

# **CS425**

# **Computer Systems Architecture**

**Fall 2023**

**Vector Processors**

# Flynn's Taxonomy of Computers

- Mike Flynn, “**Very High-Speed Computing Systems**,” Proc. of IEEE, 1966
- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor

# Data Parallelism

- Concurrency arises from performing the **same operation on different pieces of data**
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors
- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel
- SIMD exploits operation-level parallelism on different data
  - Same operation concurrently applied to different pieces of data
  - A form of ILP where instruction happens to be the same across data

# Vector Processors (1/2)

- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

```
for (i = 0; i <= 49; i++)  
    C[i] = (A[i] + B[i]) / 2
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- Basic requirements
  - Need to load/store vectors → vector registers (contain vectors)
  - Need to operate on vectors of different lengths → vector length register (VLEN)
  - Elements of a vector might be stored apart from each other in memory → vector stride register (VSTR)
    - Stride: distance in memory between two elements of a vector

# Vector Processors (1/2)

- A vector instruction performs an operation on each element in consecutive cycles
  - Vector functional units are pipelined
  - Each pipeline stage operates on a different data element
- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking needed within a vector
  - No control flow within a vector
  - Known stride allows easy address calculation for all vector elements
    - Enables prefetching of vectors into registers/cache/memory

# Vector Processor Properties

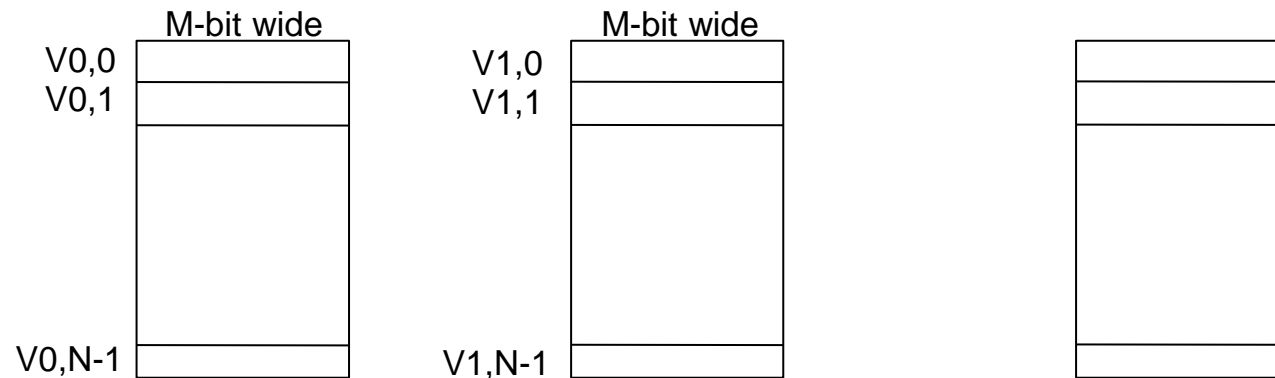
- No dependencies within a vector
  - Pipelining & parallelization work really well
  - Can have very deep pipelines, no dependencies!
- Each instruction generates a lot of work
  - Reduces instruction fetch bandwidth requirements
- Highly regular memory access pattern
- No need to explicitly code loops
  - Fewer branches in the instruction sequence
