Vector Computers

Krste Asanovic
Laboratory for Computer Science
Massachusetts Institute of Technology

Supercomputers

Definition of a supercomputer:

- Fastest machine in world at given task
- Any machine costing $30M+
- A device to turn a compute-bound problem into an I/O bound problem
- 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

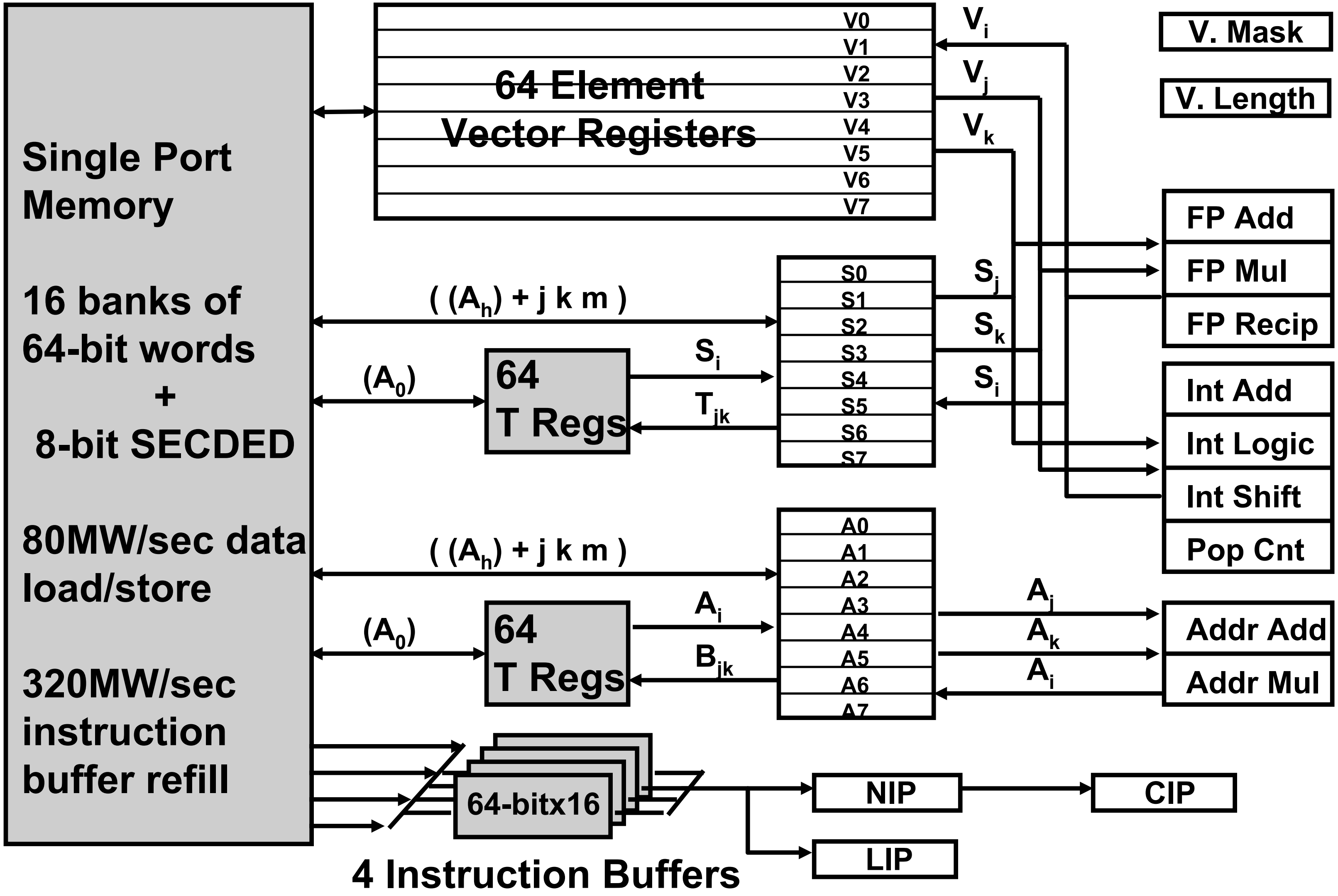
(Epitomized by Cray-1, 1976)

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

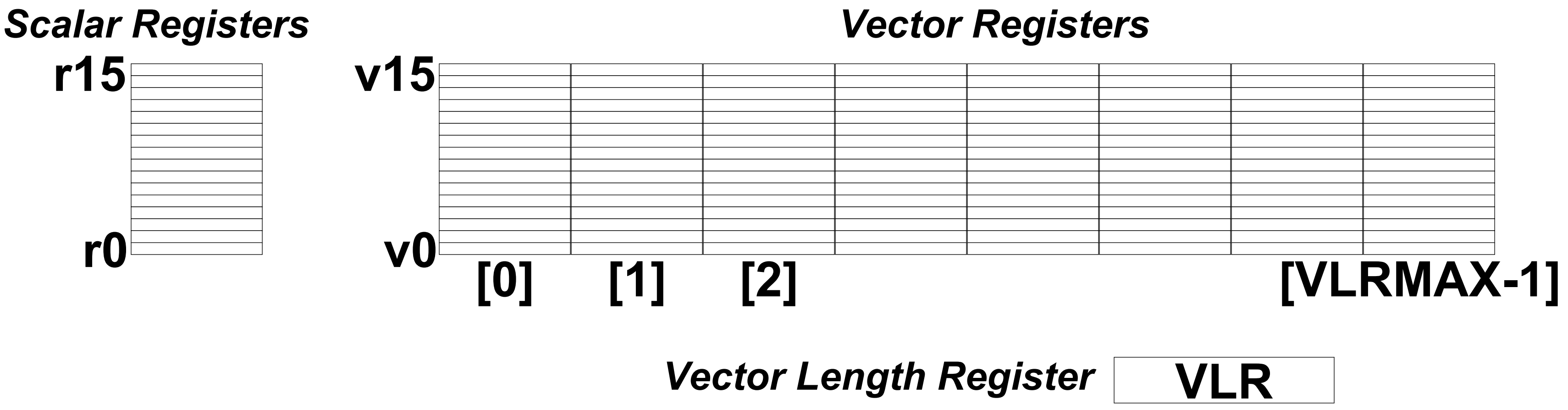
Cray-1 (1976)

Cray-1 (1976)



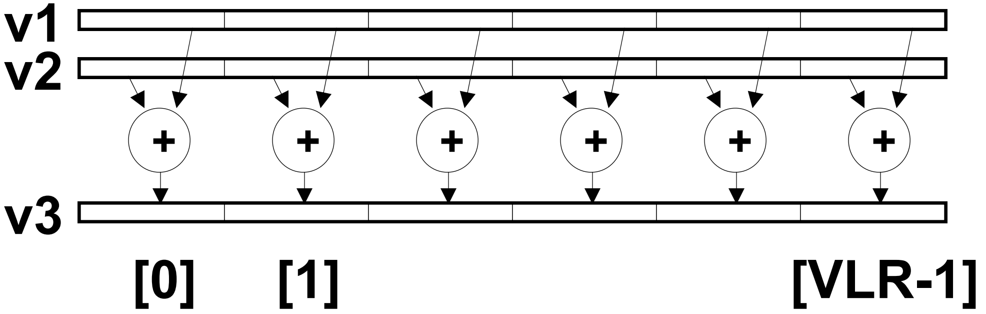
memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Programming Model



