Vector Processors Architecutre

ENCS5331: Advanced Computer Architecture

Fall 2024/2025

Instructor: Dr. Ayman Hroub

Special thanks to Prof. Onur Mutlu (ETHZ) for part of the slides

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
  - Array processor
  - 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 and 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

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

Three Variations of SIMD

- Vector Processors
- Graphics Processing Units (GPUs)
- Multimedia SIMD instruction set extensions

Vector Processors (I)

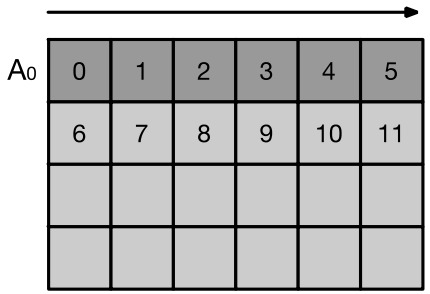
- A vector is a one-dimensional array of numbers
- Many scientific/commercial programs use vectors

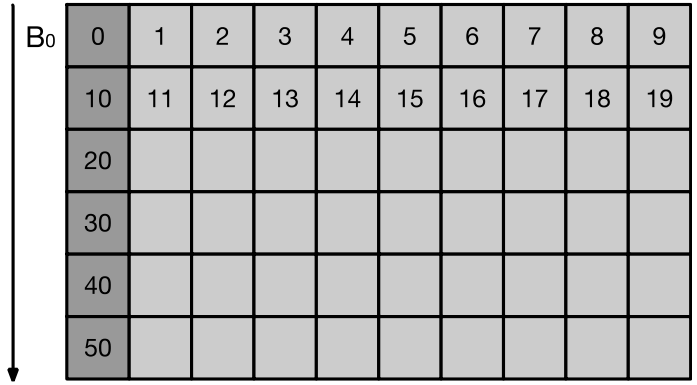
```
for (i = 0; i<=49; i++)
    C[i] = (A[i] + B[i]) / 2
```

- A vector processor is one whose instructions operate on vectors rather than scalar (single data) values
- A single instruction operates on multiple data elements in consecutive time steps using the same space (PE)
- 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 Stride Example: Matrix Multiply

- A and B matrices, both stored in memory in row-major order

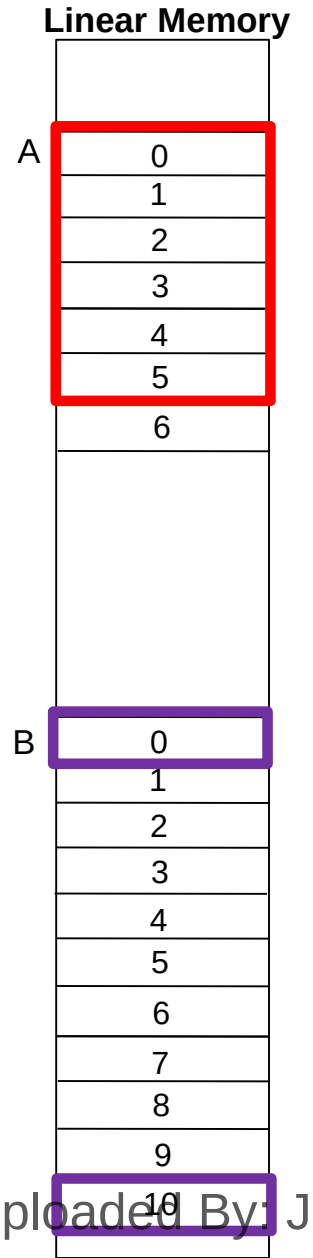




$A_{4x6}\ B_{6x10} \rightarrow C_{4x10}$

Dot product of each row vector of A with each column vector of B

- Load A's row 0 (A0 through A5) into vector register V_1
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1
- Load B's column 0 (B0 through B50) into vector register V_2
  - Each time, increment address by 10 to access the next row
  - Accesses have a stride of 10

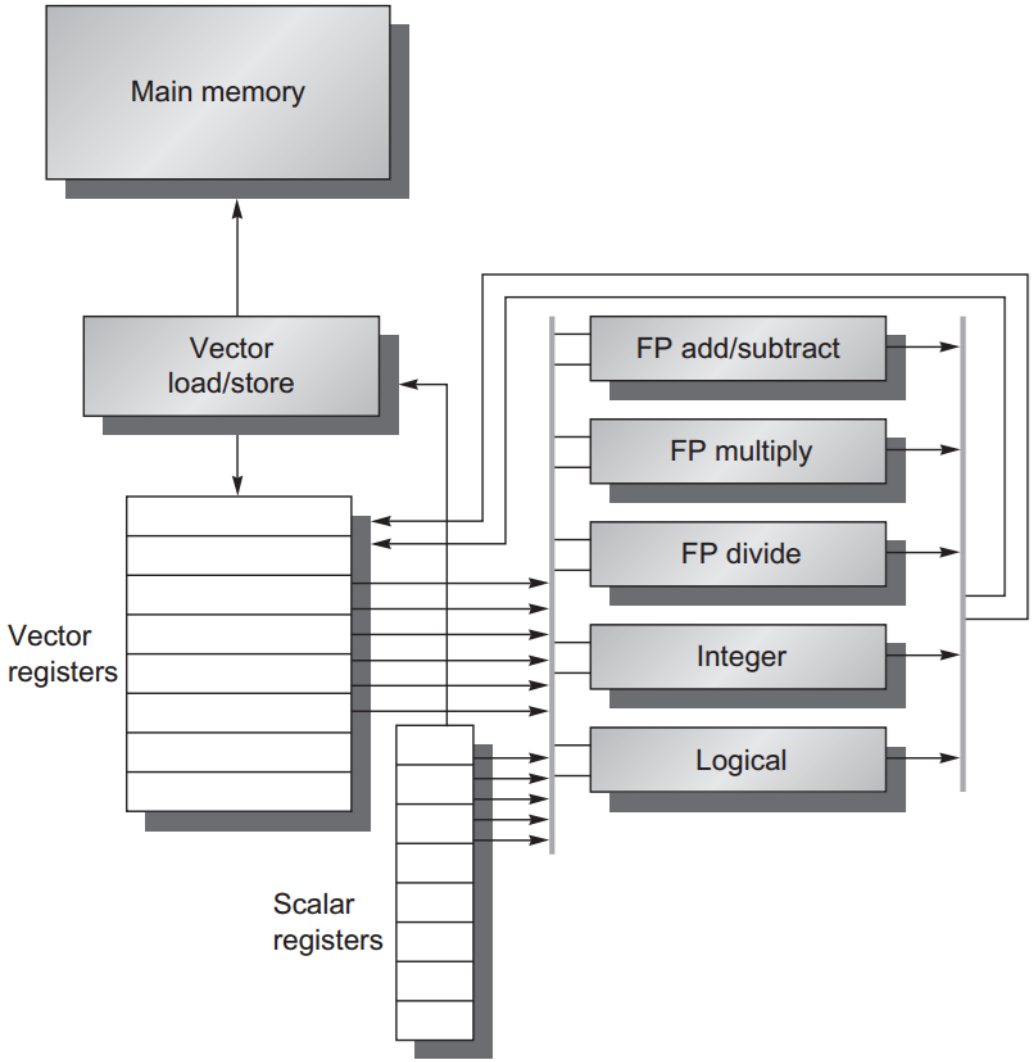


Vector Processors (II)

