

CSC 631: High-Performance Computer Architecture

Spring 2022

Lecture 10: Vectors

Supercomputer Applications

- Typical application areas
 - Military research (nuclear weapons, cryptography)
 - Scientific research
 - Weather forecasting
 - Oil exploration
 - Industrial design (car crash simulation)
 - Bioinformatics
 - Cryptography
- All involve huge computations on large data set
- Supercomputers: CDC6600, CDC7600, Cray-1, ...
 - Cray/HPE Frontier is the latest from Cray
- In 70s-80s, Supercomputer \equiv Vector Machine

Vector Supercomputers

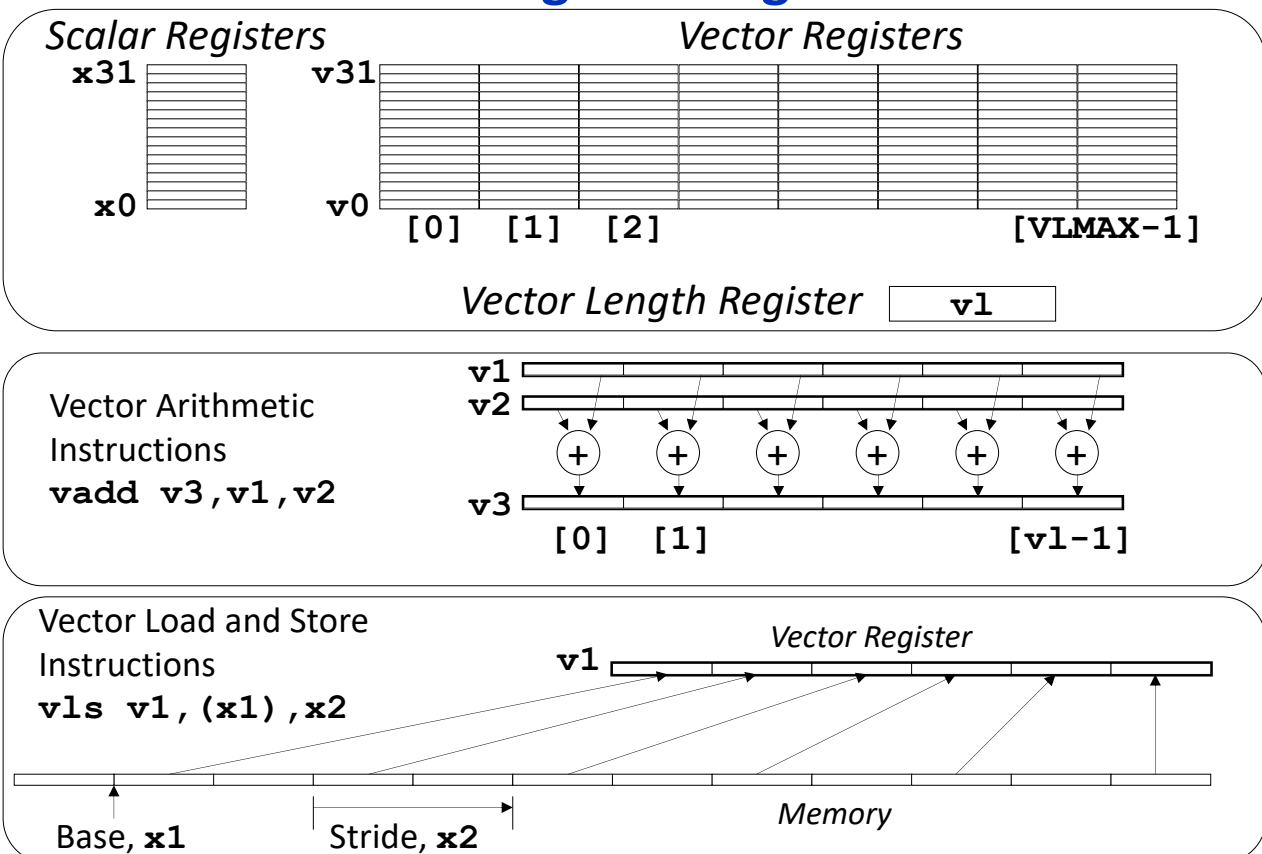


[©Cray Research, 1976]

