CS 654 Advanced Computer Architecture

Lec. 11: Vector Computers

Peter Kemper

Adapted from the slides of:
Krste Asanovic
(krste@mit.edu)
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

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

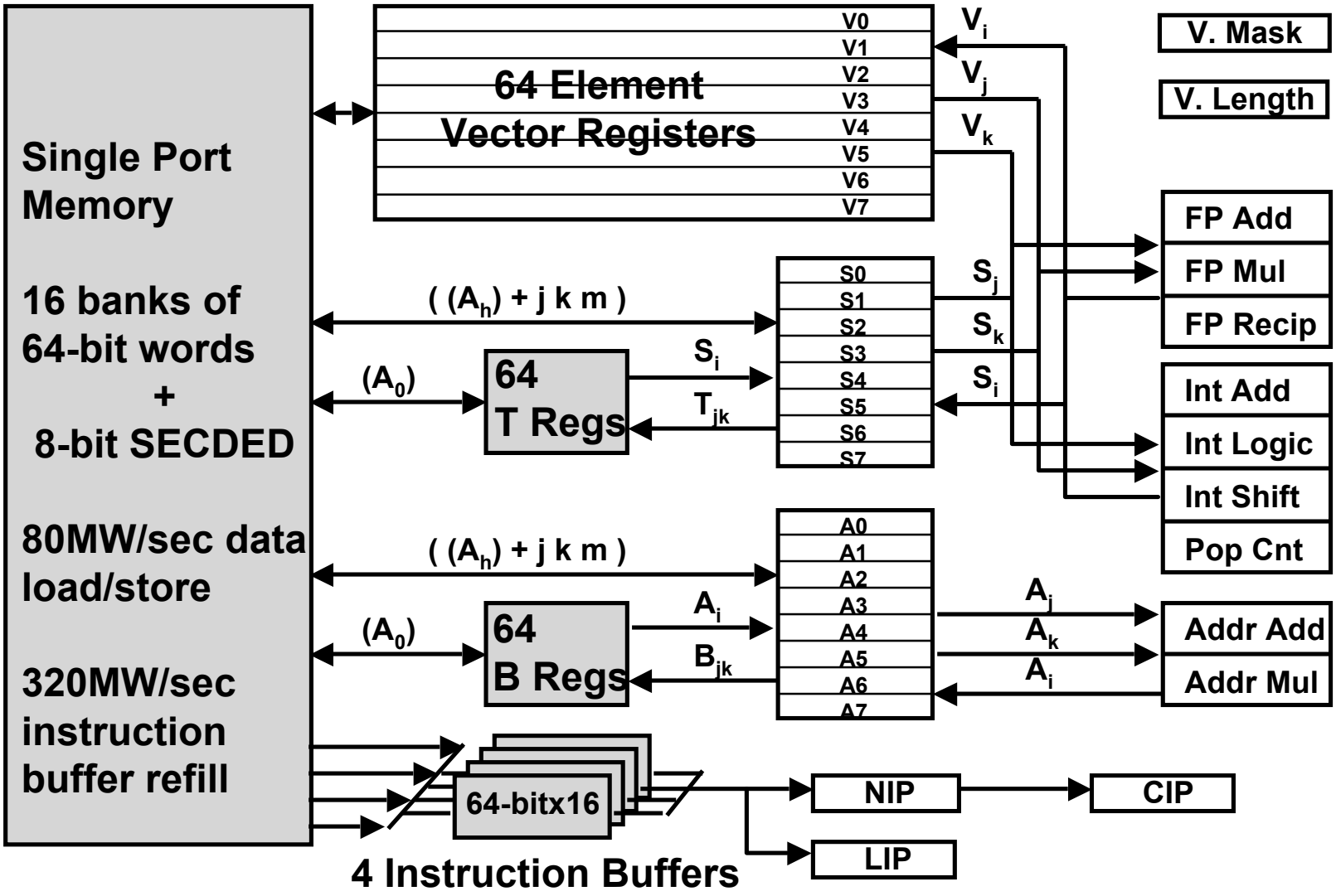
Epitomized by Cray-1, 1976:

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**

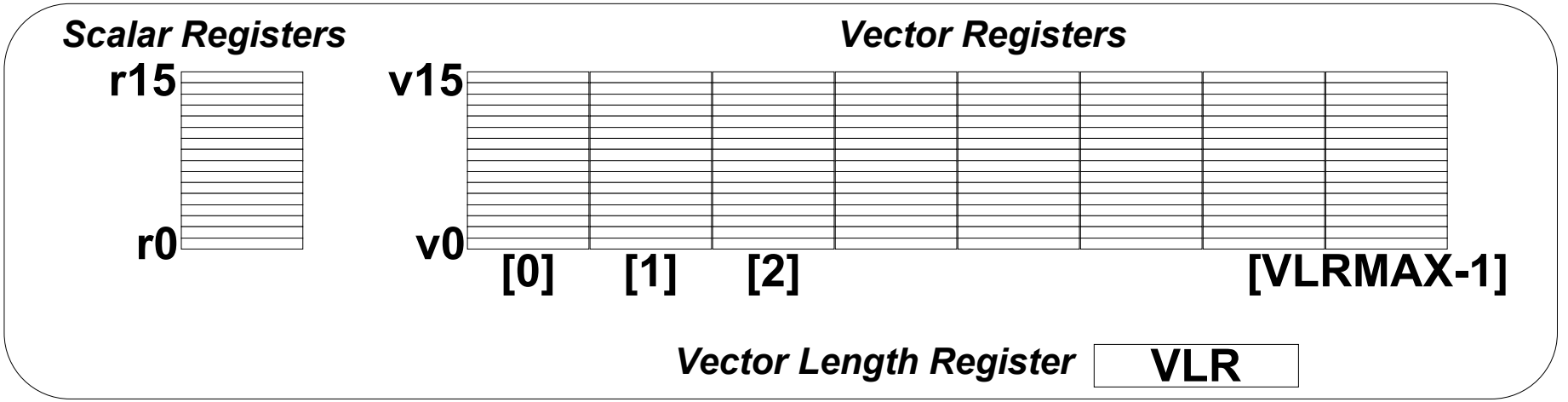
Cray-1 (1976)

Cray-1 (1976)