- Works well if parallelism is regular (data/SIMD parallelism)
  - Many vector operations
  - Very inefficient if parallelism is irregular

# Vector Processor Limitations

- Memory (bandwidth) can easily become a bottleneck when:
  - compute/memory operation balance is not maintained
  - data is not mapped appropriately to memory banks

# Vector Registers

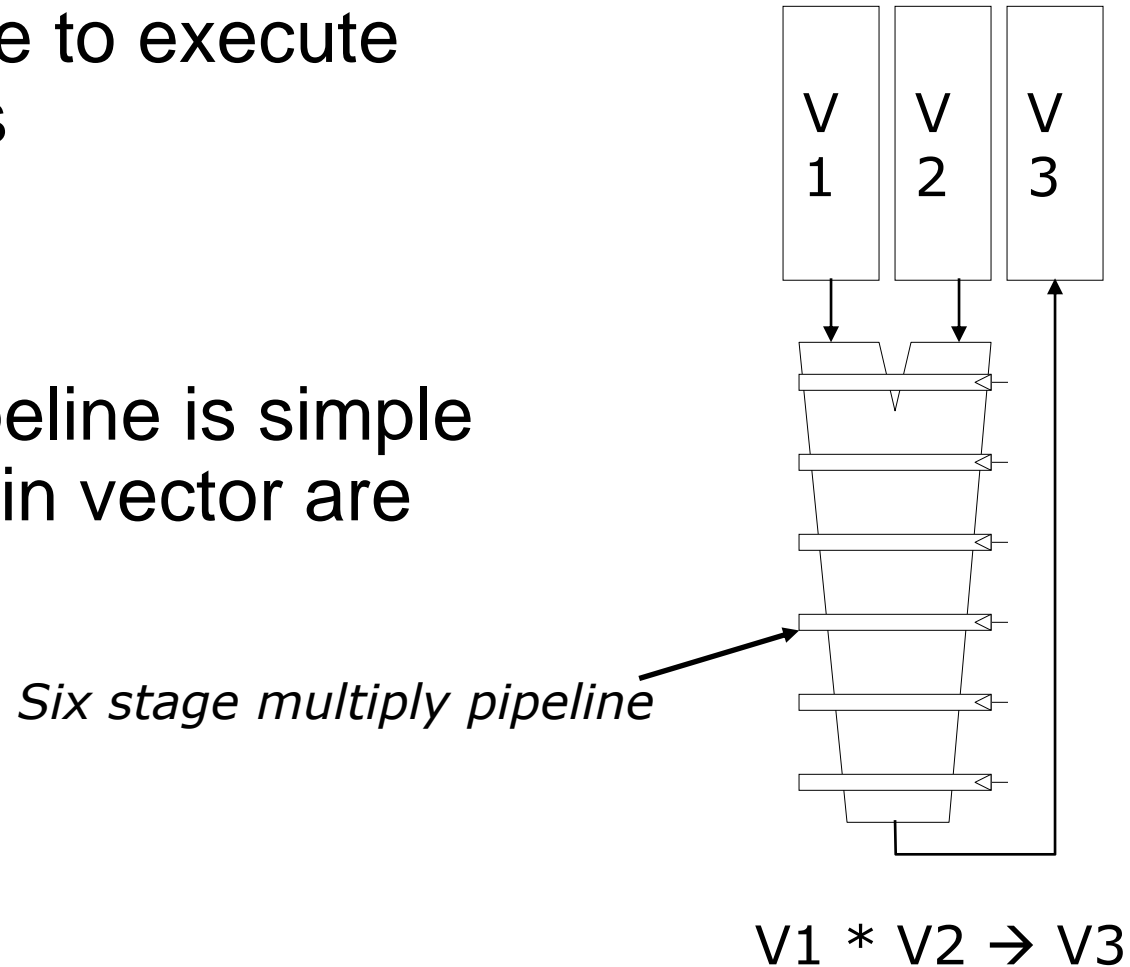
- Each **vector data register** holds N M-bit values
- **Vector control registers**: VLEN, VSTR, VMASK
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register
- **Vector Mask Register (VMASK)**
  - Indicates which elements of vector to operate on
  - Set by vector test instructions
    - e.g.,  $VMASK[i] = (V_k[i] == 0)$





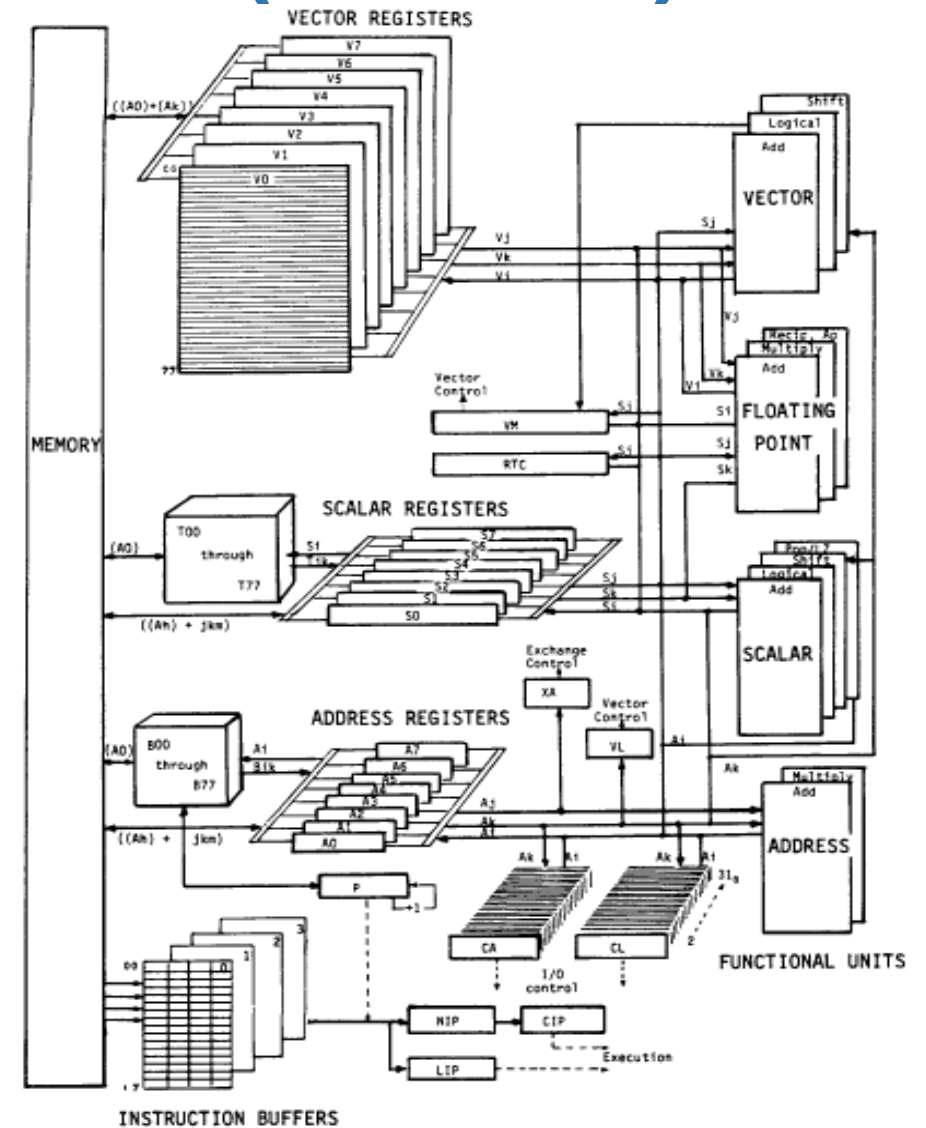
# Vector Functional Units

- Use a deep pipeline to execute element operations  
→ fast clock cycle
- Control of deep pipeline is simple because elements in vector are independent



# Vector Machine Organization (CRAY-1)

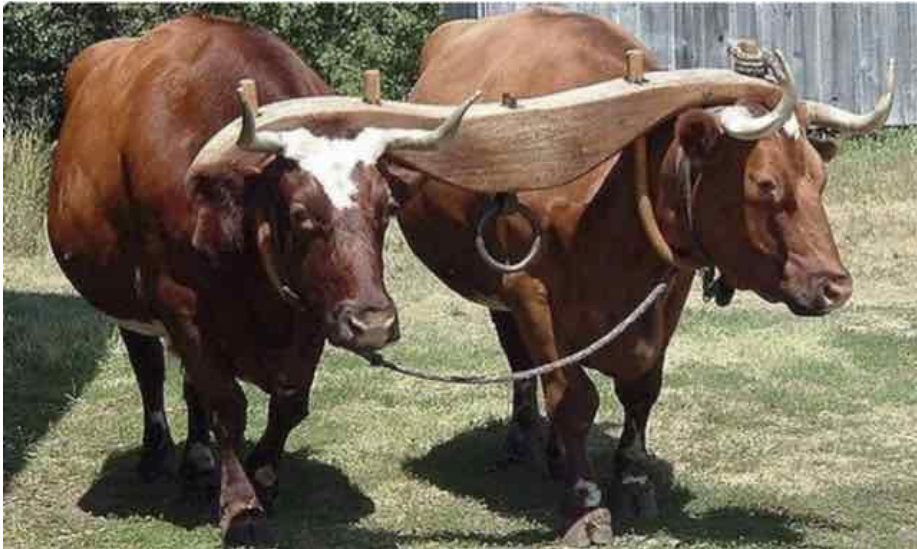
- CRAY-1
- Russell, “The CRAY-1 computer system,” CACM 1978.
- Scalar and vector modes
- 8 64-element vector registers
- 64-bits per element
- 16 memory banks
- 8 64-bit scalar registers
- 8 24-bit address registers



# Seymour Cray, the Father of Supercomputers



"If you were plowing a field, which would you rather use: **Two strong oxen** or **1024 chickens**?"

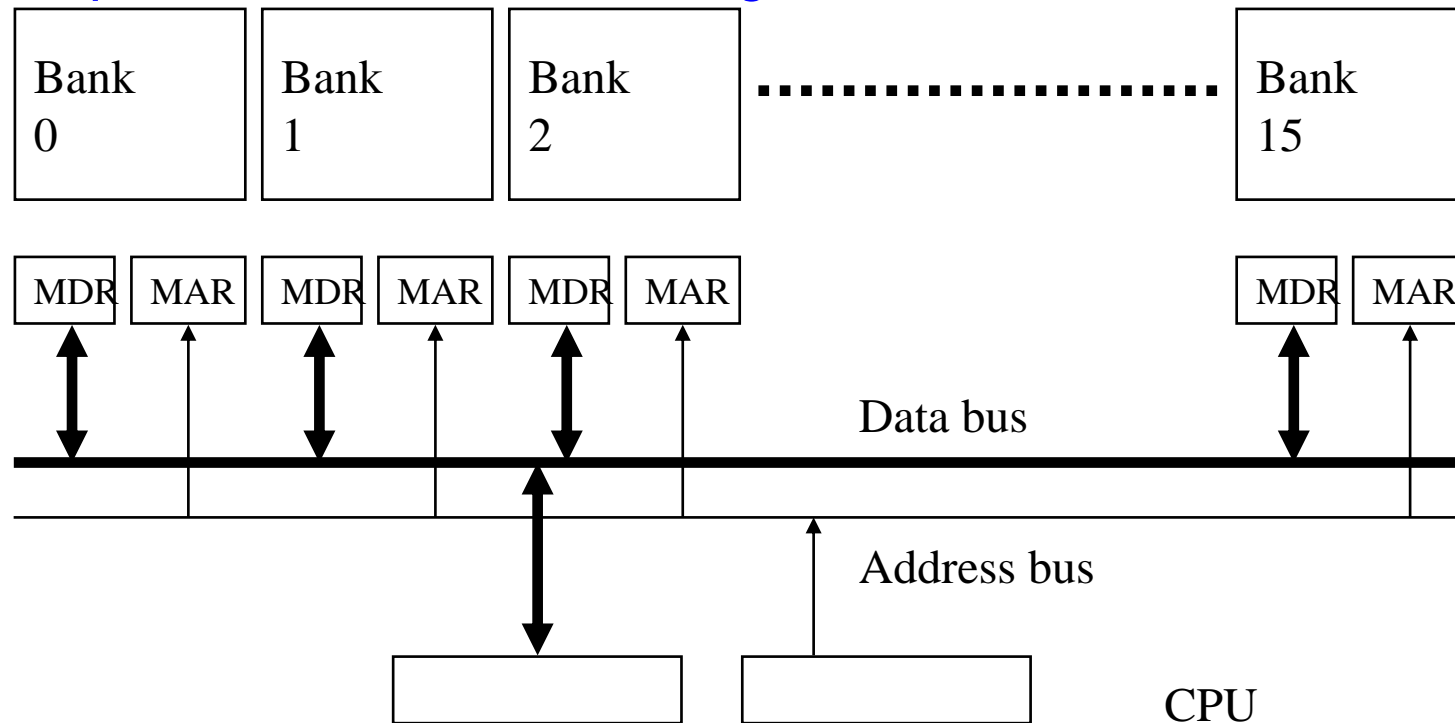


# Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements
- Elements separated from each other by a constant distance (stride)
  - Assume stride = 1 for now
- Elements can be loaded in consecutive cycles if we can start the load of one element per cycle
  - Can sustain a throughput of one element per cycle
- Question: How do we achieve this with a memory that takes more than 1 cycle to access?
- Answer: **Bank** the memory; interleave the elements across banks

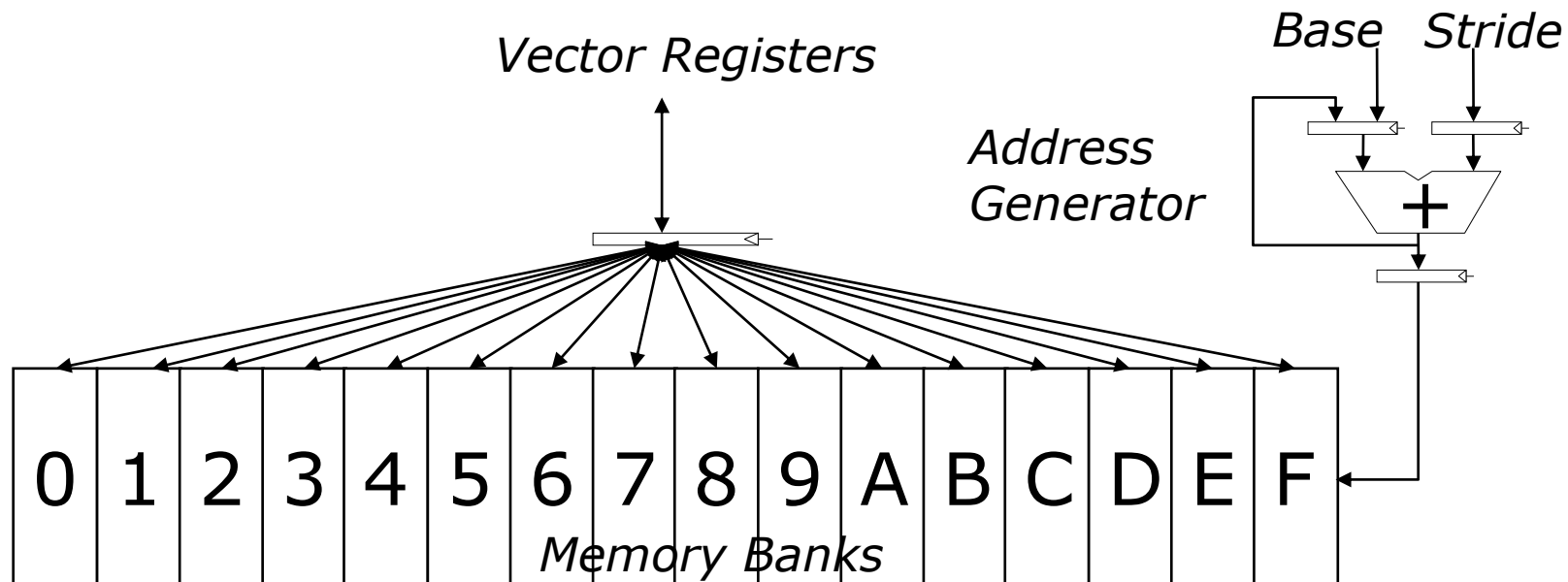
# Memory Banking

- Memory is divided into **banks** that can be accessed independently; banks share address and data buses (to minimize pin cost)
