CSE 520
Computer Architecture II

Complex Pipelining: SIMD and Vector Processors

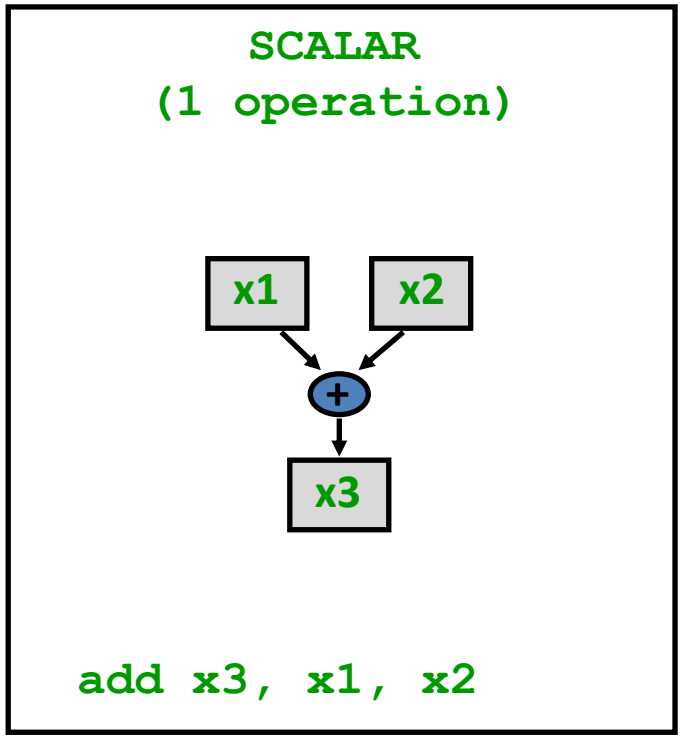
Prof. Michel A. Kinsy

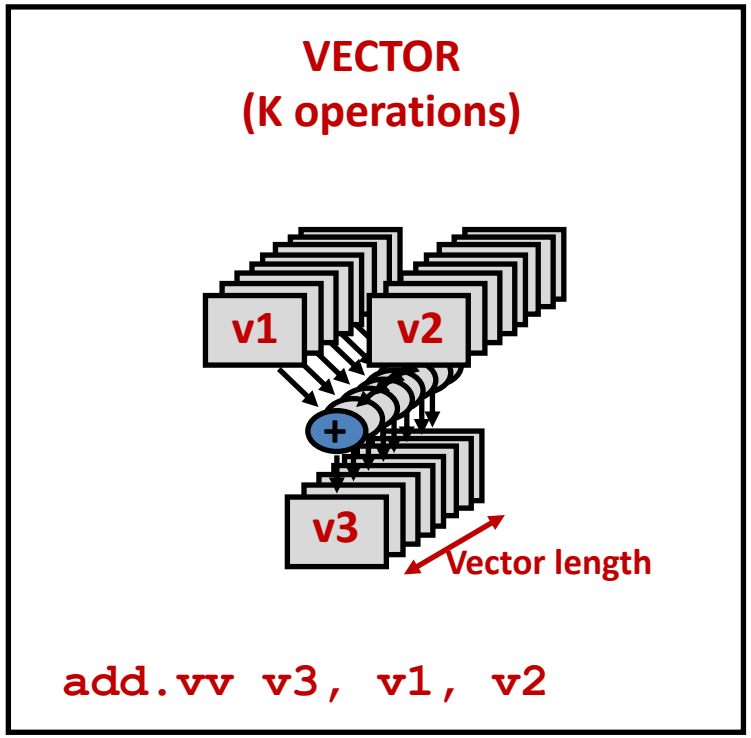
Vector Architectures

- Basic idea
  - Read sets of data elements into "vector registers"
  - Operate on those registers
  - Disperse the results back into memory
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth
- Scalar Unit
  - Load/Store Architecture
- Vector Extension
  - Vector Registers
  - Vector Instructions

Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers
  - Called “vectors"

