VECTORS COMPUTERS

Noor Mahammad Sk

Super Computer

- 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
  - CDC6600 (Cray, 1964) regarded as first supercomputer

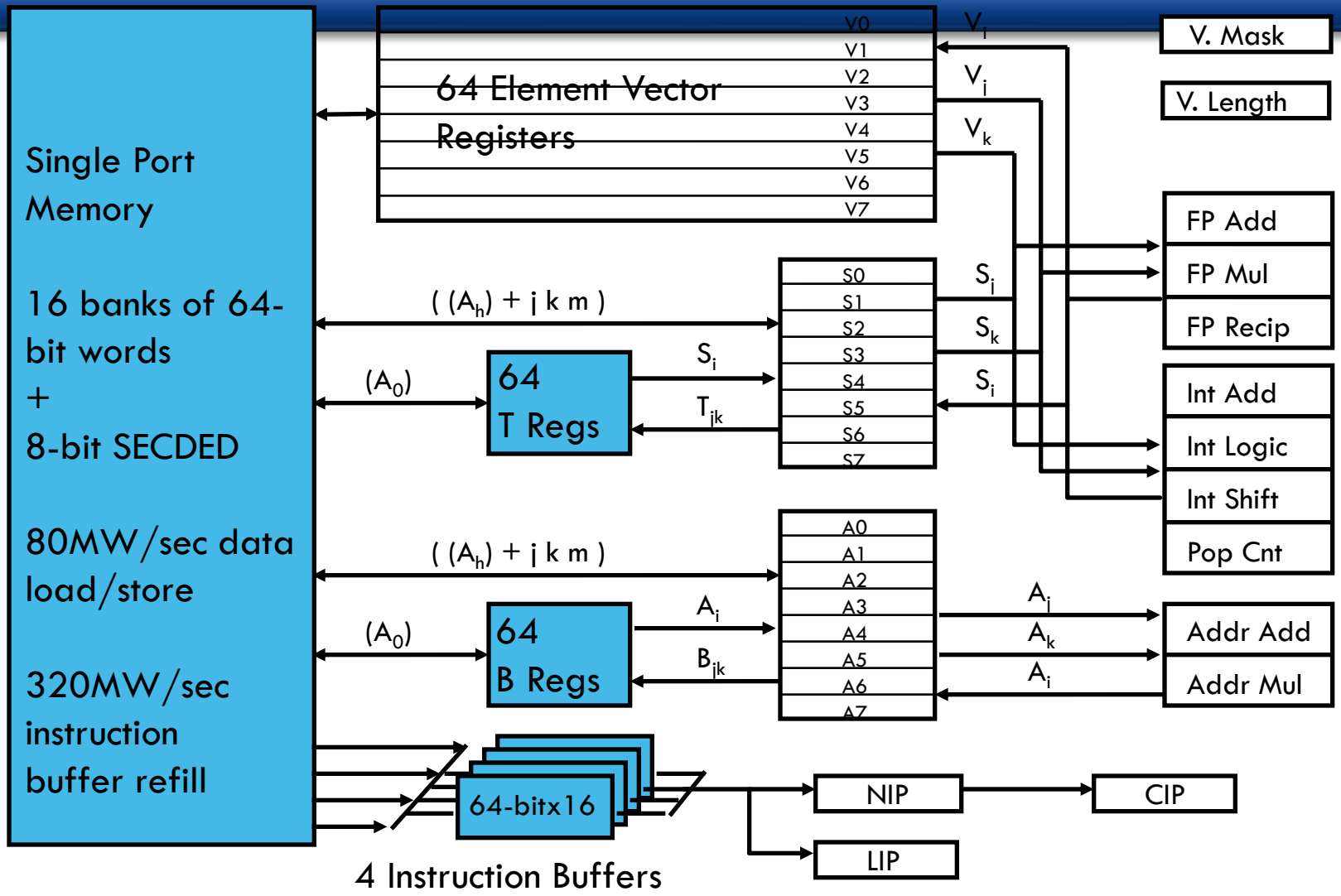
Supercomputer Applications

- 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

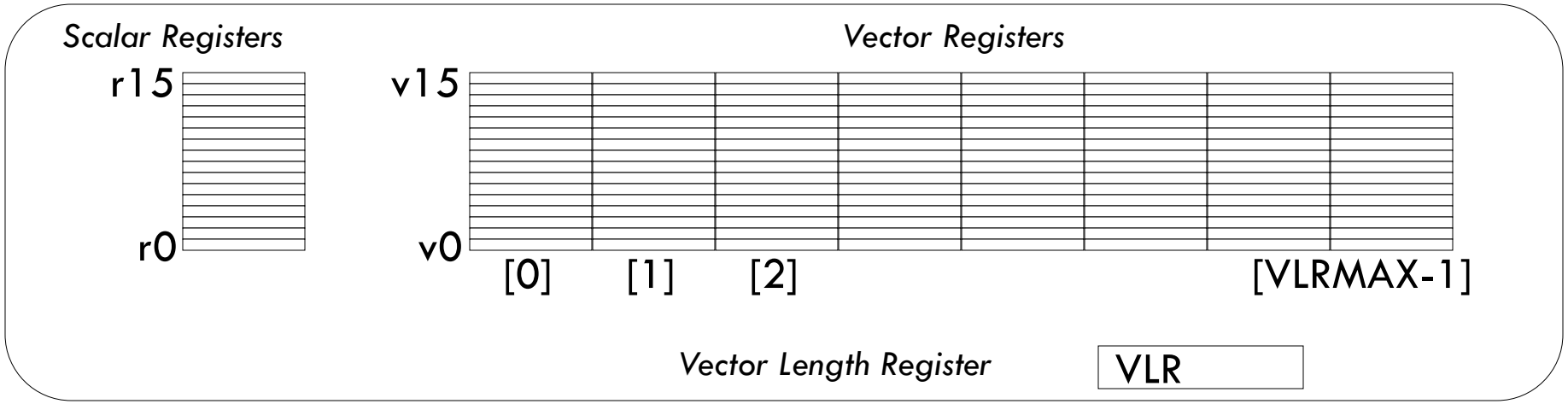
- Scalar Unit + Vector Extensions
  - Load/Store Architecture
  - Vector Registers
  - Vector Instructions
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

Cray-1 (1976)



