数据并行体系结构

吴 迪

中山大学计算机学院

2024年秋季

课程内容

- **参考材料：**
  - Computer Architecture, A Quantitative Approach, 6th Edition.
  - Chapter 4. “Data-Level Parallelism in Vector, SIMD, and GPU Architectures”

4.1 Introduction

Introduction

- **SIMD architectures can exploit significant data-level parallelism for:**
  - Matrix-oriented scientific computing
  - Media-oriented image and sound processors

- **SIMD is more energy efficient than MIMD**
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- **SIMD allows programmer to continue to think sequentially**

SIMD Parallelism

- **3 variations of SIMD:**
  - **Vector architectures**
  - **Multimedia SIMD instruction set extensions**
  - **Graphics Processor Units (GPUs)**

- **For x86 processors:**
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

4.2 Vector Architecture

Supercomputers

- **Definition of a supercomputer:**
  - Fastest machine in world at given task
  - A device to turn a compute-bound problem into an I/O bound problem
  - Any machine costing $30M+
  - Any machine designed by Seymour Cray (^_^)

- **CDC6600 (Cray, 1964) regarded as first supercomputer**

Supercomputer Applications

- **Typical application areas**
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
- **All involve huge computations on large data sets**

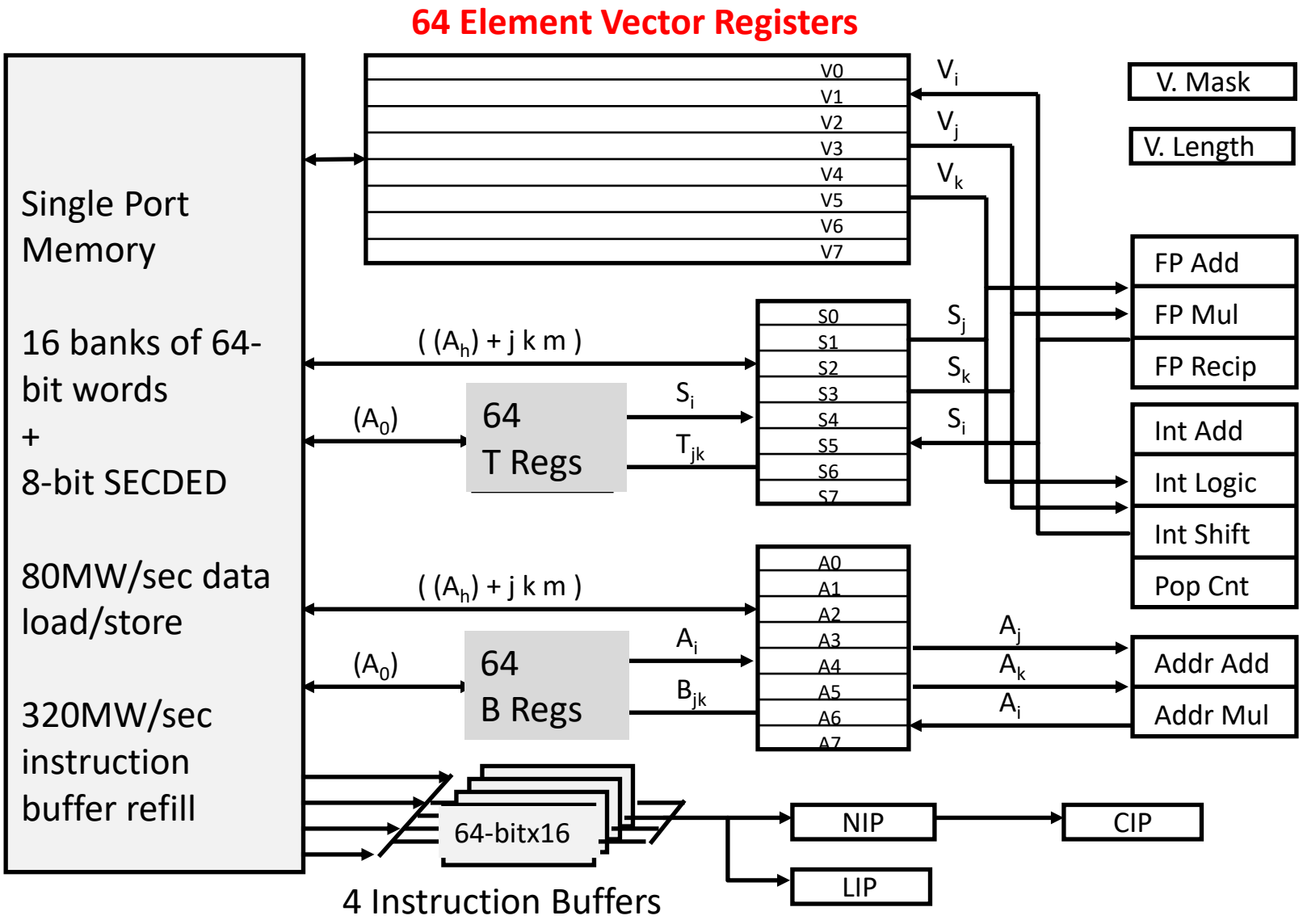
In 70s-80s, Supercomputer = Vector Machine

Vector Supercomputers

- **Epitomized by Cray-1, 1976:**
- **Scalar Unit + Vector Extensions**
  - Load/Store Architecture
  - Vector Registers
  - Vector Instructions
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Cray-1 (1976)

Cray-1 (1976)



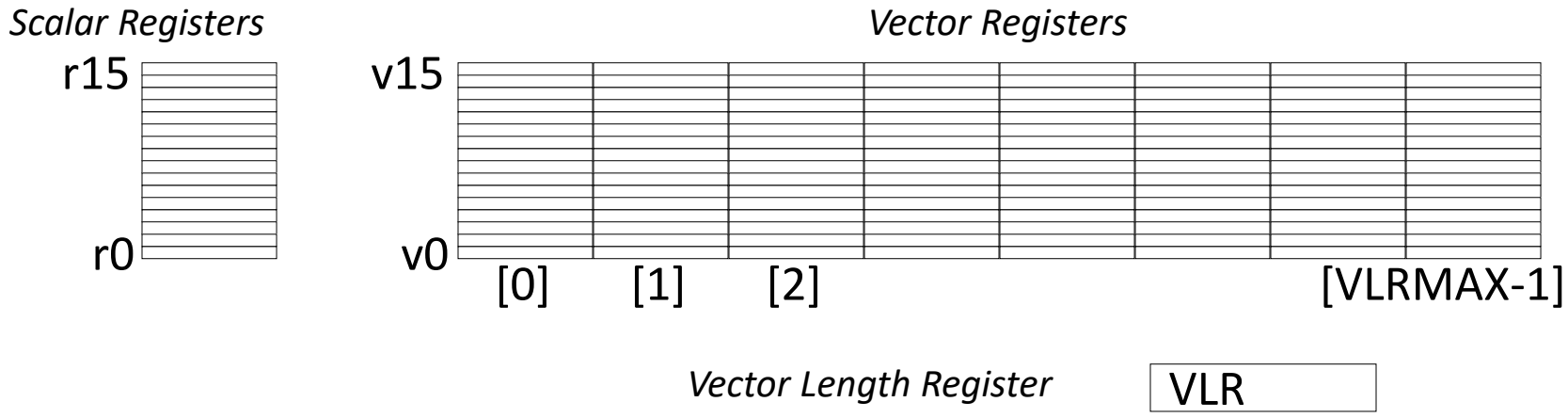
memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Architectures

- **Basic idea:**
  - Read sets of data elements into "**vector registers**"
  - **Operate** on those registers
  - Disperse the results **back into memory**

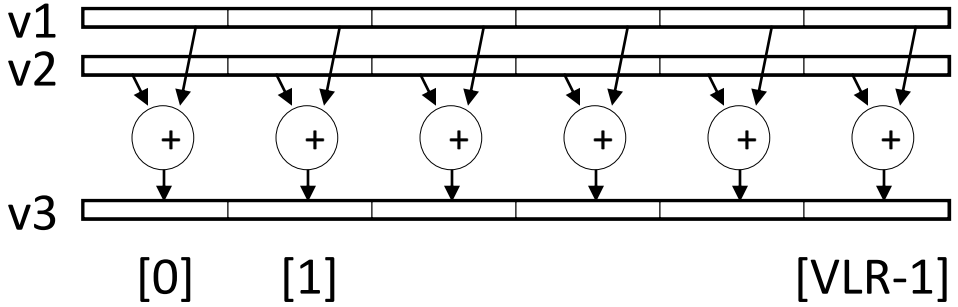
- **Registers are controlled by compiler**
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector Programming Model