- 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

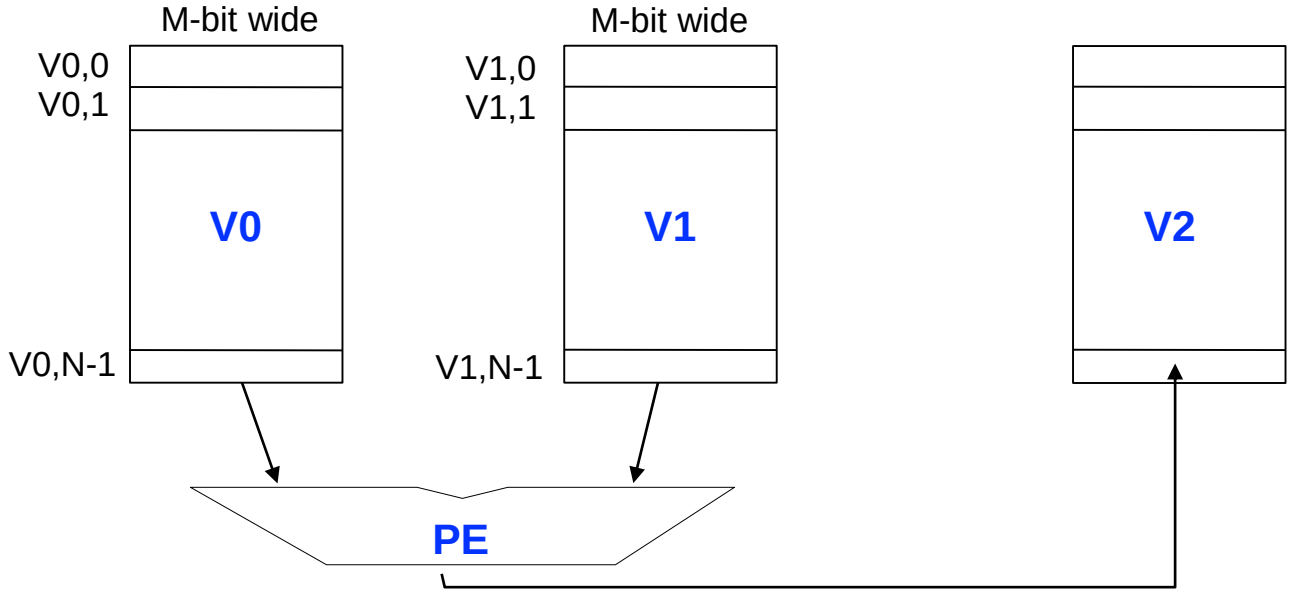
- Vector instructions allow deeper pipelines
  - No intra-vector dependencies → no hardware interlocking needed within a vector
  - No control flow within a vector

Vector Architecture Basic Strucutre



Storing Multiple Data Elements: Vector Registers

- Each vector data register holds N M-bit values
  - Each register stores a vector
  - Not a (single) scalar value as we saw before



Vector Processor Advantages

+ No dependencies within a vector
  - ❑ Pipelining & parallelization work really well
  - ❑ Can have very deep pipelines (without the penalty of deep pipelines)
+ Each instruction generates a lot of work (i.e., operations)
  - ❑ Reduces instruction fetch bandwidth requirements
  - ❑ Amortizes instruction fetch and control overhead over many data

    --> Leads to high energy efficiency per operation
+ No need to explicitly code loops
  - ❑ Fewer branches in the instruction sequence
+ Highly regular memory access pattern

Vector Processor Disadvantages

-- Works (only) if parallelism is regular (data/SIMD parallelism)

++ Vector operations

-- Very inefficient if parallelism is irregular

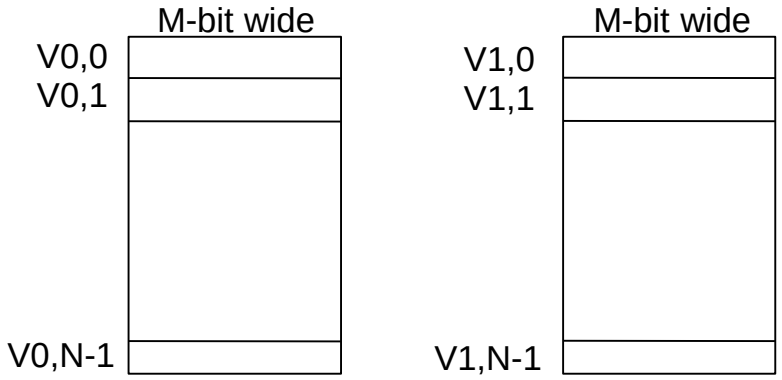
-- How about searching for a key in a linked list?

-- Memory (bandwidth) can easily become a bottleneck, especially if

1. compute/memory operation balance is not maintained
2. 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

