CS252 Graduate Computer Architecture
Spring 2014
Lecture 8: Vector Supercomputers

Krste Asanovic
`krste@eecs.berkeley.edu`
`http://inst.eecs.berkeley.edu/~cs252/sp14`

© Krste Asanovic, 2014

Last Time in Lecture 6

Overcoming the worst hazards in OoO superscalars:

- Branch prediction
- Load-Store Queues

© Krste Asanovic, 2014

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, …
- In 70s-80s, Supercomputer ≡ Vector Machine

© Krste Asanovic, 2014

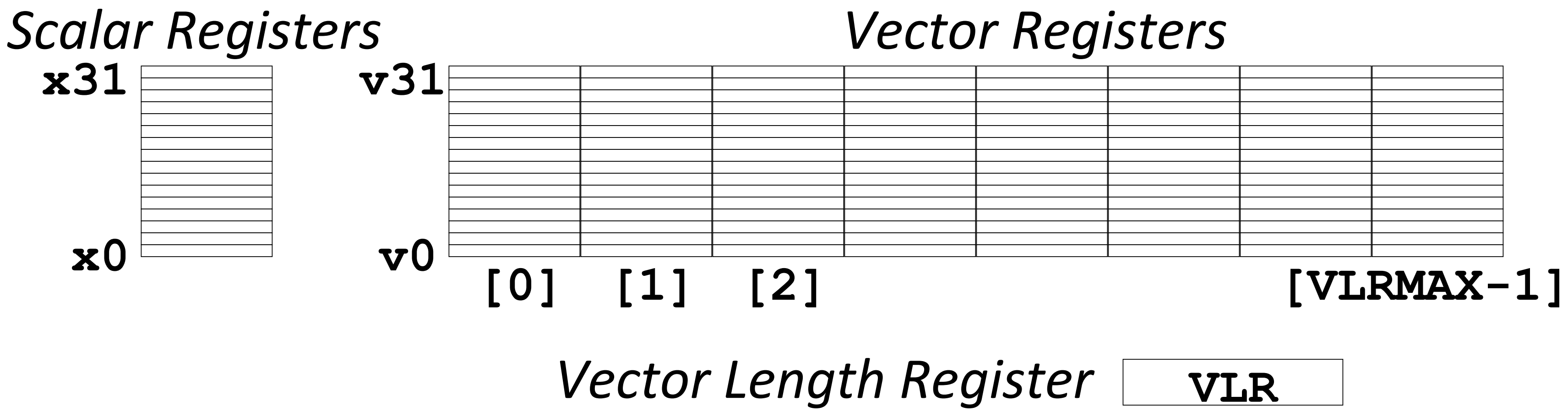
Vector Supercomputers

[©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

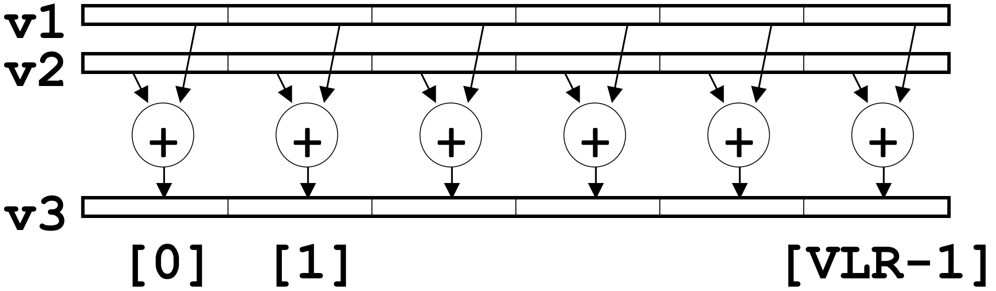
© Krste Asanovic, 2014

Vector Programming Model



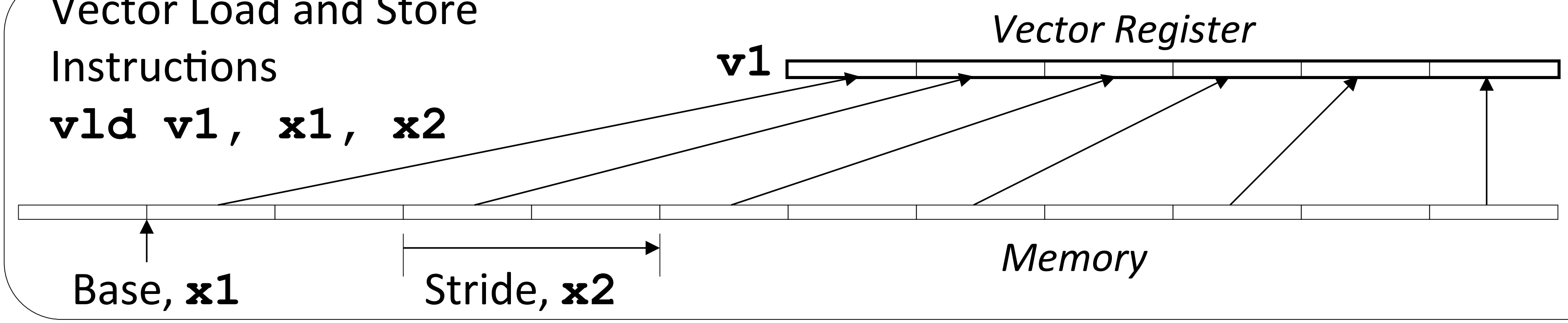
Vector Arithmetic Instructions

`vadd v3, v1, v2`



Vector Load and Store Instructions

`vld v1, x1, x2`



© Krste Asanovic, 2014

Vector Code Example

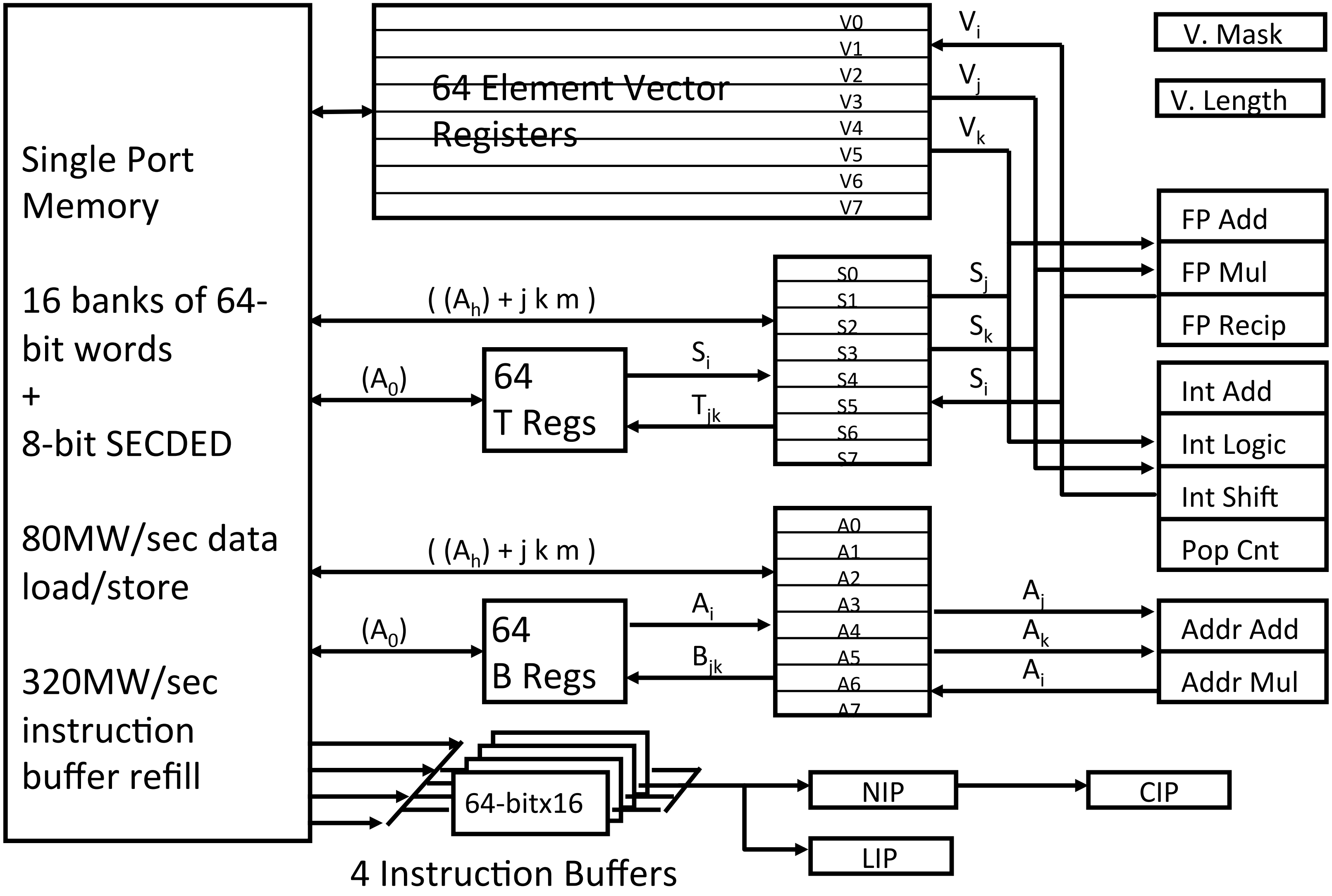
```
# C code
for (i=0; i<64; i++)
  C[i] = A[i] + B[i];
```

```
# 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, 8
  addi x2, 8
