Computer Architecture

Lecture 10 – Vector Machine (Data Level Parallel)

Tian Xia
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

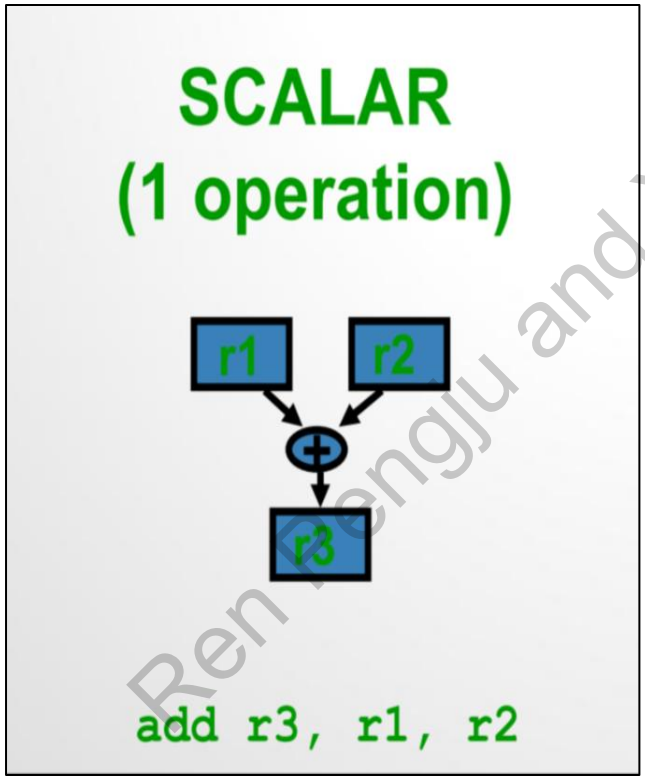
SISD、MIMD、SIMD and MIMD (Flynn's Taxonomy)

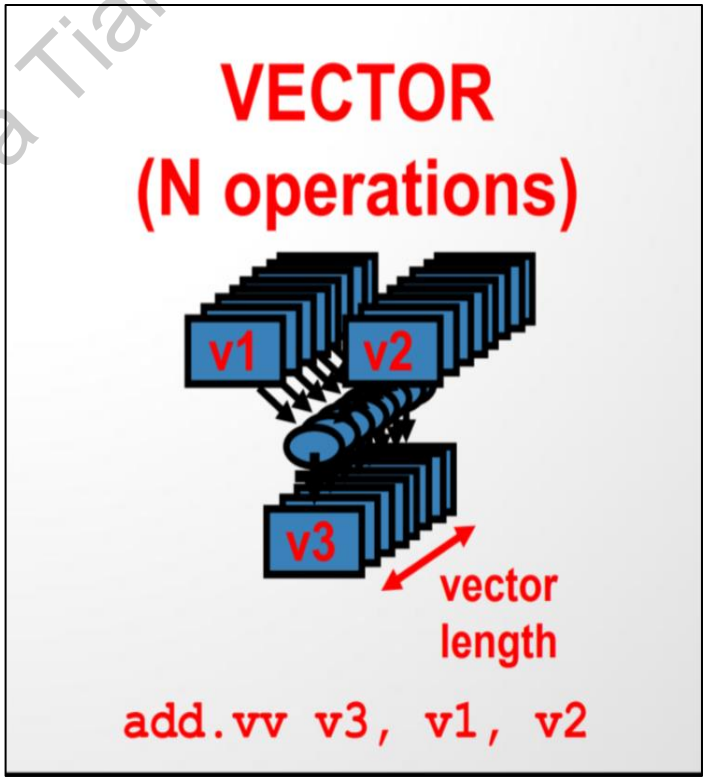
		Data Streams	
		Single	**Multiple**
Instruction Streams	**Single**	**SISD: Intel Pentium 4**	**SIMD: SSE of x86**
	Multiple	**MISD: *No example today***	**MIMD: Intel Core i7**

SISD: **S**ingle **I**nstruction stream, **S**ingle **D**ata Stream
MIMD: **M**ultiple **I**nstruction streams, **M**ultiple **D**ata Streams
SIMD: **S**ingle **I**nstruction stream, **M**ultiple **D**ata Streams
MISD: **M**ultiple **I**nstruction streams, **S**ingle **D**ata Stream

Agenda

- Vector Processors
- Single Instruction Multiple Data (SIMD)
- Instruction Set Extensions（Neon, SVE@ARM, AVX@Intel, etc.)





Modern SIMD Processors

SIMD architectures can exploit significant data-level parallelism for:

- Matrix-oriented scientific computing
- Media-oriented image and sound processors
- Machine Learning Algorithms

Most modern CPUs have SIMD architectures

- Intel SSE and MMX, AVX, AVX2 (Streaming SIMD Extension, Multimedia extensions、 Advanced Vector extensions)
- ARM NEON, MIPS MDMX

These architectures include ***instruction set extensions*** which allow both sequential and parallel instructions to be executed

Some architectures include separate SIMD coprocessors for handling these instructions

