《高级计算机系统结构》课程直播

2020. 5.9

听不到声音请及时调试声音设备，可以下课后补签到

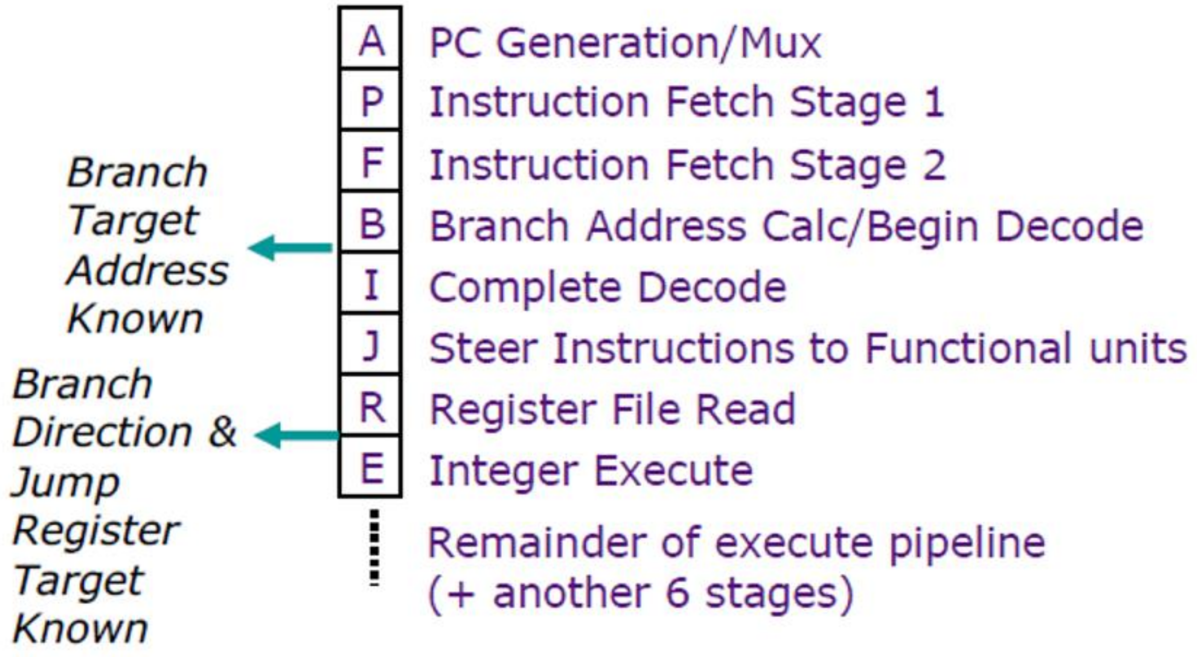
请将ZOOM名称改为“姓名”；

本节内容

- 通知：第三次作业发布，截止期5.17日
- 习题讲解
  - 转移预测
  - 多线程
- 数据级并行性
  - Vector, SIMD Architectures、GPU

From : H&P Computer Architecture: A Quantitative Approach,
Fifth Edition, （5th edition)

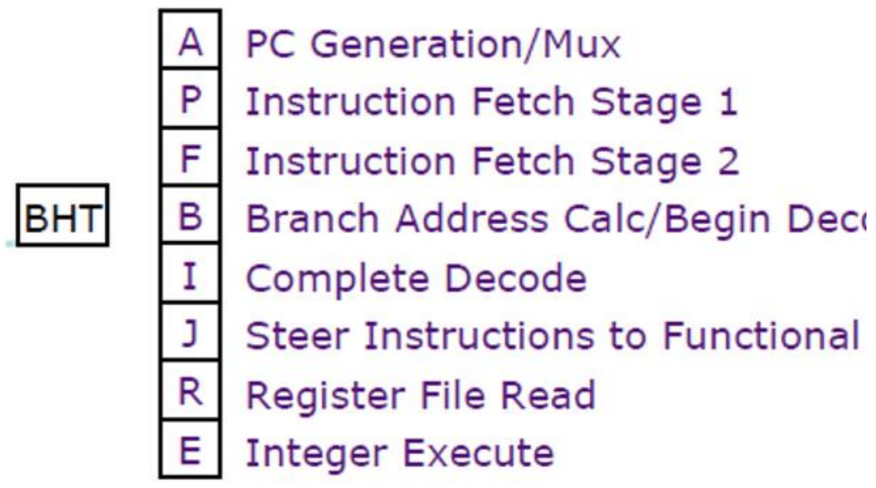
习题1：Pipeline of UltraSparc-III processor

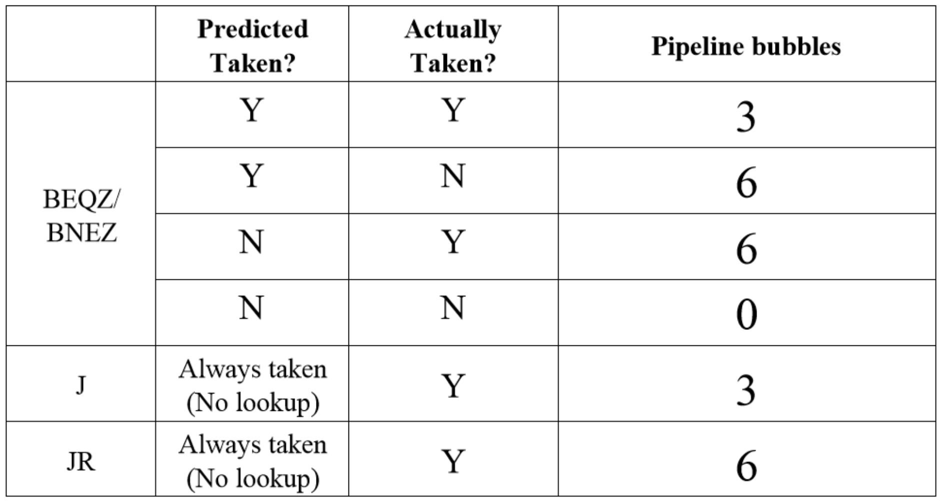

A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute
Remainder of execute pipeline (+ another 6 stages)
Branch Target Address Known
Branch Direction & Jump Register Target Known

Instruction	Taken known? (At the end of)	Target known? (At the end of)
BEQZ/BNEZ	R	B
J	B (always taken)	B
JR	B (always taken)	R

BHT 的作用

- First， we add a branch history table (BHT) in the fetch pipeline
- In the B stage (Branch Address Calc/Begin Decode), a conditional branch instruction (BEQZ/BNEZ) looks up the BHT, but an unconditional jump does not. If a branch is predicted to be taken
- some of the instructions are flushed and the PC is redirected to the calculated branch target address.
- The instruction at PC+4 is fetched by default unless PC is redirected by an older instruction.

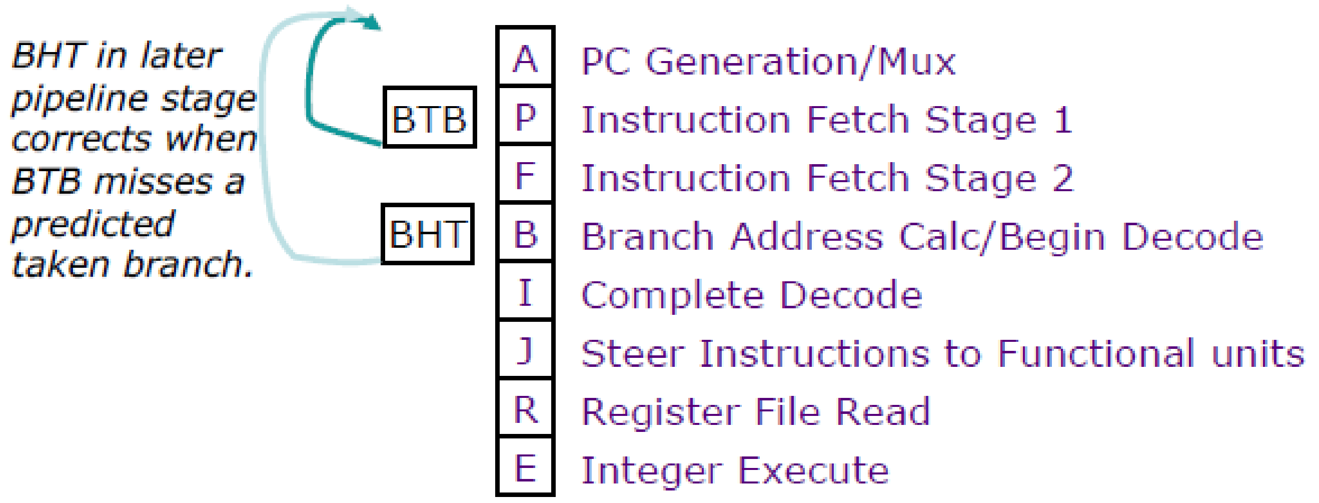

BHT
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Dec
I Complete Decode
J Steer Instructions to Functional
R Register File Read
E Integer Execute