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

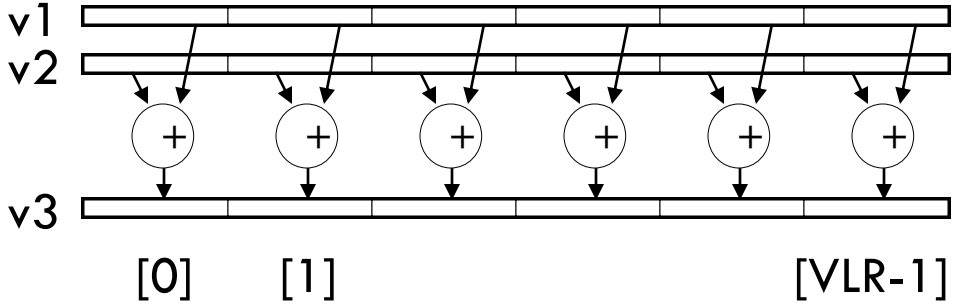
Vector Programming Model



Vector Programming Model

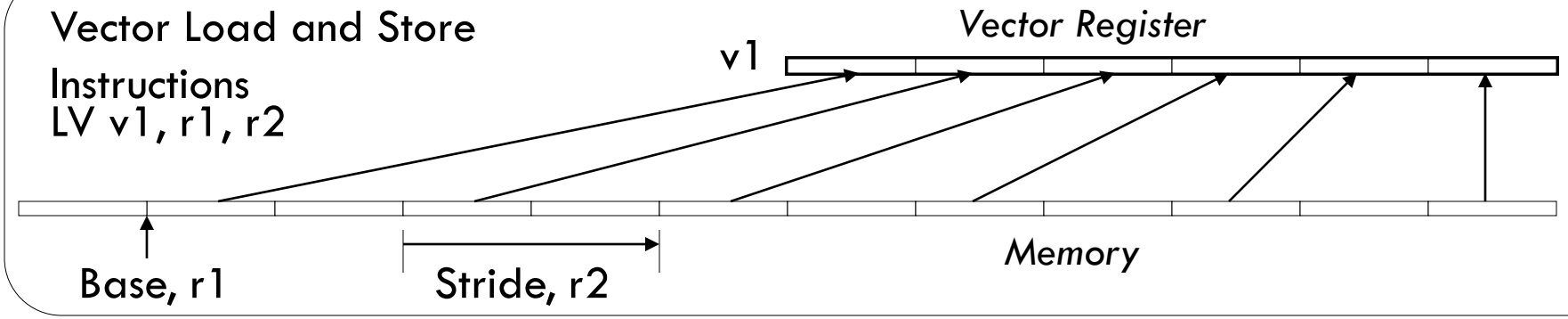
Vector Arithmetic Instructions

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
```