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

3

Vector Programming Model



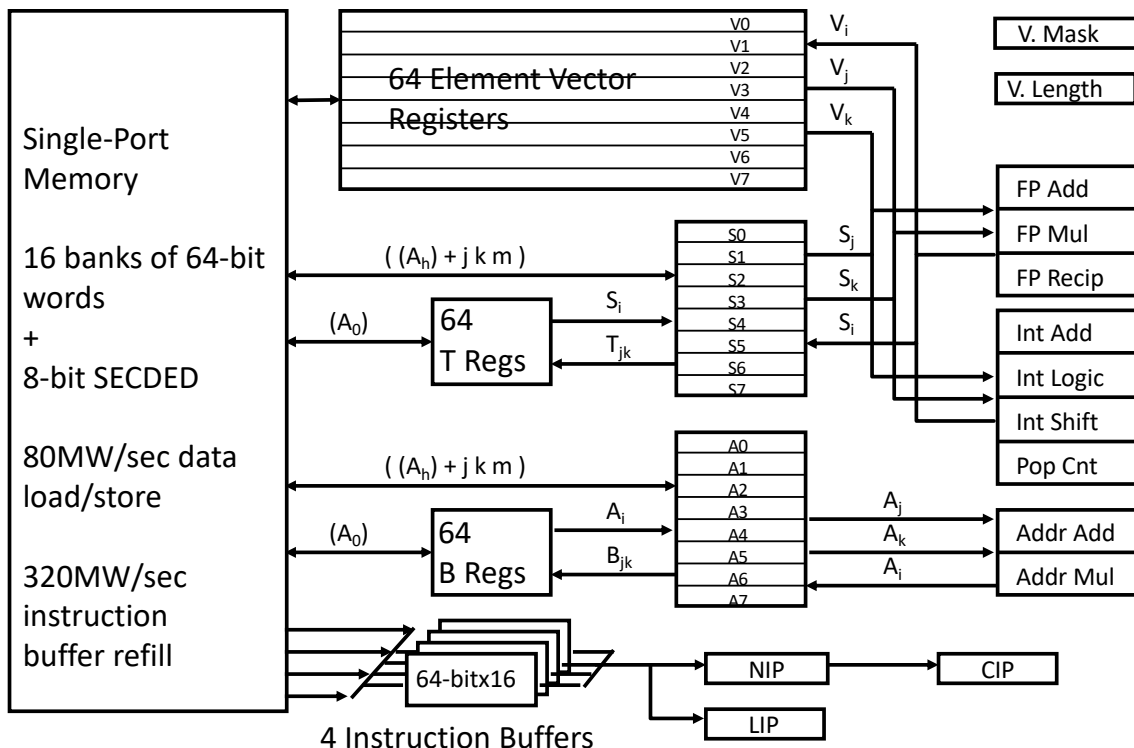
4

Vector Code Example

<pre># C code for (i=0; i<64; i++) C[i] = A[i] + B[i];</pre>	<pre># Scalar Code li x4, 64 loop: fld f1, 0(x1) fld f2, 0(x2) fadd.d f3, f1, f2 fsd f3, 0(x3) addi x1, x1, 8 addi x2, x2, 8 addi x3, x3, 8 subi x4, x4, 1 bnez x4, loop</pre>	<pre># Vector Code li x4, 64 vsetvl x4 vld v1, (x1) vld v2, (x2) vadd v3, v1, v2 vst v3, (x3)</pre>
---	--	---

5

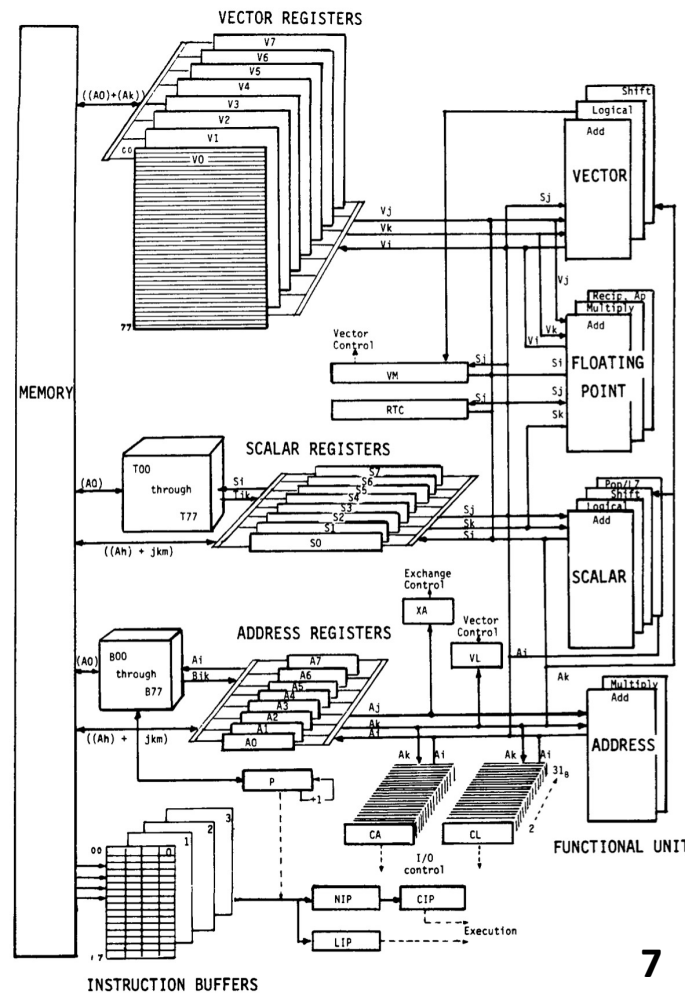
Cray-1 (1976)



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

6

Cray-1 (1976)



7

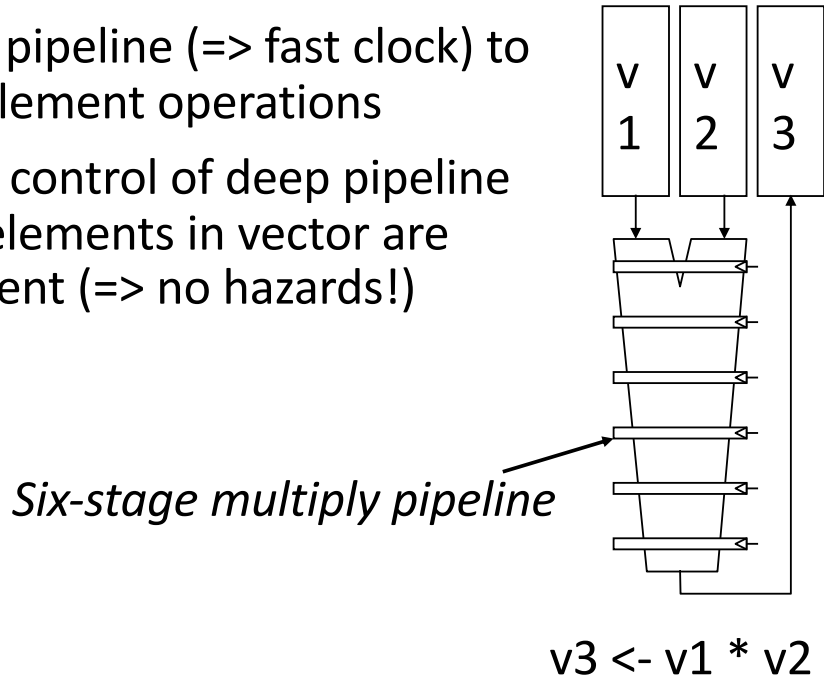
Vector Instruction Set Advantages

- 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)

