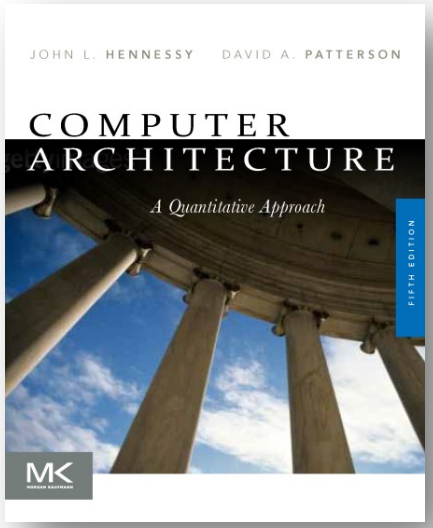


ESE 545 Computer Architecture

Data-Level Parallelism (DLP) and Vector Processing

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 ≡ ***Vector Machine***

Vector Supercomputers

Epitomized by Cray-1, 1976:

Scalar Unit + Vector Extensions

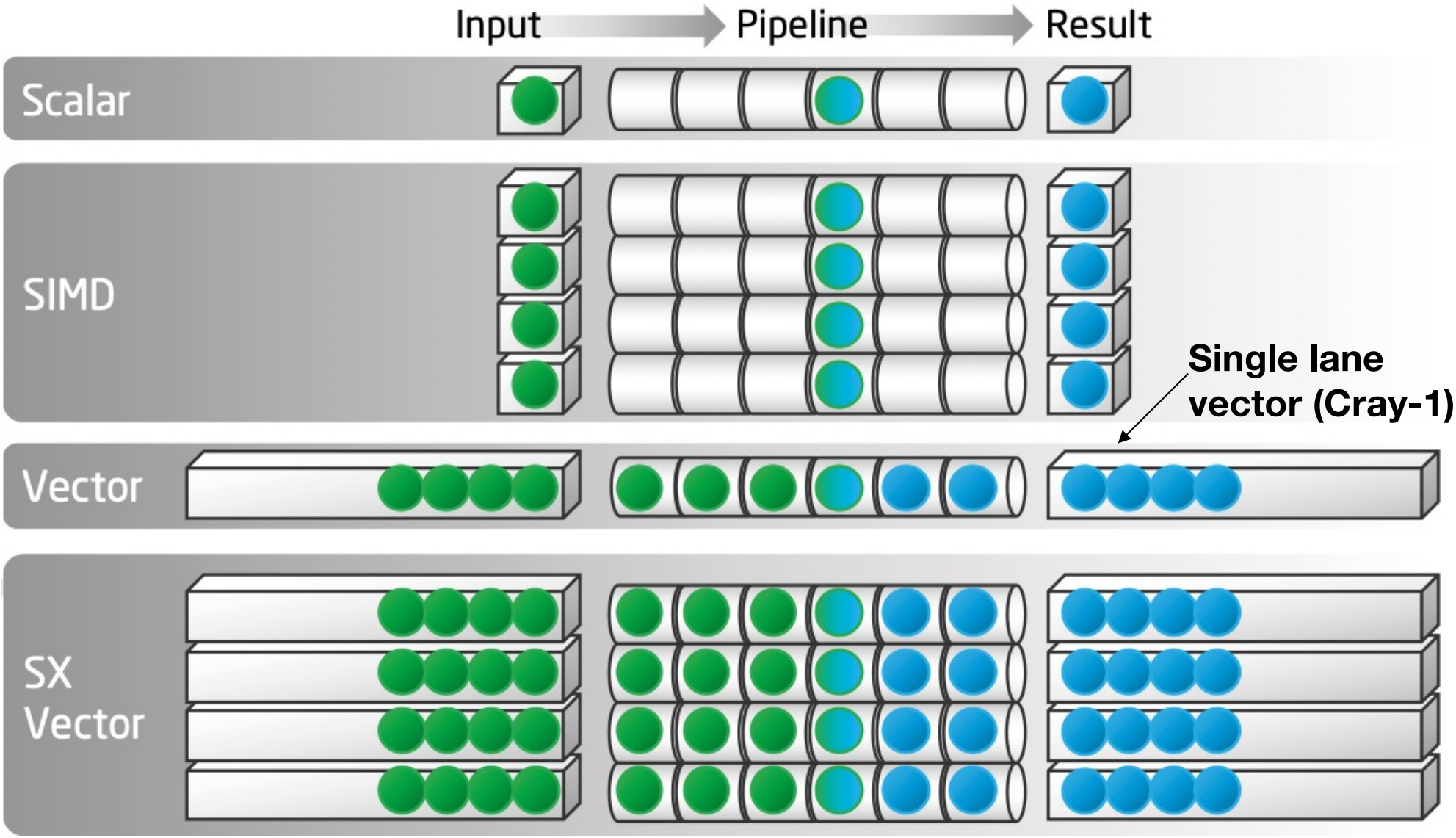
- Load/Store Architecture
- Vector Registers
- Vector Instructions
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

