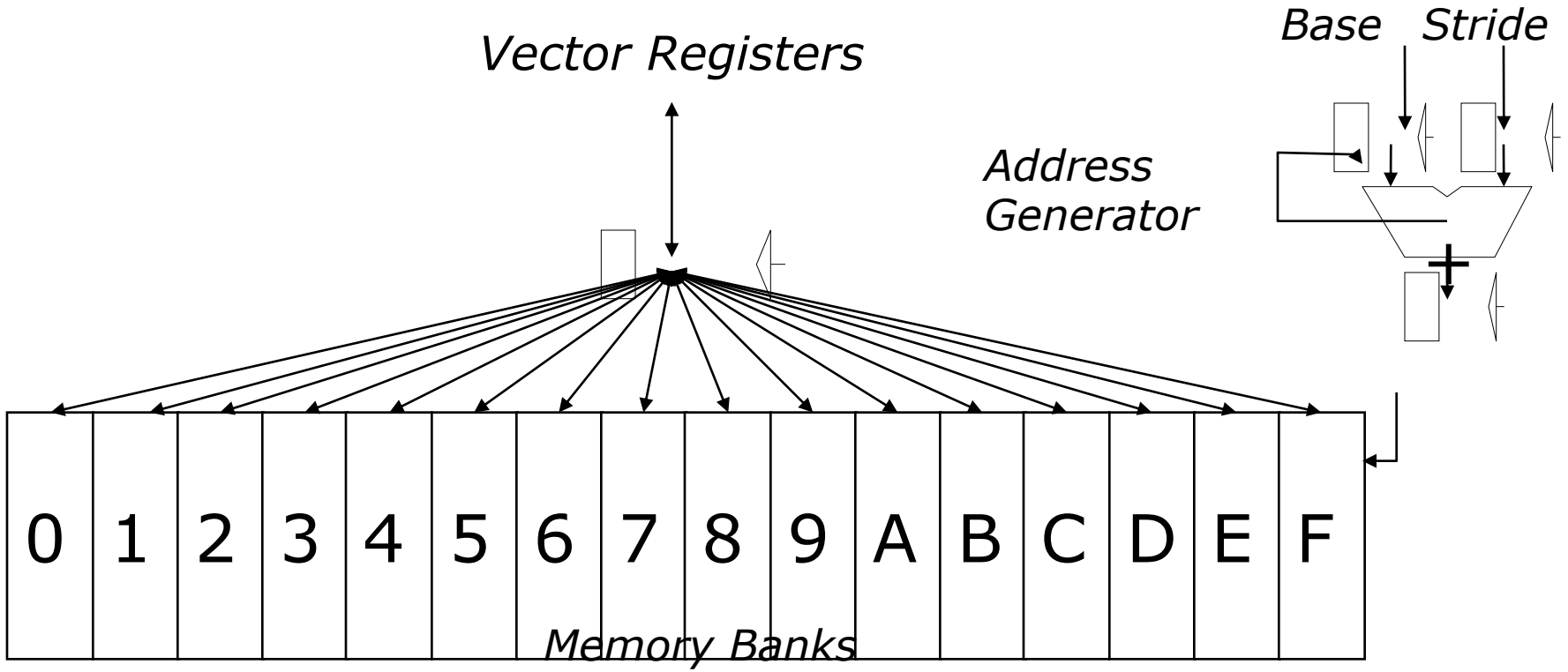
Vector Instruction Execution



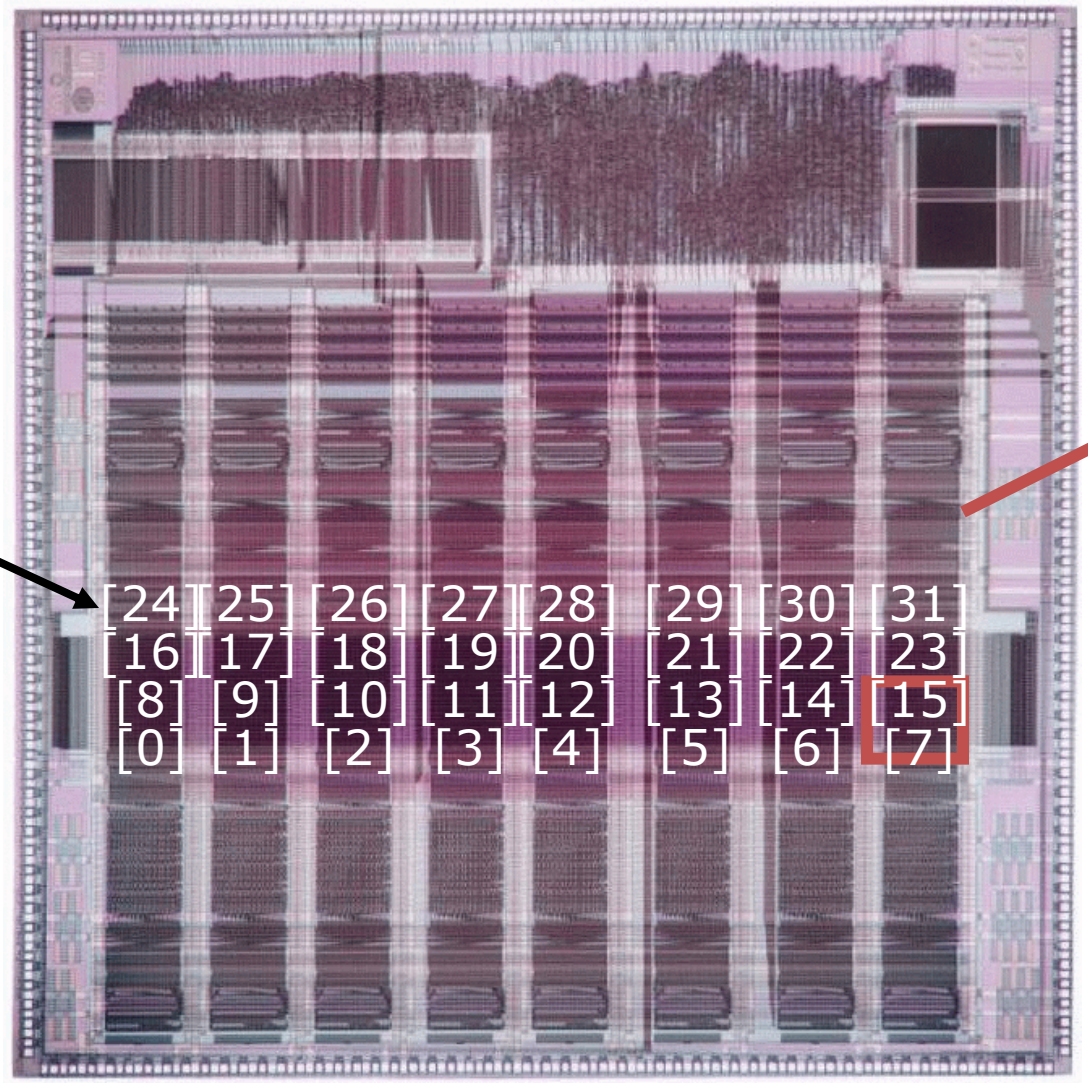
Interleaved Vector Memory System

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

- *Bank busy time*: Time before bank ready to accept next request

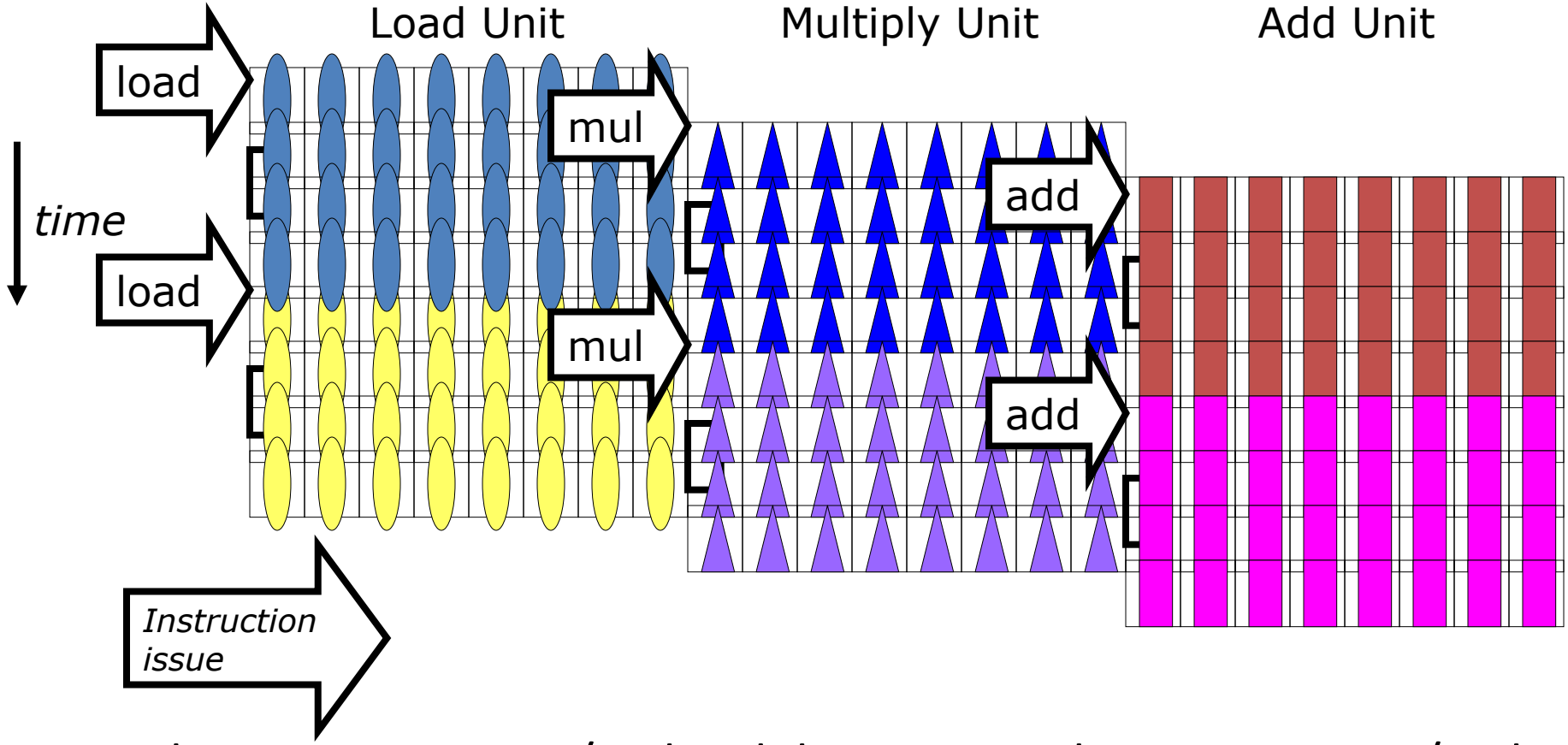


T0 Vector Microprocessor (UCB/ICSI, 1995)



