Vector Processors

some slides from:

Krste Asanovic

Electrical Engineering and Computer Sciences

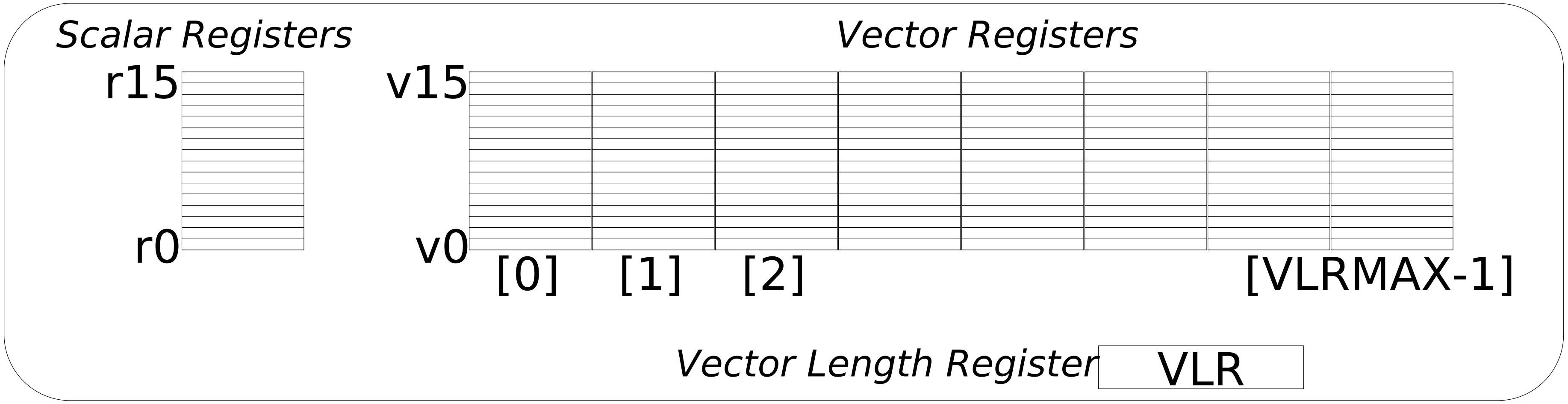
University of California, Berkeley

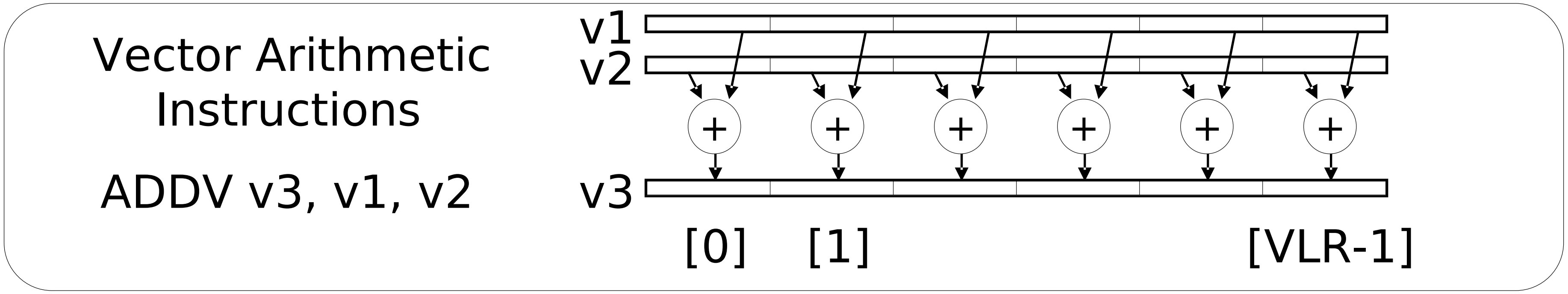
Also from David Gregg
SCSS, Trinity College Dublin

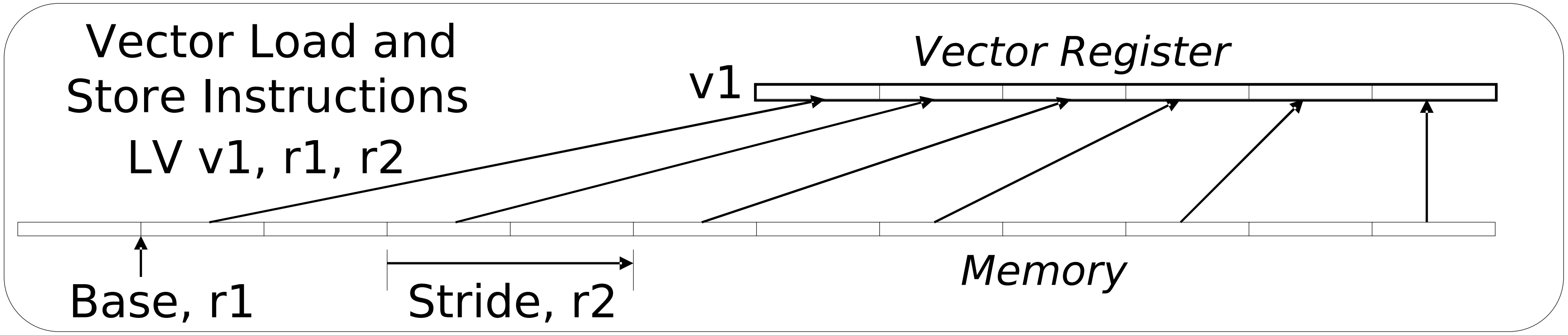
The Rise of SIMD

- SIMD is good for applying identical computations across many data elements
  - E.g. `for(i = 0; I < 100; i++) { A[i] = B[i] + C[i];}`
  - Also “data level parallelism”
- SIMD is energy efficient
  - Less control logic per functional unit