  addi x3, 8
  subi x4, 1
  bnez x4, loop
```

```
# Vector Code
  li x4, 64
  setvlr x4
  vfld v1, x1
  vfld v2, x2
  vfadd.d v3,v1,v2
  vfsd v3, x3
```

© Krste Asanovic, 2014

Cray-1 (1976)



Single Port Memory

16 banks of 64-bit words
+
8-bit SECDED

80MW/sec data load/store

320MW/sec instruction buffer refill

64 Element Vector Registers

V0
V1
V2
V3
V4
V5
V6
V7

V_i
V_j
V_k

V. Mask

V. Length

((A_h) + j k m)

(A$_0$)

64
T Regs

S_i
T_{jk}

S0
S1
S2
S3
S4
S5
S6
S7

S_j
S_k
S_i

FP Add
FP Mul
FP Recip

Int Add
Int Logic
Int Shift
Pop Cnt

((A_h) + j k m)

(A$_0$)

64
B Regs

A_i
B_{jk}

A0
A1
A2
A3
A4
A5
A6
A7

A_j
A_k
A_i

Addr Add
Addr Mul

64-bitx16

4 Instruction Buffers

NIP
CIP
LIP

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

© Krste Asanovic, 2014

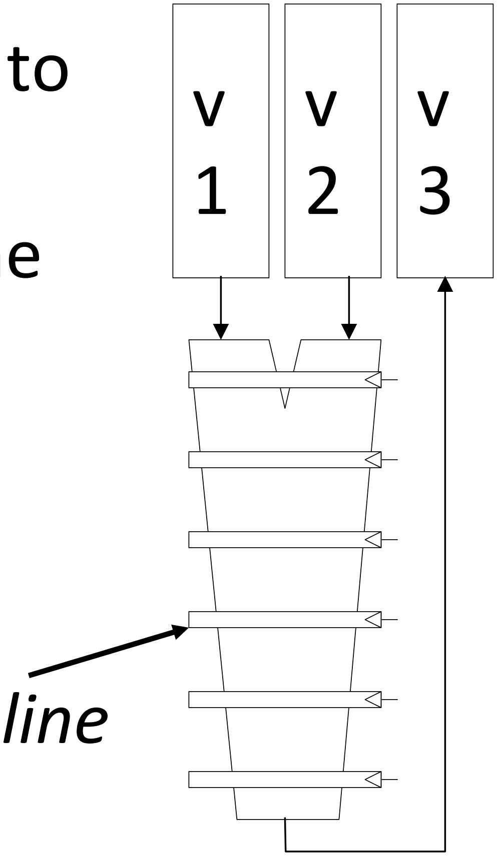
Vector Instruction Set Advantages

- 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)

© Krste Asanovic, 2014

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!)