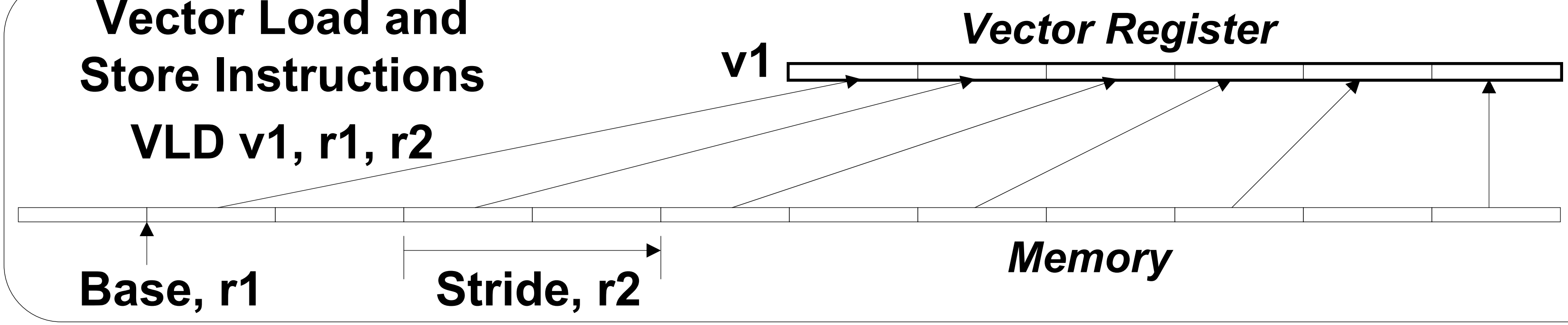
Vector Arithmetic Instructions

VADD v3, v1, v2



Vector Load and Store Instructions

VLD 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:
  ld f1, 0(r1)
  ld f2, 0(r2)
  fadd f3, f1, f2
  st f3, 0(r3)
  add r1, r1, #1
  add r2, r2, #1
  add r3, r3, #1
  sub r4, #1
  bnez r4, loop
```

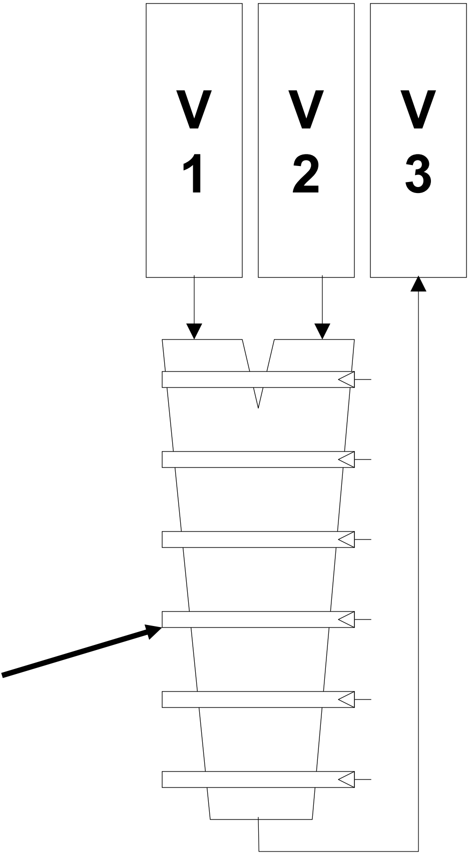
```
# Vector Code
  li vlr, #64
  lv v1, r1, #1
  lv v2, r2, #1
  faddv v3, v1, v2