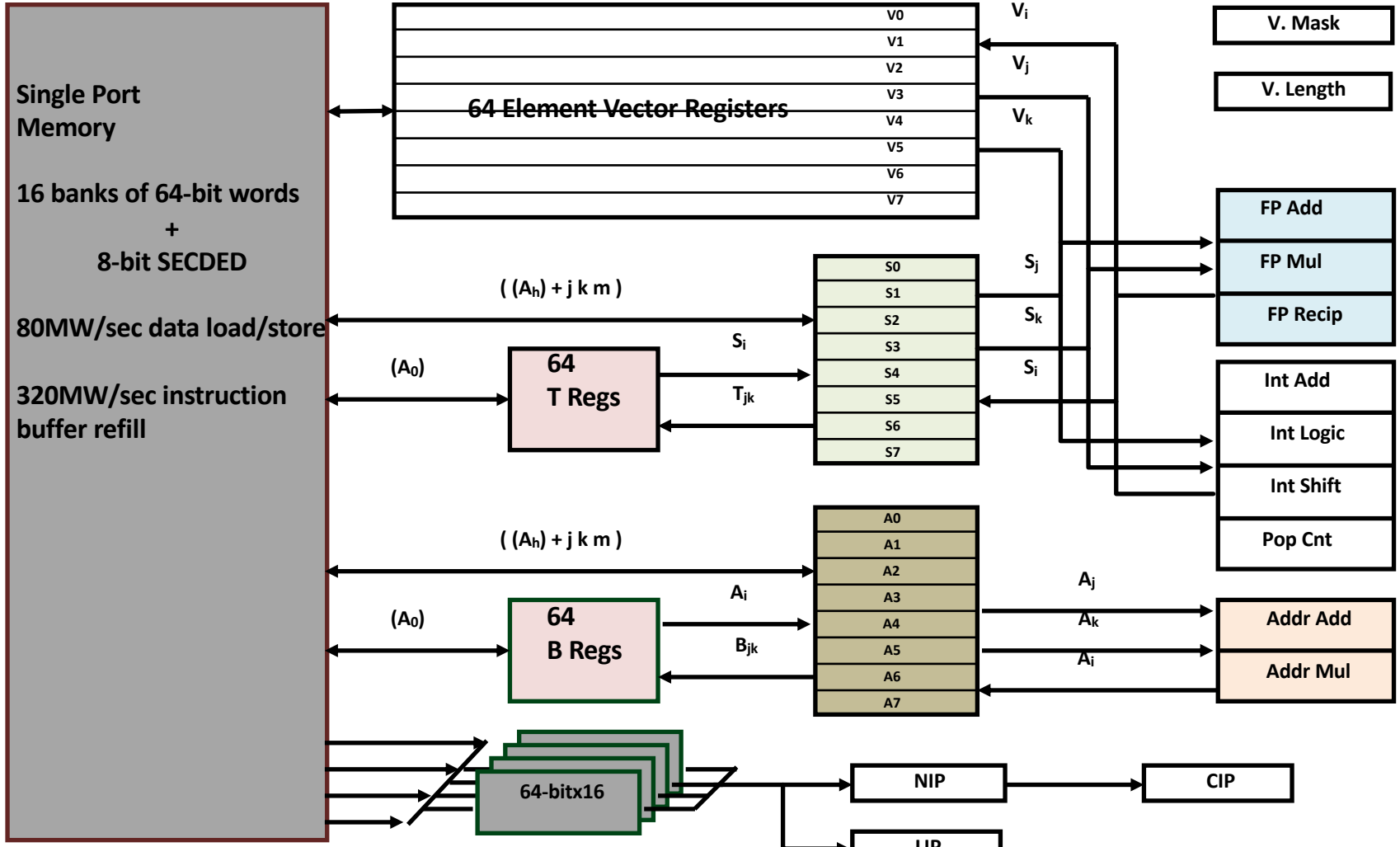




Vector Architectures

- Epitomized by Cray-1, 1976
- Types of vector processors
  - Memory-memory processors: all vector operations are memory-to-memory (CDC)
  - Vector-register processors: all vector operations except load and store are among the vector registers
    - (CRAY-1, CRAY-2, X-MP, Y-MP, NEX SX/2(3), Fujitsu)
- Implementation
  - Hardwired Control
  - Highly Pipelined Functional Units
  - Interleaved Memory System
  - No Data Caches
  - No Virtual Memory

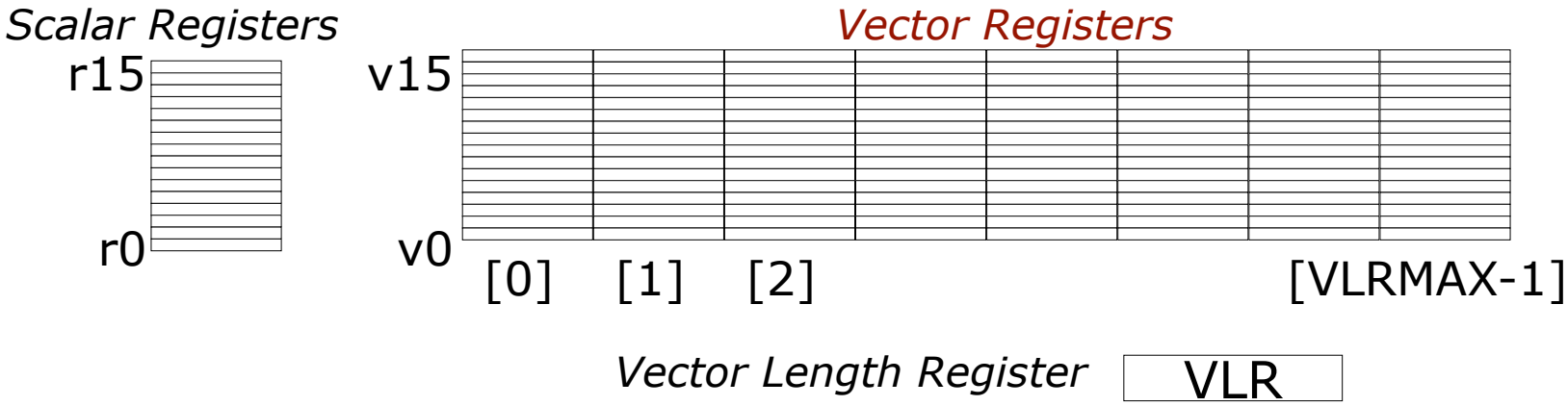
Cray-1 (1976)


Single Port Memory
16 banks of 64-bit words
+
8-bit SECDED
80MW/sec data load/store
320MW/sec instruction buffer refill
64 Element Vector Registers
V0
V1
V2
V3
V4
V5
V6
V7
Vi
Vj
Vk
V. Mask
V. Length
FP Add
FP Mul
FP Recip
Sj
Sk
Si
S0
S1
S2
S3
S4
S5
S6
S7
((Ah) + j k m)
(A0)
64
T Regs
Si
Tjk
Int Add
Int Logic
Int Shift
Pop Cnt
A0
A1
A2
A3
A4
A5
A6
A7
((Ah) + j k m)
(A0)
64
B Regs
Ai
Bjk
Aj
Ak
Ai
Addr Add
Addr Mul
64-bitx16
NIP
CIP
LIP
memory bank cycle 50 ns
4 Instruction Buffers
processor cycle 12.5 ns (80MHz)

Properties of Vector Processors

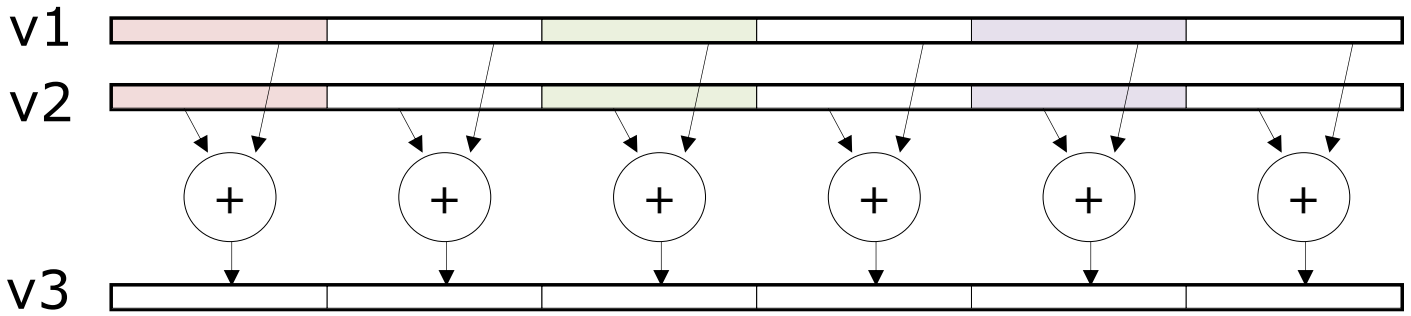
- Each result independent of previous result
  - Long pipeline, compiler ensures no dependencies
  - High clock rate
- Vector instructions access memory with known pattern
  - Highly interleaved memory
  - Amortize memory latency of over 64 elements
  - No data caches required
  - Only use instruction cache
- Reduces branches and branch problems in pipelines
  - Single vector instruction implies lots of work (loop)
  - Fewer instruction fetches