Cray-1 (1976)

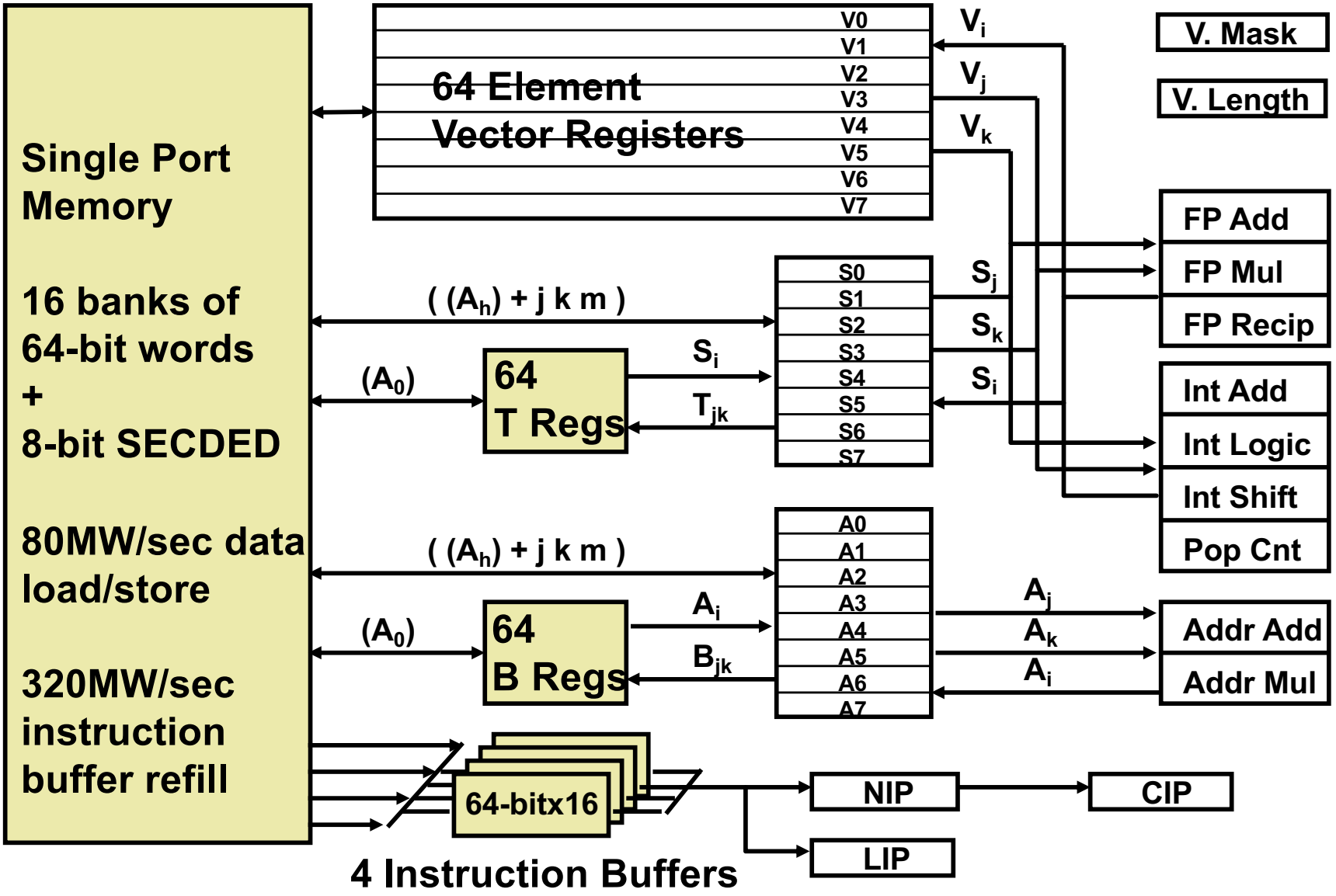
NEC SX Vector Supercomputers

Model	SX-1/2	SX-3	SX-4	SX-5	SX-6	SX-7	SX-8/8R	SX-9	SX-ACE	SX-Aurora TSUBASA	SX-Aurora TSUBASA Gen2
Year	1983	1989	1994	1998	2001	2002	2004	2007	2013	2018	2020
Technology	Bipolar	Bipolar	350 nm	250 nm	150 nm	150 nm	90 nm	65 nm	28 nm	16 nm	16 nm
CPU frequency	166 MHz	340 MHz	125 MHz	250 MHz	500 MHz	552 MHz	1.0 GHz	3.2 GHz	1.0 GHz	~1.6 GHz	~1.6 GHz