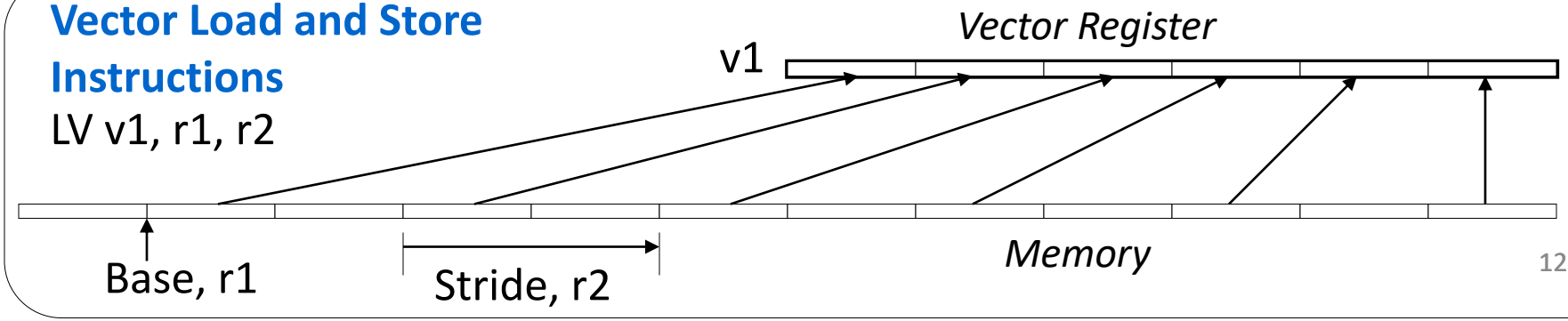
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

LV v1, r1, r2



Vector Code Example

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

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

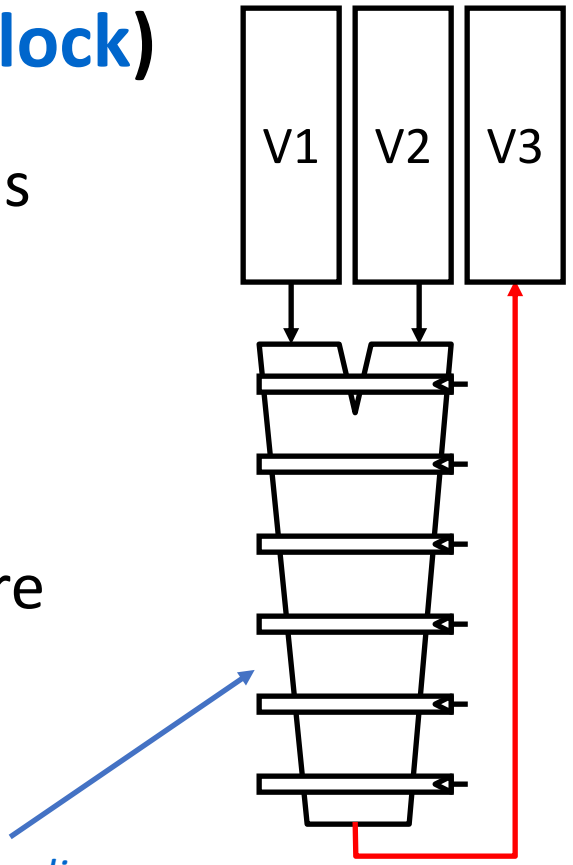
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

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 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)
- **Scalable**
  - can run same object code on more parallel pipelines or 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!)

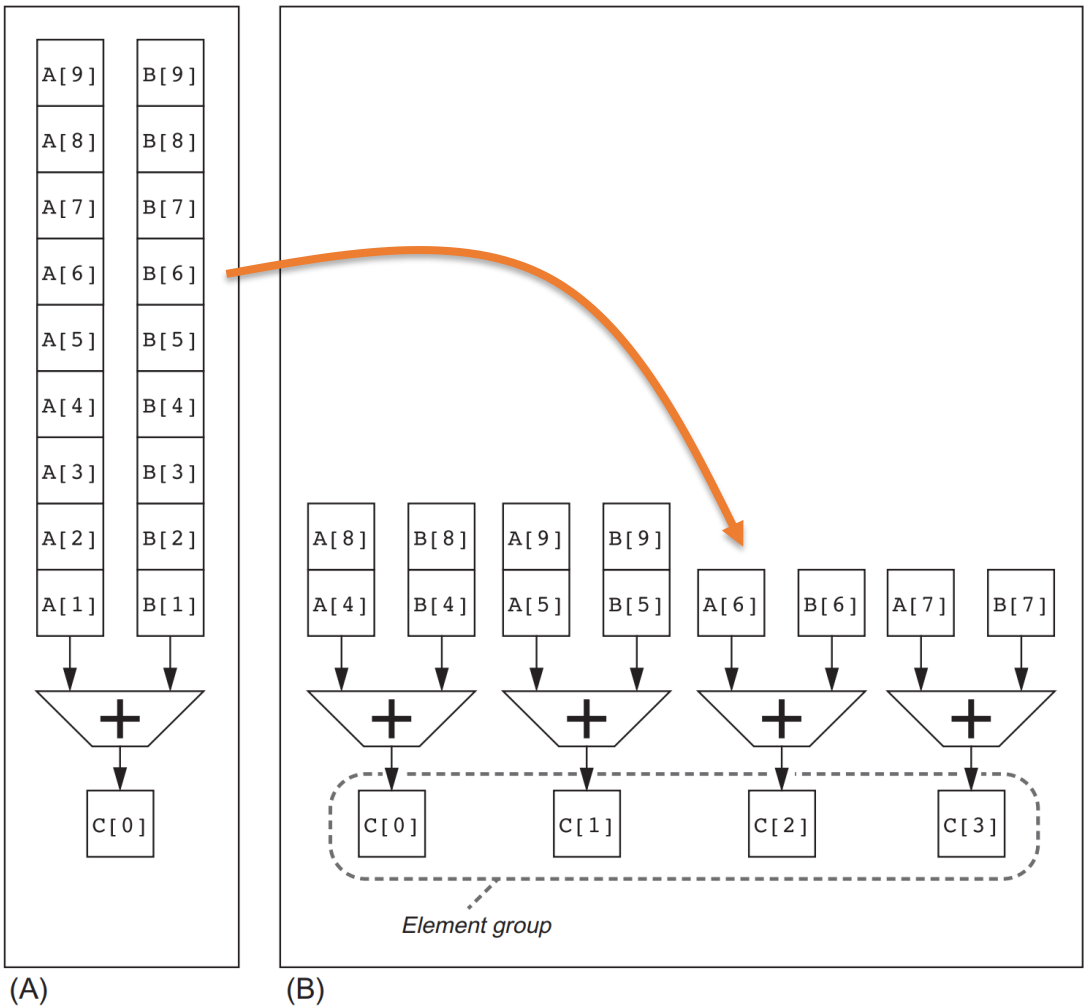


Six stage multiply pipeline

```
V3 <- v1 * v2
```

Vector Instruction Execution

- Using **multiple functional units** to execute a vector add instruction.

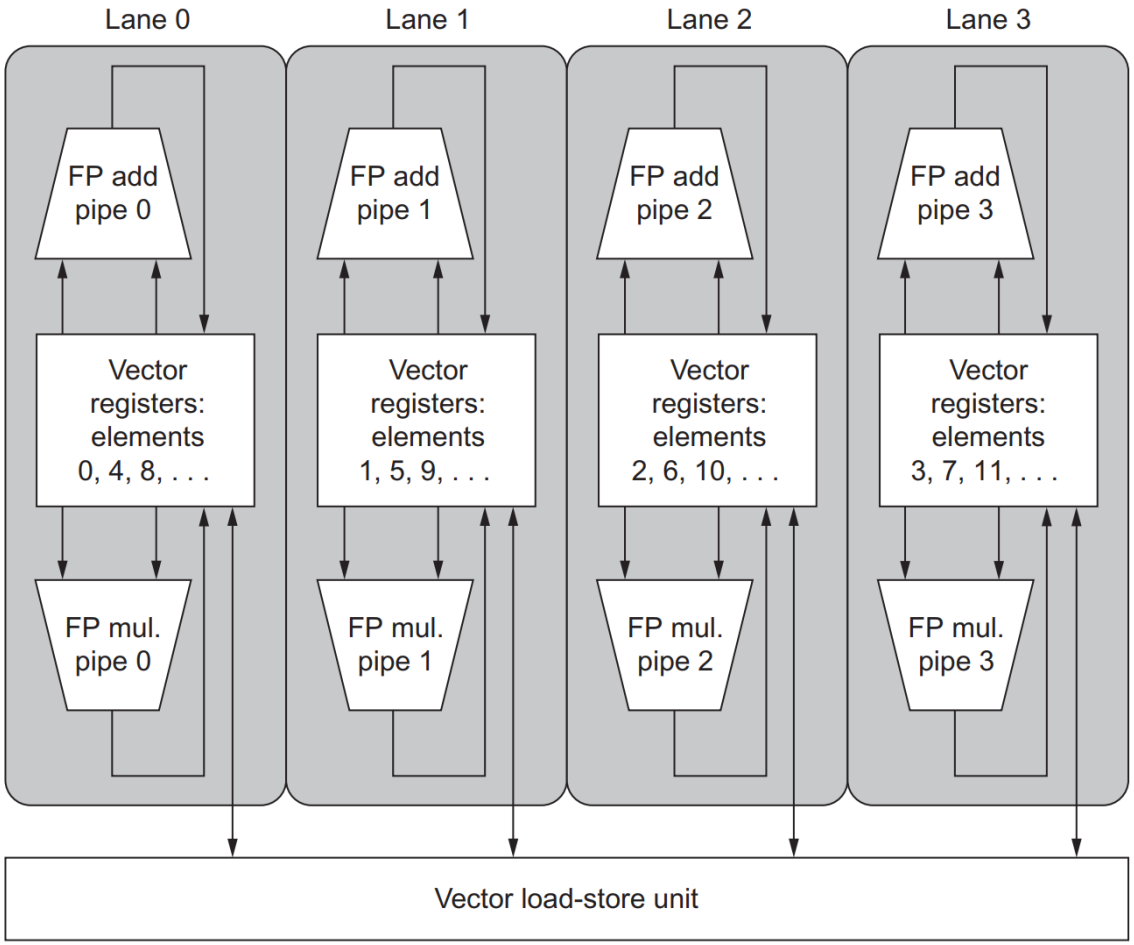


(A) on the left has a single add pipeline and can complete one addition per clock cycle