Vector Programming Model

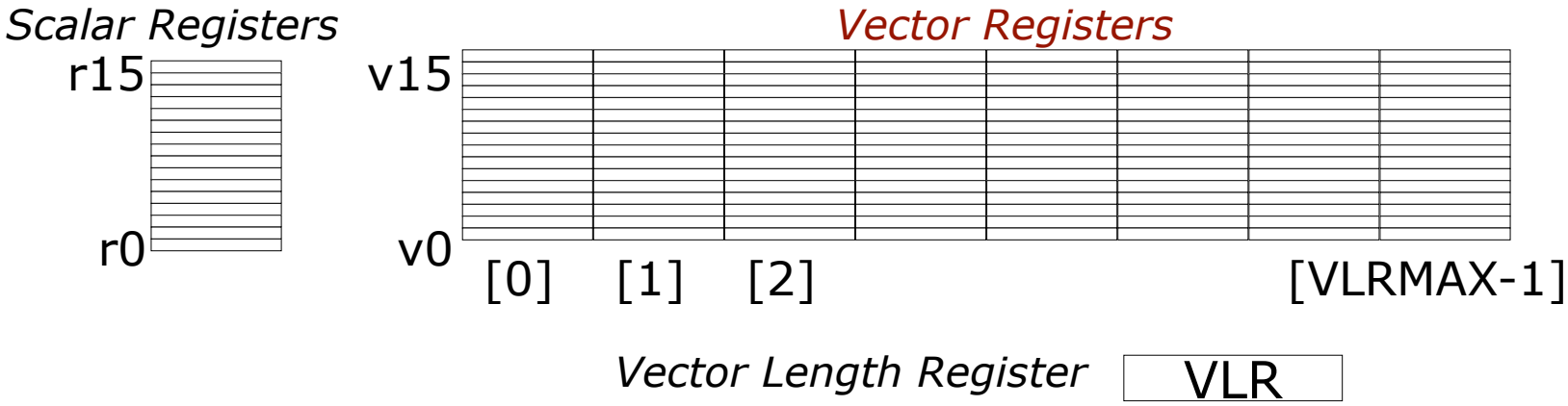


Vector Load and Store Instructions

LV v1, r1, r2

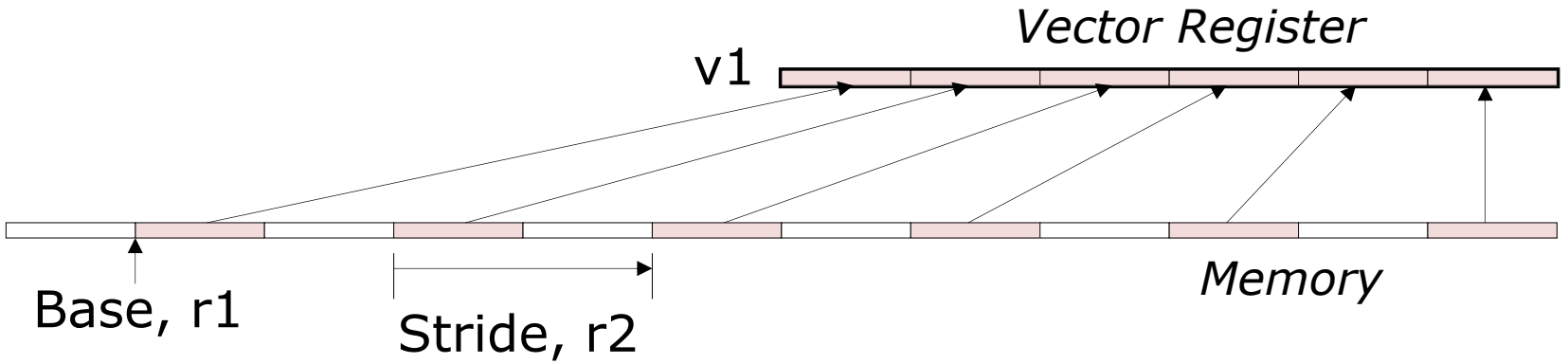


Vector Programming Model



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, 4
  DADDIU    R2, 4
  DADDIU    R3, 4
  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

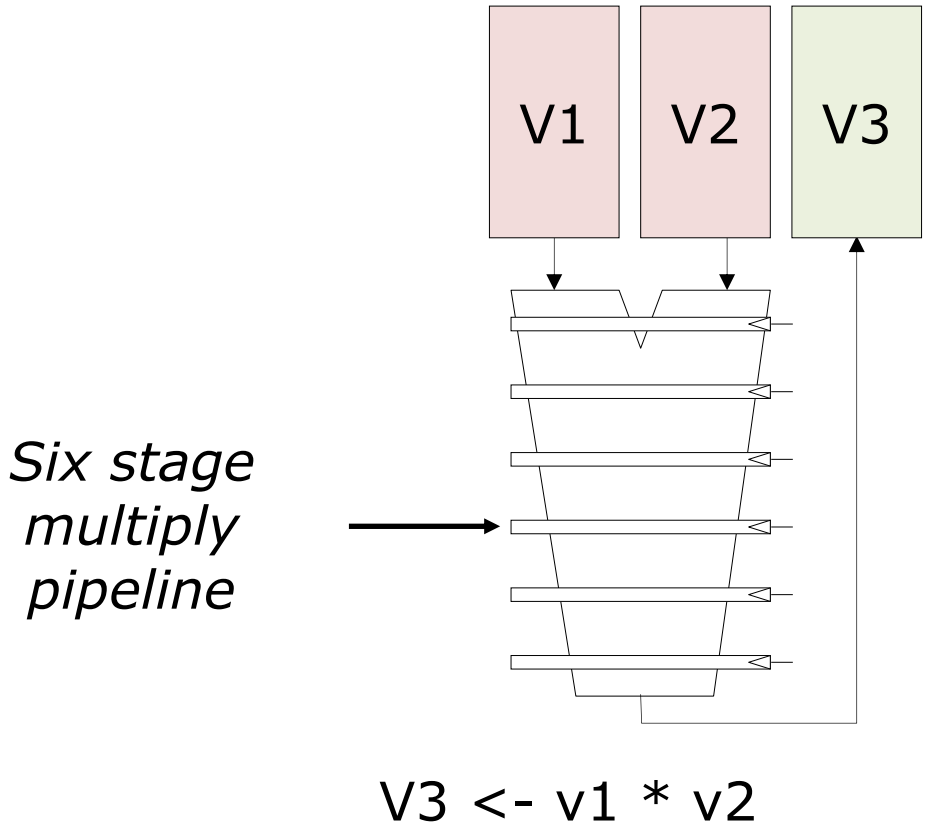
- Compact
  - one short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions

Vector Instruction Set Advantages

- Compact
  - One short instruction encodes N operations
- Expressive, tells hardware that these N operations:
  - 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

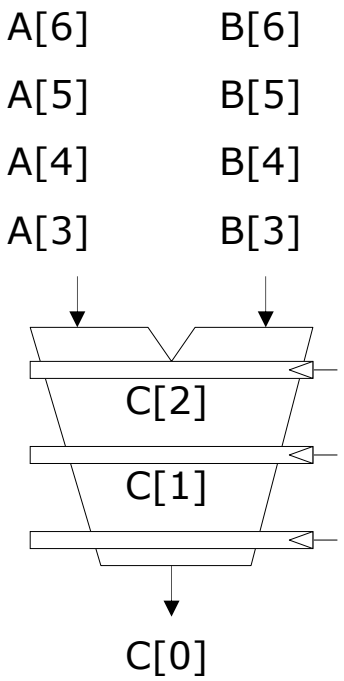
- Use deep pipeline to execute element operations
  - Fast clock
- Simplifies control of deep pipeline because elements in vector are independent
  - No hazards!