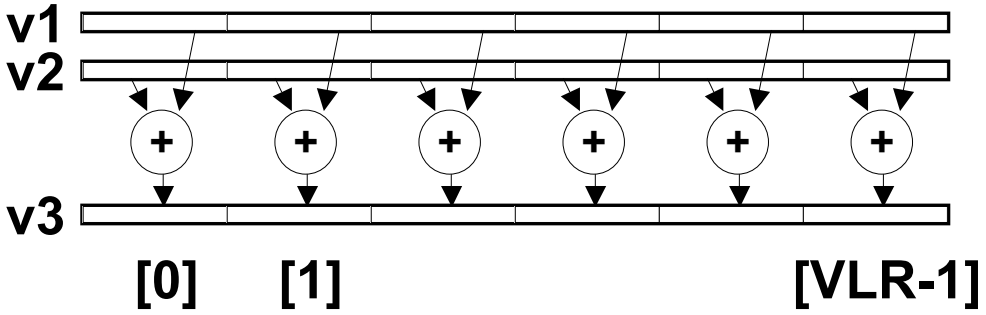
memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Programming Model



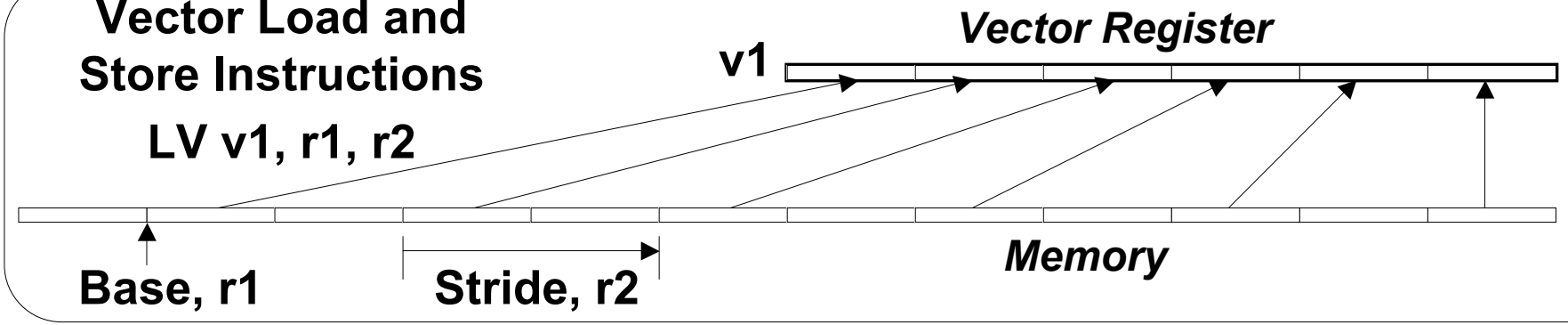
Vector Arithmetic Instructions

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
```

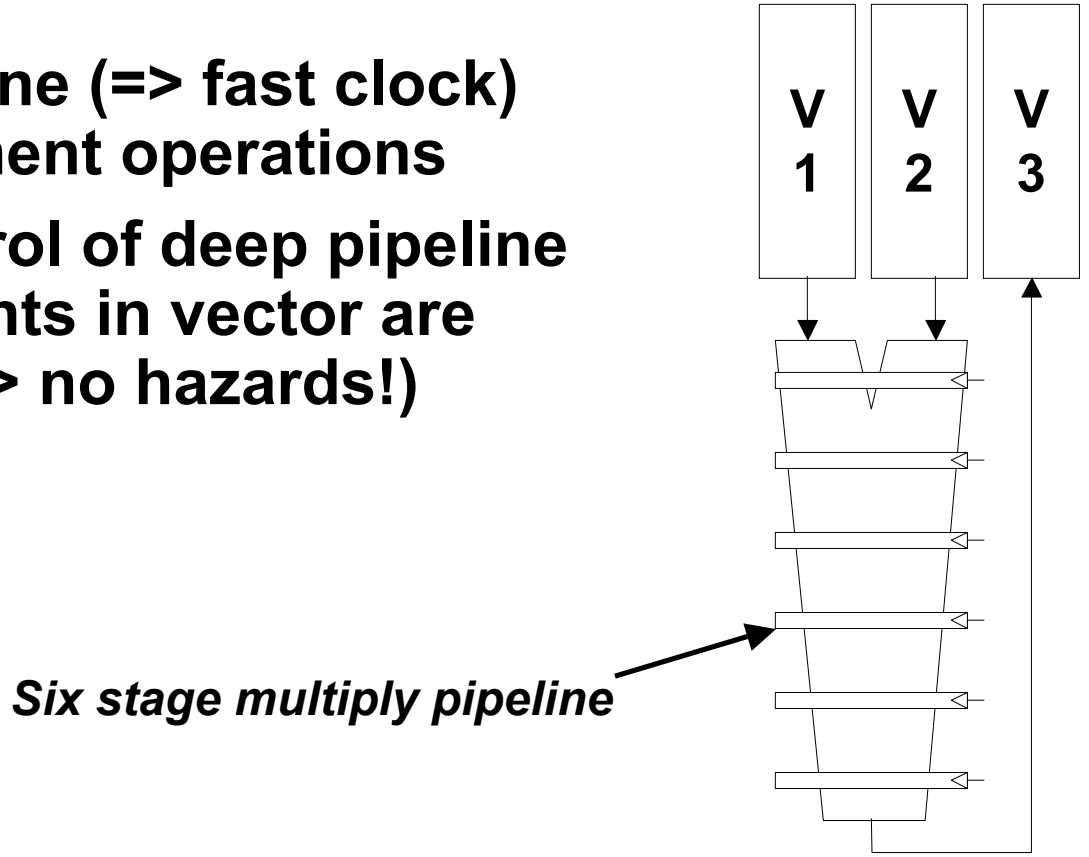
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

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!)**

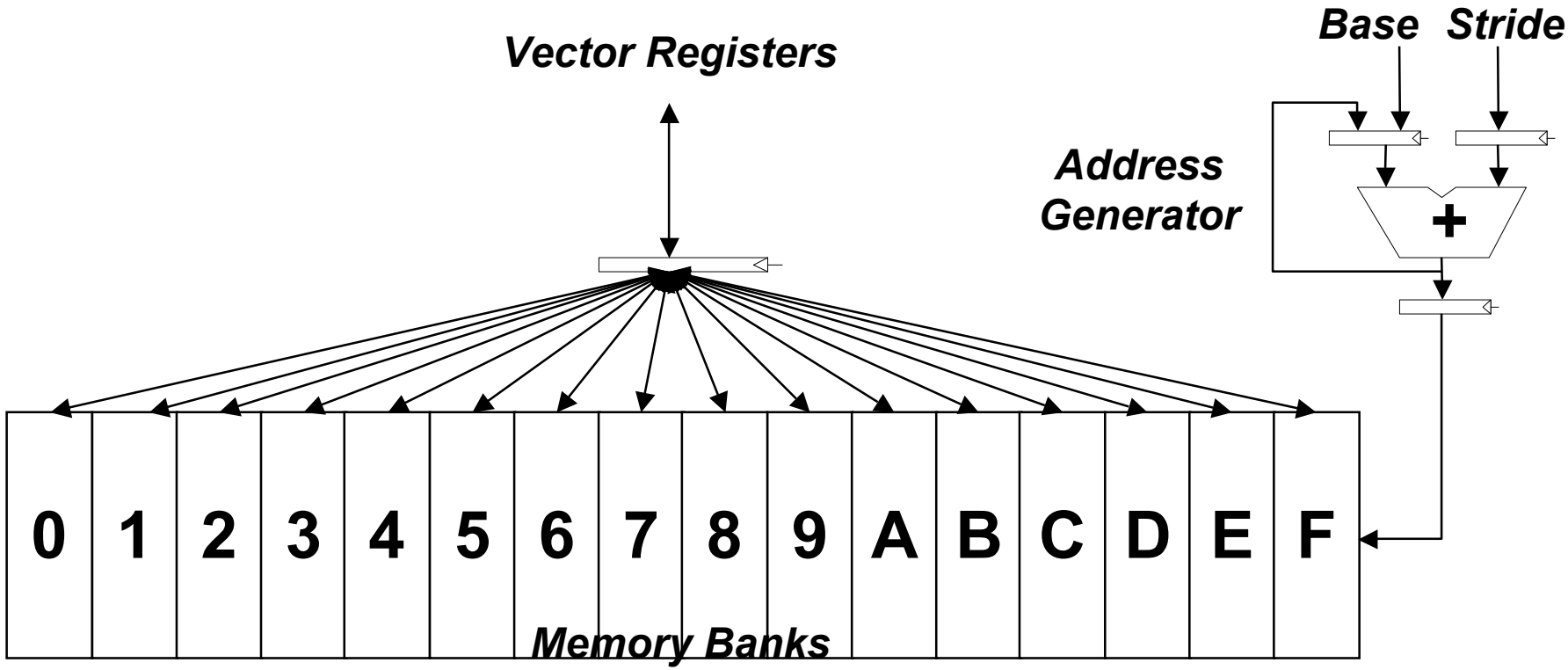