8

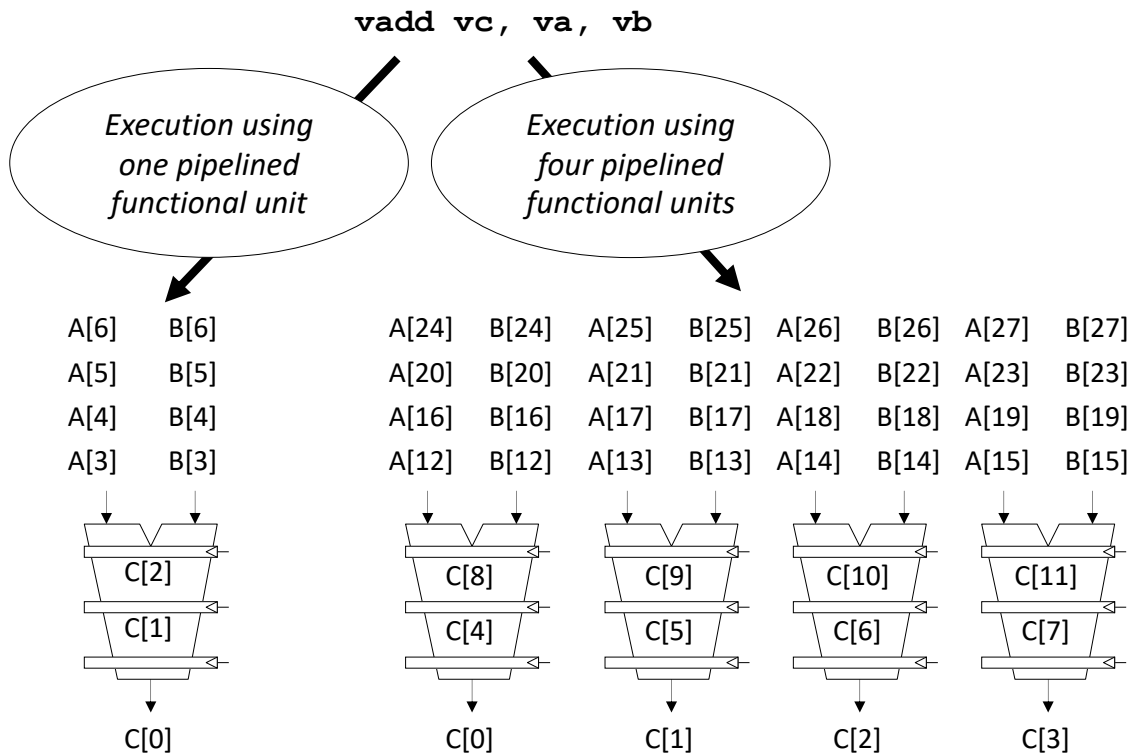
Vector Arithmetic Execution

- Use deep pipeline (\Rightarrow fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (\Rightarrow no hazards!)