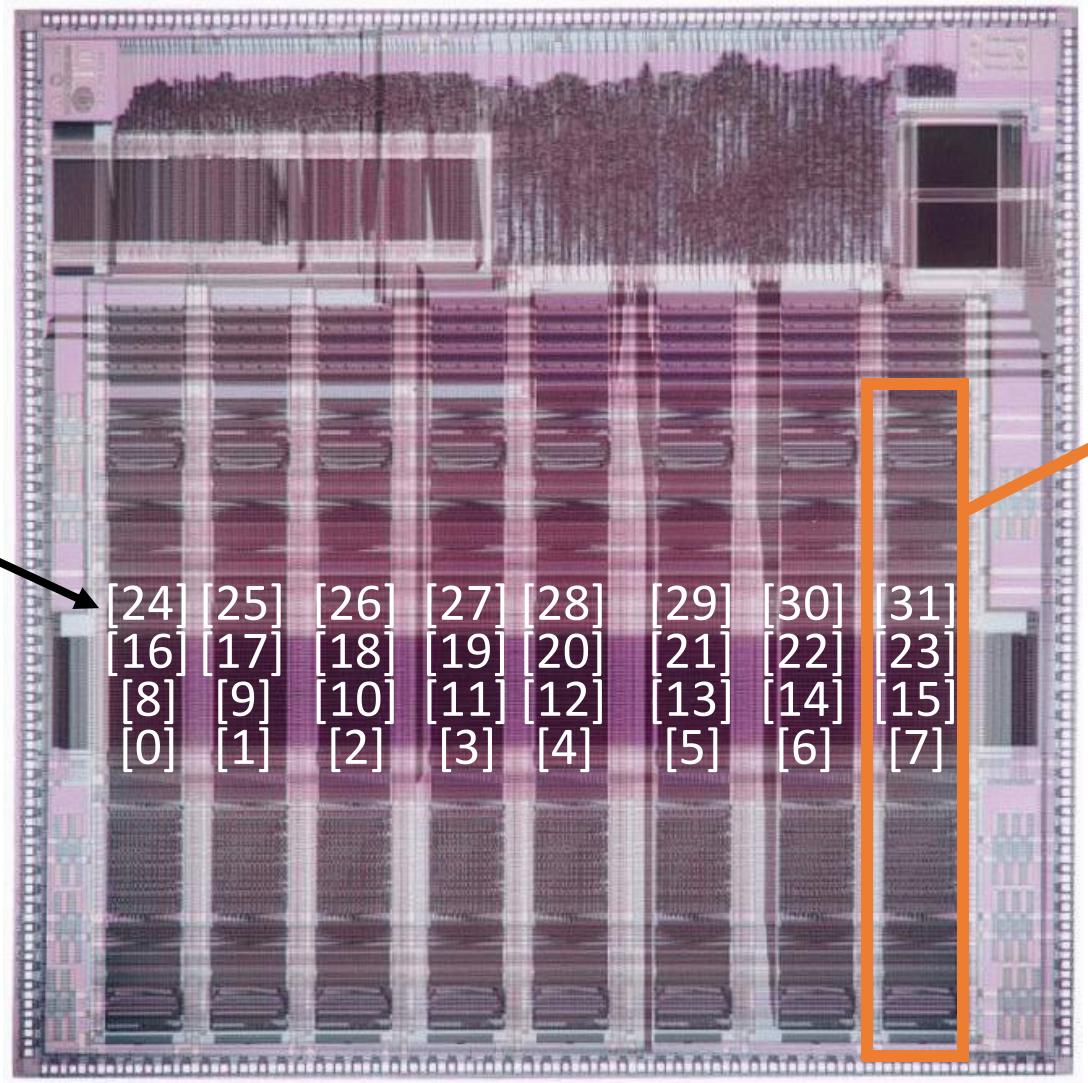
(B) on the right has four add pipelines and can complete four additions per clock cycle.

Vector Unit Structure

- Structure of a vector unit containing four lanes.

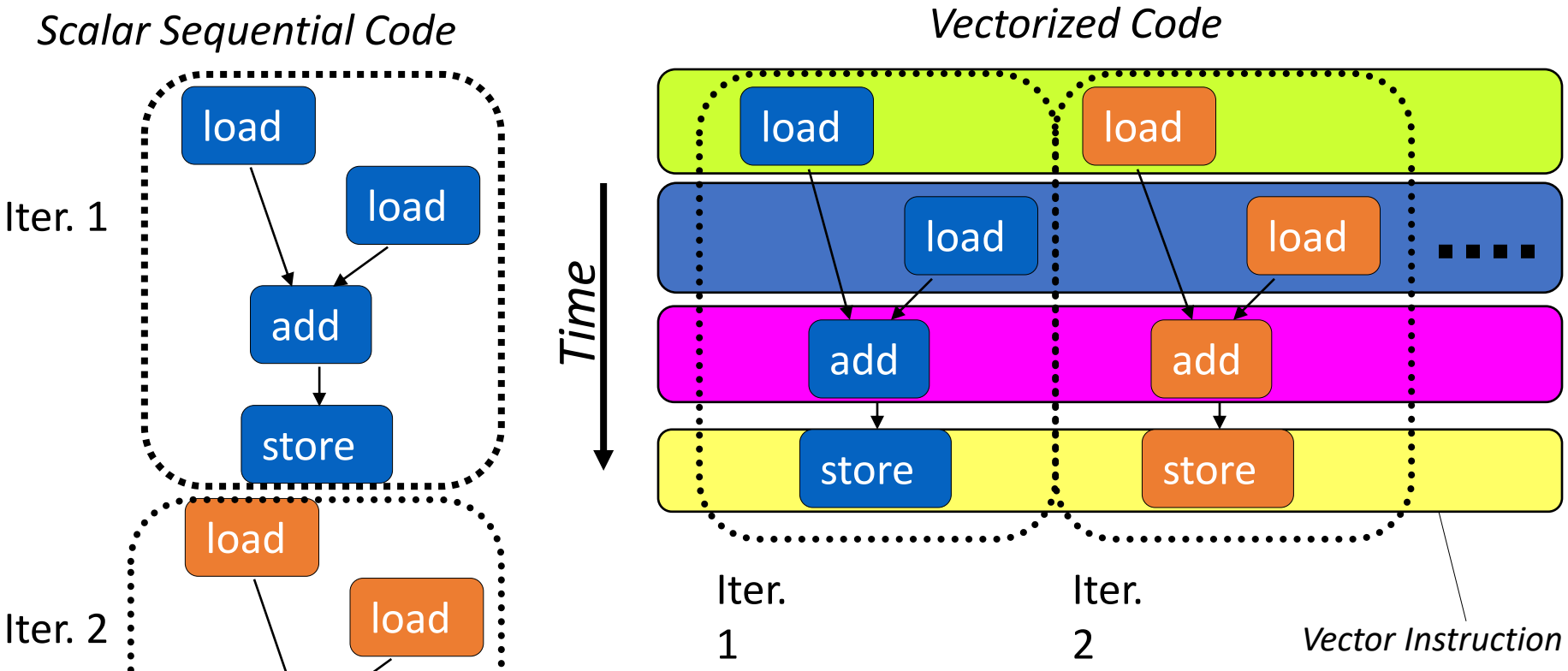


T0 Vector Microprocessor (1995)