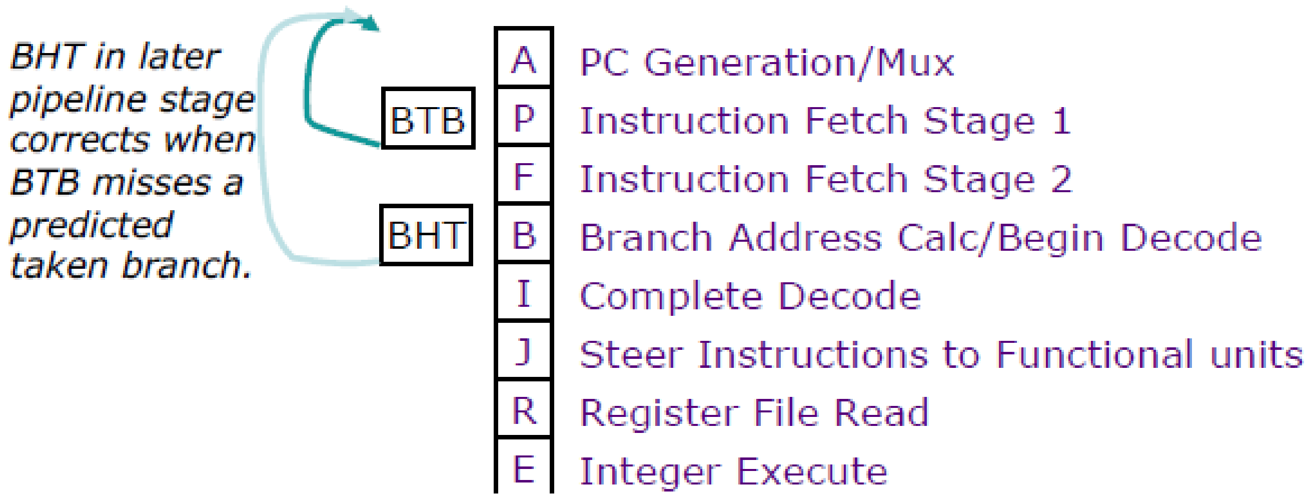
	Predicted Taken?	Actually Taken?	Pipeline bubbles
BEQZ/ BNEZ	Y	Y	3
	Y	N	6
	N	Y	6
	N	N	0
J	Always taken (No lookup)	Y	3
JR	Always taken (No lookup)	Y	6

BTB的作用

- 在取指阶段增加一个转移目标缓冲器BTB
- BTB表中,会为预测为“转移”的jump或branch 记录一对表项(当前指令的PC，转移目标指令的PC). 假设在BTB表中，转移目标即traget_PC一直是正确的（当然，转移方向的预测仍然有可能是错误的）;
- The BTB is looked up every cycle. If there is a match with the current PC, PC is redirected to the target_PC predicted by the BTB (unless PC is redirected by an older instruction); if not, it is set to PC+4.


BHT in later pipeline stage corrects when BTB misses a predicted taken branch.
BTB
BHT
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

BTB的作用


BHT in later pipeline stage corrects when BTB misses a predicted taken branch.
BTB
BHT
A PC Generation/Mux
P Instruction Fetch Stage 1
F Instruction Fetch Stage 2
B Branch Address Calc/Begin Decode
I Complete Decode
J Steer Instructions to Functional units
R Register File Read
E Integer Execute

	BTB Hit?	(BHT) Predicted Taken?	Actually Taken?	Pipeline bubbles
Conditional Branches	Y	Y	Y	1
	Y	Y	N	6
	Y	N	Y	Cannot occur
	Y	N	N	Cannot occur
	N	Y	Y	3
	N	Y	N	6
	N	N	Y	6
	N	N	N	0

习题2：多线程

- This problem evaluates the effectiveness of multithreading using a simple database benchmark.
- The benchmark searches for an entry in a linked list built from the following structure, which contains a key, a pointer to the next node in the linked list, and a pointer to the data entry.

```
struct node {
    int key;
    struct node *next;
    struct data *ptr;
}
```

习题2：多线程（续）

- The following MIPS code shows the core of the benchmark, which traverses the linked list and finds an entry with a particular key. Assume MIPS has no delay slots.

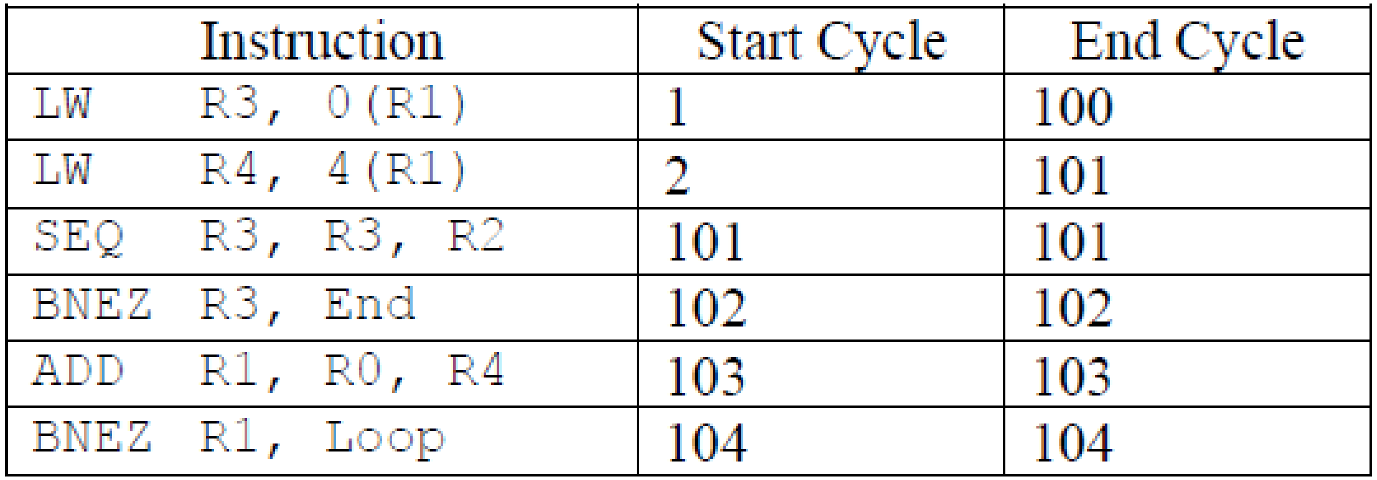
```
；R1: a pointer to the linked list
; R2: the key to find

loop: LW R3, 0(R1)          ; load a key
LW R4, 4(R1)                ; load the next pointer
SEQ R3, R3, R2             ; set R3 if R3 == R2
BNEZ R3, End               ; found the entry
ADD R1, R0, R4
BNEZ R1, Loop              ; check the next node
End:                ; R1 contains a pointer to the matching entry or zero if not found
```

- We run this benchmark on a single-issue in-order processor.
- If an instruction cannot be issued due to a data dependency, the processor stalls.
- Integer instructions take one cycle to execute and the result can be used in the next cycle.
- if SEQ is executed in cycle 1, BNEZ can be executed in cycle 2.
- We also assume that the processor has a perfect branch predictor with no penalty for both taken and not-taken branches.

the loop takes 104 cycles to execute

- Assume that our system does not have a cache. Each memory operation directly accesses main memory and takes 100 CPU cycles. The load/store unit is fully pipelined, and nonblocking. After the processor issues a memory operation, it can continue executing instructions until it reaches an instruction that is dependent on an outstanding memory operation.