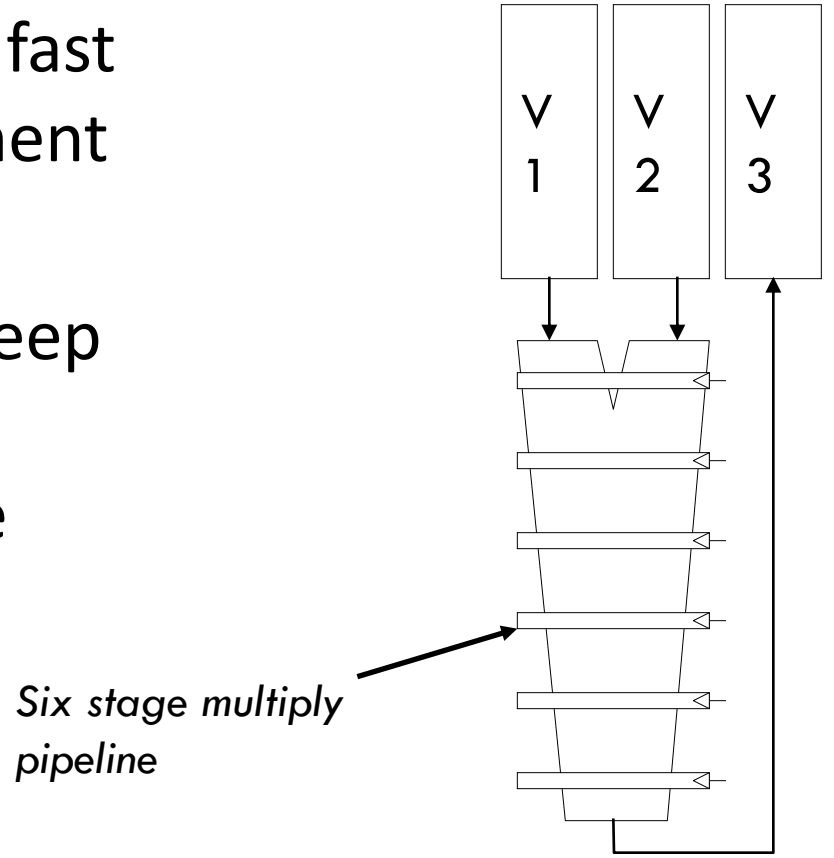
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

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

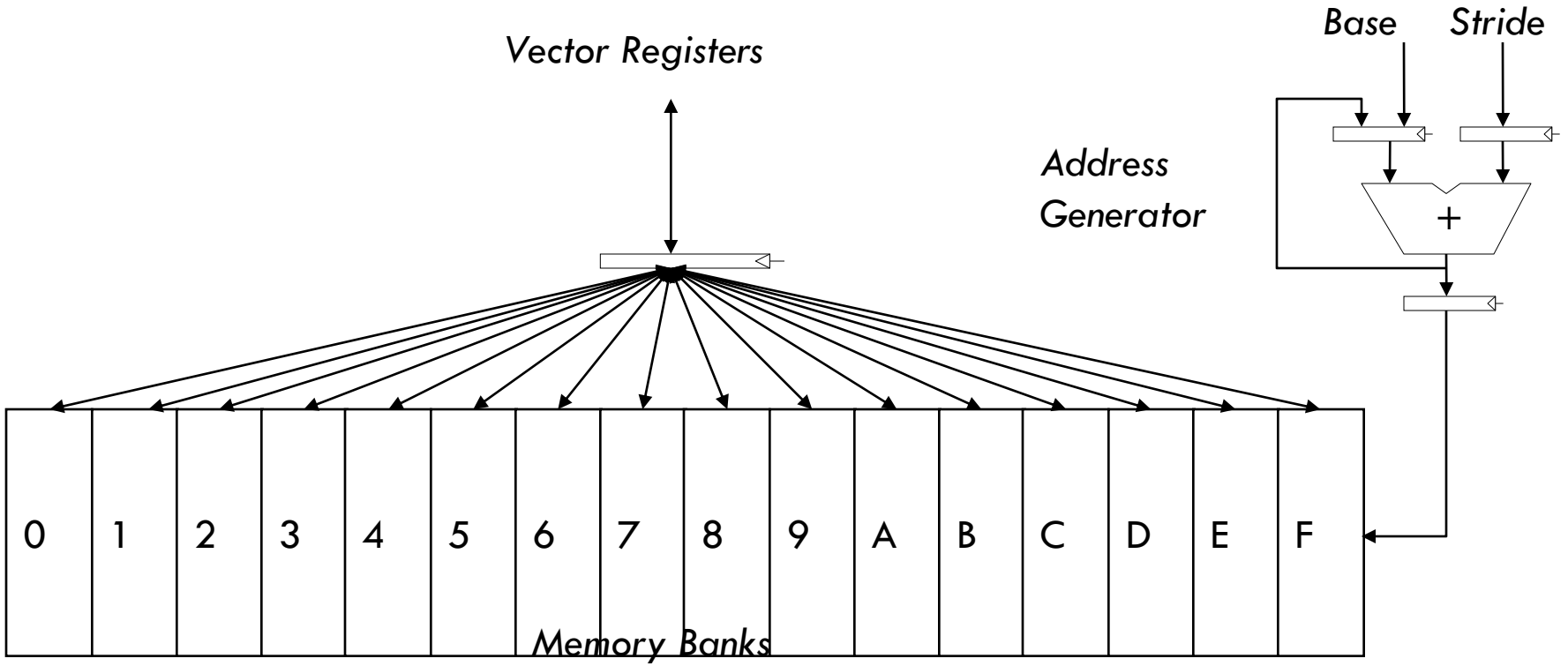
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



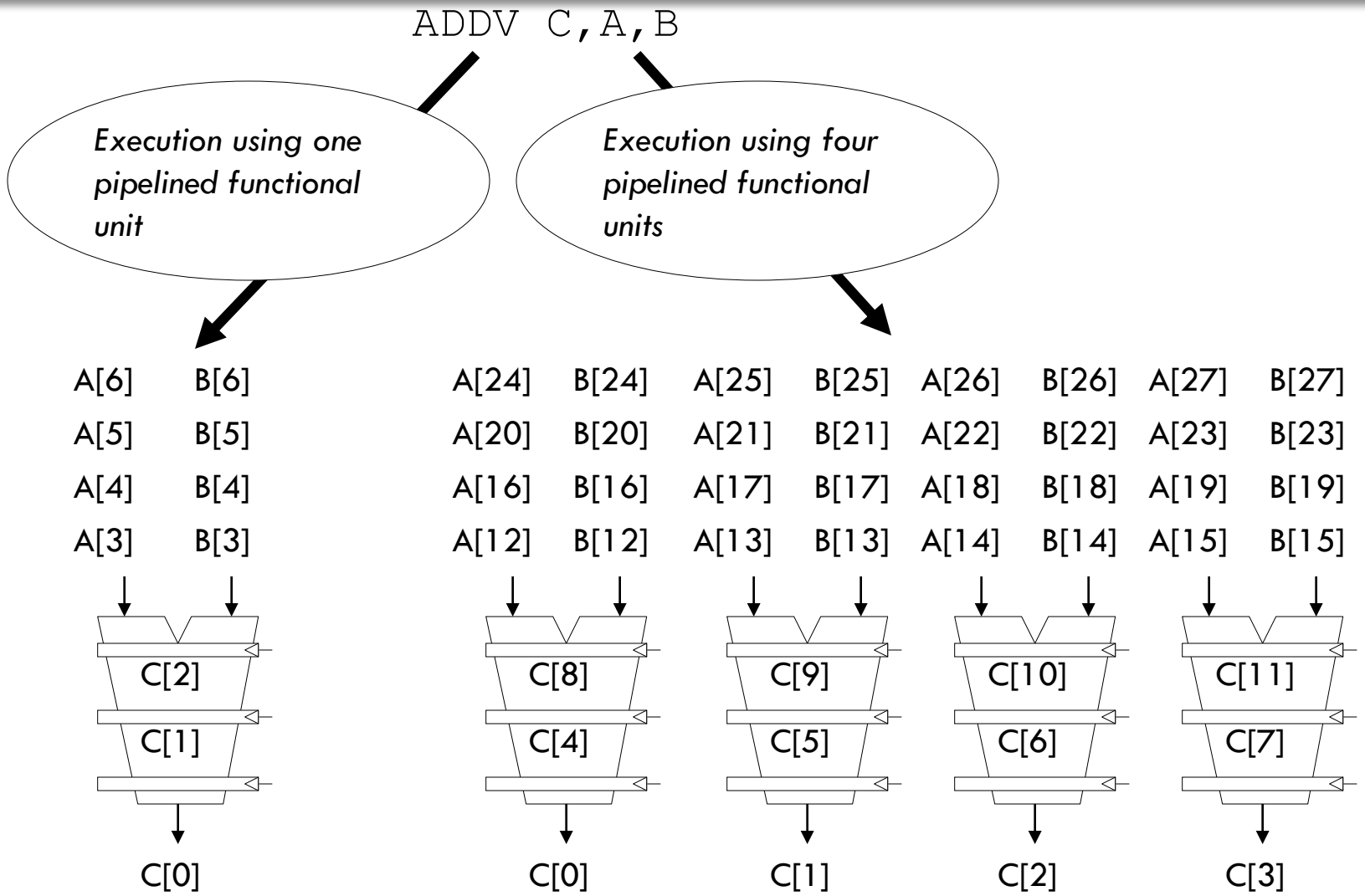