  sv v3, r3, #1
```

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!)



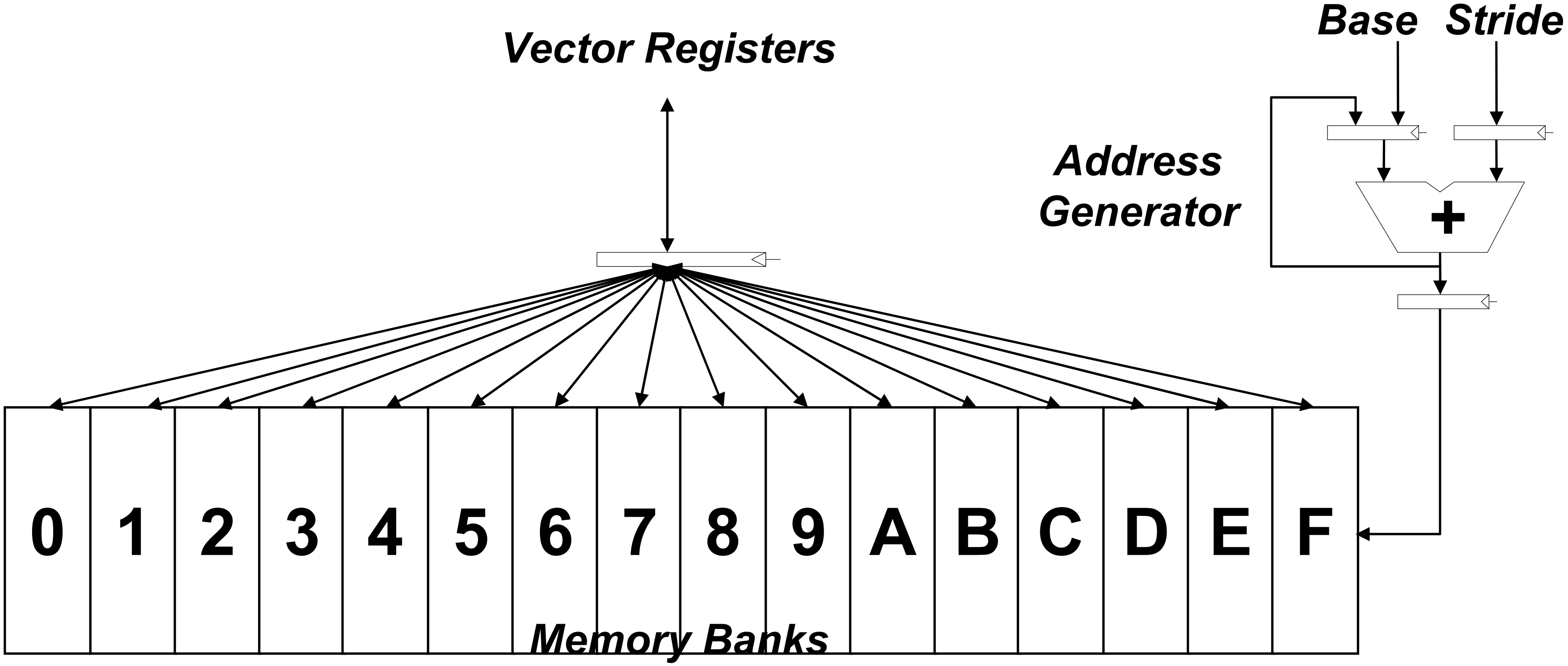
Six stage multiply pipeline

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

Vector Memory System

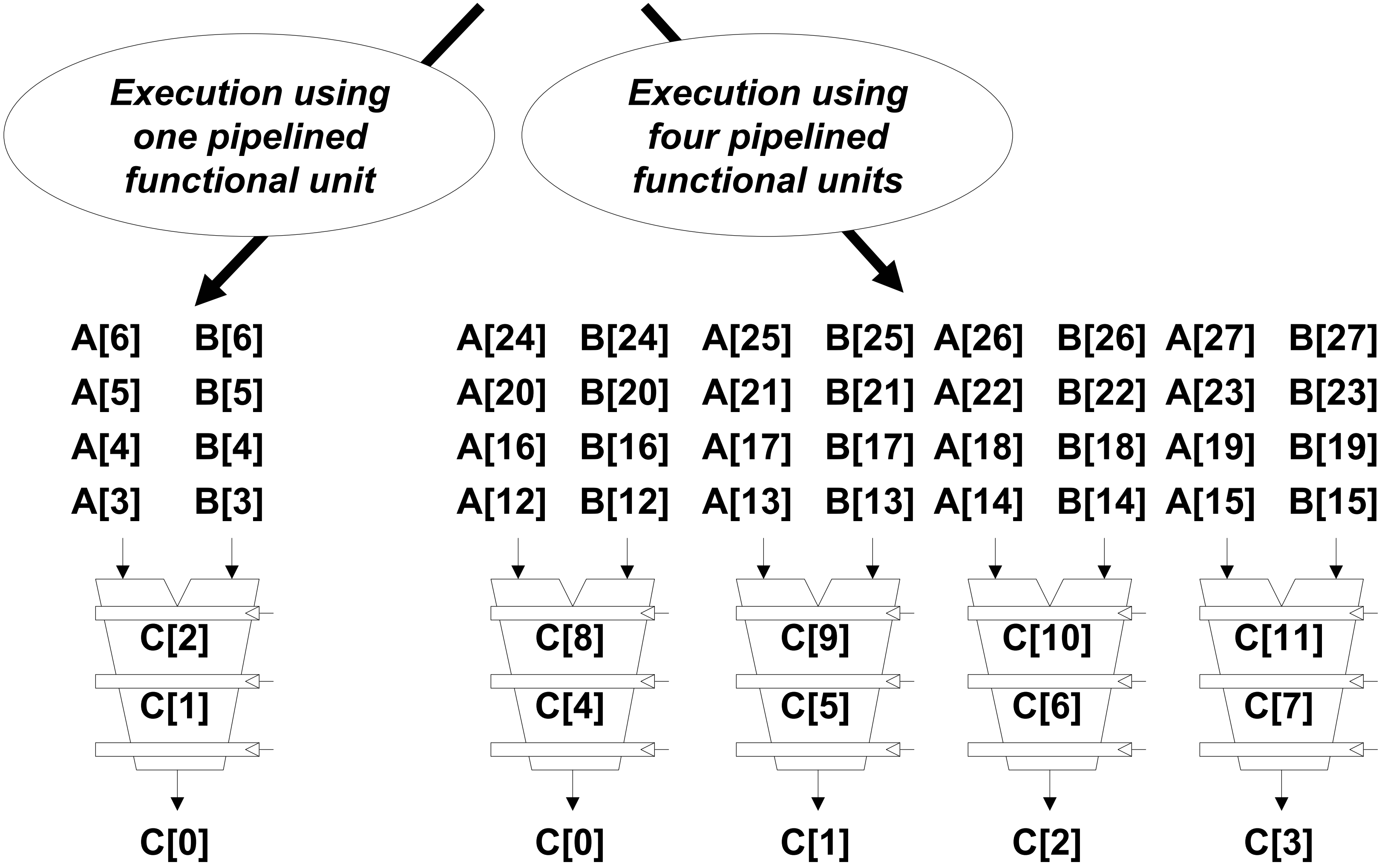
Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

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



Vector Instruction Execution

`VADD C,A,B`

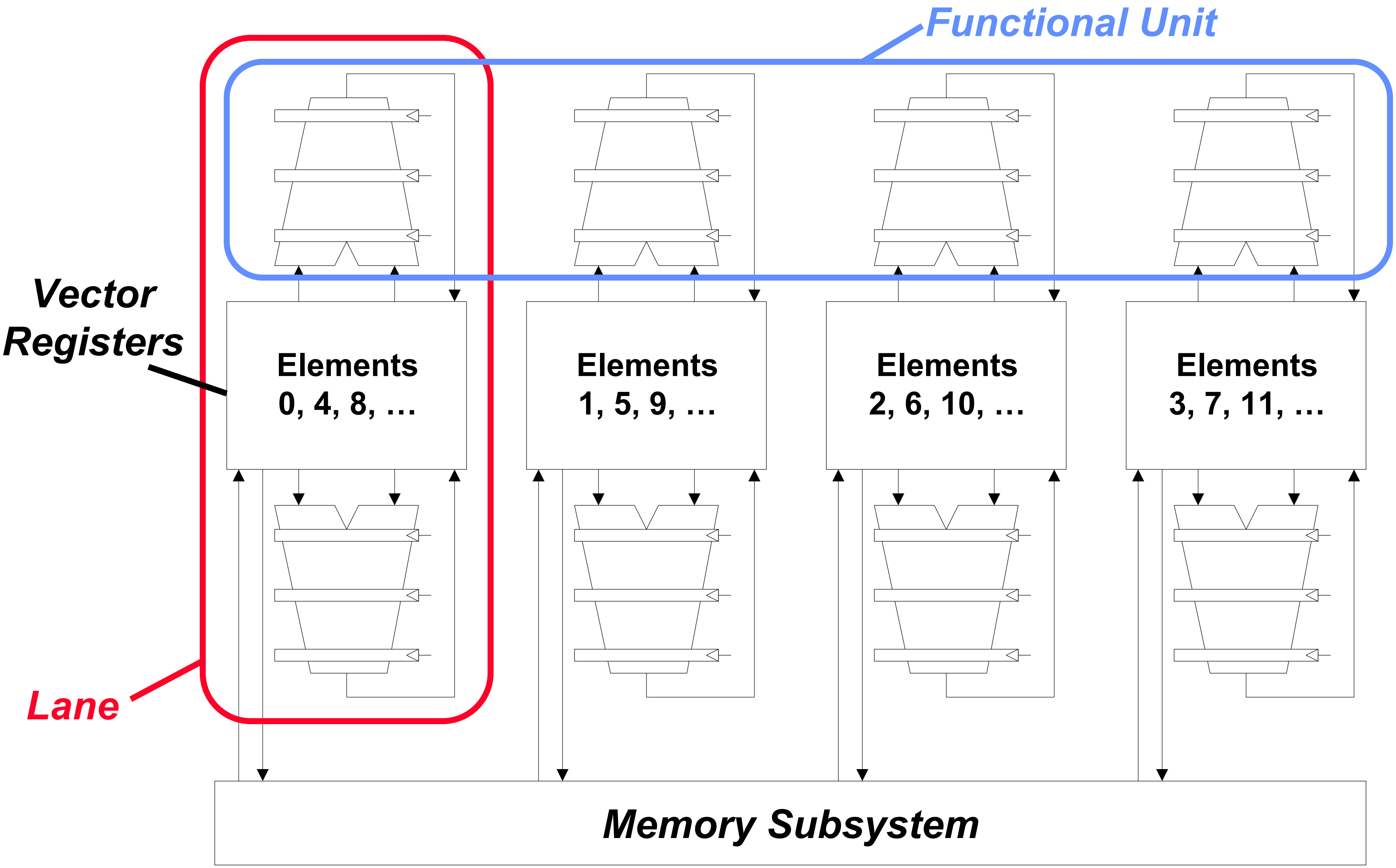