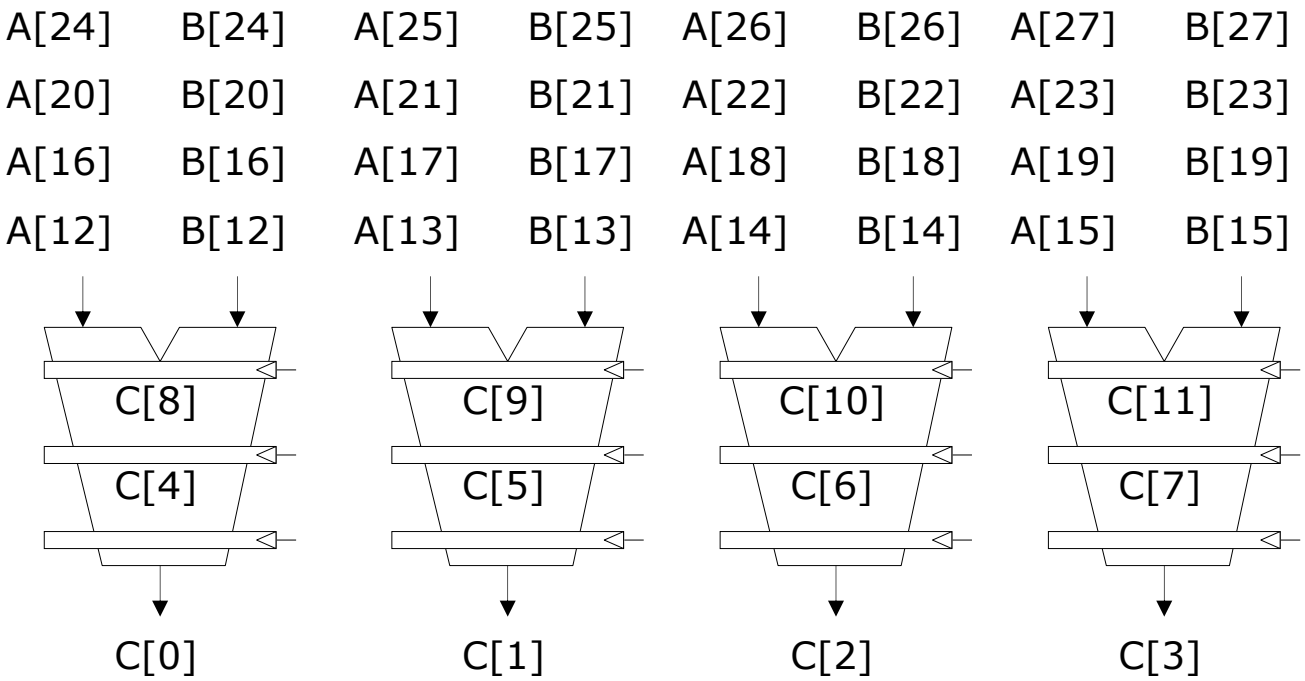
Vector Instruction Execution

- ADDV C,A,B

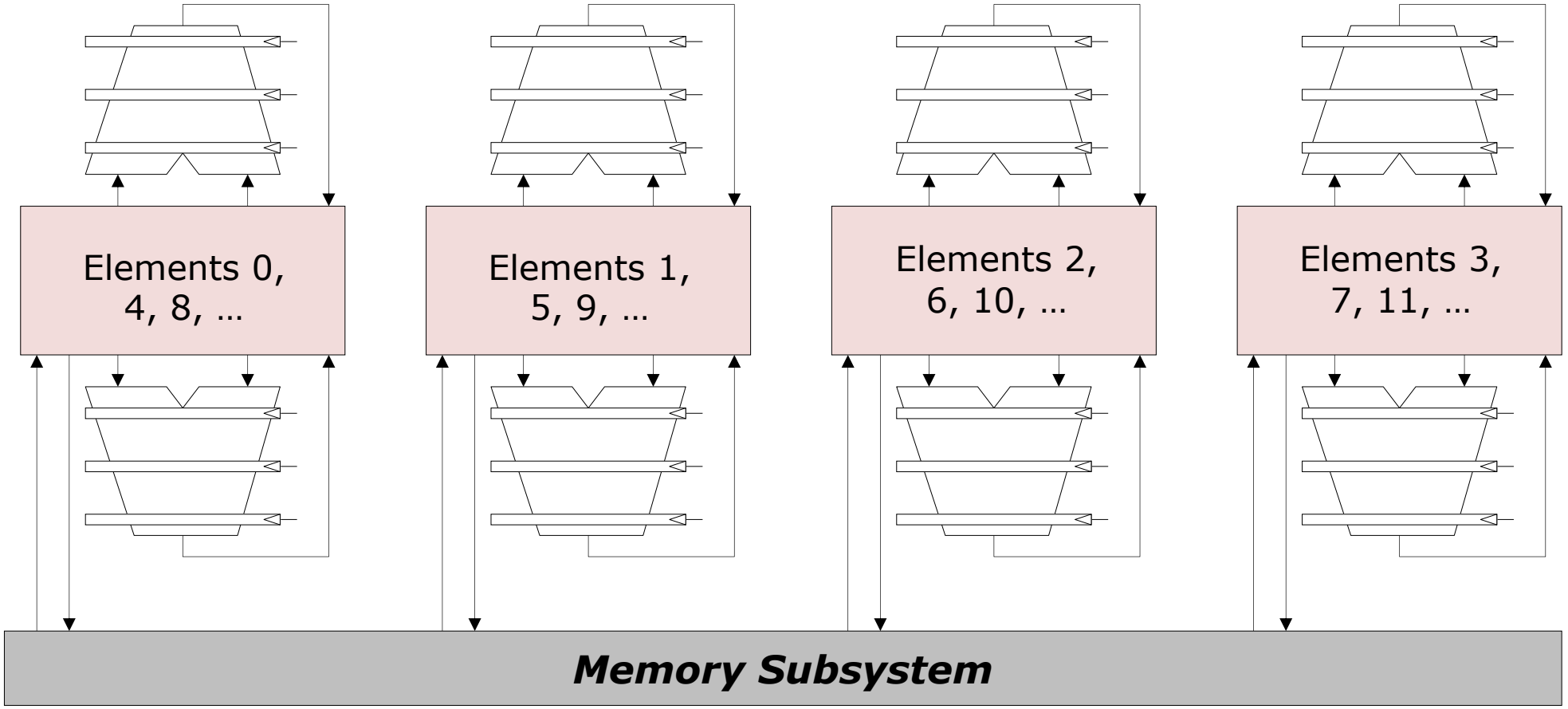
Execution using one pipelined functional unit