```
V3 <- v1 * v2
```

Vector Memory System

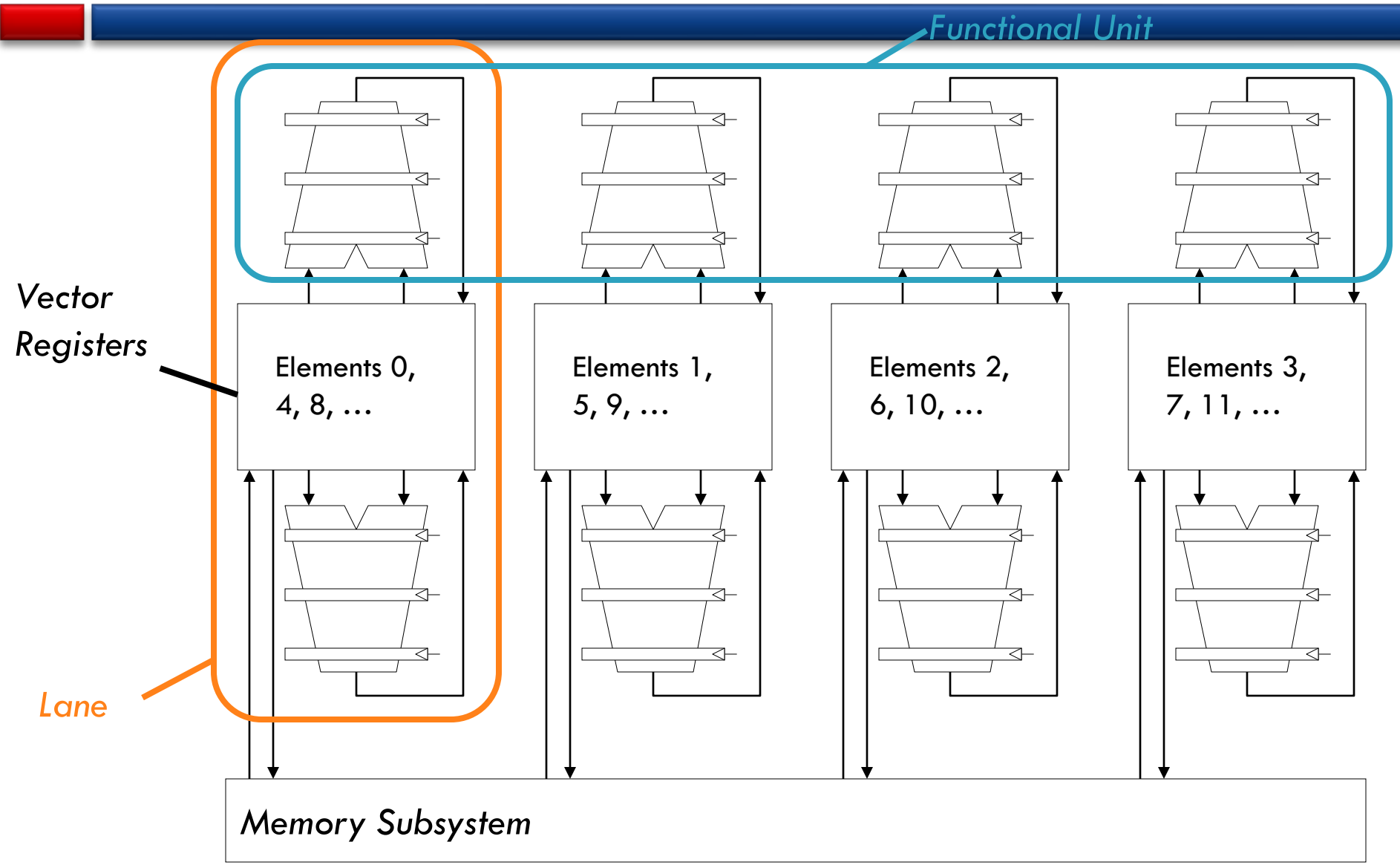
- Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency
  - *Bank busy time*: Cycles between accesses to same bank



Vector Instruction Execution



Vector Unit Structure

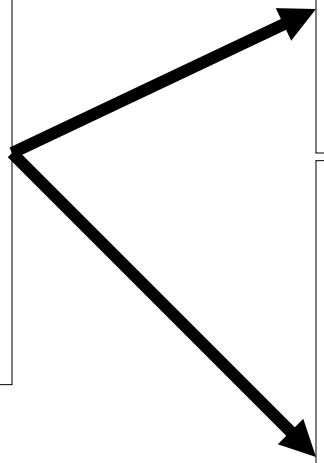


Vector Memory-Memory versus Vector Register Machines

- 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

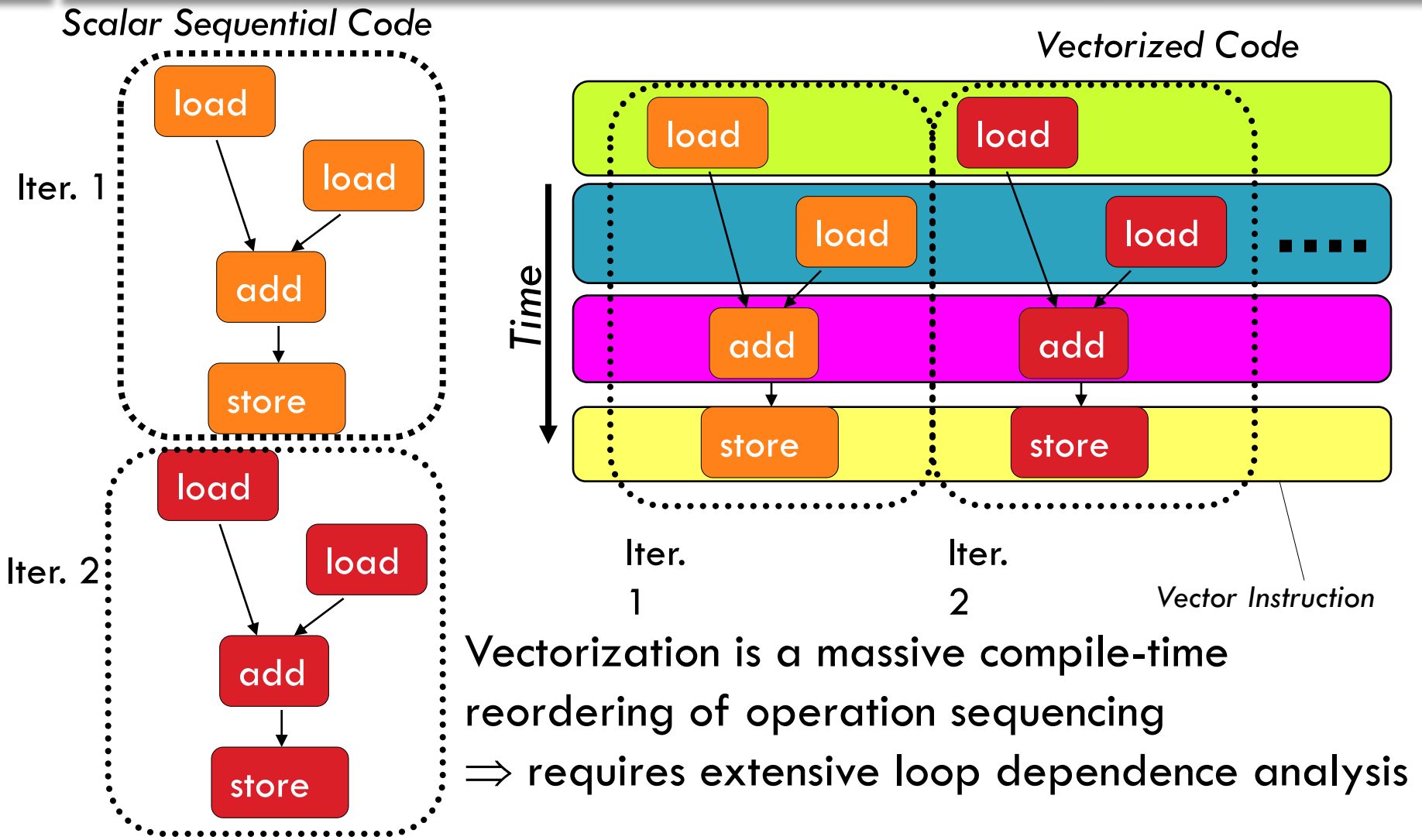
```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