- Can start and complete one bank access per cycle
- Can sustain **N parallel accesses** if all N go to different banks



# Vector Memory System

- Next address = Previous address + Stride
- If (stride == 1) && (consecutive elements interleaved across banks) && (number of banks >= bank latency), then
  - we can sustain 1 element/cycle throughput



# Scalar Code Example: Element-Wise Avg.

- For  $i = 0$  to 49
  - $C[i] = (A[i] + B[i]) / 2$
- Scalar code (instruction and its latency)

MOVI R0 = 50	1	
MOVA R1 = A	1	304 dynamic instructions
MOVA R2 = B	1	
MOVA R3 = C	1	
X: LD R4 = MEM[R1++]	11	//autoincrement addressing
LD R5 = MEM[R2++]	11	
ADD R6 = R4 + R5	4	
SHFR R7 = R6 >> 1	1	
ST MEM[R3++] = R7	11	
DECBNZ R0, X	2	//decrement and branch if NZ

# Scalar Code Execution Time (In Order)

- Scalar execution time on an in-order processor with 1 bank
  - First two loads in the loop cannot be pipelined:  $2 \cdot 11$  cycles
  - $4 + 50 \cdot 40 = 2004$  cycles
- Scalar execution time on an in-order processor with 16 banks (word-interleaved: consecutive words are stored in consecutive banks)
  - First two loads in the loop can be pipelined
  - $4 + 50 \cdot 30 = 1504$  cycles
- Why 16 banks?
  - 11-cycle memory access latency
  - Having 16 ( $>11$ ) banks ensures there are enough banks to overlap enough memory operations to cover memory latency



# Vectorizable Loops

- A loop is **vectorizable** if each iteration is independent of any other

- For  $i = 0$  to 49

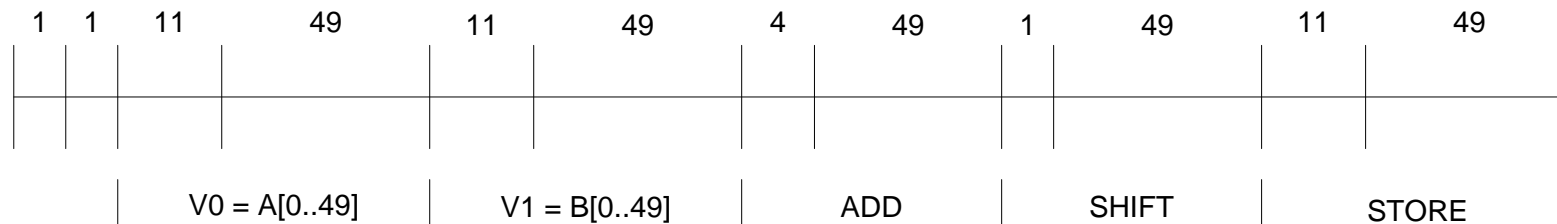
- $C[i] = (A[i] + B[i]) / 2$

- Vectorized loop (each instruction and its latency):

MOVI VLEN = 50	1	7 dynamic instructions
MOVI VSTR = 1	1	
VLD V0 = A	$11 + \text{VLEN} - 1$	
VLD V1 = B	$11 + \text{VLEN} - 1$	
VADD V2 = V0 + V1	$4 + \text{VLEN} - 1$	
VSHFR V3 = V2 >> 1	$1 + \text{VLEN} - 1$	
VST C = V3	$11 + \text{VLEN} - 1$	

# Basic Vector Code Performance

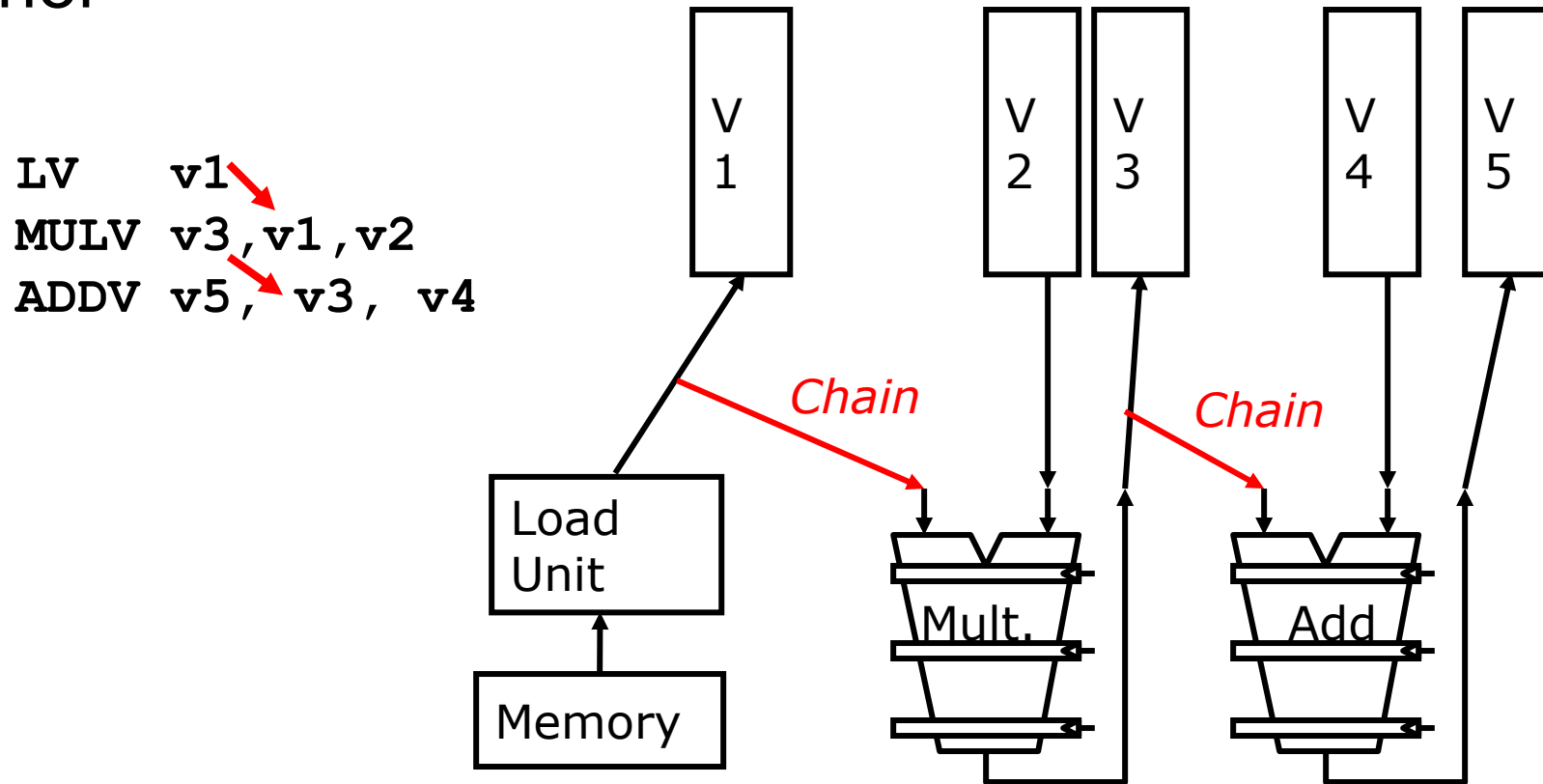
- Assume **no chaining** (no vector data forwarding)
  - i.e., output of a vector functional unit cannot be used as the direct input of another