Automatic Code Vectorization

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



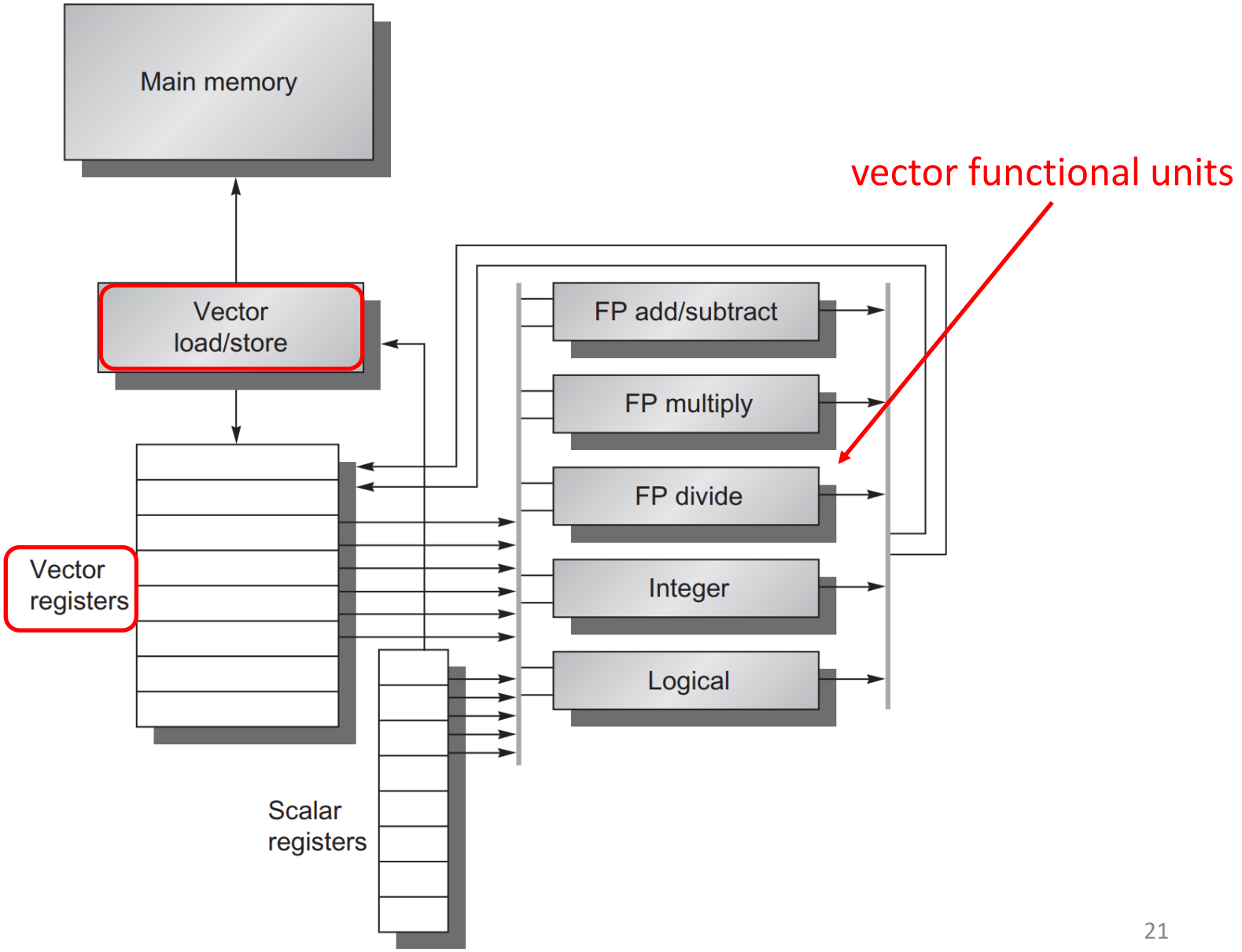
Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

RV64V

- **RV64V:** RISC-V based instructions + vector extension
  - 32 64-bit vector registers
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 31 general-purpose registers
    - 32 floating-point registers

Basic Structure of RV64V



RV64V Instructions

- **RV64V uses the suffix to indicate operand type**
  - `.vv`: two vector operands
  - `.vs` and `.sv`: vector and scalar operands
  - Example: `vsub.vv, vsub.vs`

How Vector Processors Work: An Example

- **Y = a * X + Y**
  - **DAXPY:** Double precision a*X plus Y
- **RV64G code (258 instructions):**

```
        fld     f0,a              # Load scalar a
        addi    x28,x5,#256       # Last address to load
Loop:   fld     f1,0(x5)          # Load X[i]
        fmul.d  f1,f1,f0          # a × X[i]
        fld     f2,0(x6)          # Load Y[i]
        fadd.d  f2,f2,f1          # a × X[i] + Y[i]
        fsd     f2,0(x6)          # Store into Y[i]
        addi    x5,x5,#8          # Increment index to X
        addi    x6,x6,#8          # Increment index to Y
        bne     x28,x5,Loop       # Check if done
```

How Vector Processors Work: An Example

- **RV64V Code (8 instructions):**

```
vsetdcfg 4*FP64      # Enable 4 DP FP vregs
fld      f0,a        # Load scalar a
vld      v0,x5       # Load vector X
vmul     v1,v0,f0    # Vector-scalar mult
vld      v2,x6       # Load vector Y
vadd     v3,v1,v2    # Vector-vector add
vst      v3,x6       # Store the sum
vdisable             # Disable vector regs
```

Vector Execution Time

- **Execution time depends on three factors:**
  - (1) Length of operand vectors
  - (2) Structural hazards among operations
  - (3) Data dependencies

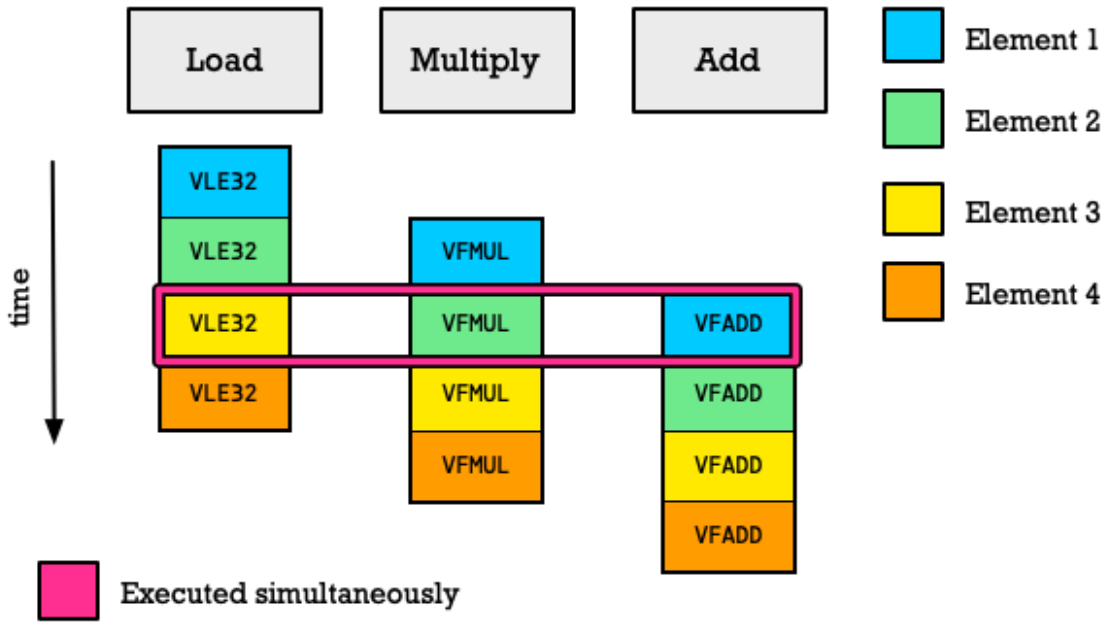
- **RV64V functional units consume one element per clock cycle**
  - Execution time is approximately the vector length

Convoy

- **Convoy (船队，车队)**
  - **Set of vector instructions that could potentially execute together**
  - instructions in a convoy ***must not*** contain any structural hazards
  - If such hazards were present, instructions need to be initiated in different convoys.