```
V3 <- v1 * v2
```

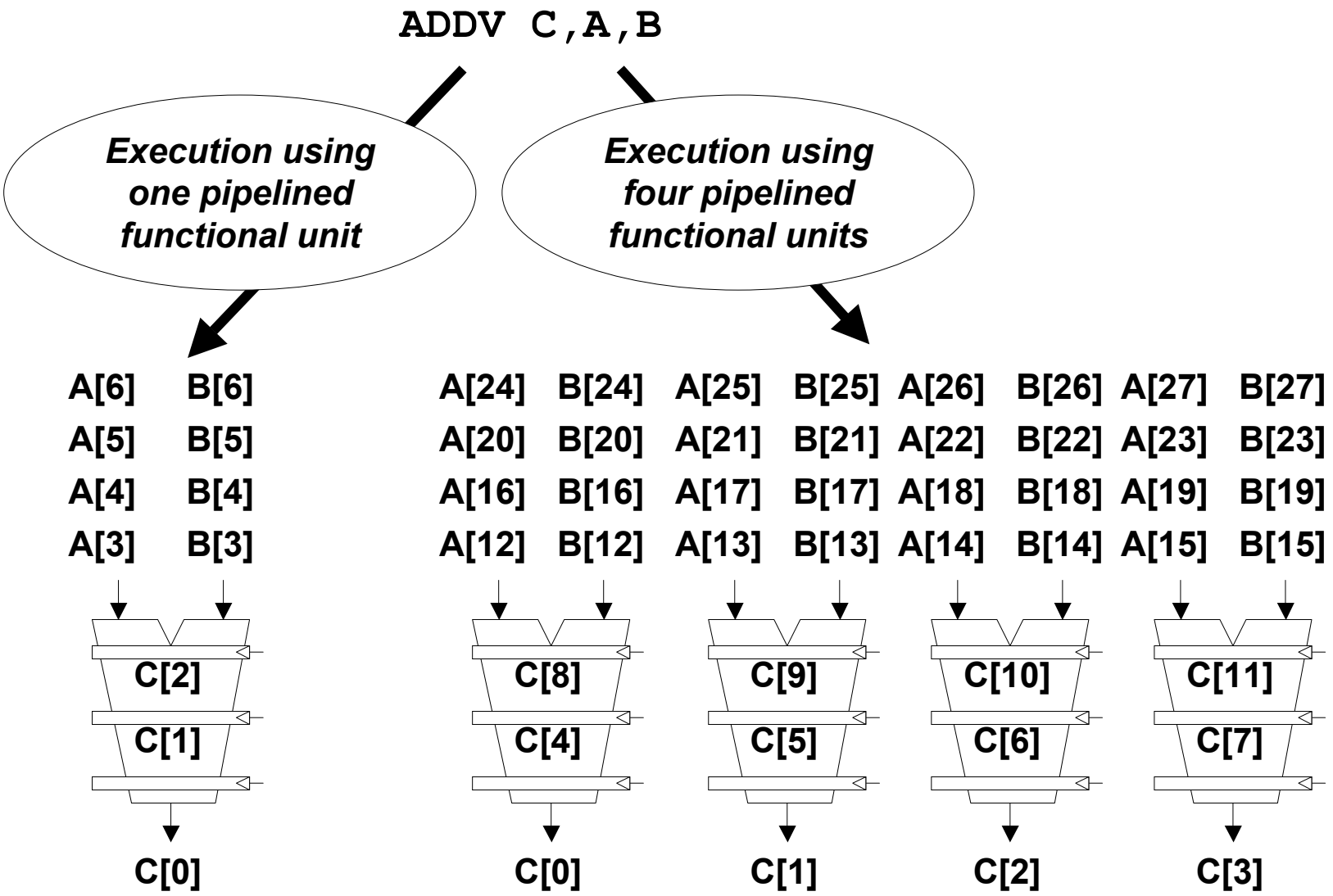
Vector Memory System

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

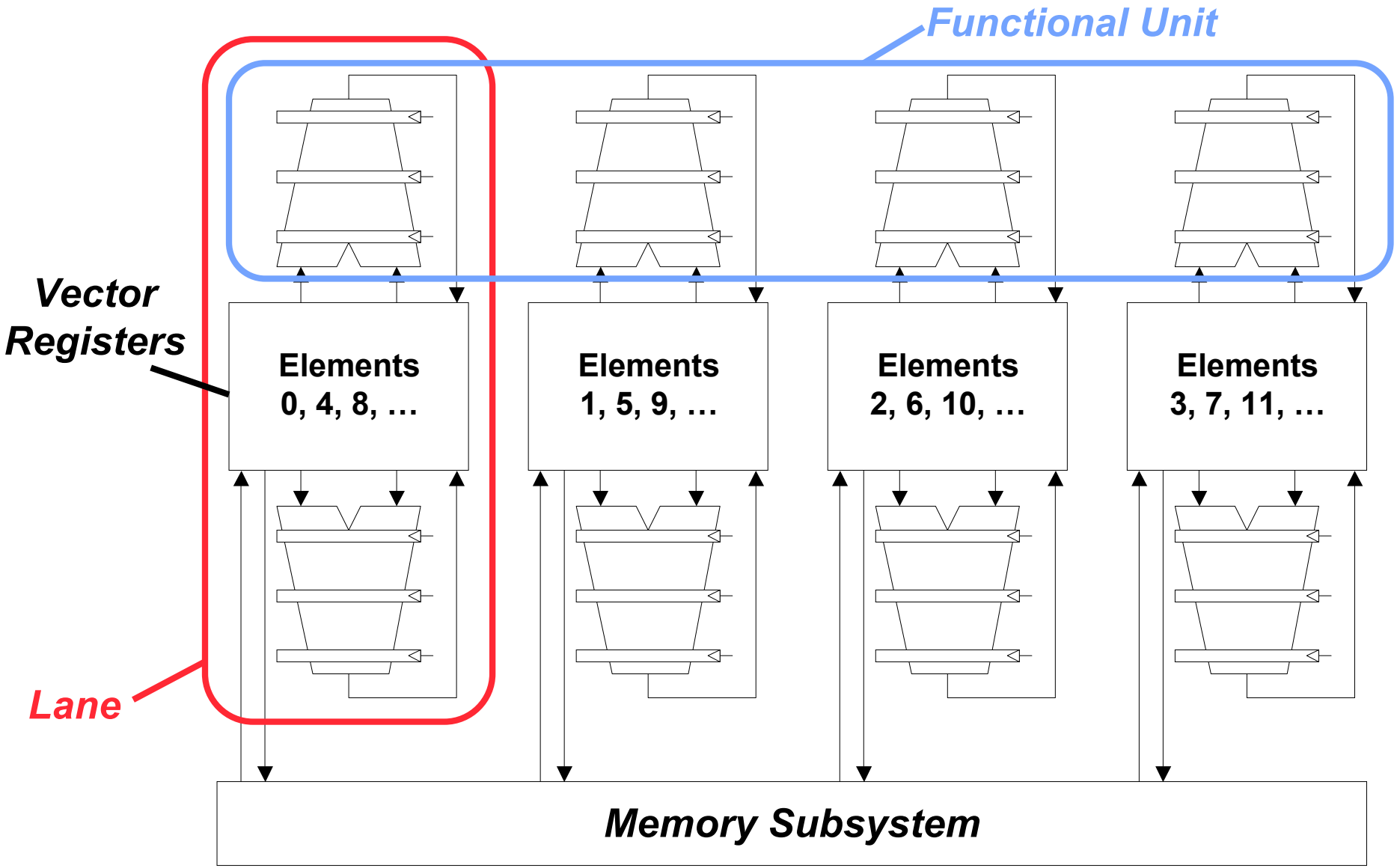
- ***Bank busy time*: Cycles between accesses to same bank**



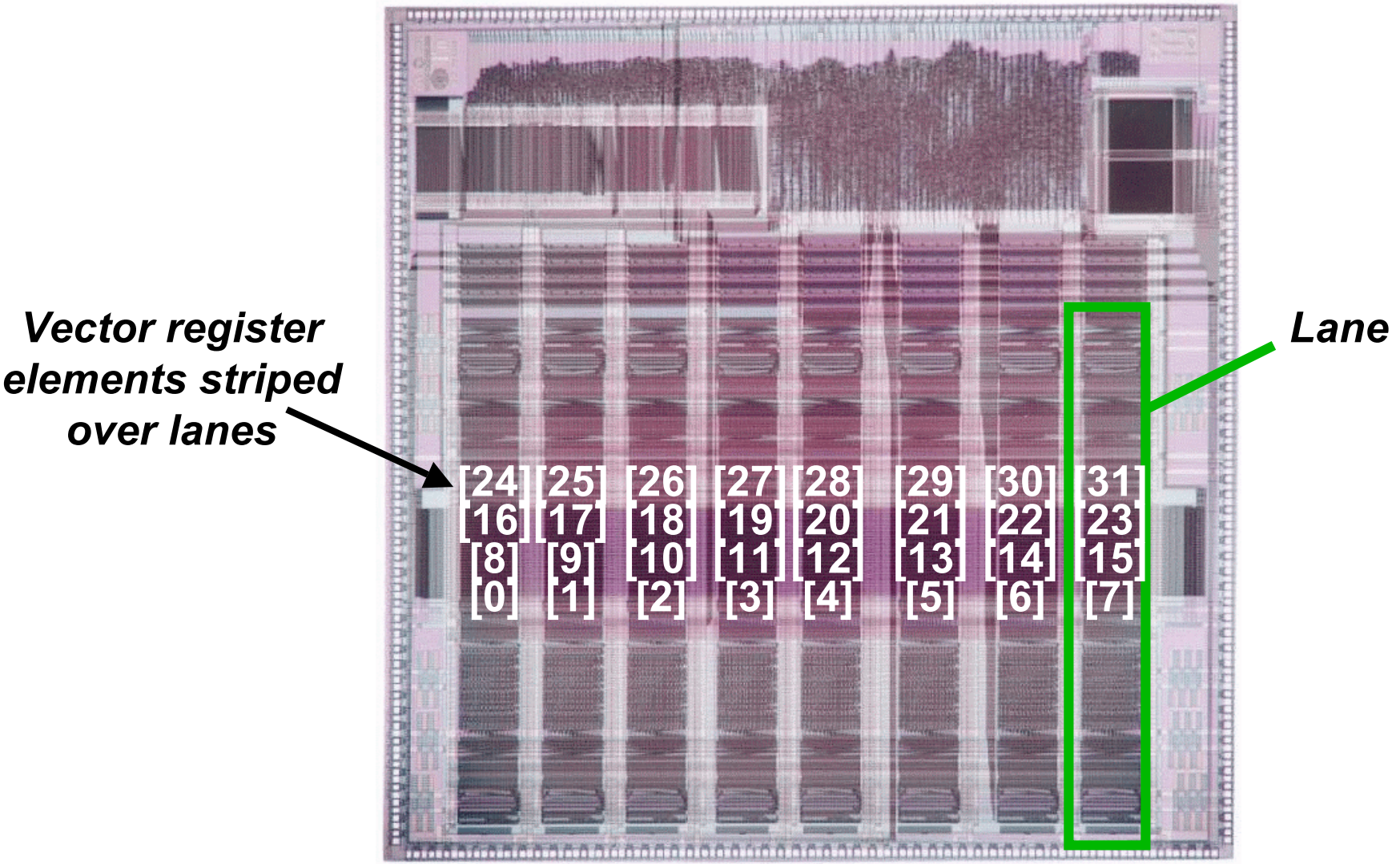
Vector Instruction Execution



Vector Unit Structure



T0 Vector Microprocessor (1995)

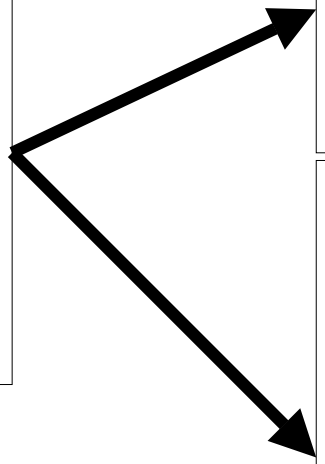


