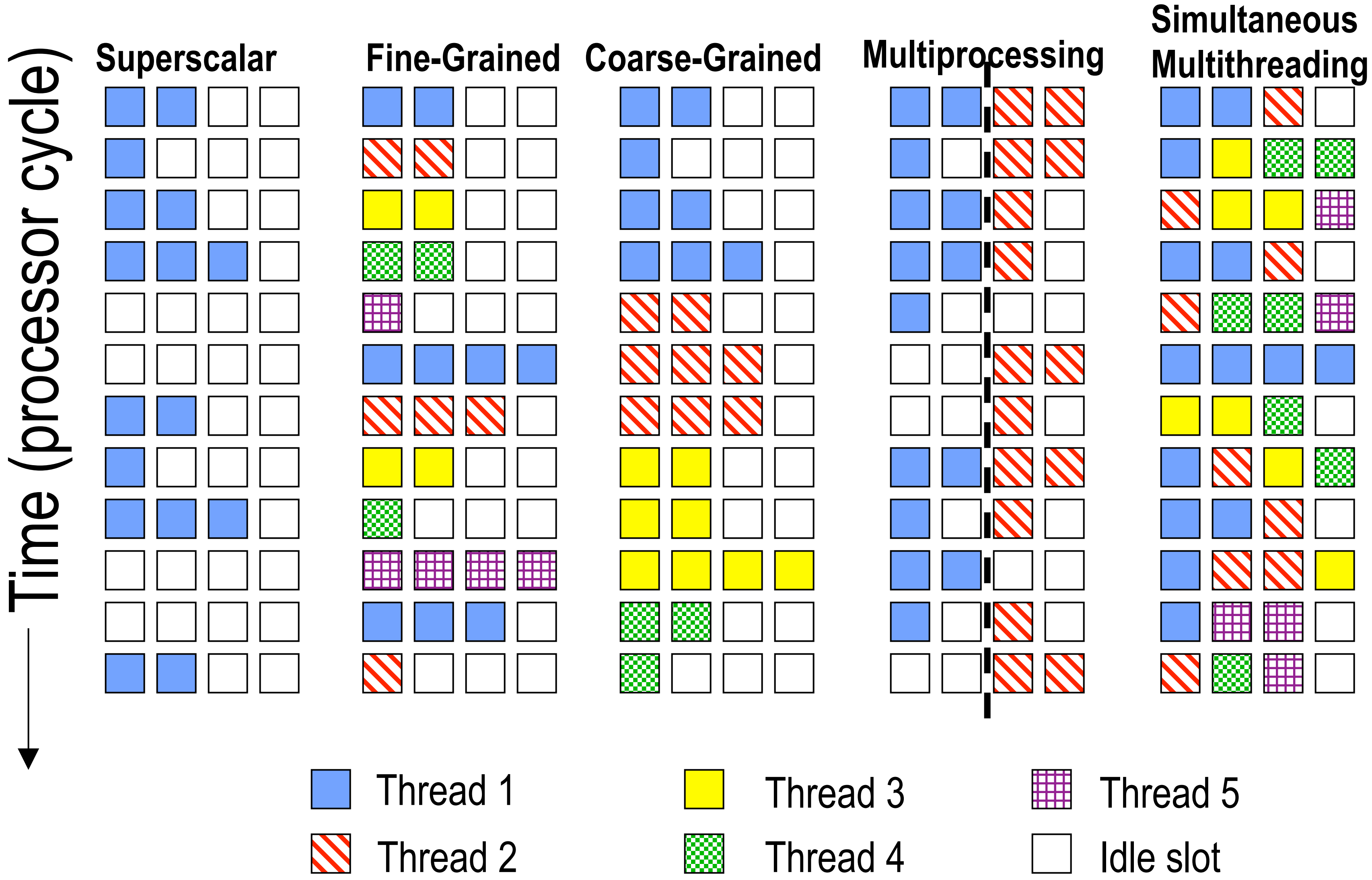
CS 152 Computer Architecture and Engineering

Lecture 15: Vector Computers

Dr. George Michelogiannakis
EECS, University of California at Berkeley
CRD, Lawrence Berkeley National Laboratory

http://inst.eecs.berkeley.edu/~cs152

Last Time Lecture 14: Multithreading



Question of the Day

- Can Vector and VLIW combine?

Supercomputers

- Definition of a supercomputer:
- Fastest machine in world at given task
  - Performs at or near the currently highest operational rate for computers
- 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

CDC 6600 *Seymour Cray, 1963*

- A fast pipelined machine with 60-bit words
  - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
  - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - over 100 sold ($7-10M each)

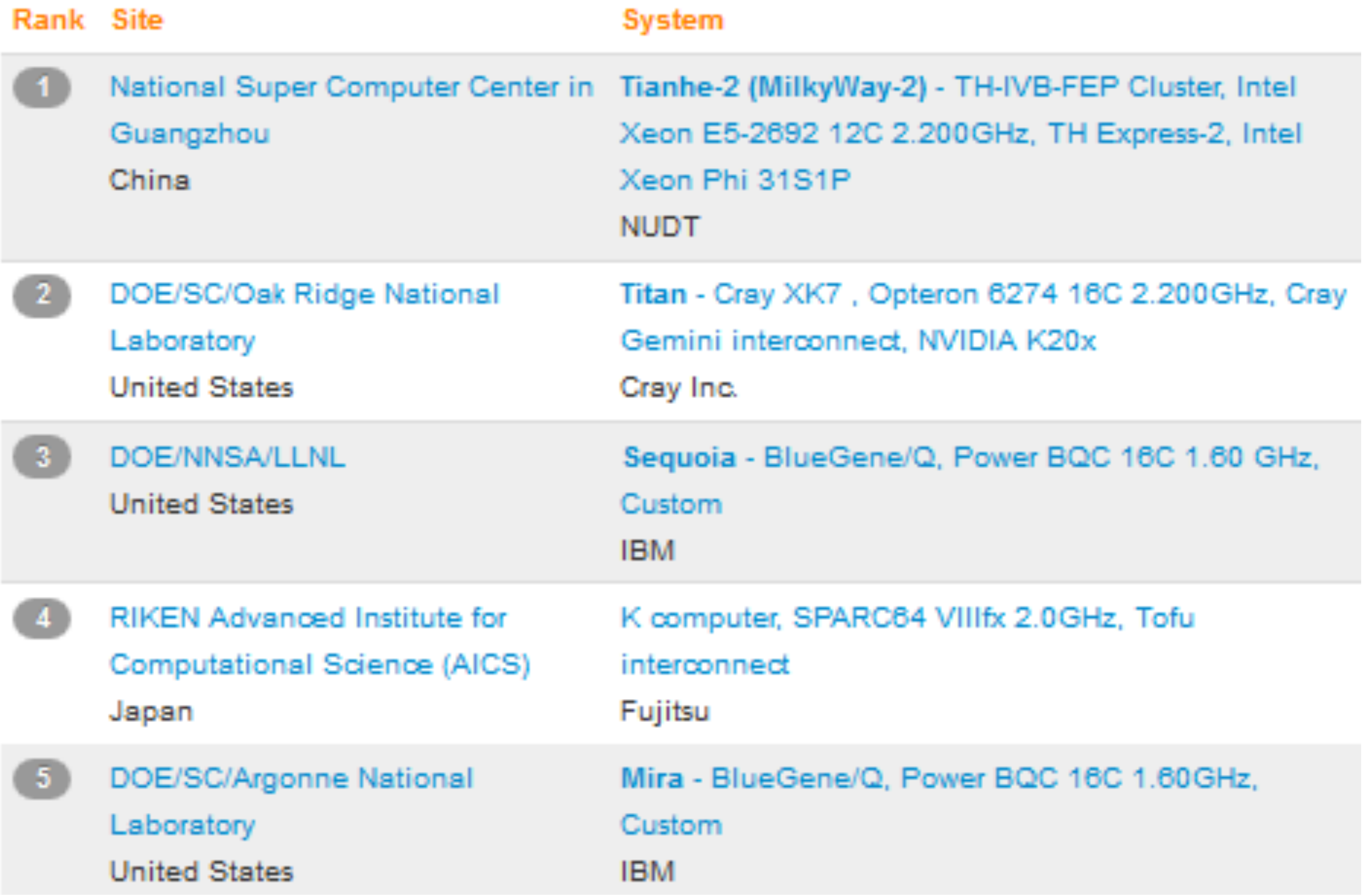
IBM Memo on CDC6600

Thomas Watson Jr., IBM CEO, August 1963:

> *"Last week, Control Data ... announced the 6600 system. I understand that in the laboratory developing the system there are only 34 people including the janitor. Of these, 14 are engineers and 4 are programmers... Contrasting this modest effort with our vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer."*

To which Cray replied: *"It seems like Mr. Watson has answered his own question."*

Top 500 Systems

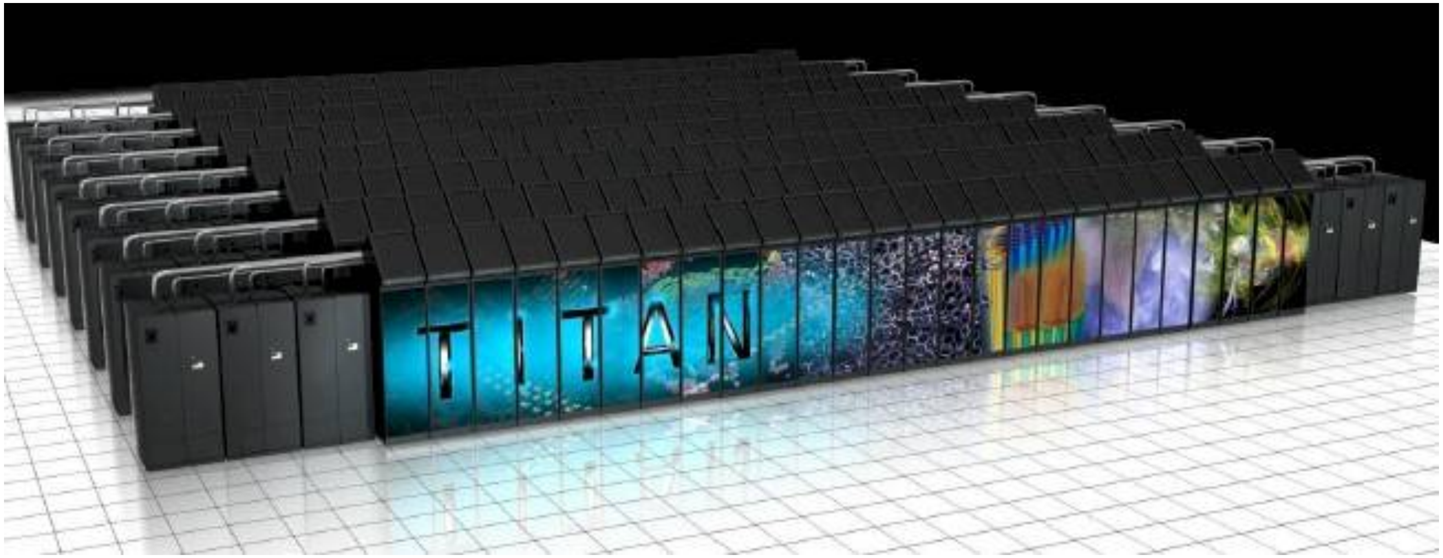


Rank	Site	System
1	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT
2	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.
3	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM
4	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu
5	DOE/SC/Argonne National Laboratory United States	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz, Custom IBM

LINPACK & LAPACK: Software libraries for performing linear algebra

Oak Ridge Titan

- 560,640 cores
- LinkPack performance 17,590 TFlop/s
- Theoretical peak 27,112.5 TFlop/s
- 8,209.00 kW
- 710,144 GB
- Opteron 6274 16C 2.2GHz



NERSC (LBNL) Cori

- Cray XC40 supercomputer
- Theoretical Peak performance 1.92 Petaflops/sec
- 1,630 computes nodes, 52,160 cores in total
- Cray Aries high-speed interconnect with Dragonfly topology as on Edison (0.25 µs to 3.7 µs MPI latency, ~8GB/sec MPI bandwidth)
- Aggregate memory: 203 TB
- Scratch storage capacity: 30 PB

CDC 6600: A Load/Store Architecture

- Separate instructions to manipulate three types of reg.