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 elements

Automatic Code Vectorization

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



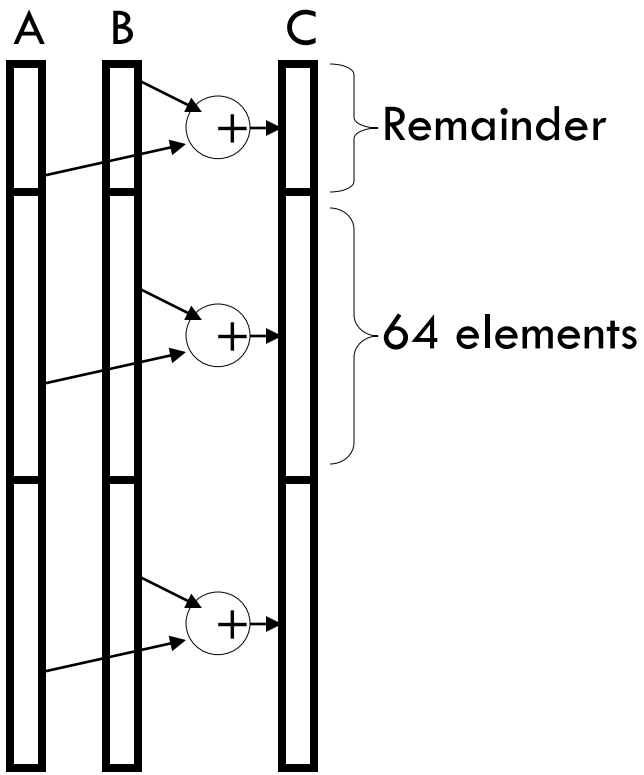
Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Vector Stripmining

- Problem: Vector registers have finite length
- Solution: Break loops into pieces that fit into vector registers, *"Stripmining"*

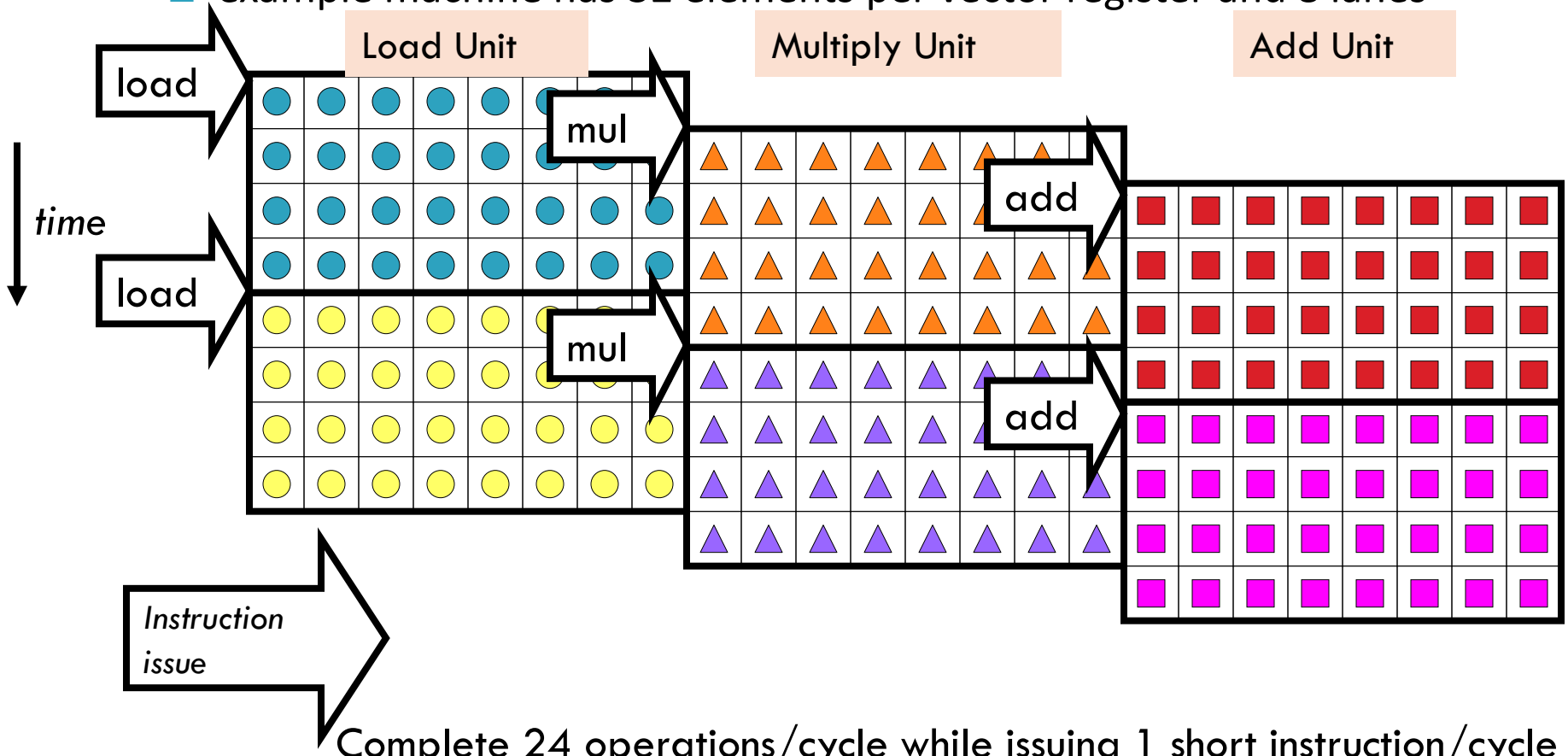
```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```