  - **The entire vector register needs to be ready** before any element of it can be used as part of another operation
- One memory port (one address generator)
- 16 memory banks (word-interleaved)



- 285 cycles

# Vector Chaining

- **Vector chaining:** Data forwarding from one vector functional unit to another

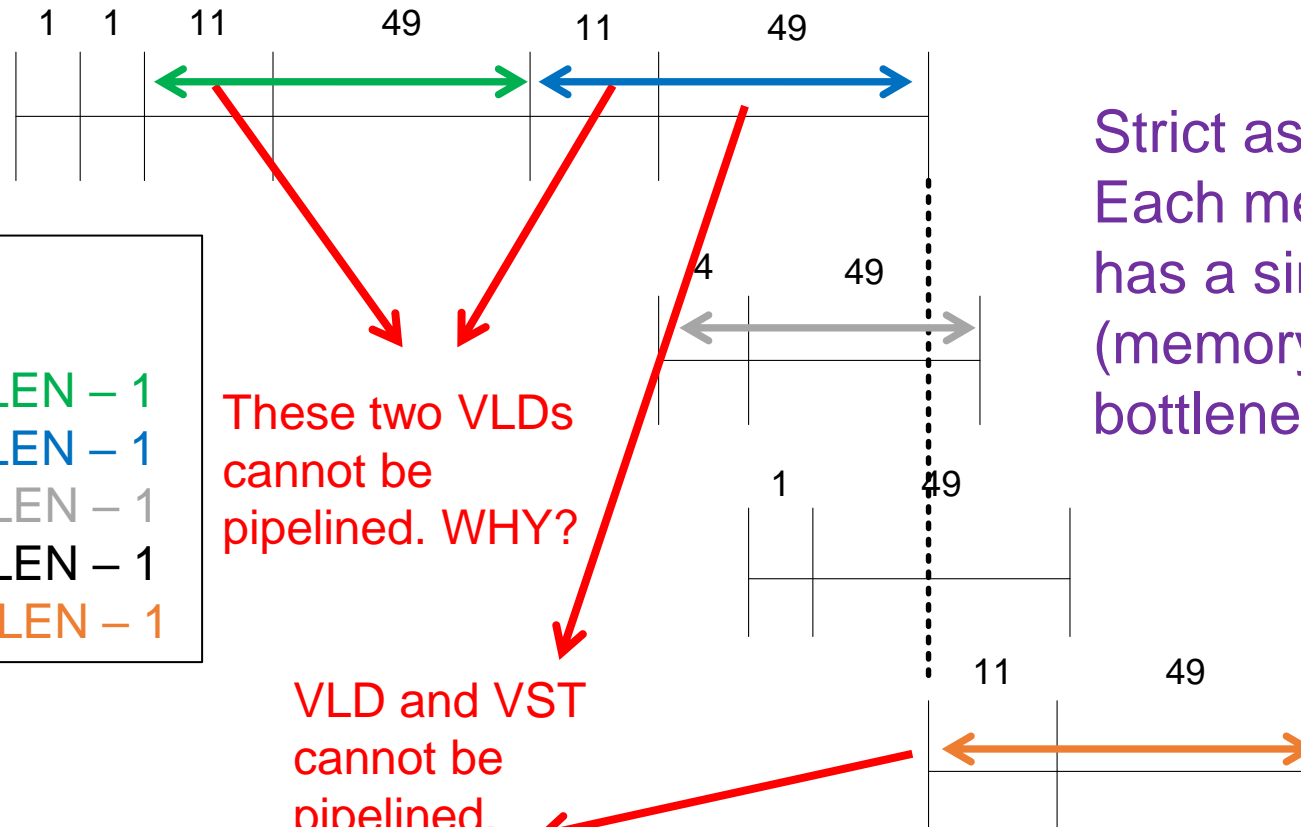


# Vector Code Performance - Chaining

- **Vector chaining:** Data forwarding from one vector functional unit to another

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	$11 + VLEN - 1$
VLD V1 = B	$11 + VLEN - 1$
VADD V2 = V0 + V1	$4 + VLEN - 1$
VSHFR V3 = V2 >> 1	$1 + VLEN - 1$
VST C = V3	$11 + VLEN - 1$

- 182 cycles



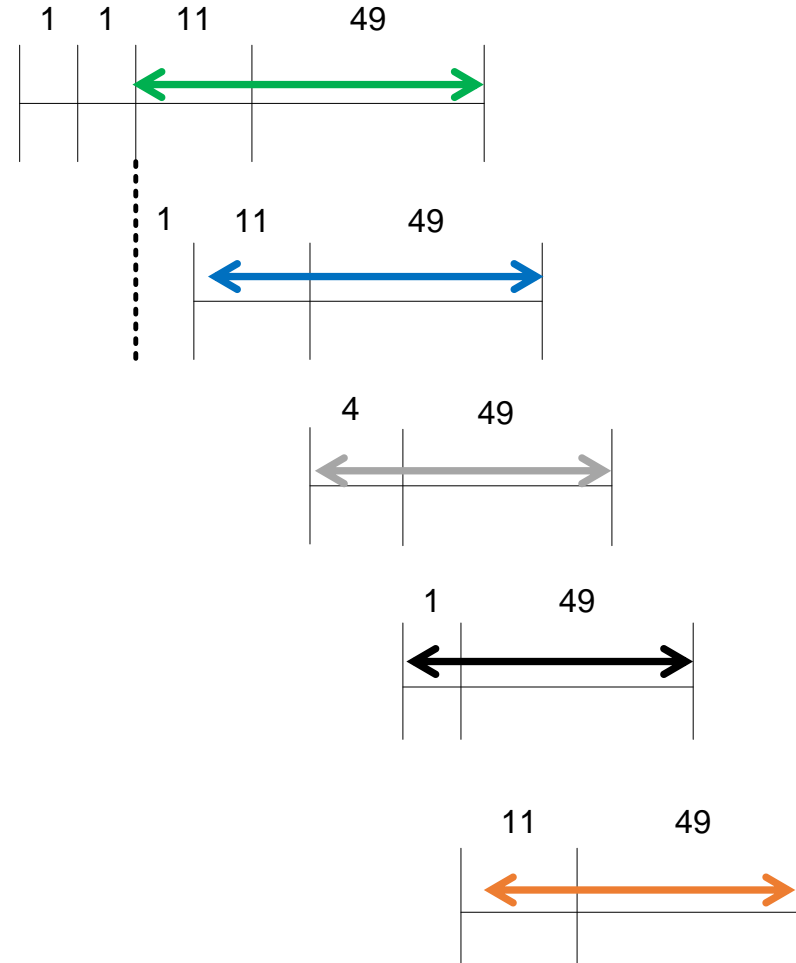
Strict assumption:  
Each memory bank has a single port  
(memory bandwidth bottleneck)

# Vector Code Performance – Multiple Memory Ports

- Chaining and 2 load ports, 1 store port in each bank

MOVI VLEN = 50	1
MOVI VSTR = 1	1
VLD V0 = A	11 + VLEN - 1
VLD V1 = B	11 + VLEN - 1
VADD V2 = V0 + V1	4 + VLEN - 1
VSHFR V3 = V2 >> 1	1 + VLEN - 1
VST C = V3	11 + VLEN - 1

- 79 cycles
- 19x perf. improvement!  
– was 1504 cycles for scalar



# Data Element Num vs. Max Vector Length

- What if # data elements  $>$  # elements in a vector register?
  - Idea: Break loops so that each iteration operates on # elements in a vector register
    - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where  $VLEN = 64$
    - 1 iteration where  $VLEN = 15$  (need to change value of  $VLEN$ )
  - Called **vector stripmining**

# Irregular Data Layout?

- What if vector data is not stored in a strided fashion in memory?  
(irregular memory access to a vector)
  - Idea: Use indirection to combine/pack elements into vector registers
  - Called scatter/gather operations

# Gather/Scatter Operations

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
LV vD, rD          # Load indices in D vector  
LVI vC, rC, vD     # Load indirect from rC base  
LV vB, rB          # Load B vector  
ADDV.D vA, vB, vC  # Do add  
SV vA, rA          # Store result
```