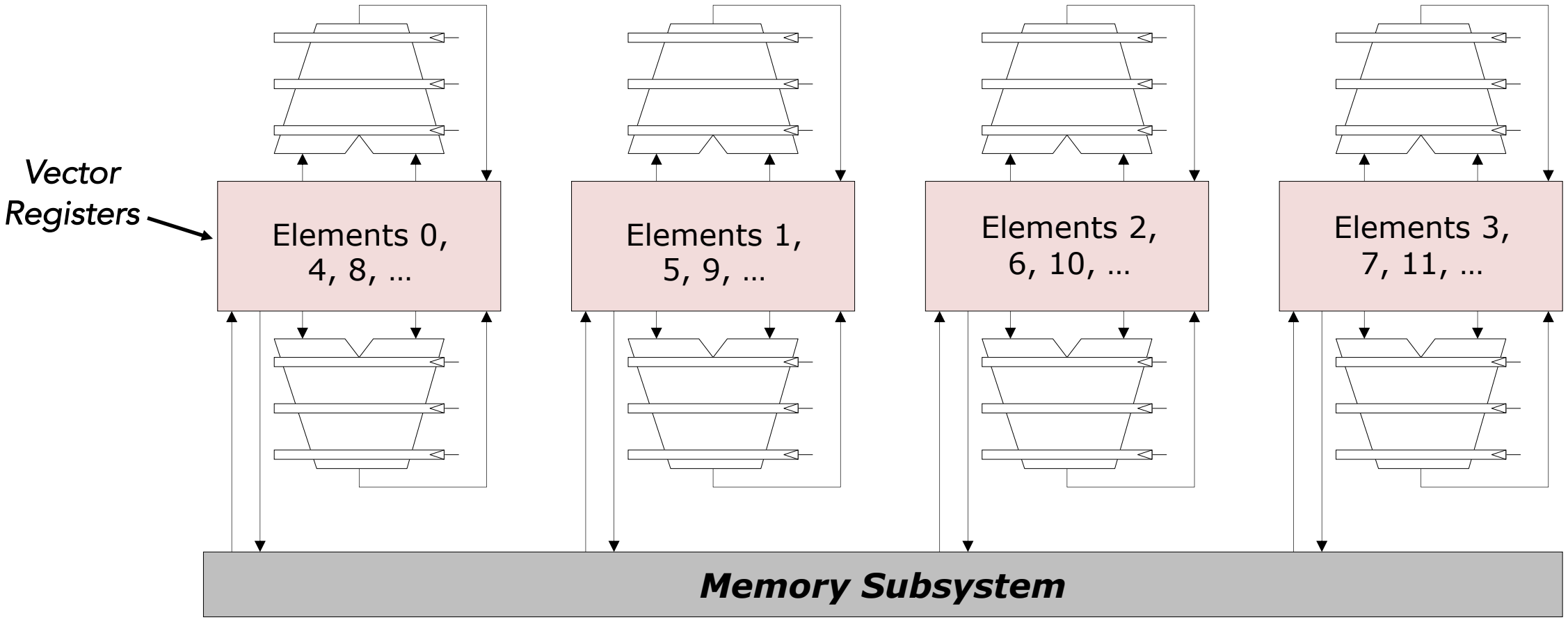
Execution using four pipelined functional units