Vector Instruction Parallelism

- **多条向量指令可重叠执行(链接技术)**
  - 例如：每个向量 32 个元素，8 lanes（车道）



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Execution Time

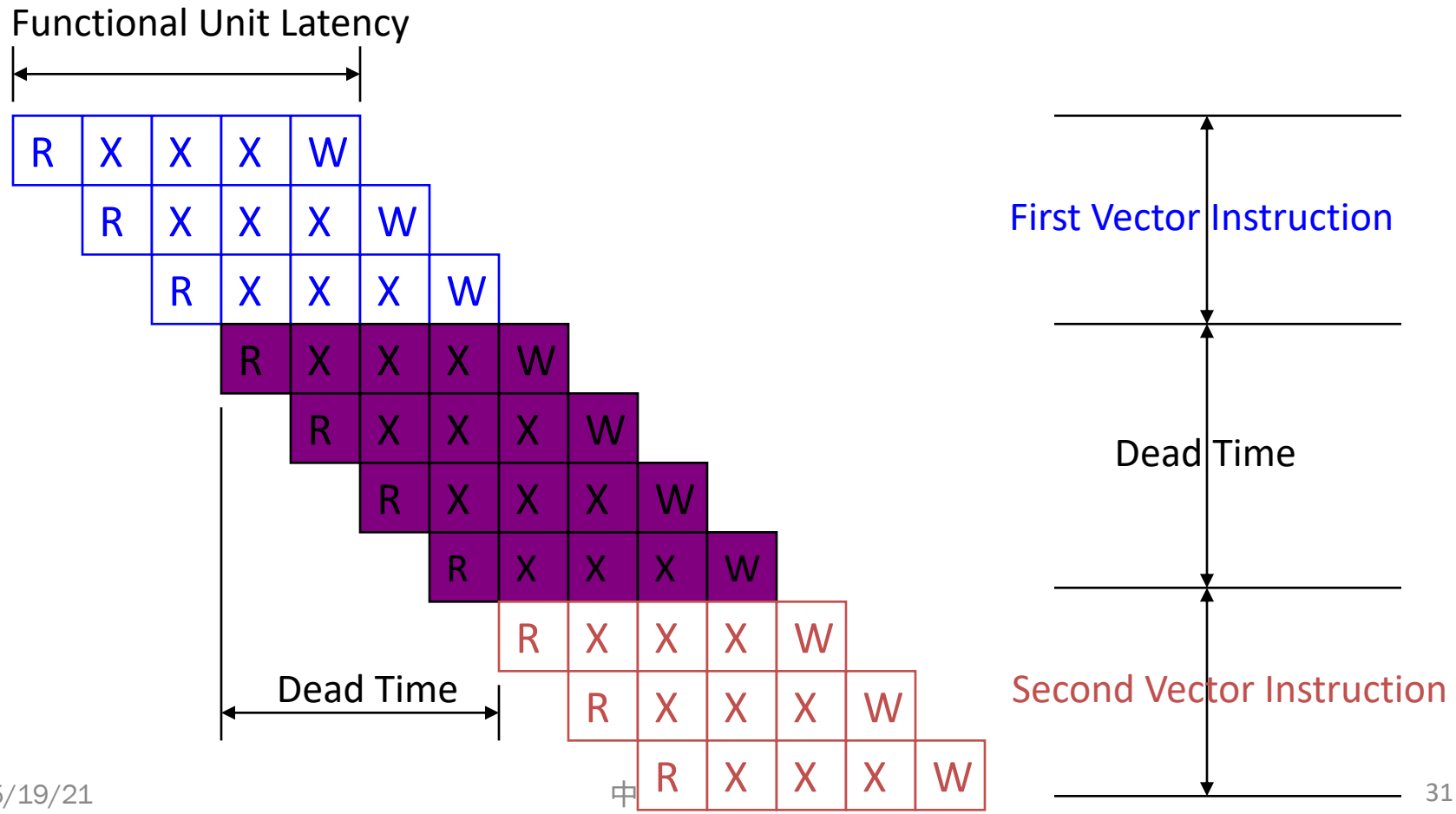
- *Time* = f(vector length, data dependencies, struct. hazards)
- *Initiation rate*: 功能部件消耗向量元素的速率
- *Convoy:* 可在同一时钟周期开始执行的指令集合 (no structural or data hazards)
- *Chime:* 执行一个*convoy*所花费的大致时间（approx. time）
- *m convoys take m chimes*;
  - 如果每个向量长度为*n*, 那么*m*个*convoys* 所花费的时间是*m*个*chimes*
  - 每个*chime*所花费的时间是*n*个*clocks*，该程序所花费的总时间大约为*m x n clock cycles* (忽略额外开销; 当向量长度较长时这种近似是合理的）

```
1: LV    V1,Rx     ;load vector X
2: MULV  V2,F0,V1  ;vector-scalar mult.
   LV    V3,Ry     ;load vector Y
3: ADDV  V4,V2,V3  ;add
4: SV    Ry,V4     ;store the result
```

4 convoys, 1 lane, VL=64
=> 4 x 64 = 256 clocks
(or 4 clocks per result)

Vector Startup

- **向量启动时间由两部分构成**
  - 功能部件延时：一个操作通过功能部件的时间
  - 截止时间或恢复时间（dead time or recovery time ）：运行下一条向量指令的间隔时间



VMIPS Start-up Time

Start-up time: FU 部件流水线的深度

Operation	Start-up penalty (from CRAY-1)
Vector load/store	12
Vector multiply	7
Vector add	6

Assume convoys don't overlap; vector length = n

Convoy	*Start*	*1st result*	*last result*	
1. LV	0	12	11+n	(12+n-1)
2. MULV, LV	12+n	12+n+7	18+2n	*Multiply startup*
	12+n	12+n+12	23+2n	*Load start-up*
3. ADDV	24+2n	24+2n+6	29+3n	*Wait convoy 2*
4. SV	30+3n	30+3n+12	41+4n	*Wait convoy 3*

Vector Length

- 当向量的长度不是64时（假设向量寄存器的长度是64）怎么办？
- vector-length register (VLR) 控制特定向量操作的长度, 包括向量的load/store. (当然一次操作的向量的长度不能 > 向量寄存器的长度) 例如：

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

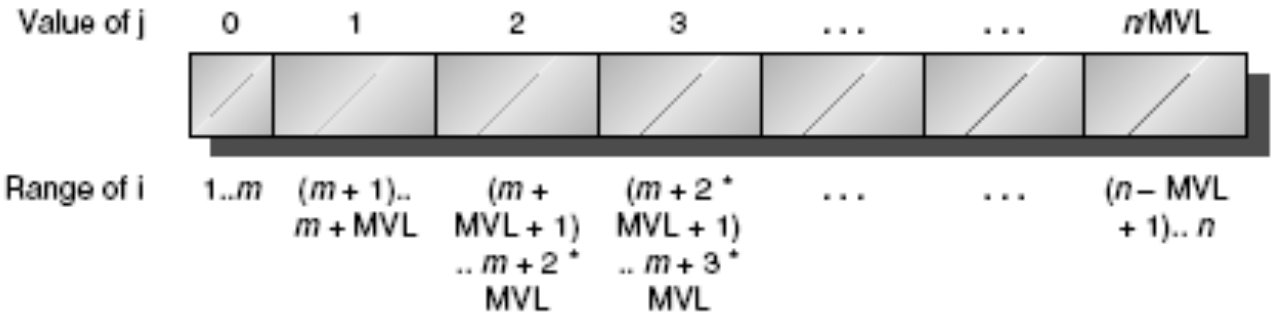
n的值只有在运行时才能知道

n > Max. Vector Length (MVL)怎么办？

Strip Mining（分段开采）

- 假设Vector Length > Max. Vector Length (MVL)?
- Strip mining: 产生新的代码，使得每个向量操作的元素数 ≤ MVL
- 第一次循环做最小片(n mod MVL), 以后按VL = MVL操作

```
   low = 1
   VL = (n mod MVL)  /*find the odd size piece*/
   do 1  j = 0, (n / MVL)  /*outer loop*/
       do 10 i = low, low+VL-1  /*runs for length VL*/
                Y(i) = a*X(i) + Y(i)  /*main operation*/
10     continue
       low = low+VL  /*start of next vector*/
       VL = MVL  /*reset the length to max*/
1      continue
```