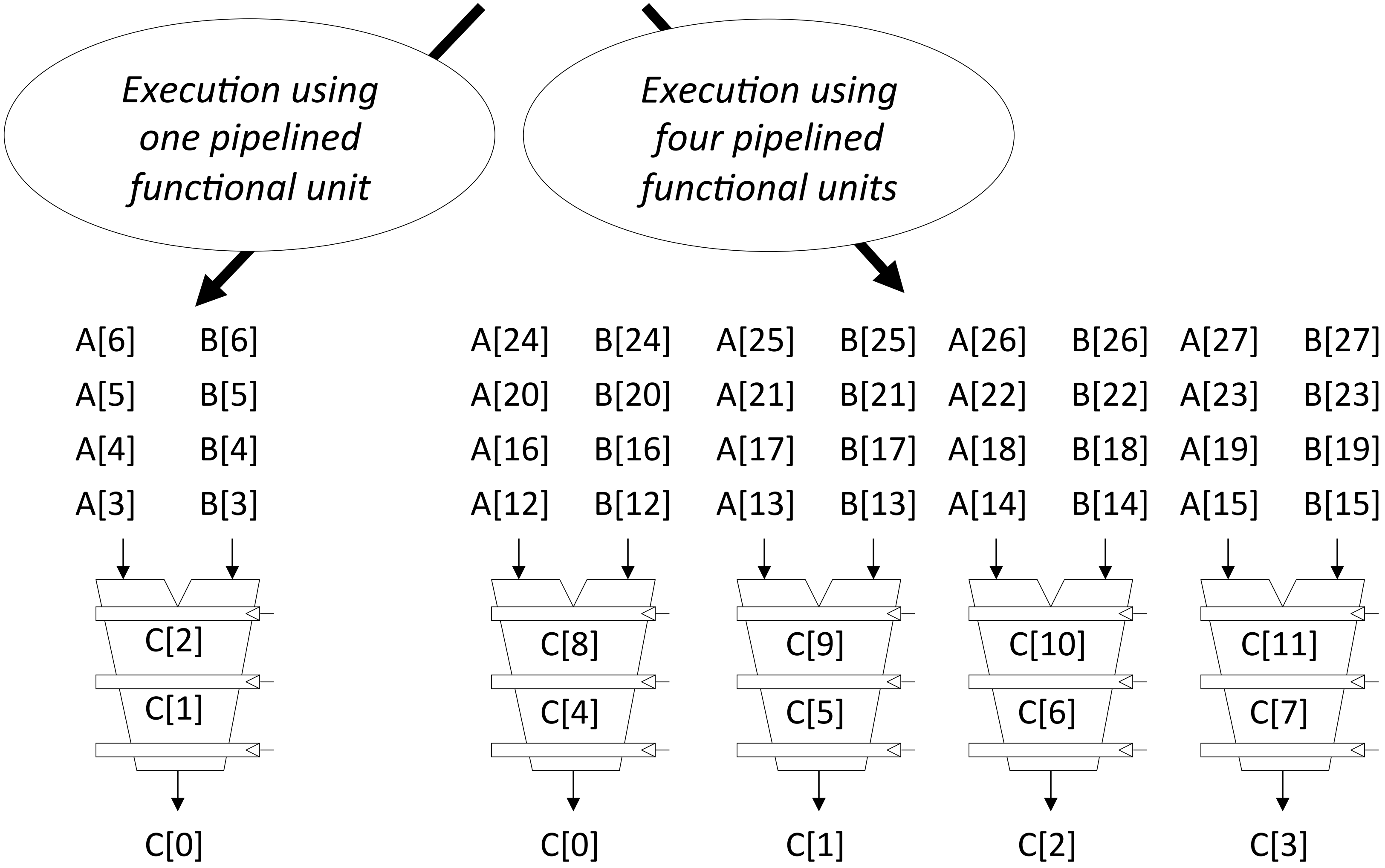
Six-stage multiply pipeline

v3 <- v1 * v2

© Krste Asanovic, 2014

Vector Instruction Execution

`vfadd.d vc, va, vb`



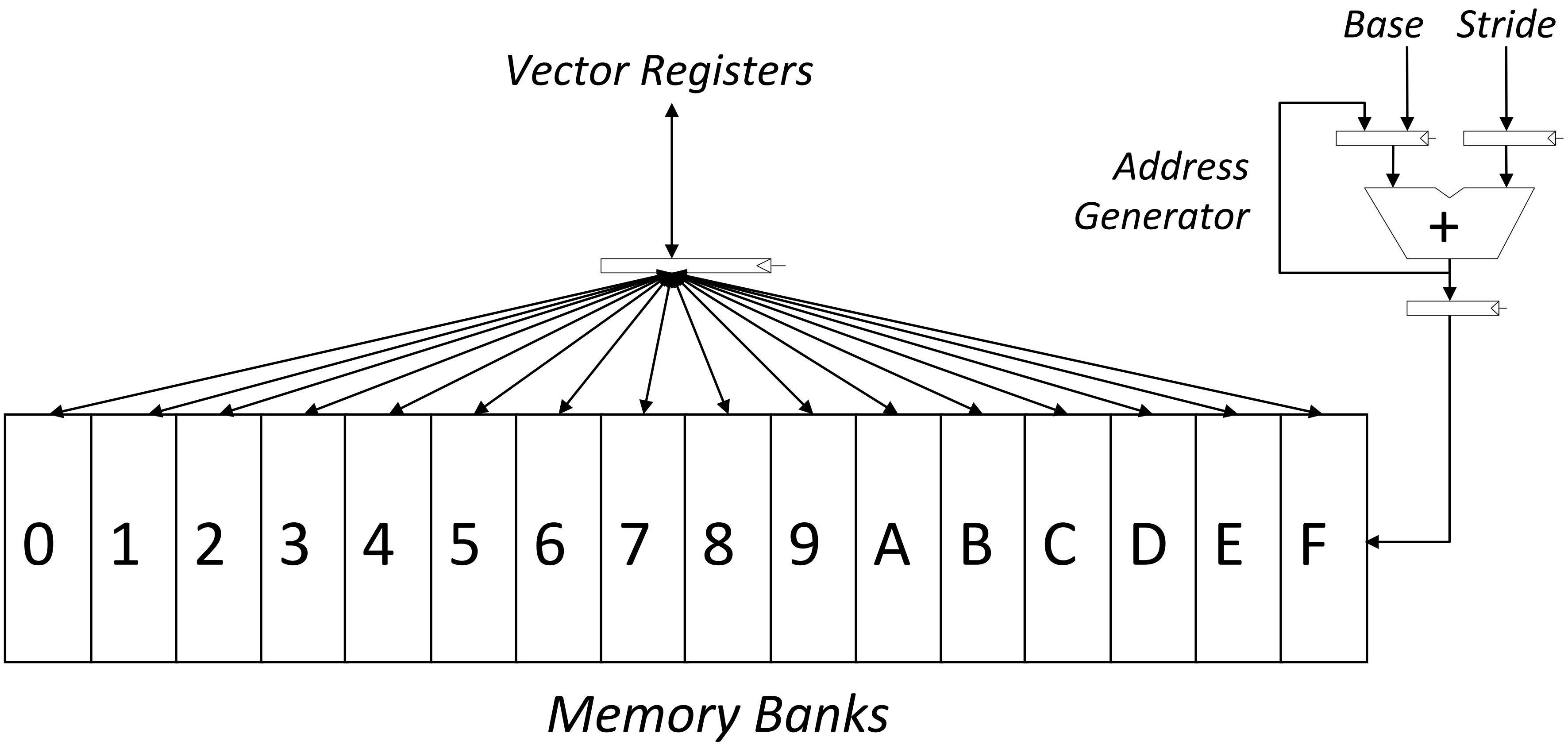
Execution using one pipelined functional unit