- The following table summarizes the execution time for each instruction. From the table, the loop takes 104 cycles to execute.



Instruction	Start Cycle	End Cycle
`LW   R3, 0(R1)`	1	100
`LW   R4, 4(R1)`	2	101
`SEQ  R3, R3, R2`	101	101
`BNEZ R3, End`	102	102
`ADD  R1, R0, R4`	103	103
`BNEZ R1, Loop`	104	104

Problem A.

- Now we add zero-overhead multithreading to our pipeline.
- A processor executes multiple threads, each of which performs an independent search. Hardware mechanisms schedule a thread to execute each cycle.
- In our first implementation, the processor switches to a different thread every cycle using fixed round robin scheduling. Each of the N threads executes one instruction every N cycles.
- What is the minimum number of threads that we need to fully utilize the processor, i.e., execute one instruction per cycle?

- How does multithreading affect throughput (number of keys the processor can find within a given time) and latency (time processor takes to find an entry with a specific key)?

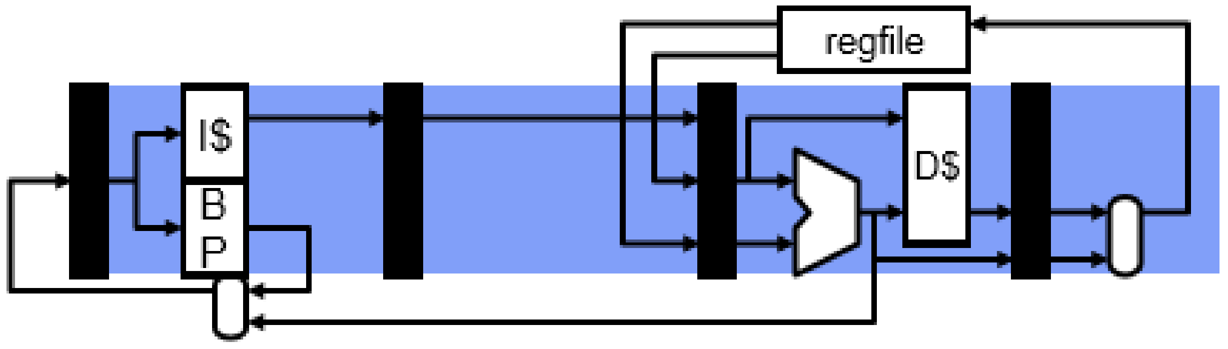
- Assume the processor switches to a different thread every cycle and is fully utilized.

- We change the processor to only switch to a different thread when an instruction cannot execute due to data dependency.

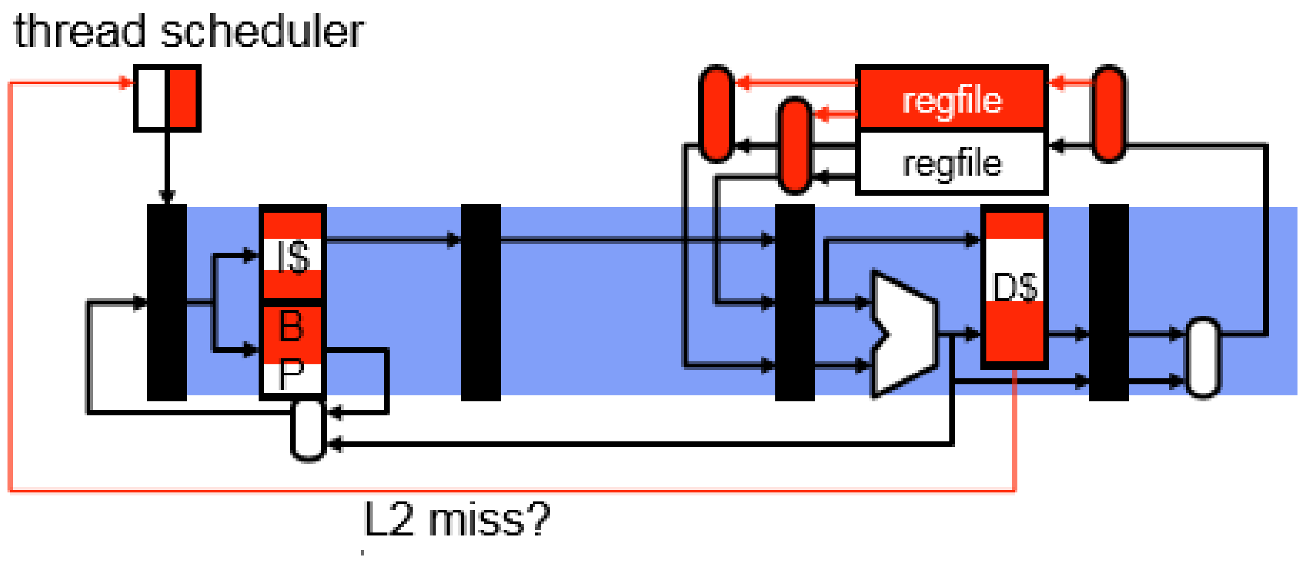
- What is the minimum number of threads to fully utilize the processor now?

- Note that the processor issues instructions in-order in each thread.