```
        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

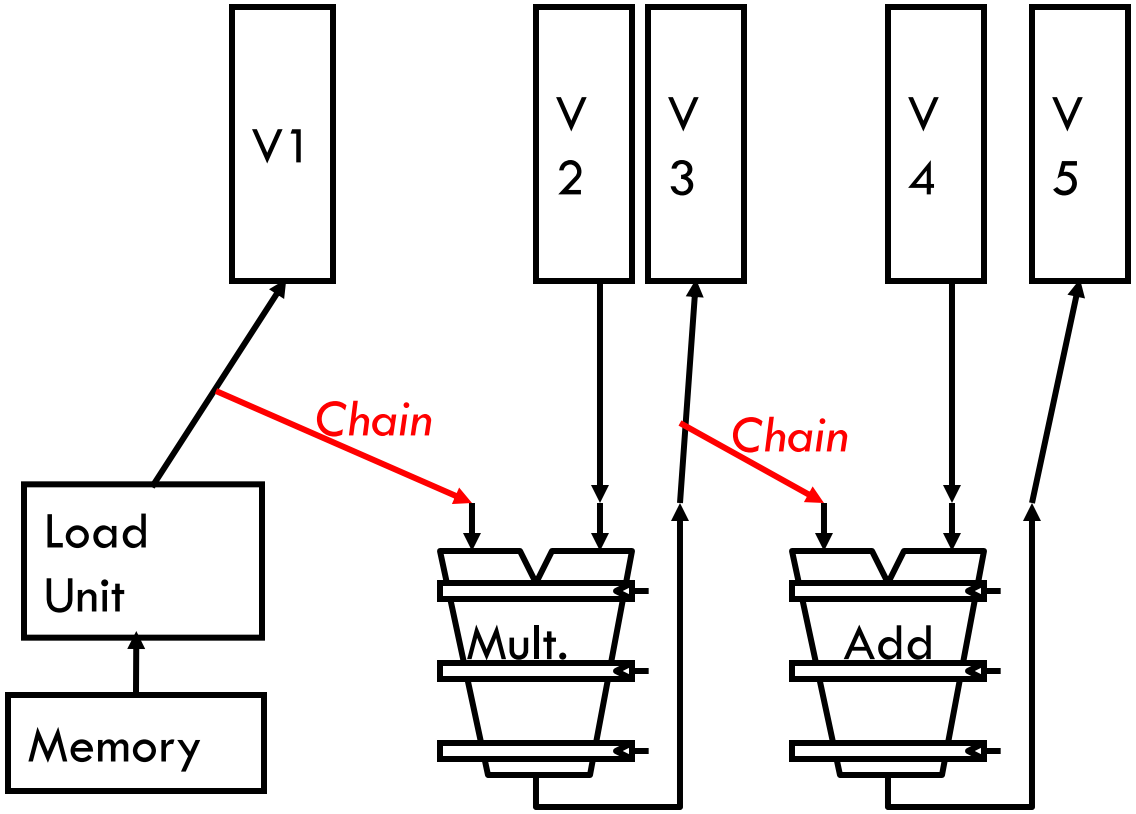


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

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

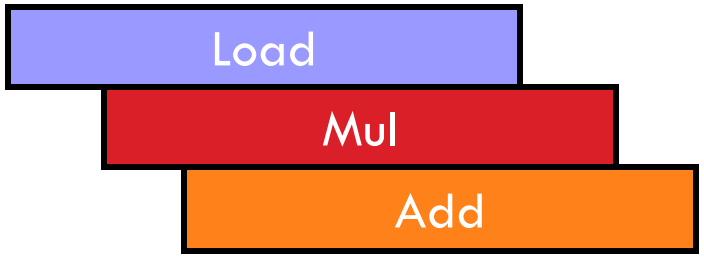
```
LV    v1
MULV  v3,v1,v2
ADDV  v5, v3, v4
```



Vector Chaining Advantage

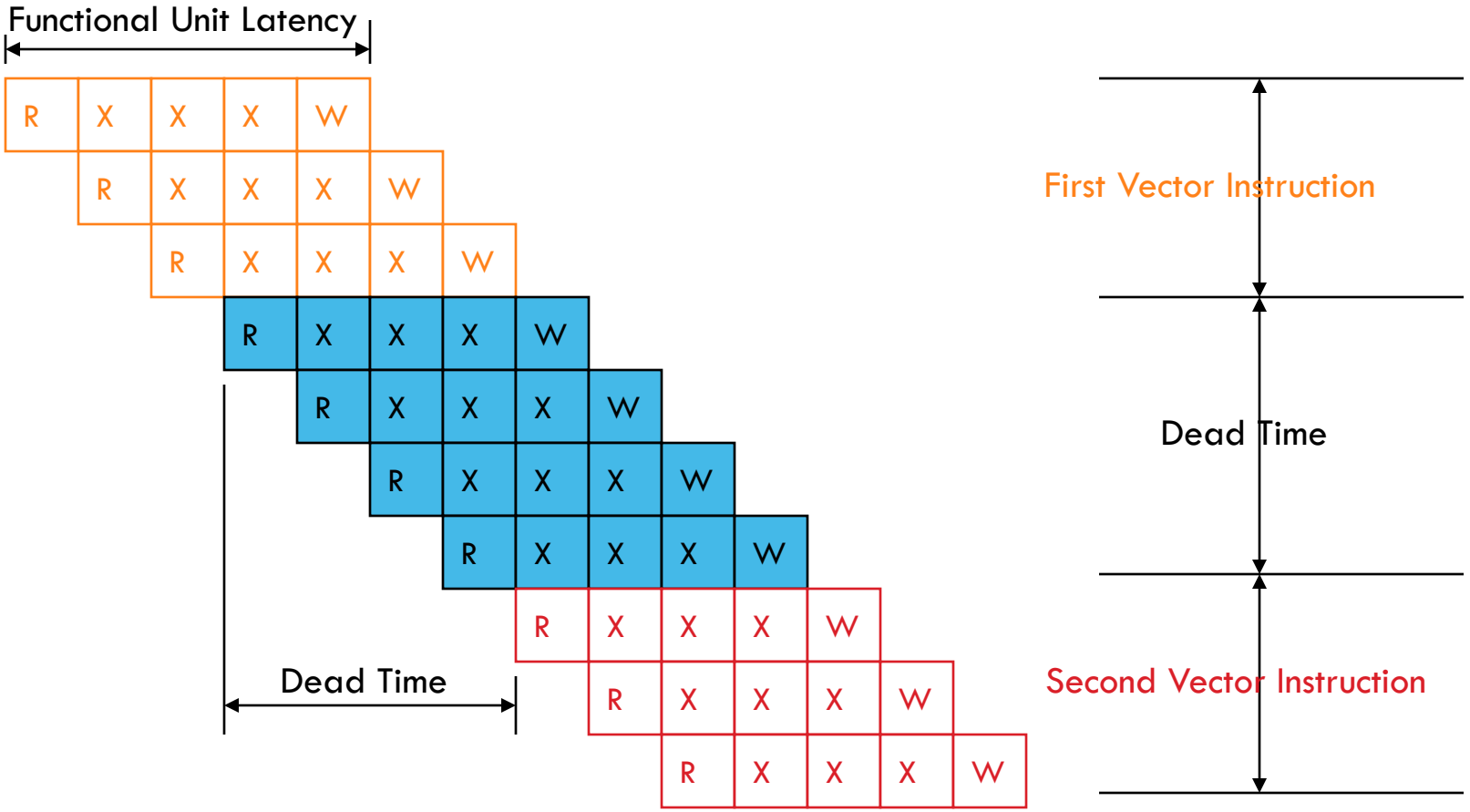
- Without chaining, must wait for last element of result to be written before starting dependent instruction