Execution using one pipelined functional unit

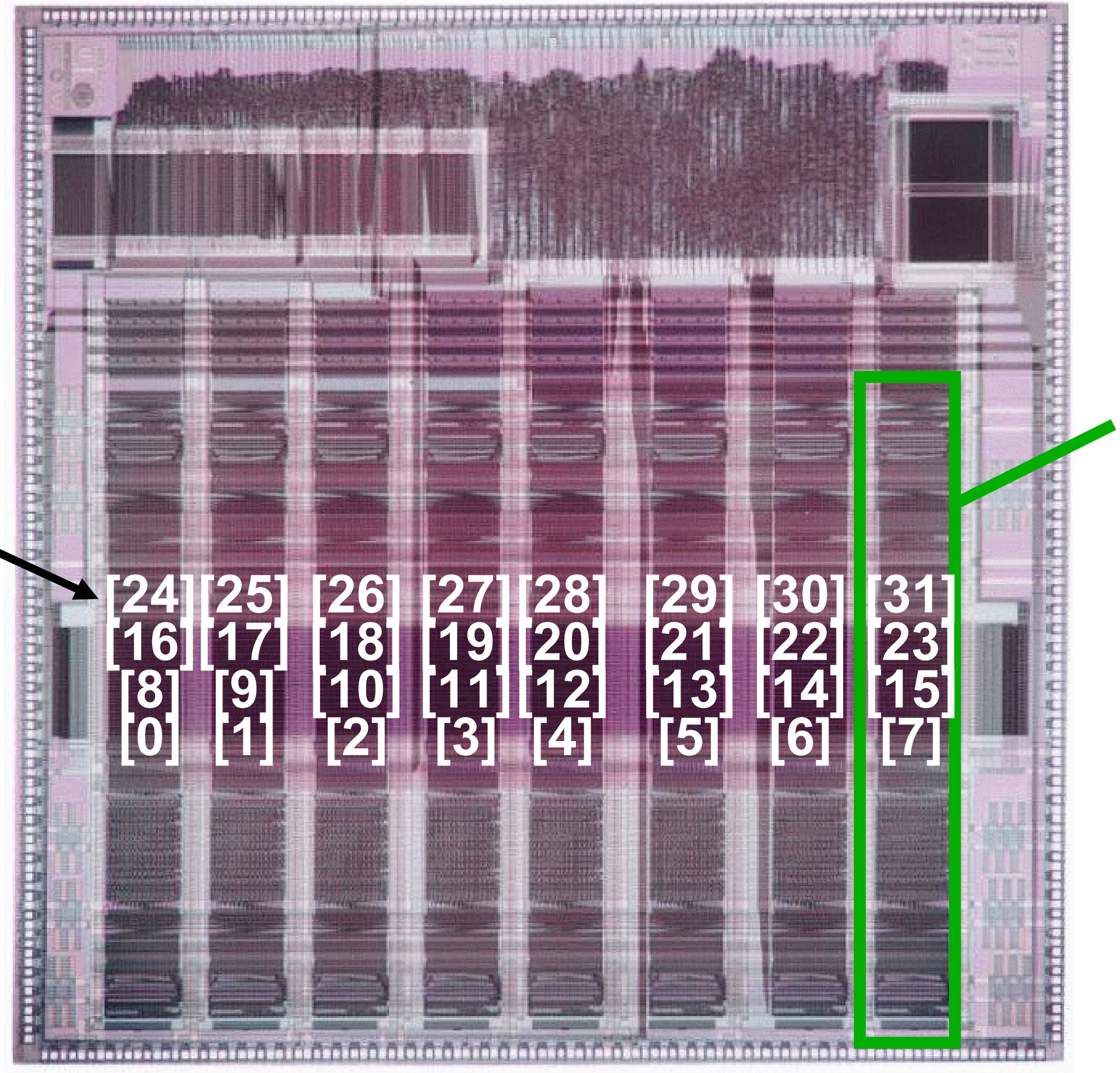
Execution using four pipelined functional units

A[6] B[6]
A[5] B[5]
A[4] B[4]
A[3] B[3]
C[2]
C[1]
C[0]
A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

Vector Unit Structure



T0 Vector Microprocessor (1995)

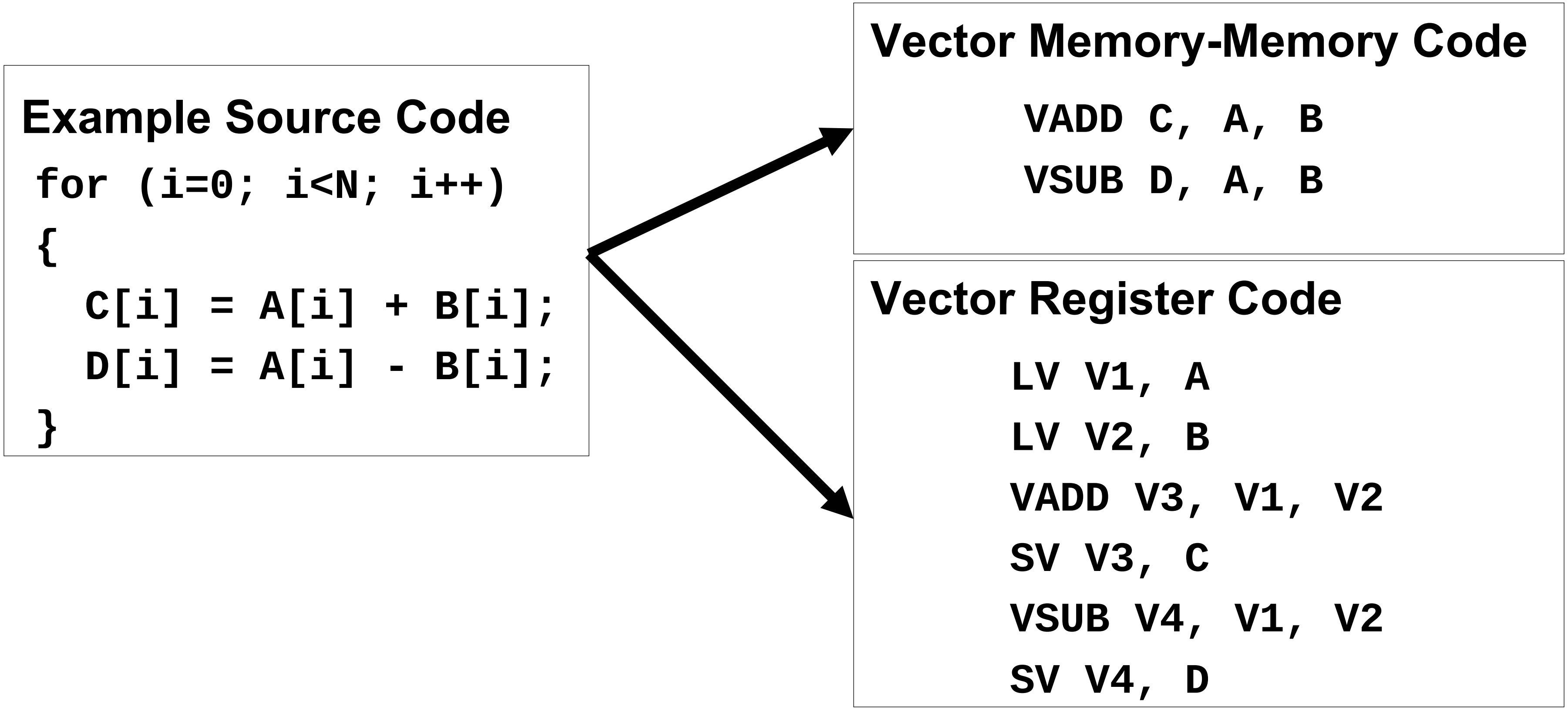


Vector register elements striped over lanes

Lane

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