  - Less instruction fetch and decode energy
- SIMD computations tend to bandwidth-efficient and latency tolerant
  - Memory systems (caches, prefetchers, etc) are good at sequential scans through arrays
- Easy examples
  - Dense linear algebra
  - Computer graphics (which includes a lot of dense linear algebra)
  - Machine vision
  - Digital signal processing
- Harder examples – can be made to work to gain benefits above
  - Database queries
  - Sorting

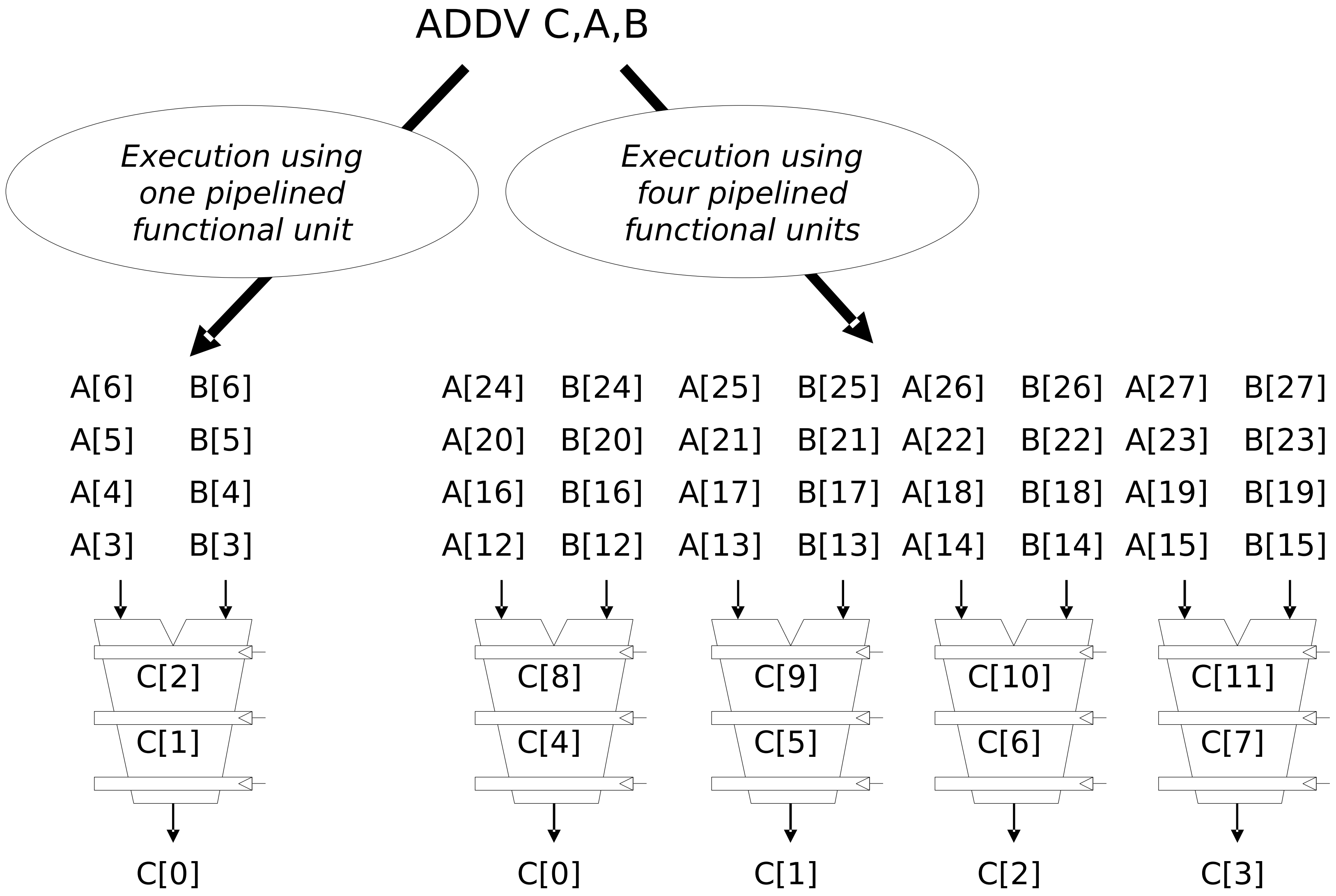
Vector Programming Model







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]