Strip Mining的向量执行时间计算

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

试计算A=B×s，其中A,B为长度为200的向量（每个向量元素占8个字节），s是一个标量。向量寄存器长度为64。各功能部件的启动时间如前所述，求总的执行时间，(T_{loop} = 15)

```
        ADDI R2,R0,#1600    ;total # bytes in vector
        ADD R2,R2,Ra        ;address of the end of A vector
        ADDI R1,R0,#8       ;loads length of 1st segment
        MOVI2S  VLR,R1      ;load vector length in VLR
        ADDI R1,R0,#64      ;length in bytes of 1st segment
        ADDI R3,R0,#64      ;vector length of other segments
Loop:   LV  V1,Rb           ;load B
        MULSV V2,V1,Fs      ;vector * scalar
        SV Ra,V2            ;store A
        ADD Ra,Ra,R1        ;address of next segment of A
        ADD  Rb,Rb,R1       ;address of next segment of B
        ADDI R1,R0,#512     ;load byte offset next segment
        MOVI2S VLR,R3       ;set length to 64 elements
        SUB R4,R2,Ra        ;at the end of A?
        BNEZ R4,Loop        ;if not, go back
```

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

Tstart = 12 + 7 + 12 = 31

T200 = 660+4*31 = 784

每一元素的执行时间 = 784/200 = 3.9

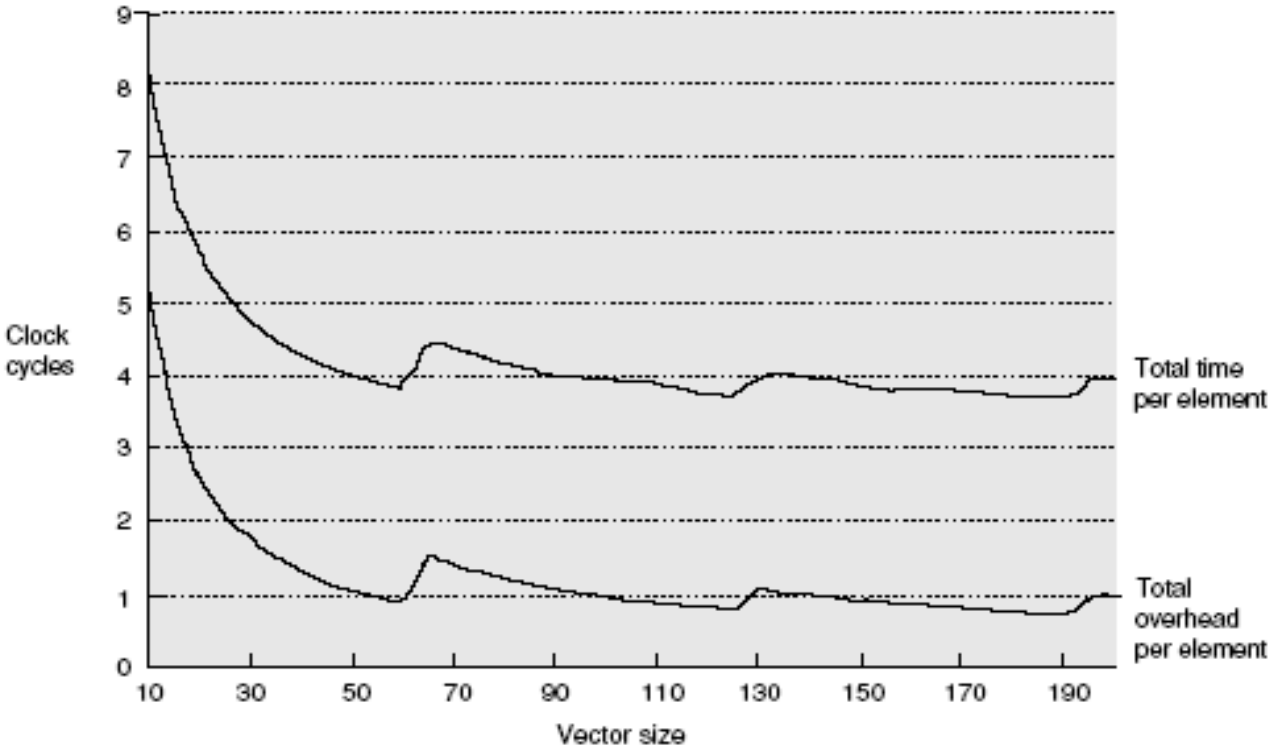


Figure G.9 The total execution time per element and the total overhead time per element versus the vector length for the example on page G-19. For short vectors the total start-up time is more than one-half of the total time, while for long vectors it reduces to about one-third of the total time. The sudden jumps occur when the vector length crosses a multiple of 64, forcing another iteration of the strip-mining code and execution of a set of vector instructions. These operations increase T_n by $T_{loop} + T_{start}$.

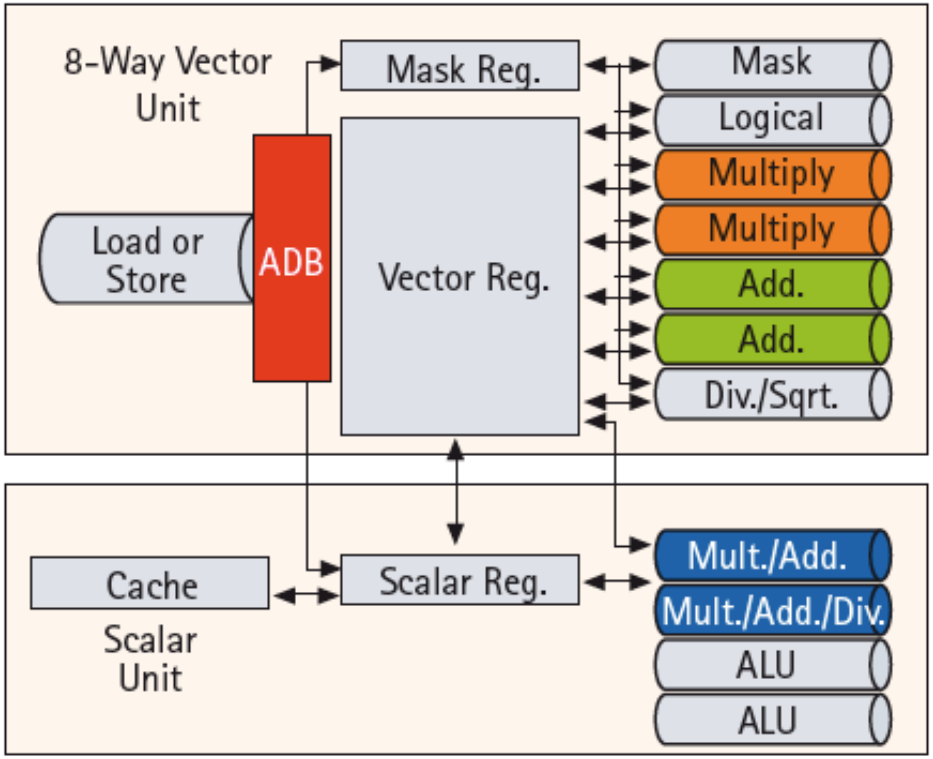
Common Vector Metrics

- **R_∞: 当向量长度为无穷大时的向量流水线的最大性能。常在评价峰值性能时使用，单位为MFLOPS**
  - 实际问题是向量长度不会无穷大，start-up的开销还是比较大的
  - R_n 表示向量长度为n时的向量流水线的性能
- **$N_{1/2}$: 达到R∞ 一半的值所需的向量长度，是评价向量流水线start-up 时间对性能的影响。**
- **N_V：向量流水线方式的工作速度优于标量串行方式工作时所需的向量长度临界值。**
  - 该参数既衡量建立时间，也衡量标量、向量速度比对性能的影响

Example Vector Machines

Machine	Year	Clock(MHZ)	Regs	Elements	Fus	LSUs
Cray 1	1976	80	8	64	6	1
Cray XMP	1983	120	8	64	8	2L, 1S
Cray YMP	1988	166	8	64	8	2L, 1S
Cray C-90	1991	240	8	128	8	4
Cray T-90	1996	455	8	128	8	4
Conv. C-1	1984	10	8	128	4	1
Conv. C-4	1994	133	16	128	3	1
Fuj. VP200	1982	133	8-256	32-1024	3	2
Fuj. VP300	1996	100	8-256	32-1024	3	2
NEC SX/2	1984	160	8+8K	256+var	16	8
NEC SX/3	1995	400	8+8K	256+var	16	8

A Modern Vector Super: NEC SX-9 (2008)



- 65nm CMOS technology
- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/cycle)
- Scalar unit (1.6 GHz)
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache

- Memory system provides 256GB/s DRAM bandwidth per CPU
- Up to 16 CPUs and up to 1TB DRAM form shared-memory *node*
  - total of 4TB/s bandwidth to shared DRAM memory
- Up to 512 nodes connected via 128GB/s network links (message passing between nodes)

Picture from NEC article “A hardware overview of SX-6 and SX-7 supercomputer”