Execution using four pipelined functional units

© Krste Asanovic, 2014

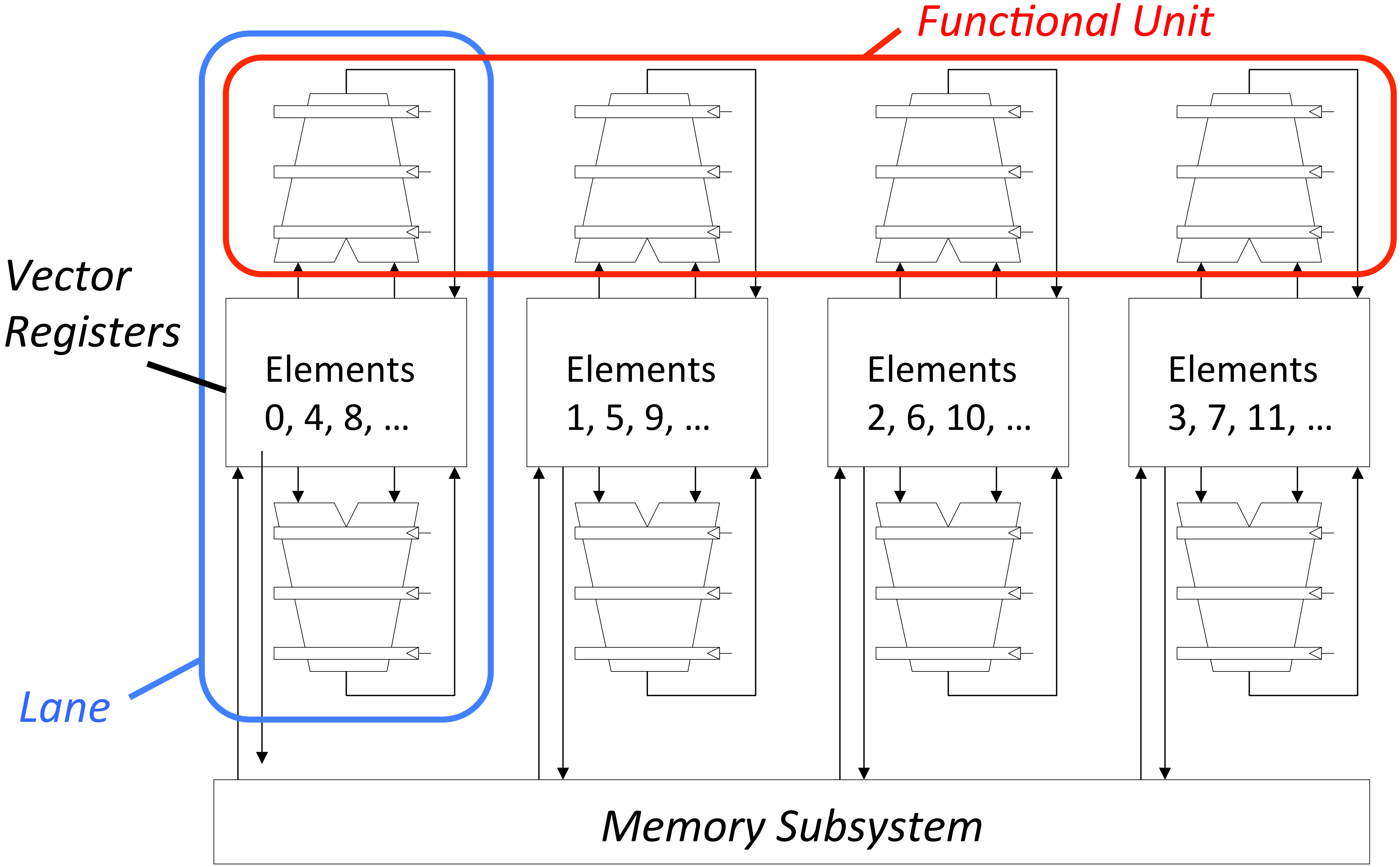
Interleaved Vector Memory System

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



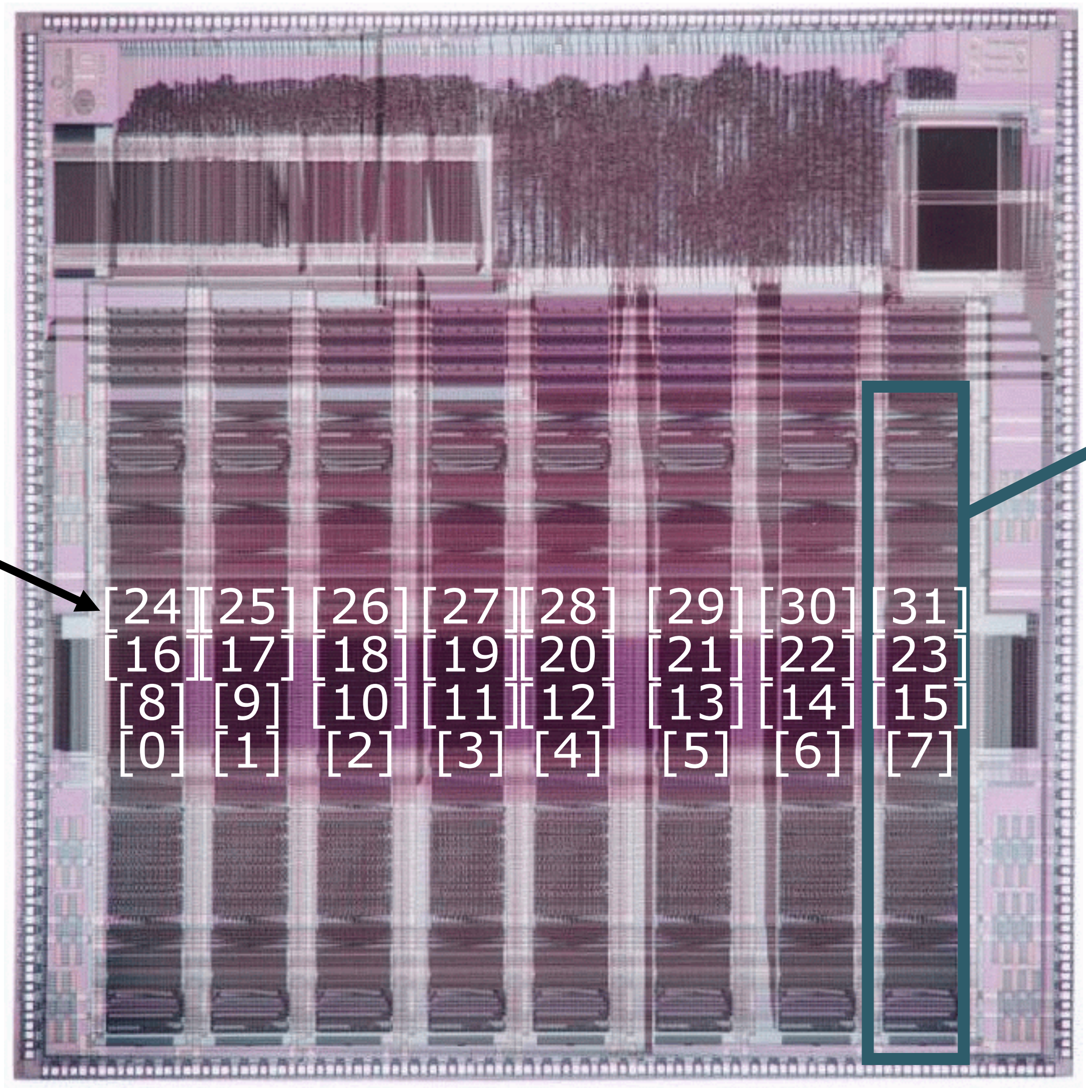
© Krste Asanovic, 2014

Vector Unit Structure



© Krste Asanovic, 2014

T0 Vector Microprocessor (UCB/ICSI, 1995)