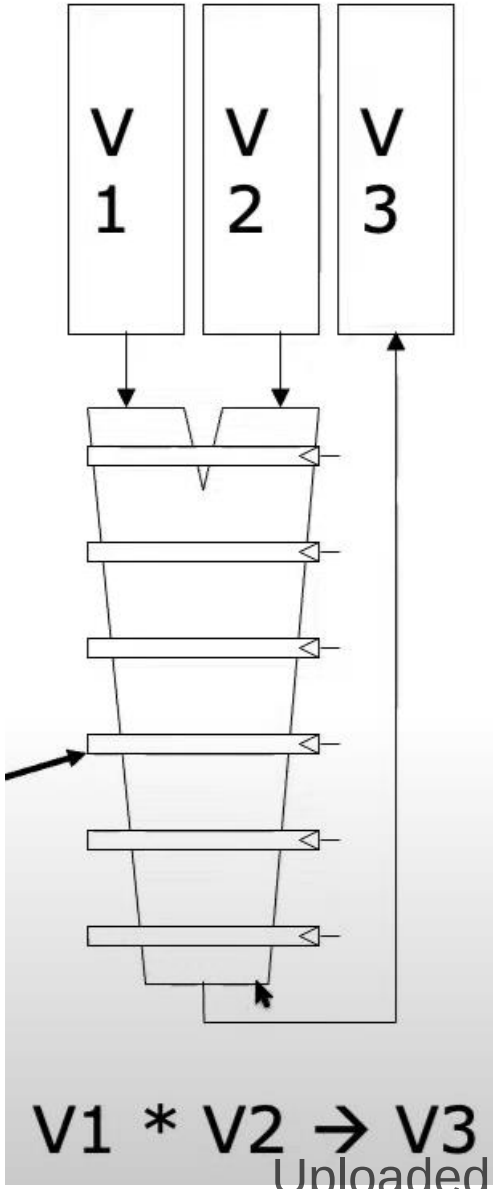


Vector Functional Units

- Use a deep pipeline to execute element operations

  → fast clock cycle

- Control of deep pipeline is simple because elements in vector are independent

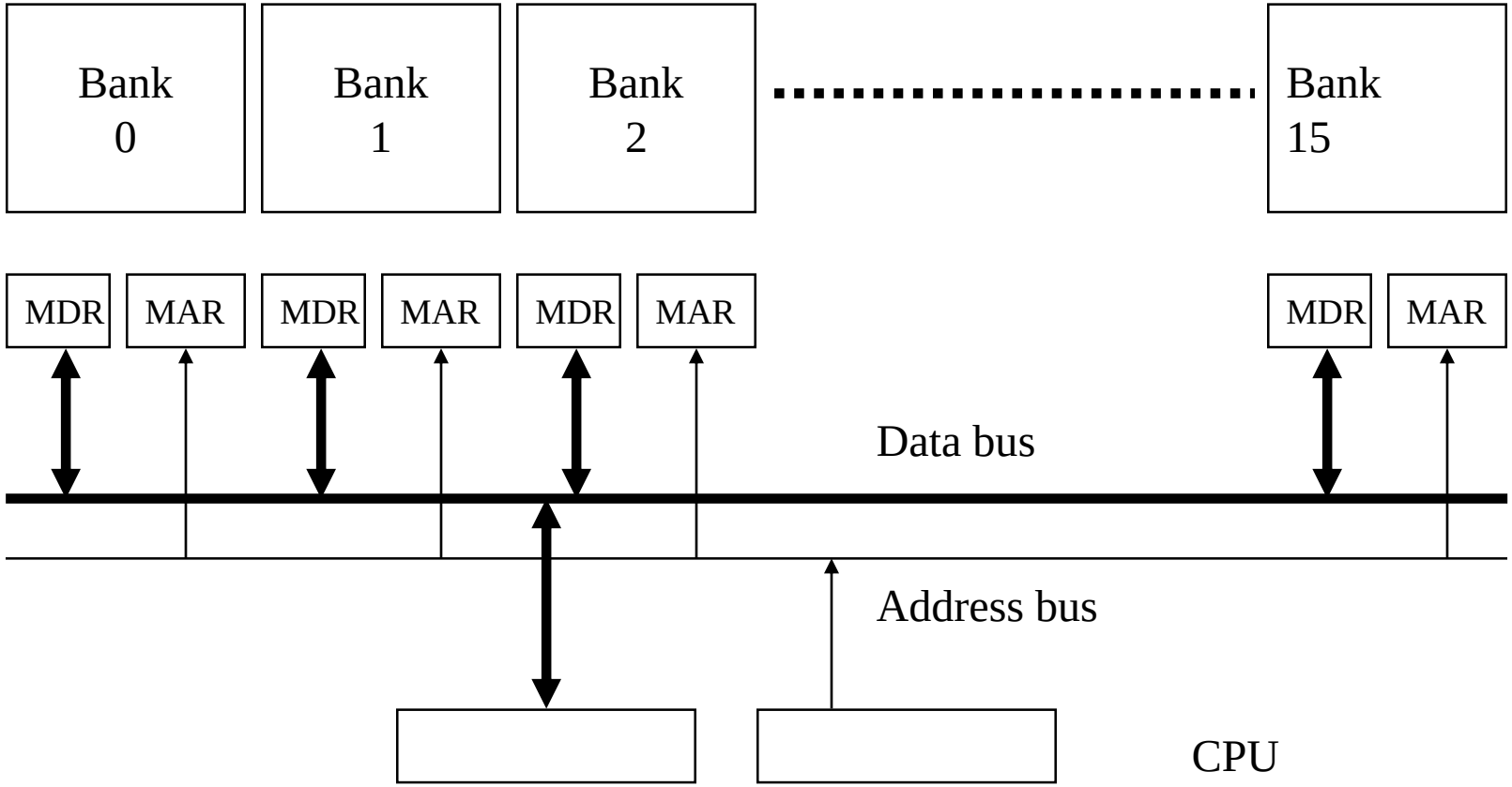


Six stage multiply pipeline

Slide credit: Krste Asanovic

Memory Banking

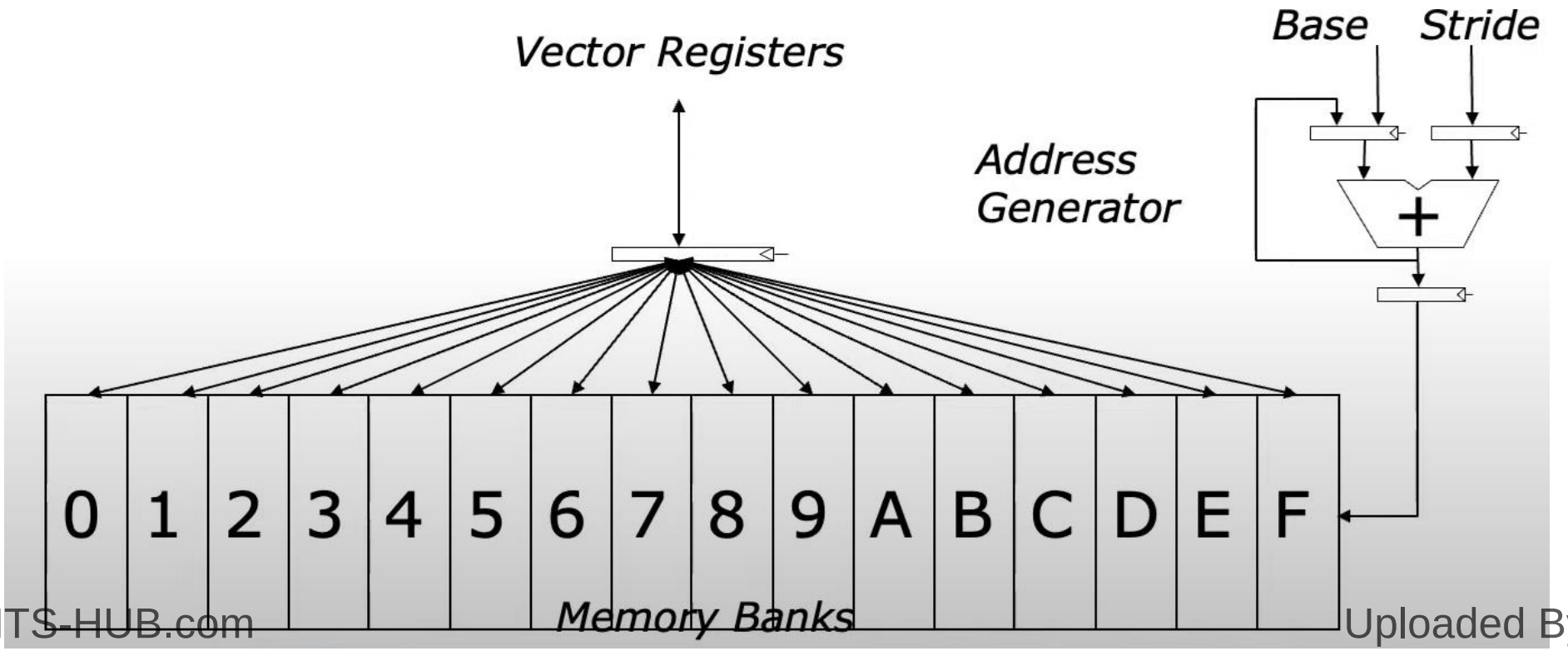
- Memory is divided into banks that can be accessed independently; banks share address and data buses (to reduce memory chip pins)
- Can start and complete one bank access per cycle
- Can sustain N concurrent accesses if all N go to different banks