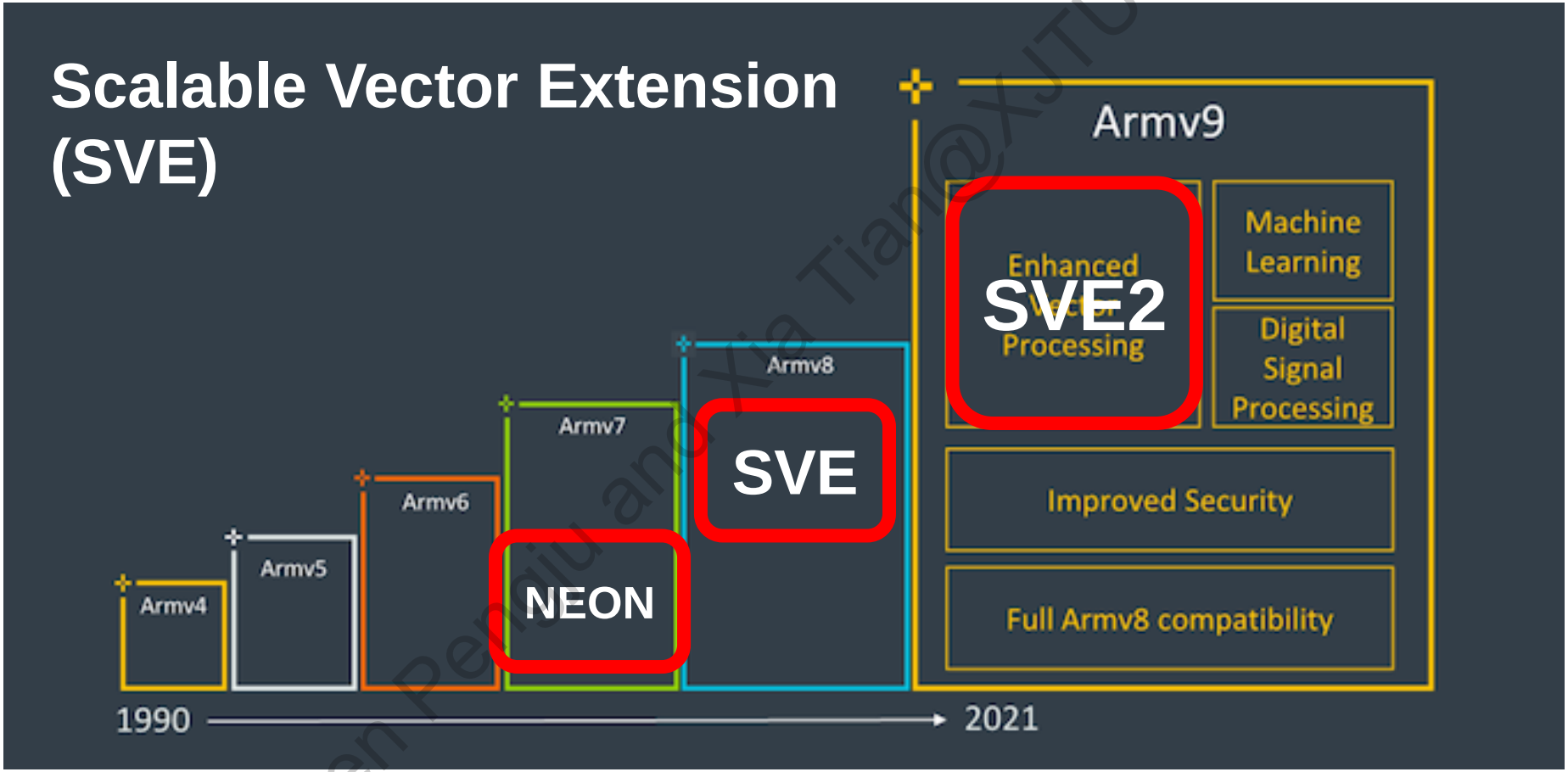
ARM NEON

- Included in Cortex-A8 and Cortex-A9 processors

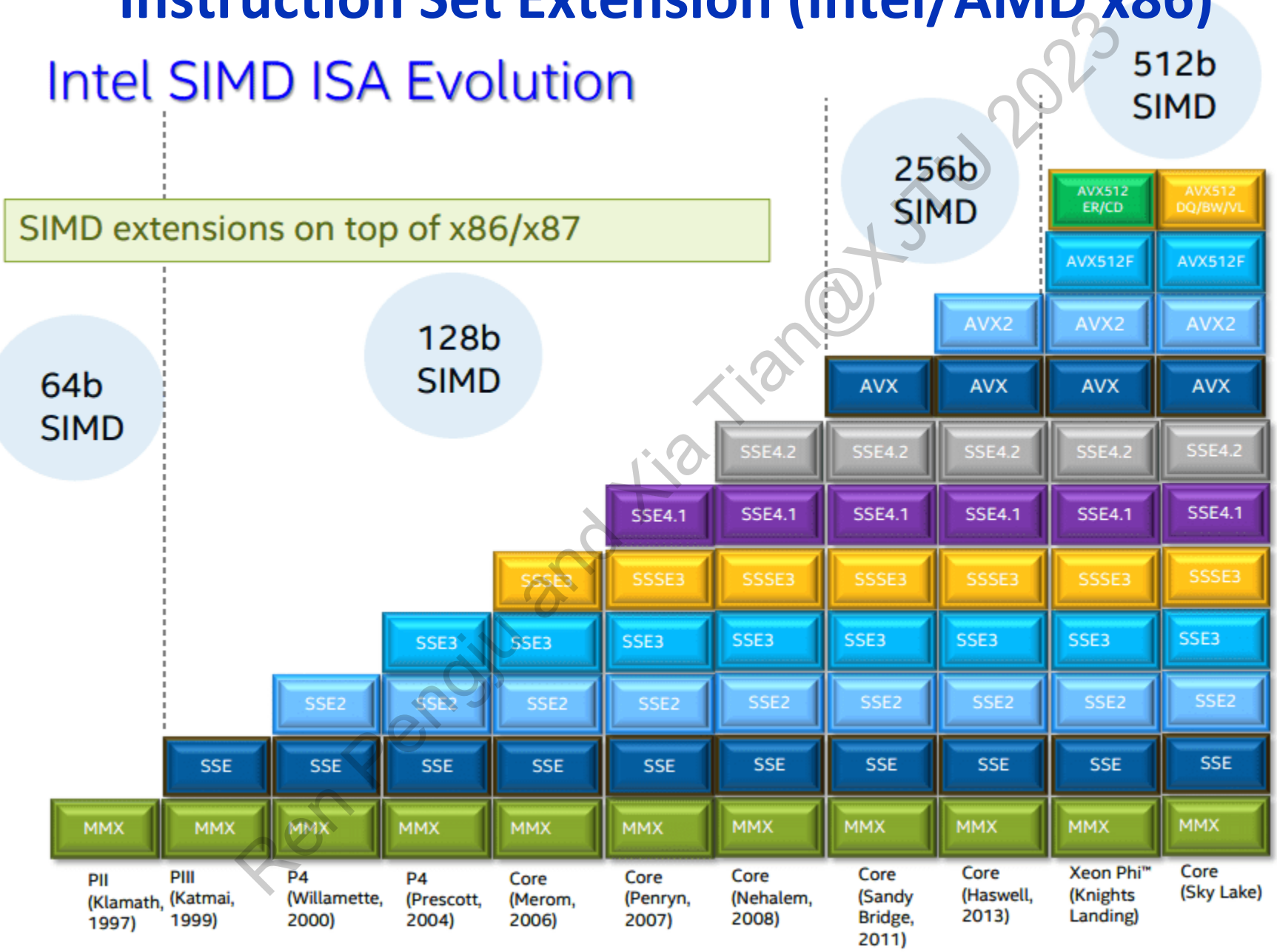
Intel SSE and AVX

- Introduced in 1999 in the Pentium III processor
- AVX512 currently used in Xeon Core series

Instruction Set Extension (ARM)



Instruction Set Extension (Intel/AMD x86)

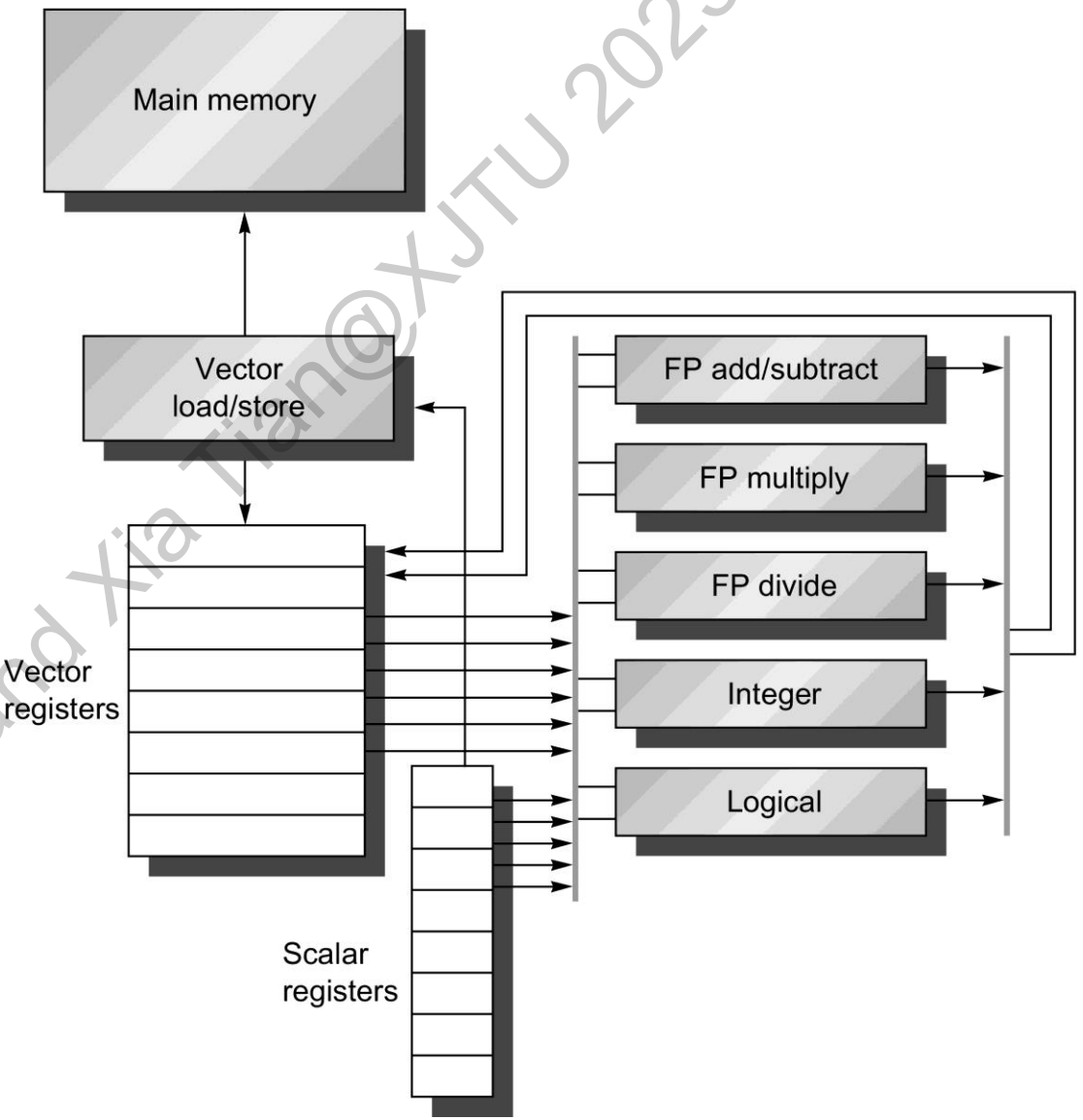


Vector Processor

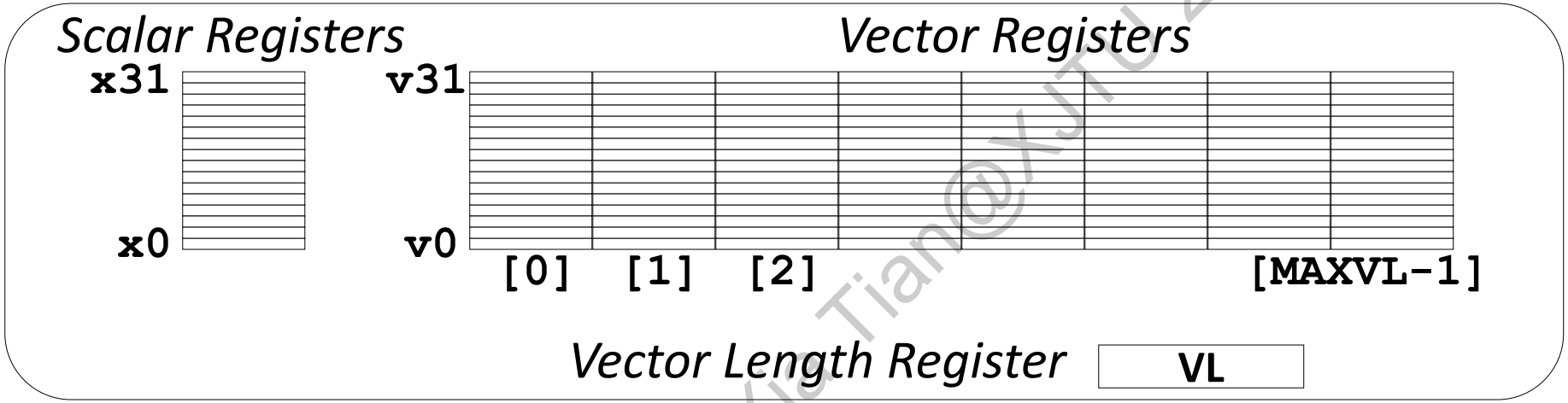
- Basic idea:
  - **Load** sets of data elements into "vector registers"
  - **Operate** on those registers
  - **Disperse** the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth
- Overcoming limitations of ILP:
  - Dramatic reduction in **fetch and decode** bandwidth.
  - No **data hazard** between elements of the same vector. Data hazard logic is required only between two vector instructions.
  - Heavily **interleaved memory** banks. Hence latency of initiating memory access versus cache access is amortized.
  - Since loops are reduced to vector instructions, there are no **control hazards**.
  - Good performance for **poor locality**

RV64V Extension (RISC-V Vector Extension)

- **Vector Register：** 32x64bit (16 read and 8 write ports)
- **Vector Functional Units：** Each unit is fully Pipelined
- **Vector Load/Store Unit**
- **Scalar register**: normal 31 general-purpose registers