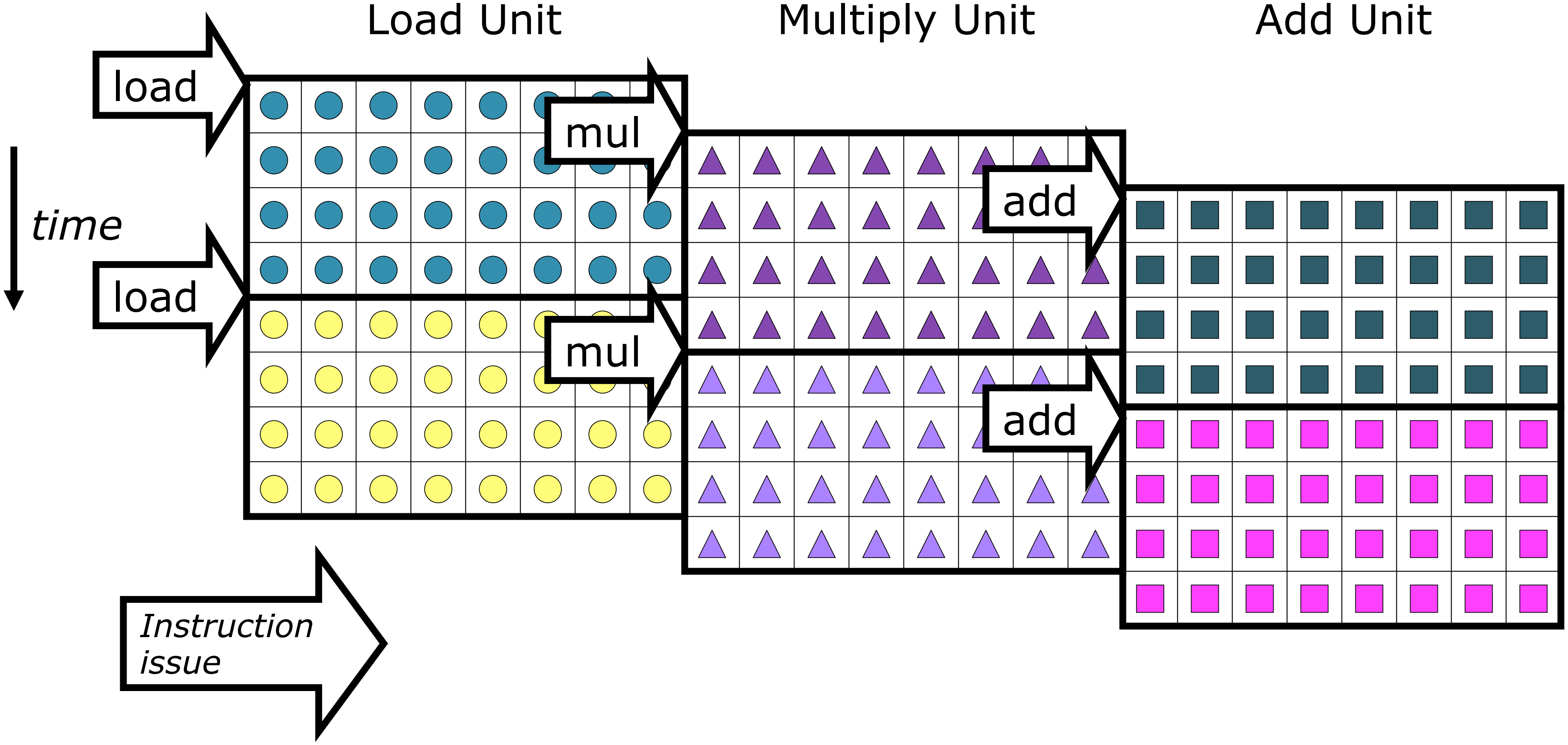
Vector register elements striped over lanes

Lane

© Krste Asanovic, 2014

Vector Instruction Parallelism

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



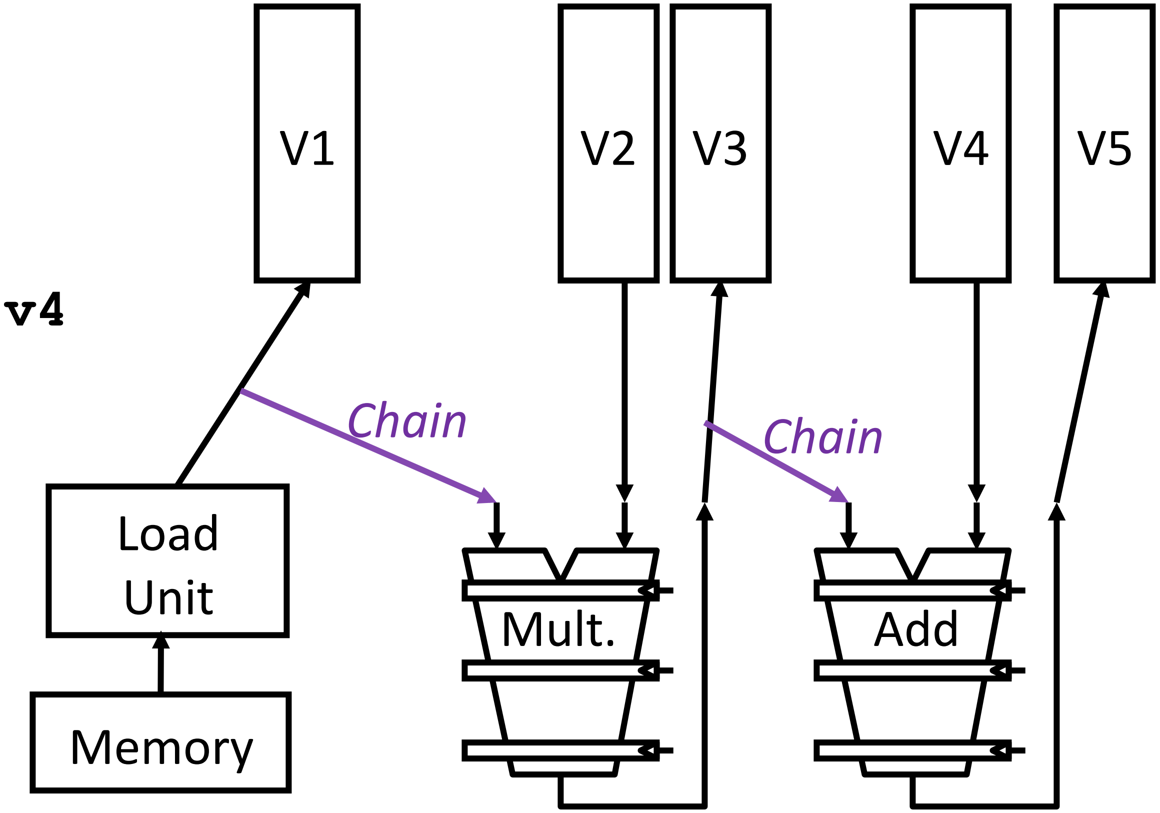
Complete 24 operations/cycle while issuing 1 short instruction/cycle

© Krste Asanovic, 2014

Vector Chaining

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

```
vld   v1
vfmul v3,v1,v2
Vfadd v5, v3, v4
```