Picture credit: Derek Chiou

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 in clock cycles)

```
    MOVI R0 = 50                1
    MOVA R1 = A                 1
    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
```

304 dynamic instructions

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*11 cycles
  - 4 + 50*40 = 2004 cycles

- Scalar execution time on an in-order processor with 1 bank with 2 memory ports (two different memory accesses can be serviced concurrently) OR 2 banks (where arrays A and B are stored in different banks)
  - First two loads in the loop can be pipelined: 1 + 11 cycles
  - 4 + 50*30 = 1504 cycles

Vectorizable Loops

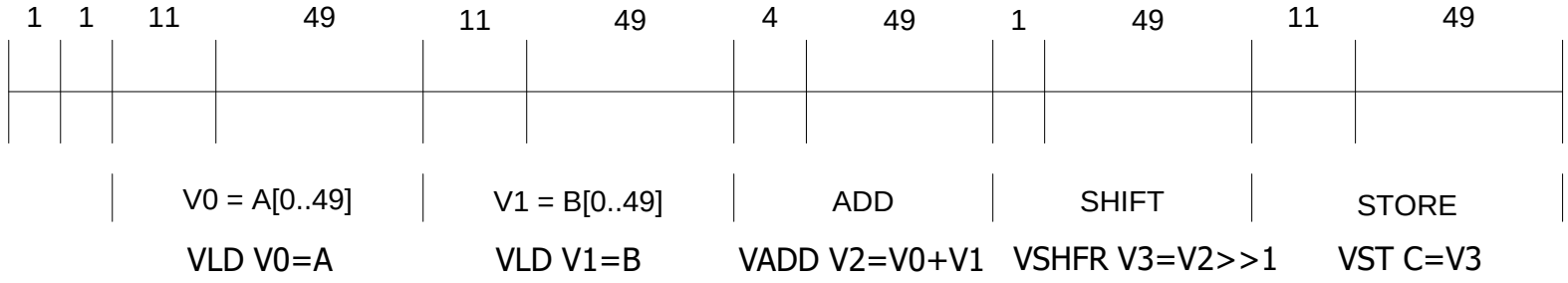
- A loop is vectorizable if each iteration is independent of any other
- For i = 0 to 49
  - C[i] = (A[i] + B[i]) / 2
- Vectorized loop (each instruction and its latency):

MOVI VLEN = 50	1
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

7 dynamic instructions

Basic Vector Code Performance (I)

- 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
- 1 memory port (one address generator) per bank
- 16 memory banks (word-interleaved: consecutive elements of an array are stored in consecutive banks)



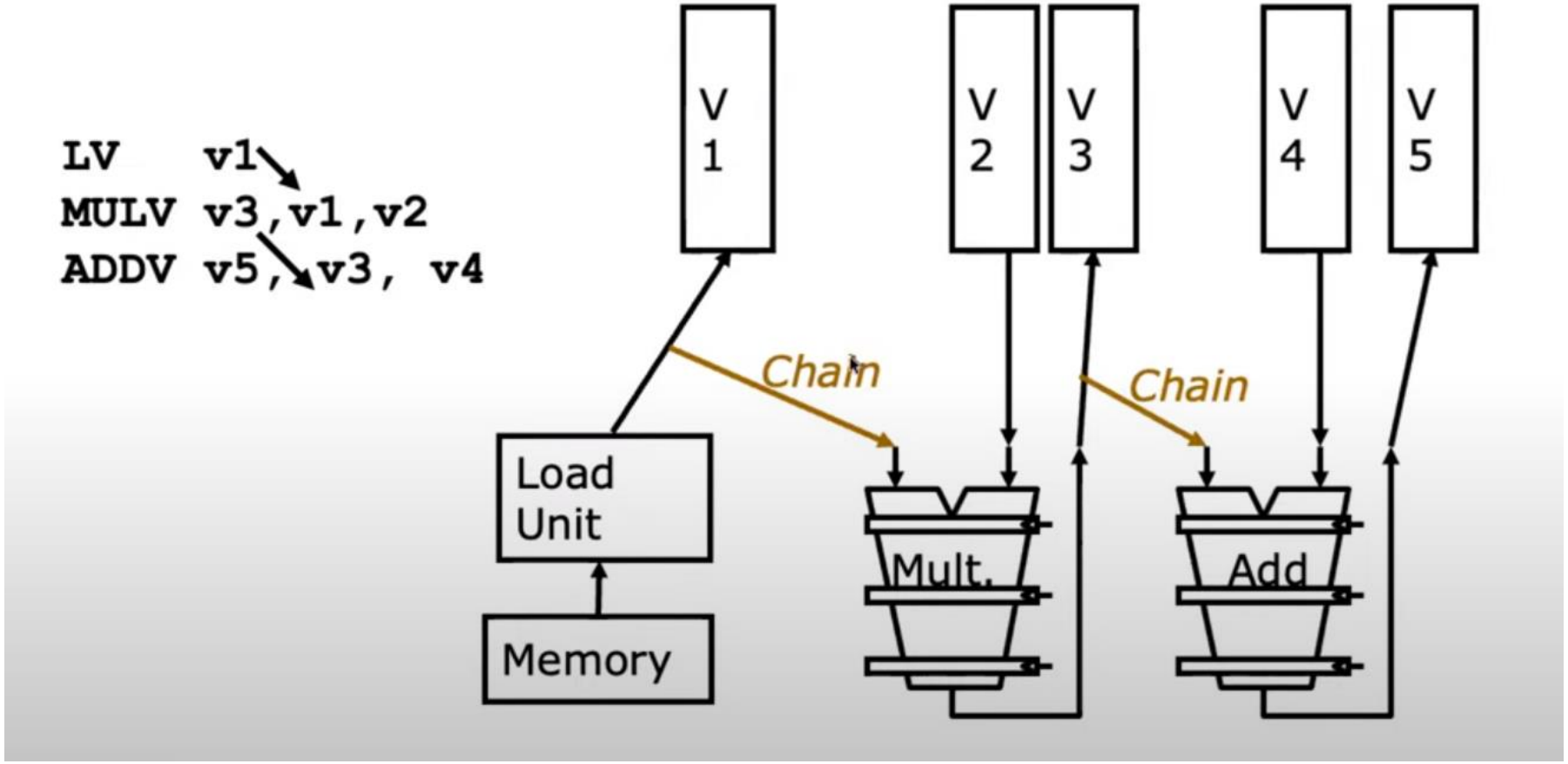
- 285 cycles

Basic Vector Code Performance (II)

- 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
- The above assumes a unit stride (i.e., stride = 1)
  - Correct for our example program
- What if stride > 1?
  - How do you ensure we can access 1 element per cycle when memory latency is 11 cycles?