Vector Programming Model (RISC-V)



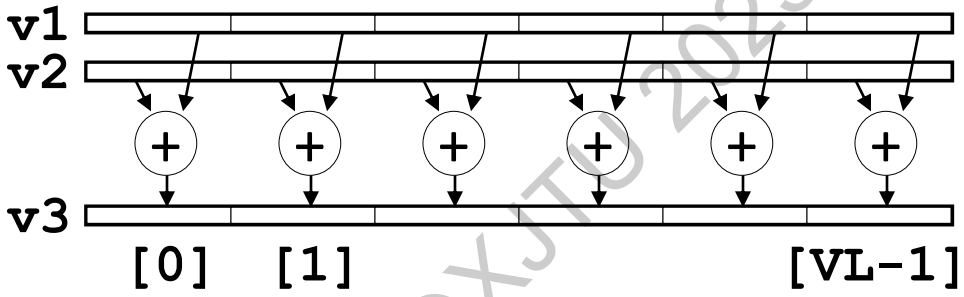
Dynamic data type: If a vector register has 2048-bit width, then it can hold:

- 128 16-bit elements (e.g. 128 **Int16** numbers)
- 32 64-bits elements (e.g. 32 **Double-Float** numbers)
- ……

Vector Programming Model (RISC-V)

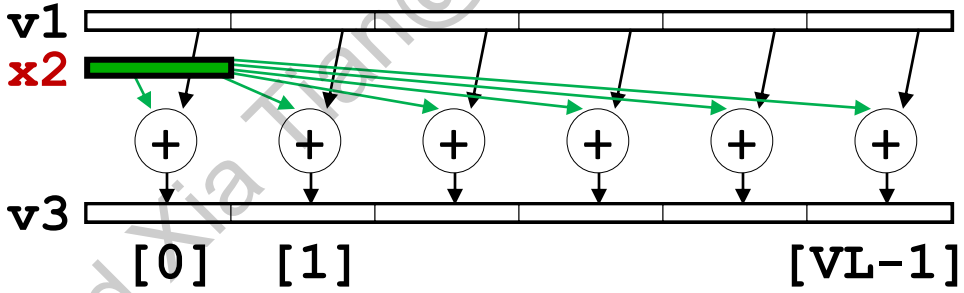
Vector Arithmetic Instructions

`vadd(.i).vv v3,v1,v2`



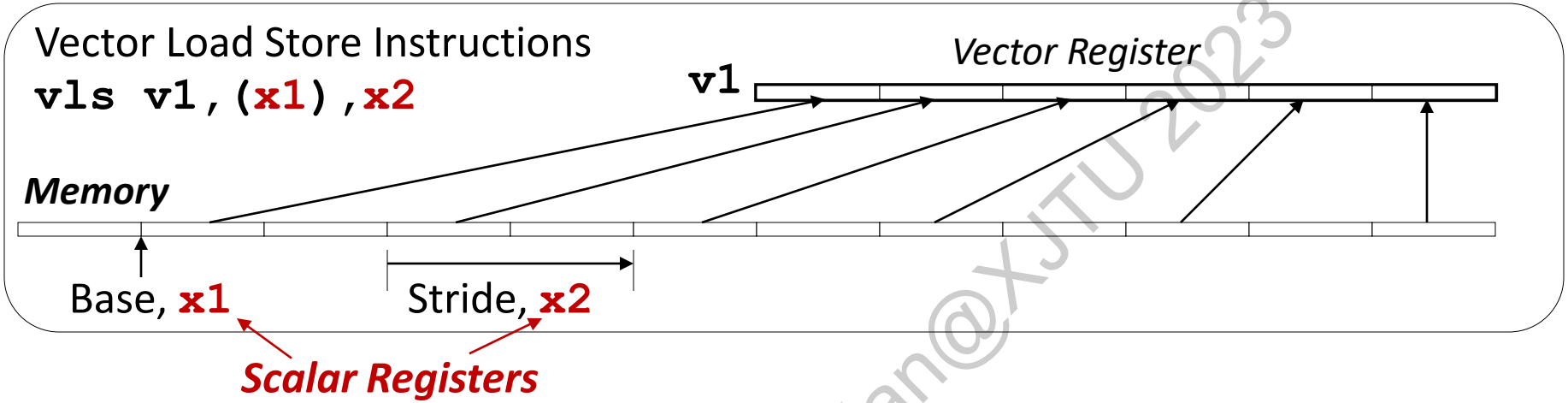
Vector Arithmetic Instructions

`vadd(.i).vs v3,v1,x2`

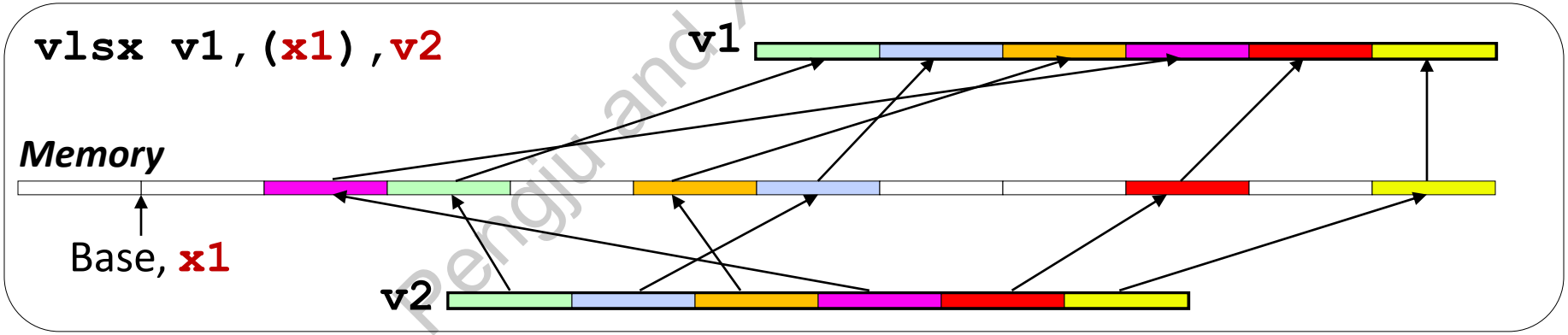


- Vector Arithmetic Instructions can use both **vector and scalar registers**
- They are followed with **Suffix**:
  - **.vv** = both operand are vector
  - **.vs** = second operand is a scalar
  - **.sv** = first operand is a scalar register.

Vector Programming Model (RISC-V)



- Access a **contiguous** block of memory (Continuous load/store)
- Access memory in a **fixed stride** pattern (Strided load/store)



- Access a group of **arbitrary addresses** in memory
- **Gather** (load) and **Scatter** (store)

Vector Code Example

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

```
# Scalar Code
  li x4, 64
  li x6, a
loop:
  fld f1, 0(x1)
  fld f2, 0(x2)
  fmul.d f3,f1,x6
  fadd.d f4,f1,f2
  fsd f4, 0(x3)
  addi x1, 8
  addi x2, 8
  addi x3, 8
  subi x4, 1
  bnez x4, loop
```

```
# Vector Code
  li x4, 64
  li x6, a
  setvl x4
  vld v1, x1
  vld v2, x2
  vmul.d.vs v3,v1,x6
  vadd.d.vv v4,v3,v2
  vst v4, x3
```

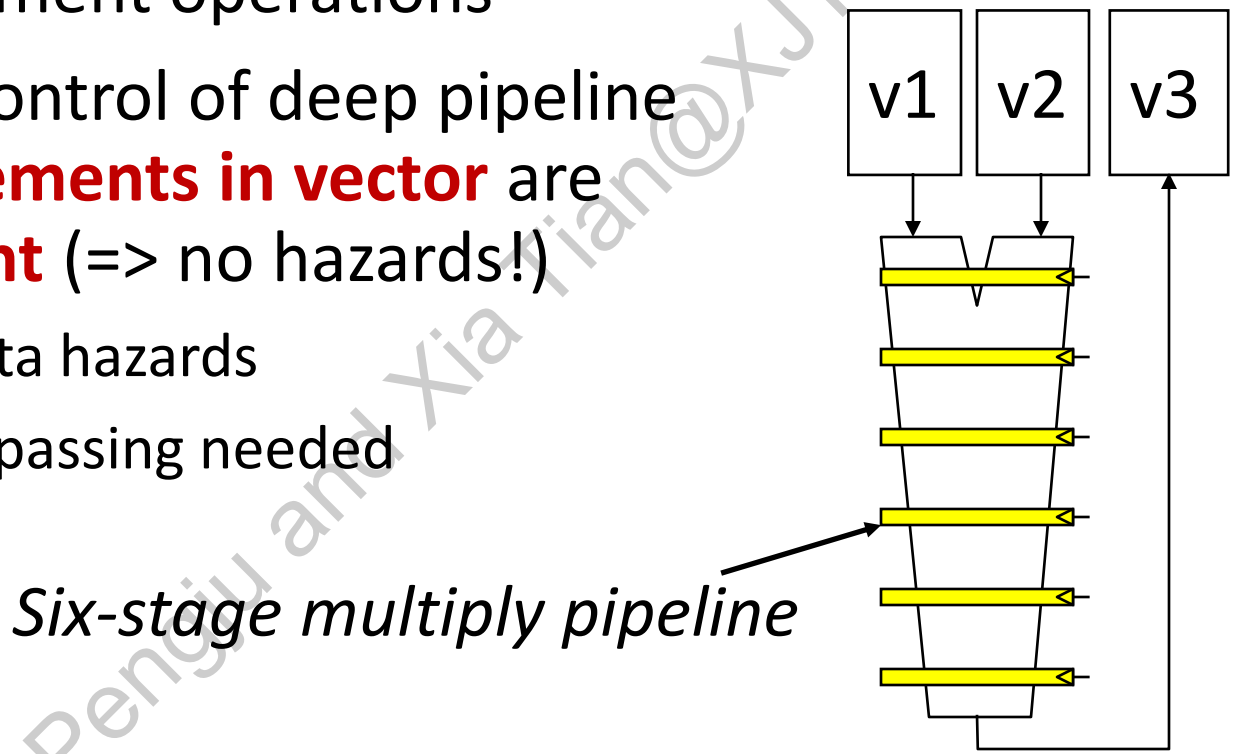

- **Less code lines**: 640+ Instructions → 8 Instructions
- **Explicit independency**: less dependency checks
- **Programming-friendly**: maintain classical code styles.

Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)
- Scalable
  - can run same code on more parallel pipelines (lanes)

Vector Arithmetic Execution

- Use **deep pipeline** (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because **elements in vector** are **independent** (=> no hazards!)
  - No data hazards
  - No bypassing needed

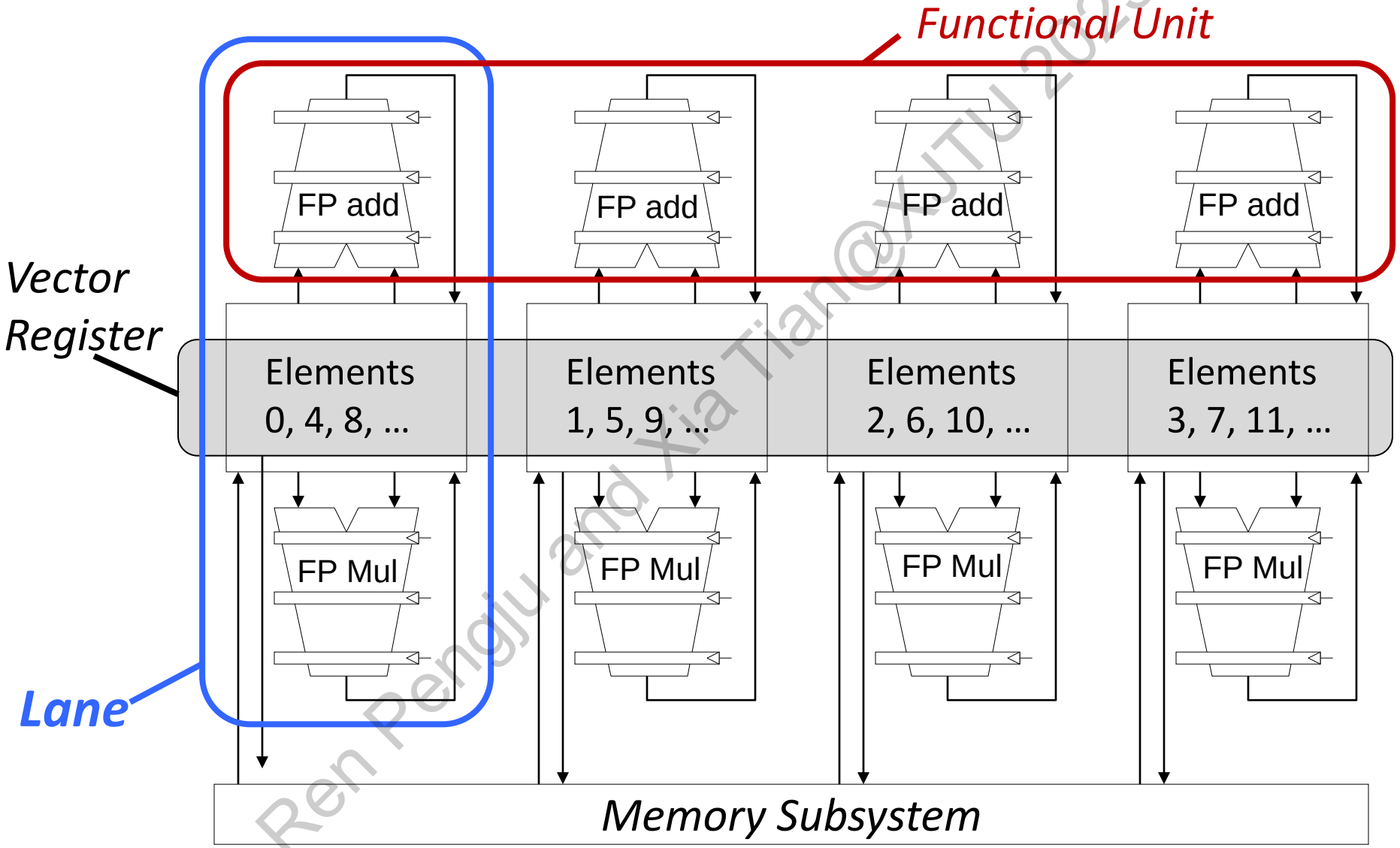


Vector Processor Optimization

How can a vector processor execute a single vector faster than one element per clock cycle ?

- **Multiple Lanes: beyond one element/cycle**

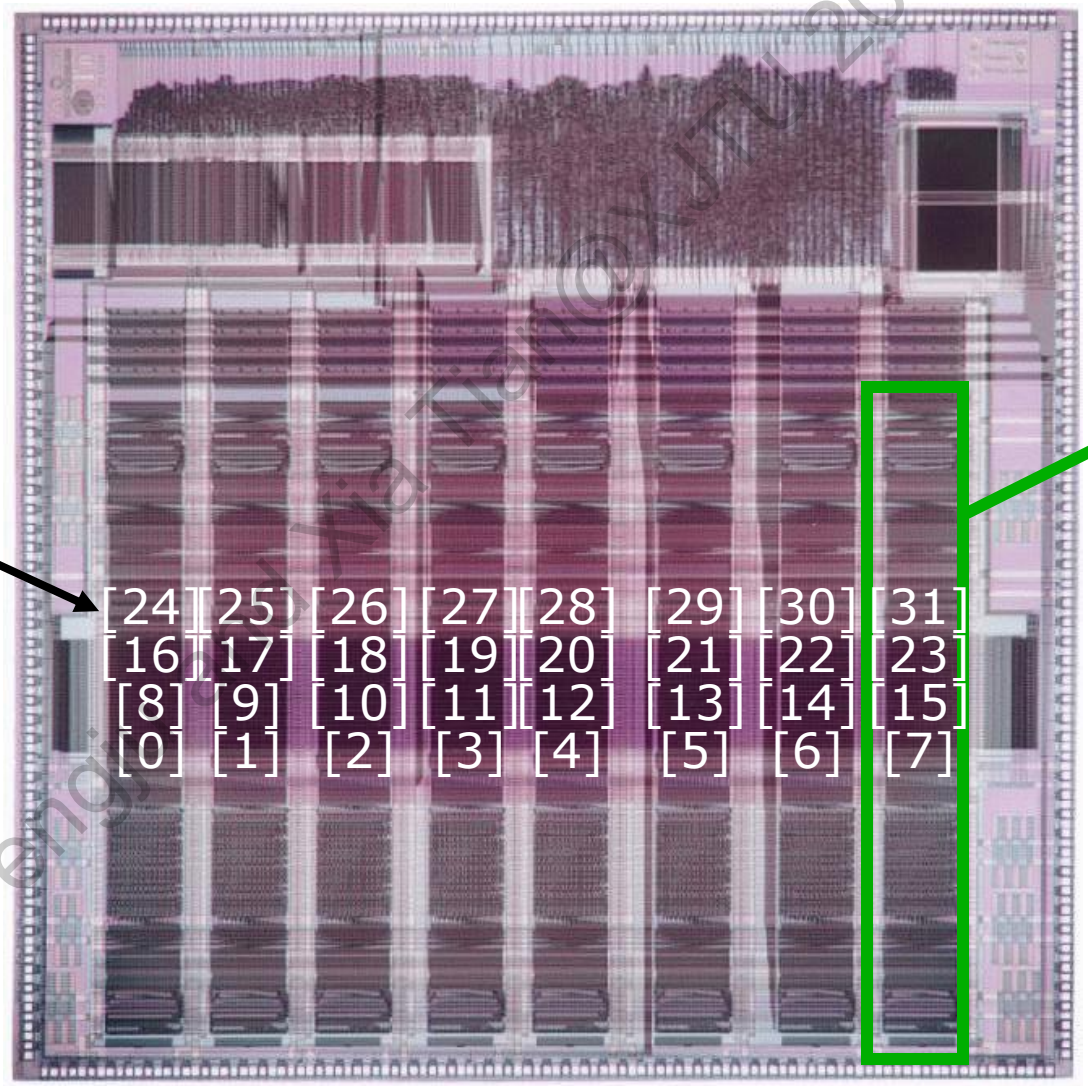
Vector Unit Structure- Multiple Lanes



The same element position in the input and output registers is referred to as a lane.