Vector Chaining Advantage

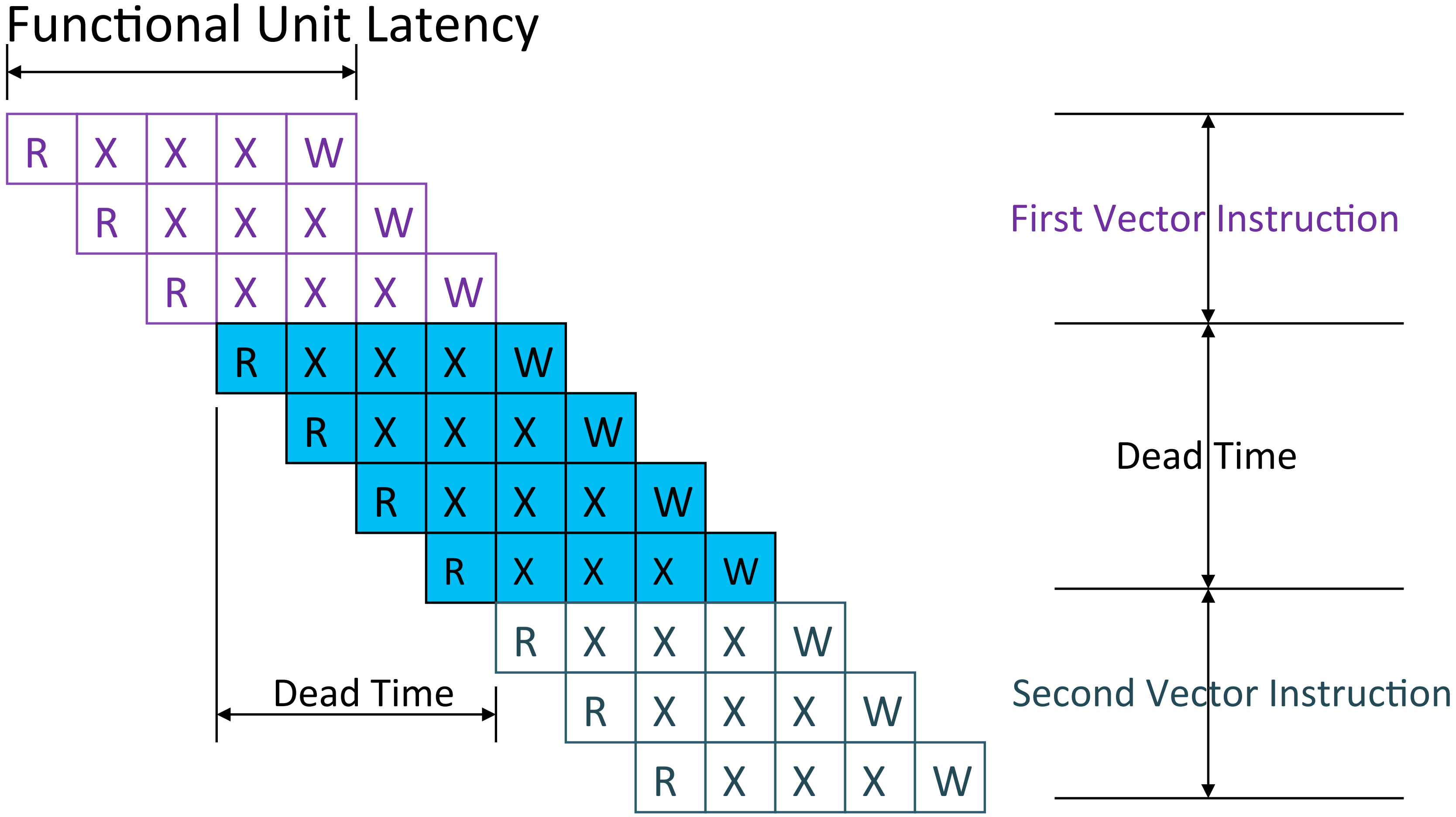
- Without chaining, must wait for last element of result to be written before starting dependent instruction

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

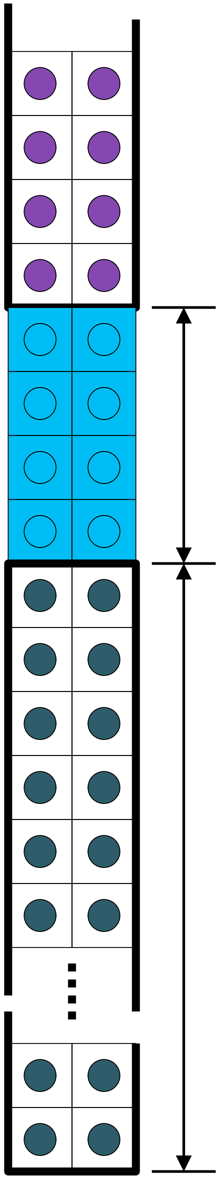
© Krste Asanovic, 2014

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)



Dead Time and Short Vectors

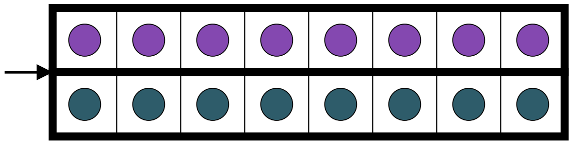


4 cycles dead time

64 cycles active

Cray C90, Two lanes
4 cycle dead time
Maximum efficiency 94% with 128 element vectors