```
VADD C, A, B
VSUB D, A, B
```

Vector Register Code

```
LV V1, A
LV V2, B
VADD V3, V1, V2
SV V3, C
VSUB 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

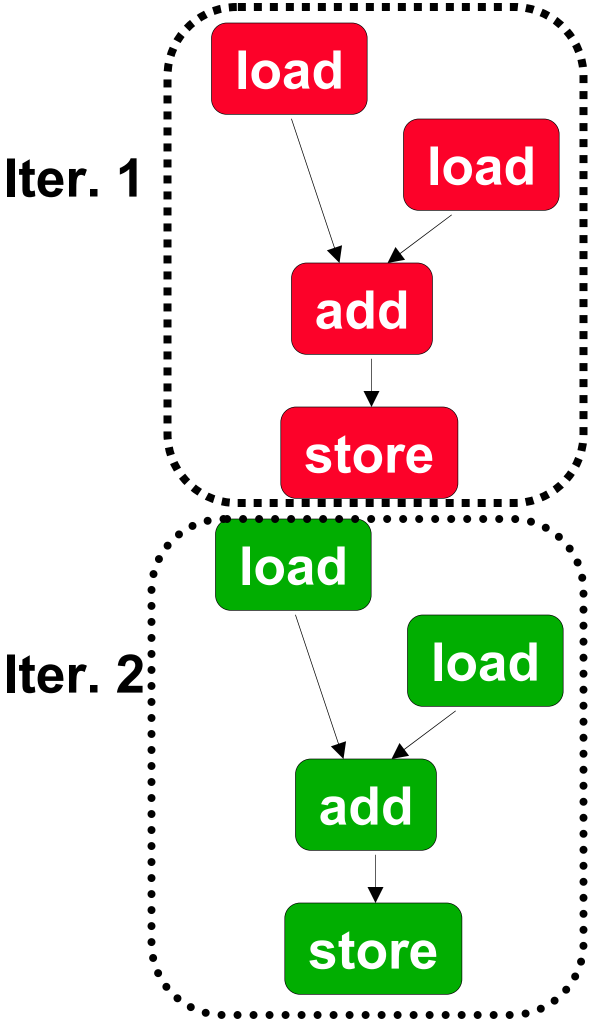
⇒***Apart from CDC follow-ons (Cyber-205, ETA-10) all major vector machines since Cray-1 have had vector register architectures***

(we ignore vector memory-memory from now on)

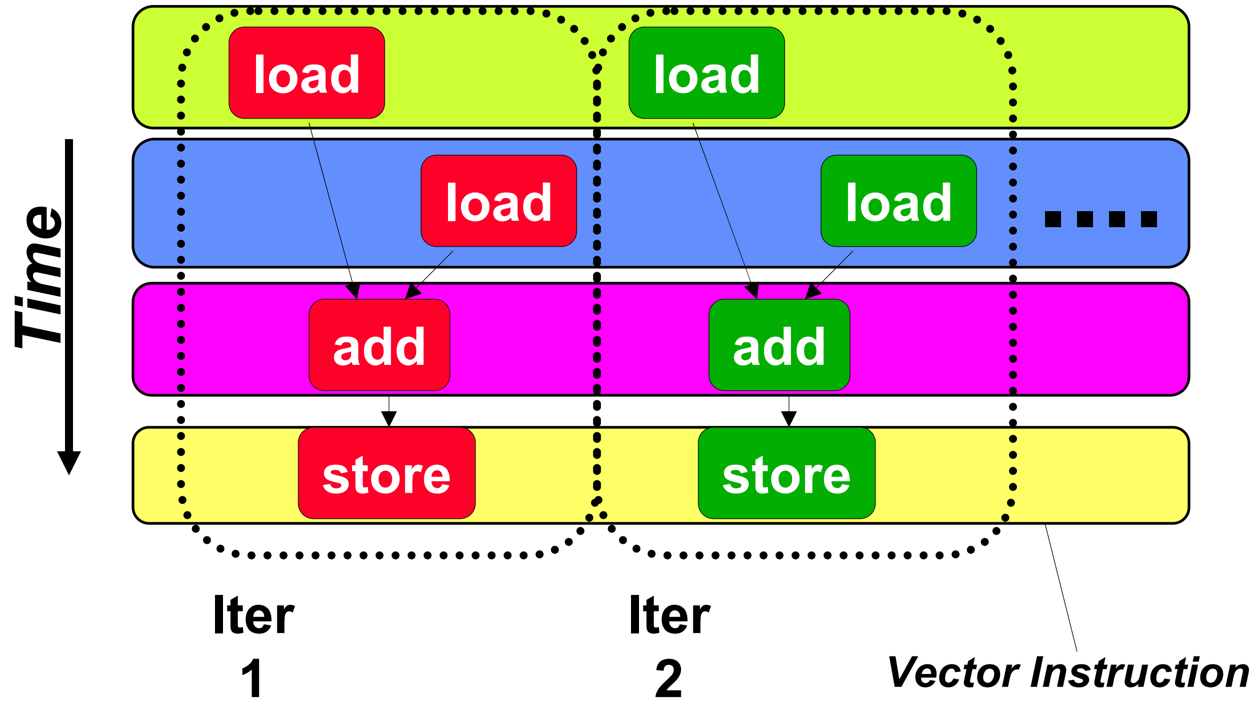
Automatic Code Vectorization

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

Scalar Sequential Code



Vectorized Code



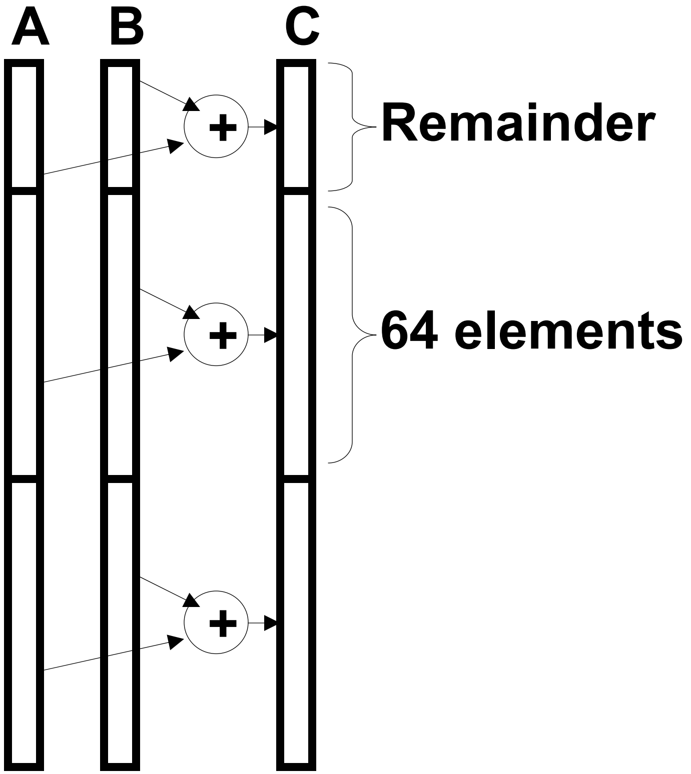
Vectorization is a massive compile-time reordering of operation sequencing ⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