Vector Memory-Memory versus Vector Register Machines

- Vector memory-memory instructions hold all vector operands in main memory (not vector registers)
- 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];
}
```

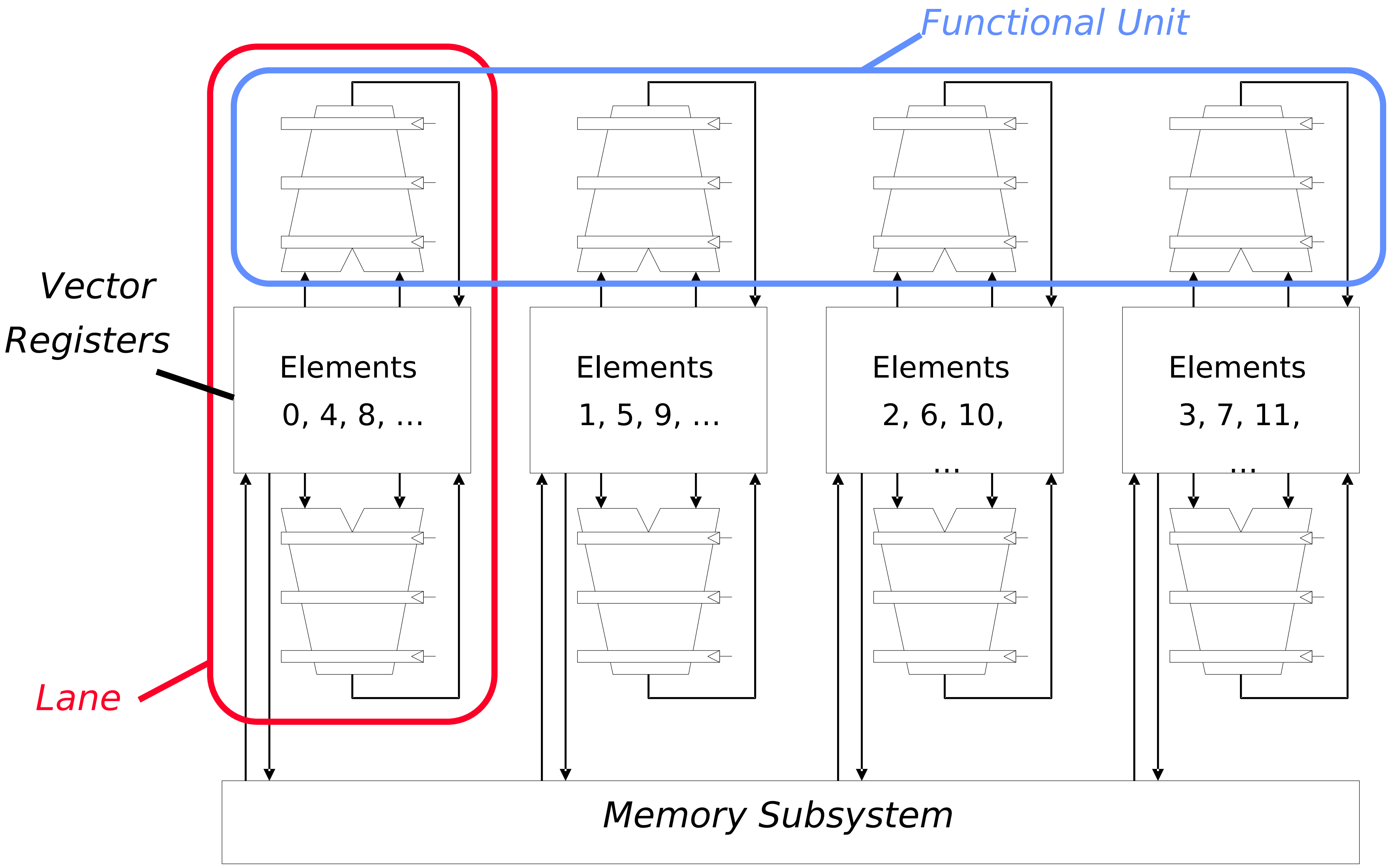
Vector Memory-Memory Code

```
ADDV C, A, B
SUBV D, A, B
```

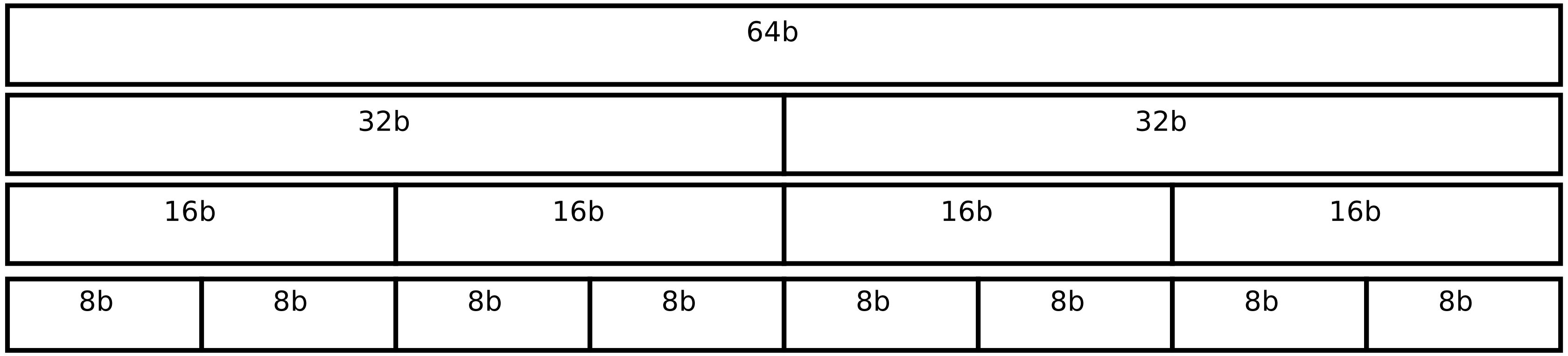
Vector Register Code

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

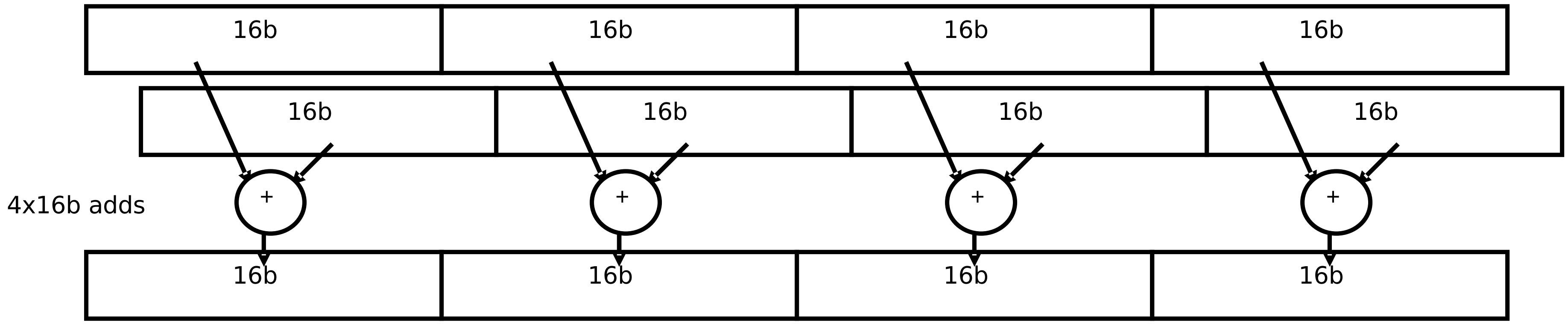
Vector Unit Structure



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)