9

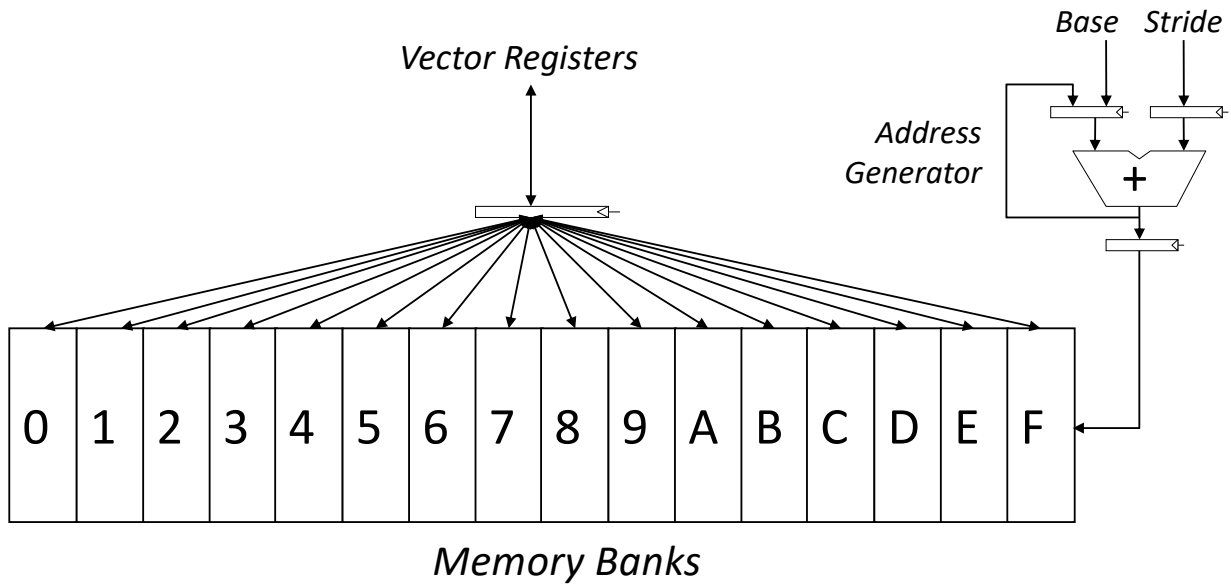
Vector Instruction Execution



10

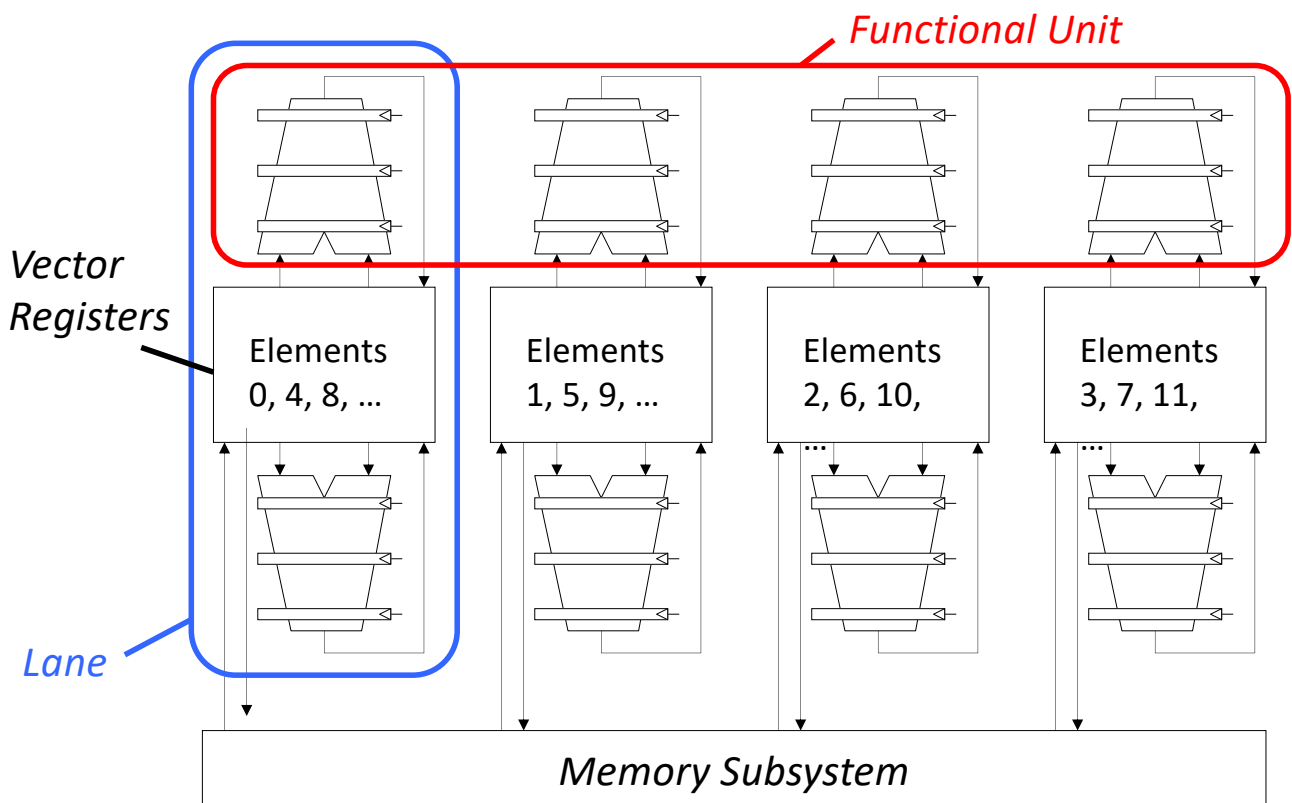
Interleaved Vector Memory System

- Bank busy time: Time before bank ready to accept next request
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency



11

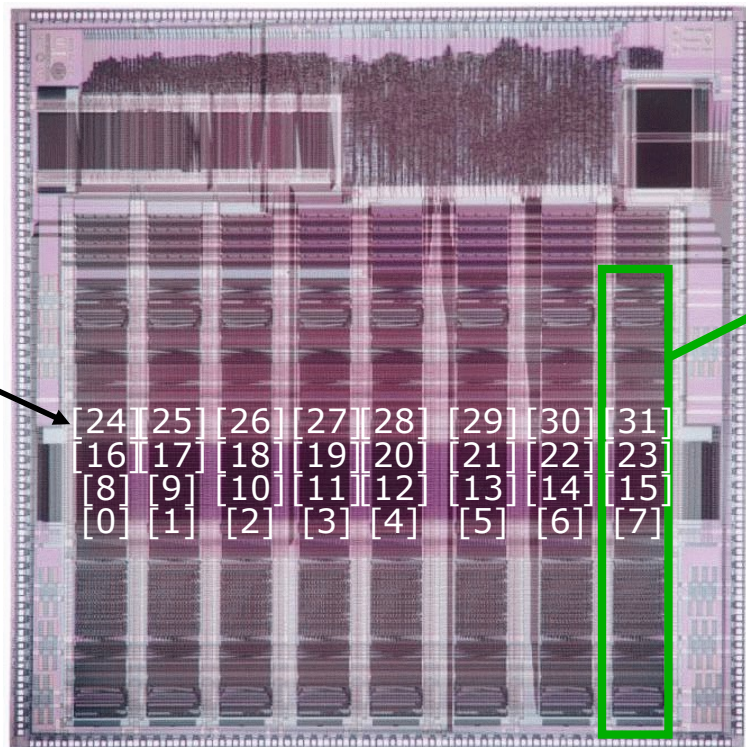
Vector Unit Structure



12

T0 Vector Microprocessor (UCB/ICSI, 1995)