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

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

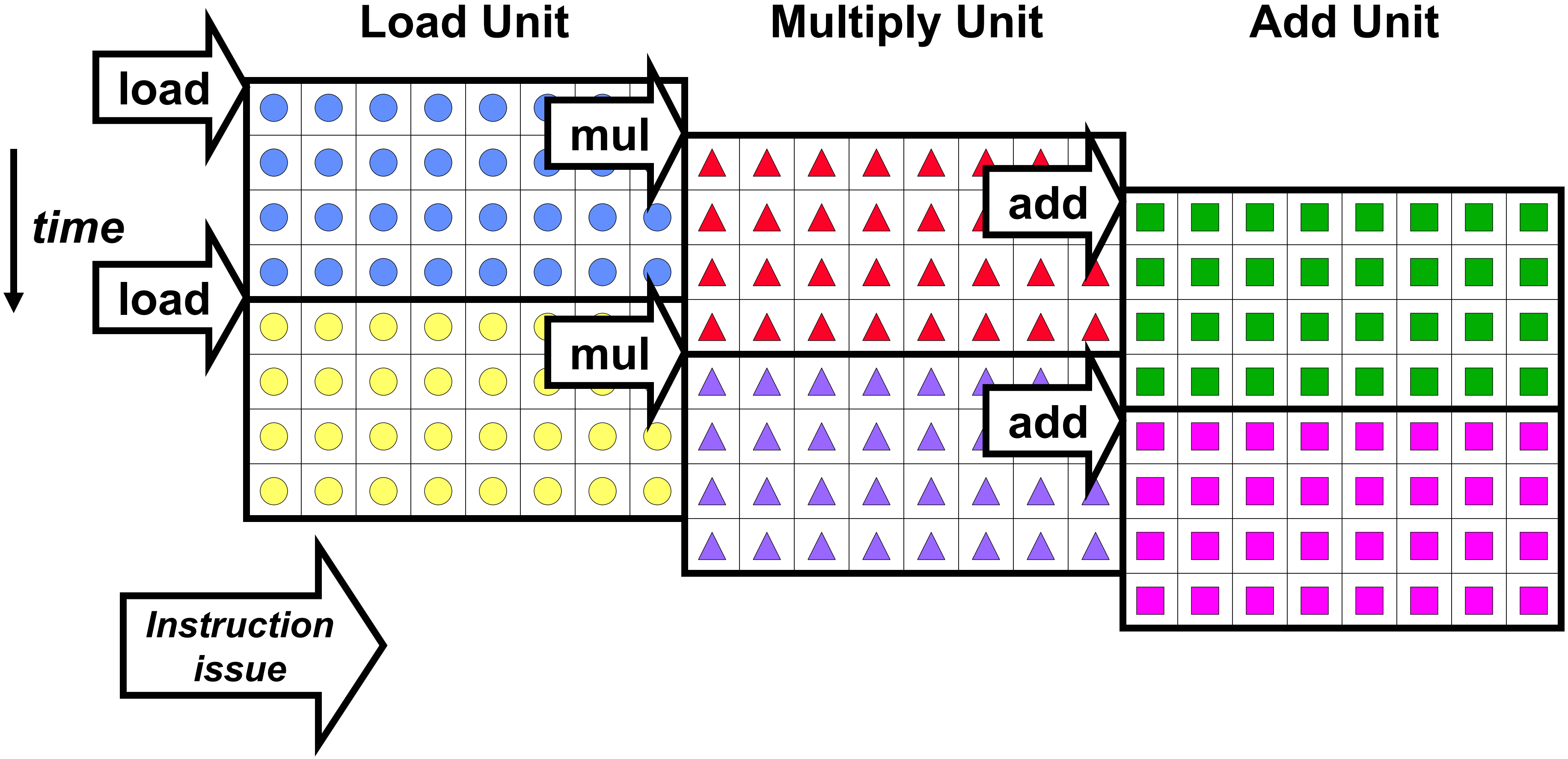


```
 and r1, N, 63  # N mod 64
 move vlr, r1   # Do remainder
loop:
 lv v1, rA
 add rA, rA, r1 # Bump pointer
 lv v2, rB
 add rB, rB, r1
 vadd v3, v1, v2
 sv v3, rC
 add rC, rC, r1
 sub N, N, r1   # Subtract elements
 move vlr, 64   # Reset full length
 move r1, 64
 bgtz N, loop   # Any more to do?
```