Vector Chaining

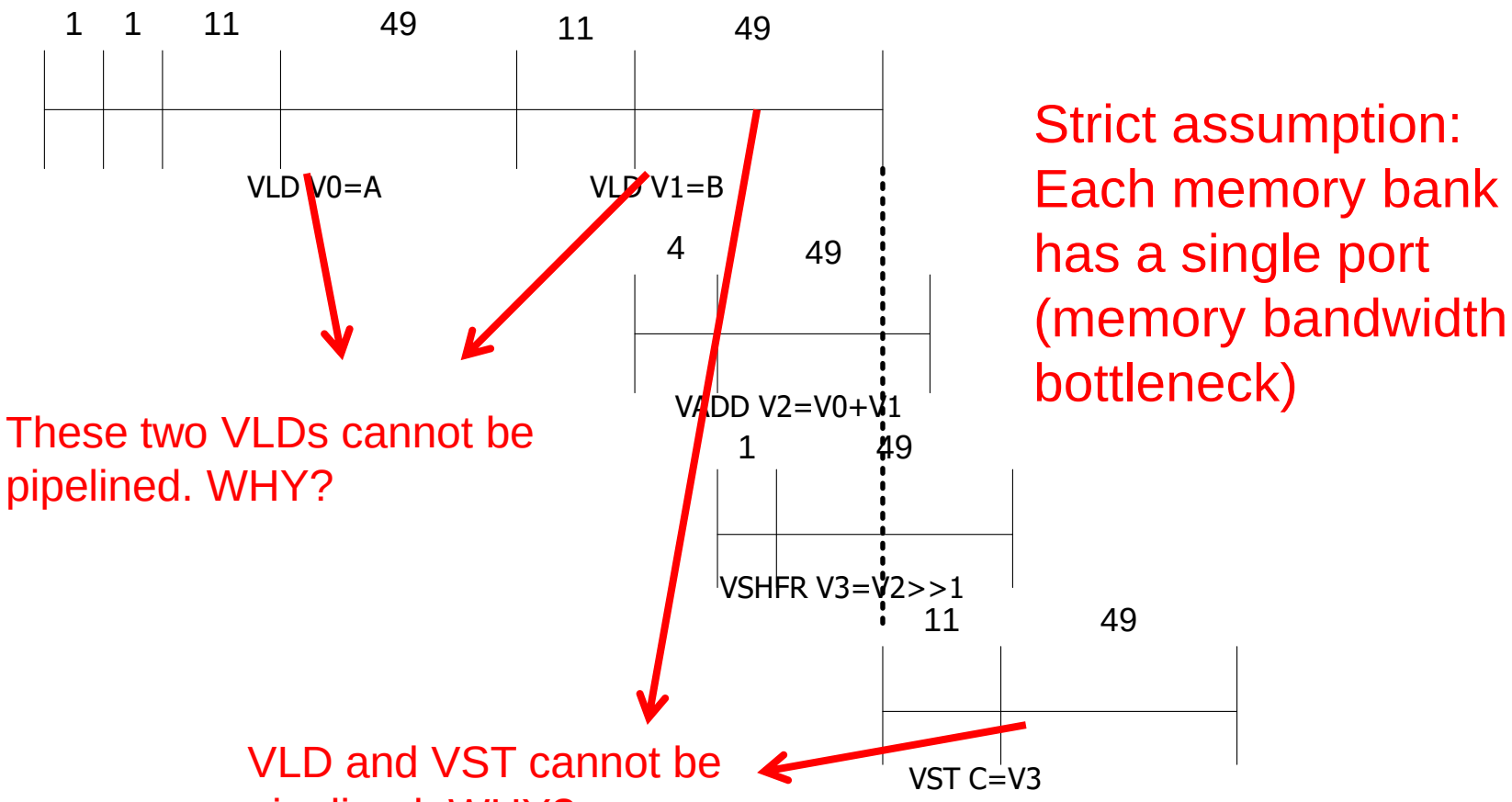
- Vector chaining: Data forwarding from one vector functional unit to another



Slide credit: Krste Asanovic

Vector Code Performance - Chaining

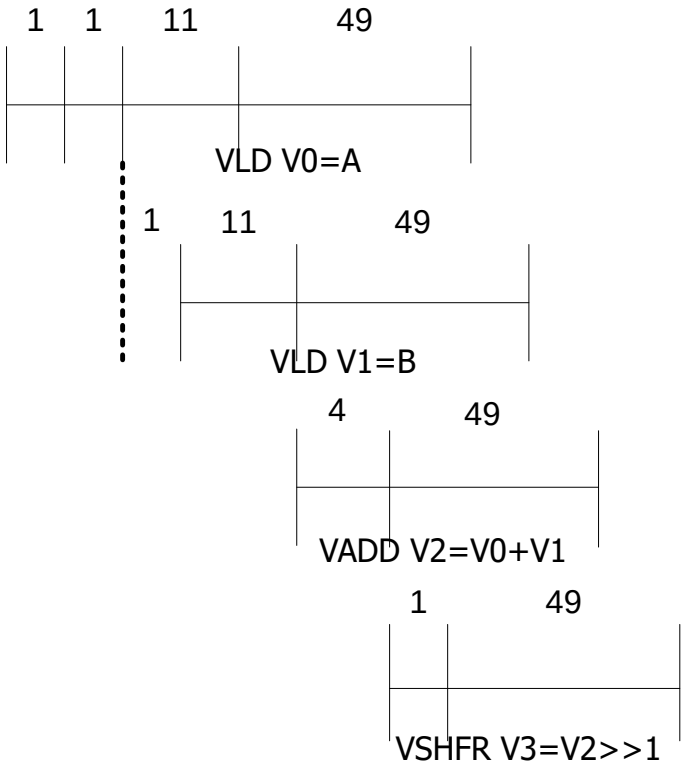
- Vector chaining: Data forwarding from one vector functional unit to another



- 182 cycles

Vector Code Performance – Multiple Memory Ports

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



- 79 cycles
- 19X perf. improvement!

Questions (I)

- 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

Questions (II)

- 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
  - Doing so also helps with avoiding useless computation on sparse vectors (i.e., vectors where many elements are 0)

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) or indirect indexing
- 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)
0	3.14
2	6.5
6	71.2
7	2.71

Stored Vector (in Memory)	
Base+0	3.14
Base+1	X
Base+2	6.5
Base+3	X
Base+4	X
Base+5	X
Base+6	71.2
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)?

```
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]
```