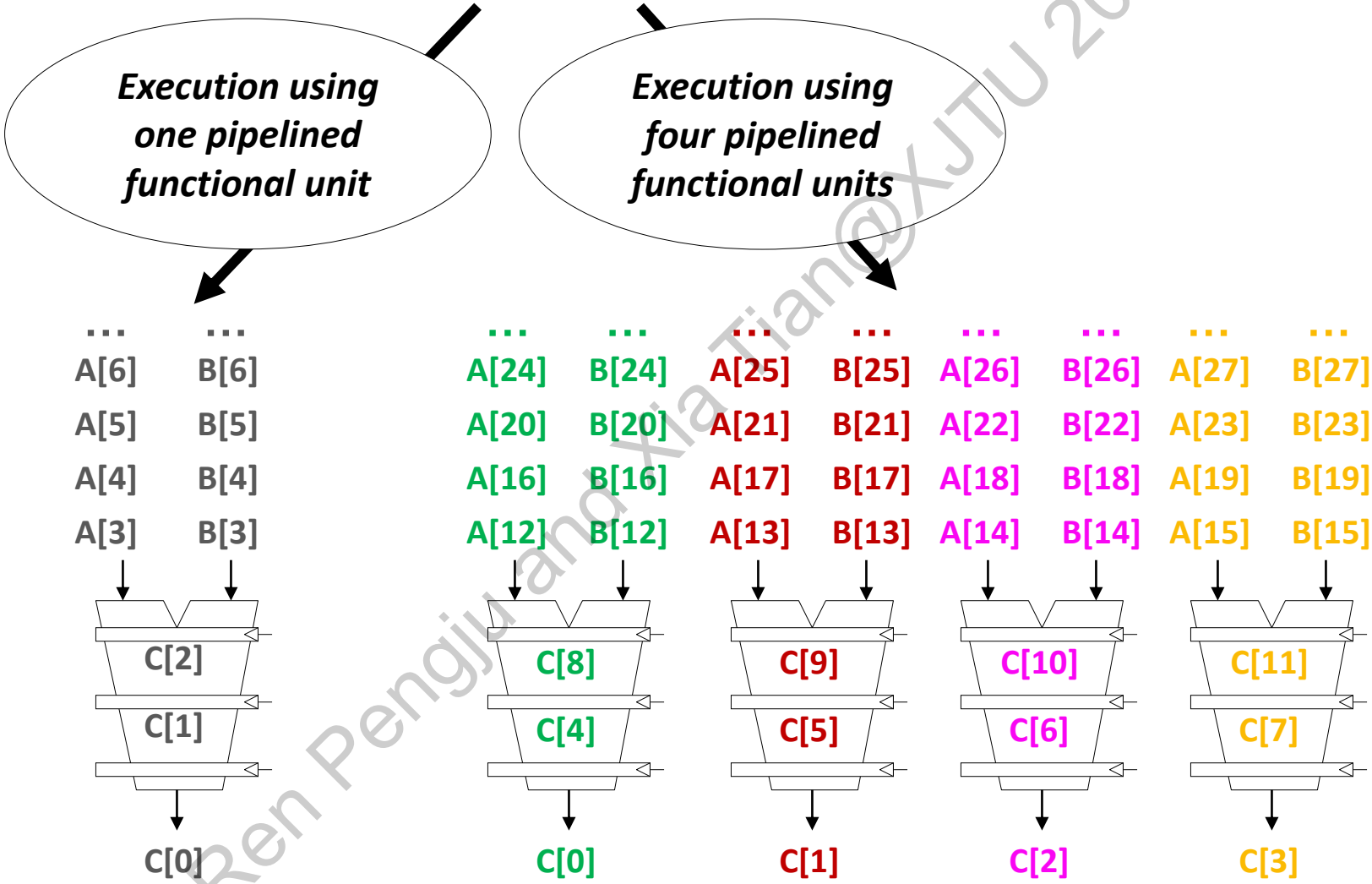
There cannot be a carry or overflow from one lane to another

T0 Vector Microprocessor (UCB/ICSI, 1995)



Vector Instruction Execution

`vmul vc, va, vb` (Vector length=32)



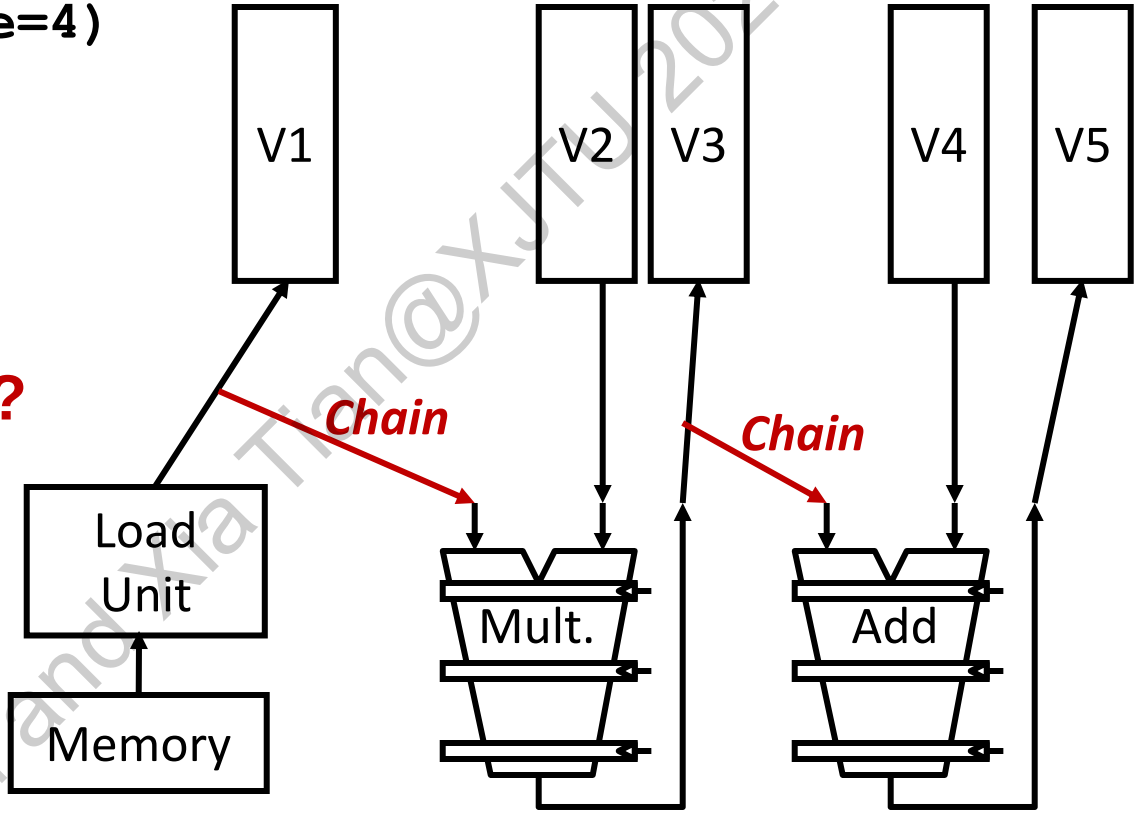
Latency = 32 +2 cycles

Latency = 32/4 +2 = 10 cycles

Vector Chaining

```
(Vector length=32, Lane=4)
vld  v1
vmul v3,v1,v2
vadd v5,v3,v4
```

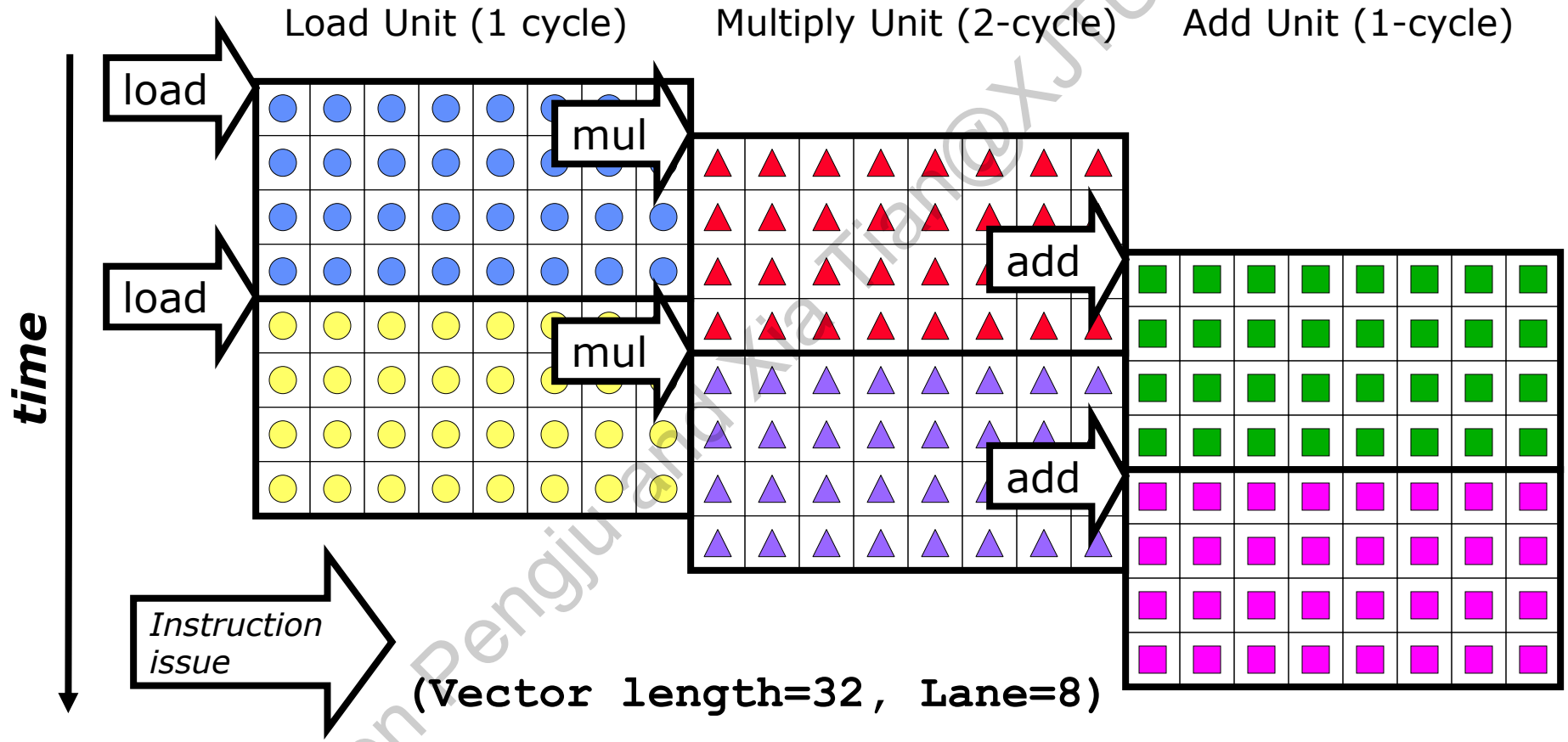
Have to wait 10 cycles ?