8 60-bit data registers (X)
8 18-bit address registers (A)
8 18-bit index registers (B)

- All arithmetic and logic instructions are reg-to-reg

6	3	3	3
opcode	i	j	k

Ri ← (Rj) op (Rk)

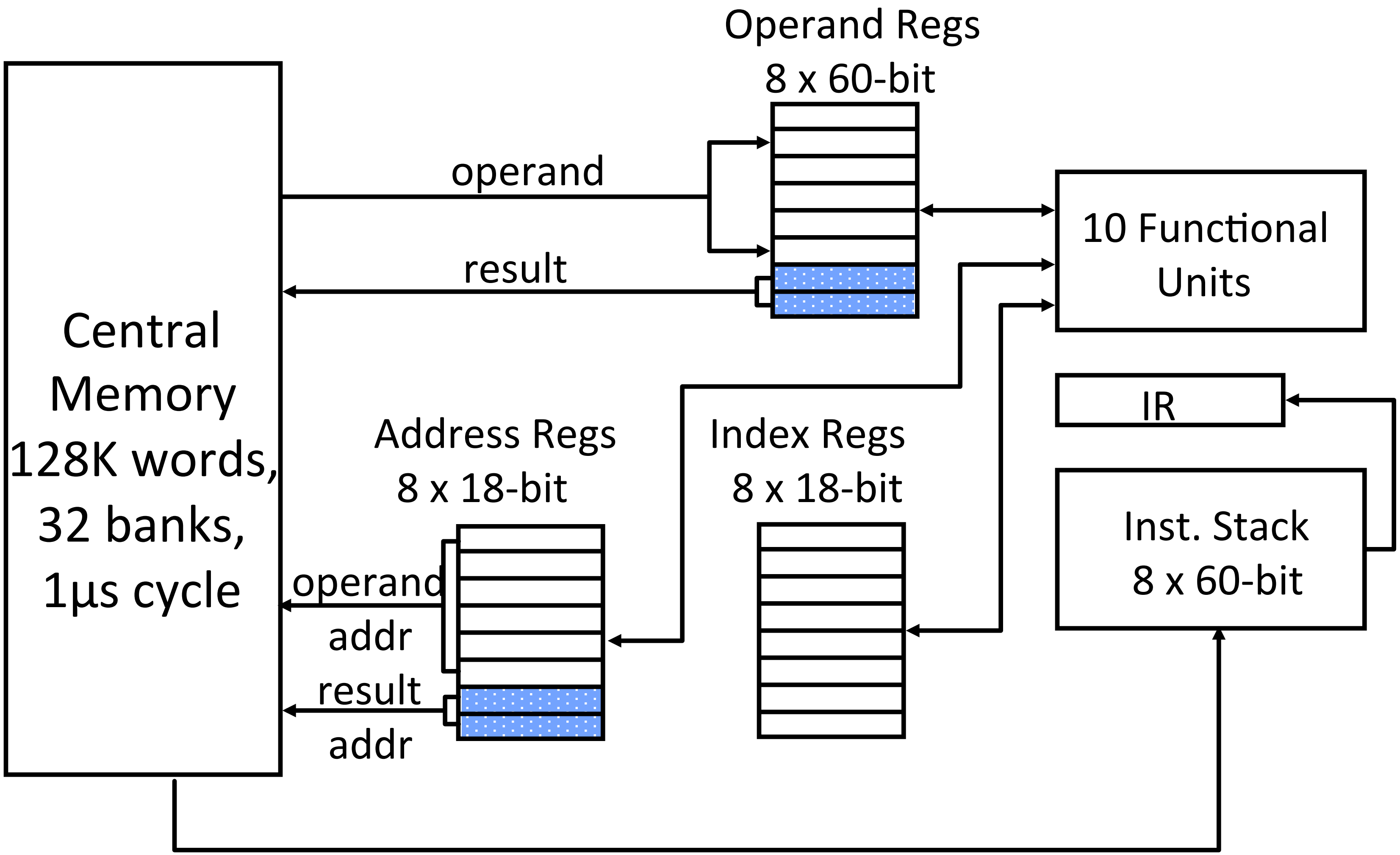
- Only Load and Store instructions refer to memory!

6	3	3	18
opcode	i	j	disp

Ri ← M[(Rj) + disp]

Touching address registers 1 to 5 initiates a load
6 to 7 initiates a store
- *very useful for vector operations*

CDC 6600: Datapath



CDC6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
  - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions inbetween
- CDC6600 has multiple parallel but unpipelined functional units
  - E.g., 2 separate multipliers
- Follow-on machine CDC7600 used pipelined functional units
  - Foreshadows later RISC designs

CDC6600: Vector Addition

```
        B0  <- - n
loop:   JZE   B0, exit
        A0  <-  B0 + a0           load X0
        A1  <-  B0 + b0           load X1
        X6  <-  X0 + X1
        A6  <-  B0 + c0           store X6
        B0  <-  B0 + 1
        jump loop
```

Ai = address register
Bi = index register
Xi = data register

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 sets

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

VLIW vs Vector

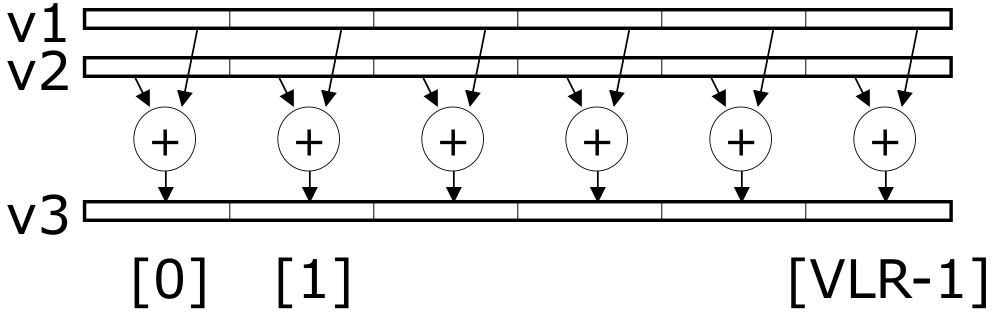
- VLIW takes advantage of instruction level parallelism (ILP) by specifying instructions to execute in parallel

Int Op 1	Int Op 2	Mem Op 1	Mem Op 2	FP Op 1	FP Op 2

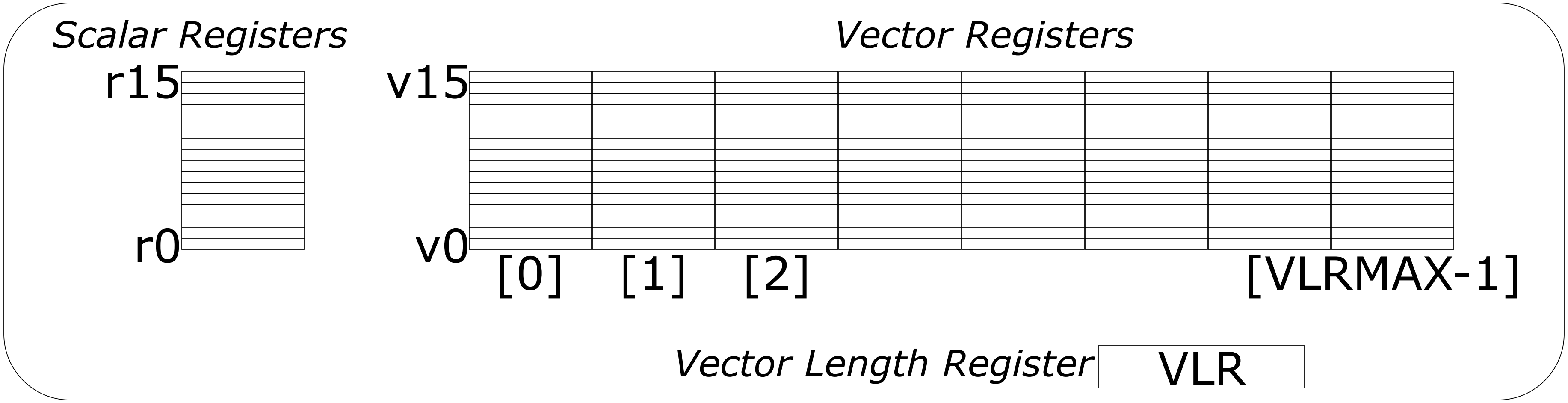
- Vector architectures perform the same operation on multiple data elements
  - **Data-level parallelism**

Vector Arithmetic Instructions

ADDV v3, v1, v2

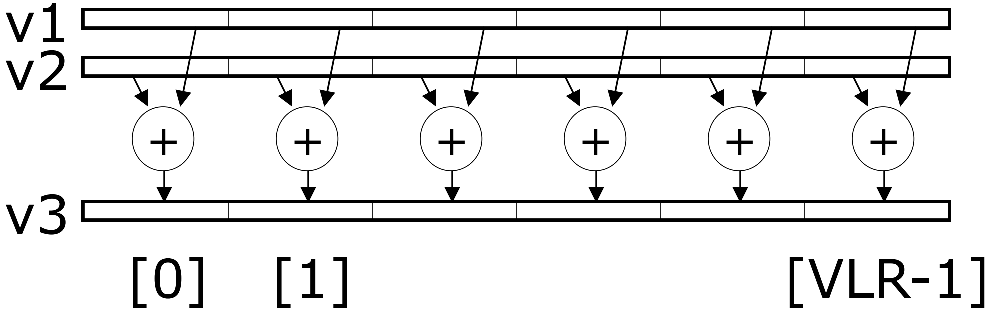


Vector Programming Model



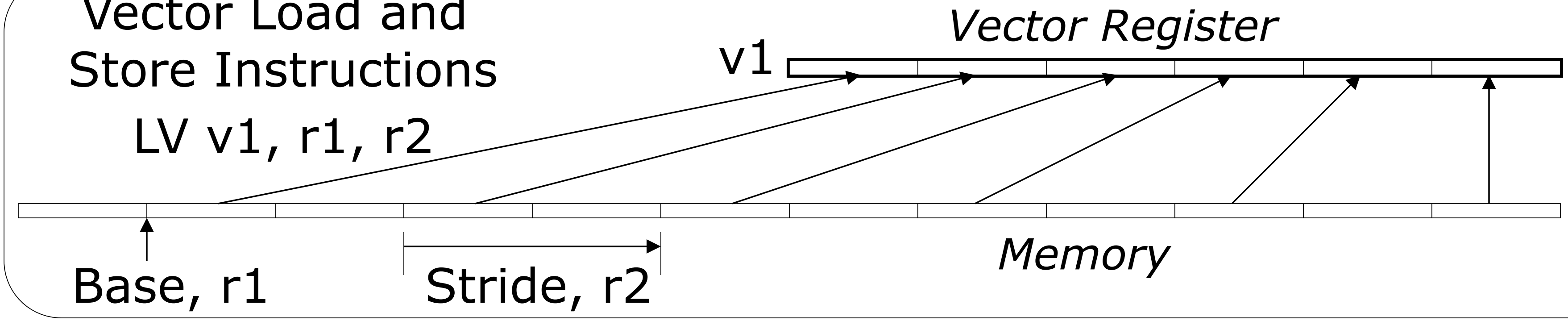
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