# Gather/Scatter Operations

- Gather/scatter operations often implemented in hardware to handle **sparse vectors (matrices)**
- Vector loads and stores use an index vector which is added to the base register to generate the addresses
- *Scatter* example

Index Vector	Data Vector (to Store)	Stored Vector (in Memory)	
0	3.14	Base+0	3.14
2	6.50	Base+1	X
6	71.20	Base+2	6.50
7	2.71	Base+3	X
		Base+4	X
		Base+5	X
		Base+6	71.20
		Base+7	2.71

# Conditional Operations in a Loop

- What if some operations should not be executed on a vector (based on a dynamically-determined condition)?

```
loop:    for (i=0; i<N; i++)  
        if (a[i] != 0) then b[i]=a[i]*b[i]
```

- Idea: **Masked operations**

- VMASK register is a bit mask determining which data element should not be acted upon

VLD V0 = A

VLD V1 = B

VMASK = (V0 != 0)

VMUL V1 = V0 \* V1

VST B = V1

- This is **predicated execution**. Execution is *predicated* on mask bit.

# Another Example with Masking

```
for (i = 0; i < 64; ++i)
  if (a[i] >= b[i])
    c[i] = a[i]
  else
    c[i] = b[i]
```

Steps to execute the loop in SIMD code

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

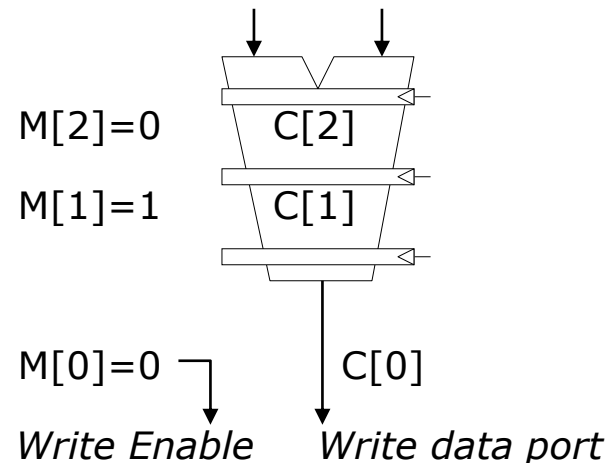
A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

# Masked Vector Instructions

## Simple Implementation

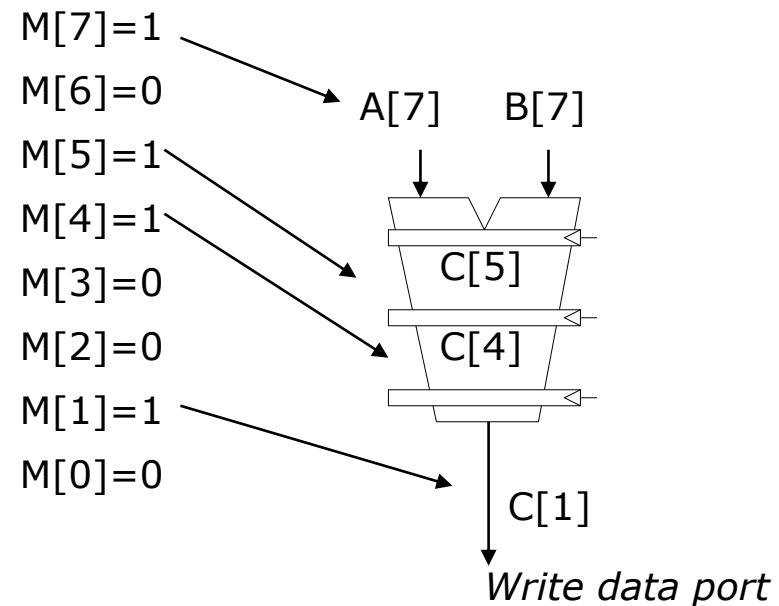
- execute all N operations, turn off result writeback according to mask