Vector register elements striped over lanes

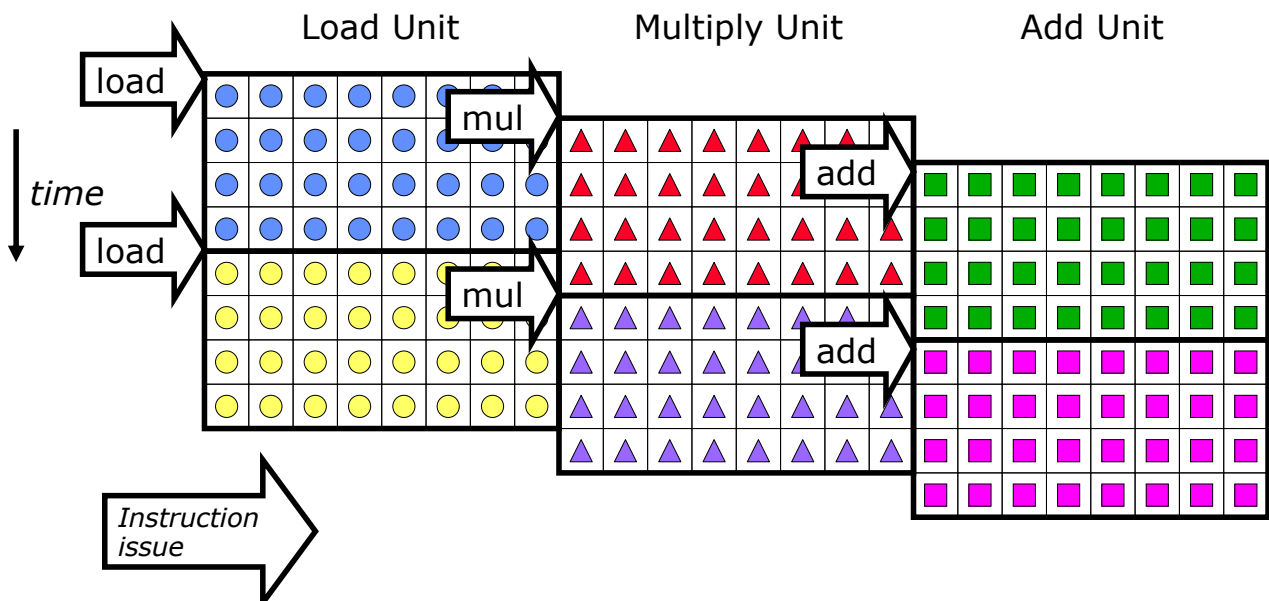


Lane

13

Vector Instruction Parallelism

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

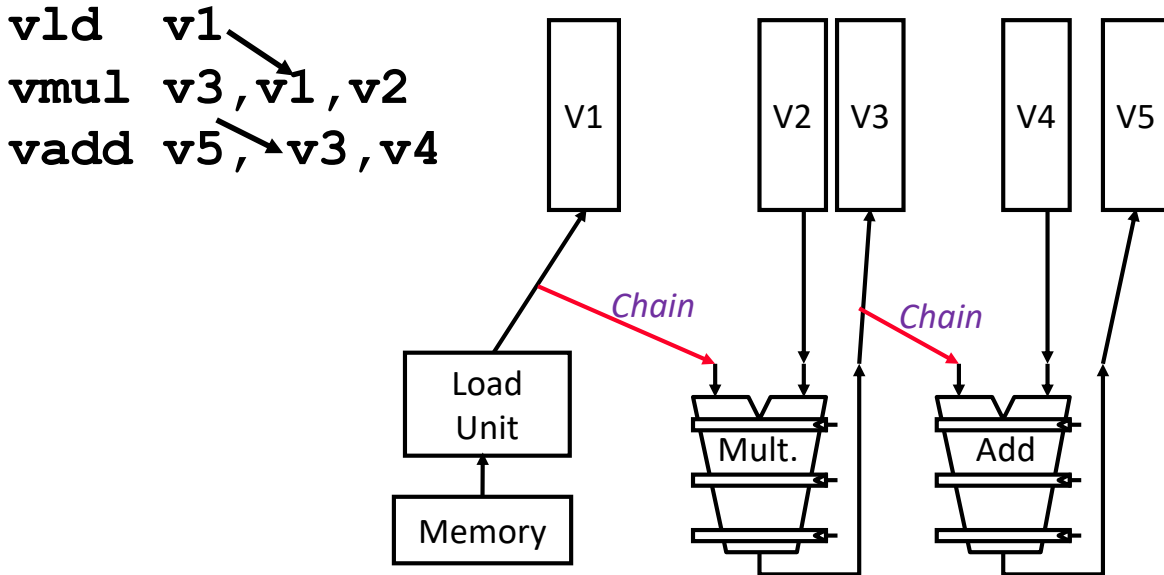


Complete 24 operations/cycle while issuing 1 short instruction/cycle

14

Vector Chaining

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



15

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

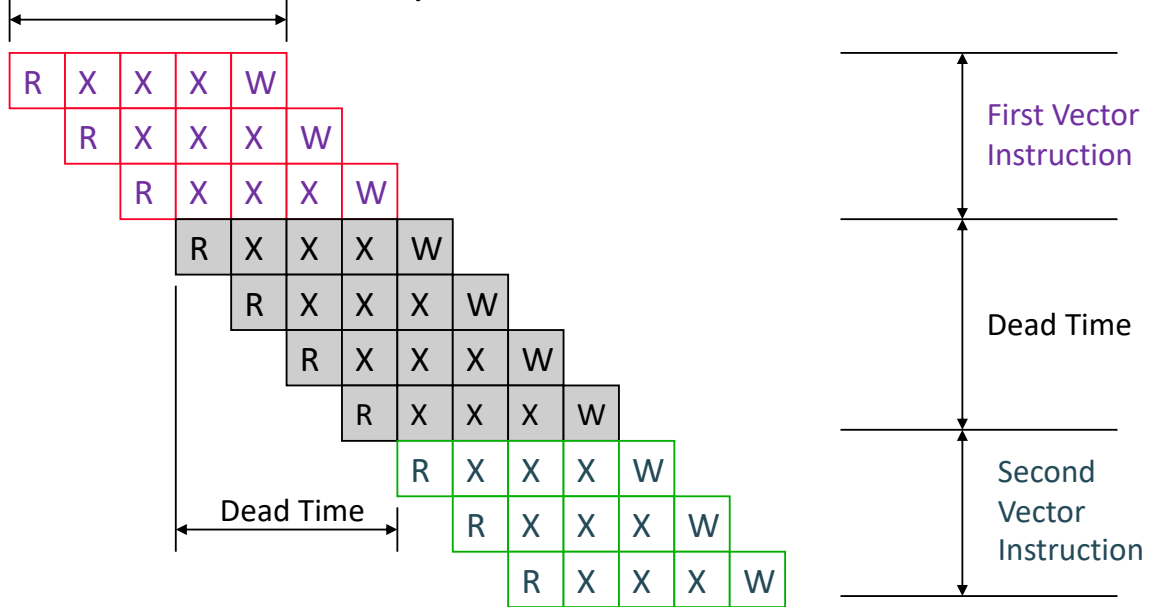