LV v1, r1, r2



Control Information

- VLR limits the highest vector element to be processed by a vector instruction
  - VLR is loaded prior to executing the vector instruction with a special instruction

- Stride for load/stores:
  - Vectors may not be adjacent in memory addresses
  - E.g., different dimensions of a matrix
  - Stride can be specified as part of the load/store

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
```

Flynn's Taxonomy

- Single instruction, single data (SISD)
  - E.g., our in-order processor
- Single instruction, multiple data (SIMD)
  - Multiple processing elements, same operation, different data
  - Vector
  - Multiple processing units execute the same instruction on different data in a lockstep. Either all complete or none do. Therefore, all units have to execute the same instruction at a given time
- Multiple instruction, multiple data (MIMD)
  - Multiple autonomous processors executing different instructions on different data
  - Most common and general parallel machine
- Multiple instruction, single data (MISD)
  - Why would anyone do this?

More Categories

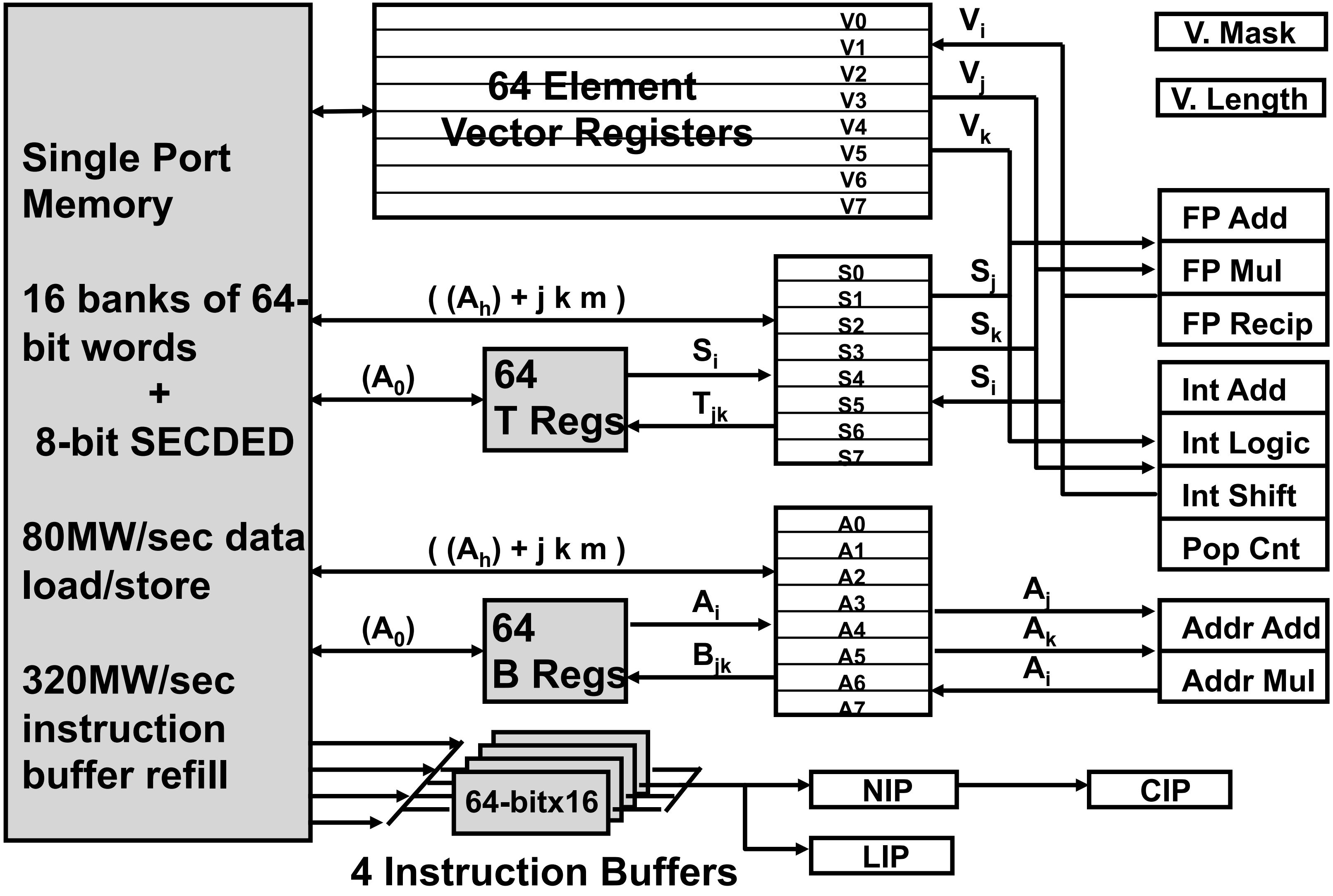
- Single program, multiple data (SPMD)
  - Multiple autonomous processors execute the program at independent points
  - Difference with SIMD: SIMD imposes a lockstep
  - Programs at SPMD can be at independent points
  - SPMD can run on general purpose processors
  - Most common method for parallel computing

- Multiple program, multiple data (MPMD)
  - Multiple autonomous processors simultaneously operating at least 2 independent programs

Vector Supercomputers

- Epitomy: 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

Cray-1 (1976)



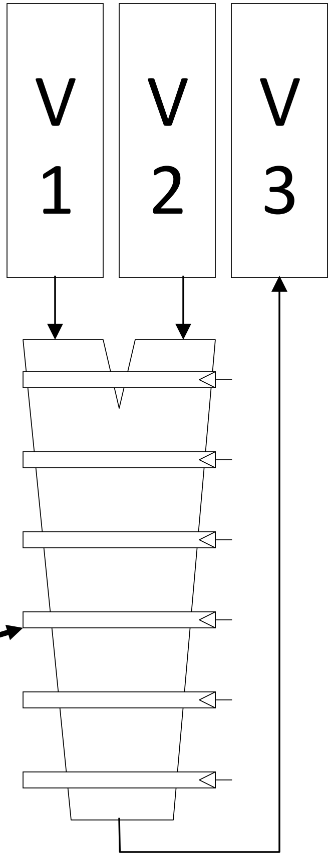
***memory bank cycle* 50 ns** ***processor cycle* 12.5 ns (80MHz)**

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)

Vector Arithmetic Execution

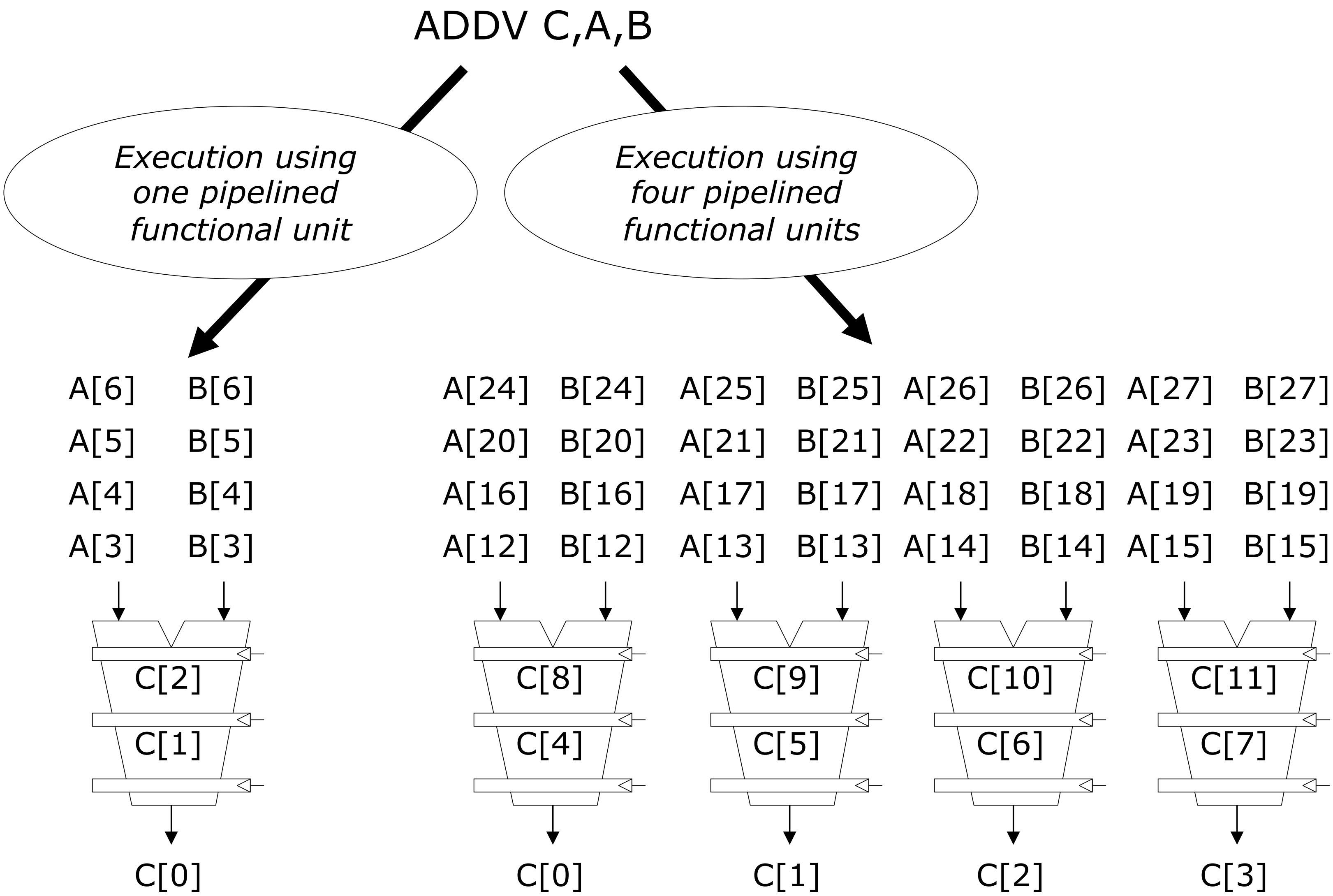
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



Six stage multiply pipeline

V3 <- v1 * v2

Vector Instruction Execution



ADDV 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]

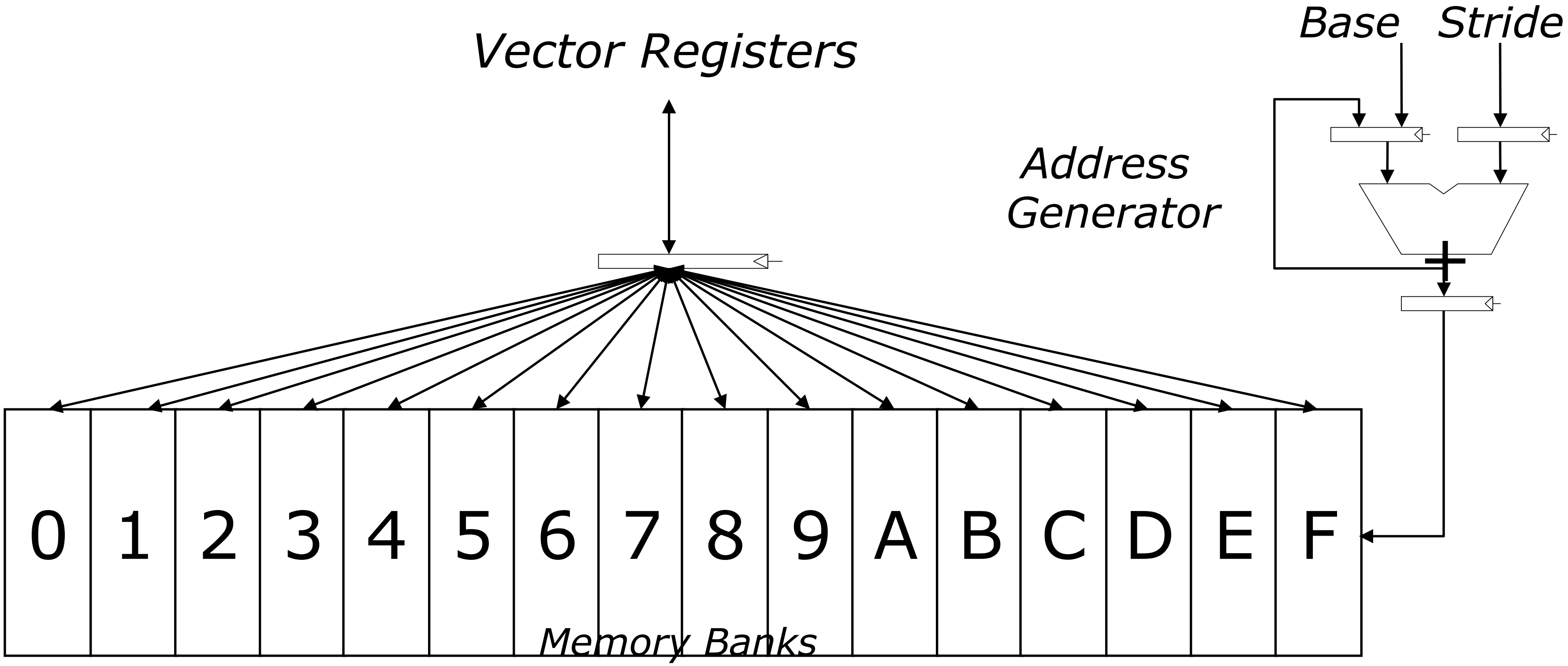