粗粒度多线程：Coarse-Grain Multithreading

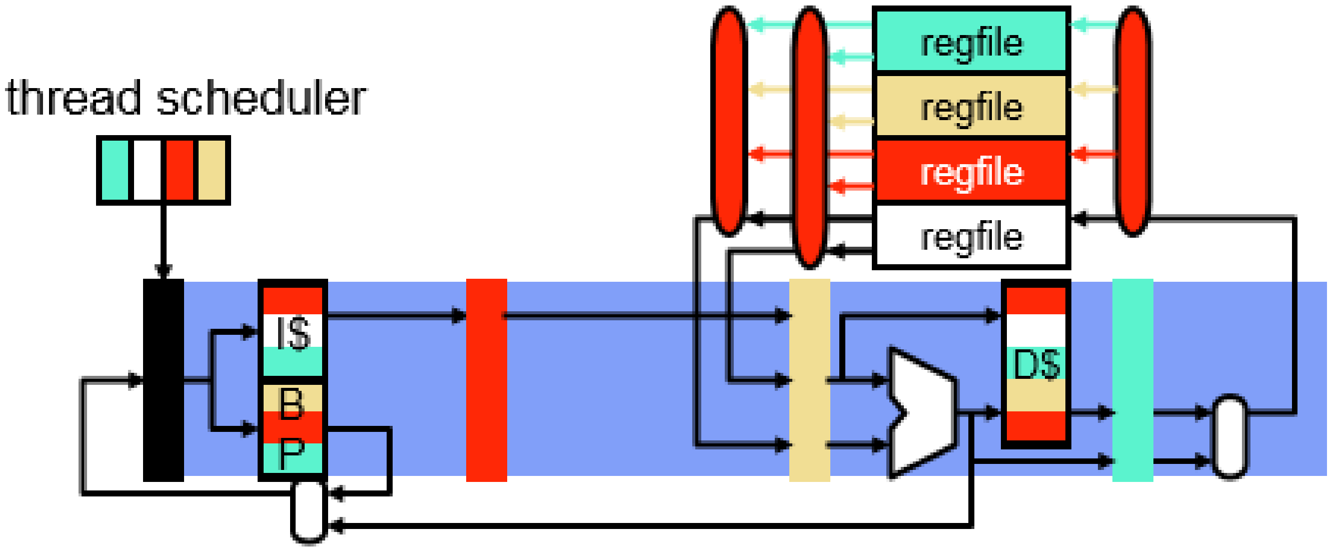


CGMT

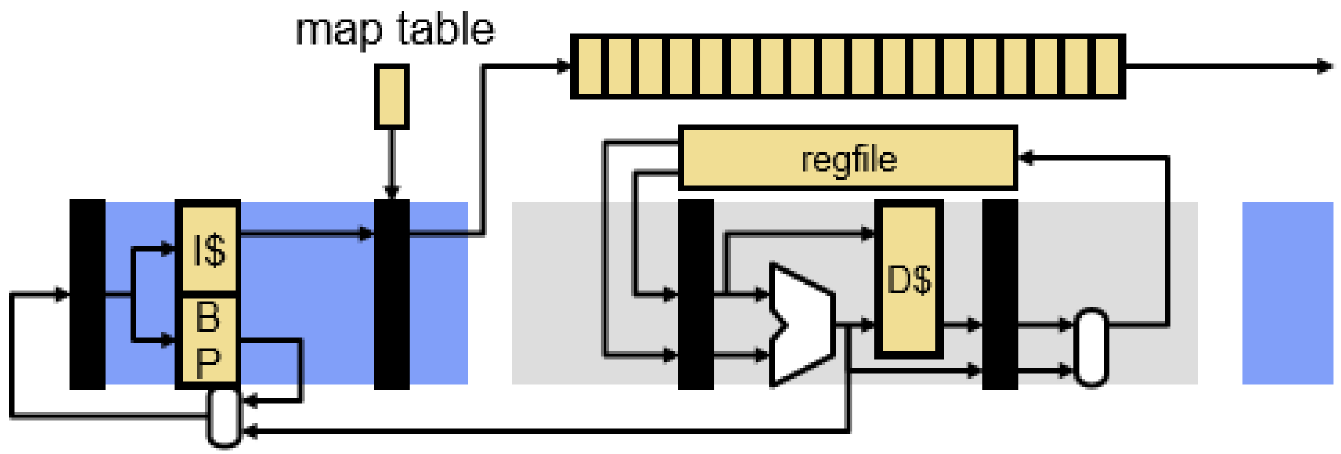


细粒度多线程：Fine-Grain Multithreading

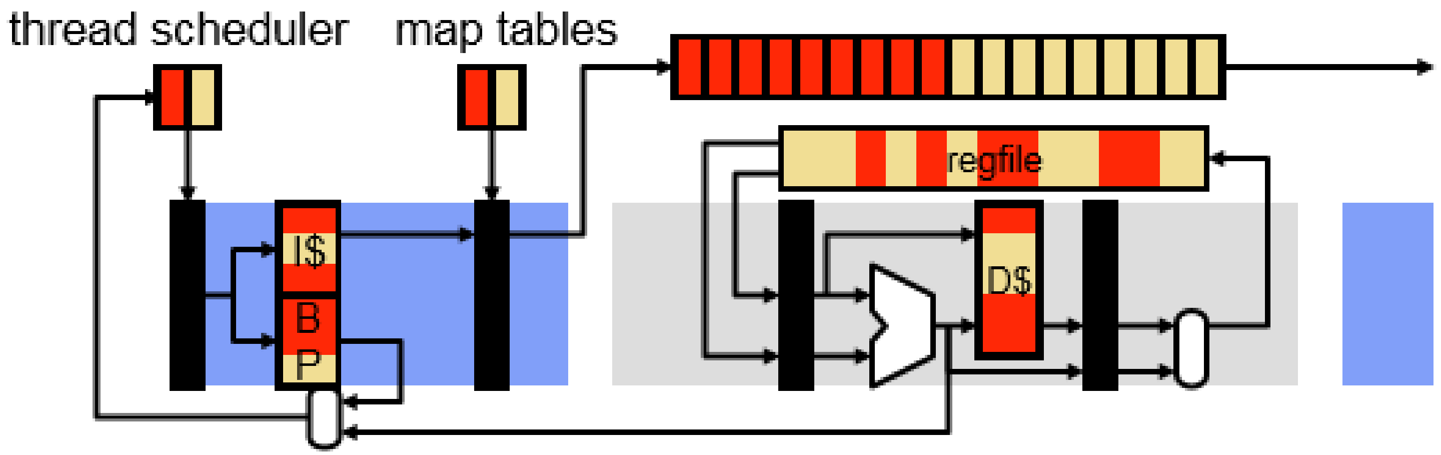
- FGMT
  - **Multiple threads in pipeline at once**
  - (Many) more threads