Vector Memory-Memory versus Vector Register Machines

- 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

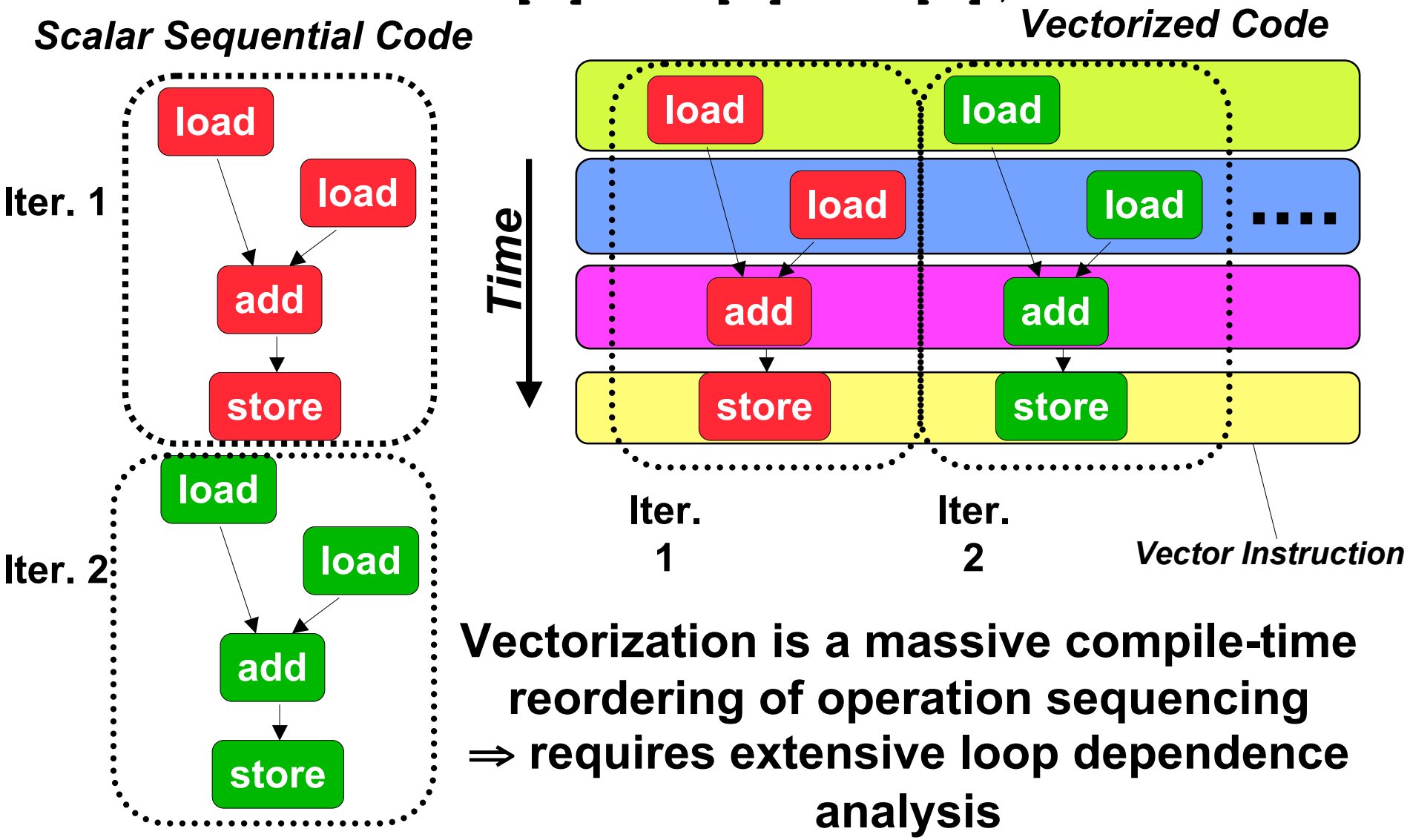
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  –
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  –
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 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];
```