Vector Chaining

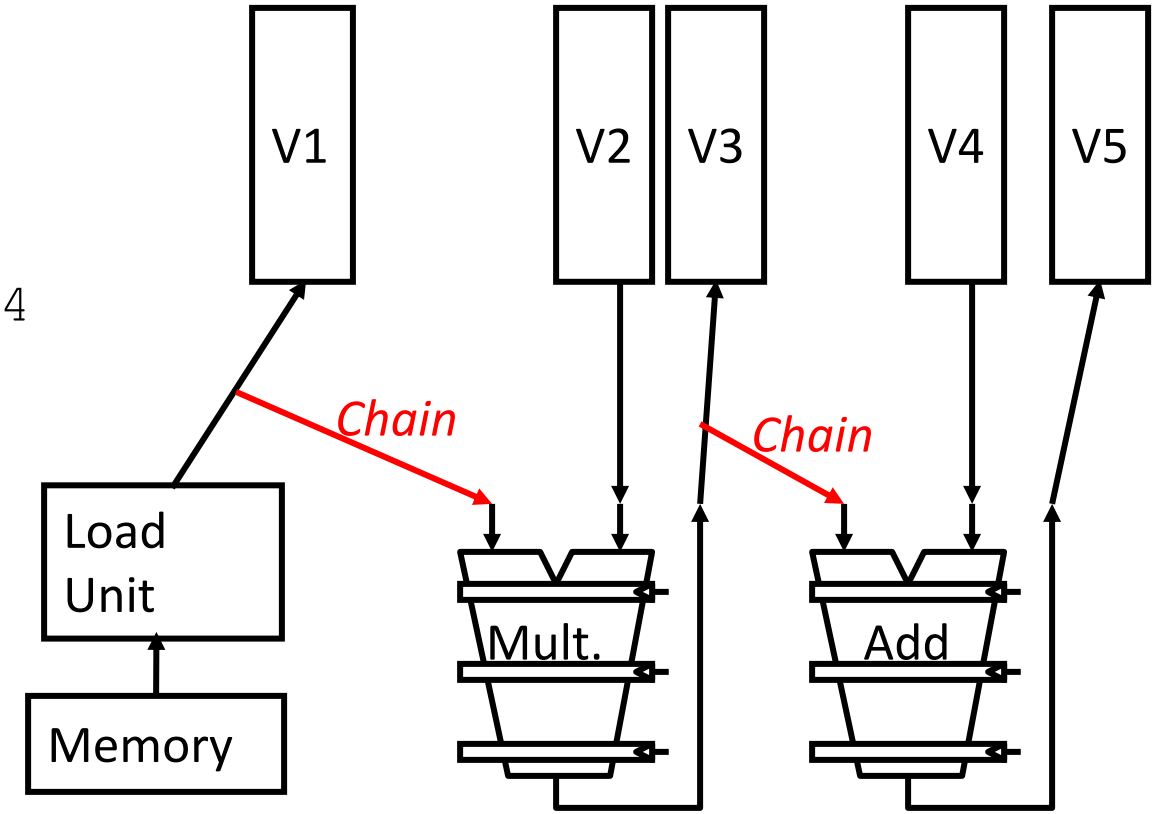
- **Sequences with read-after-write dependency hazards placed in same convoy via *chaining***
  - Results from one functional unit can be passed directly to the next functional unit
  - without being stored in the vector registers first.



Vector Chaining

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

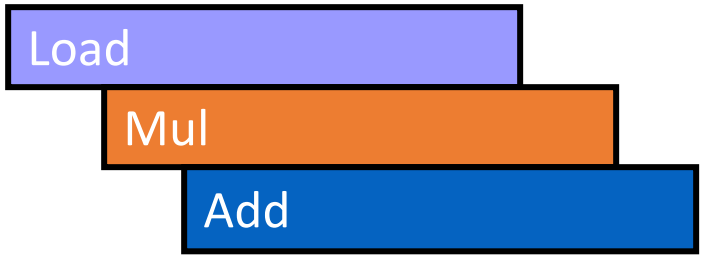
```
LV   v1
MULV v3,v1,v2
ADDV v5,v3, v4
```



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



Example

```
vld       v0,x5       # Load vector X
vmul      v1,v0,f0    # Vector-scalar multiply
vld       v2,x6       # Load vector Y
vadd      v3,v1,v2    # Vector-vector add
vst       v3,x6       # Store the sum
```

Three Convoys:

```
1     vld       vmul
2     vld       vadd
3     vst
```

Questions

- **How can a vector processor execute a single vector faster than one element per clock cycle?**
  - Multiple elements per clock cycle improve performance.
- **How does a vector processor handle programs where the vector lengths are not the same as the maximum vector length (`mvl`)?**
  - most application vectors don't match the architecture vector length
  - we need an efficient solution to this common case.

Questions

- **What happens when there is an IF statement inside the code to be vectorized?**
  - More code can vectorize if we can efficiently handle conditional statements.
- **What does a vector processor need from the memory system?**
  - Without sufficient memory bandwidth, vector execution can be futile（无效的）.

Questions

- **How does a vector processor handle multiple dimensional matrices?**
  - This popular data structure must vectorize for vector architectures to do well.
- **How does a vector processor handle sparse matrices?**
  - This popular data structure must be vectorized too.
- **How do you program a vector computer?**
  - If mismatch to programming languages, their compilers may not get widespread use.

Vector Length Register

Vector Length Register

- **In a real program, the length of a particular vector operation is often unknown at compile time**
  - n might also be a parameter subject to change during execution.

```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```

Vector Length Register

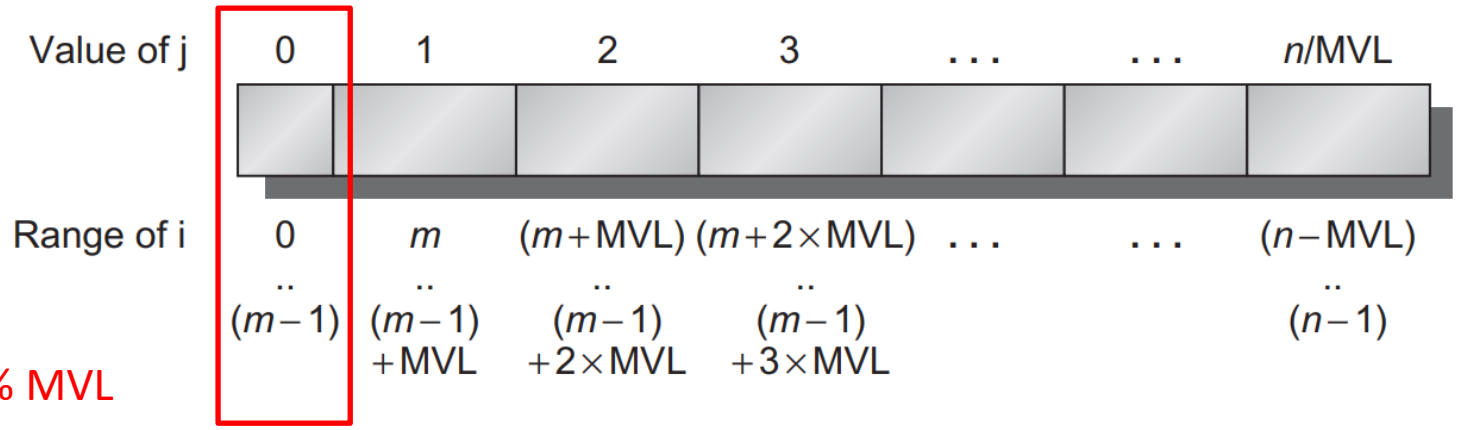
- **The solution to these problems is to add a vector-length register (`vl`).**
  - `vl` controls the length of any vector operation.
  - The value in the vl cannot be greater than the **maximum vector length (`mvl`).**

```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```

Problem: What if the value of n is not known at compile time and thus may be greater than mvl?

Strip Mining

- **Solution: Strip mining**
  - the generation of code such that each vector operation is done for a **`size <= mvl`**.
  - One loop handles any number of iterations that is a multiple of **`mvl`**
  - another loop that handles any remaining iterations and must be less than **`mvl`** .



m = n % MVL

Vector Length Register

- **RISC-V has a better solution than a separate loop for strip mining.**
- **The instruction `setvl` writes the smaller of the mvl and the loop variable n into vl.**
  - If n is greater than mvl, then the fastest the loop can compute is mvl values at time, so setvl sets vl to mvl.
  - If n is smaller than mvl, it should compute only on the last n elements in this final iteration of the loop, so setvl sets vl to n.

Vector Length Register

```
        vsetdcfg  2 DP FP     # Enable 2 64b Fl.Pt. registers
        fld       f0,a        # Load scalar a
loop:   setvl     t0,a0       # vl = t0 = min(mvl,n)
        vld       v0,x5       # Load vector X
        slli      t1,t0,3     # t1 = vl * 8 (in bytes)
        add       x5,x5,t1    # Increment pointer to X by vl*8
        vmul      v0,v0,f0    # Vector-scalar mult
        vld       v1,x6       # Load vector Y
        vadd      v1,v0,v1    # Vector-vector add
        sub       a0,a0,t0    # n -= vl (t0)
        vst       v1,x6       # Store the sum into Y
        add       x6,x6,t1    # Increment pointer to Y by vl*8
        bnez      a0,loop     # Repeat if n != 0
        vdisable              # Disable vector regs
```

Vector Conditional Execution

Vector Conditional Execution

- **Problem: Programs that contain IF statements in loops cannot be vectorized**
  - IF statements introduce control dependences into a loop.
  - if the inner loop could be run for the iterations for which `X[i]!=0`, then the subtraction could be vectorized

```
for (i = 0; i < 64;   i=i+1)
  if (X[i] != 0)
    X[i] = X[i] - Y[i];
```

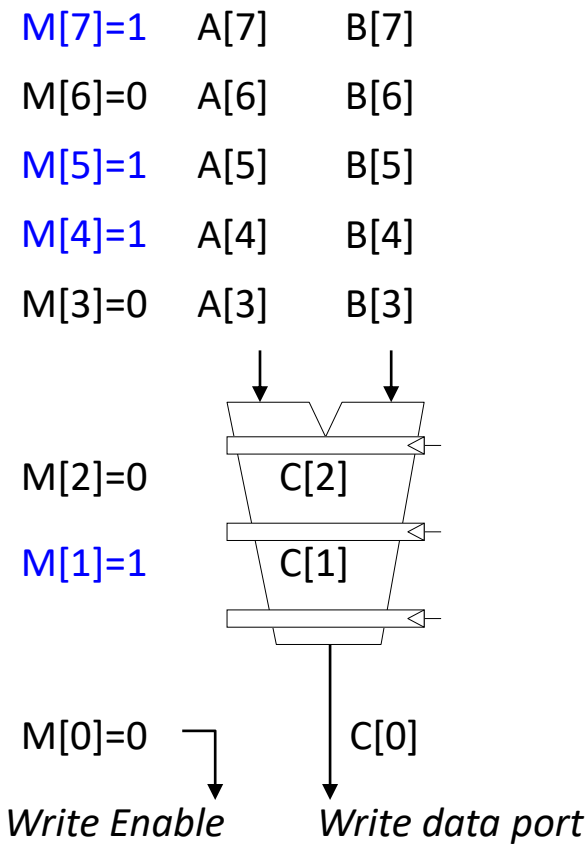
Vector Conditional Execution

- **Vector-mask control.**
  - **Predicate(断言) registers** hold the mask and essentially provide conditional execution of each element operation in a vector instruction.
  - These registers use a **boolean vector** to control the execution of a vector instruction.
    - When the predicate register p0 is set, all following vector instructions operate only on the vector elements whose corresponding entries in the predicate register are 1.