  - Intel AVX spec, 256b vector registers (expandable up to 1024b)

Modern Vector Computers

- Modern vector machines have relatively short vectors
  - ARM NEON: 16 bytes, Intel SSE: 16 bytes, Intel AVX:32 bytes
  - But the trend is for them to grow longer
  - AVX-512 has 64 byte vectors, i.e. 16 floats
- Older vector machines used much longer vectors
  - Cray 1 vector supercomputer used vectors of 64 floats, i.e. 256 bytes

- Modern vector machines always use vector registers for everything
  - Modern short vector registers of 16 or 32 bytes don’t require much space on chip
- Some (but not all) older vector machines took data directly from memory

Modern Vector Computers

- Modern vector machines use a separate arithmetic/floating point unit per lane
  - Four parallel floating point adders/multipliers in SSE implementations
- Older vector machines used as little as one arithmetic/floating point unit to implement vector instruction
  - But <u>very</u> deeply pipelined
  - Goal was to push as much work through the pipelined FP unit as possible

- Vector architectures are increasingly important
  - Especially for low-energy computation
  - It’s worthwhile looking back to the time when vector computers were last really popular and successful

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 $\equiv$ Vector Machine

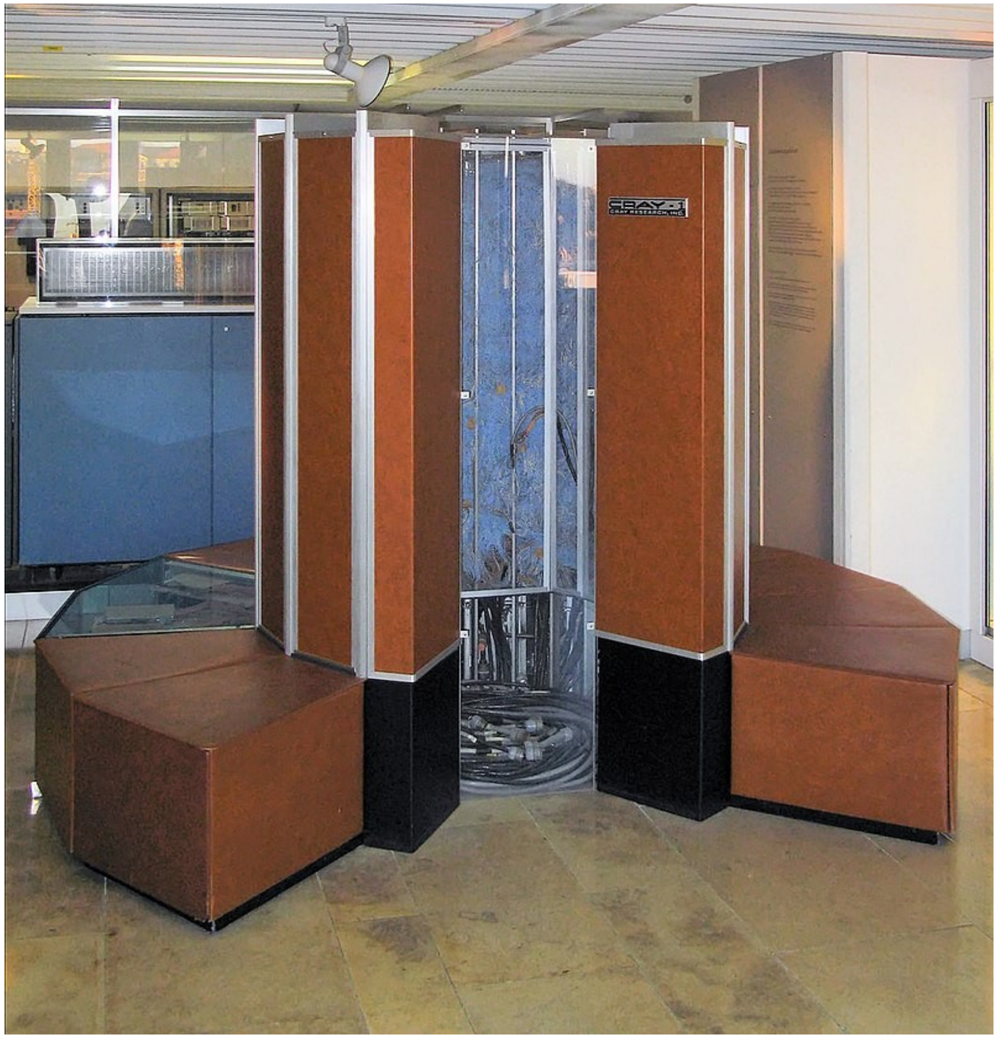
Vector Supercomputers

Epitomized by Cray-1, 1976:

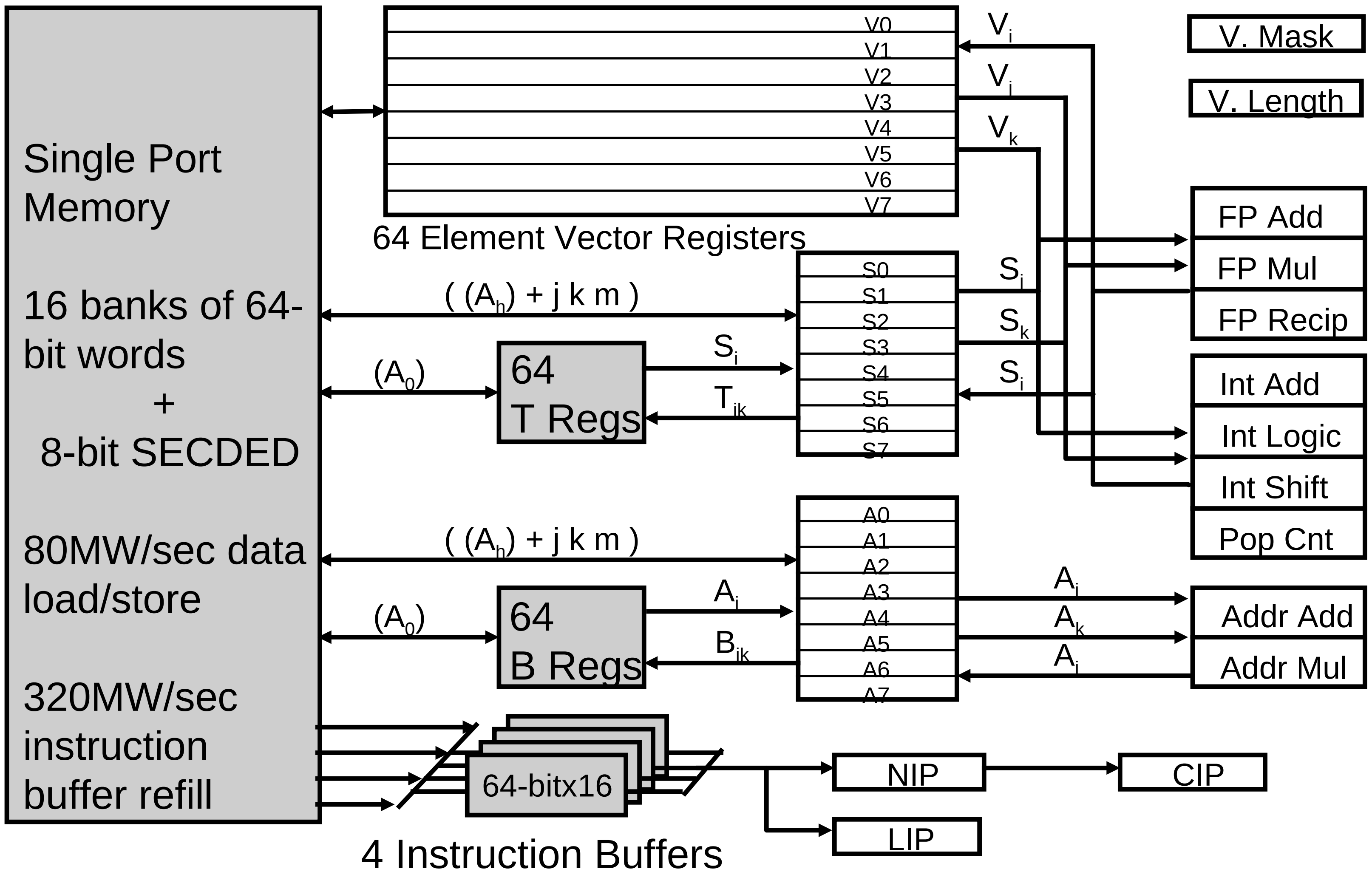
Scalar Unit + Vector Extensions

- 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)

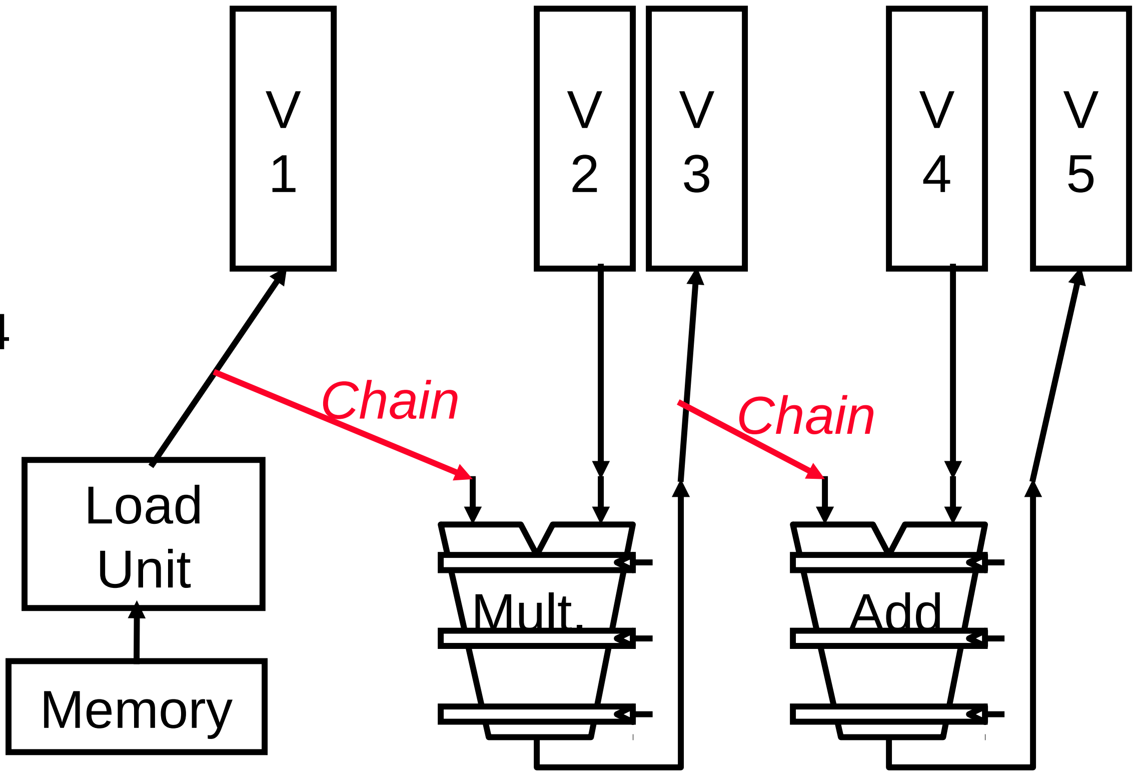


memory bank cycle 50 ns   *processor cycle* 12.5 ns (80MHz)

Vector Chaining