CPU performance	1.3 GF	5.5 GF	2.0 GF	8.0 GF	8.0 GF	8.8 GF	16.0 GF	102.4 GF	256.0 GF	~2.45 TF	~3.07 TF
CPU Memory bandwidth	10.7 GB/s	12.8 GB/s	16.0 GB/s	64.0 GB/s	32.0 GB/s	35.3 GB/s	64.0 GB/s	256.0 GB/s	256.0 GB/s	~1.22TB/s	~1.53TB/s

Basic Idea of Vector Processing

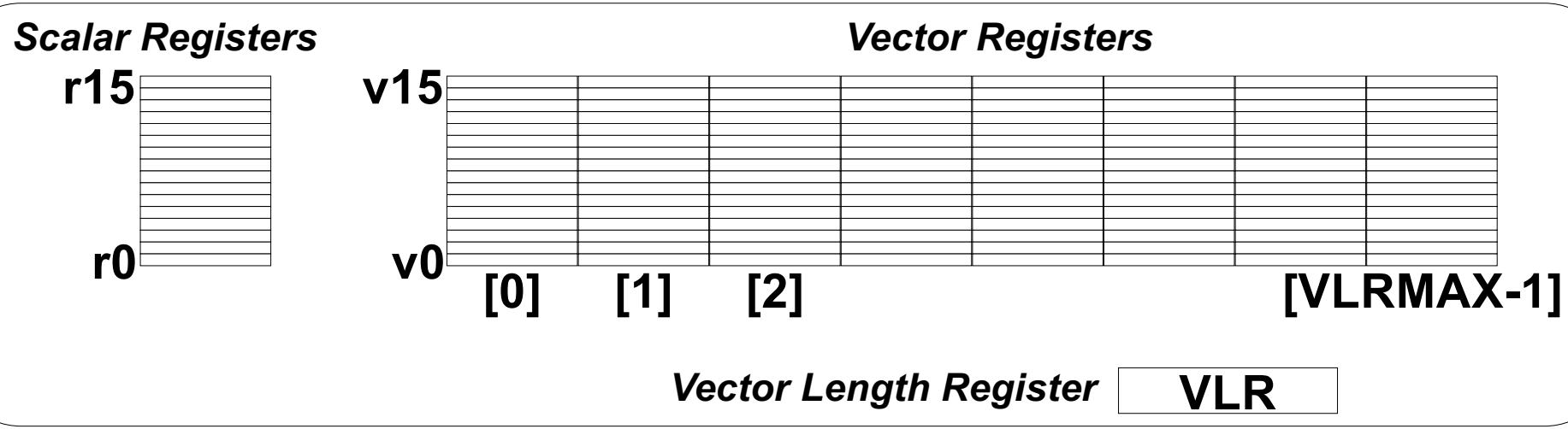


Cray-1 (1976)



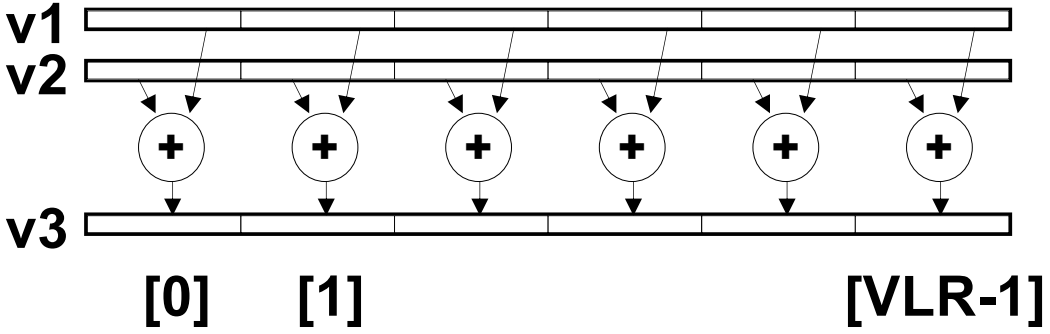
memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Vector Programming Model