M[7]=1 A[7] B[7]  
M[6]=0 A[6] B[6]  
M[5]=1 A[5] B[5]  
M[4]=1 A[4] B[4]  
M[3]=0 A[3] B[3]



## Density-Time Implementation

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



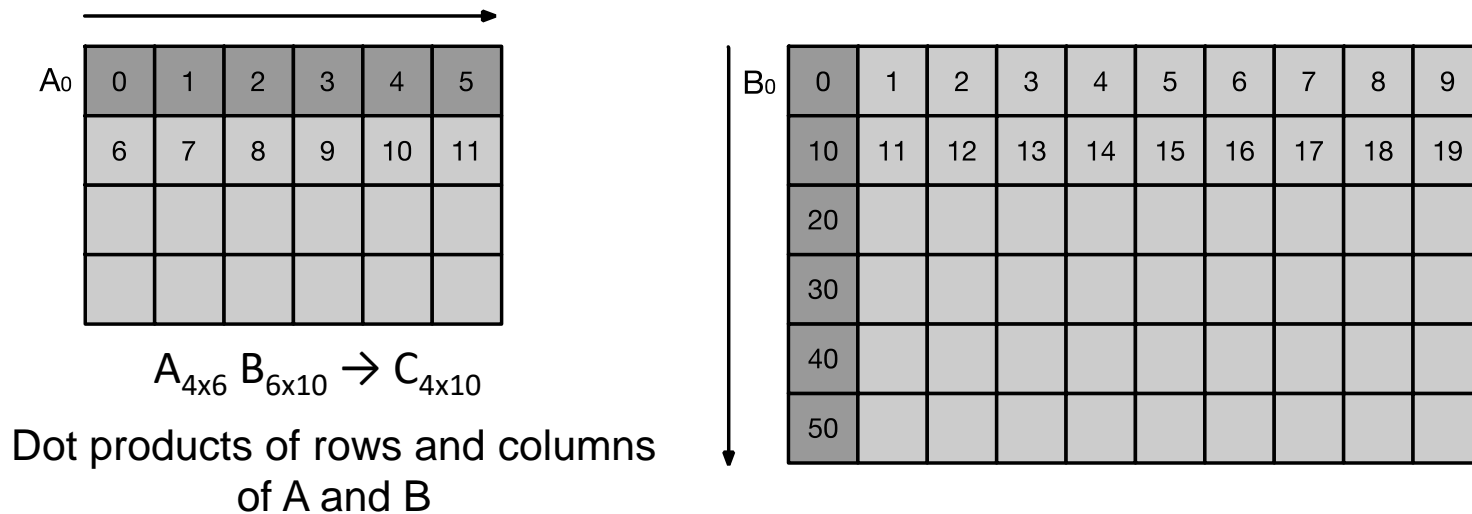
Which one is better?  
Tradeoffs?

# Some Issues

- Stride and banking
  - As long as they are *relatively prime* to each other and there are enough banks to cover bank access latency, we can sustain 1 element/cycle throughput
- Storage of a matrix
  - **Row major**: Consecutive elements in a row are laid out consecutively in memory
  - **Column major**: Consecutive elements in a column are laid out consecutively in memory
  - You need to change the stride when accessing a row versus column

# Matrix Multiplication

- A and B, both in row-major order



- A: Load  $A_0$  into vector register  $V_1$ 
  - Each time, increment address by one to access the next column
  - Accesses have a **stride of 1**
- B: Load  $B_0$  into vector register  $V_2$ 
  - Each time, increment address by 10
  - Accesses have a **stride of 10**

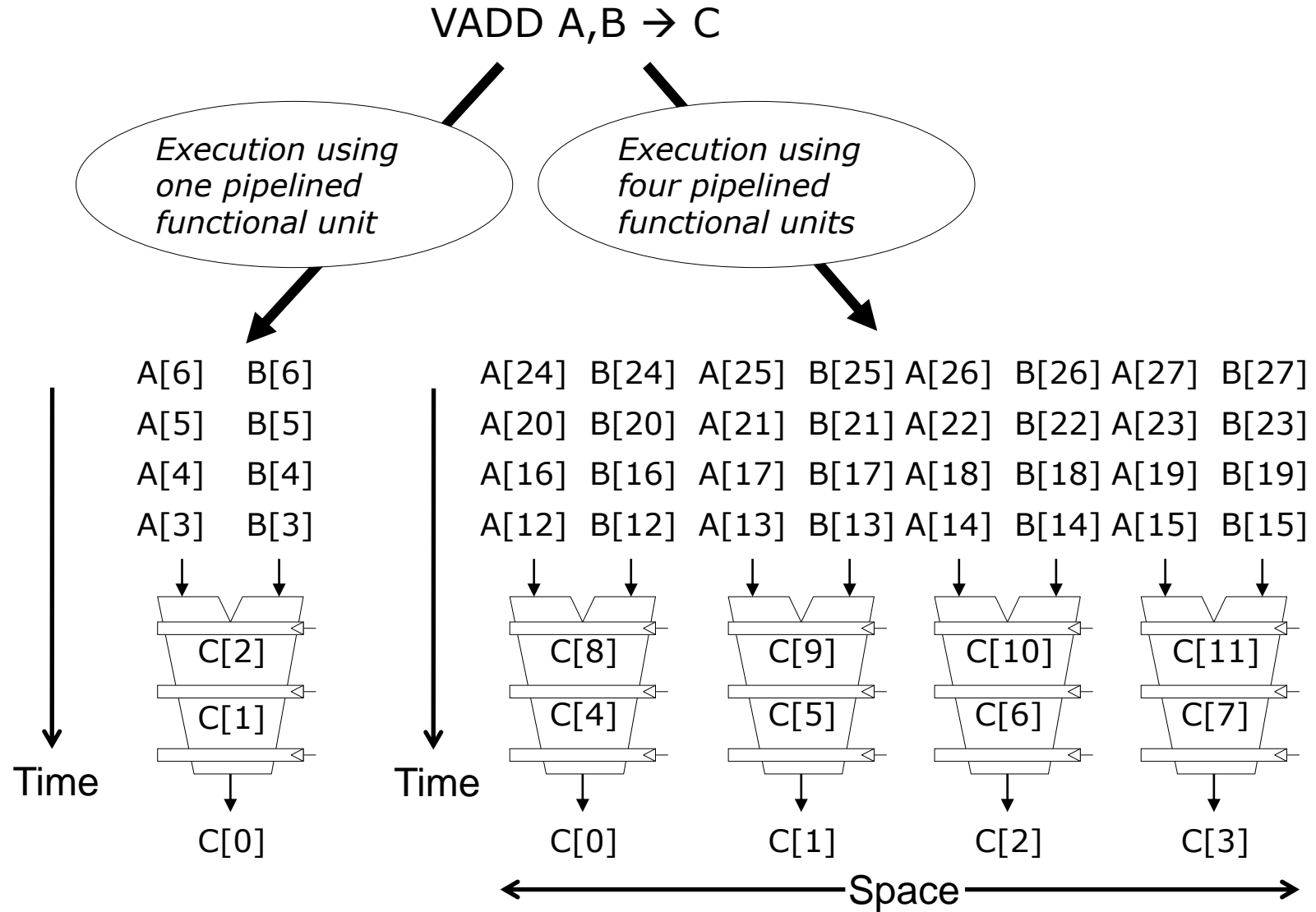
Different strides can lead to **bank conflicts**

How do we minimize them?