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

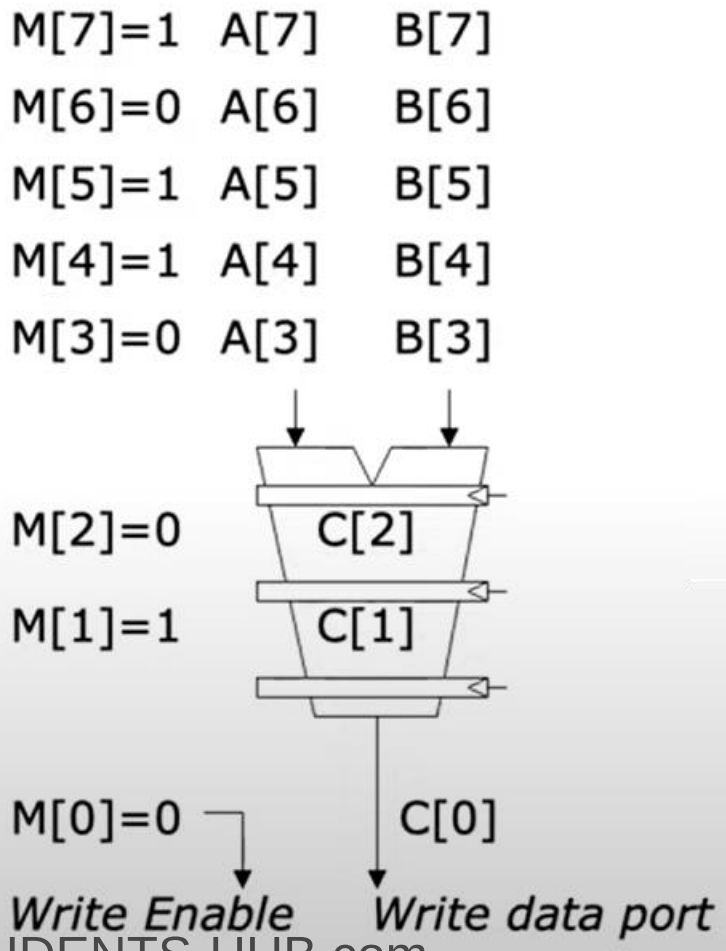
Steps to execute the loop in SIMD code

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

Masked Vector Instructions

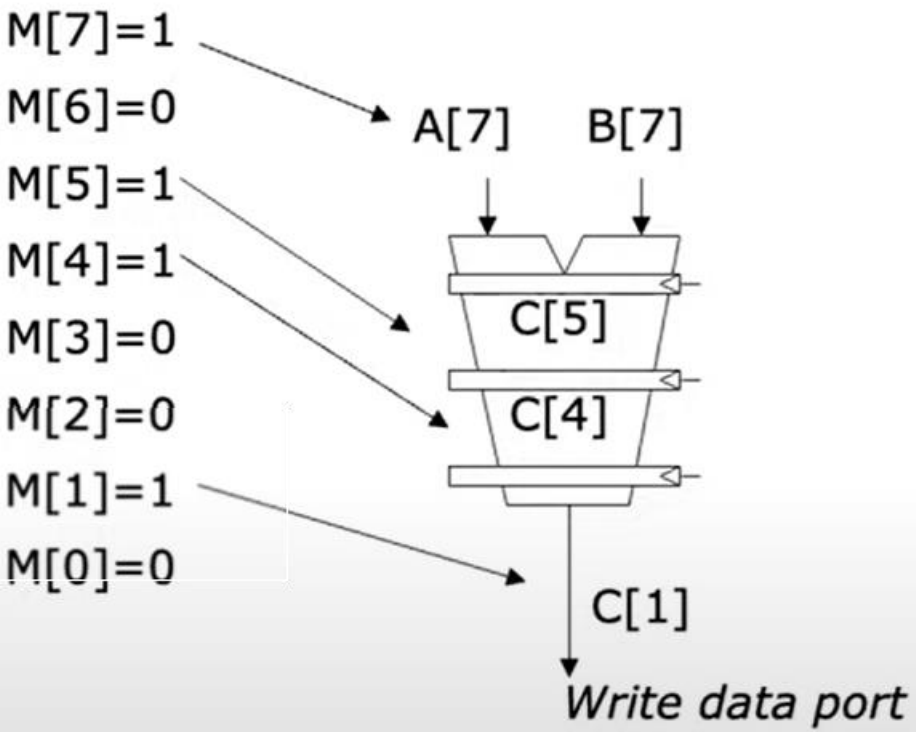
Simple Implementation

- 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



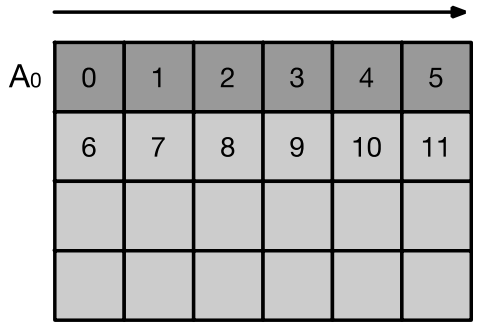
Some Issues

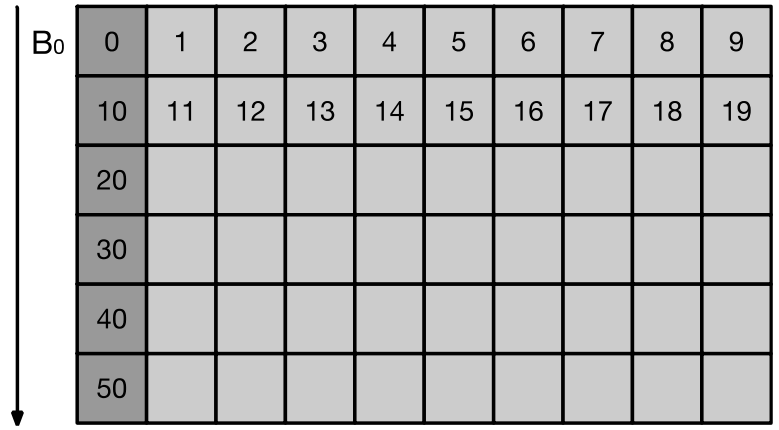
- Stride and bank count
  - As long as stride and bank count are *relatively prime* to each other and there are enough banks to cover bank access latency, we can sustain 1 element/cycle throughput

- Storage format 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

Bank Conflicts in Matrix Multiplication

- A and B matrices, both stored in memory in row-major order





$A_{4x6}\ B_{6x10} \rightarrow C_{4x10}$

Dot product of each row vector of A with each column vector of B

- Load A's row 0 into vector register V_1
  - Each time, increment address by 1 to access the next column
  - Accesses have a stride of 1
- Load B's column 0 into vector register V_2
  - Each time, increment address by 10
  - Accesses have a stride of 10