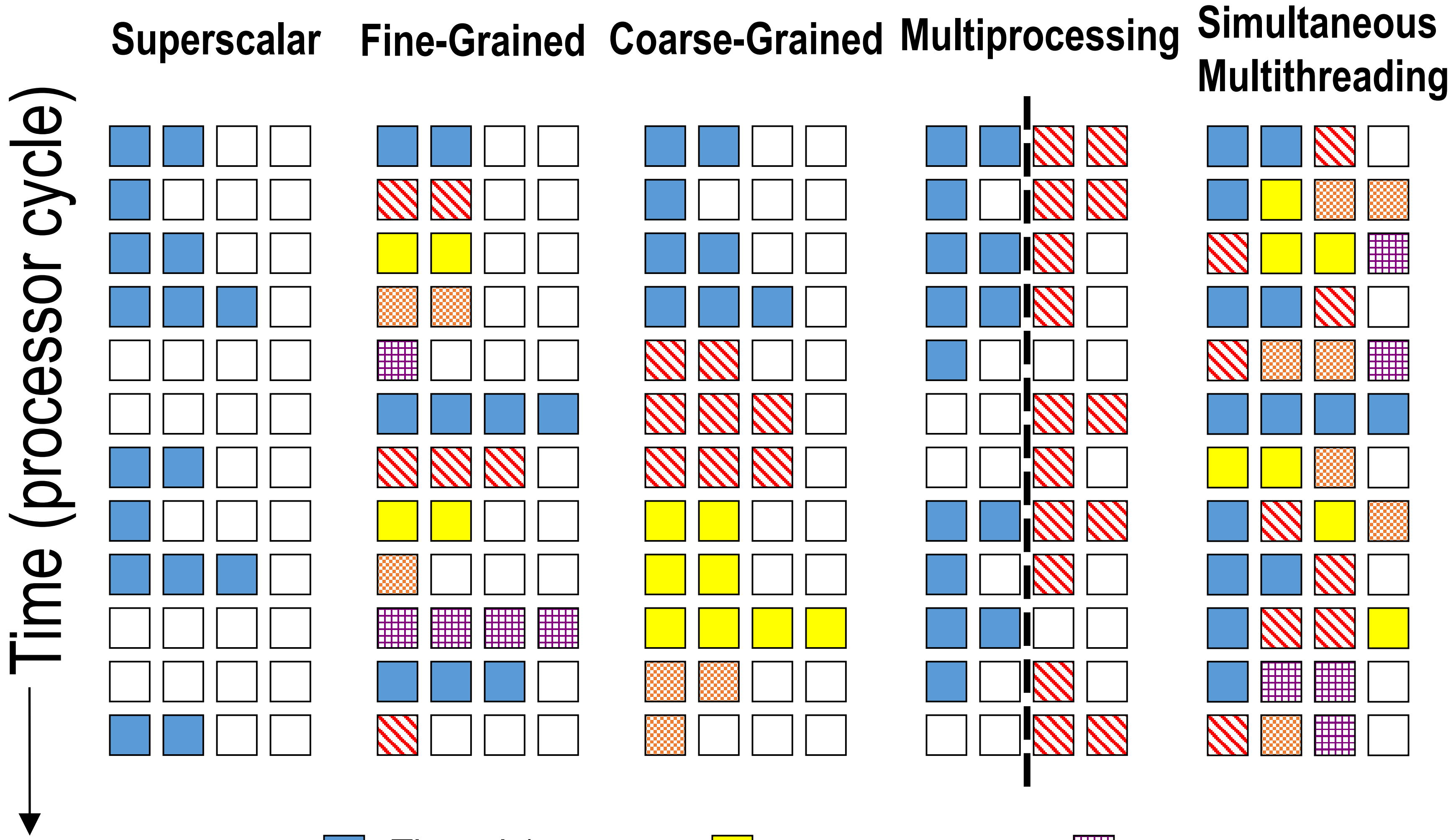
同时多线程：Simultaneous Multithreading



- SMT
  - Replicate map table, share (larger) physical register file



Summary: Multithreaded Categories



🟦	Thread 1	🟨	Thread 3	▦	Thread 5
▧	Thread 2	▩	Thread 4	☐	Idle slot

数据级并行性

Data Level Parallelism

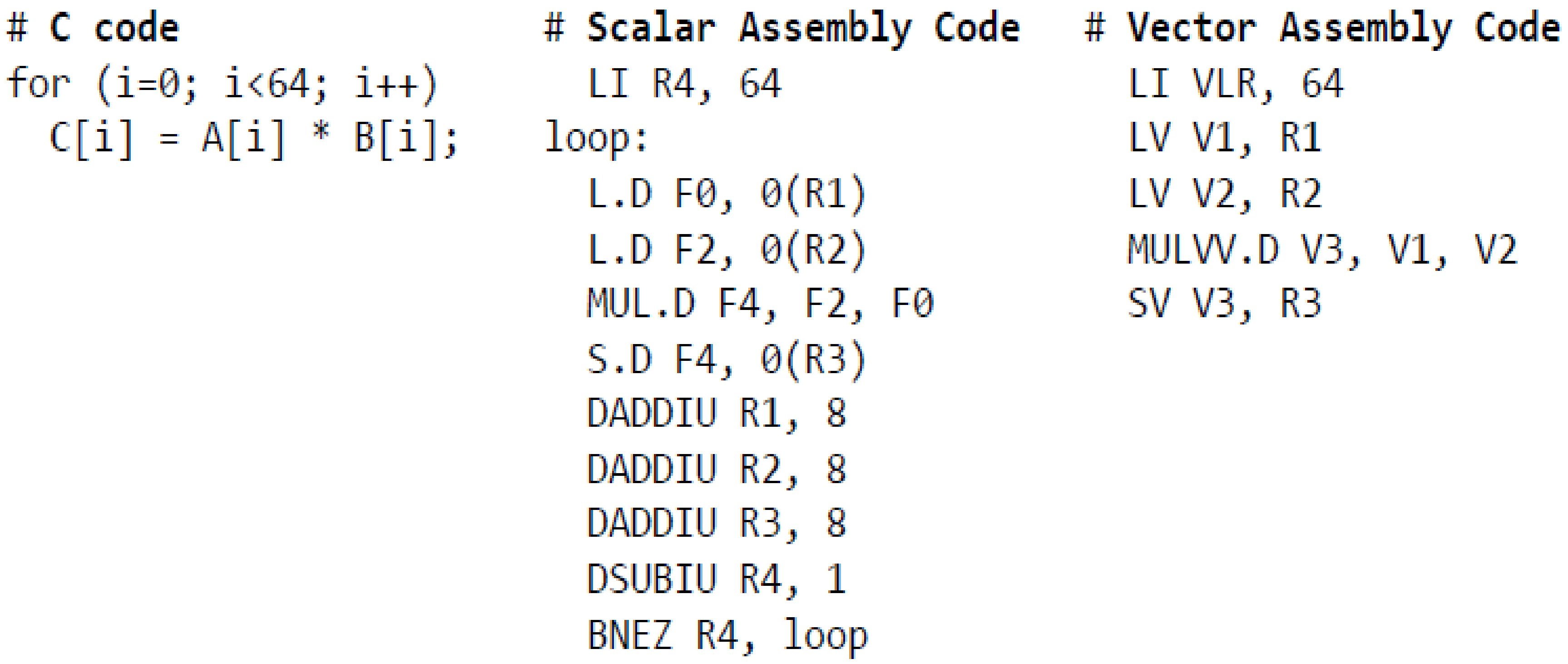
Data-Level Parallelism

- 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

SIMD 结构的种类

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

Vector Code Element-by-Element Multiplication



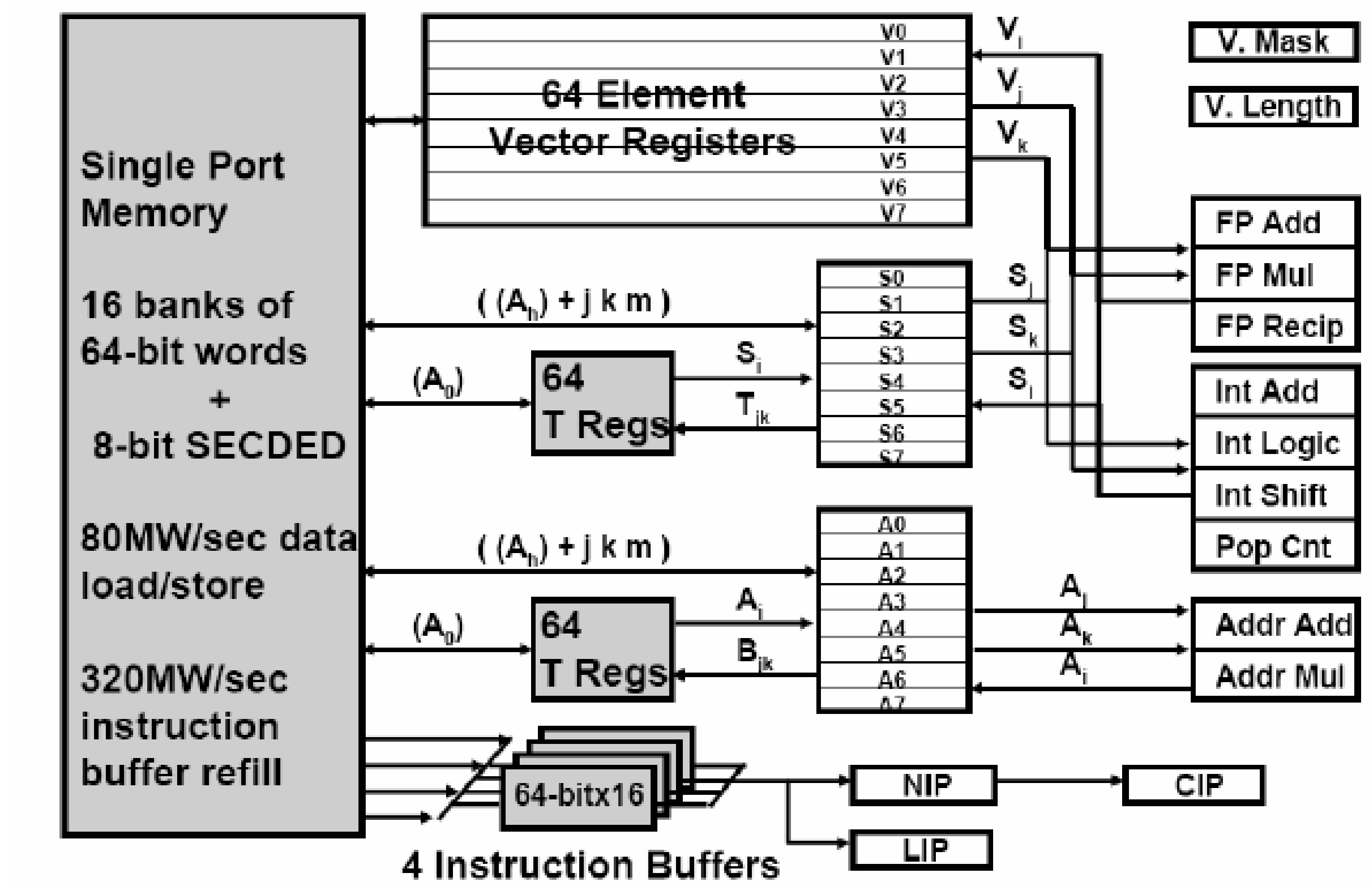
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] * B[i];
```

```
# Scalar Assembly Code
  LI R4, 64
loop:
  L.D F0, 0(R1)
  L.D F2, 0(R2)
  MUL.D F4, F2, F0
  S.D F4, 0(R3)
  DADDIU R1, 8
  DADDIU R2, 8
  DADDIU R3, 8
  DSUBIU R4, 1
  BNEZ R4, loop
```