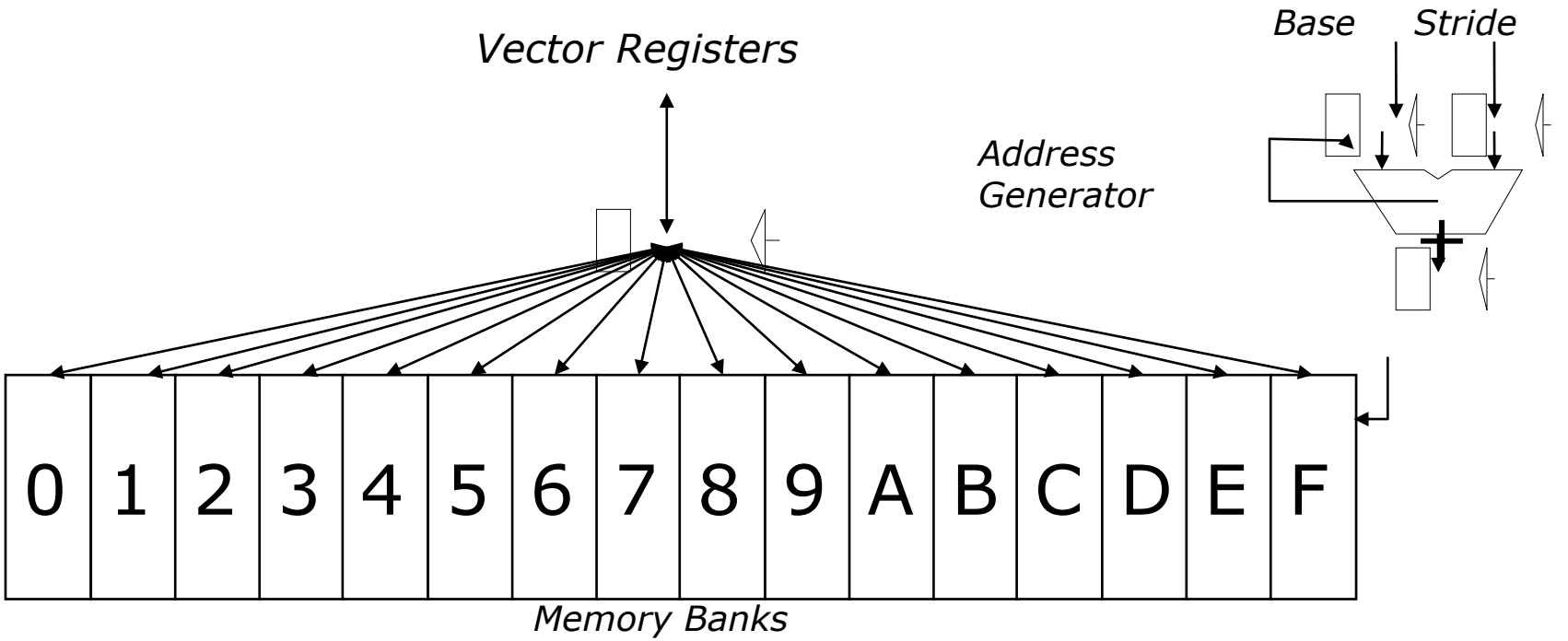
Vector Linpack Performance (MFLOPS)

Matrix Inverse (gaussian elimination)

Machine	Year	Clock(Mhz)	100x100	1kx1k	Peak(Procs)
Cray 1	1976	80	12	110	160(1)
Cray XMP	1983	120	121	218	940(4)
Cray YMP	1988	166	150	307	2,667(8)
Cray C-90	1991	240	387	902	15,238(16)
Cray T-90	1996	455	705	1603	57,600(32)
Conv. C-1	1984	10	3	--	20(1)
Conv. C-4	1994	136	160	2531	3,240(4)
Fuj. VP200	1982	133	18	422	533(1)
NEC SX/2	1984	166	43	885	1,300(1)
NEC SX/3	1995	400	368	2757	25,600(4)

Interleaved Vector Memory System

- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
  - *Bank busy time*: Time before bank ready to accept next request
  - *If stride = 1 & consecutive elements interleaved across banks & number of banks >= bank latency, then can sustain 1 element/cycle throughput*



Example(AppF F-15) Suppose we want to fetch a vector of 64 elements starting at byte address 136,and a memory access takes 6 clocks. Bank busy time is 2 clocks. How many memory banks must we have to support one fetch per clock cycle? With what addresses are the banks accessed? When will the various elements arrive at the CPU?

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

Figure F.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

Vector Stride

- 假设处理顺序相邻的元素在存储器中不顺序存储。例如

```
      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)
```

- B 或 C 的两次访问不会相邻 (相隔800 bytes)
- *stride*: 向量中相邻元素间的距离
  => LVWS (load vector with stride) instruction
- Strides => 会导致体冲突
  (e.g., stride = 32 and 16 banks)

Memory operations

- **Load/store 操作成组地在寄存器和存储器之间移动数据**
- **三类寻址方式**
  - Unit stride（单步长)
    - Fastest
  - Non-unit (constant) stride（常数步长)
  - Indexed (gather-scatter) (间接寻址)
    - 等价于寄存器间接寻址方式
    - 对稀疏矩阵有效
    - 用于向量化操作的指令增多

DAXPY (Y = a × X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
LD      F0,a
ADDI    R4,Rx,#512  ;last address to load
loop: LD      F2, 0(Rx)   ;load X(i)
      MULTD   F2,F0,F2    ;a*X(i)
      LD      F4, 0(Ry)   ;load Y(i)
      ADDD    F4,F2,F4    ;a*X(i) + Y(i)
      SD      F4,0(Ry)    ;store into Y(i)
      ADDI    Rx,Rx,#8    ;increment index to X
      ADDI    Ry,Ry,#8    ;increment index to Y
      SUB     R20,R4,Rx   ;compute bound
      BNZ     R20,loop    ;check if done
```