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

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

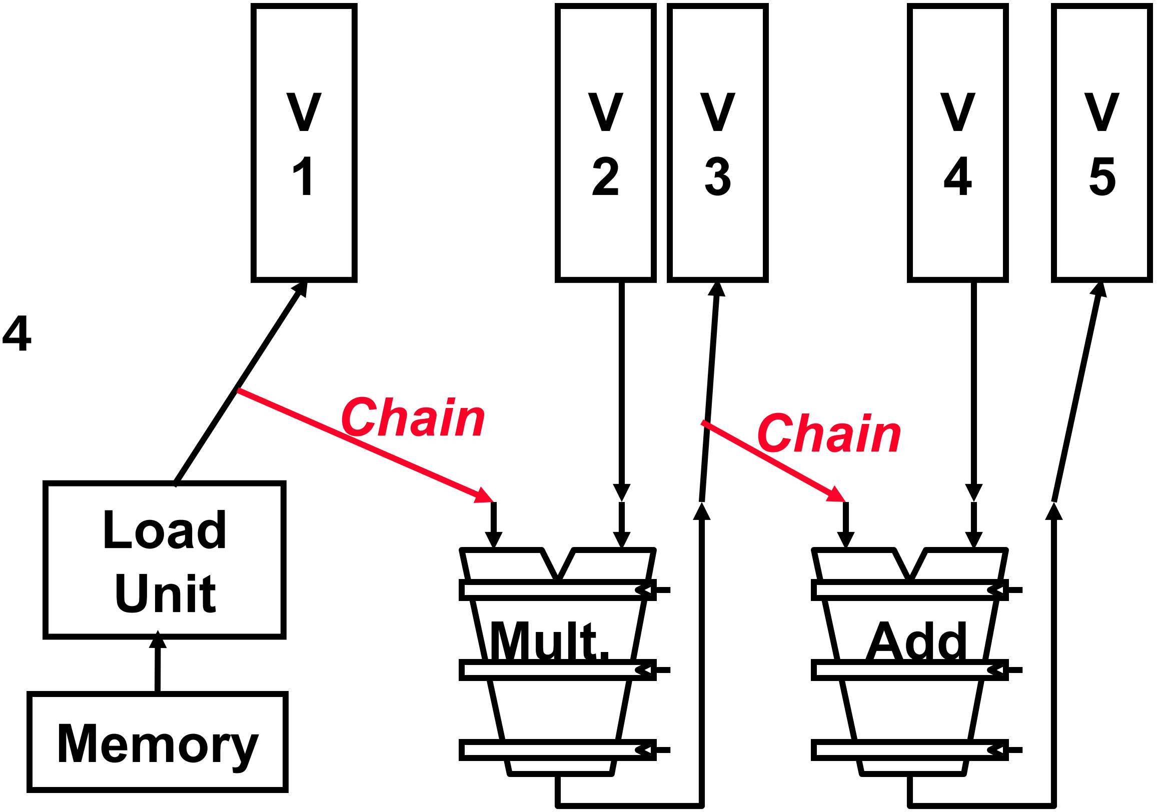


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

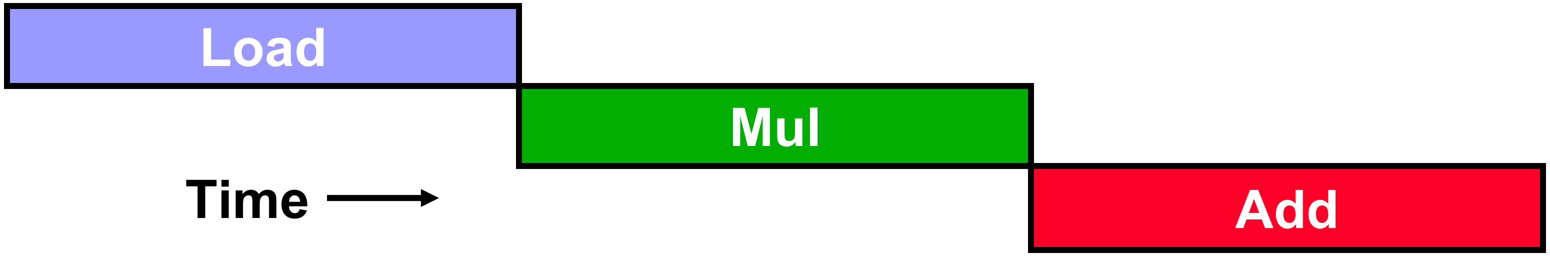
- Vector version of register bypassing
  - introduced with Cray-1

```
lv   v1
vmul v3,v1,v2
vadd v5, v3, v4
```

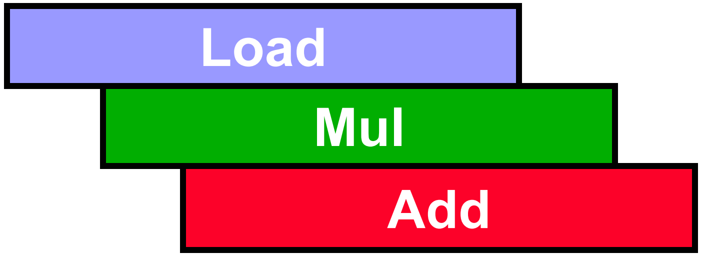


Vector Chaining Advantage

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



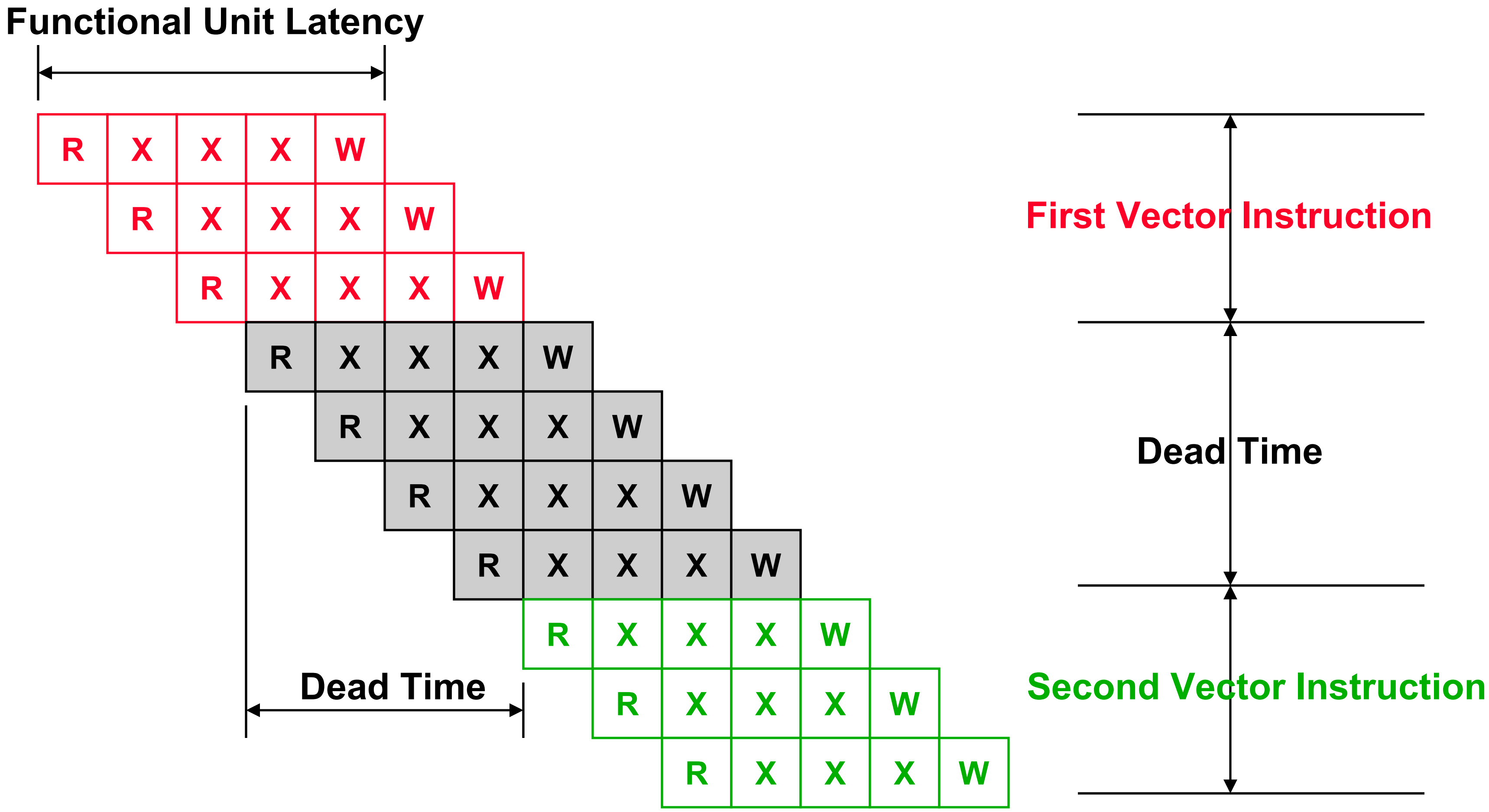
- With chaining, can start dependent instruction as soon as first result appears



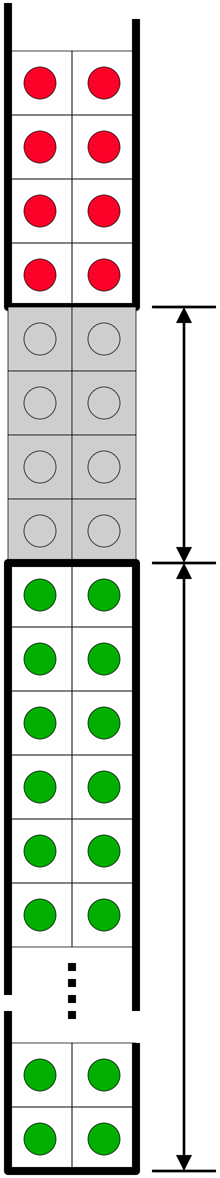
Vector Startup

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)