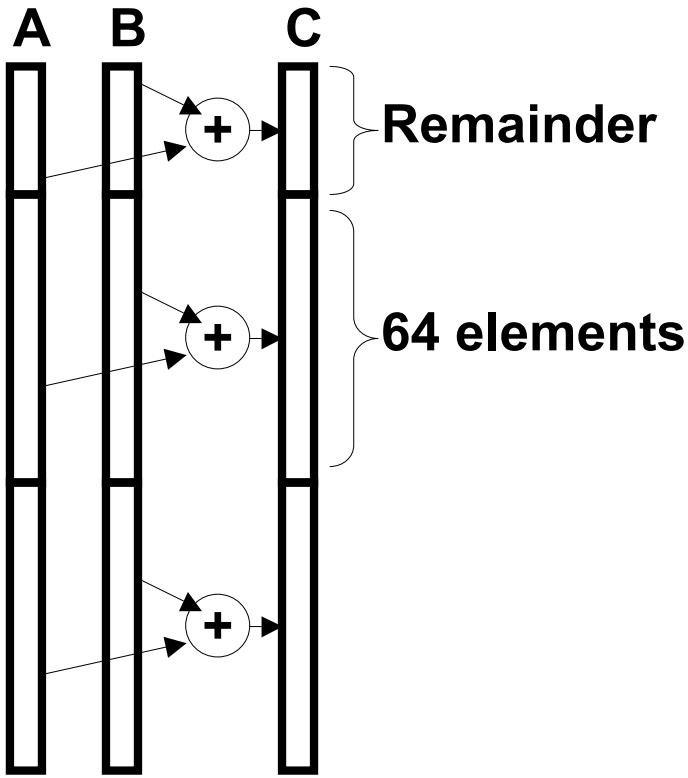
Vectorization is a massive compile-time reordering of operation sequencing ⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