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

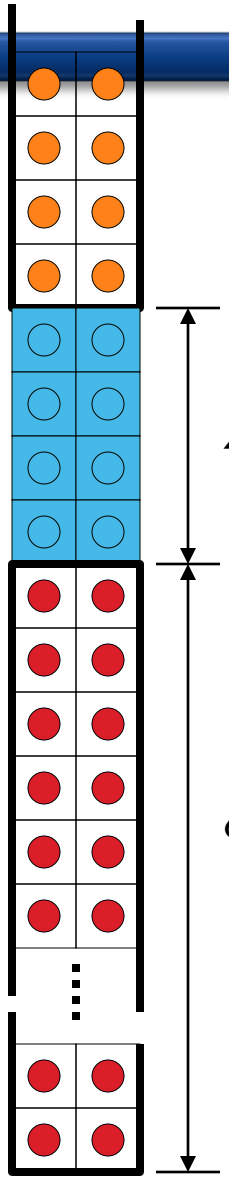


Vector Startup

- 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

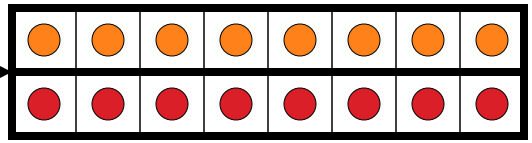
64 cycles active

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];
```

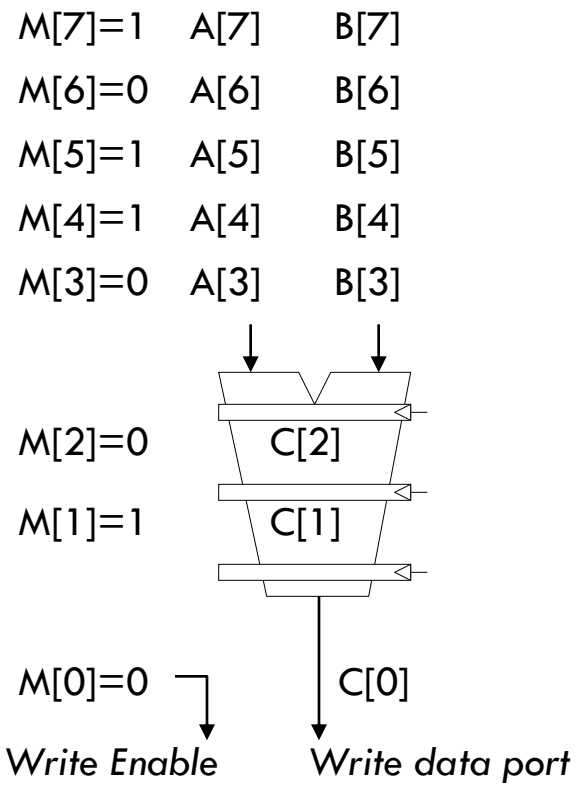
- Solution: Add vector *mask* (or *flag*) registers
  - vector version of predicate registers, 1 bit per element
- …and *maskable* vector instructions
  - vector operation becomes NOP at elements where mask bit is clear
- Code example:

```
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

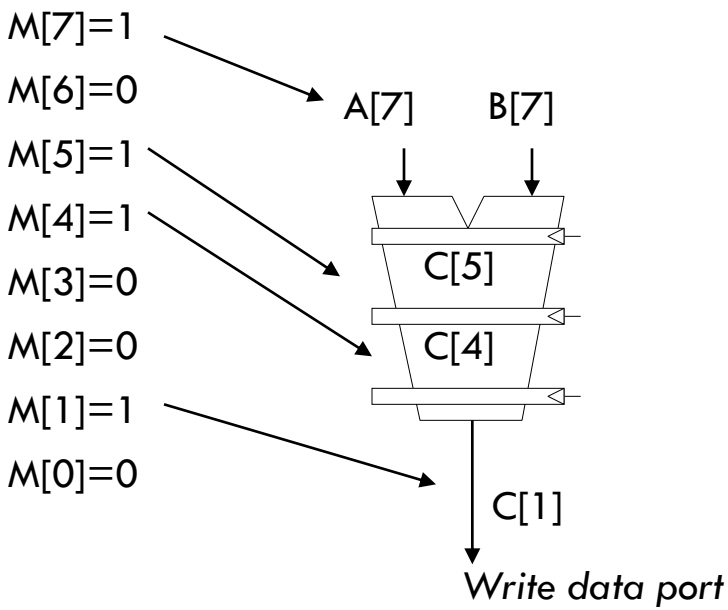
Simple Implementation

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



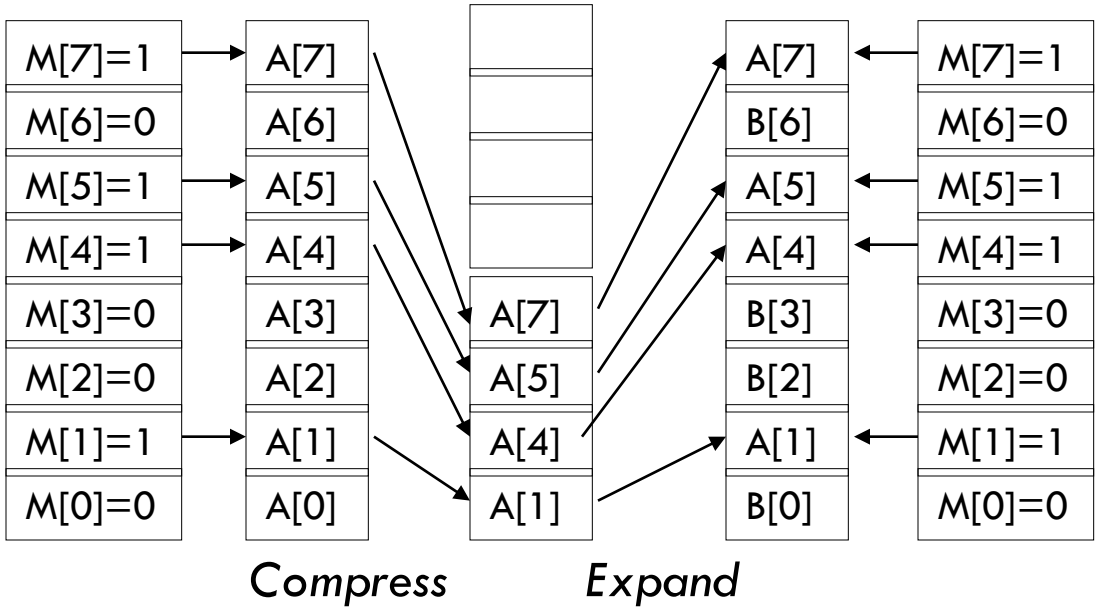
Density-Time Implementation

- 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

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

Graphical Processing Units

- Given the hardware invested to do graphics well, how can be supplement it to improve performance of a wider range of applications?

- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is “Single Instruction Multiple Thread”

Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS

NVIDIA GPU Architecture

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files

- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

Example

- Multiply two vectors of length 8192
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - 512 threads per block
  - SIMD instruction executes 32 elements at a time
  - Thus grid size = 16 blocks
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

Terminology

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

NVIDIA Instruction Set Arch.

- ISA is an abstraction of the hardware instruction set
  - "Parallel Thread Execution (PTX)"
  - Uses virtual registers
  - Translation to machine code is performed in software
  - Example:

```
shl.s32         R8, blockIdx, 9     ; Thread Block ID * Block size (512 or 29)
add.s32         R8, R8, threadIdx   ; R8 = i = my CUDA thread ID
ld.global.f64   RD0, [X+R8]         ; RD0 = X[i]
ld.global.f64   RD2, [Y+R8]         ; RD2 = Y[i]
mul.f64 R0D, RD0, RD4   ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 R0D, RD0, RD2   ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0  ; Y[i] = sum (X[i]*a + Y[i])
```

Conditional Branching

- Like vector architectures, GPU branch hardware uses internal masks
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

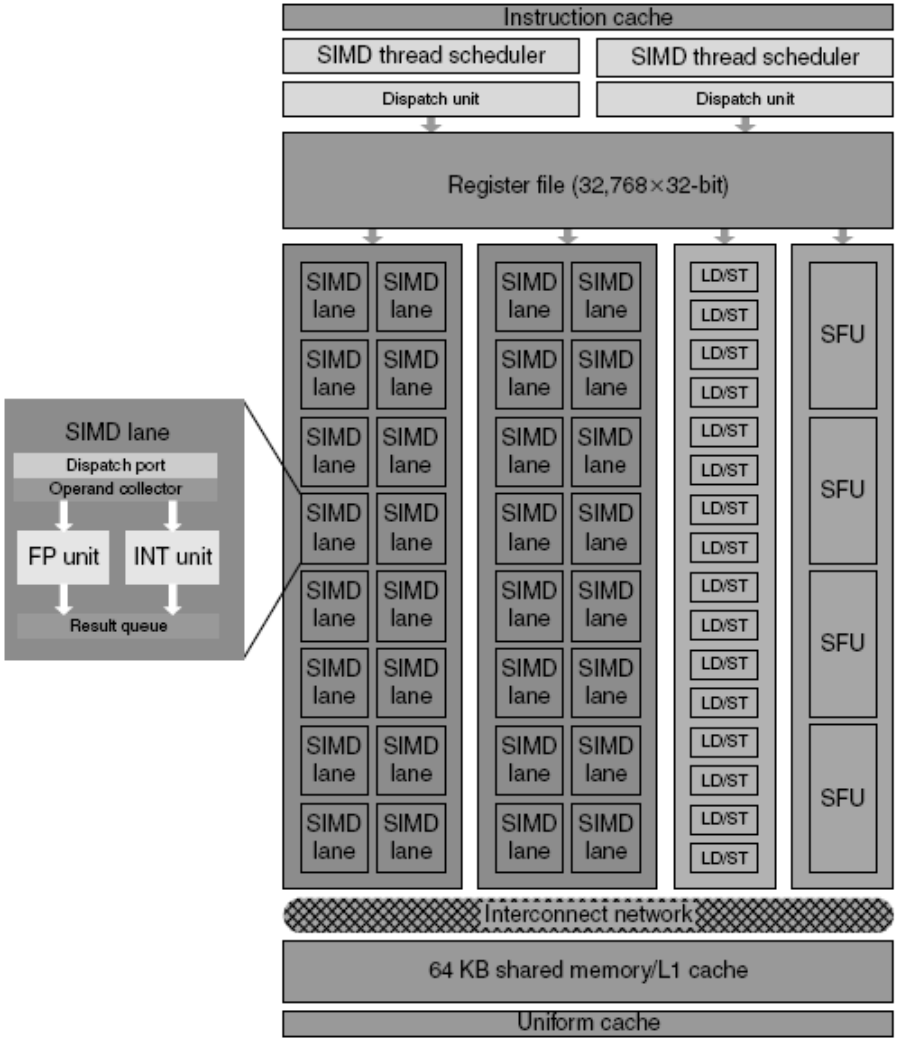
NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - "Private memory"
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write GPU memory

Fermi Architecture Innovations

- Each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles
- Fast double precision
- Caches for GPU memory
- 64-bit addressing and unified address space
- Error correcting codes
- Faster context switching
- Faster atomic instructions

Fermi Multithreaded SIMD Proc.



Fermi streaming multiprocessor (SM)