# Minimizing Bank Conflicts

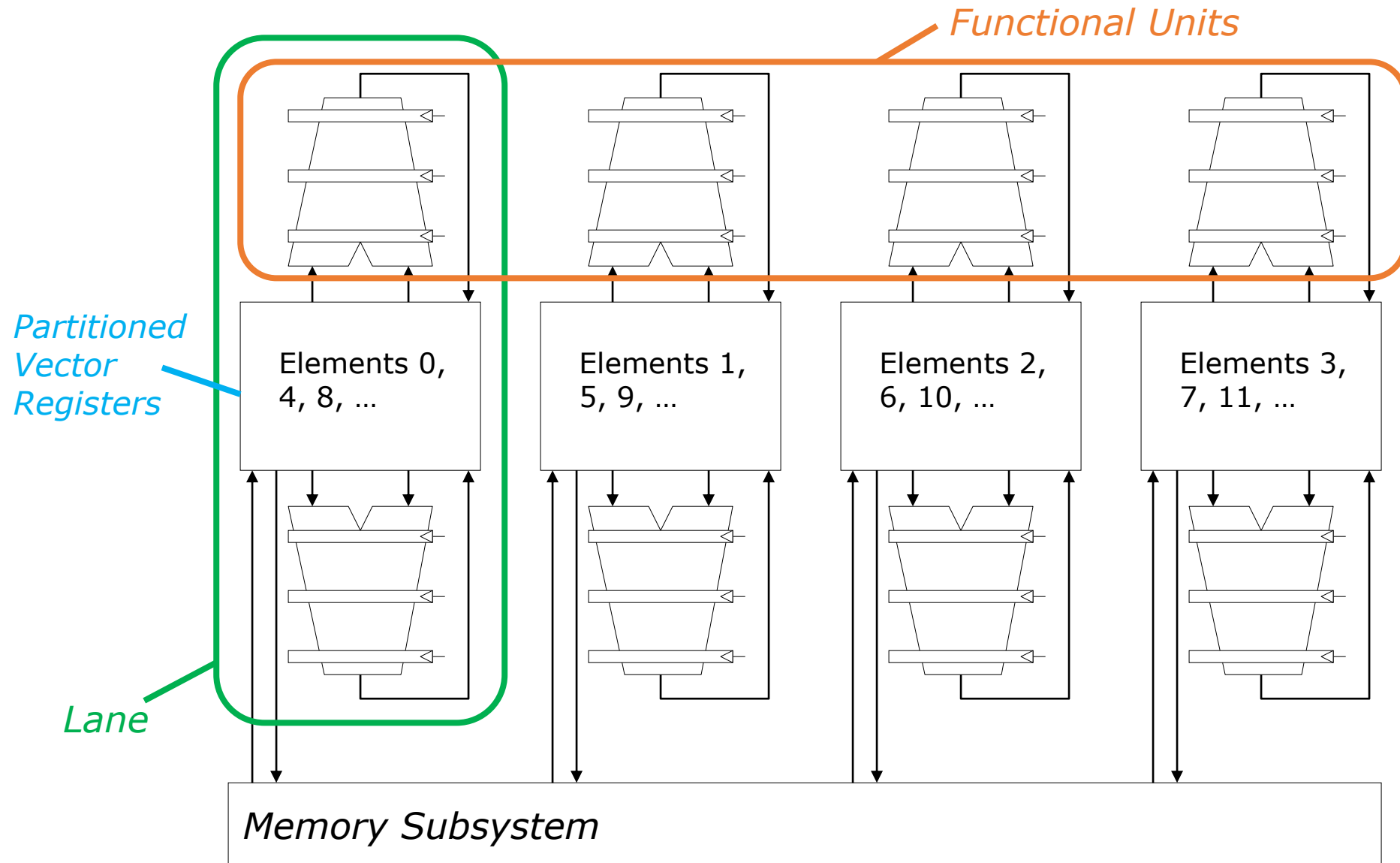
- More banks
- Better data layout to match the access pattern
  - Is this always possible?
- Better mapping of address to bank
  - E.g., randomized mapping
  - Rau, “Pseudo-randomly interleaved memory,” ISCA 1991.