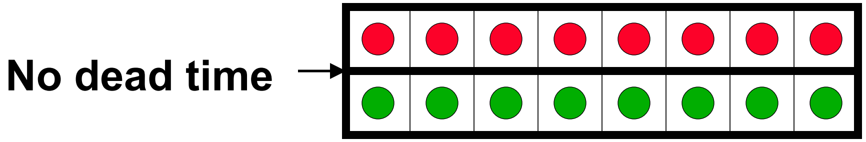
No Dead Time => Shorter Vectors



Cray C90, Two lanes

4 cycle dead time

Maximum efficiency 94% with 128 element vectors



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
lvx vC, rC, vD    # Load indirect from rC base
lv vB, rB         # Load B vector
vadd 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
lvx vA, rA, vB    # Gather initial A values
vadd vA, vA, 1    # Increment
svx 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];
    else
        A[i] = C[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

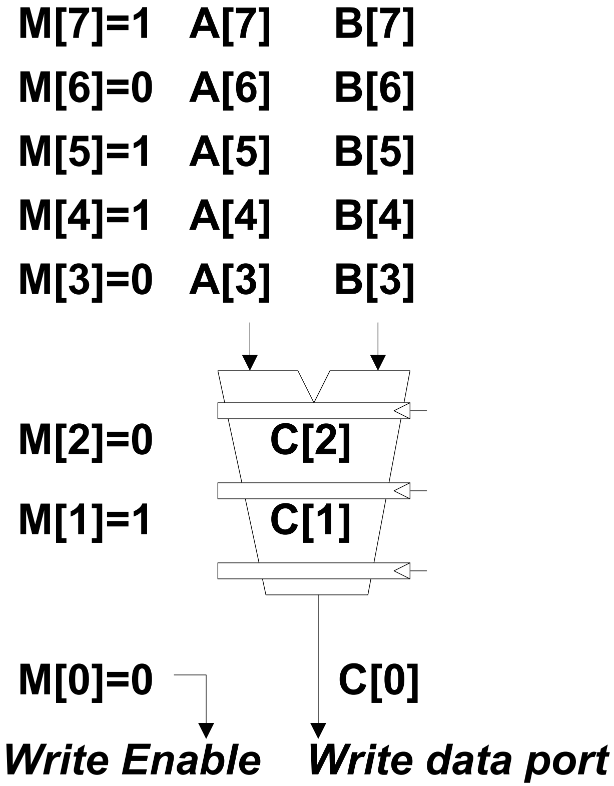
Code example:

```
lv vA, rA        # Load A vector
mgtz m0, vA      # Set bits in mask register m0 where A>0
lv.m vA, rB, m0  # Load B vector into A under mask
fnot m1, m0      # Invert mask register
lv.m vA, rC, m1  # Load C vector into A under mask
sv vA, rA        # Store A back to memory (no mask)
```

Masked Vector Instructions

Simple Implementation

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

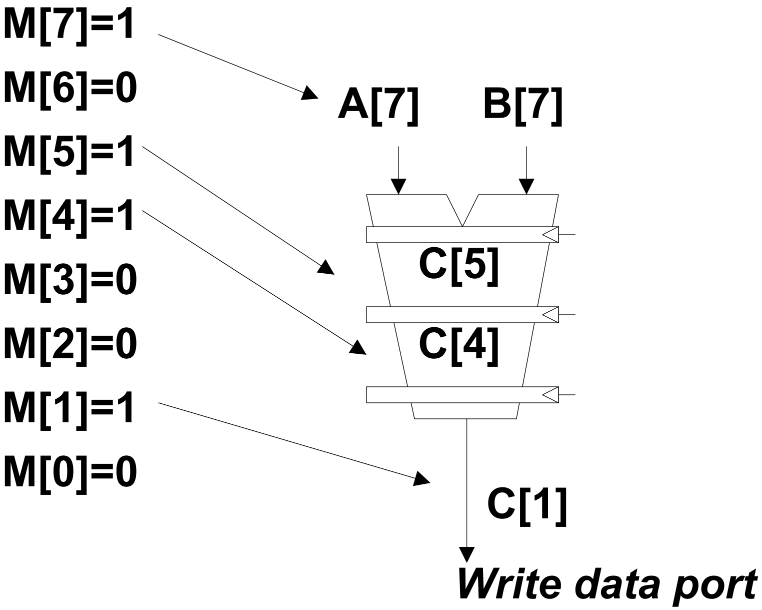


M[7]=1 A[7] B[7]
M[6]=0 A[6] B[6]
M[5]=1 A[5] B[5]
M[4]=1 A[4] B[4]
M[3]=0 A[3] B[3]
M[2]=0 C[2]
M[1]=1 C[1]
M[0]=0 C[0]

Write Enable *Write data port*

Density-Time Implementation

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



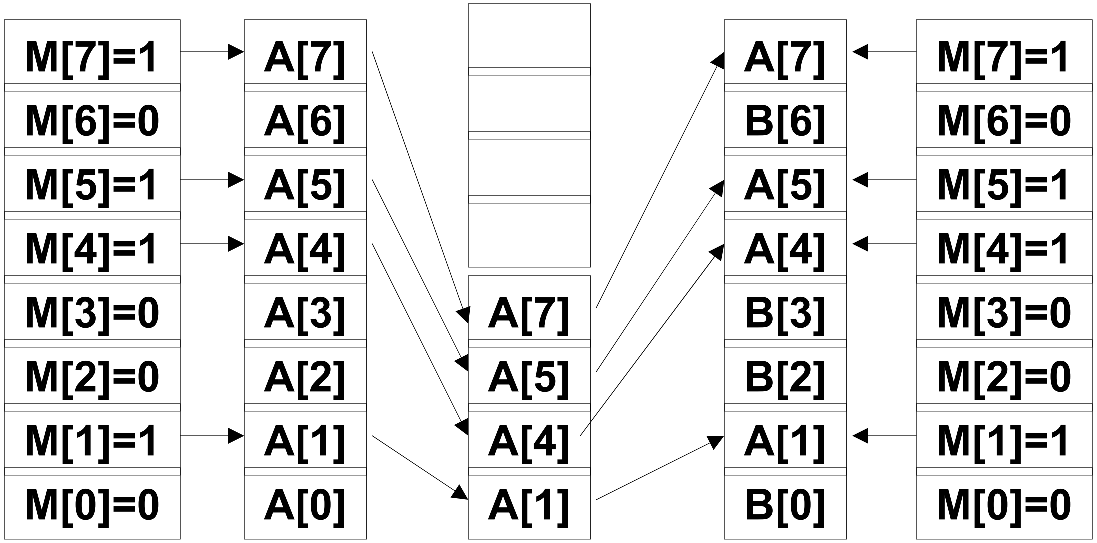
M[7]=1
M[6]=0
M[5]=1
M[4]=1
M[3]=0
M[2]=0
M[1]=1
M[0]=0

A[7] B[7]
C[5]
C[4]
C[1]

Write data port

Compress/Expand Operations

- Compress packs non-masked elements into contiguous vector
  - population count of mask vector gives packed vector length
- Expand performs inverse operation



Compress *Expand*

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-5 (1998)

- **CMOS Technology**
  - 250MHz clock (312 MHz in 2001)
  - CPU fits on one multi-chip module
  - SDRAM main memory (up to 128GB)
- **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 (256 elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 16 lanes, 8 GFLOPS peak (32 FLOPS/cycle)
  - 1 load & store unit (32x8 byte accesses/cycle)
  - 64 GB/s memory bandwidth per processor
- **SMP structure**
  - 16 CPUs connected to memory through crossbar
  - 1 TB/s shared memory bandwidth

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