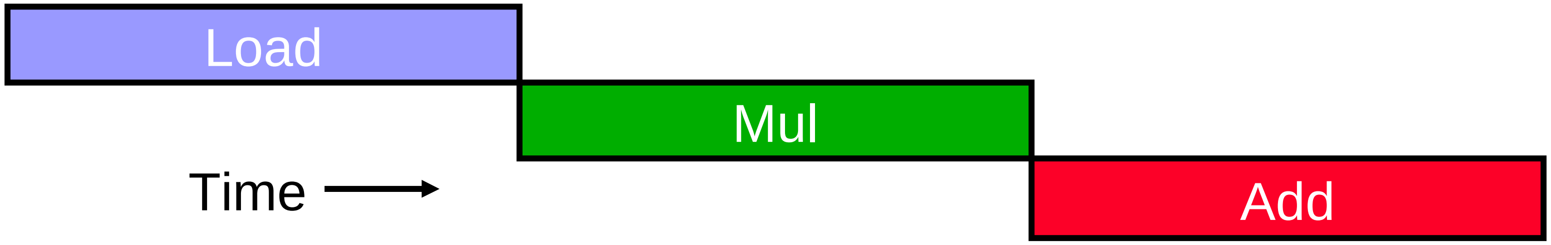
- Vector version of register bypassing
  - introduced with Cray-1

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

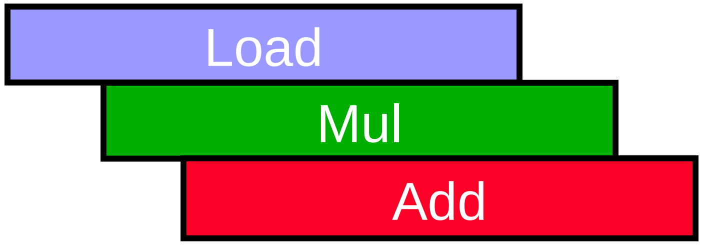


Vector Chaining Advantage

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



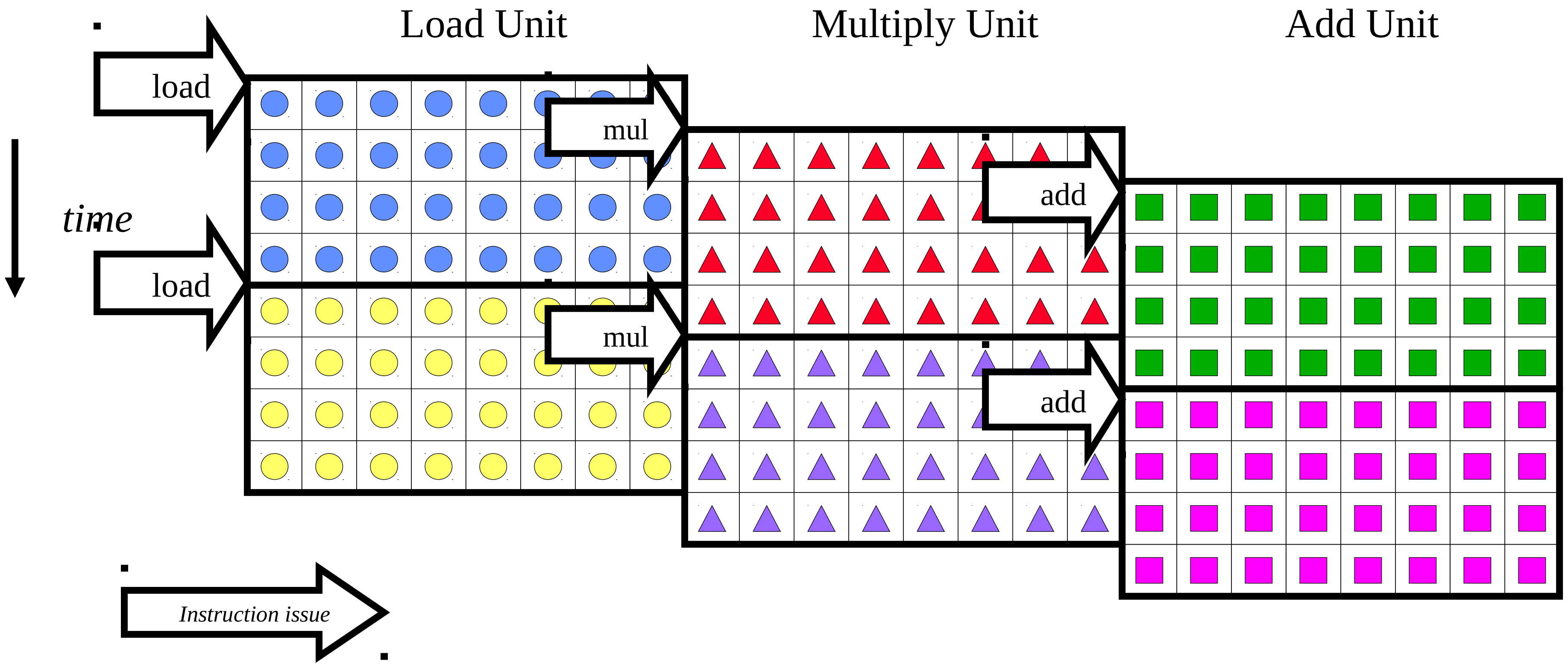
- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

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

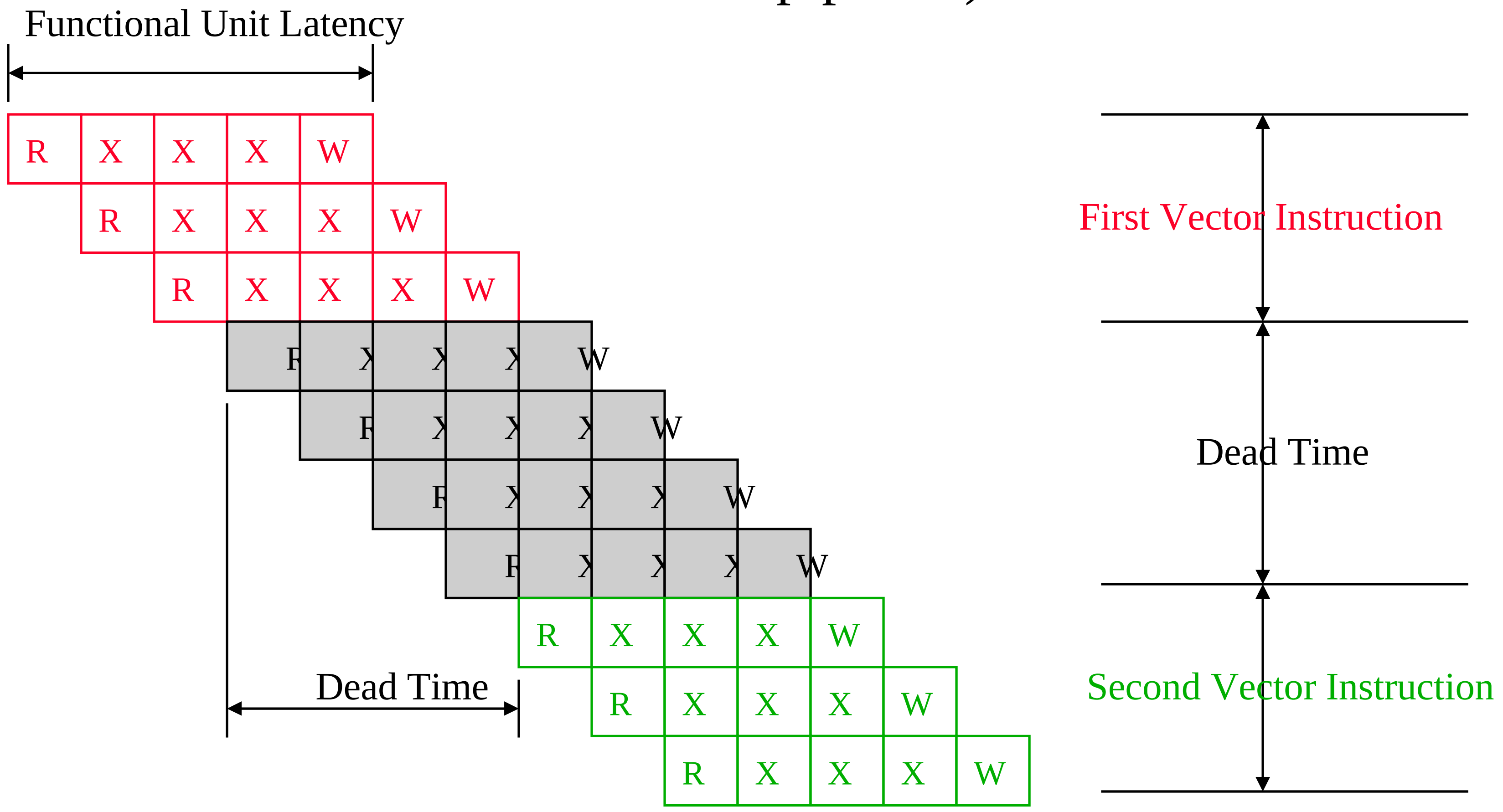


Complete 24 operations/cycle while issuing 1 short instruction/cycle