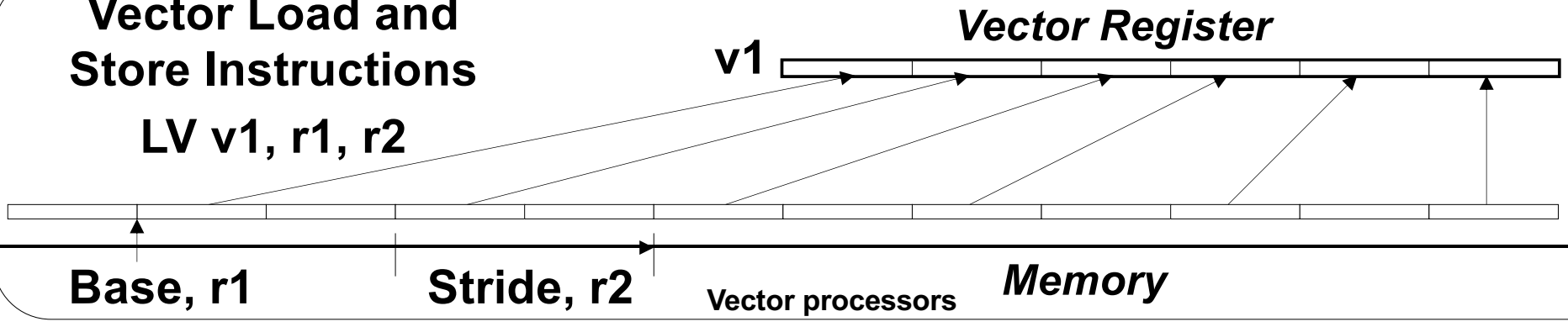
Vector Arithmetic Instructions

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
```

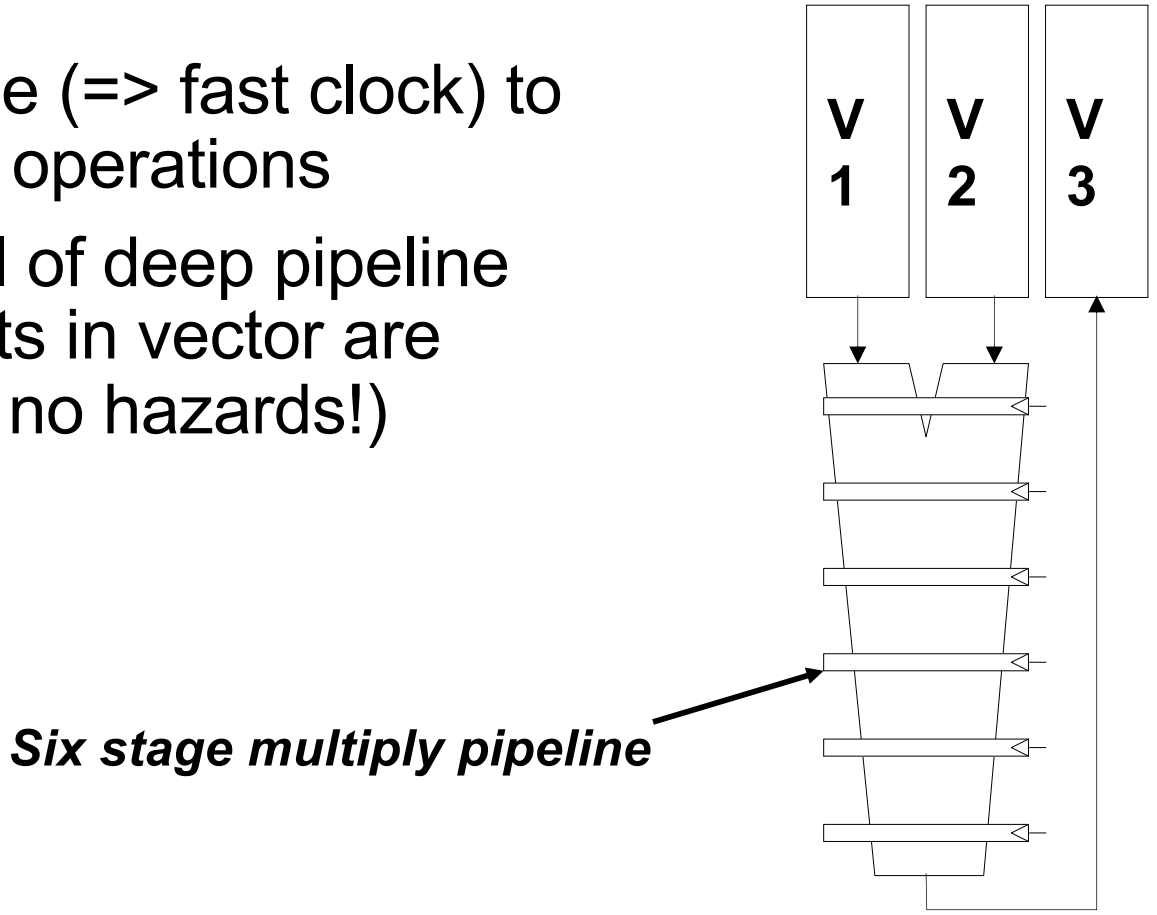
```
# Vector Code
  LI VLR, 64
  LV V1, R1
  LV V2, R2