```
# Vector Assembly Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  MULVV.D V3, V1, V2
  SV V3, R3
```

向量处理机的基本组成单元

Cray－1（1976）



内存体读写：50ns； 处理器周期：12.5ns（80MHz）

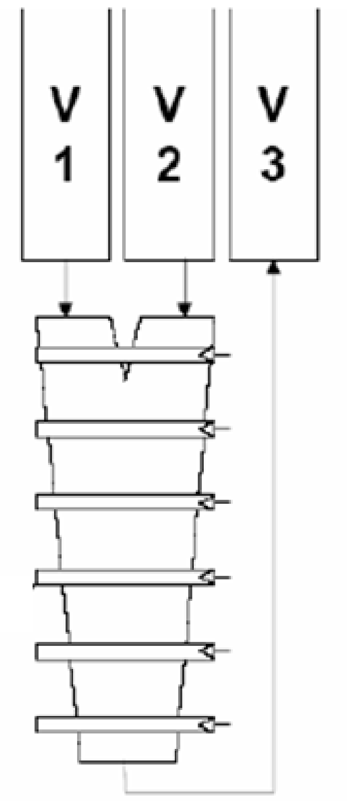
向量体系结构

- 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**
- 向量处理机性能优化
  - 链接技术
  - 条件执行

Vector Arithmetic Execution

- Use deep pipeline (=> fast clock) to execute element operations

- Simplifies control of deep pipeline because elements in vector are independent
  - no data hazards!
  - no bypassing needed

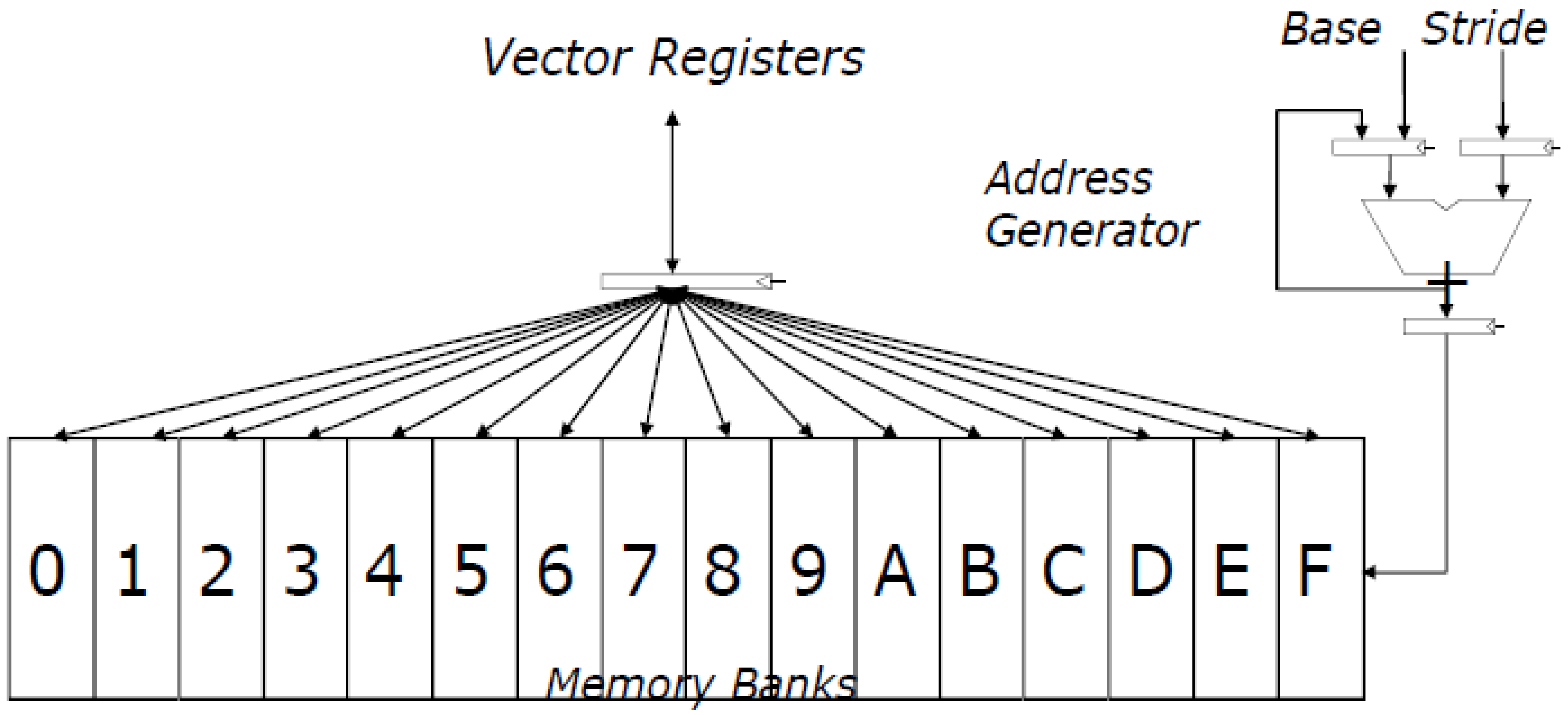


V3 <- v1 * v2

Interleaved Vector Memory System

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

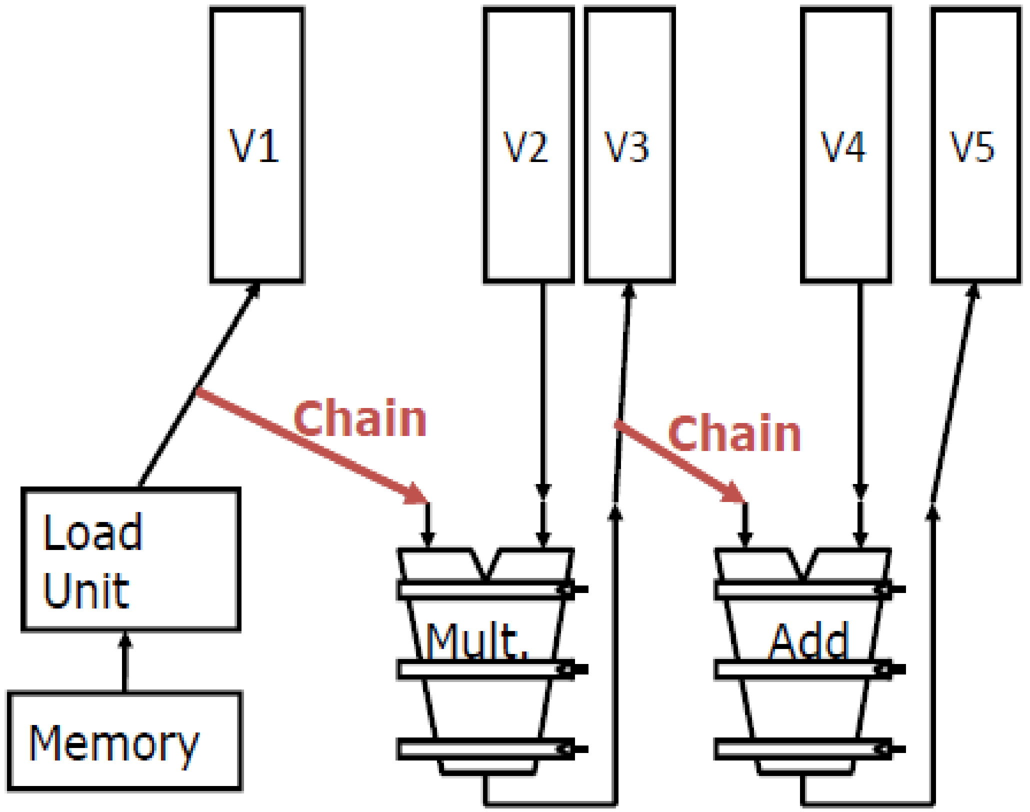
- *Bank busy time*: Time before bank ready to accept next request



Vector Chaining

- Vector version of register bypassing
  - introduced with Cray-1

```
LV     V1
MULVV  V3,V1,v2
ADDVV  V5,V3, v4
```



Vector Conditional Execution

- Problem: Want to vectorize loops with conditional code:

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

- Solution: Add vector mask (or flag) registers
- Code example:

```
CVM              # Turn on all elements
LV VA, RA        # Load entire A vector
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
```

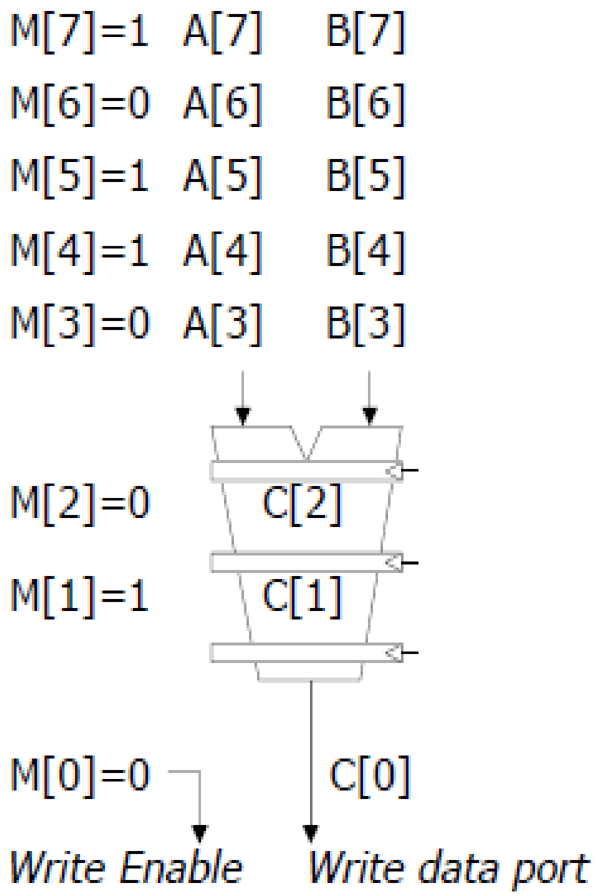
Masked Vector Instructions

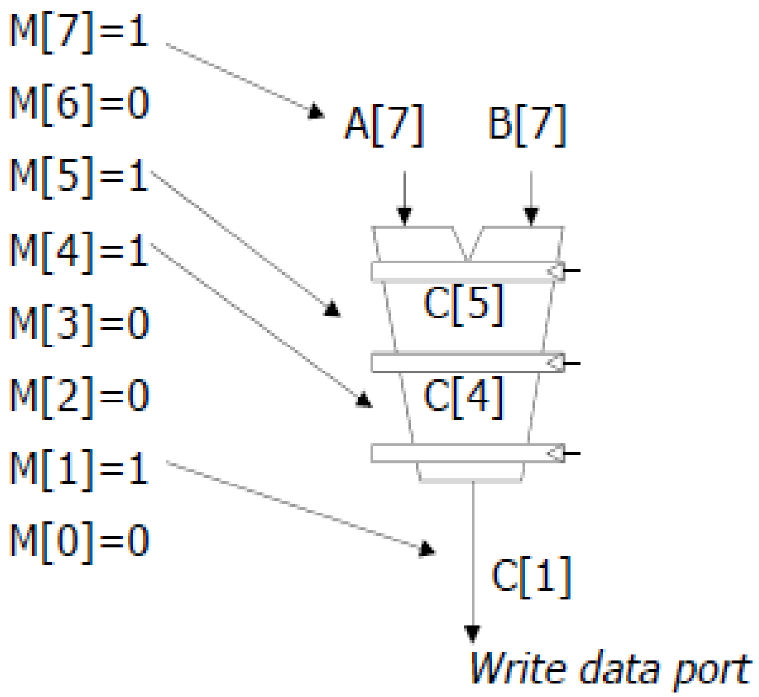
Simple Implementation

- 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



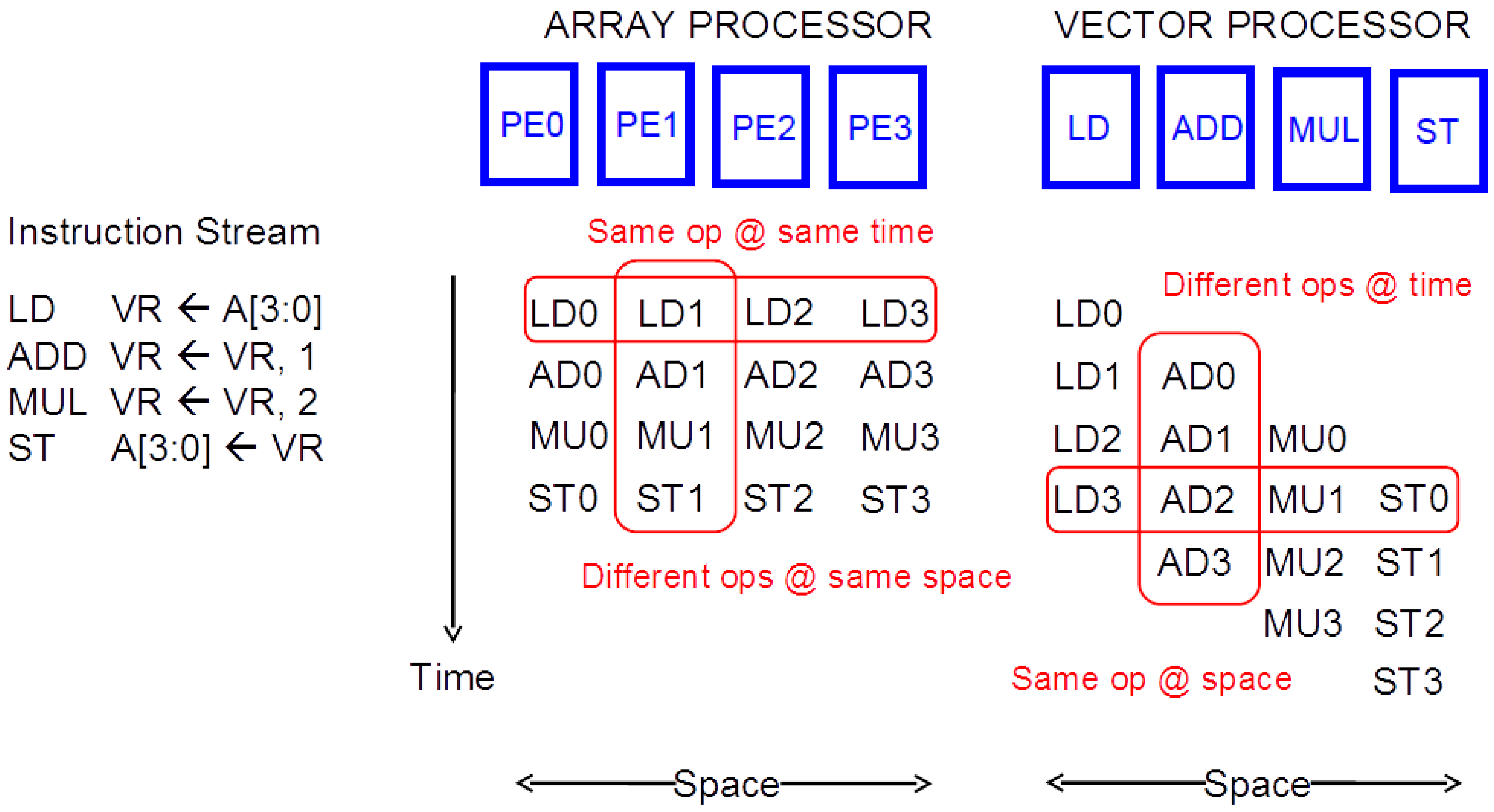


Example Vector Machines

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

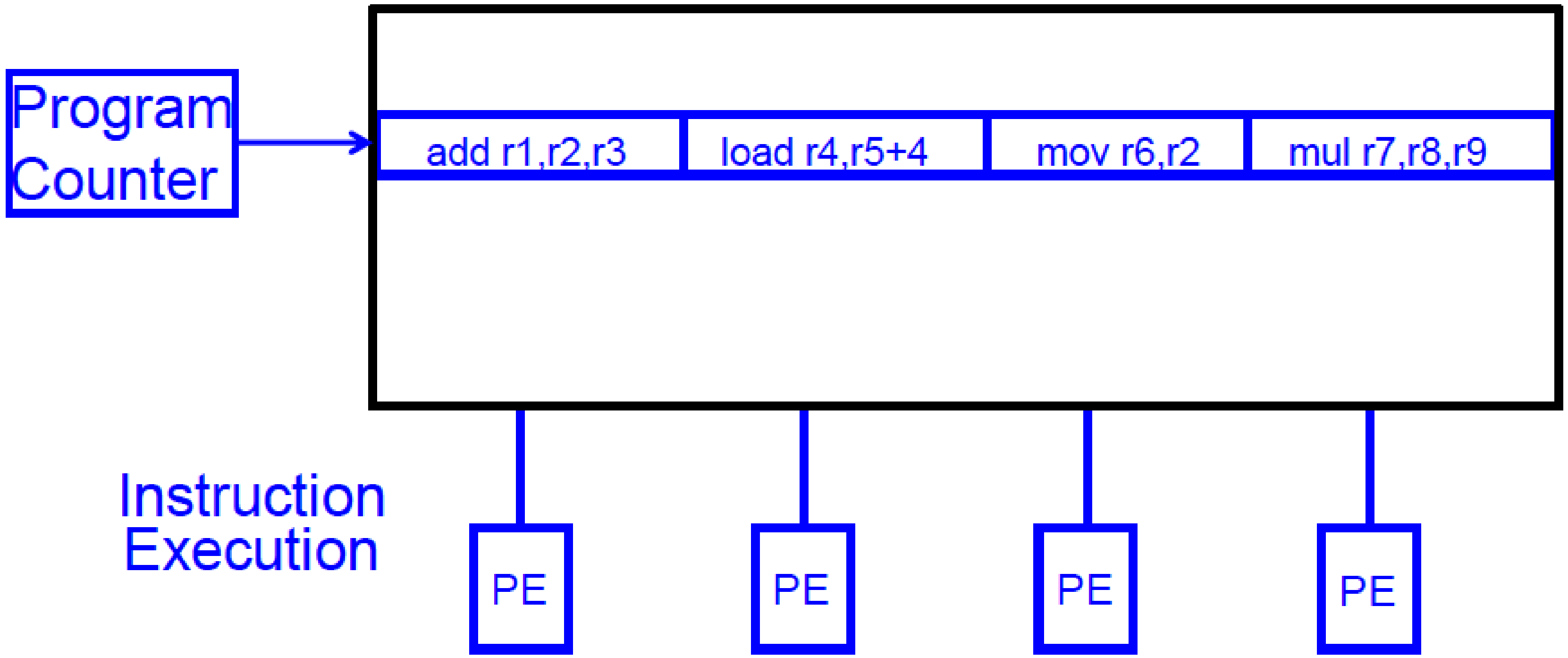
Array vs. Vector Processors

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。



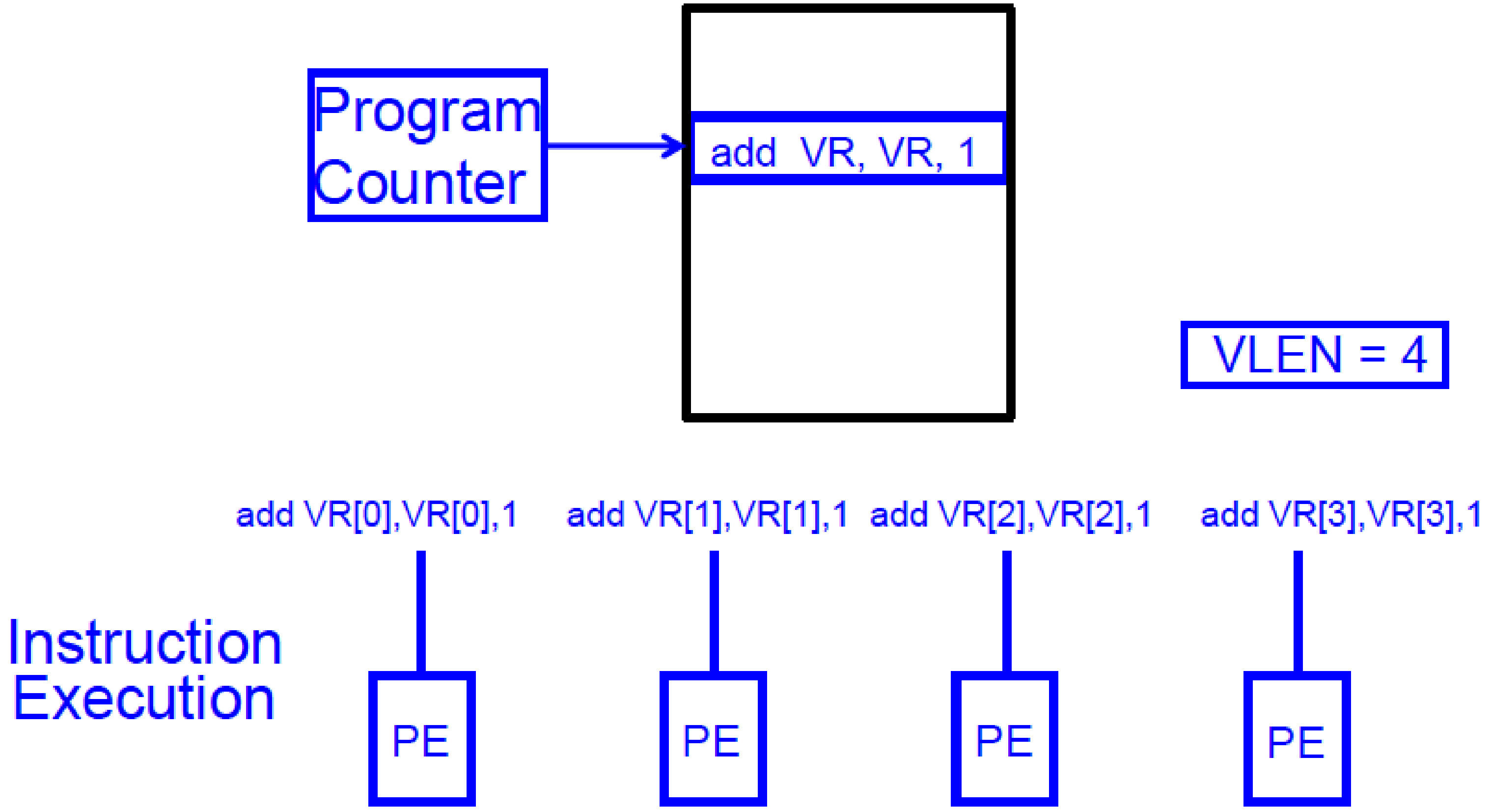
SIMD Array Processing vs. VLIW

- 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



Data-Level Parallelism in SIMD extensions

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

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

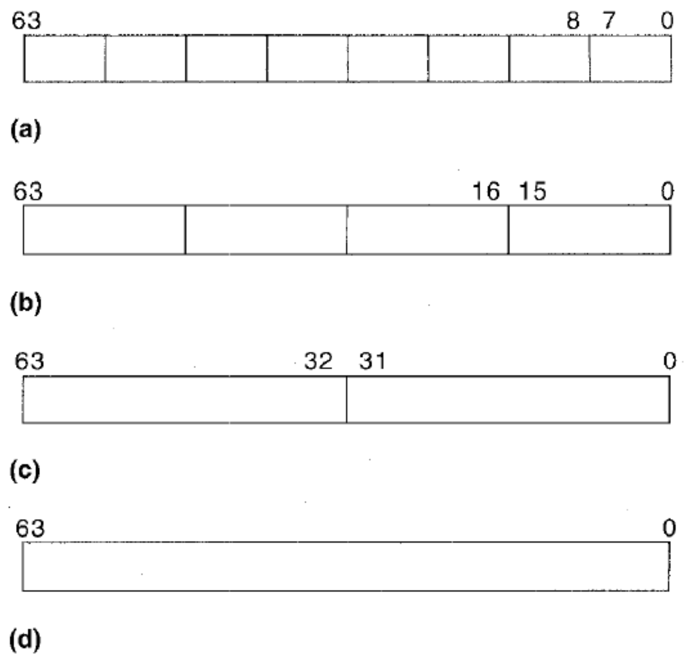


Figure 1. MMX technology data types: packed byte (a), packed word (b), packed doubleword (c), and quadword (d).

- 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.

Example SIMD Code

- Example DXPY:

```
        L.D         F0,a            ;load scalar a
        MOV         F1, F0          ;copy a into F1 for SIMD MUL
        MOV         F2, F0          ;copy a into F2 for SIMD MUL
        MOV         F3, F0          ;copy a into F3 for SIMD MUL
        DADDIU      R4,Rx,#512      ;last address to load
Loop:               L.4D F4,0[Rx]   ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D      F4,F4,F0        ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D        F8,0[Ry]        ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D      F8,F8,F4        ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D        0[Ry],F8        ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU      Rx,Rx,#32       ;increment index to X
        DADDIU      Ry,Ry,#32       ;increment index to Y
        DSUBU       R20,R4,Rx       ;compute bound
        BNEZ        R20,Loop        ;check if done
```

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)

下一节

- 数据级并行性
  - GPU

From : H&P Computer Architecture: A Quantitative Approach,
Fifth Edition, （5th edition)

谢　谢！