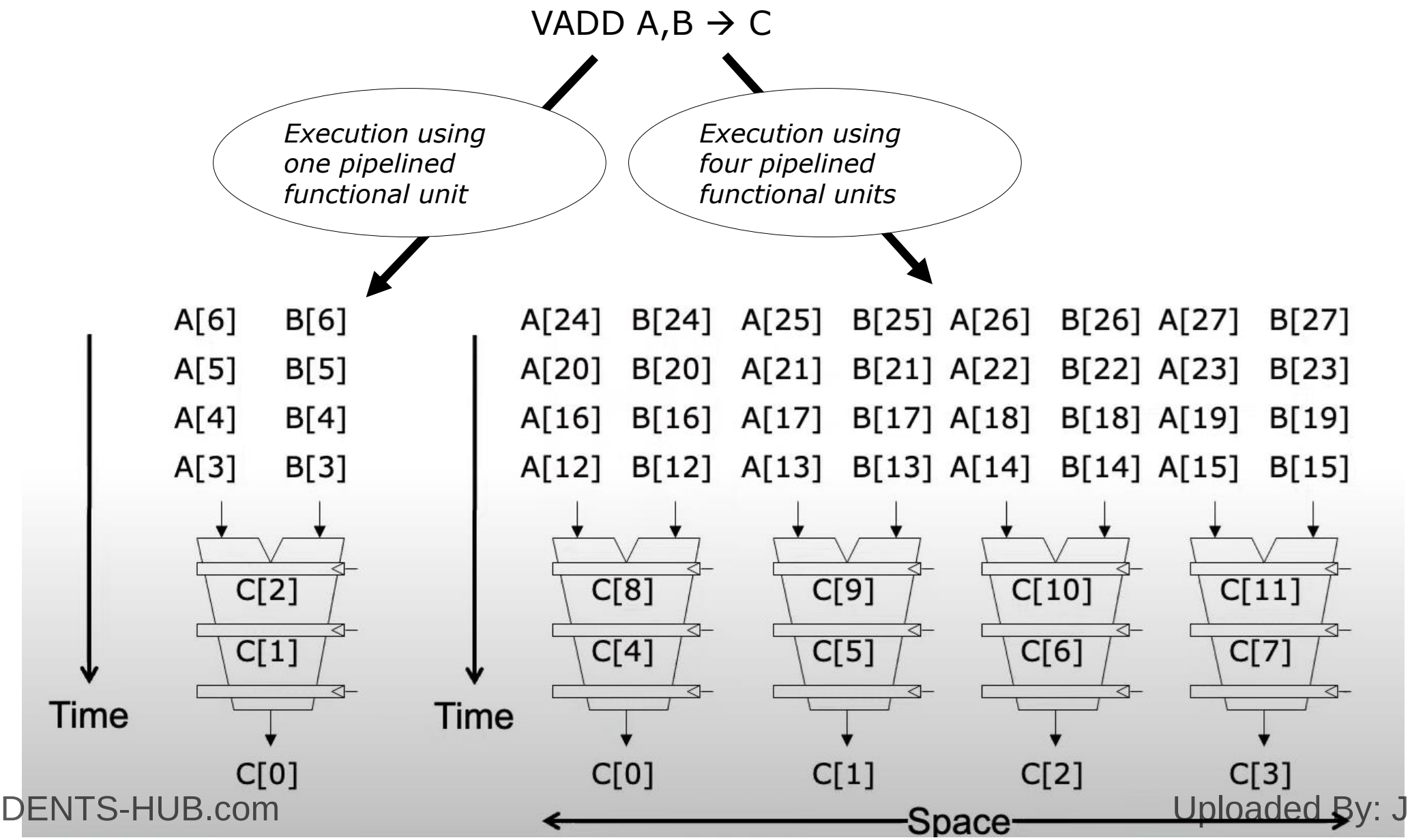
Different strides can lead to bank conflicts

How do we minimize them?

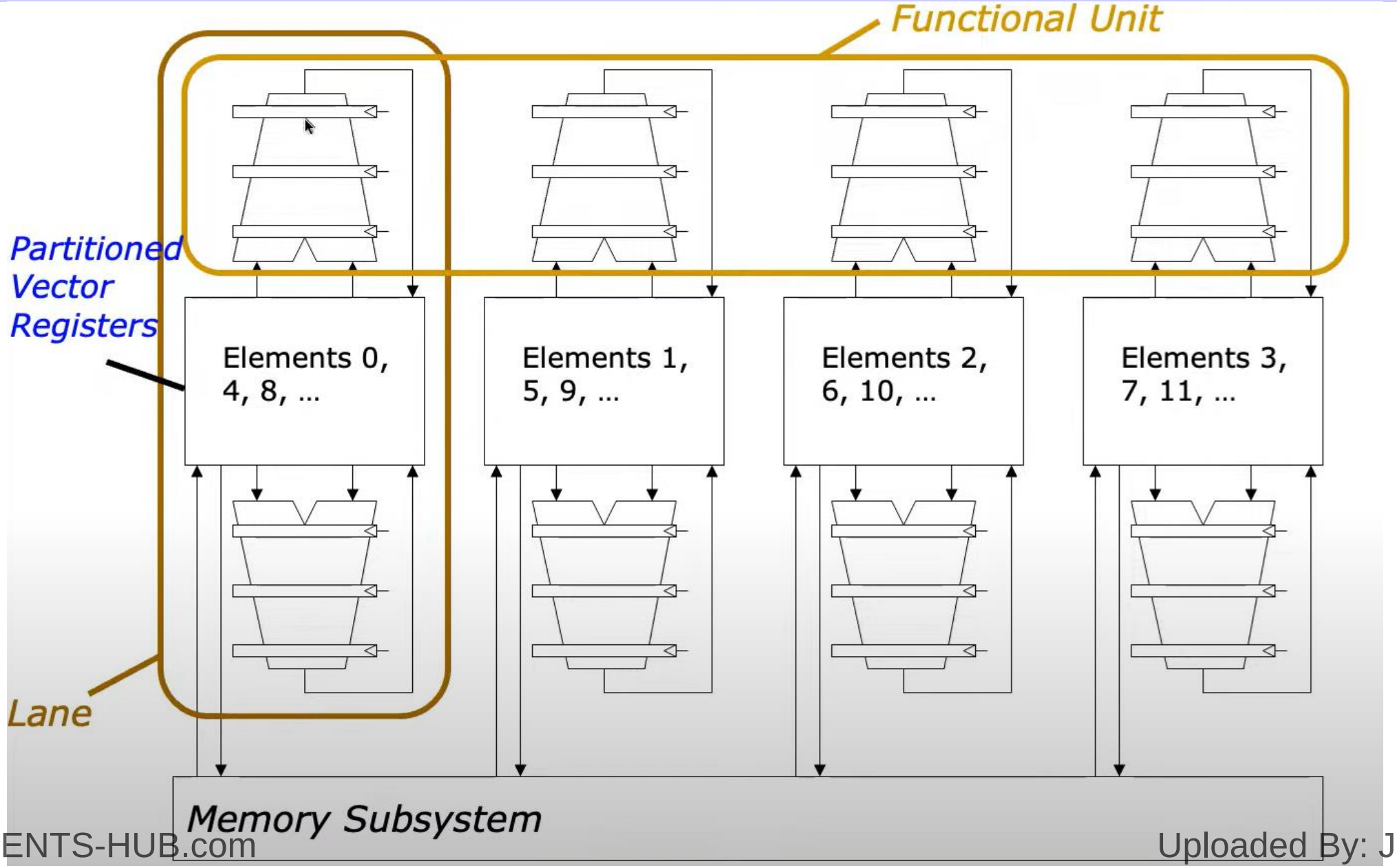
Minimizing Bank Conflicts

- More banks
- More ports in each bank
- Better data layout to match the access pattern
  - Is this always possible?
- Better mapping of address to bank
  - E.g., randomized mapping