```
LD      F0,a        ;load scalar a
LV      V1,Rx       ;load vector X
MULTS   V2,F0,V1    ;vector-scalar mult.
LV      V3,Ry       ;load vector Y
ADDV    V4,V2,V3    ;add
SV      Ry,V4       ;store the result
```

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

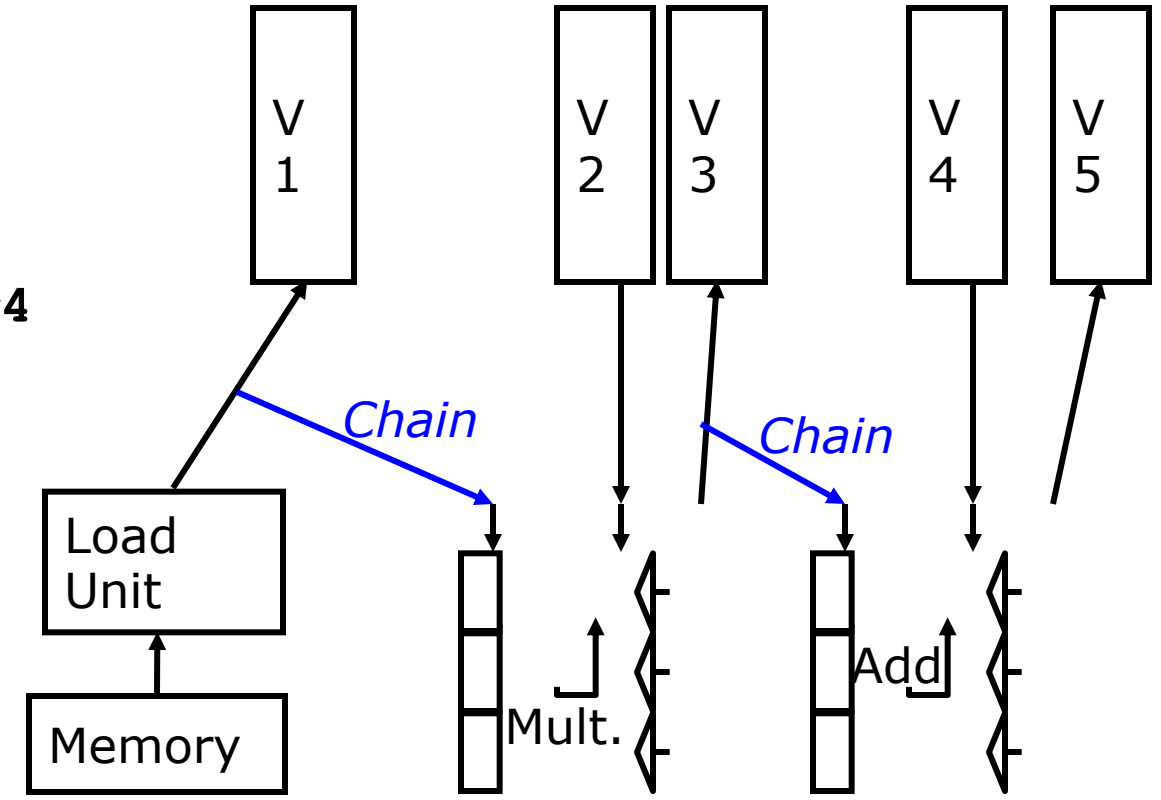
64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Vector Opt#1: Vector Chaining

- 寄存器定向路径的向量机版本
- 首次在Cray-1上使用

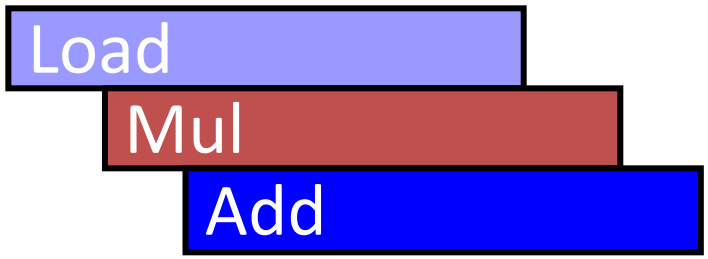
```
LV    v1
MULV  v3,v1,v2
ADDV  v5,v3, v4
```



Vector Chaining Advantage

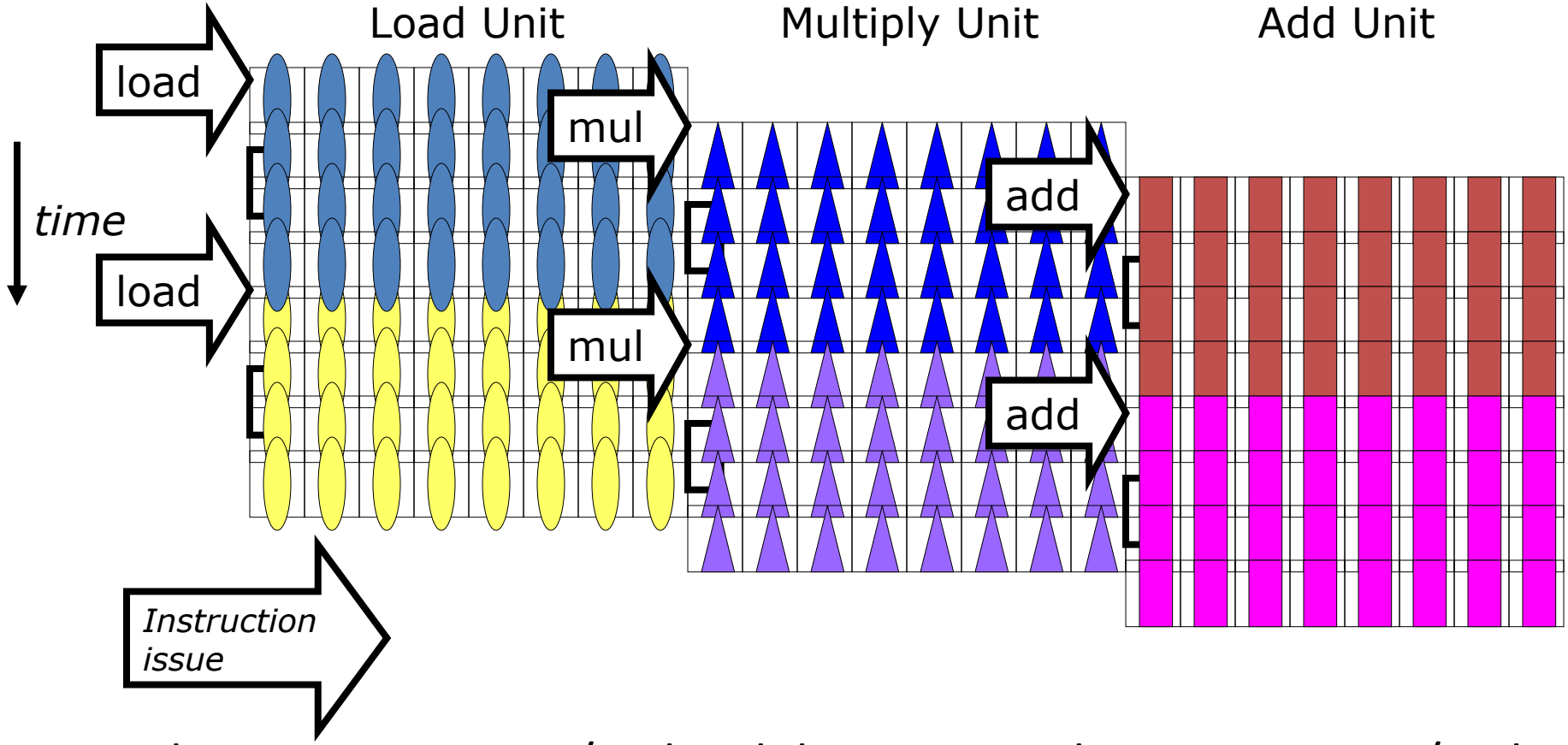
- 不采用链接技术，必须处理完前一条指令的最后一个元素，才能启动下一条相关的指令

- 采用链接技术，前一条指令的第一个结果出来后，就可以启动下一条相关指令的执行



Vector Instruction Parallelism

- **多条向量指令可重叠执行(链接技术)**
  - 例如：每个向量 32 个元素，8 lanes（车道）



Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Opt #2: Conditional Execution

- **Suppose:**

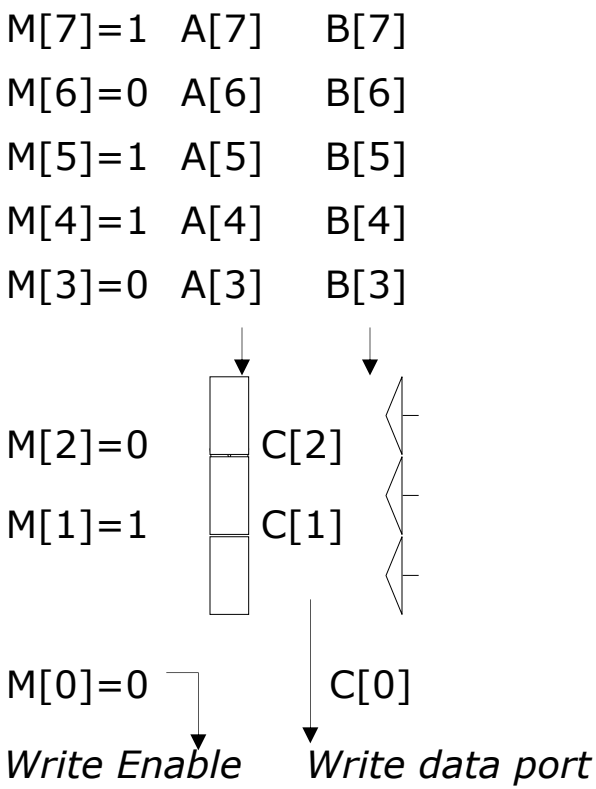
```
      do 100 i = 1, 64
           if (A(i) .ne. 0) then
               A(i) = A(i) - B(i)
           endif
100   continue
```

- ***vector-mask control*** **使用长度为MVL的布尔向量控制向量指令的执行**
- ***当vector-mask register*** **使能时，向量指令操作仅对 vector-mask register中对应位为1的分量起作用**

Masked Vector Instructions

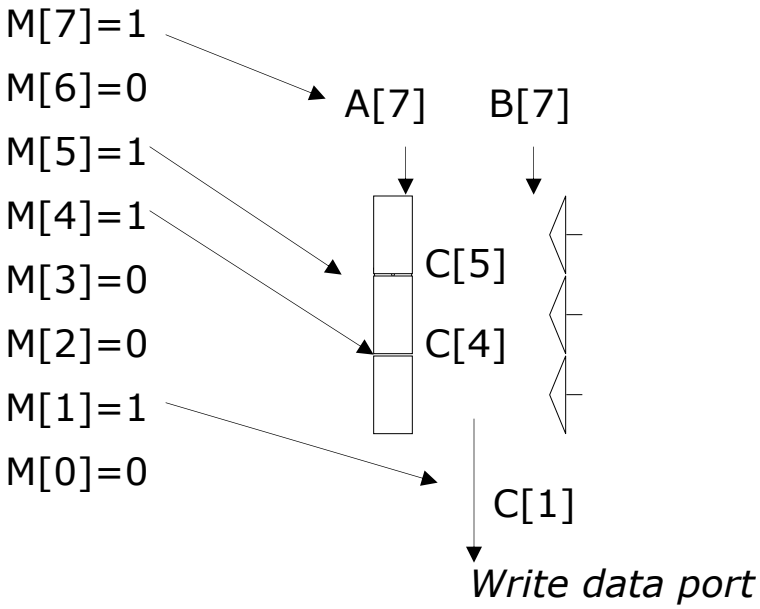
Simple Implementation

- execute all N operations, turn off result writeback according to mask



Density-Time Implementation

- scan mask vector and only execute elements with non-zero masks



```
LV   V1,Ra          ;  load vector A into V1
LV V2,Rb            ;  load vector B
L.D F0,#0           ; load FP zero into F0
SNEVS.D V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D V1,V1,V2     ;subtract under vector mask
CVM                  ;set the vector mask to all 1s
SV Ra,V1            ;store the result in A
```

- **使用vector-mask寄存器的缺陷**
  - 简单实现时，条件不满足时向量指令仍然需要花费时间
  - 有些向量处理器带条件的向量执行仅控制向目标寄存器的写操作，可能会有除法错。

Vector Opt #3: Sparse Matrices

- **Suppose:**