16

Vector Startup

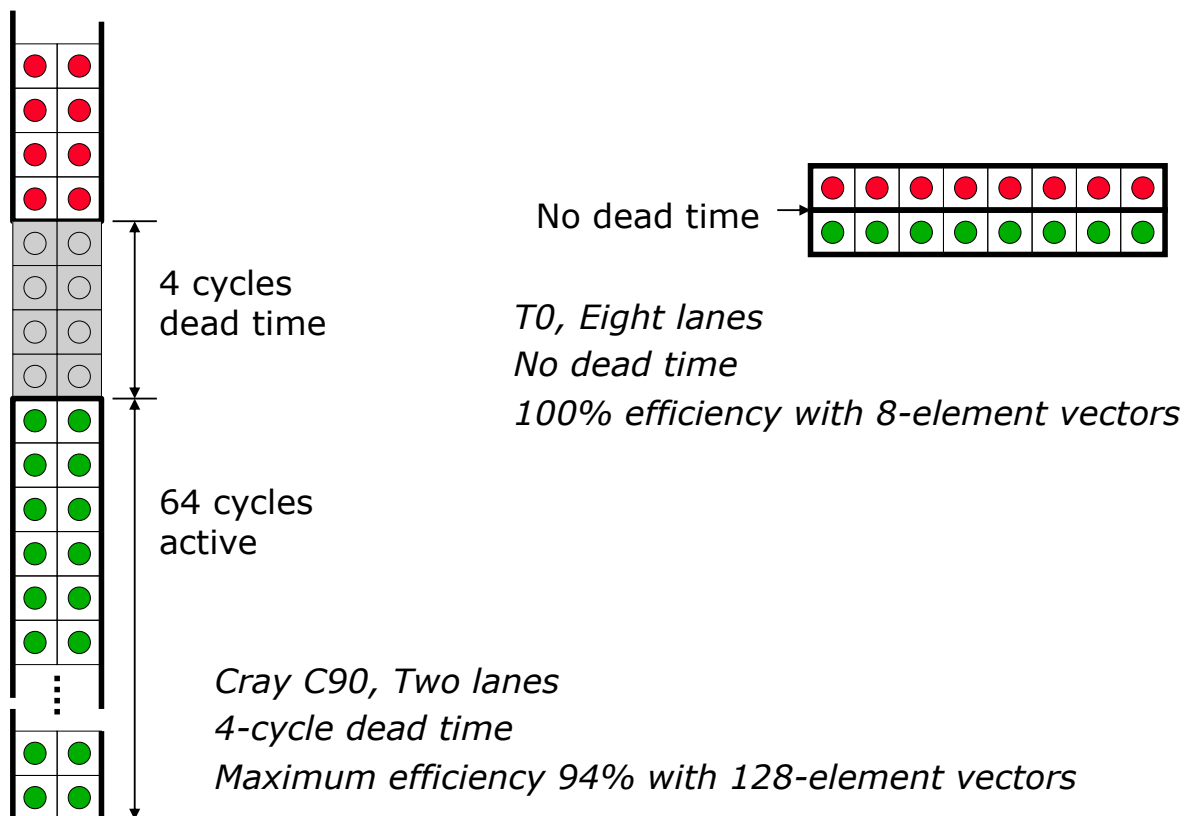
- Two components of vector startup penalty
 - functional unit latency (time through pipeline)
 - dead time or recovery time (time before another vector instruction can start down pipeline)

Functional Unit Latency



17

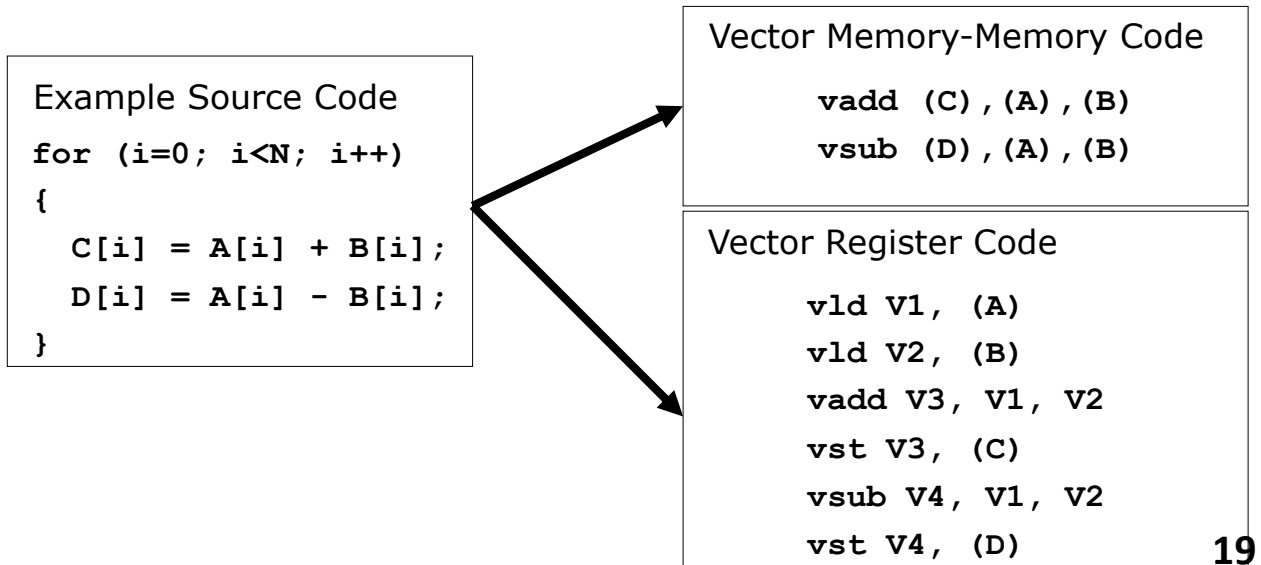
Dead Time and Short Vectors



18

Vector Memory-Memory versus 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