C[8] C[9] C[10] C[11]
C[4] C[5] C[6] C[7]
C[0] C[1] C[2] C[3]

How Do Vector Architectures Affect Memory?

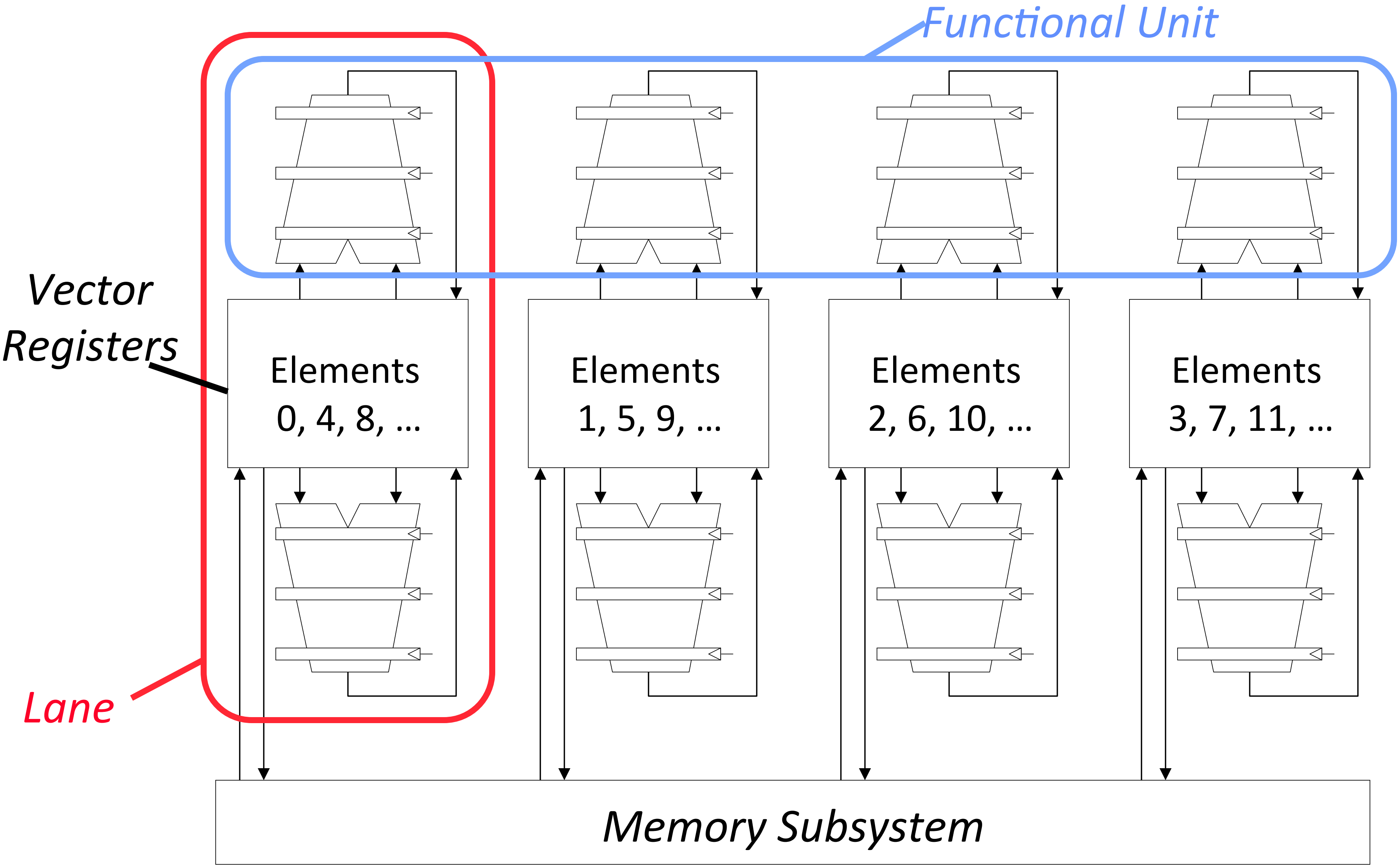
Interleaved Vector Memory System

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

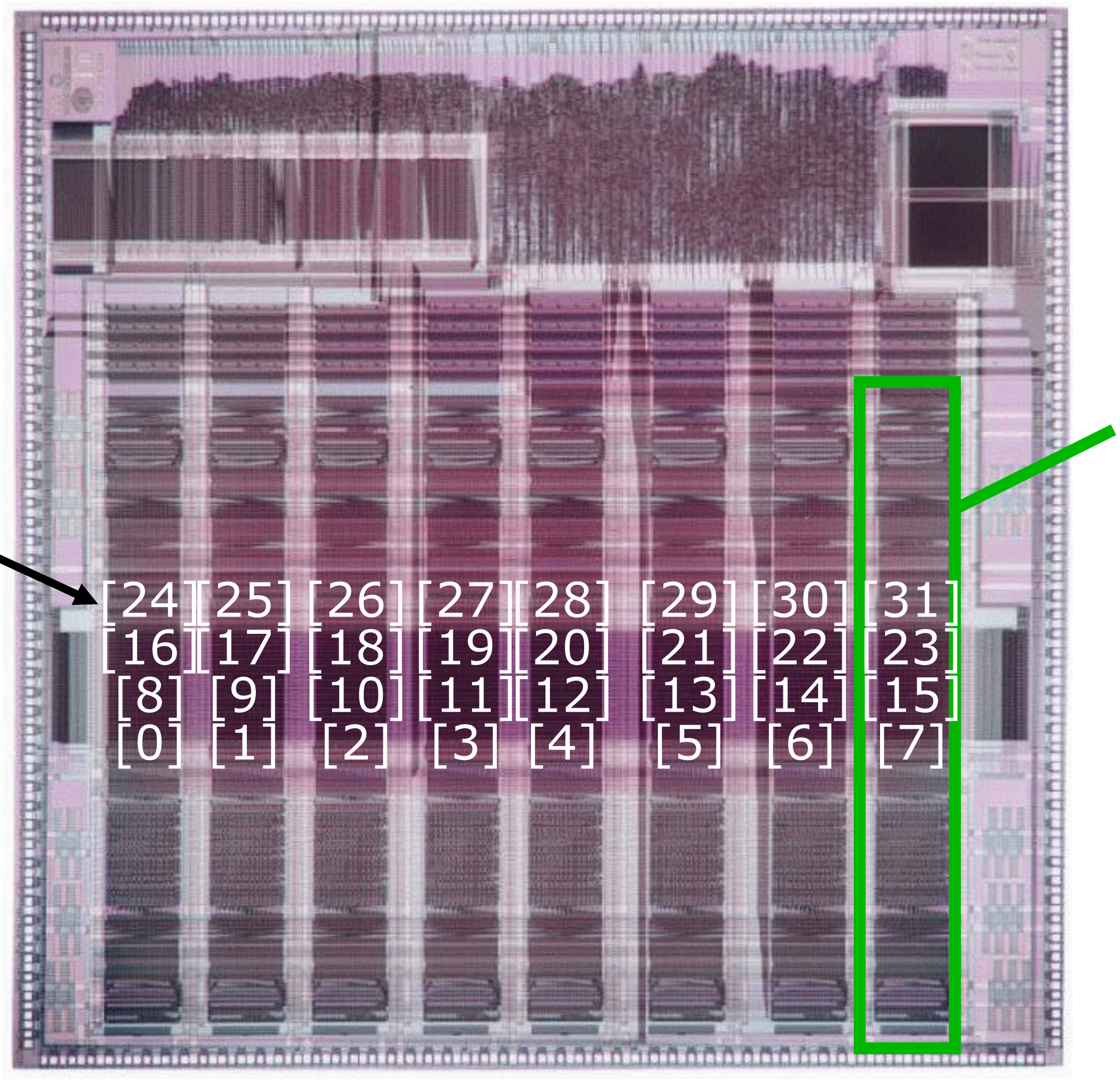
- *Bank busy time*: Time before bank ready to accept next request



Vector Unit Structure



T0 Vector Microprocessor (UCB/ICSI, 1995)

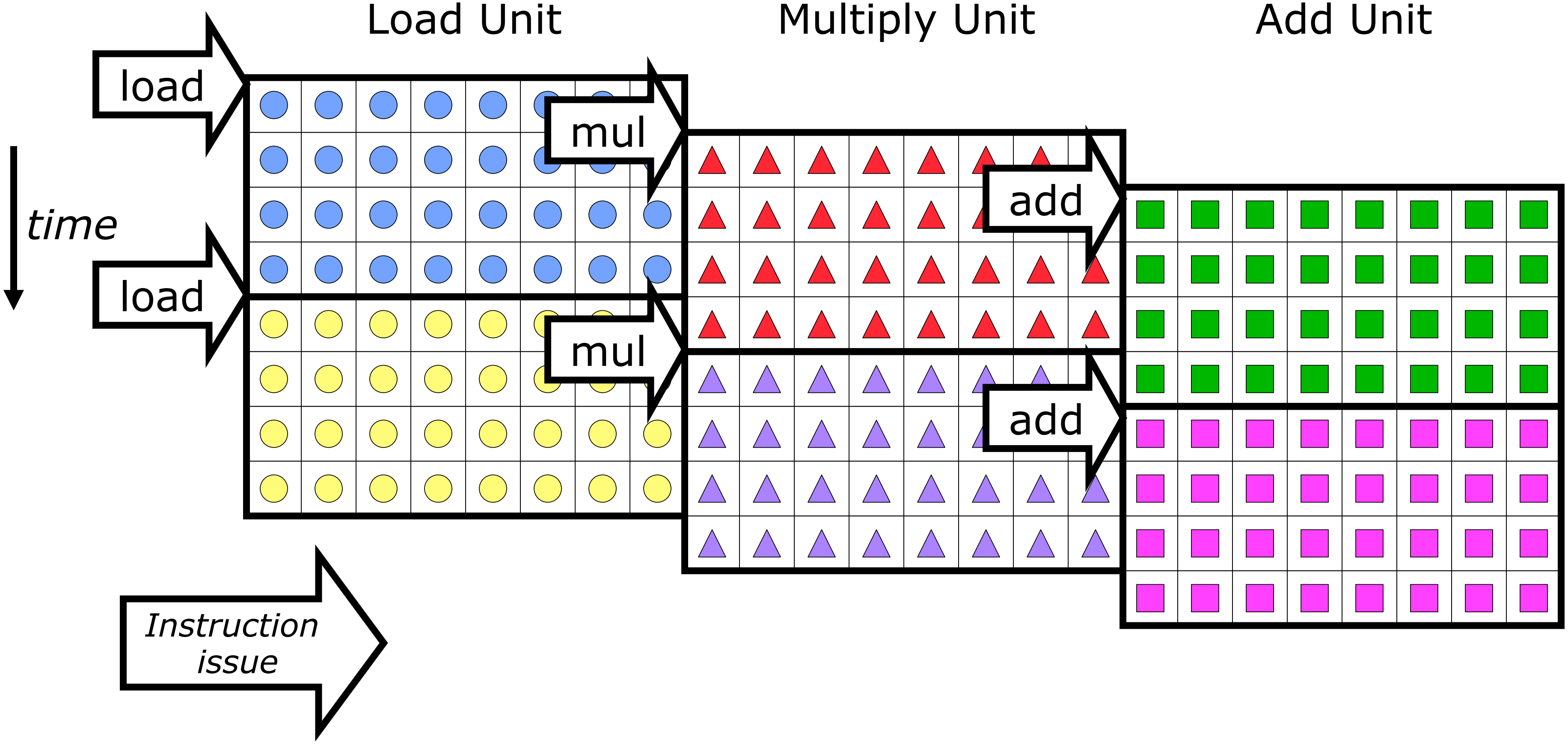


Vector register elements striped over lanes

Lane

Vector Instruction Parallelism

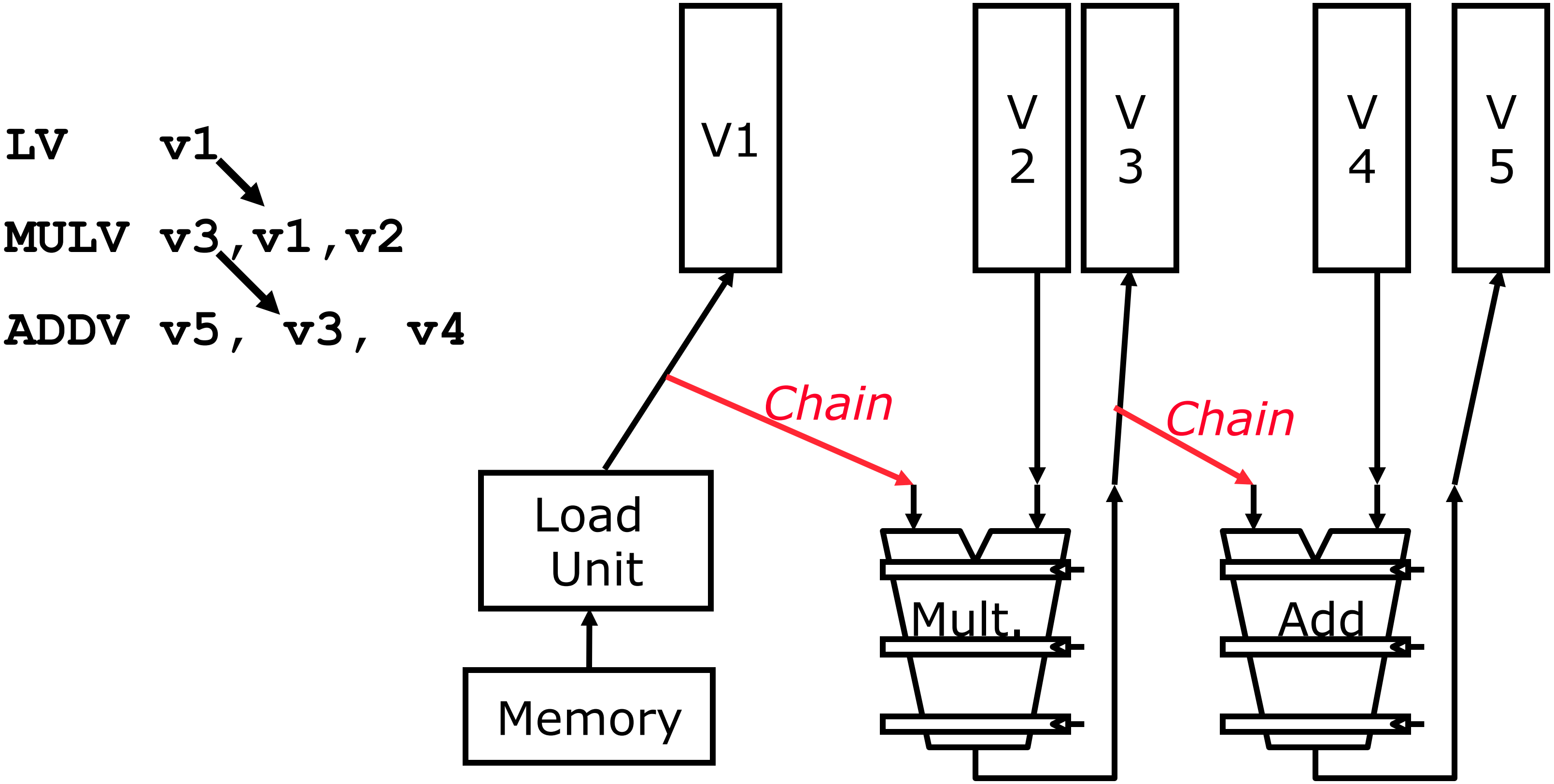
- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes



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

Vector Chaining

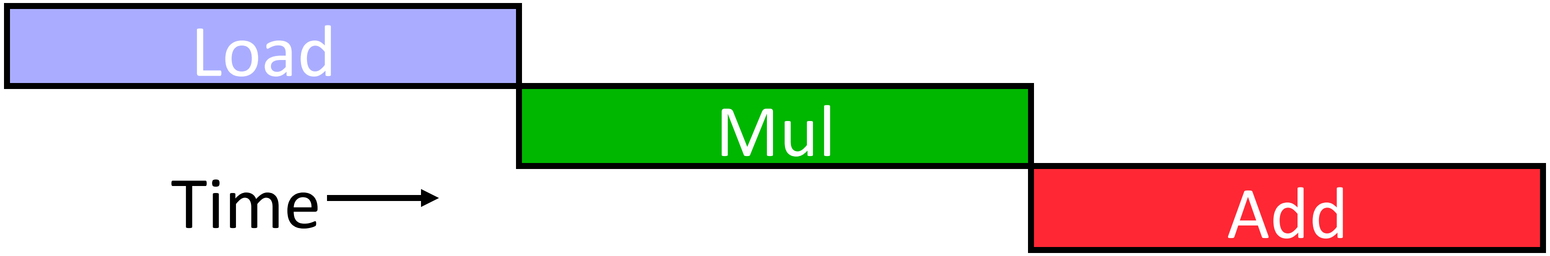
- Vector version of register bypassing
  - introduced with Cray-1



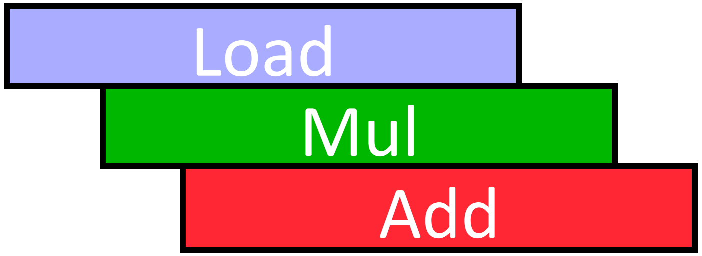
```
LV    v1
MULV  v3,v1,v2
ADDV  v5, v3, v4
```

Vector Chaining Advantage

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

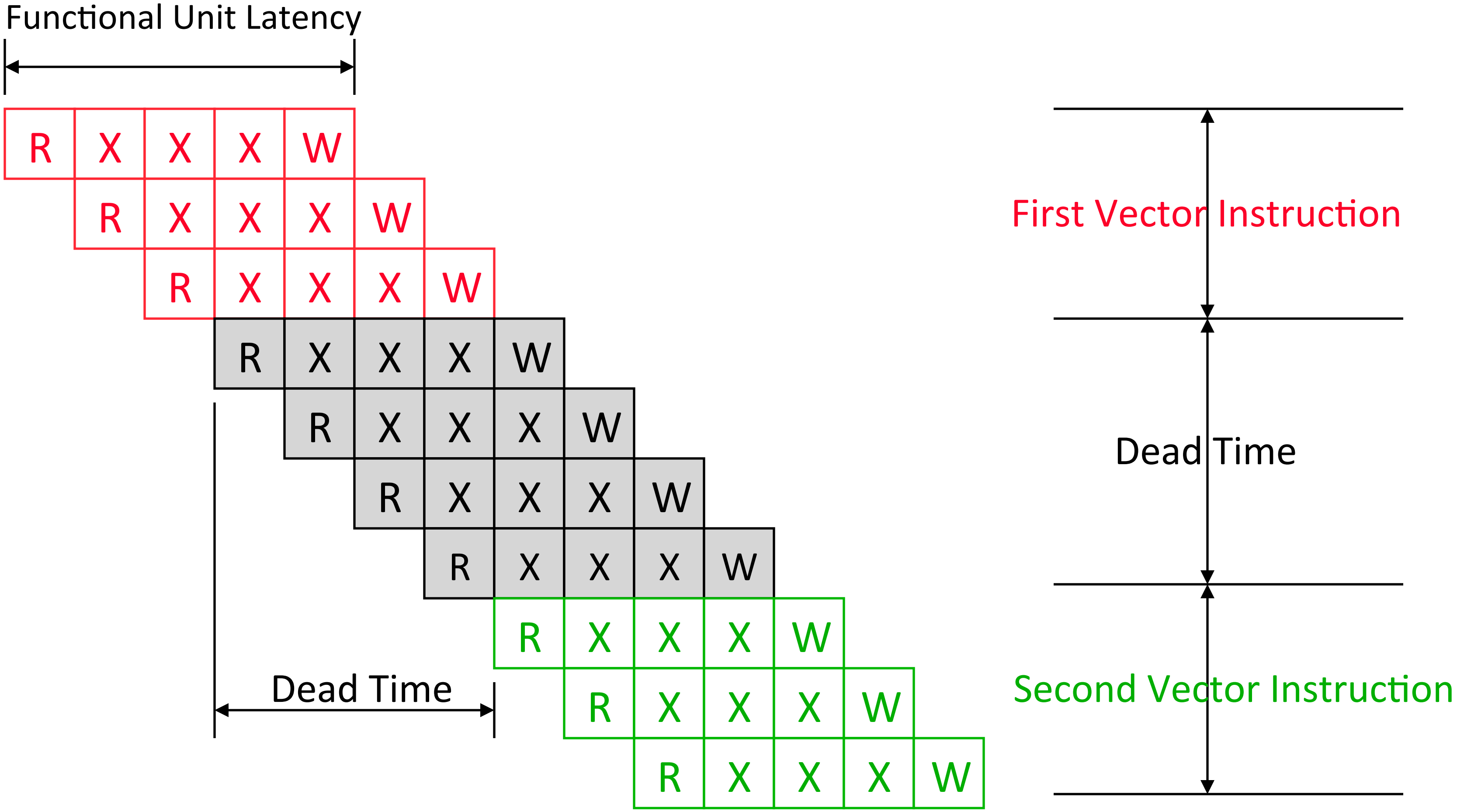


- With chaining, can start dependent instruction as soon as first result appears