No dead time



T0, Eight lanes
No dead time
100% efficiency with 8 element vectors

© Krste Asanovic, 2014

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

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
```

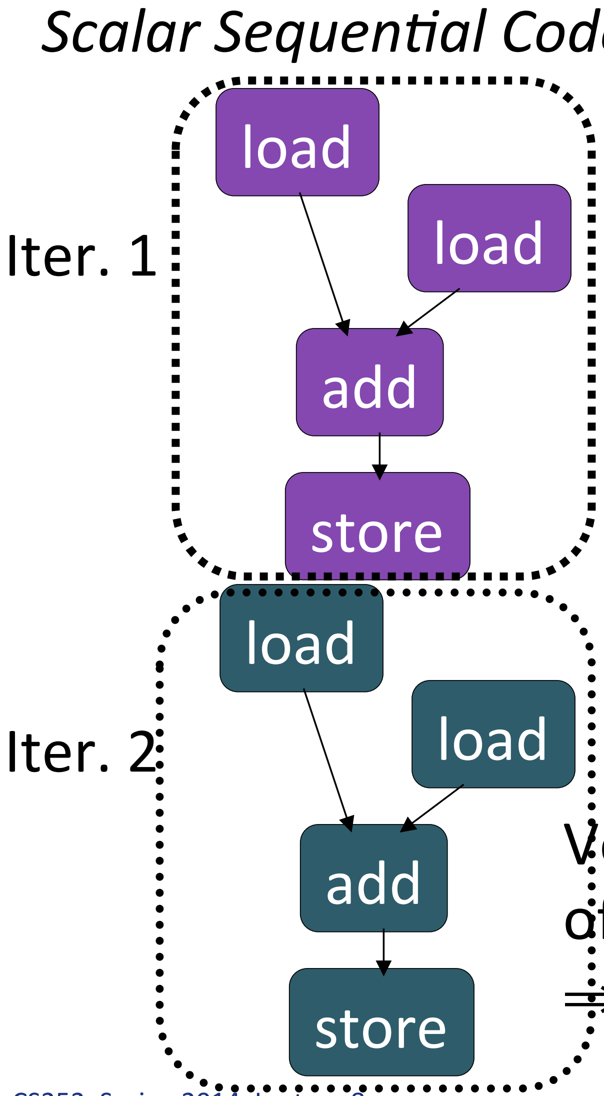
Vector Memory-Memory vs. Vector Register Machines

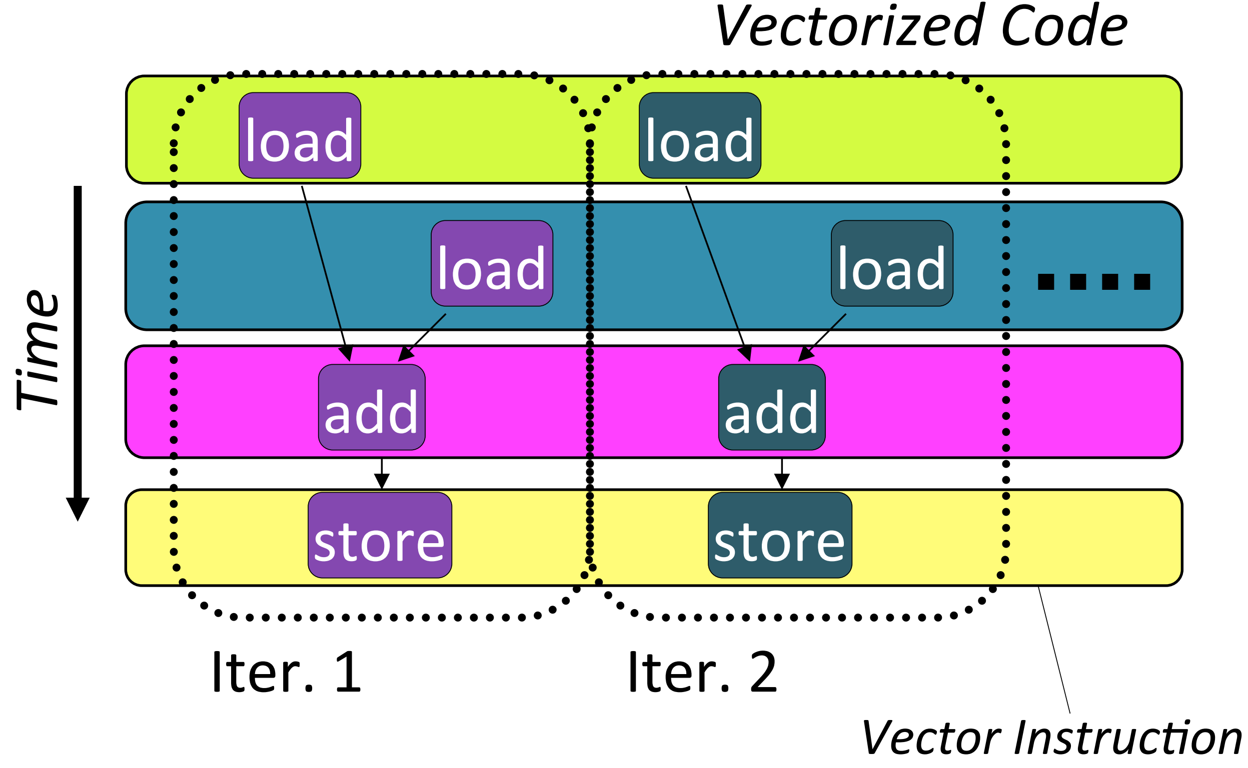
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency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

© Krste Asanovic, 2014

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

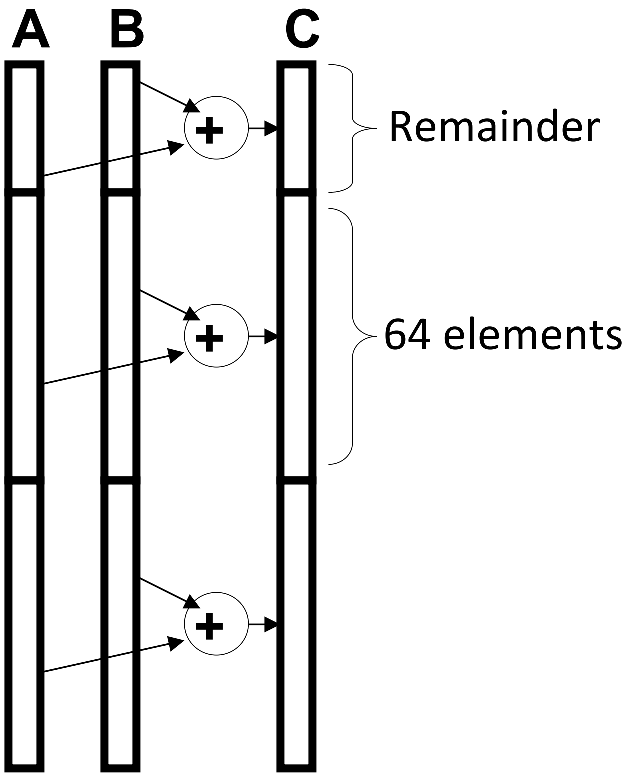
© Krste Asanovic, 2014

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
 setvlr x1        # Do remainder
loop:
 vld v1, xA
 sll x2, x1, 3  # Multiply by 8
 add xA, x2     # Bump pointer
 vld v2, xB
 add xB, x2
 vfadd.d v3, v1, v2
 vsd v3, xC
 add xC, x2
 sub xN, x1     # Subtract elements
 li x1, 64
 setvlr x1      # Reset full length
 bgtz xN, loop # Any more to do?
```

© Krste Asanovic, 2014

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:

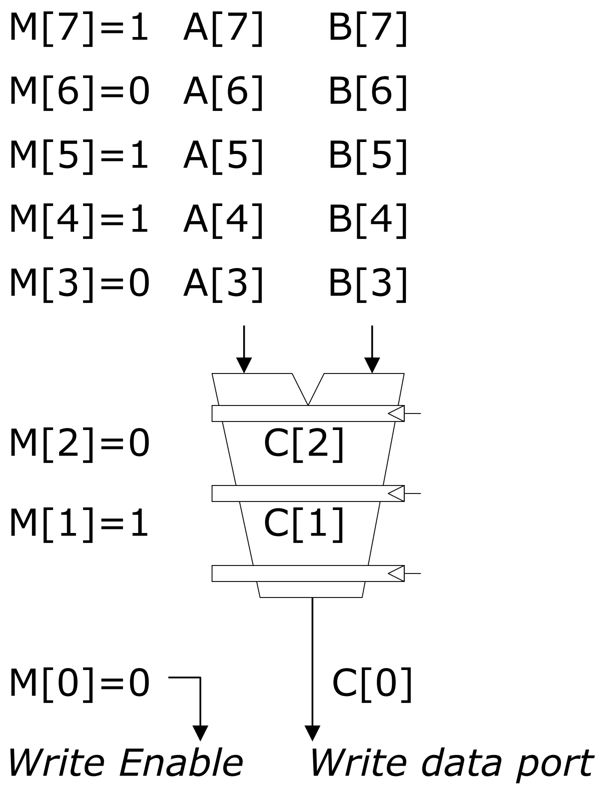
```
cvm               # Turn on all elements
vld vA, xA        # Load entire A vector
vfsgts.d vA, f0 # Set bits in mask register where A>0
vld vA, xB        # Load B vector into A under mask
vsd vA, xA        # Store A back to memory under mask
```

© Krste Asanovic, 2014

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]

M[2]=0 C[2]

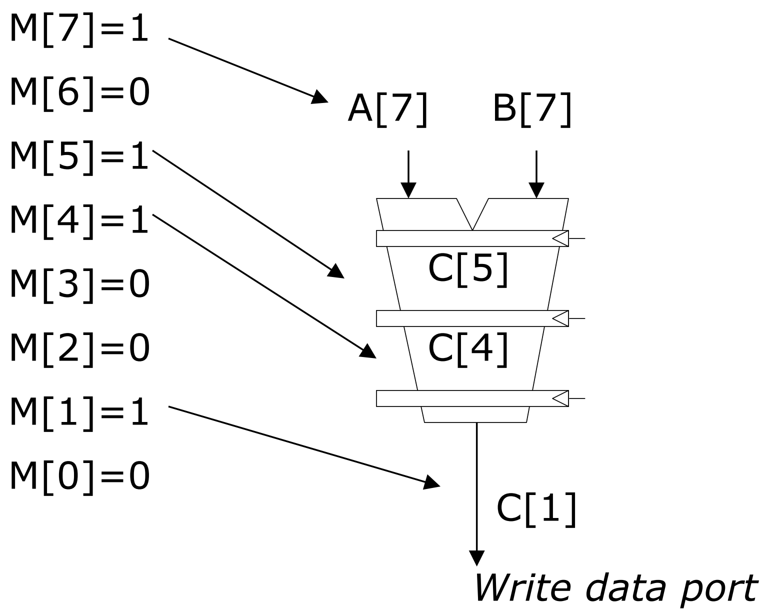
M[1]=1 C[1]