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

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

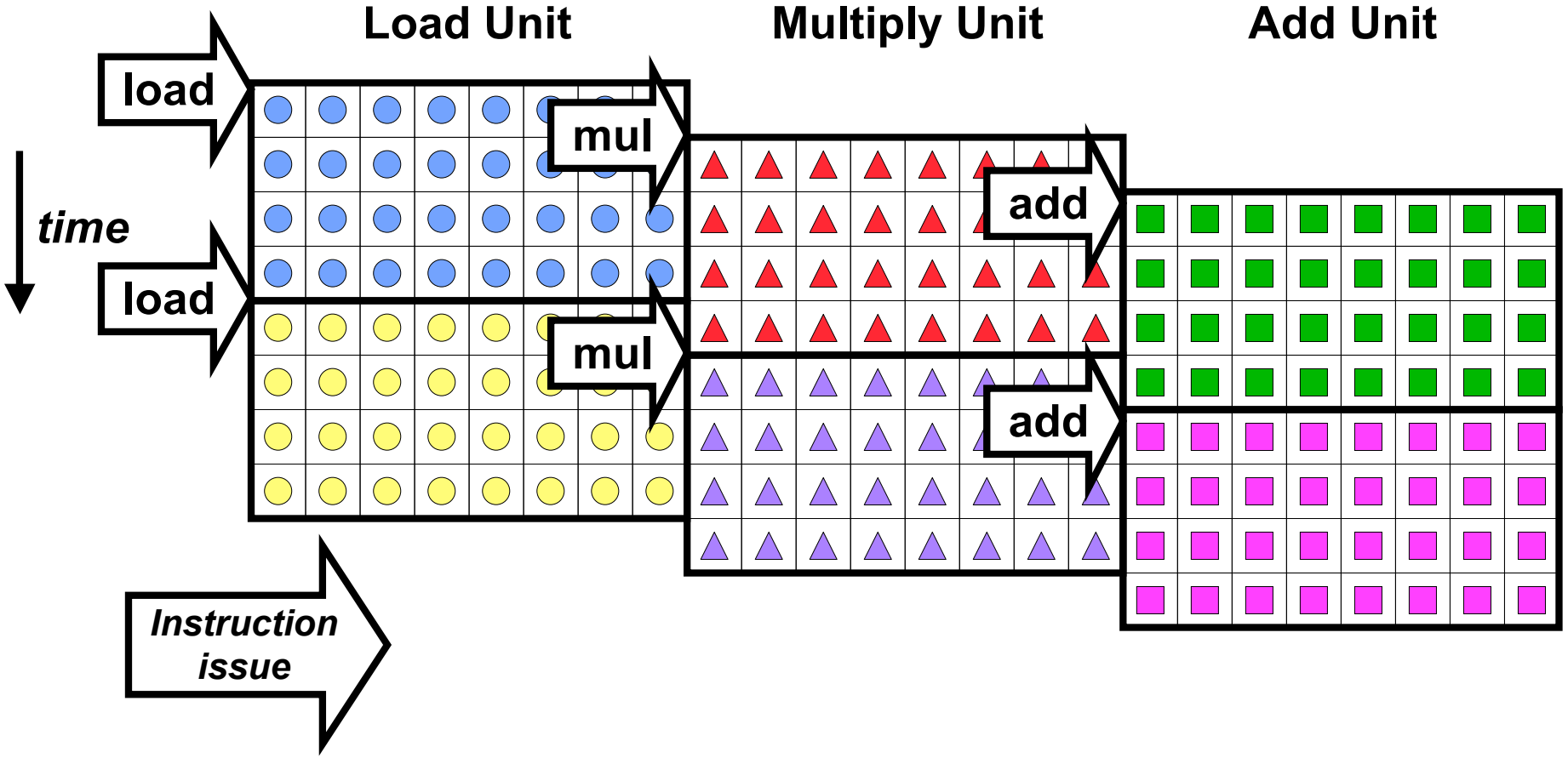



```
        ANDI R1, N, 63    # N mod 64
        MTC1 VLR, R1      # Do remainder
loop:
        LV V1, RA
        DSLL R2, R1, 3    # Multiply by 8
        DADDU RA, RA, R2  # Bump pointer
        LV V2, RB
        DADDU RB, RB, R2
        ADDV.D V3, V1, V2
        SV V3, RC
        DADDU RC, RC, R2
        DSUBU N, N, R1 # Subtract elements
        LI R1, 64
        MTC1 VLR, R1    # Reset full length
        BGTZ N, loop    # Any more to do?
```