- Vector version of register bypassing
  - Chaining allows vector operation to start as soon as the **individual elements** of its vector source operand become available
- With Vector Chaining, vadd waits for **2 cycles**

Vector Instruction Parallelism

- Can overlap execution of multiple vector instructions
  - example machine has **32 elements** per vector register and **8 lanes**



- Complete **24 operations/cycle**
- Issuing **3 vector instruction/4 cycles**

Vector Chaining Advantage

- Without chaining, must **wait for last element of result** to be written before starting dependent instruction

- With chaining, can start dependent instruction **as soon as first result appears**

Vector Processor Optimization

How can a vector processor execute a single vector faster than one element per clock cycle ?

- **Multiple Lanes: beyond one element/cycle**

How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length ?

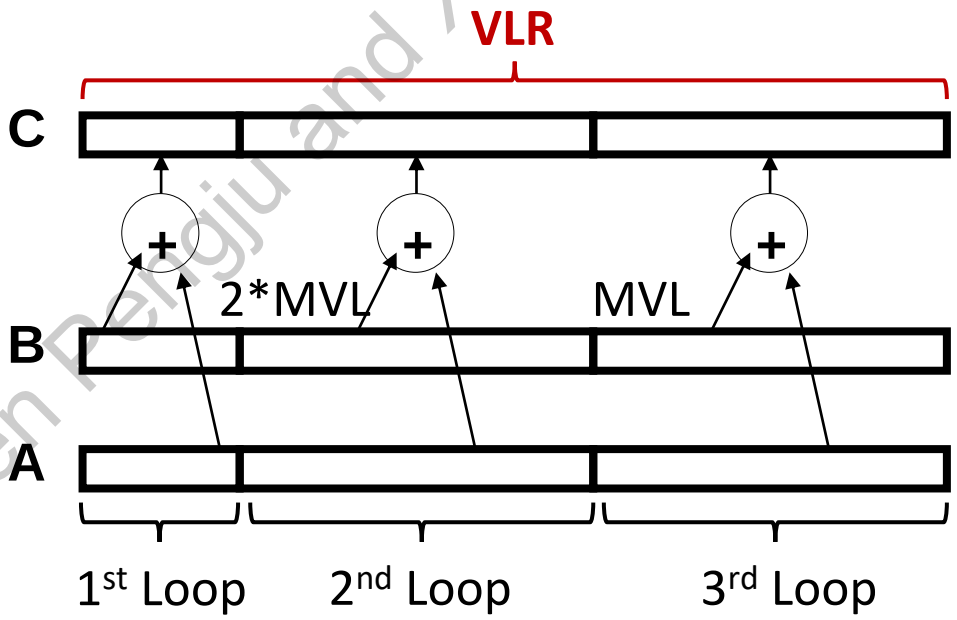
- **Vector-length Registers: Handling loops not equal to MVL (Strip Mining)**

Vector Strip mining

Problem: What happens if the length is not matching the length of the vector registers?

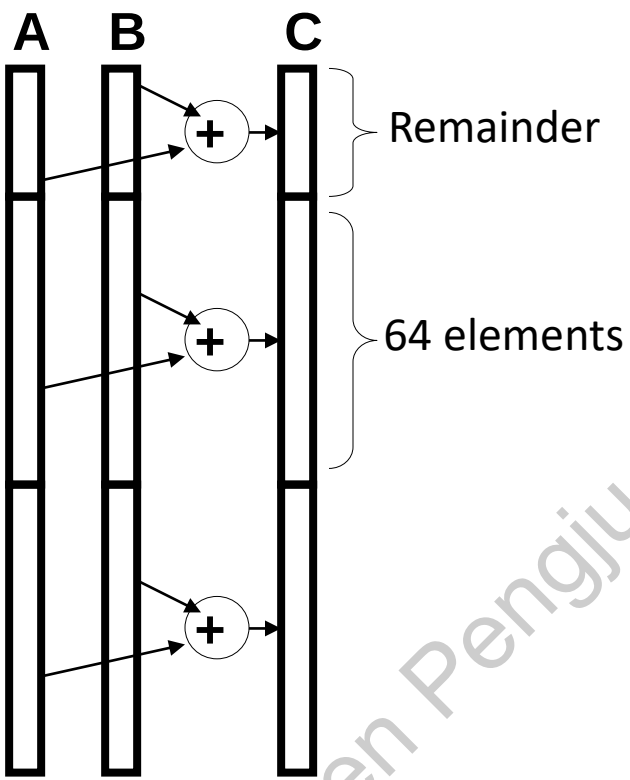
A **vector-length register (VLR)** contains the number of elements used within a vector

Solution: *"Strip mining" split a large loop into loops less or equal the maximum vector length (MVL)*



Vector Strip mining: Example 1

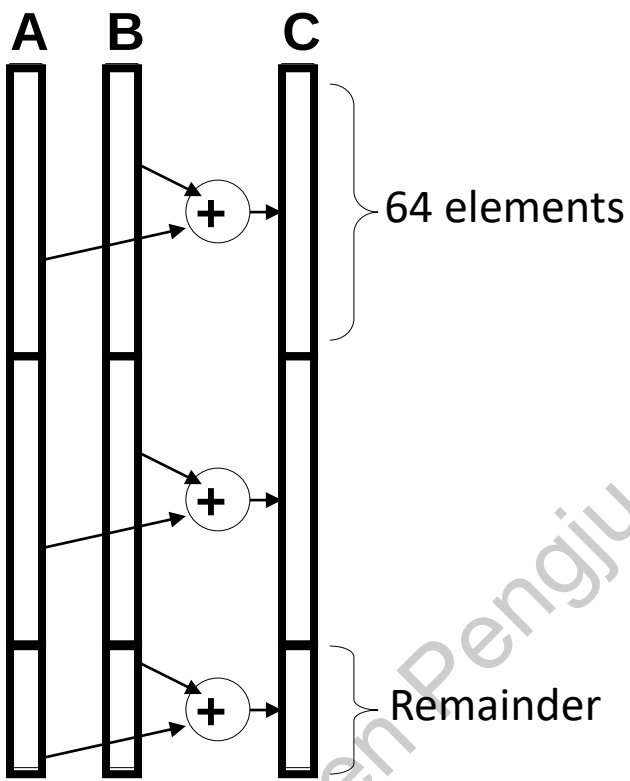
```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```




```
  andi x1,xN,63 # N mod 64
  setvl x1      # Do remainder
loop:
  vld v1,(xA)
  sll x2,x1,3   # Multiply by 8
  add xA,x2     # Bump A pointer
  vld v2,(xB)
  add xB,x2     # Bump B pointer
  vadd v3,v1,v2
  vst v3,(xC)
  add xC,x2     # Bump C pointer
  sub xN,x1     # Subtract elements
  li x1,64
  setvl x1      # Reset full length
  bgtz xN,loop  # Continue if xN>0
```

Vector Strip mining: Example 2

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




```
loop:
  setvl xN,64     # vl=min(xN,64)
  vld v1,(xA)
  sll x2,x1,3     # Multiply by 8
  add xA,x2       # Bump A pointer
  vld v2,(xB)
  add xB,x2       # Bump B pointer
  vadd v3,v1,v2
  vst v3,(xC)
  add xC,x2       # Bump C pointer
  sub xN,xN,64    # Subtract elements
  bltz xN,loop    # Any more to do?
```

Vector Processor Optimization

How can a vector processor execute a single vector faster than one element per clock cycle ?

- **Multiple Lanes: beyond one element/cycle**

How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length ?

- **Vector-length Registers: Handling loops not equal to MVL (strip Mining)**

What happens when there is an **IF-ELSE statement** inside the code to be vectorized ?

- **Predicate Registers: vector-mask control**

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 registers**:

– vector version of predicate registers, **1 bit per element**

…and *maskable* vector instructions:

– vector operation becomes bubble ("**NOP**") at elements where mask bit is zero

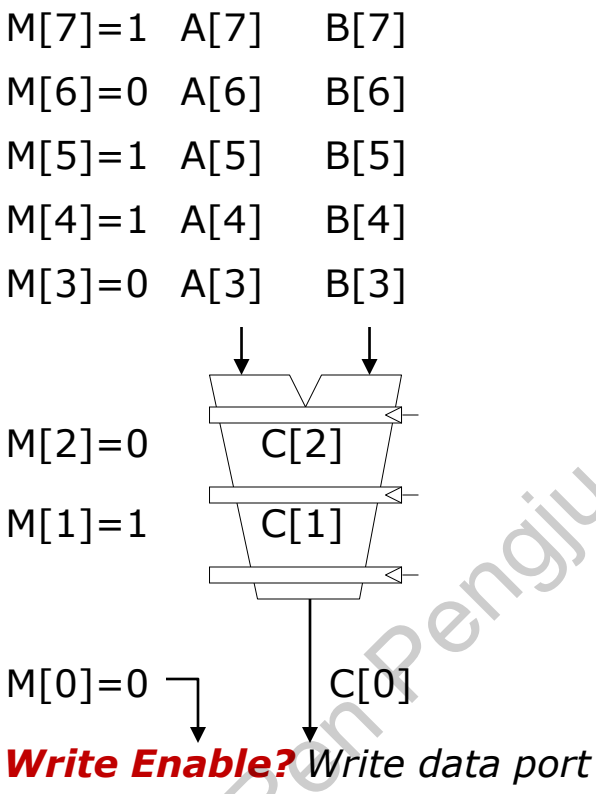
Code example:

```
cvm              # Turn on all elements(clear vector masks)
vld v1,(x1)      # Load entire A vector
Vmgt.vi v0,v1,0  # Set bits in mask register where A>0
vld v2,(x2)      # Load B vector into A under mask
vst v2,(xA),v0.t# Store A back to memory under mask
```

Masked Vector Instructions

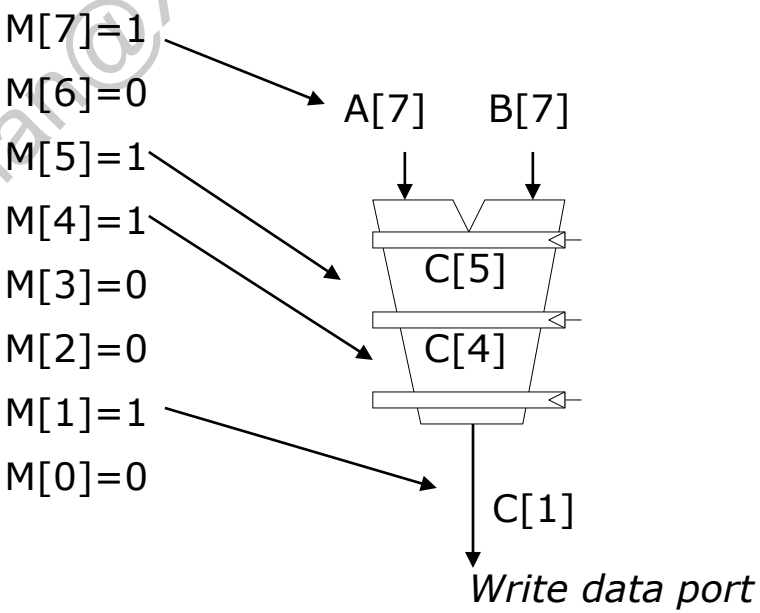
Simple Implementation

- 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



Vector Processor Optimization

How can a vector processor execute a single vector faster than one element per clock cycle ?

- **Multiple Lanes: beyond one element/cycle**

How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length ?

- **Vector-length Registers: Handling loops not equal to MVL (strip Mining)**

What happens when there is an IF statement inside the code to be vectorized ?

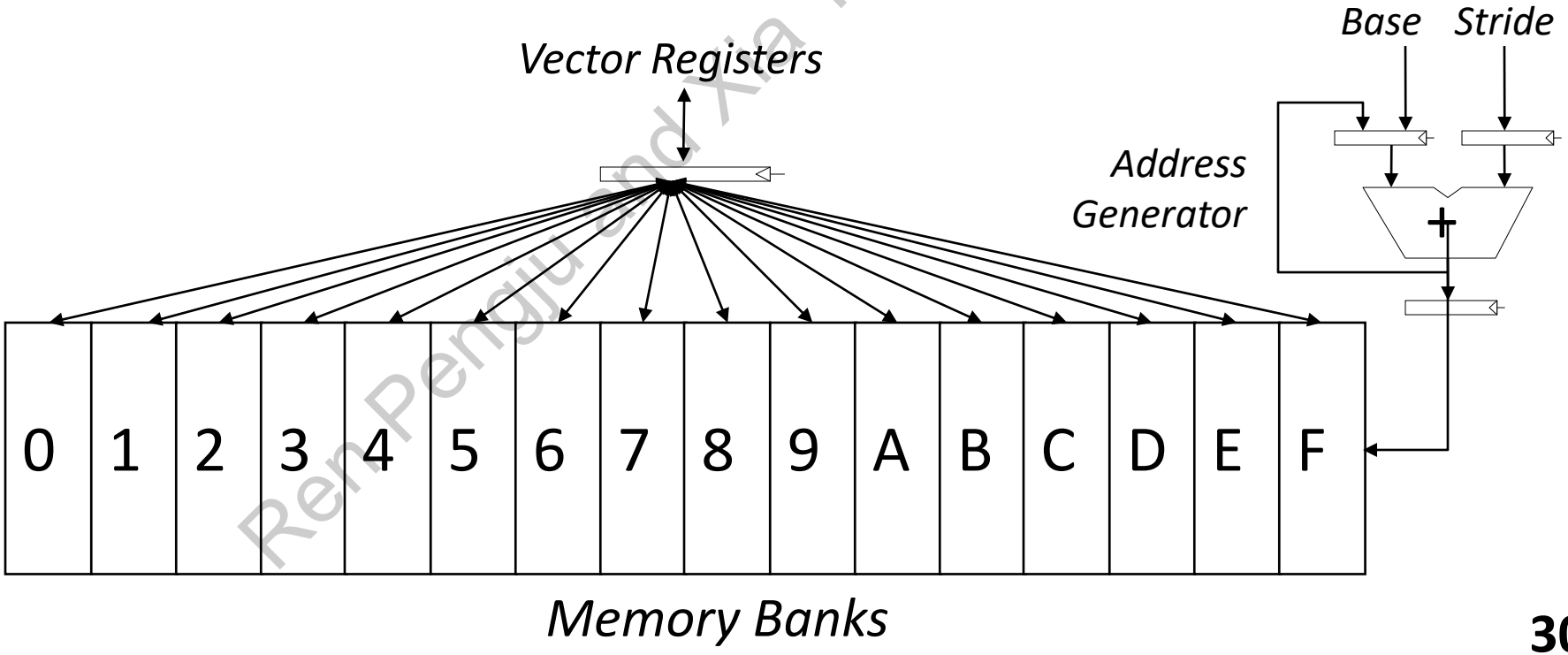
- **Predicate Registers: vector-mask control**

What does a vector processor need from the memory system ?

- **Memory banks: supplying bandwidth for vector Load/Store Units**

Interleaved Vector Memory System

- Memory system must be **designed to support** high bandwidth for vector loads and stores
  - E.g. **16 Banks**, each has **4-cycle latency** between two responses
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words (need independent bank addressing)
  - Support multiple vector processors sharing the same memory



Vector Processor Optimization

How can a vector processor execute a single vector faster than one element per clock cycle ?

- **Multiple Lanes: beyond one element/cycle**

How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length ?

- **Vector-length Registers: Handling loops not equal to MVL (strip Mining)**

What happens when there is an IF statement inside the code to be vectorized ?

- **Predicate Registers: vector-mask control**

What does a vector processor need from the memory system ?

- **Memory banks: supplying bandwidth for vector Load/Store Units**

How does a vector processor handle multiple dimensional matrices ?

- **Data structure must vectorize**
- **Auto-vectorizing**

Stride: Handling Multidimensional Arrays

Problem: Want to vectorize rows/columns

```
for (i=0; i<100; i++)
    for (j=0; j<100; j++){
        A[i][j] = 0.0
        for (k=0; k<100; k++)
            A[i][j]=A[i][j]+B[i][k]*D[k][j];
```

Row Column

Solution: ***nonunit strides***

vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)

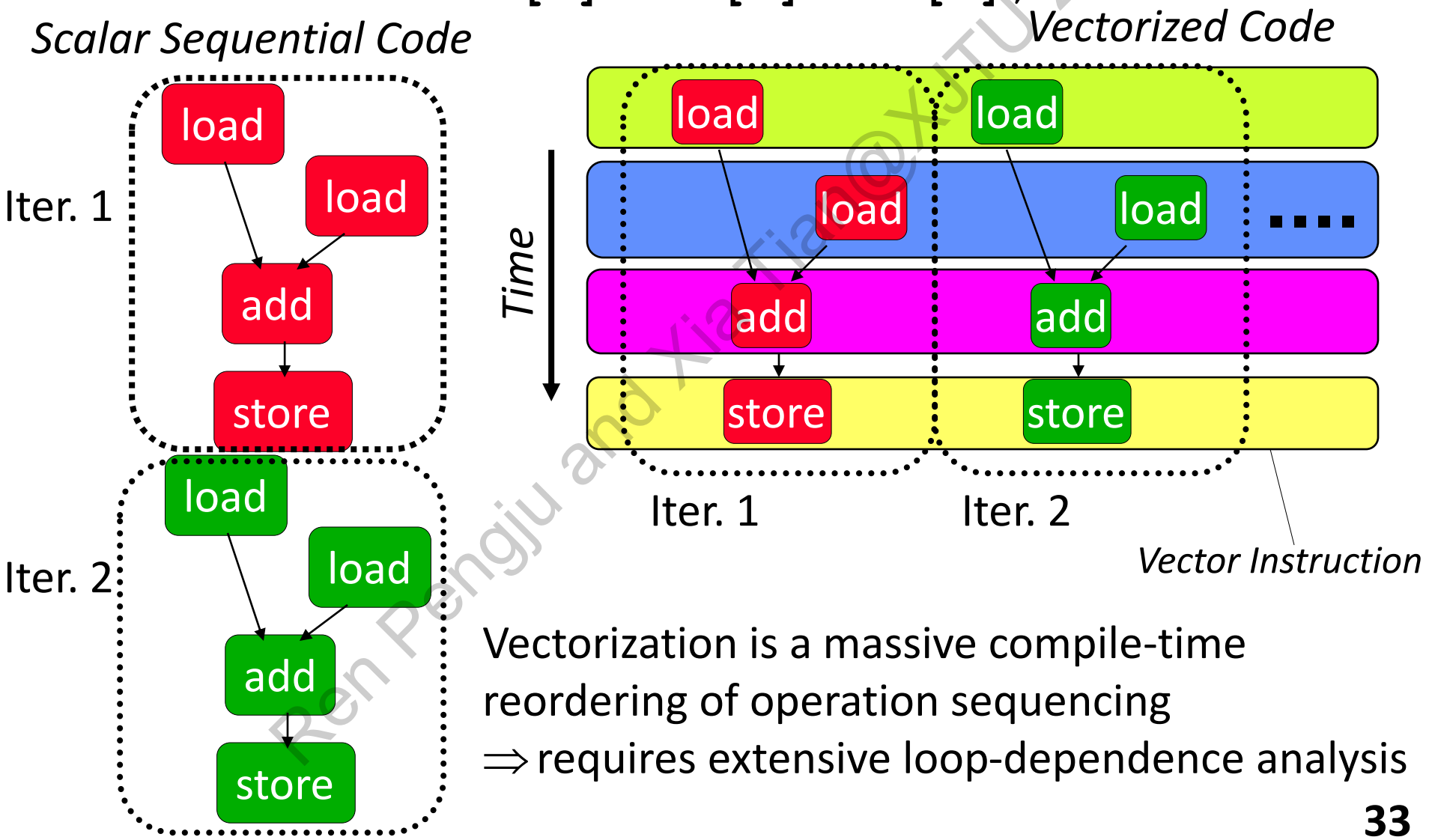
Access **non-sequential memory locations** and to **reshape** them into a dense structure is one of the major advantages of a vector architecture.

RV64V: *VLDS* (load vector with stride)

VSTS (store vector with stride)

Automatic Code Vectorization

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



Vectorization is a massive compile-time reordering of operation sequencing

$\Rightarrow$ requires extensive loop-dependence analysis

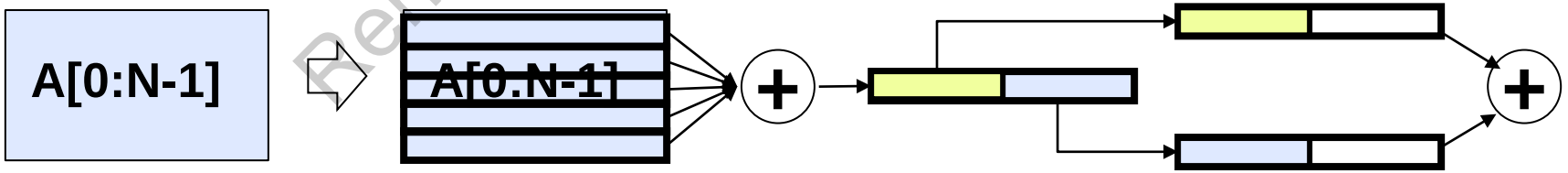
Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0          # Vector of VL partial sums
for(i=0; i<N; i+=VL)     # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1]; # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;           # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of partials
} while (VL>1)
```