Masked Vector Instructions

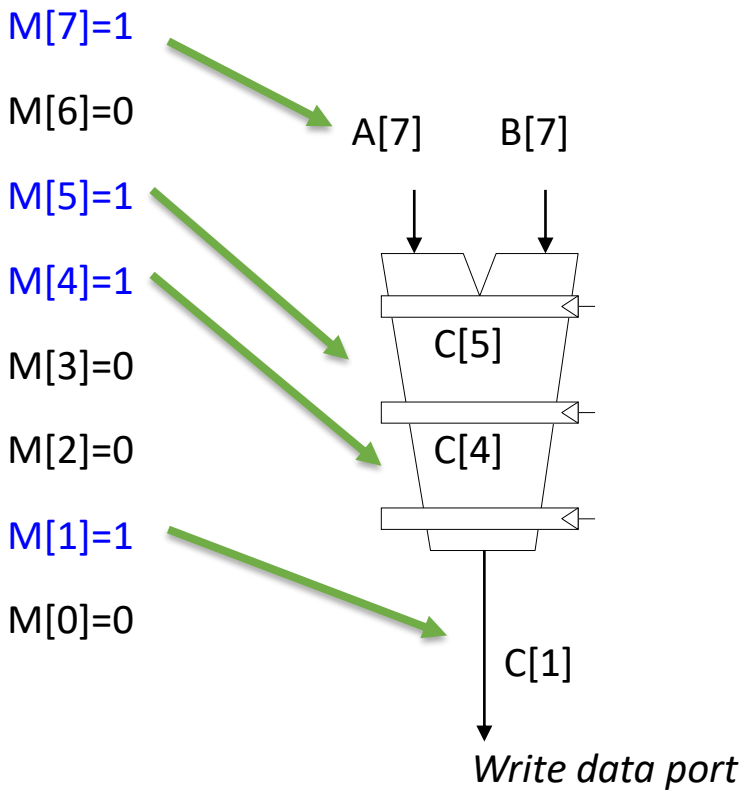
Simple Implementation

– 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 Mask Registers

- Use predicate register to "disable" elements:

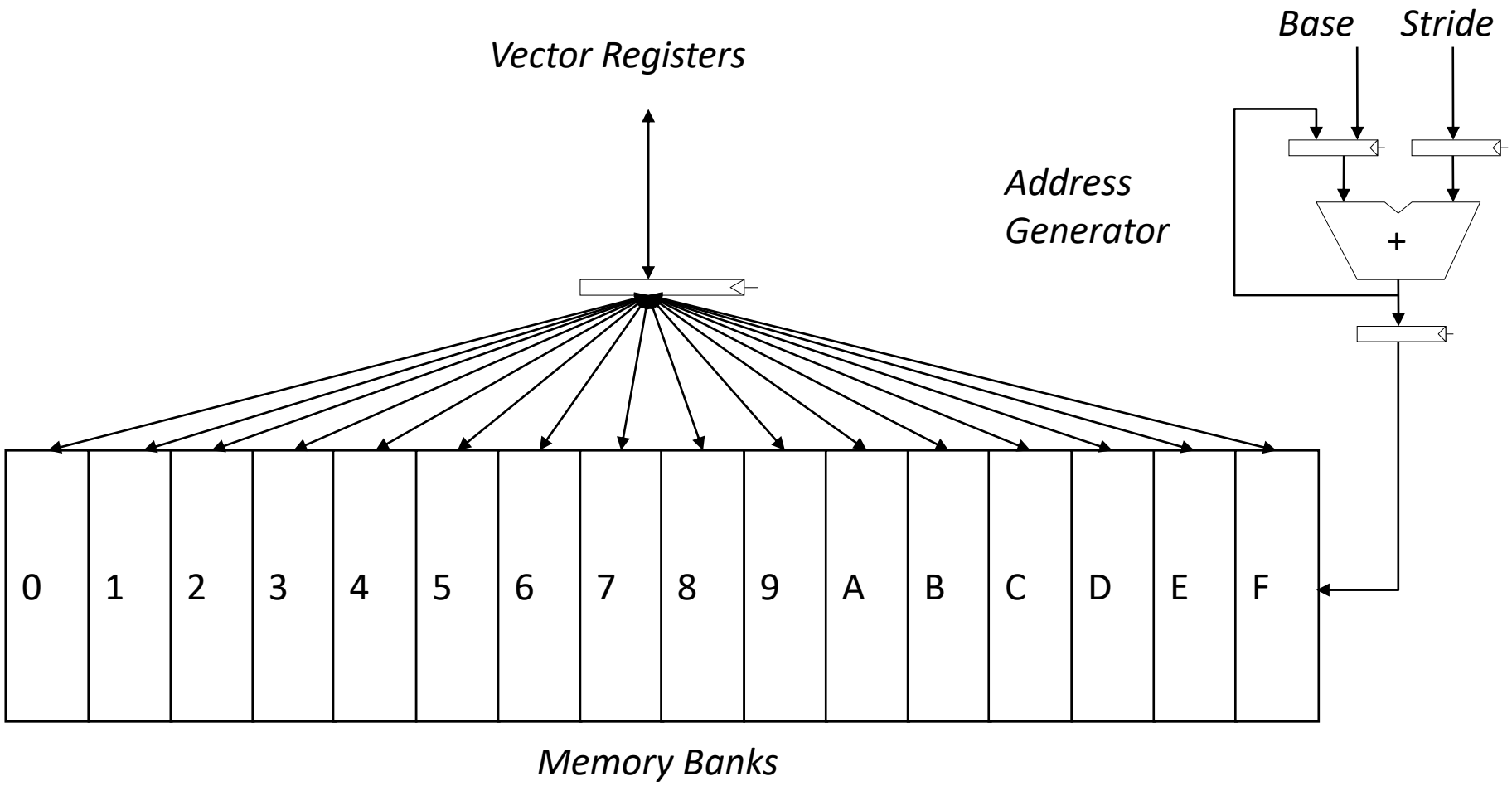
```
vsetdcfg   2*FP64 # Enable 2 64b FP vector regs
vsetpcfgi  1      # Enable 1 predicate register
vld        v0,x5  # Load vector X into v0
vld        v1,x6  # Load vector Y into v1
fmv.d.x    f0,x0  # Put (FP) zero into f0
vpne       p0,v0,f0      # Set p0(i) to 1 if v0(i)!=f0
vsub       v0,v0,v1      # Subtract under vector mask
vst        v0,x5  # Store the result in X
vdisable          # Disable vector registers
vpdisable         # Disable predicate registers
```

Vector Memory Subsystem

Vector Memory Subsystem

- **Memory system must be designed to support high bandwidth for vector loads and stores**
- **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 Memory Subsystem



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

- Bank busy time: Cycles between accesses to same bank

Vector Memory Subsystem

- **Example:**
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - **Question： How many memory banks needed?**
    - 32x(4+2)x15/2.167 = 1344 banks

Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory
  - The first vector machines, CDC Star-100 ('73) and TI ASC ('71), were memory-memory machines
  - Cray-1 ('76) was first vector register machine

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

- **Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?**
  - All operands must be read in and out of memory
- **VMMAs make if difficult to overlap execution of multiple vector operations, why?**
  - Must check dependencies on memory addresses
- **VMMAs incur greater startup latency**

Vector Memory-Memory vs. Vector Register Machines

All major vector machines since Cray-1 have had vector register architectures

(we ignore vector memory-memory from now on)

Handling Multidimensional Arrays

Handling Multidimensional Arrays in Vector Architectures

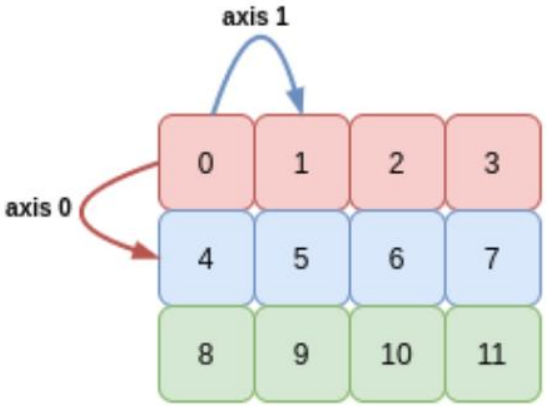
- The position in memory of adjacent elements in a vector may **not be sequential**.
- We could **vectorize** the multiplication of each row of B with each column of D
  - we must consider how to address adjacent elements in B and adjacent elements in D.