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

– example machine has 32 elements per vector register and 8 lanes

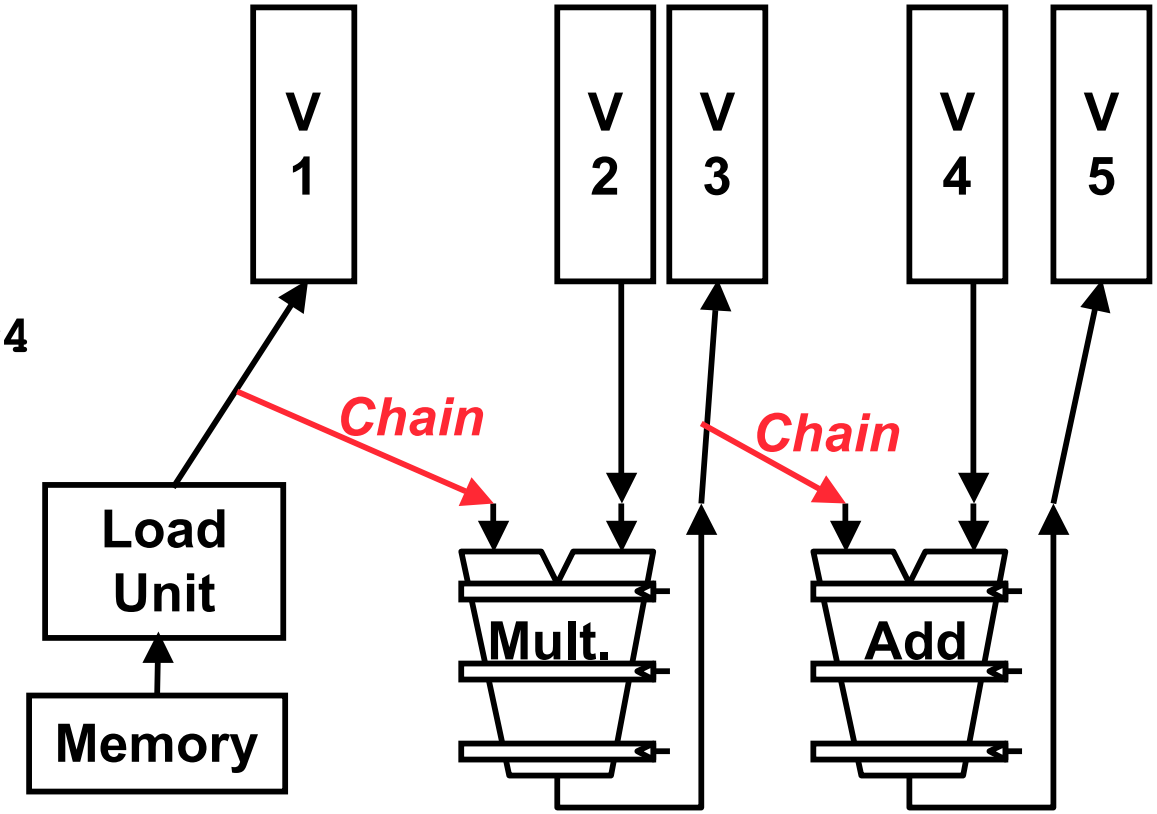


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

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

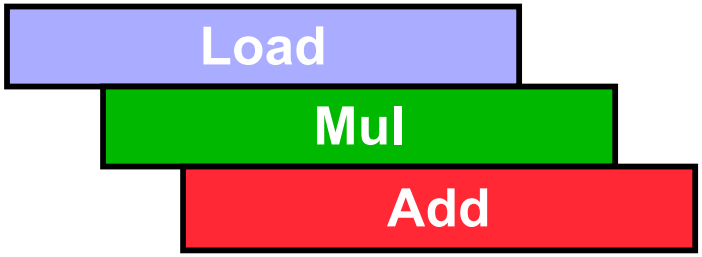
```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



Vector Chaining Advantage

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

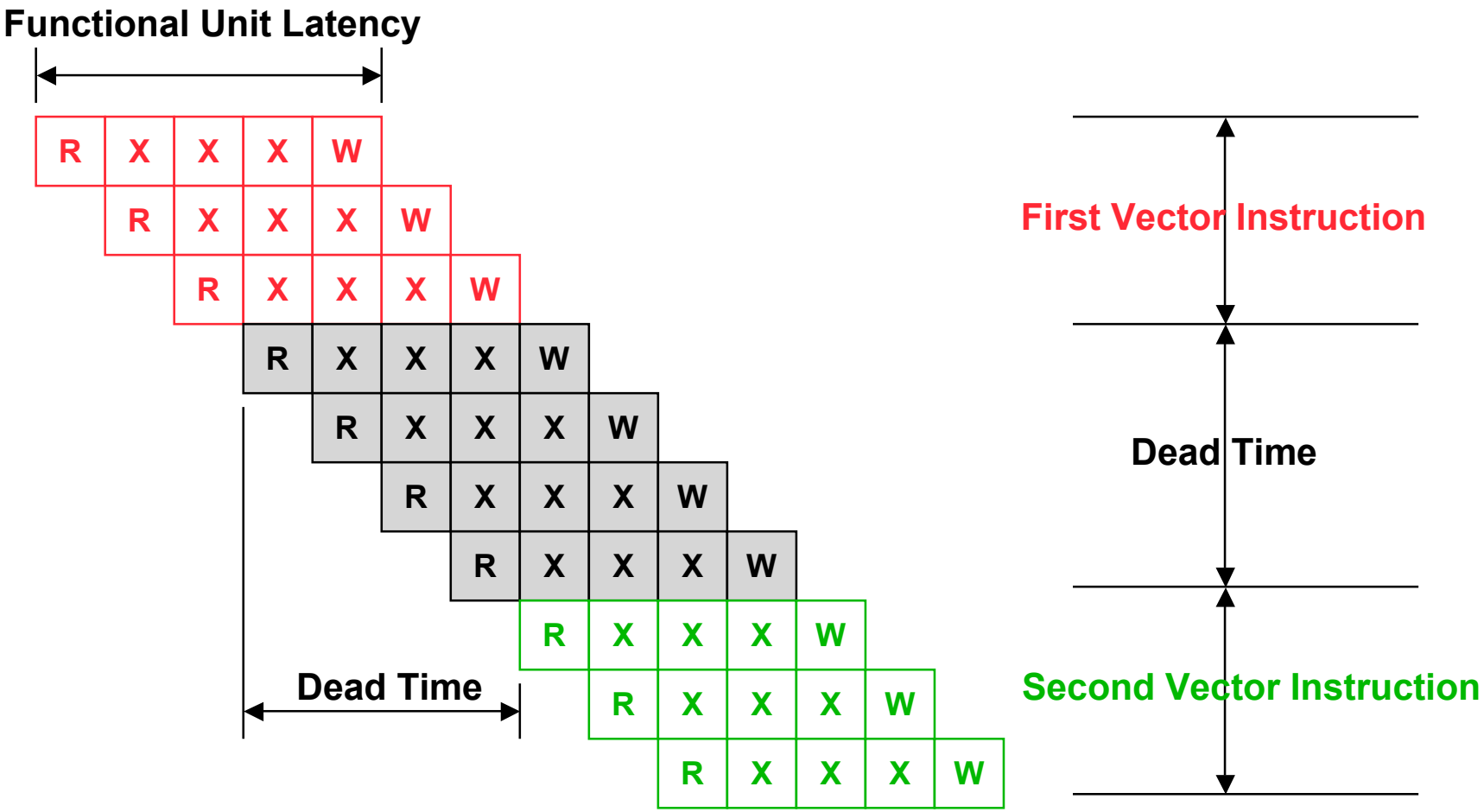
- With chaining, can start dependent instruction as soon as first result appears



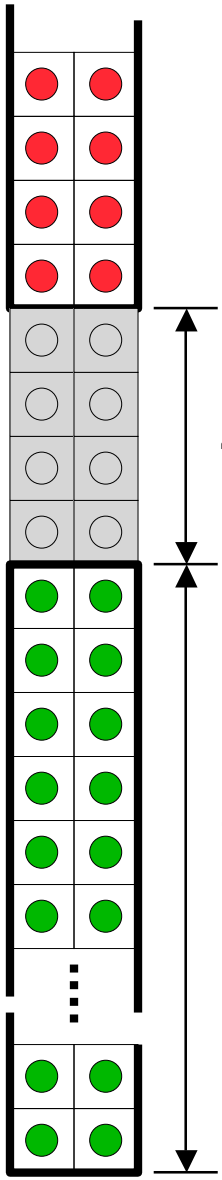
Vector Startup

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

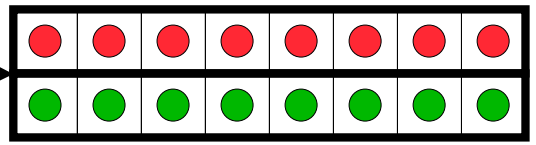
64 cycles active

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

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*)

```
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
```

Vector Scatter/Gather

Scatter example:

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

Is following a correct translation?