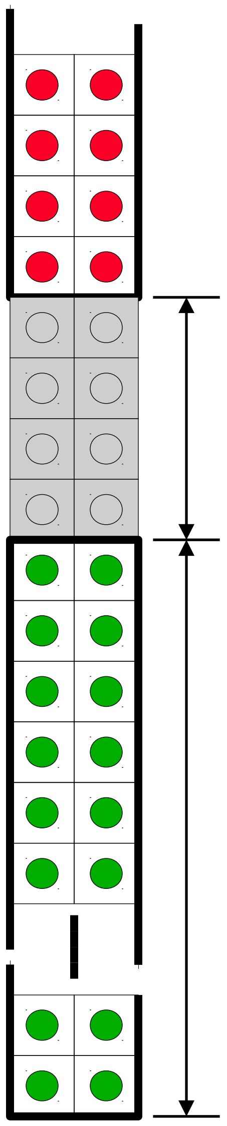
Vector Startup

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

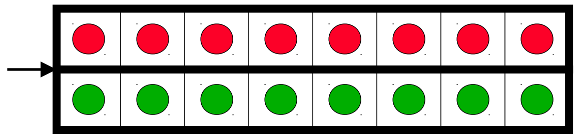
64 cycles active

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]]++;
```

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

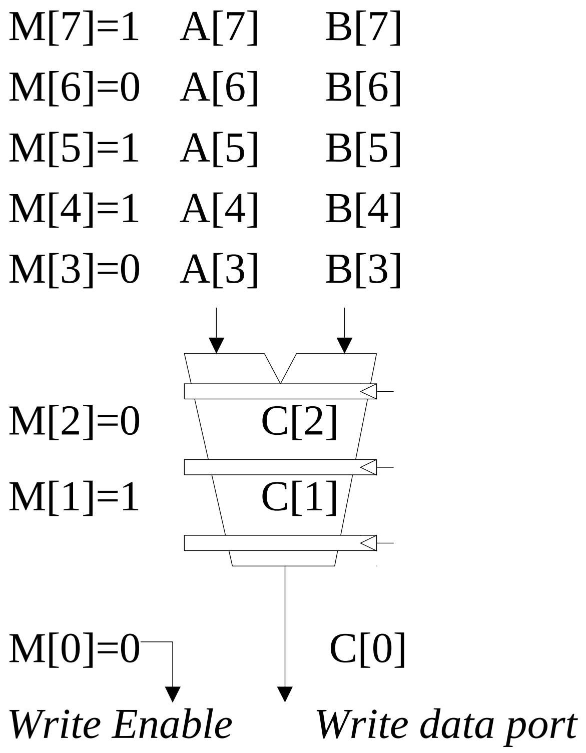
Code example:

```
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

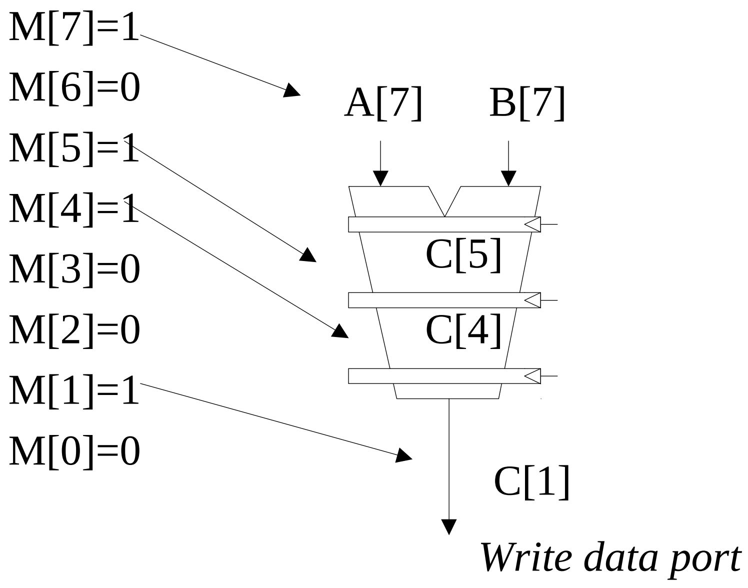
Simple Implementation

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



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)
```