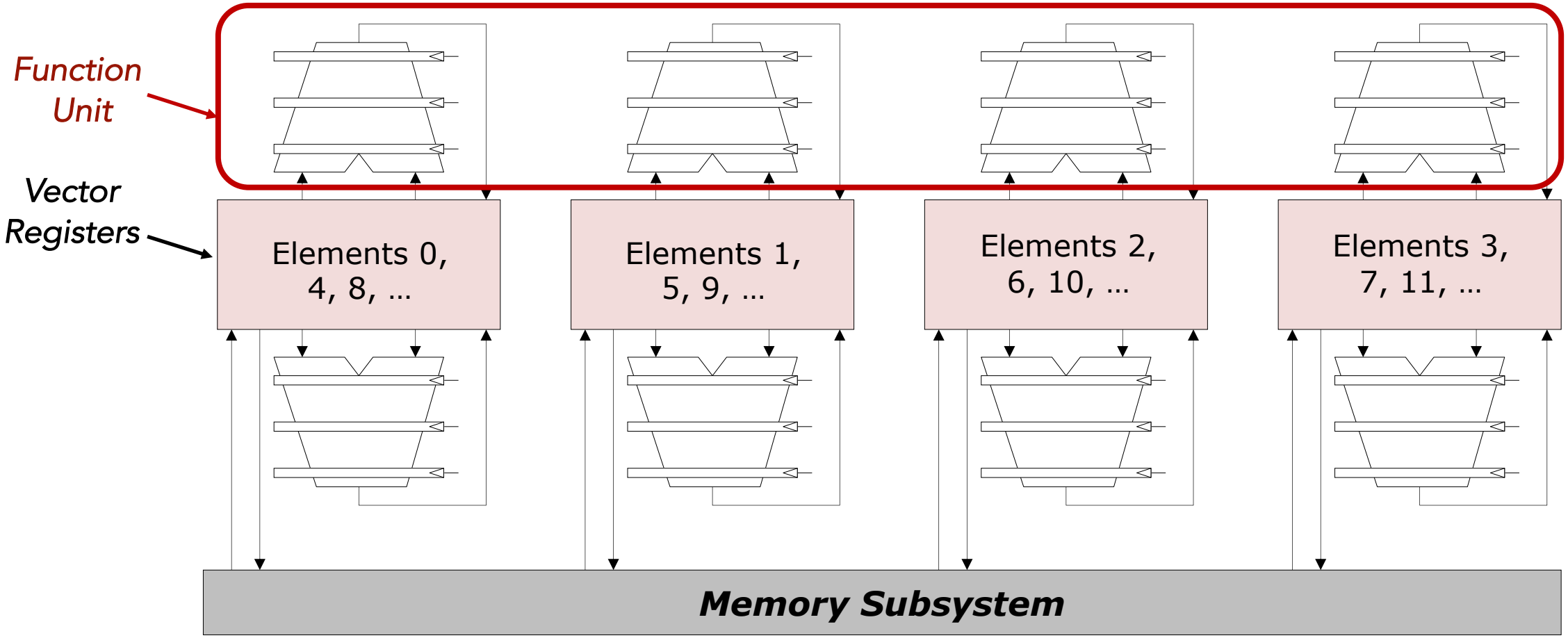
Vector Unit Structure



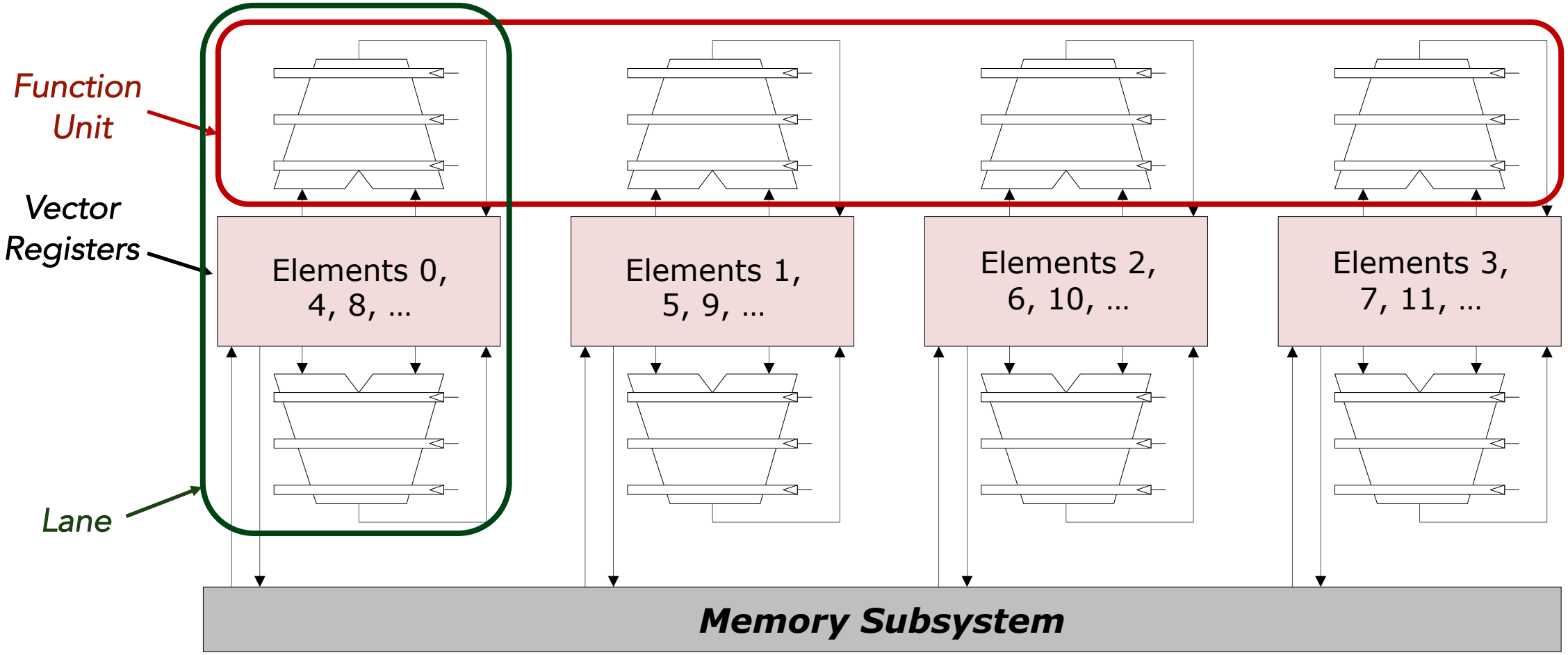
Vector Unit Structure



Vector Unit Structure



Vector Unit Structure

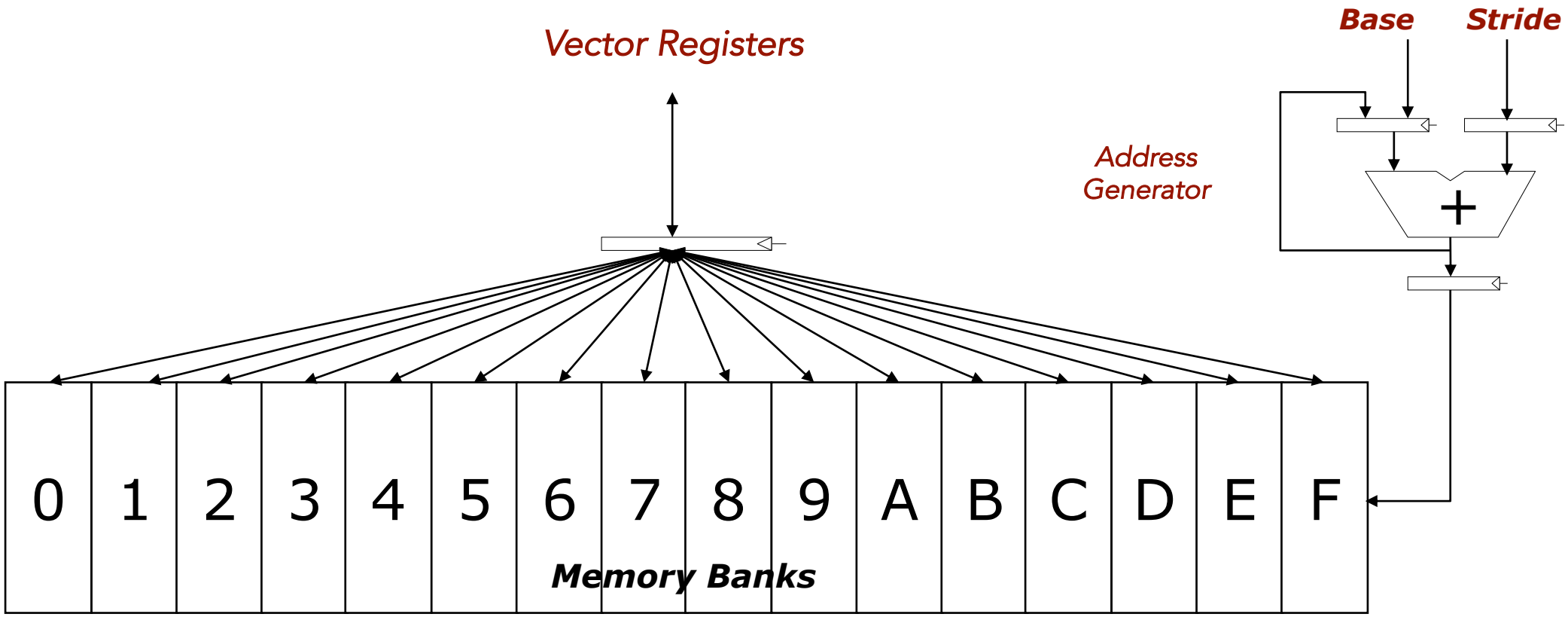


Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory
- Example
  - 32 processors, each generating 4 loads and 2 stores/cycle
  - Processor cycle time is 2.167 ns, SRAM cycle time is 15 ns
  - How many memory banks needed?
    - 32x6=192 accesses,
    - 15/2.167≈7 processor cycles
    - →1344!