```
LV vB, rB          # Load indices in B vector
LVI vA, rA, vB     # Gather initial A values
ADDV vA, vA, 1     # Increment
SVI vA, rA, vB     # Scatter incremented values
```

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 NOP at elements where mask bit is clear

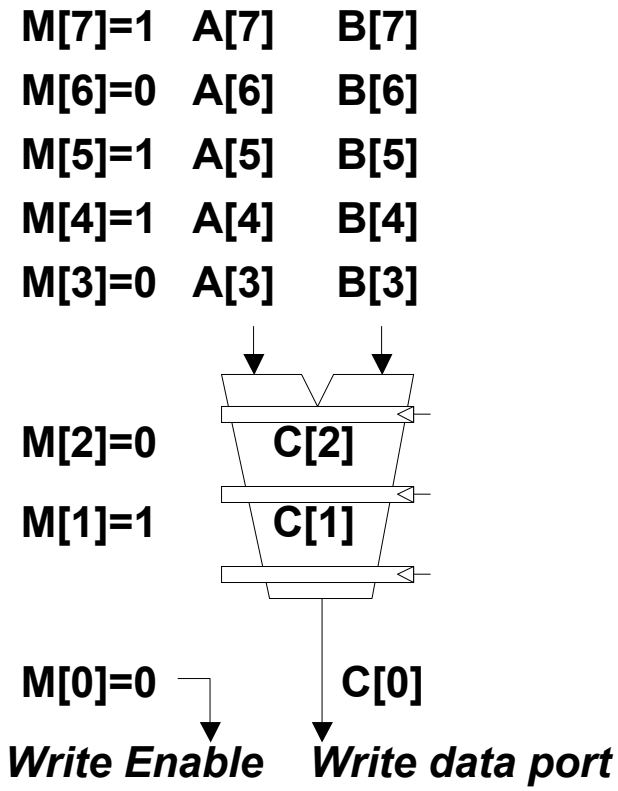
Code example:

```
CVM              # Turn on all elements
LV vA, rA        # Load entire A vector
SGTVS.D vA, F0   # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

Masked Vector Instructions

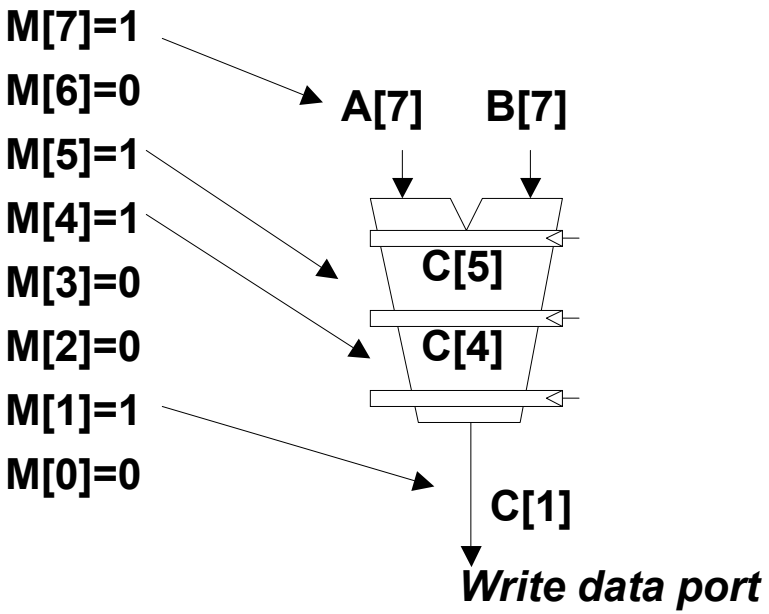
Simple Implementation

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



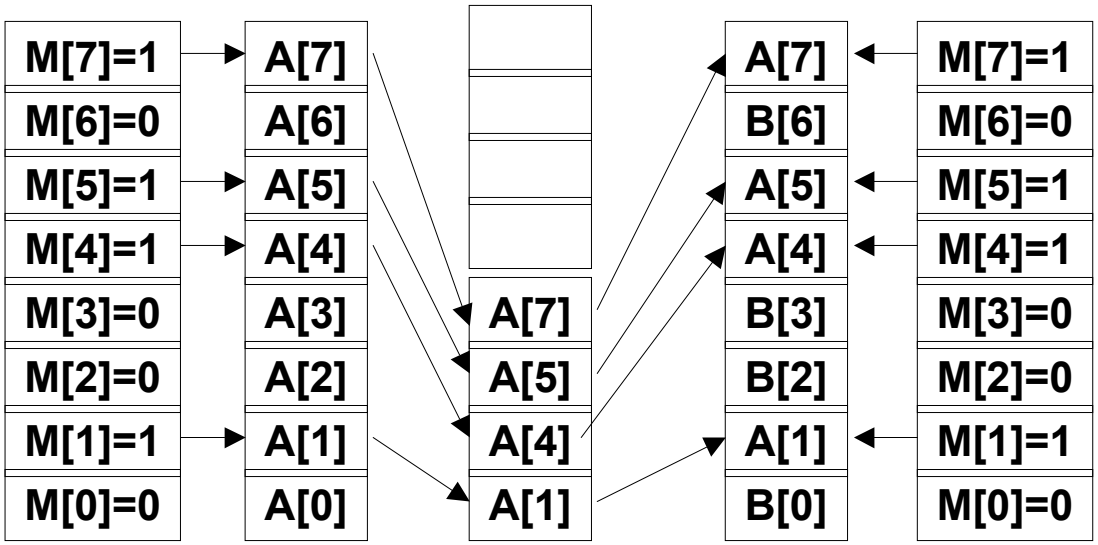
Density-Time Implementation

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



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

A Modern Vector Super: NEC SX-6 (2003)

- CMOS Technology
  - 500 MHz CPU, fits on single chip
  - SDRAM main memory (up to 64GB)
- Scalar unit
  - 4-way superscalar with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- Vector unit
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
  - 1 load & store unit (32x8 byte accesses/cycle)
  - 32 GB/s memory bandwidth per processor
- SMP structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB/s shared memory bandwidth (4096 interleaved banks)

Multimedia Extensions

- Very short vectors added to existing ISAs for micros
- Usually 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit registers (Altivec, SSE2)
- 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