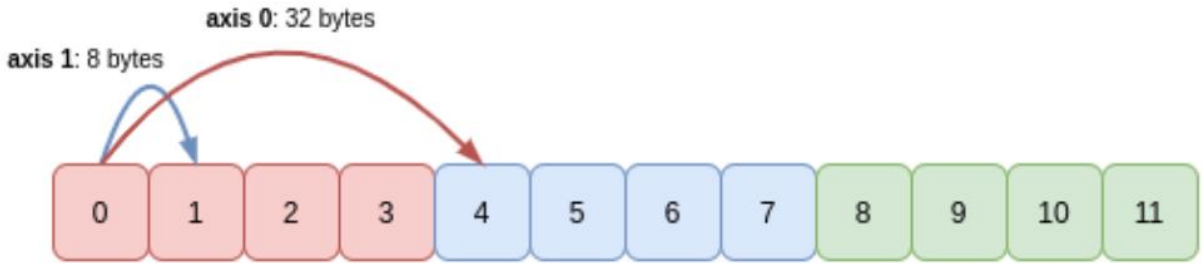
```
for (i = 0; i < 100;  i=i+1)
  for (j = 0; j < 100;  j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

A=B*D

Stride (步幅)

- **Stride**：the distance separating elements to be gathered into a single vector register





Stride

- matrix D has a stride of 100 double words (800 bytes)
- matrix B has a stride of 1 double word (8 bytes).

```
for (i = 0; i < 100;  i=i+1)
  for (j = 0; j < 100;  j=j+1) {
    A[i][j] = 0.0;
    for (k = 0; k < 100; k=k+1)
      A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

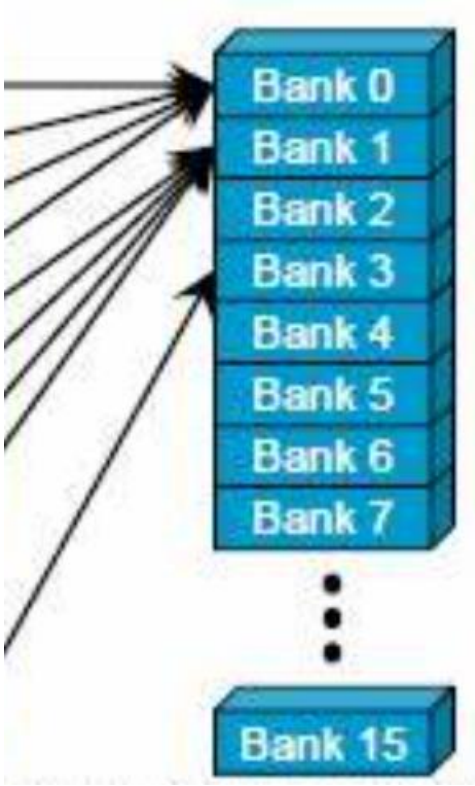
Stride

- **Non-unit stride： strides greater than one**
  - Once a vector is loaded into a vector register, it acts as if it had logically adjacent elements.

- **Handle non-unit strides using only vector load and vector store operations with stride capability.**
  - The ability to access **nonsequential memory locations** and to **reshape them into a dense structure**
  - one of the major advantages of a vector architecture

Stride

- It is possible to request accesses from the same bank frequently.
- When multiple accesses contend for a bank, a memory bank conflict occurs.



Handling Sparse Matrices

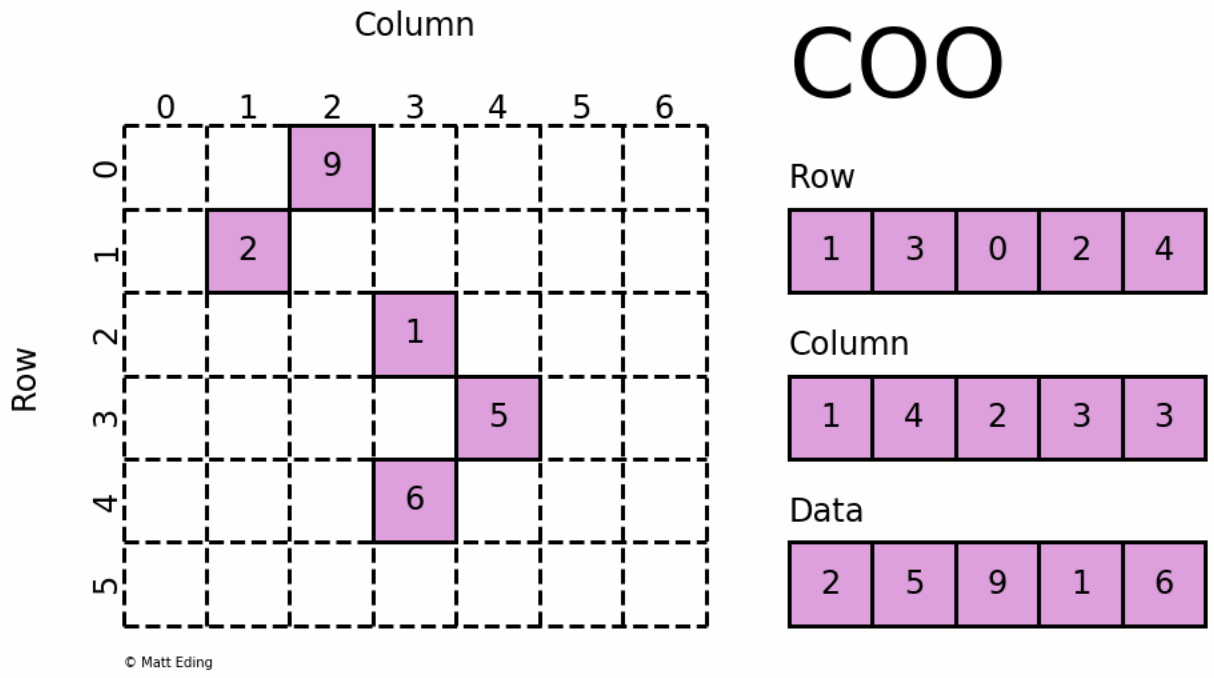
Handling Sparse Matrices in Vector Architectures

- **Sparse matrices are common**

$$\begin{bmatrix} 0 & 9 & 0 & 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 6 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Handling Sparse Matrices in Vector Architectures

- In a sparse matrix, the elements of a vector are usually stored in some compacted form and then accessed indirectly.



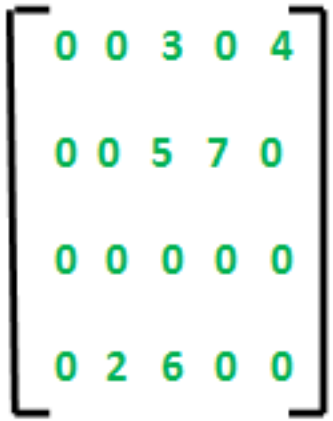
Handling Sparse Matrices in Vector Architectures

- **This code implements a sparse vector sum on the arrays A and C**
  - using index vectors K and M to designate the nonzero elements of A and C.

```
for (i = 0; i < n;  i=i+1)
  A[K[i]] = A[K[i]] + C[M[i]];
```