# Vector Instruction Execution





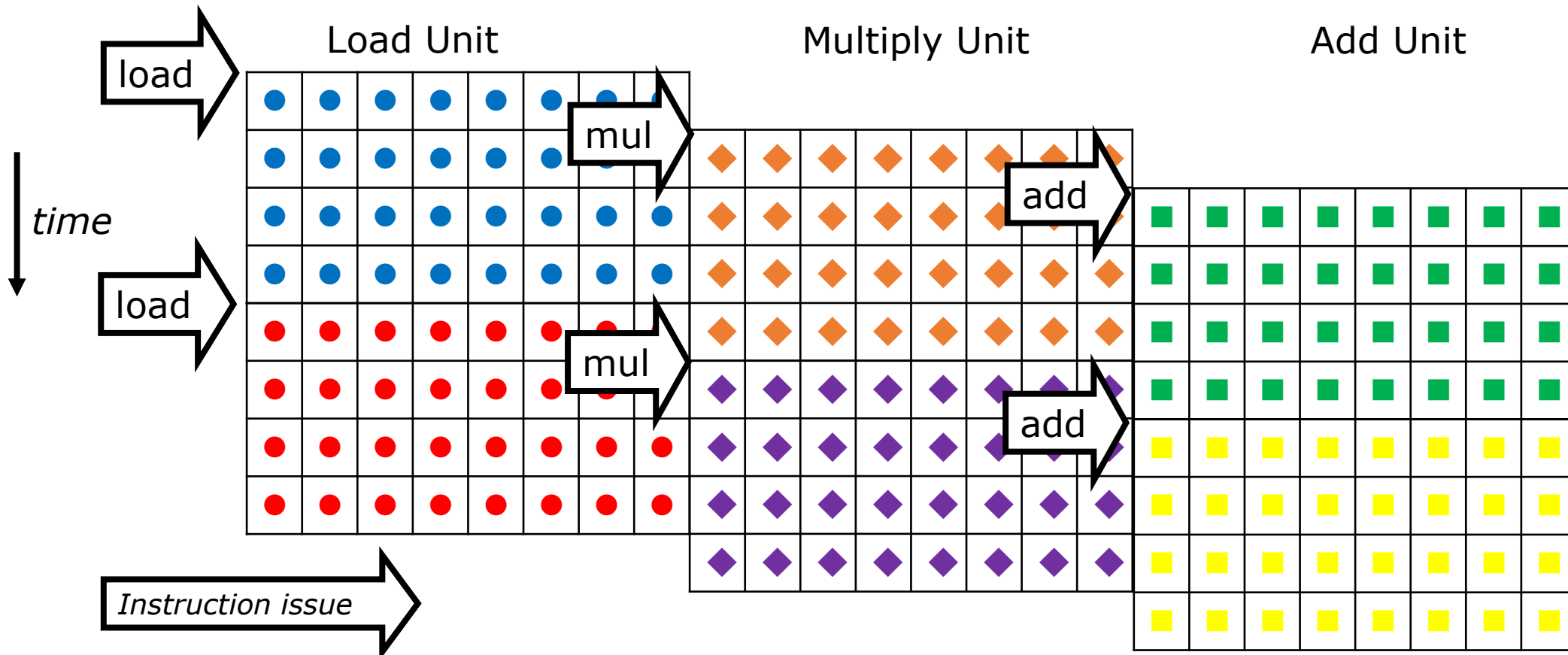
# Vector Unit Structure



# Vector Instruction Level Parallelism

Can overlap execution of multiple vector instructions

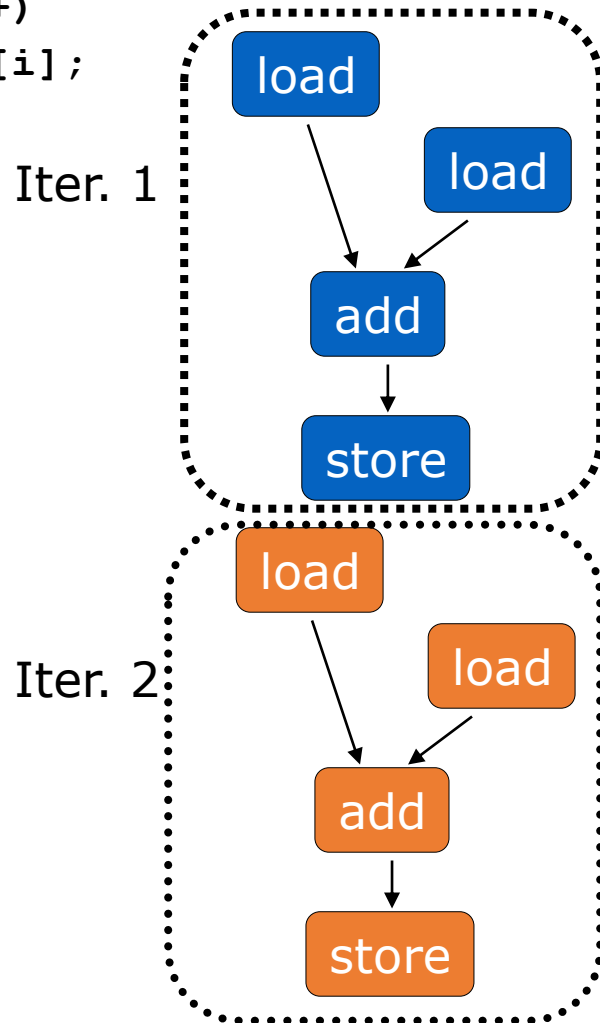
- Example machine has 32 elements per vector register and 8 lanes
- Completes 24 operations/cycle while issuing 1 vector instruction/cycle



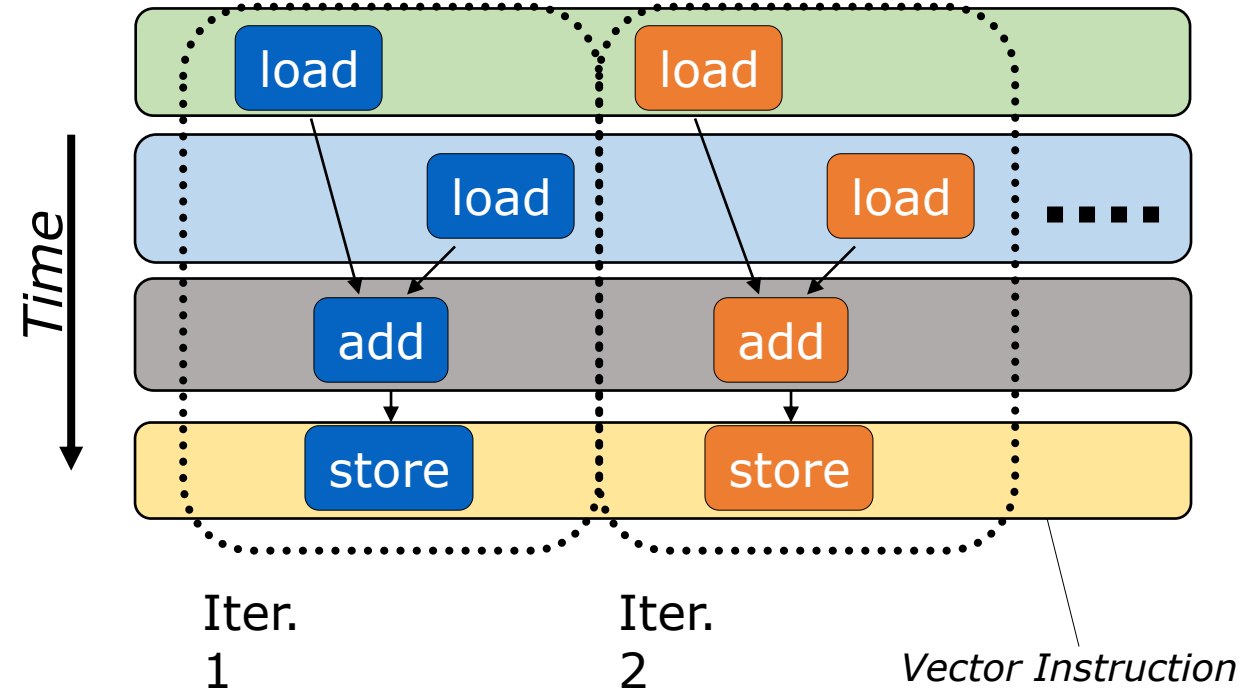
# Automatic Code Vectorization

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

*Scalar Sequential Code*



*Vectorized Code*



Vectorization is a compile-time reordering of operation sequencing  $\Rightarrow$  requires extensive loop dependence analysis

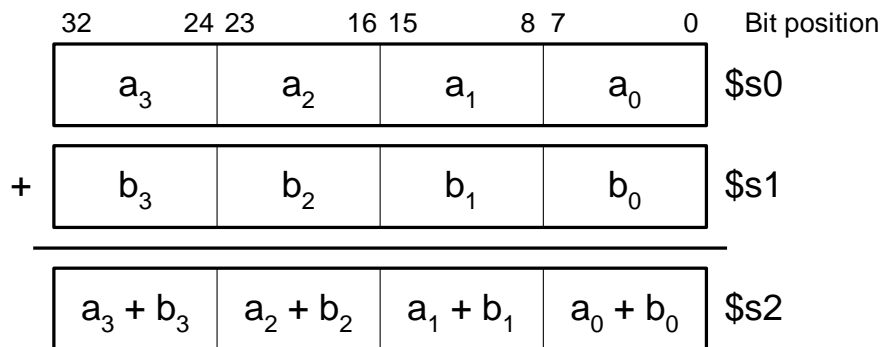
# Vector/SIMD Processing Summary

- Vector/SIMD machines are good at exploiting **regular data-level parallelism**
  - Same operation performed on many data elements
  - Improve performance, simplify design (no intra-vector dependencies)
- **Performance improvement limited by vectorizability** of code
  - Scalar operations limit vector machine performance
  - Remember **Amdahl's Law**
  - CRAY-1 was the fastest SCALAR machine at its time!
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec
  - ARM Advanced SIMD/NEON & SVE, RISC-V Vector Extension

# SIMD ISA Extensions

- Single Instruction Multiple Data (SIMD) extension instructions
  - Single instruction acts on multiple pieces of data at once
  - Common application: graphics
  - Perform short arithmetic operations (also called packed arithmetic)
- For example: add four 8-bit numbers
- Must modify ALU to eliminate carries between 8-bit values

```
padd8 $s2, $s0, $s1
```



# Intel Pentium MMX Operations

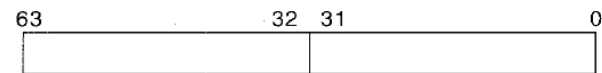
- Idea: One instruction operates on multiple data elements **simultaneously**
  - Designed with multimedia (graphics) operations in mind



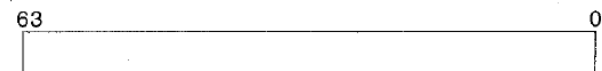
(a)



(b)



(c)



(d)

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 is always equal to 1.

Peleg and Weiser, “[MMX Technology Extension to the Intel Architecture](#),”  
IEEE Micro, 1996.

# Vector Extensions for ARM & RISC-V

- ARM Scalable Vector Extension (SVE)
  - <https://developer.arm.com/documentation/102476/0100/?lang=en>
  - [https://gitlab.com/arm-hpc/training/bsc\\_training\\_materials/-/blob/master/Slides/7%20-%20Vectorization%20with%20SVE.pptx](https://gitlab.com/arm-hpc/training/bsc_training_materials/-/blob/master/Slides/7%20-%20Vectorization%20with%20SVE.pptx)
- RISC-V Vector Extensions
  - <https://github.com/riscv/riscv-v-spec/blob/master/v-spec.adoc>
  - <https://github.com/riscv/riscv-v-spec/releases/download/v1.0/riscv-v-spec-1.0.pdf>