```
          do   100 i = 1,n
100       A(K(i)) = A(K(i)) + C(M(i))
```

- ***gather*** **(LVI) operation 使用*index vector*中给出的偏移再加基址来读取 => a nonsparse vector in a vector register**
- **这些元素以密集的方式操作完成后，再使用同样的index vector存储到稀疏矩阵的对应位置**
- **这些操作编译时可能无法完成。主要原因：编译器无法预知Ki以及是否有数据相关**
- **使用CVI 设置步长（ index 0, 1xm, 2xm, ..., 63xm ）**

Sparse Matrix Example

- Cache (1993) vs. Vector (1988)

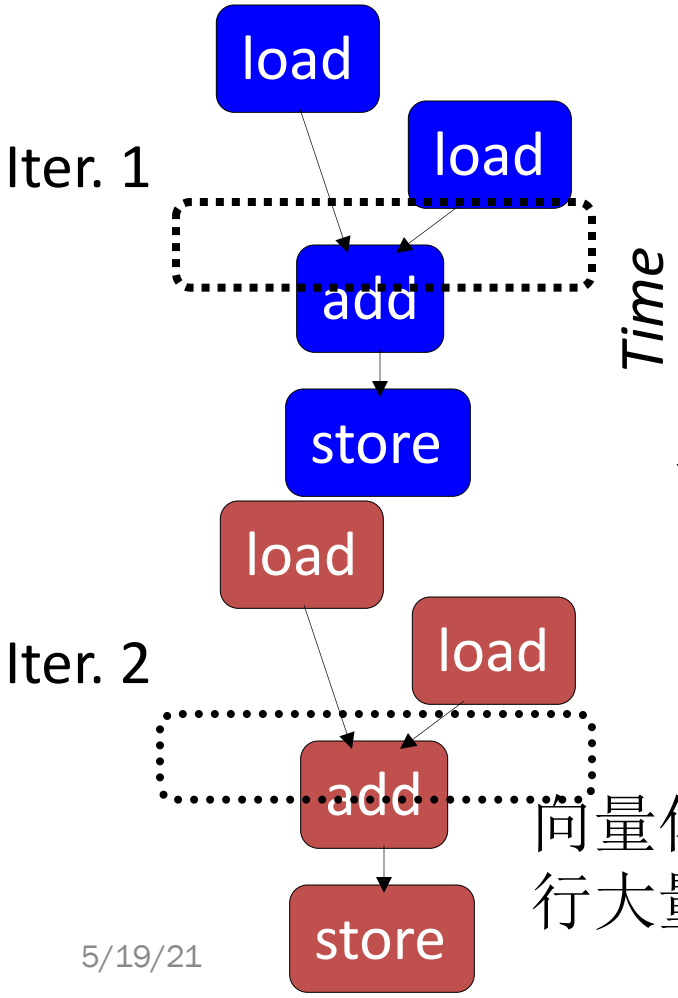
	IBM RS6000	Cray YMP
Clock	72 MHz	167 MHz
Cache	256 KB	0.25 KB
Linpack	140 MFLOPS	160 (1.1)
Sparse Matrix (Cholesky Blocked)	17 MFLOPS	125 (7.3)

- Cache: 1 address per cache block (32B to 64B)
- Vector: 1 address per element (4B)

Automatic Code Vectorization

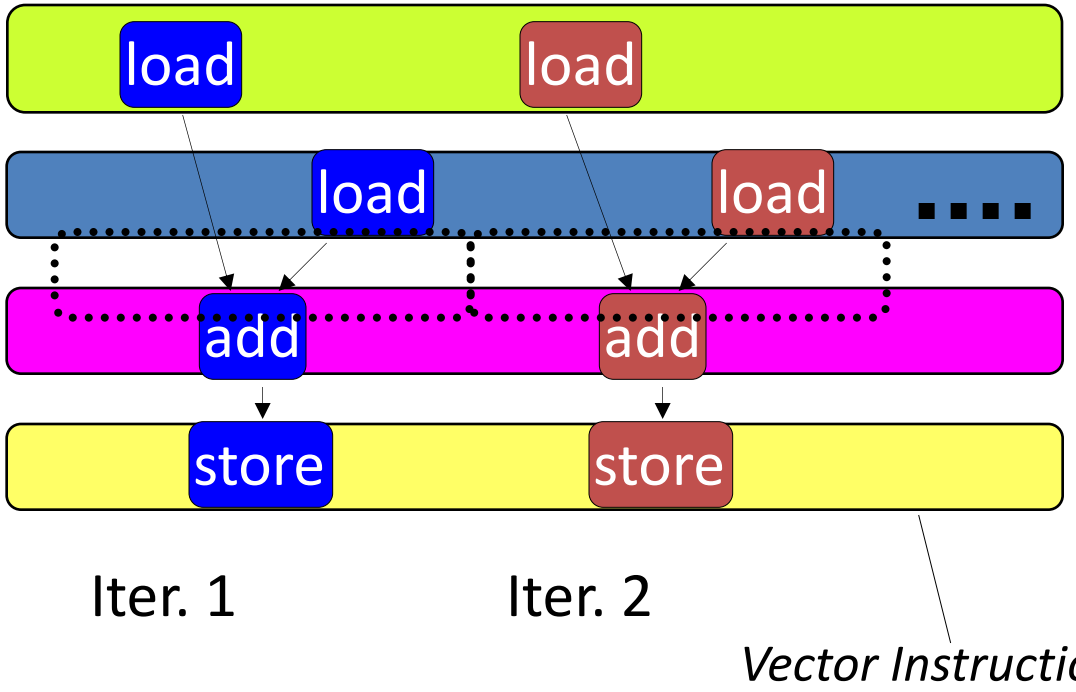
```
for (i=0; i < N; i++)
    C[i] = A[i] + B[i];
```

Scalar Sequential Code



Time

Vectorized Code



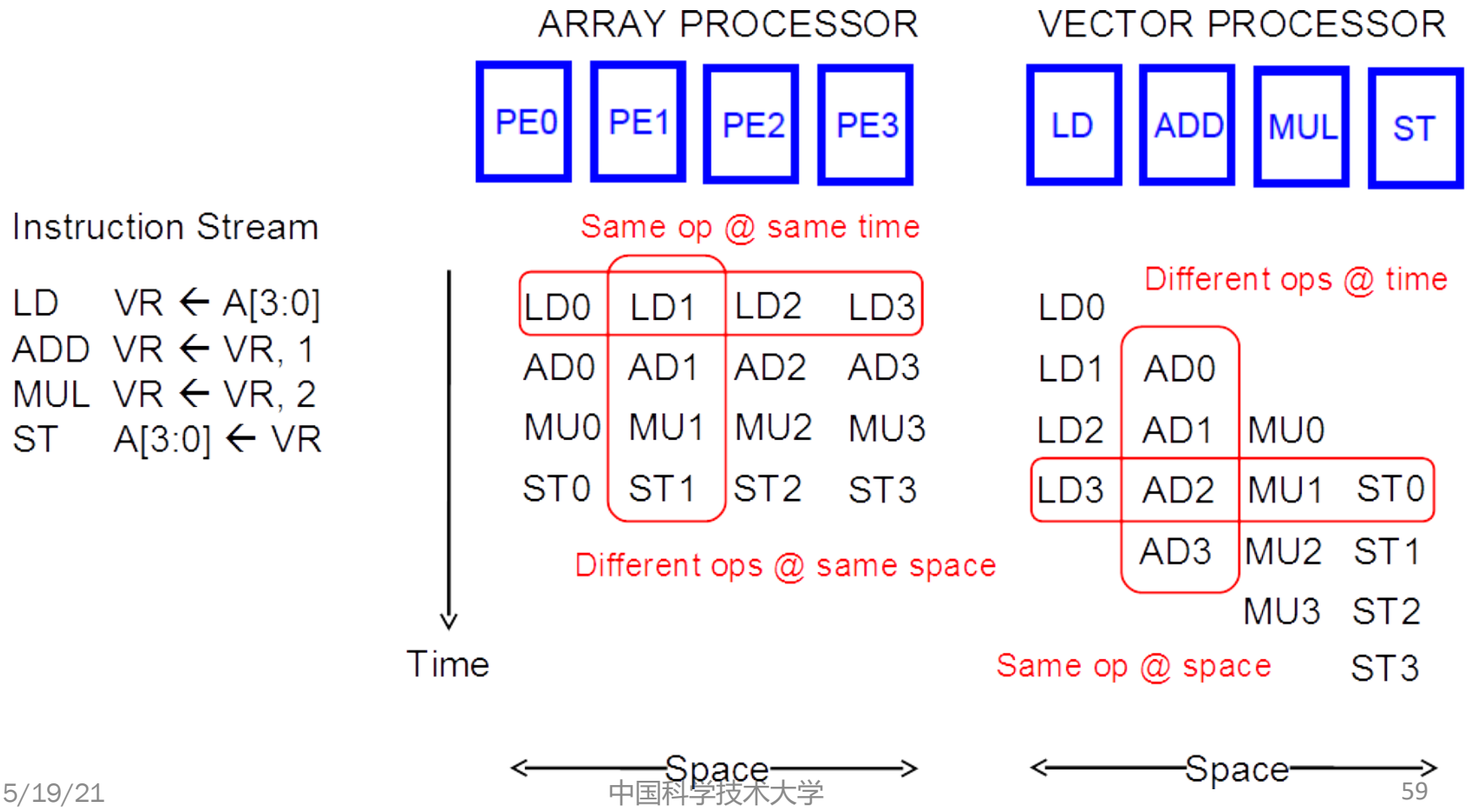
向量化是指在编译期间对操作重定序⇒ 需要进行大量的循环相关分析

Vector/SIMD Processing Summary

- **Vector/SIMD 机器适合挖掘数据级并行**
  - 同样的操作作用于不同的数据元素
  - 向量内的元素操作独立，可有效提高性能，简化设计
- **性能的提升受限于代码的向量化**
  - 标量操作限制了向量机的性能
  - Amdahl's Law
- **很多ISA包含SIMD操作指令**
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

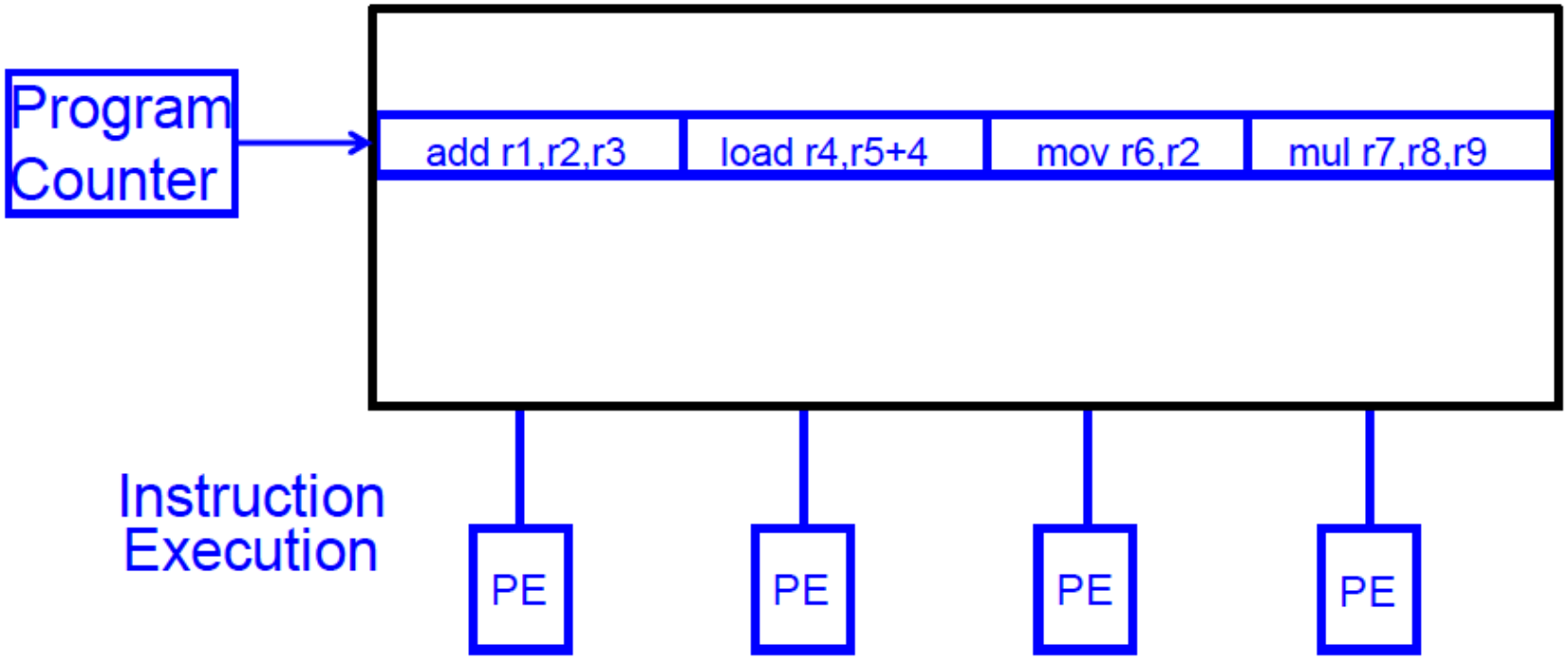
Array vs. Vector Processors

Array processor：又称为并行处理机、SIMD处理器。其核心是一个由**多个处理单元**构成的**阵列**，用**单一的控制部件**来控制多个处理单元对**各自的数据**进行**相同的运算**和操作。



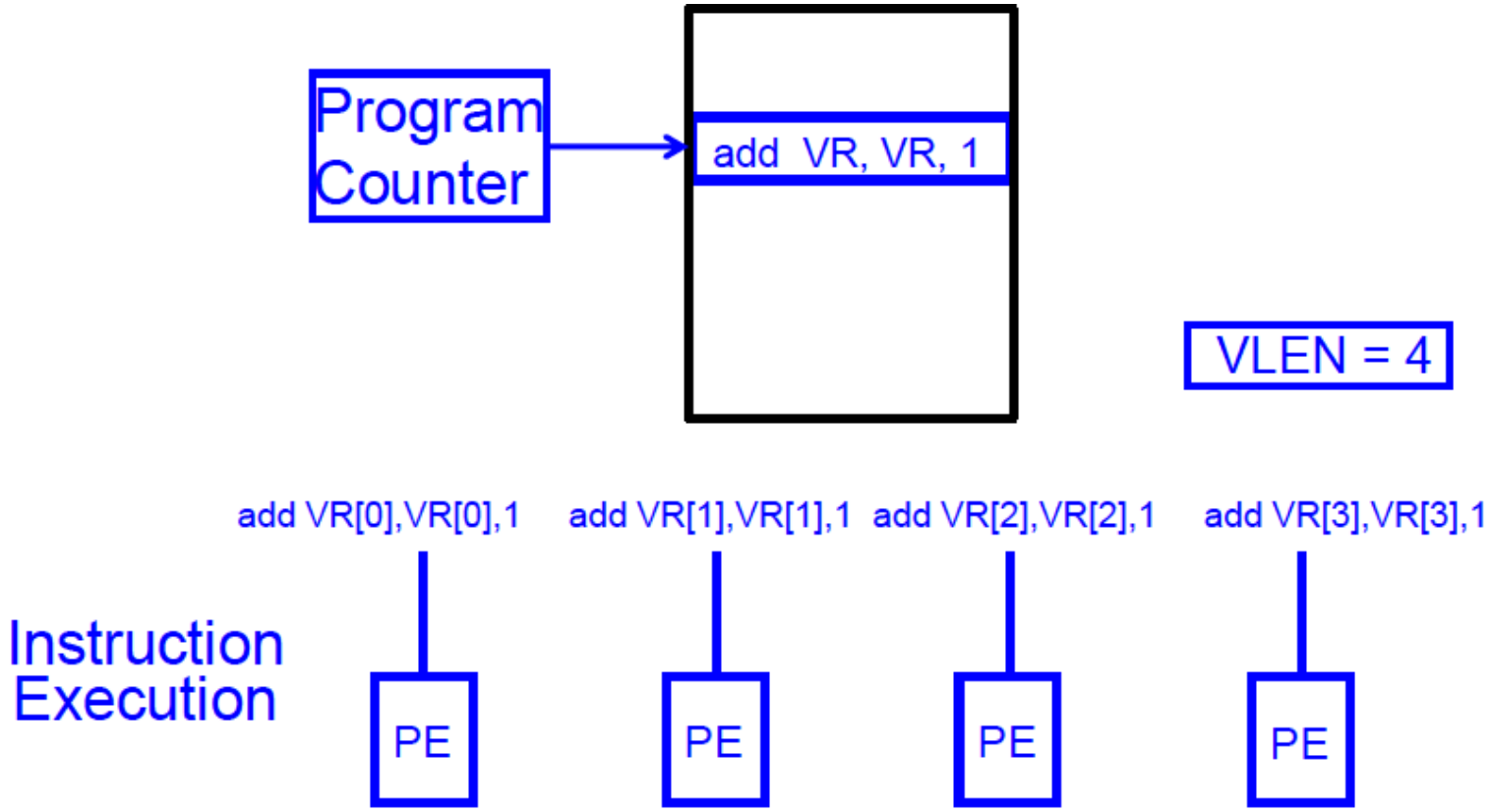
SIMD Array Processing vs. VLIW

- VLIW: 多个独立的操作由编译器封装在一起