Vector Memory System

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



Stride

- Consider

```
for (i = 0; i < 100; i=i+1) {
        for (j = 0; j < 100; j=j+1) {
            A[i][j] = 0.0;
            for (k = 0; k < 100; k=k+1)
            A[i][j] = A[i][j] + B[i][k] * D[k][j];
        }
}
```

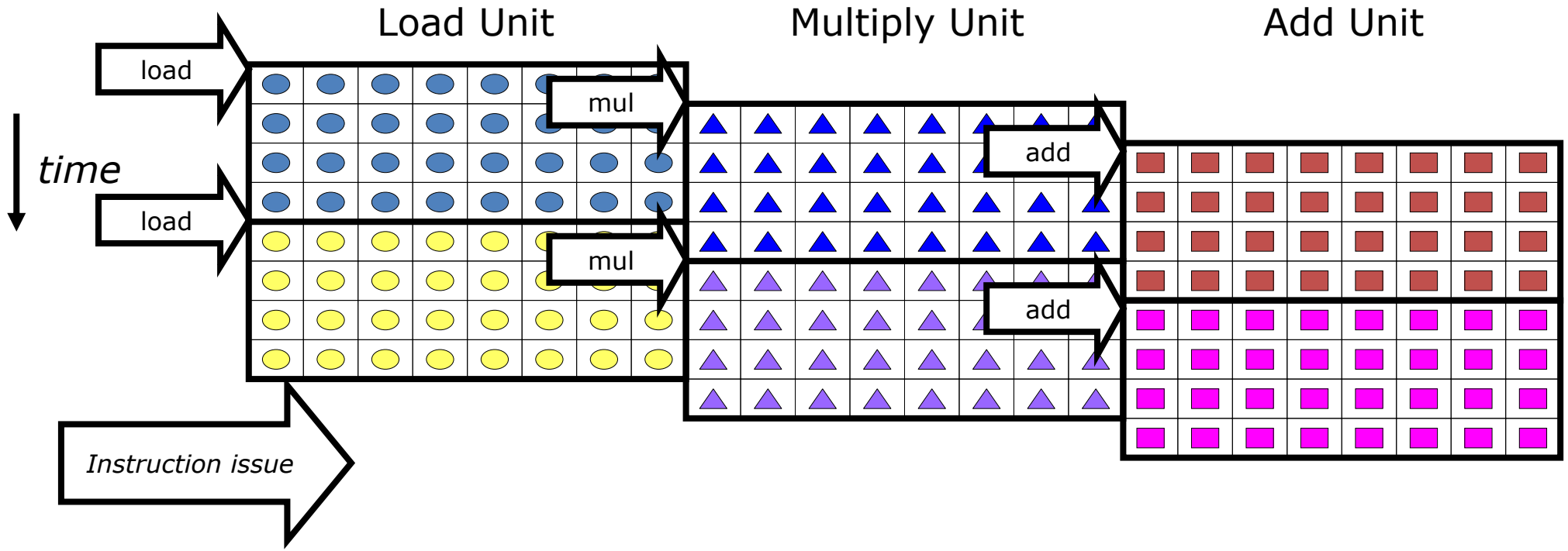
- Must vectorize multiplication of rows of B with columns of D
- Use *non-unit stride*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride, #banks) < bank busy time (in # of cycles)

Stride

- Example
- 8 memory banks with a bank busy time of 6 cycles and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?
- Answer
  - Stride of 1: number of banks is greater than the bank busy time, so it takes
    - 12+64 = 76 clock cycles → 1.2 cycle per element
  - Stride of 32: the worst-case scenario happens when the stride value is a multiple of the number of banks, which this is! Every access to memory will collide with the previous one! Thus, the total time will be:
    - 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element!