Vector Instruction Execution



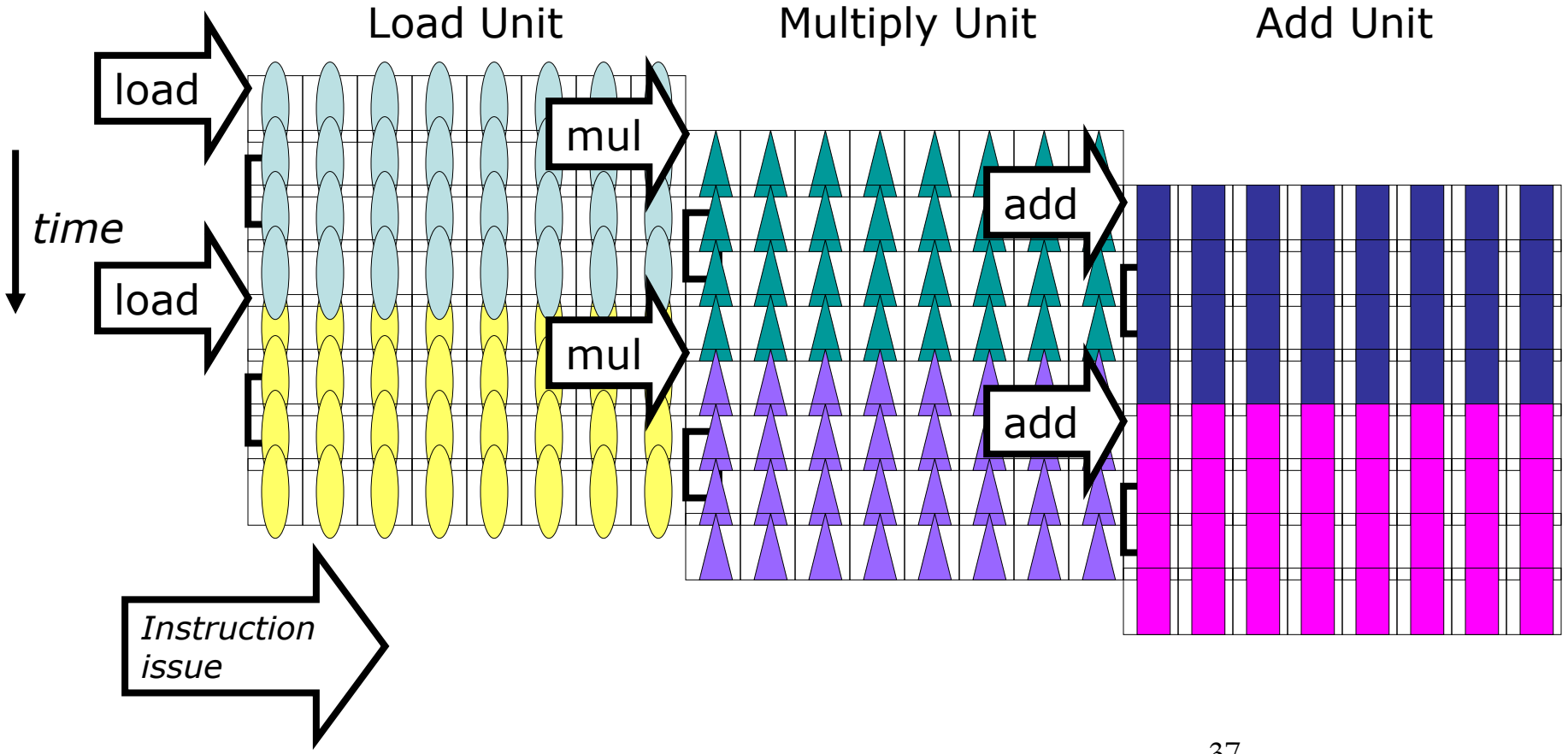
Vector Unit Structure



Vector Instruction Level Parallelism

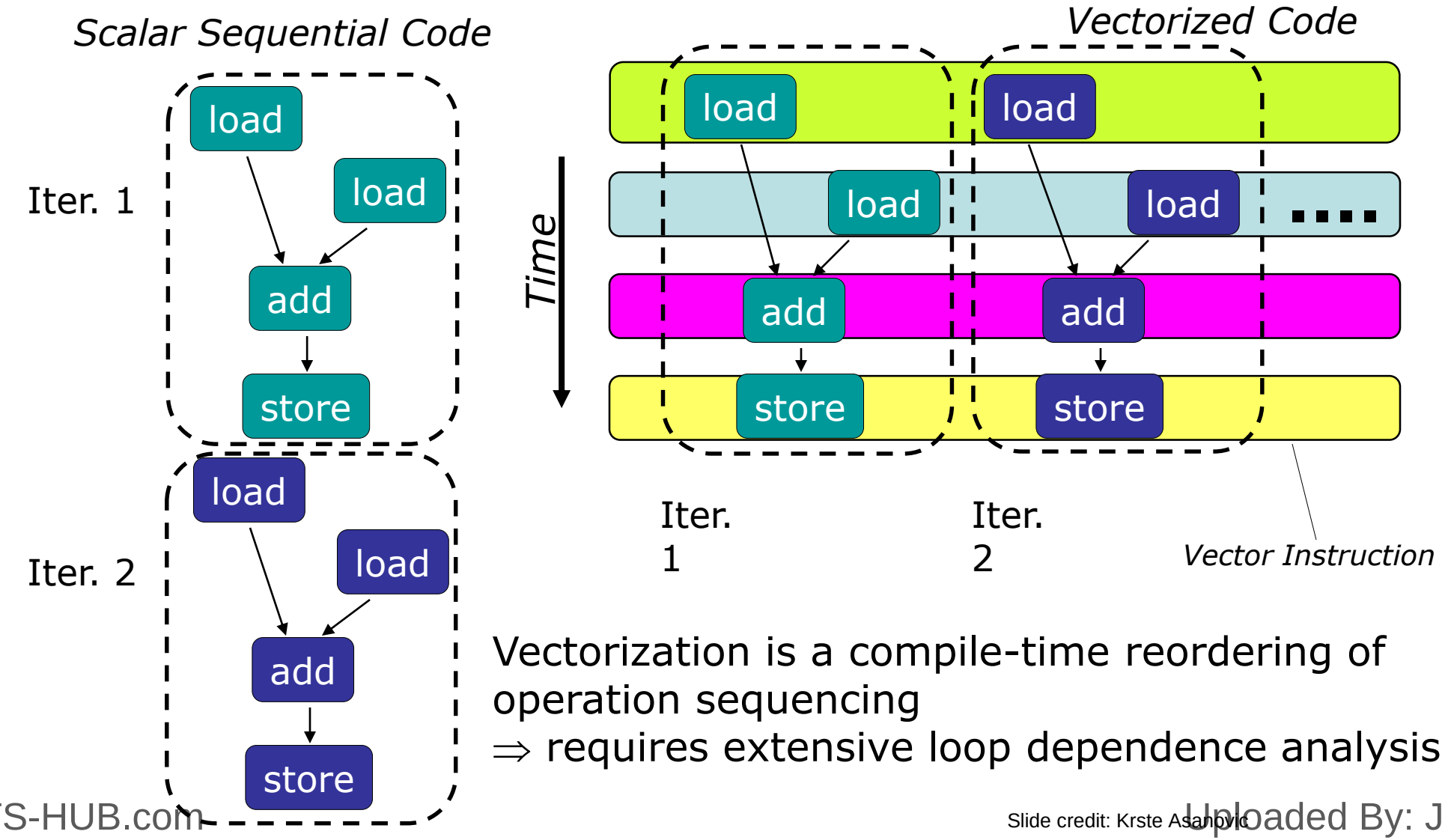
Can overlap execution of multiple vector instructions

- Example machine has 32 elements per vector register and 8 lanes
- Example with 24 operations/cycle (steady state) while issuing 1 vector instruction/cycle



Automatic Code Vectorization

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



Vectorization is a compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Slide credit: Krste Asanović

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
- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSEn/AVX, PowerPC AltiVec, ARM Advanced SIMD, RISC V V (RVV)