SIMD Array Processing vs. VLIW

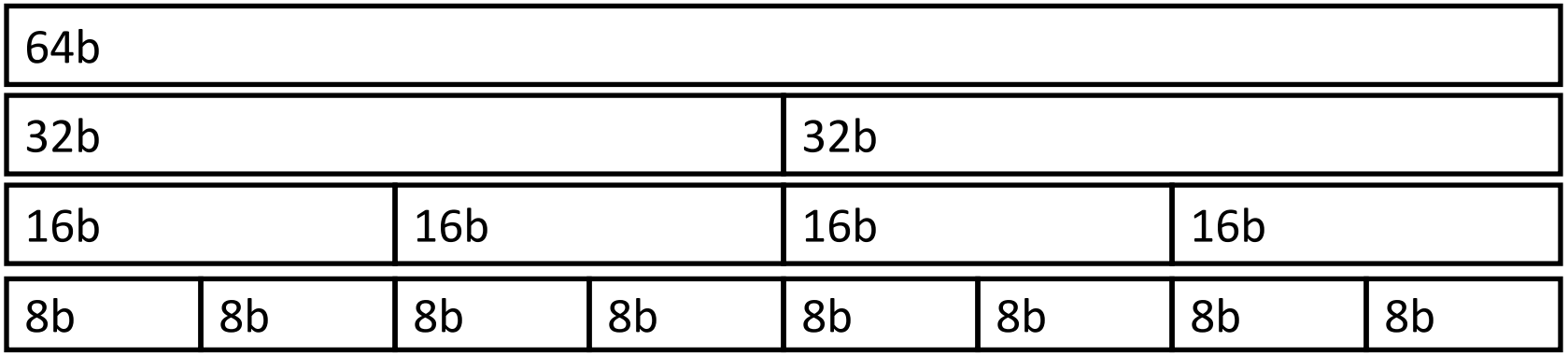
- **Array processor: 单个操作作用在多个不同的数据元素上**

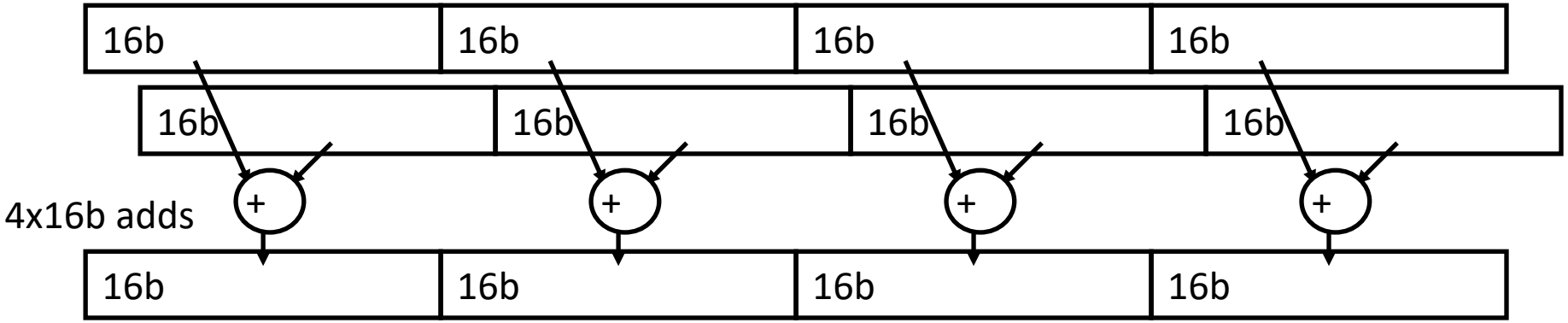


Multimedia Extensions (aka SIMD extensions)

- **在已有的ISA中添加一些向量长度很短的向量操作指令**
- **将已有的 64-bit 寄存器拆分为 2x32b or 4x16b or 8x8b**
  - 1957年，Lincoln Labs TX-2 将36bit datapath 拆分为 2x18b or 4x9b
  - 新的设计具有较宽的寄存器
    - 128b for PowerPC Altivec, Intel SSE2/3/4
    - 256b for Intel AVX (Advanced Vector Extensions)
- **单条指令可实现寄存器中所有向量元素的操作**

Multimedia Extensions (aka SIMD extensions)





Intel Pentium MMX Operations

- **idea: 一条指令操作同时作用于不同的数据元**
  - 全阵列处理
  - 用于多媒体操作

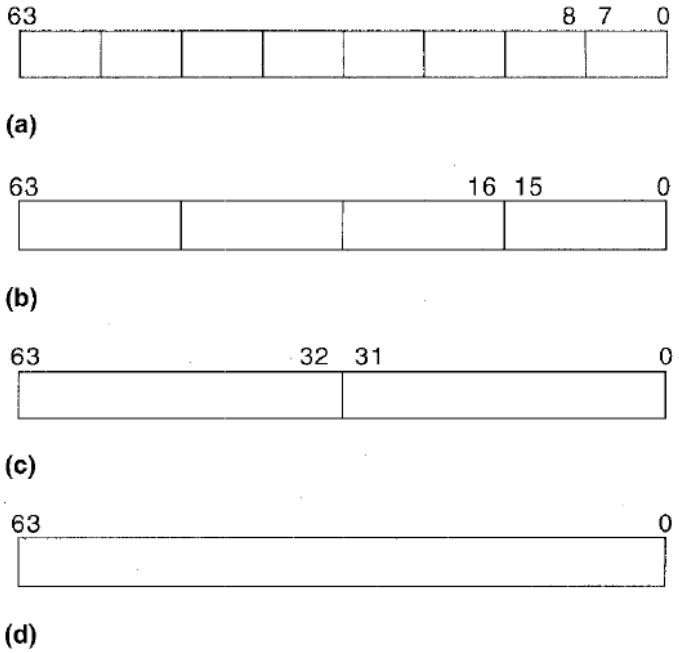


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

- No VLEN register
- Opcode determines data type:
  - 8 8-bit bytes
  - 4 16-bit words
  - 2 32-bit doublewords
  - 1 64-bit quadword
- Stride always equal to 1.

MMX Example: Image Overlaying (I)

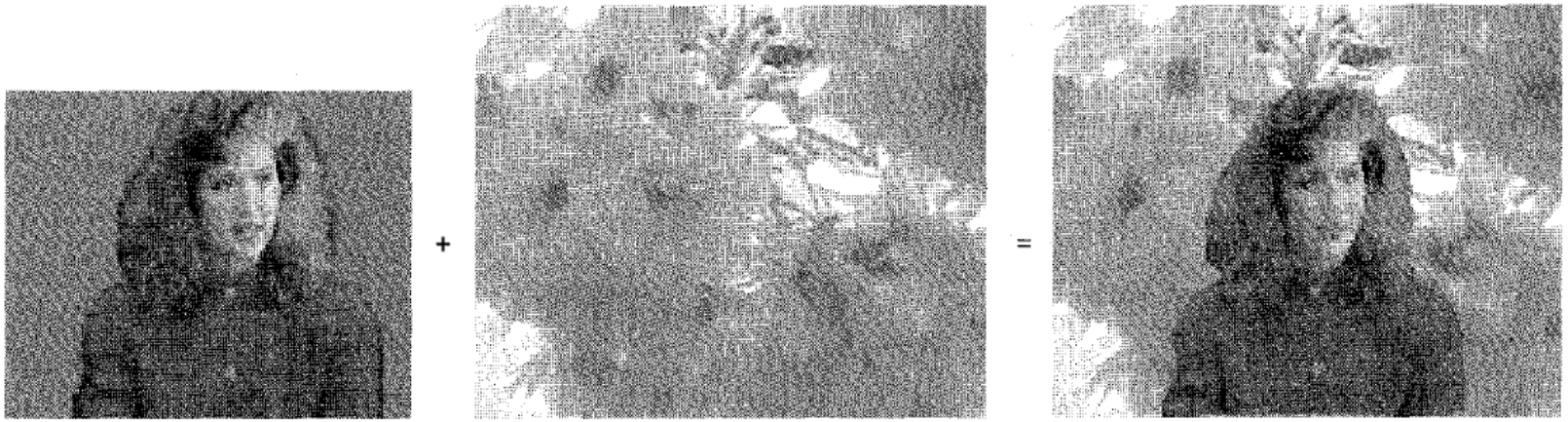


Figure 8. Chroma keying: image overlay using a background color.

PCMPEQB MM1, MM3

MM1	Blue	Blue	Blue	Blue	Blue	Blue	Blue	Blue
MM3	X7!=blue	X6!=blue	X5=blue	X4=blue	X3!=blue	X2!=blue	X1=blue	X0=blue
MM1	0x0000	0x0000	0xFFFF	0xFFFF	0x0000	0x0000	0xFFFF	0xFFFF

Bitmask

Figure 9. Generating the selection bit mask.

MMX Example: Image Overlaying (II)

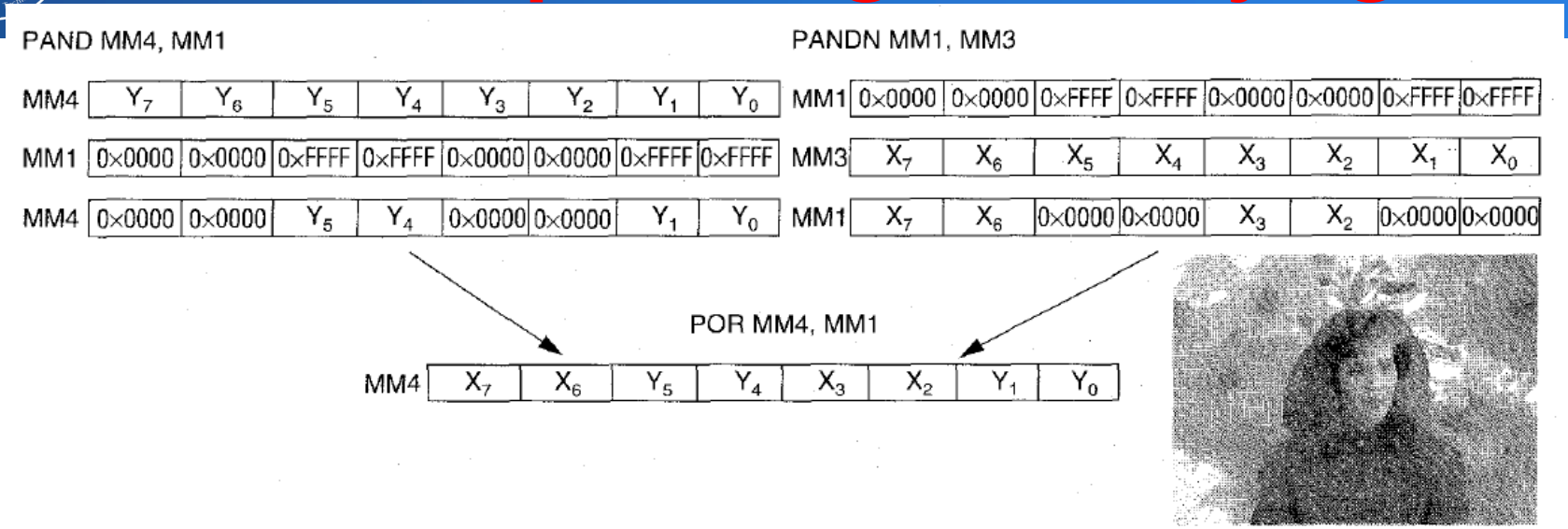


Figure 10. Using the mask with logical MMX instructions to perform a conditional select.