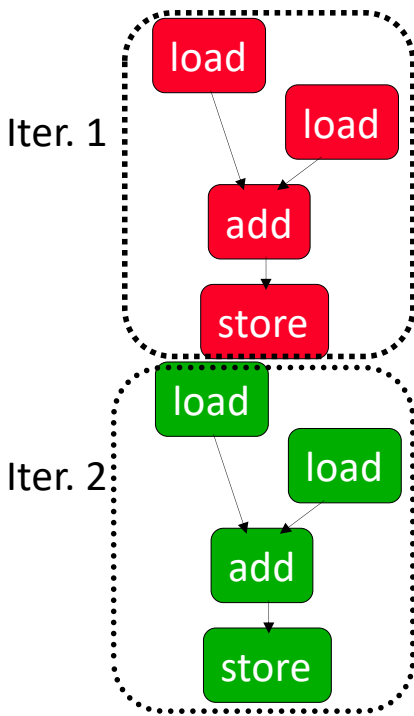
Vector Memory-Memory vs. Vector Register Machines

- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
 - All operands must be read in and out of memory
- VMMA makes it difficult to overlap execution of multiple vector operations, why?
 - Must check dependencies on memory addresses
- VMMA incur greater startup latency
 - Scalar code was faster on CDC Star-100 for vectors < 100 elements
 - For Cray-1, vector/scalar breakeven point was around 2-4 elements
- Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures
- (we ignore vector memory-memory from now on)

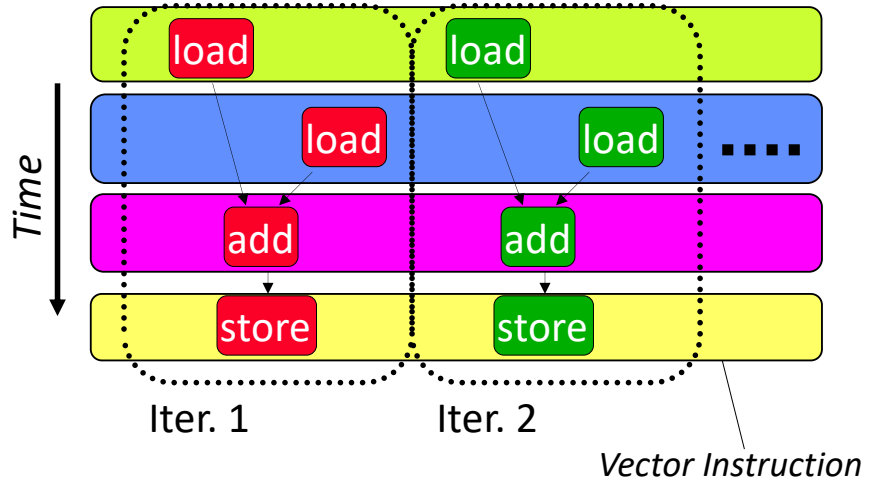
Automatic Code Vectorization

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

Scalar Sequential Code



Vectorized Code



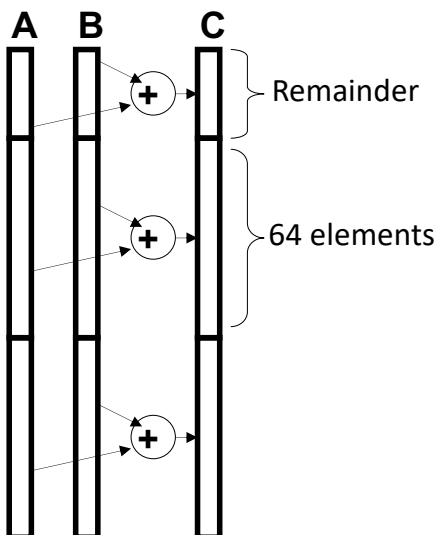
Vectorization is a massive compile-time reordering of operation sequencing
 ⇒ requires extensive loop-dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, "Stripmining"

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



```
andi x1, xN, 63 # N mod 64
vsetvl x1      # Do remainder
loop:
vld v1, (xA)
slli x2, x1, 3 # Multiply by 8
add xA, xA, x2 # Bump pointer
vld v2, (xB)
add xB, xB, x2
vadd v3, v1, v2
vst v3, (xC)
add xC, xC, x2
sub xN, xN, x1 # Subtract elements
li x1, 64
vsetvl x1      # Reset full length
bgtz xN, loop # Any more to do?
```

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

- vector version of predicate registers, 1 bit per element

...and *maskable* vector instructions

- vector operation becomes bubble (“NOP”) at elements where mask bit is clear

Code example:

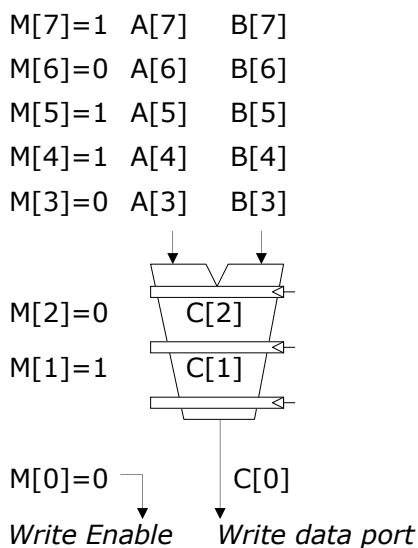
```
cvm                # Turn on all elements
vld vA, (xA)       # Load entire A vector
vgt vA, f0         # Set bits in mask register where A>0
vld vA, (xB)       # Load B vector into A under mask
vst vA, (xA)       # Store A back to memory under mask
```