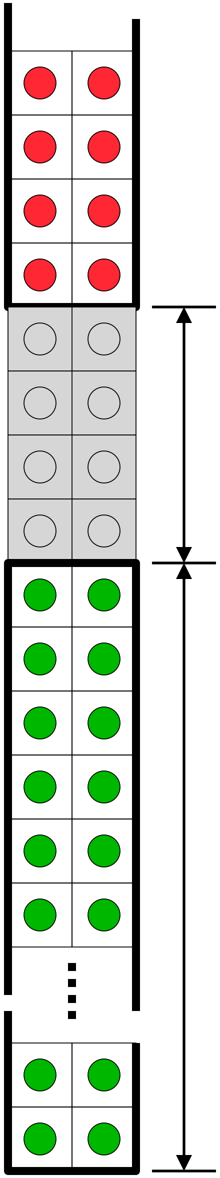


Vector Startup

- 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). Some pipelines reduce control logic by requiring dead time between instructions to the same vector unit



Dead Time and Short Vectors

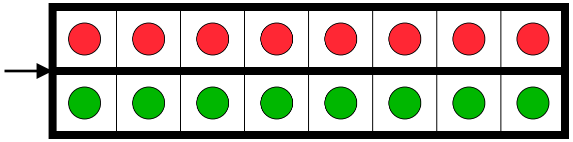


4 cycles dead time

64 cycles active

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

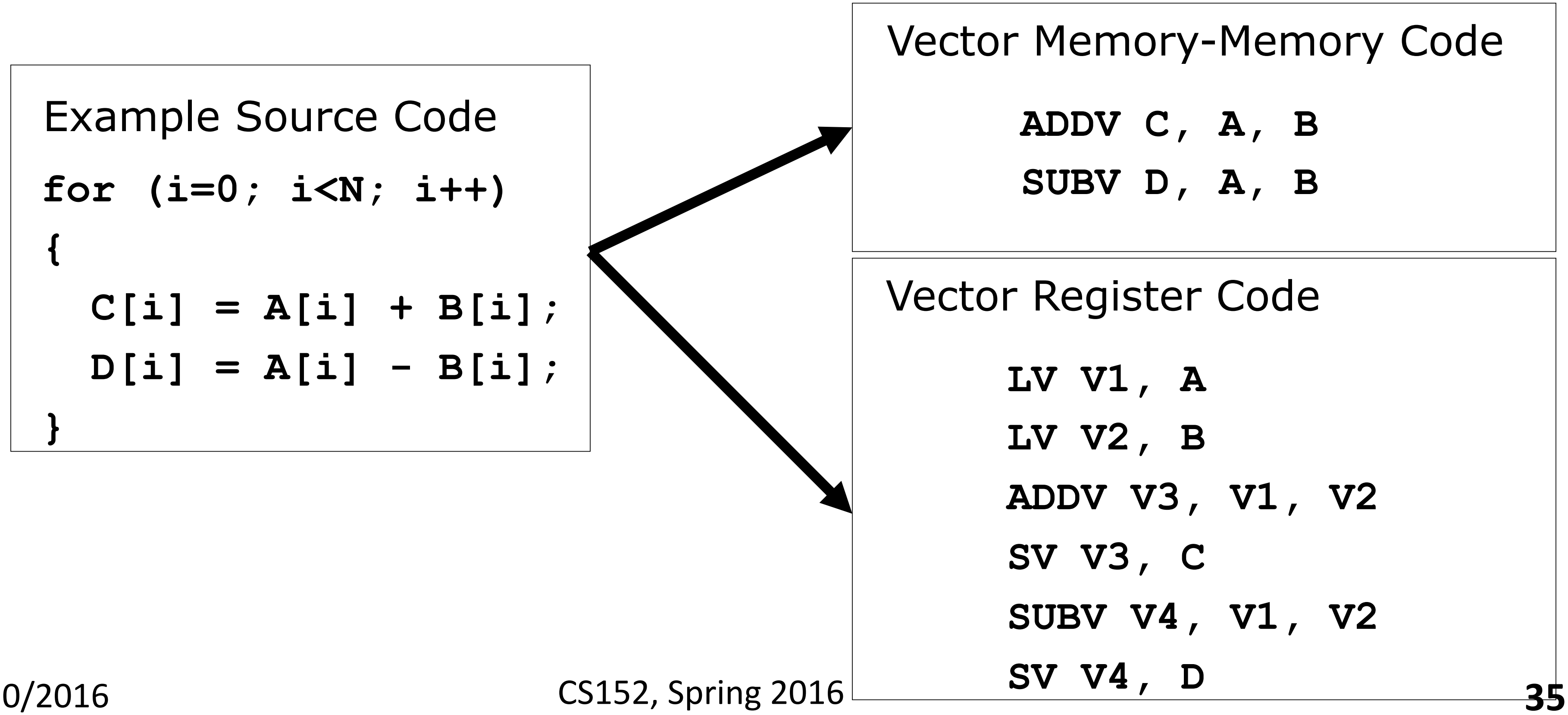
No dead time



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

Vector 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

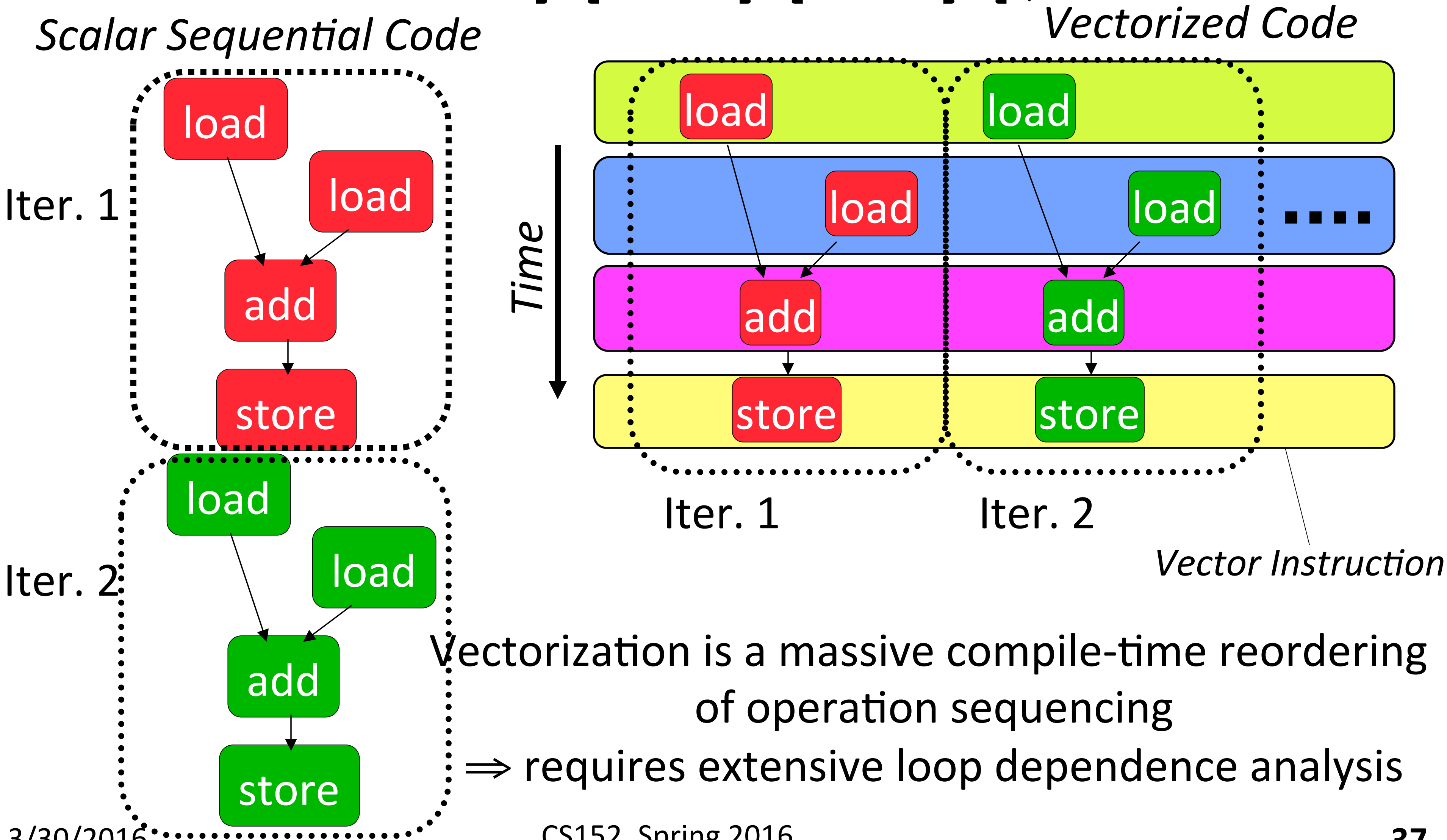
```
LV V1, A
LV V2, B
ADDV V3, V1, V2
SV V3, C
SUBV V4, V1, V2
SV V4, D
```

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 (VMM) for vectors < 100 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

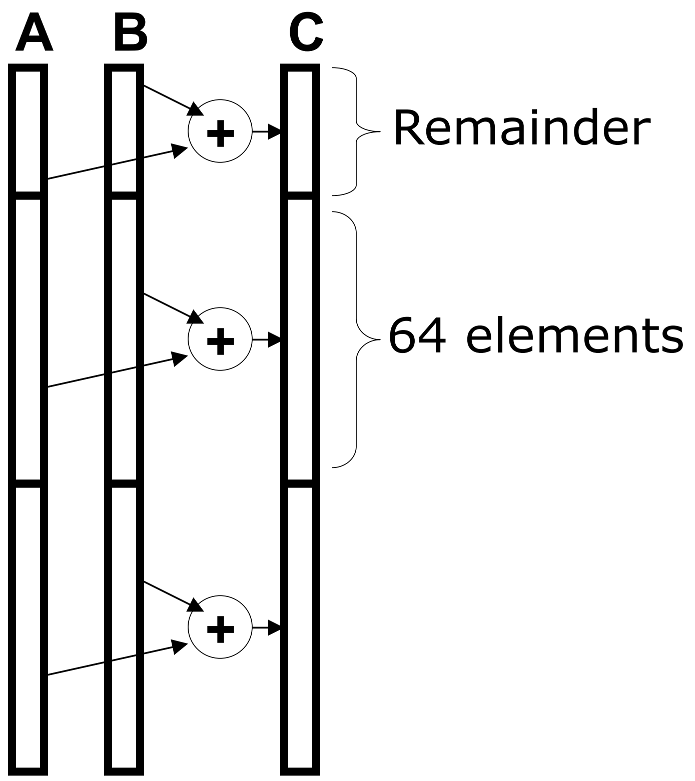
⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in 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 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

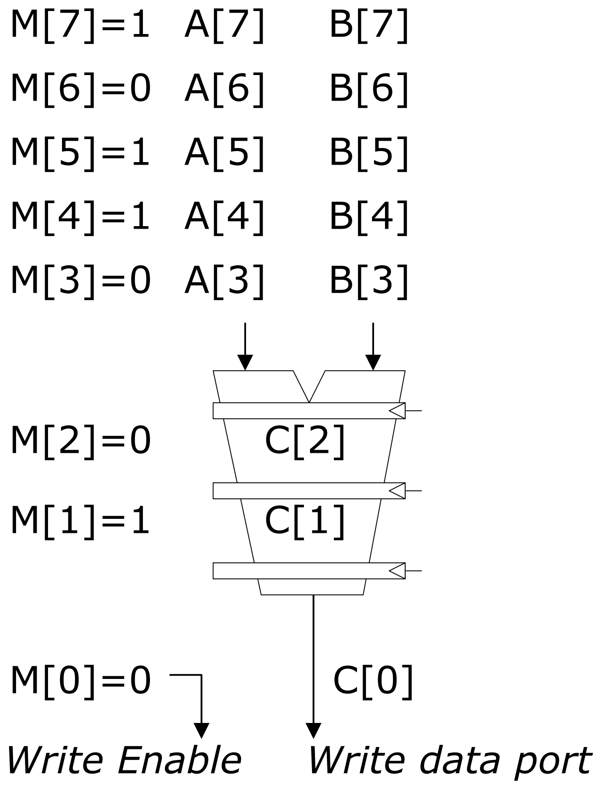
Code example:

```
CVM              # Turn on all elements
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

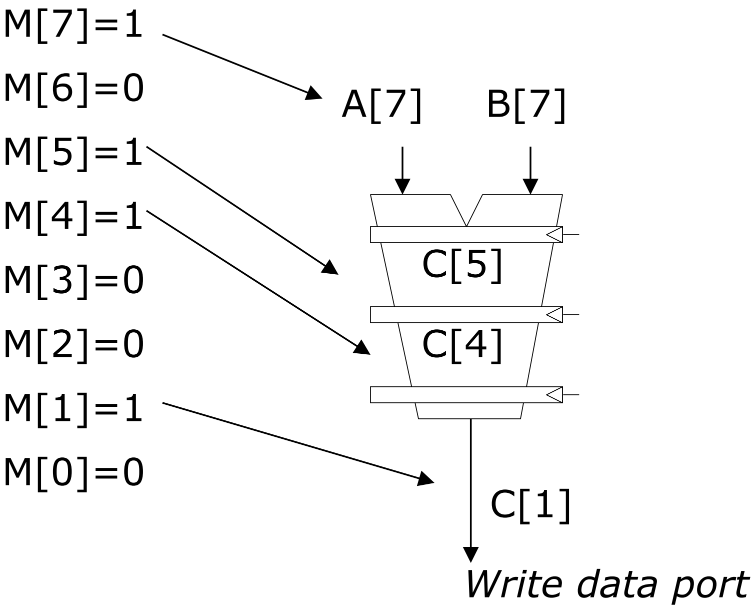
Simple Implementation

– 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



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)
```

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

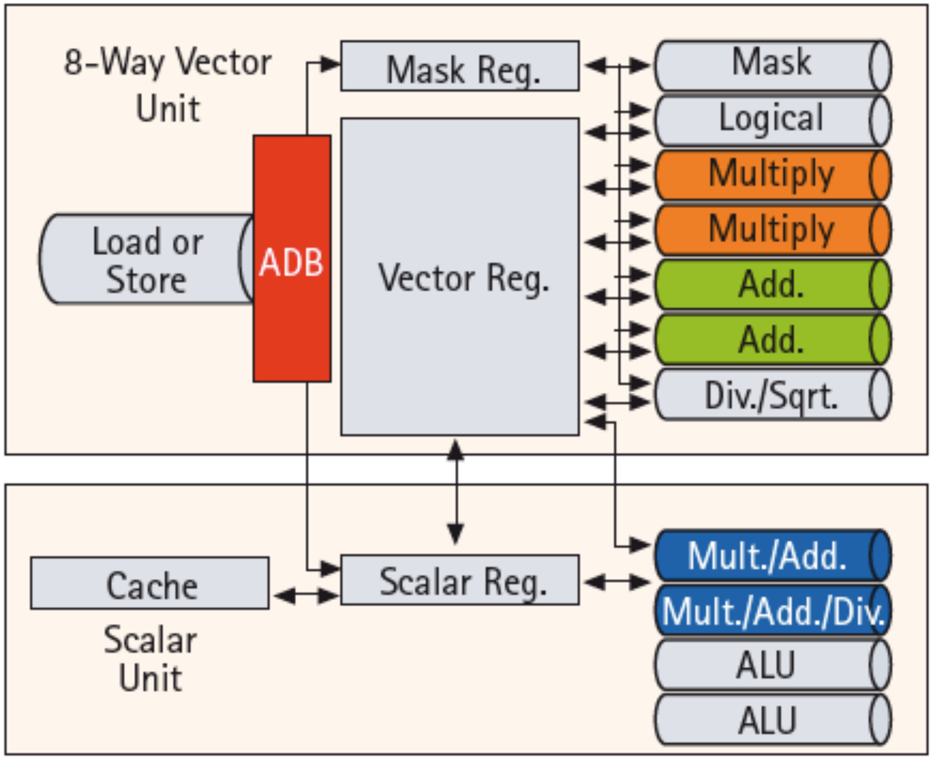
Histogram example:

```
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
```

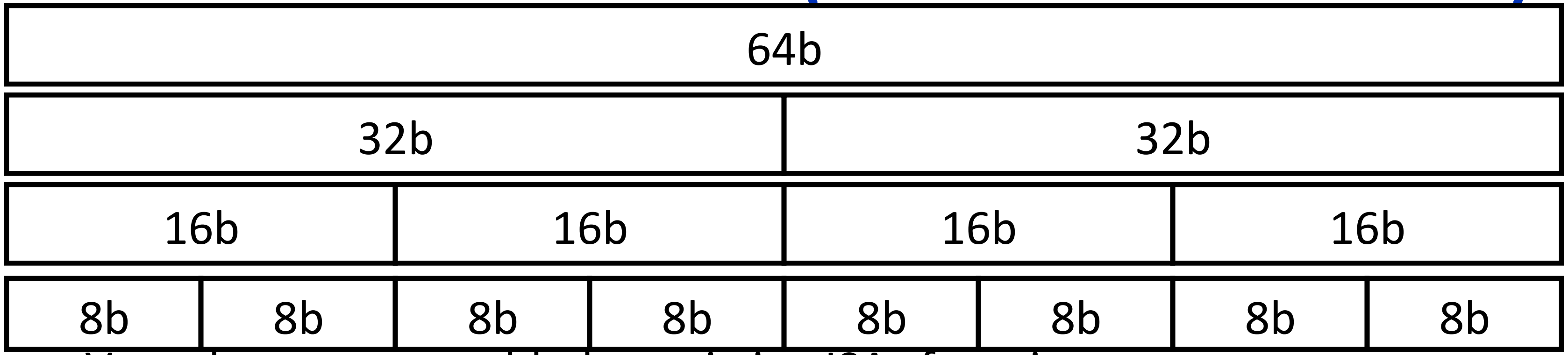
A Modern Vector Super: NEC SX-9 (2008)



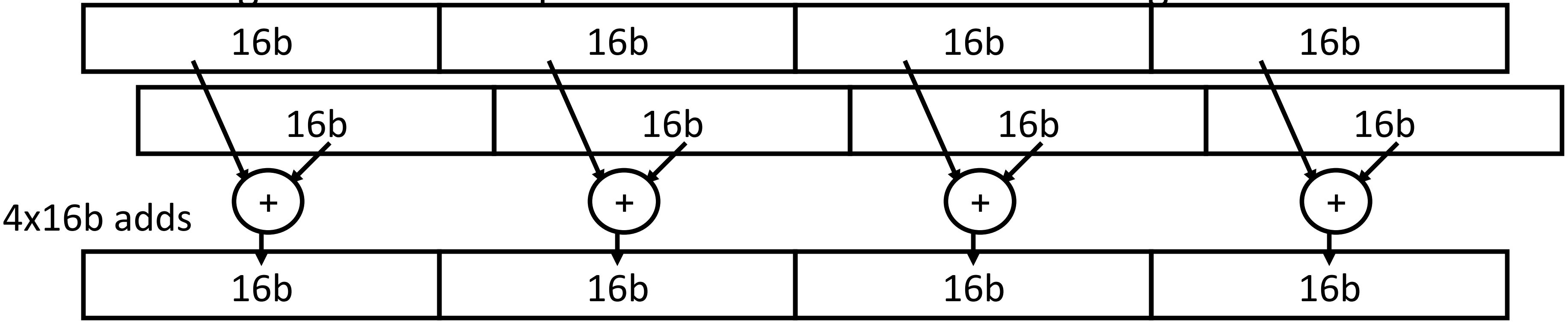
- 65nm CMOS technology
- Vector unit (3.2 GHz)
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 64-bit functional units: 2 multiply, 2 add, 1 divide/sqrt, 1 logical, 1 mask unit
  - 8 lanes (32+ FLOPS/cycle, 100+ GFLOPS peak per CPU)
  - 1 load or store unit (8 x 8-byte accesses/ 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)

Multimedia Extensions (aka 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 for Intel AVX
- Single instruction operates on all elements within register

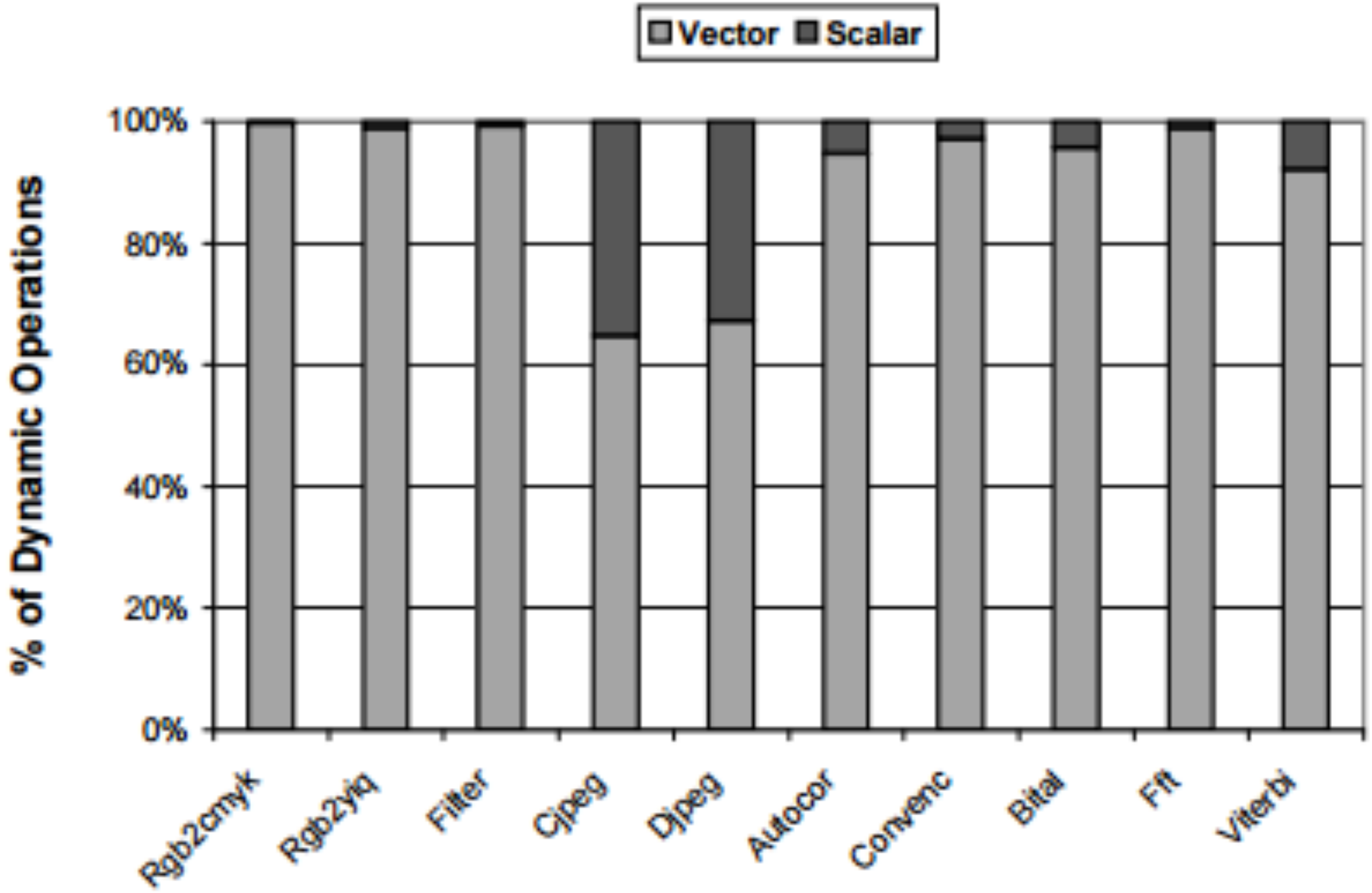


Multimedia Extensions versus Vectors

- 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)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

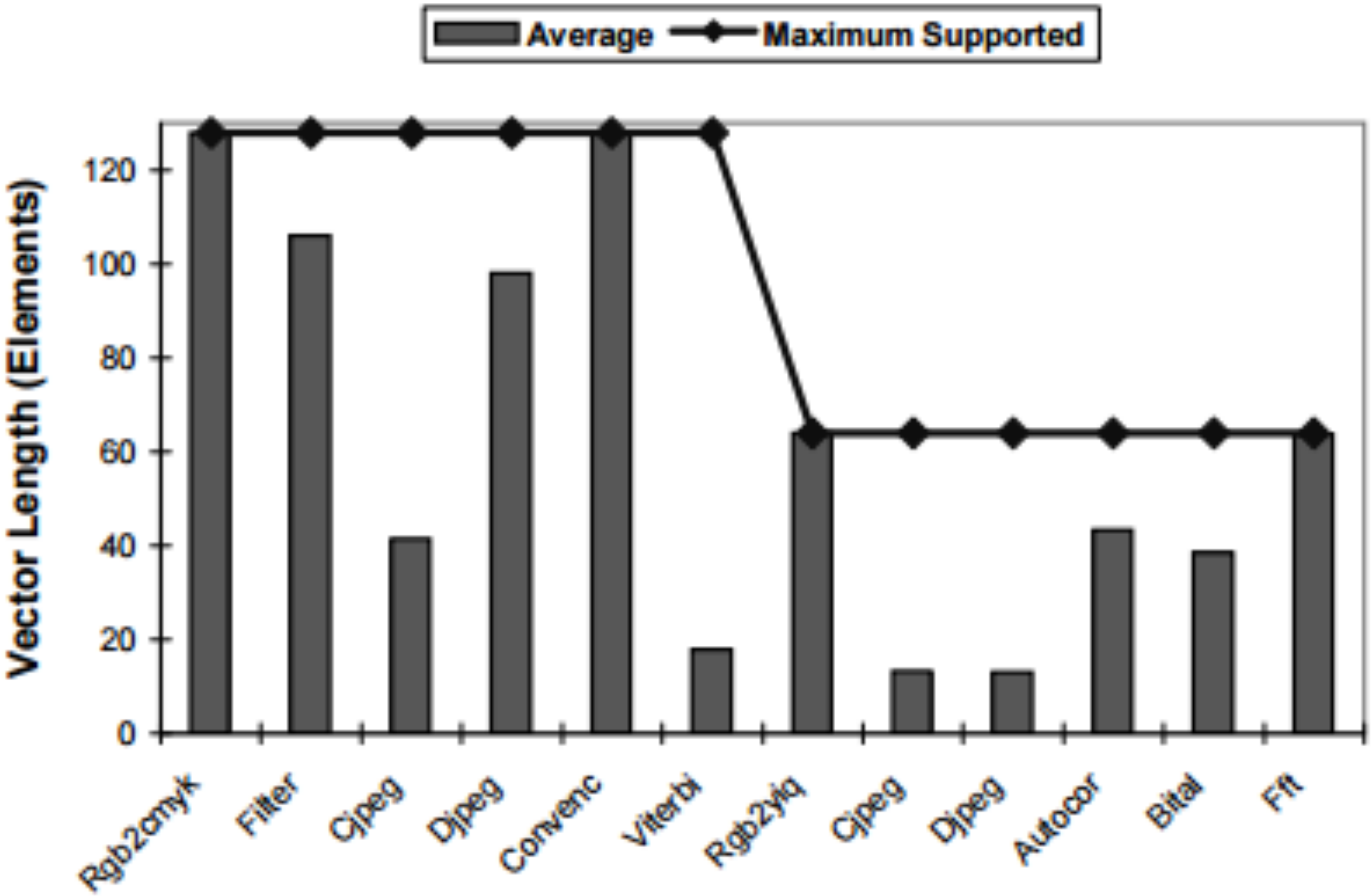
Degree of Vectorization

- Compilers are good at finding data-level parallelism

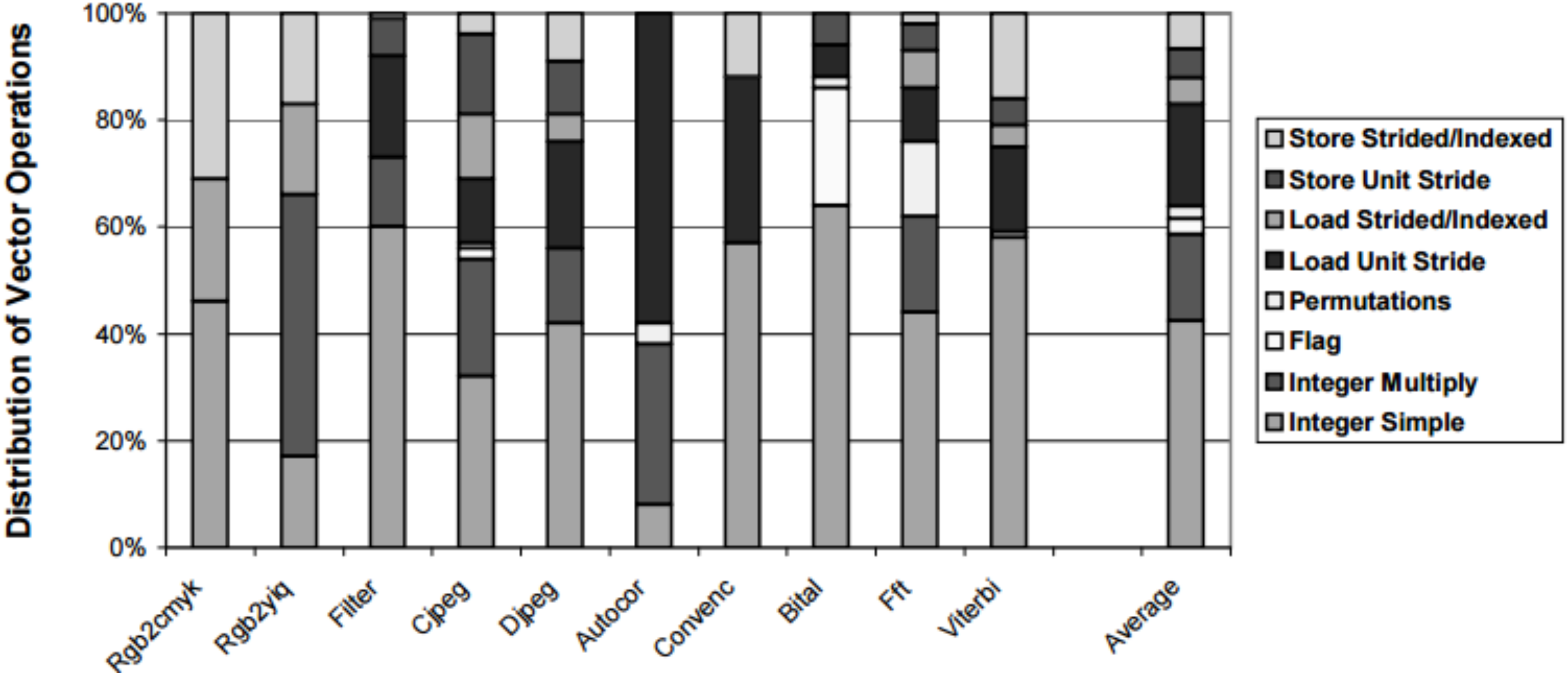


Average Vector Length

- Maximum depends on if becnhmarks use 16 bit or 32 bit operations



Distribution of Instructions



Question of the Day

- Can Vector and VLIW combine?

- Yes!
- Fujitsy FR-V can process both VLIW and vector instructions
- Exploits both instruction- and data-level parallelism

Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
- MIT material derived from course 6.823
- UCB material derived from course CS252
- "Vector Vs. Superscalar and VLIW Architectures for Embedded Multimedia Benchmarks". Christos Kozyrakis and David Patterson. MICRO-35. 2002