M[0]=0 C[0]

Write Enable *Write data port*

Density-Time Implementation

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



M[7]=1

M[6]=0

M[5]=1

M[4]=1

M[3]=0

M[2]=0

M[1]=1

M[0]=0

A[7] B[7]

C[5]

C[4]

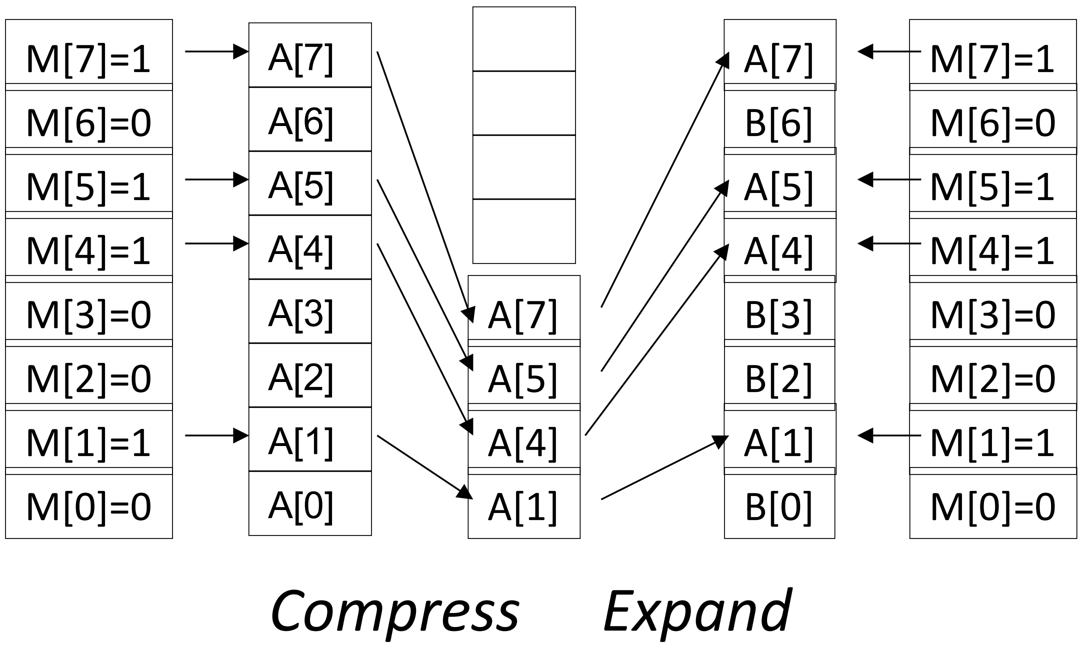
C[1]

Write data port

© Krste Asanovic, 2014

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

© Krste Asanovic, 2014

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)
```

© Krste Asanovic, 2014

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
vdli vC, xC, xD     # Load indirect from rC base
vld vB, xB          # Load B vector
vfadd.d vA,vB,vC    # Do add
vsd vA, xA          # Store result
```

Vector 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
vldi vA, xA, vB   # Gather initial A values
vadd vA, vA, 1    # Increment
vsdi vA, xA, vB   # Scatter incremented values
```

© Krste Asanovic, 2014

Vector Memory Models

- Most vector machines have a very relaxed memory model, e.g.

```
vsd 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!

- Requires explicit memory barrier or fence

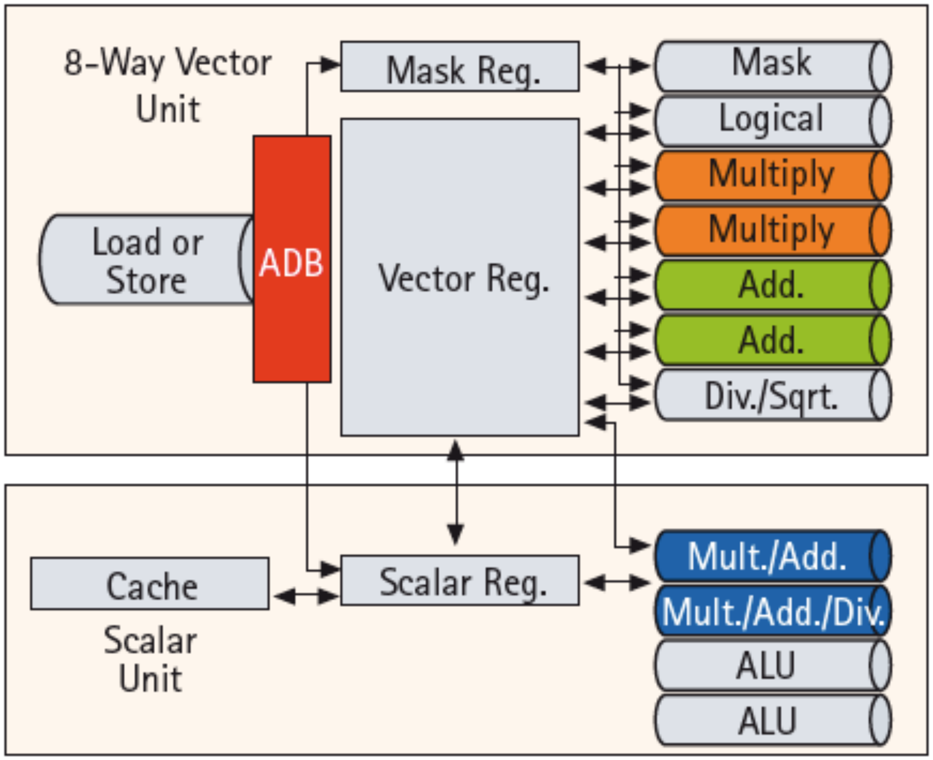
```
vsd v1, x1    # Store vector to x1
fence.vs.vl   # Enforce ordering s->l
vld v2, x1    # Load vector from x1
```

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

© Krste Asanovic, 2014

A Recent Vector Super: NEC SX-9 (2008)



[©NEC]

- 65nm CMOS technology
- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/ cycle)
- Scalar unit (1.6 GHz)
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache

- Memory system provides 256GB/s DRAM bandwidth per CPU
- Up to 16 CPUs and up to 1TB DRAM form shared-memory *node*
  - total of 4TB/s bandwidth to shared DRAM memory
- Up to 512 nodes connected via 128GB/s network links (message passing between nodes)

[New announcement SX-ACE, 4x16-lane vector CPUs on one chip]

Acknowledgements

- This course is partly inspired by previous MIT 6.823 and Berkeley CS252 computer architecture courses created by my collaborators and colleagues:
    - Arvind (MIT)
    - Joel Emer (Intel/MIT)
    - James Hoe (CMU)
    - John Kubiatowicz (UCB)
    - David Patterson (UCB)

© Krste Asanovic, 2014