23

Masked Vector Instructions

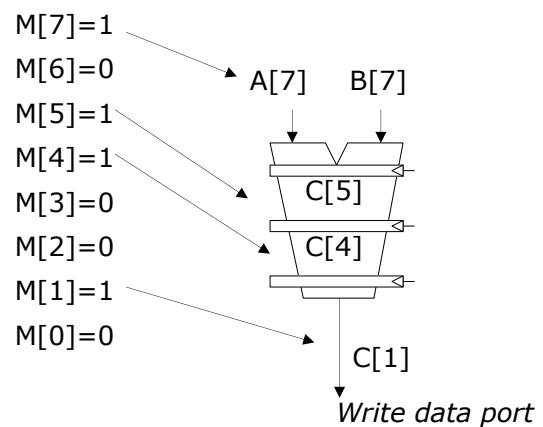
Simple Implementation

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



Density-Time Implementation

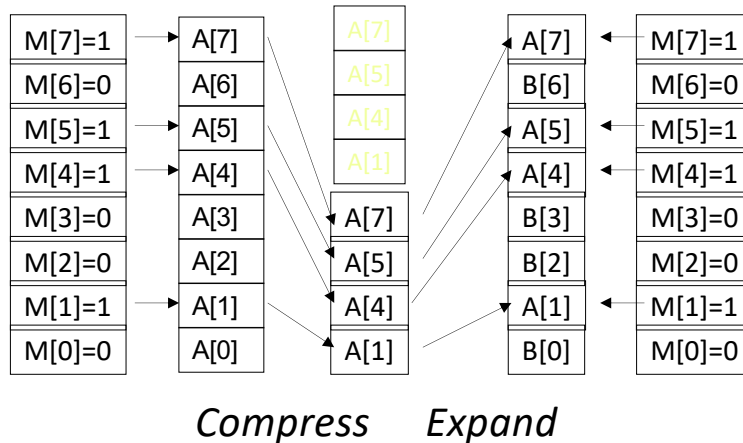
- scan mask vector and only execute elements with non-zero masks



24

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

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. partials
} while (VL>1)
```

Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*Gather*)

```
vld vD, (xD)          # Load indices in D vector  
vlx vC, (xC), vD      # Load indexed from xC base  
vld vB, (xB)          # Load B vector  
vadd vA, vB, vC       # Do add  
vst vA, (xA)          # Store result
```

27

Histogram with Scatter/Gather

Histogram example:

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

Is following a correct translation?

```
vld vB, (xB)          # Load indices in B vector  
vlx vA, (xA), vB      # Gather initial A values  
vadd vA, vA, 1        # Increment  
vsx vA, (xA), vB      # Scatter incremented values
```

28

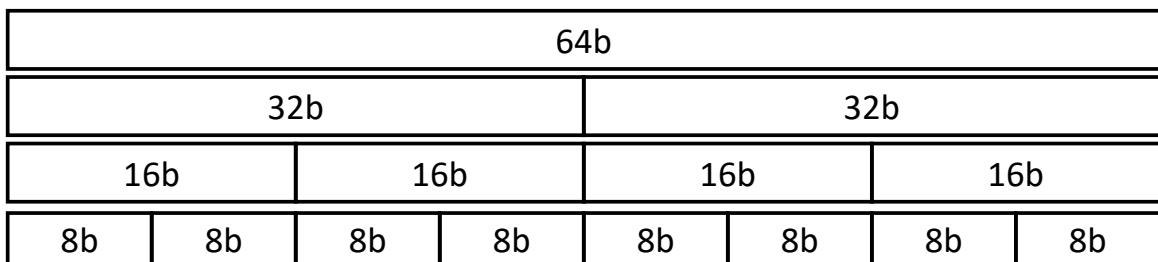
Vector Memory Models

- Some vector machines have a very relaxed memory model, e.g.
 - `vst v1, (x1) # Store vector to x1`
 - `vld v2, (x1) # Load vector from x1`
 - No guarantee that elements of v2 will have value of elements of v1 even when store and load execute by *same* processor!
- So require explicit memory barrier or fence
 - `vst v1, (x1) # Store vector to x1`
 - `fence # Enforce ordering s->l`
 - `vld v2, (x1) # Load vector from x1`

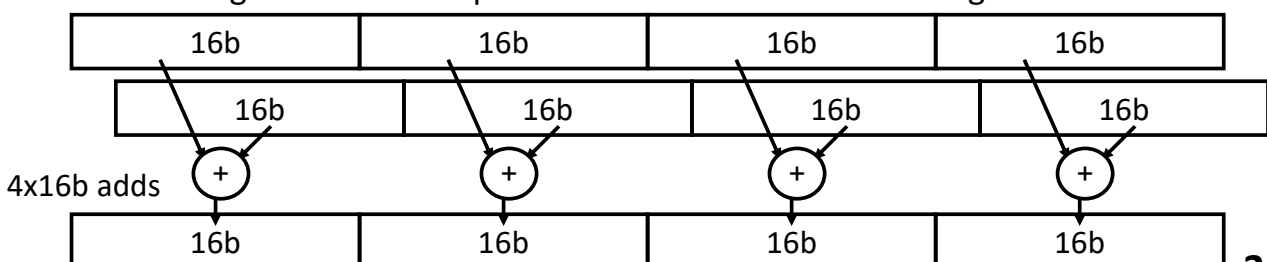
Vector machines support highly parallel memory systems (multiple lanes and multiple load and store units) with long latency (100+ clock cycles)

- hardware coherence checks would be prohibitively expensive
- vectorizing compiler can eliminate most dependencies

Packed SIMD Extensions



- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
 - Lincoln Labs TX-2 from 1957 had 36b datapath split into 2x18b or 4x9b
 - Newer designs have wider registers
 - 128b for PowerPC AltiVec, Intel SSE2/3/4
 - 256b/512b for Intel AVX
- Single instruction operates on all elements within register



Packed SIMD versus Vectors

- Limited instruction set:
 - no vector length control
 - no strided load/store or scatter/gather
 - unit-stride loads must be aligned to 64/128-bit boundary
- Limited vector register length:
 - requires superscalar dispatch to keep multiply/add/load units busy
 - loop unrolling to hide latencies increases register pressure
- Trend towards fuller vector support in microprocessors
 - Better support for misaligned memory accesses
 - Support of double-precision (64-bit floating-point)
 - Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b), gather added, scatter to follow
 - ARM Scalable Vector Extensions (SVE)