Vector Processor Optimization

How can a vector processor execute a single vector faster than one element per clock cycle ?

- **Multiple Lanes: beyond one element/cycle**

How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length ?

- **Vector-length Registers: Handling loops not equal to MVL (strip Mining)**

What happens when there is an IF statement inside the code to be vectorized ?

- **Predicate Registers: vector-mask control**

What does a vector processor need from the memory system ?

- **Memory banks: supplying bandwidth for vector Load/Store Units**

How does a vector processor handle multiple dimensional matrices ?

- **Data structure must vectorize**

How does a vector processor handle sparse matrices ?

- **Vector scatter/gather ： indexed （gather) … = a[b[i]]**
  indexed (scatter) a[b[i]]=…

Vector Scatter-Gather

Problem: Handling **indirect index access**

Solution: *Gather-Scatter operations*

vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)

- Consider:

  for (i = 0; i < n; i=i+1)

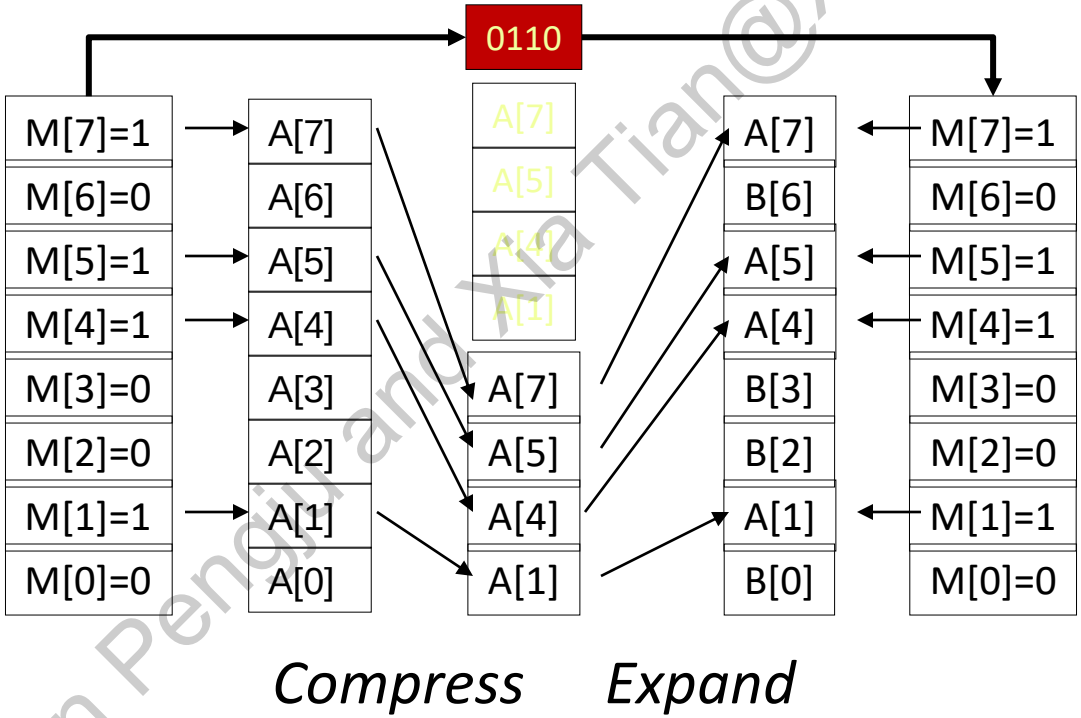
  A[K[i]] = A[K[i]] + C[M[i]];

- Use index vector K[] and M[]:

```
vsetdcfg 4*FP64          # 4 64b FP vector registers
vld      v0, x7          # Load K[]
vldx     v1, x5, v0      # Load A[K[]]
vld      v2, x28         # Load M[]
vldx     v3, x6, v2      # Load C[M[]]
vadd     v1, v1, v3      # Add them
vstx     v1, x5, v0      # Store A[K[]]
vdisable                 # Disable vector registers
```

Compress/Expand Operations

- Compress packs non-masked elements from one vector register contiguously at start of destination vector register
  - population count of mask vector gives packed vector length
- Expand performs inverse operation



Used for density-time conditionals and also for general selection operations

Example of Compress Operations

Compress an array (stream) of values

values = | 3 | 0 | 4 | 1 | 0 | 0 | 3 | 1 |

into

result = | 3 | 4 | 1 | 3 | 1 |

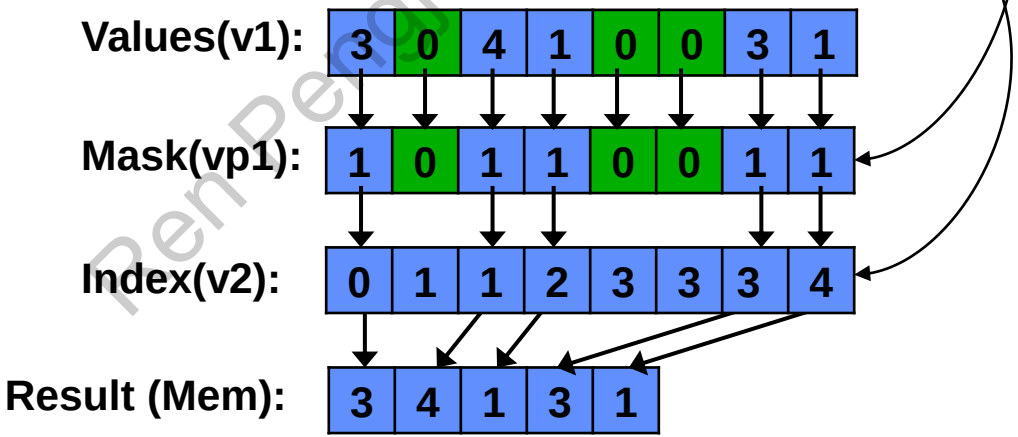
- Step 1: Generate an array of 0/1 flags (mask) :

Flag = | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |

- Step 2: Compute an exclusive add scan of flags to get index

Index = | 0 | 1 | 1 | 2 | 3 | 3 | 3 | 4 |

- Step 3: "**Scatter**" values into result at index, masked by flags



Summary Performance Optimizations

- **Multiple Parallel Lanes, or Pipes**
  - Allows vector operation to be performed in parallel on multiple elements of the vector
- **Strip Mining**
  - Generates code to allow vector operands whose size is less than or greater than size of vector registers
- **Vector Chaining**
  - Equivalent to data forwarding in vector processors
  - Results of one pipeline are fed into operand registers of another pipeline
- **Increase Memory Bandwidth**
  - Memory banks are used to reduce load/store latency
  - Allow multiple simultaneous outstanding memory requests
- **Scatter and Gather**
  - Retrieves data elements scattered throughout memory and packs them into sequential vectors in vector registers
  - Promotes data locality and reduces data pollution

Advantages of Vector Processors

- **Require Lower Instruction Bandwidth**
  - Reduced by fewer fetches and decodes
- **Easier Strided Addressing of Main Memory**
  - Load/Store units access memory with known patterns
- **Elimination of Memory Waste (good spatial locality)**
  - Unlike cache access, every data element that is requested by the processor is actually used – no cache misses
  - Latency only occurs once per vector during pipelined loading
- **Simplification of Control Hazards (less dependency)**
  - Loop-related control hazards from the loop are eliminated
- **Scalable Platform**
  - Increase performance by using more hardware resources
- **Reduced Code Size**
  - Short, single instruction can describe N operations

Next Lecture：Multithreading and Multicore (Thread-level Parallel)

Acknowledgements

- Some slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - David Patterson (UCB)
  - David Wentzlaff (Princeton University)

- MIT material derived from course 6.823
- UCB material derived from course CS252 and CS 61C