Gather-Scatter

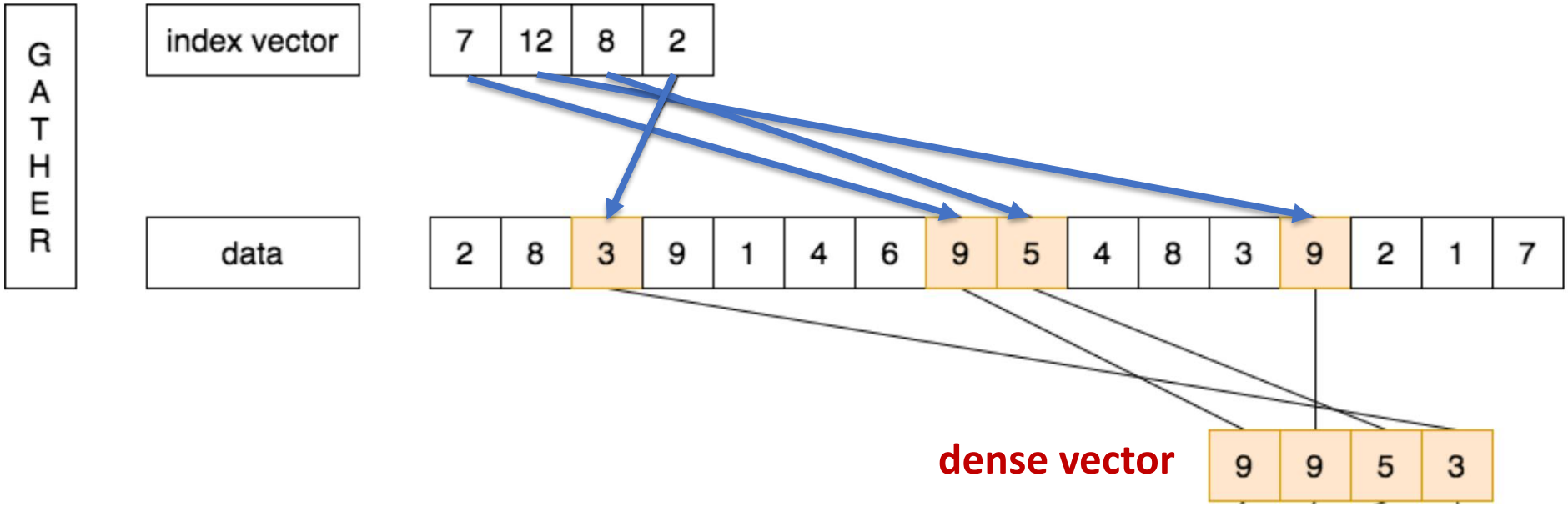
- **The primary mechanism for supporting sparse matrices is gather-scatter operations using index vectors.**
  - Support moving between a compressed representation (i.e., zeros are not included) and normal representation (i.e., zeros are included) of a sparse matrix.



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

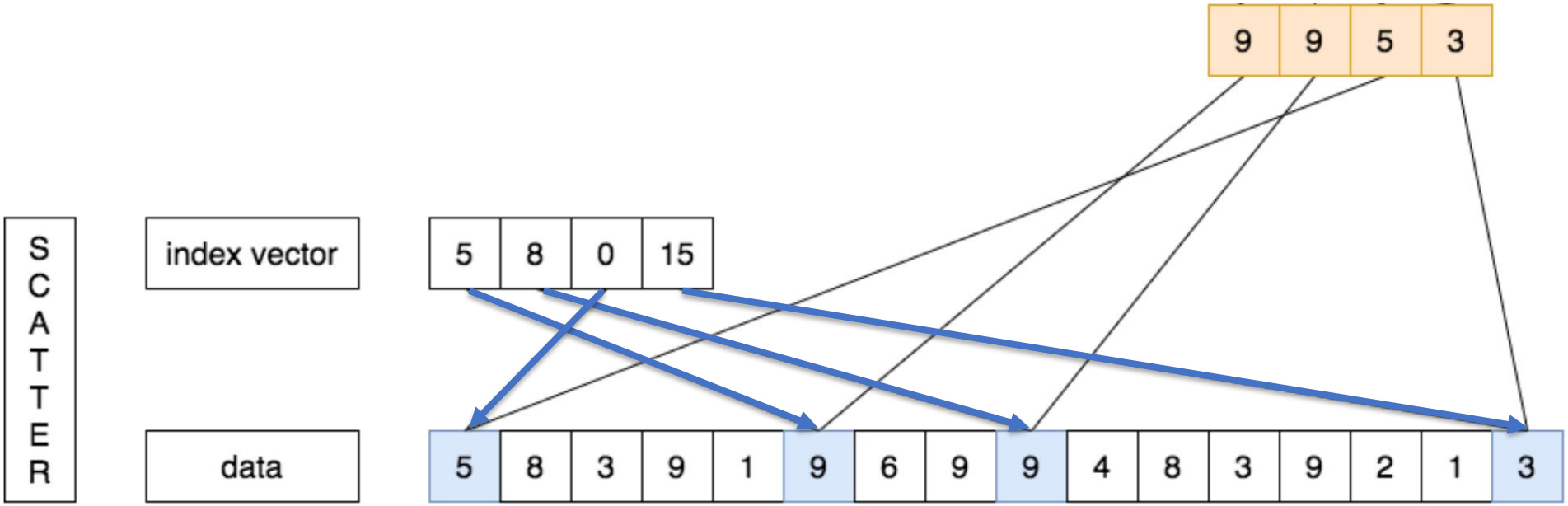
Gather-Scatter

- **A gather operation takes an index vector and fetches the vector**
  - whose elements are at the addresses given by adding a base address to the offsets given in the index vector.
  - The result is a **dense vector** in a vector register.



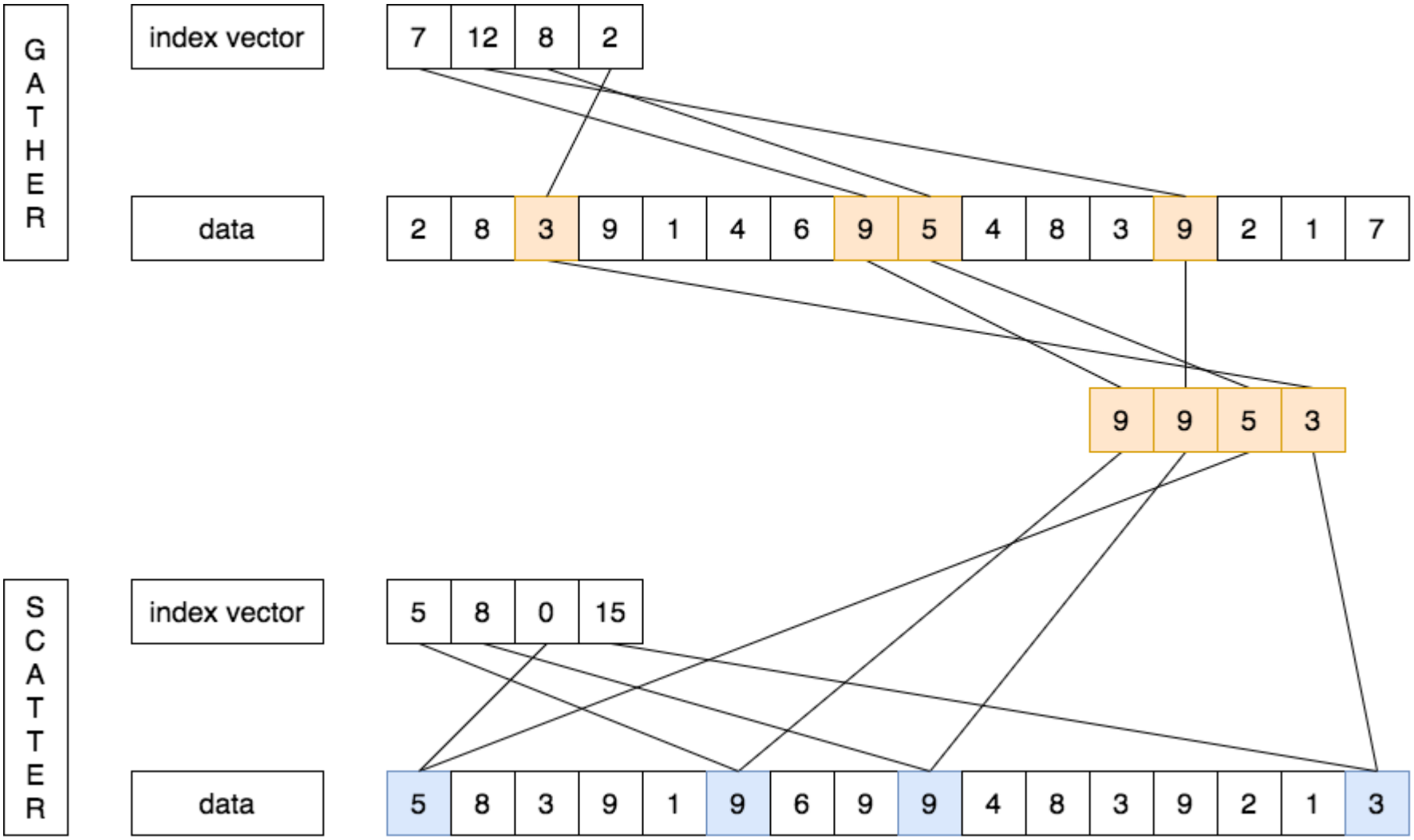
Gather-Scatter

- **Sparse vector can be stored in an expanded form by a scatter store**
  - using the same index vector.



Gather-Scatter

- **Hardware support for such operations is called gather-scatter.**



Gather-Scatter Example

- **`vldx`** (load vector indexed or gather)
- **`vstx`** (store vector indexed or scatter).

```
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[]
vldi      v3, x6, v2)   # Load C[M[]]
vadd      v1, v1, v3    # Add them
vstx      v1, x5, v0)   # Store A[K[]]
vdisable                # Disable vector registers
```

x5, x6, x7, and x28 contain the starting addresses of the vectors

Programming Vector Architectures

- **Compilers** can tell programmers at **compile time** whether a section of code will **vectorize or not**
- **Programmers** can provide **hints** to compiler

Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, with programmer aid	Speedup from hint optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Summary

- **Vector is alternative model for exploiting ILP**
- **If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than out-of-order machines**
- **Design issues include:**
  - number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- **Fundamental design issue is memory bandwidth**