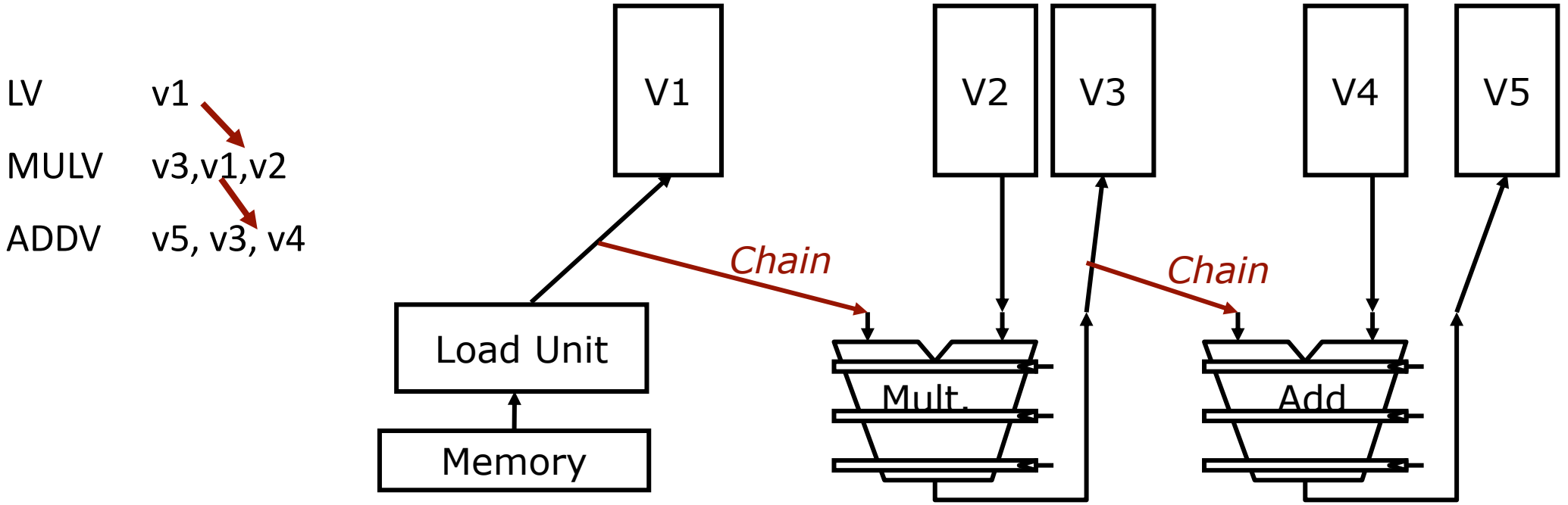
Vector Instruction Parallelism

- 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



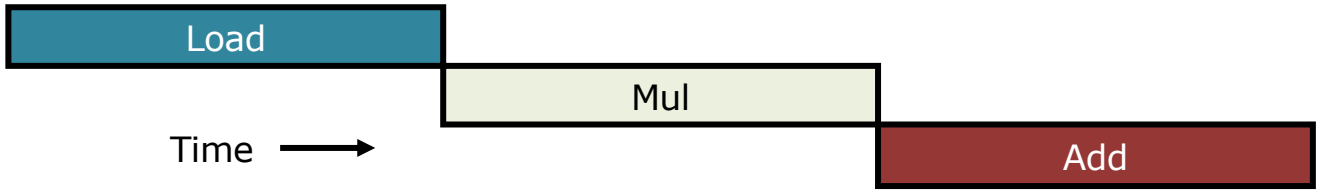
Vector Chaining or Bypassing

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

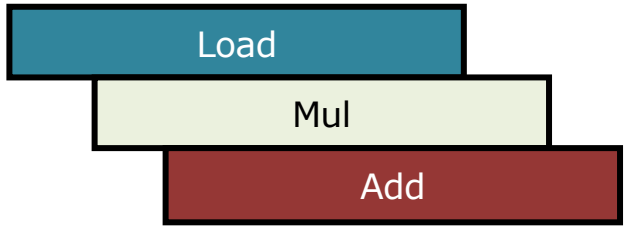


Vector Chaining Advantage

- 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 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
  - 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
```

Vector Processor

- Vector Register: fixed length bank holding a single vector
  - Has at least 2 read and 1 write ports
  - Typically, 8-32 vector registers, each holding 64-128 64-bit elements
- Vector Functional Units (FUs): fully pipelined, start new operation every clock
  - Typically, 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- Vector Load-Store Units (LSUs): fully pipelined unit to load or store a vector; may have multiple LSUs
- Scalar registers: single element for FP scalar or address
- Cross-bar to connect FUs , LSUs, registers

Modern Vector: NEC SX-6 (2003)

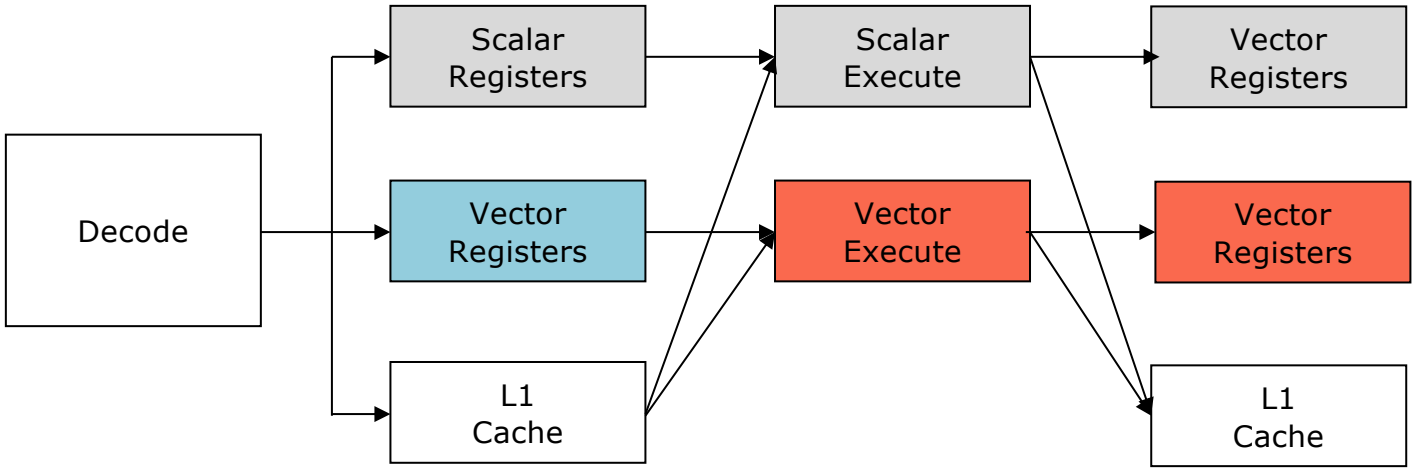
- CMOS Technology
  - 500 MHz CPU, fits on single chip
  - SDRAM main memory (up to 64GB)
- Scalar unit
  - 4-way superscalar, with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- SMP structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB/s shared memory bandwidth (4096 interleaved banks)

Modern Vector: NEC SX-6 (2003)

- 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

Larrabee x86 with vectors

- Short in-order instruction pipeline
- Separate scalar and vector units and register sets
  - Vector unit: 16 32-bit ops/clock
- Fast access to L1 cache
- L1 connects to core's portion of the L2 cache



Larrabee Vector Architecture

- Data types
  - Int32, Float32 and Float64 data
- Vector operations
  - Two input/one output operations
  - Full complement of arithmetic and media operations
    - Fused multiply-add (three input arguments)
  - Mask registers select lanes to write
  - Swizzle the vector elements on register read

Larrabee Vector Architecture

- Memory access
  - Vector load/store including scatter/gather
  - Data replication on read from memory
  - Numeric type conversion on memory read

Larrabee Motivation

- Design experiment: not a real 10-core chip!

# CPU cores	2 out of order	10 in-order
Instructions per issue	4 per clock	2 per clock
VPU lanes per core	4-wide SSE	16-wide
L2 cache size	4 MB	4 MB
Single-stream	4 per clock	2 per clock
Vector throughput	8 per clock	160 per clock

- 20 times the multiply-add operations per clock

Data in chart taken from Seiler, L., Carmean, D., et al. 2008. Larrabee: A many-core x86 architecture for visual computing.

Some Applications

- Multimedia Processing (compress., graphics, audio synth, image proc.)
- Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)
- Lossy Compression (JPEG, MPEG video and audio)
- Lossless Compression (Zero removal, RLE, Differencing, LZW)
- Cryptography (RSA, DES/IDEA, SHA/MD5)
- Speech and handwriting recognition
- Operating systems/Networking (`memcpy`, `memset`, parity, checksum)
- Databases (hash/join, data mining, image/video serving)
- Language run-time support (stdlib, garbage collection)

Next Class

- Hardware Multithreading