```
Movq     mm3, mem1    /* Load eight pixels from
                         woman's image
Movq     mm4, mem2    /* Load eight pixels from the
                         blossom image
Pcmpeqb  mm1, mm3
Pand     mm4, mm1
Pandn    mm1, mm3
Por      mm4, mm1
```

Figure 11. MMX code sequence for performing a conditional select.

Multimedia Extensions versus Vectors

- **受限的指令集:**
  - 无向量长度控制
  - Load/store操作无 常数步长寻址和 scatter/gather操作
  - loads 操作必须64/128-bit 边界对齐
- **受限的向量寄存器长度:**
  - 需要超标量发射以保持multiply/add/load 部件忙
  - 通过循环展开隐藏延迟增加了寄存器读写压力
- **在微处理器设计中向全向量化发展**
  - 更好地支持非对齐存储器访问
  - 支持双精度浮点数操作 (64-bit floating-point)
  - Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

-Review

- **向量处理机性能评估**
  - 向量指令流执行时间: Convey, Chimes, Start-up time
  - 其他指标： R∞ , N1/2 , NV
- **向量机的存储器访问**
  - 存储器组织：独立存储体、多体交叉方式
  - Stride : 固定步长（ 1 or 常数）, 非固定步长（ index ）
- **基于向量机模型的优化**
  - 链接技术
  - 有条件执行
  - 稀疏矩阵的操作
- **多媒体扩展指令**
  - 扩展的指令类型较少
  - 向量寄存器长度较短
- **GPU**

Recap: Vector/SIMD Processing Summary

- **Vector/SIMD 机器适合挖据规整的数据级并行**
  - 同样的操作作用在许多数据元素上
  - 提高性能、设计简单（向量内的操作相互独立）
- **性能的提升受限于代码的向量化**
  - 标量操作限制着向量机的性能
- **很多已有的ISA扩展了一些SIMD操作**
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD

Summary：向量体系结构

- **向量处理机基本概念**
  - 基本思想：两个向量的对应分量进行运算，产生一个结果向量
- **向量处理机基本特征**
  - VSIW-一条指令包含多个操作
  - 单条向量指令内所包含的操作相互独立
  - 以已知模式访问存储器-多体交叉存储系统
  - 控制相关少
- **向量处理机基本结构**
  - **向量指令并行执行**
  - **向量运算部件的执行方式-流水线方式**
  - **向量部件结构-多“道”结构-多条运算流水线**
- **向量处理机性能评估**
  - **向量指令流执行时间:** Convey, Chimes, Start-up time
  - **其他指标：** R_∞ , $N_{1/2}$, N_V
- **向量处理机性能优化**
  - 链接技术
  - 条件执行
  - 稀疏矩阵

Acknowledgements

- **These slides contain material developed and copyright by:**
  - John Kubiatowicz (UCB)
  - Krste Asanovic (UCB)
  - John Hennessy (Standford)and David Patterson (UCB)
  - Chenxi Zhang (Tongji)
  - Muhamed Mudawar (KFUPM)
- **UCB material derived from course CS152、CS252、CS61C**
- **KFUPM material derived from course COE501、COE502**