  ADDV.D V3, V1, V2
  SV V3, R3
```

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*

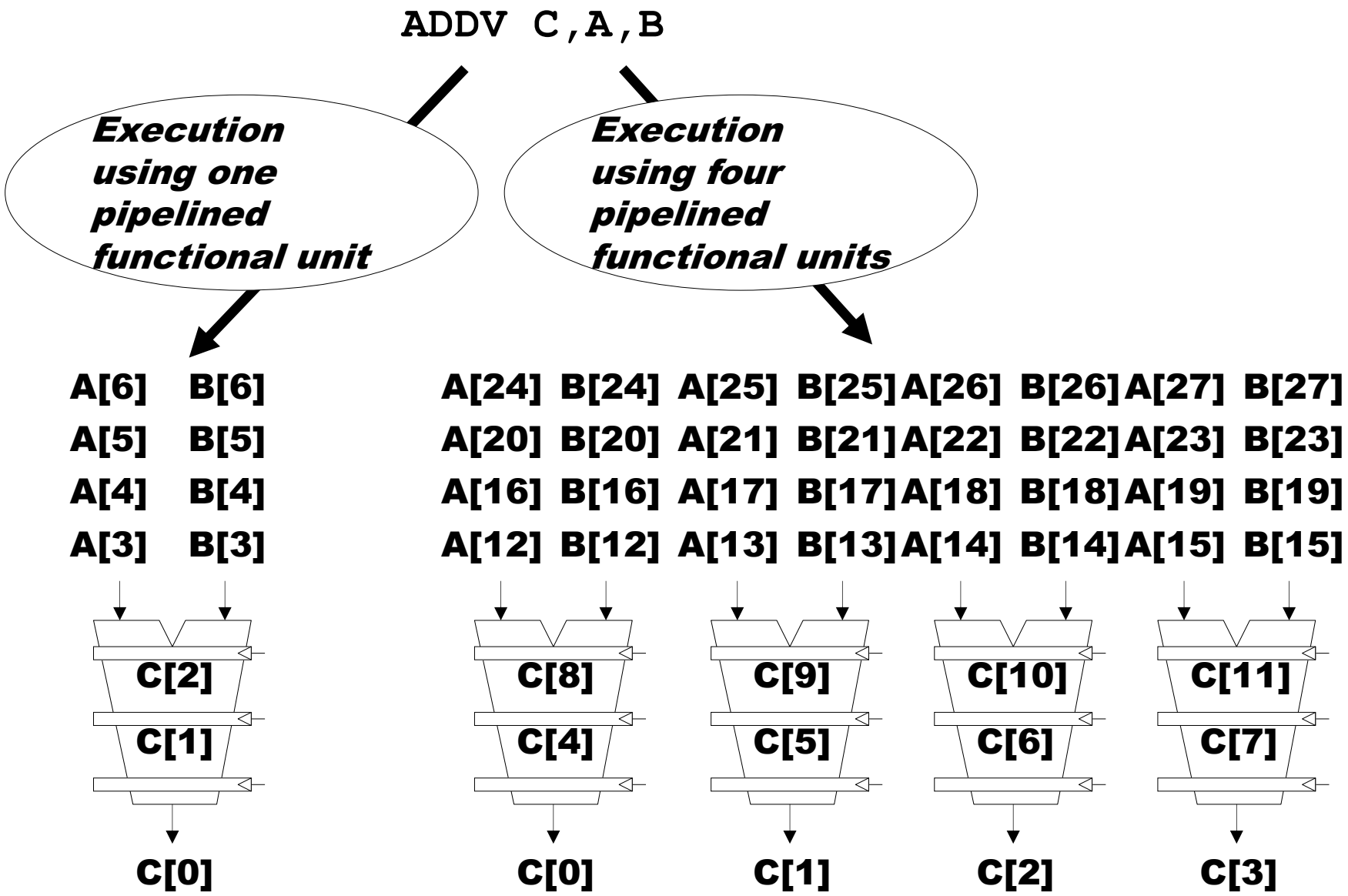
Vector Arithmetic Execution

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



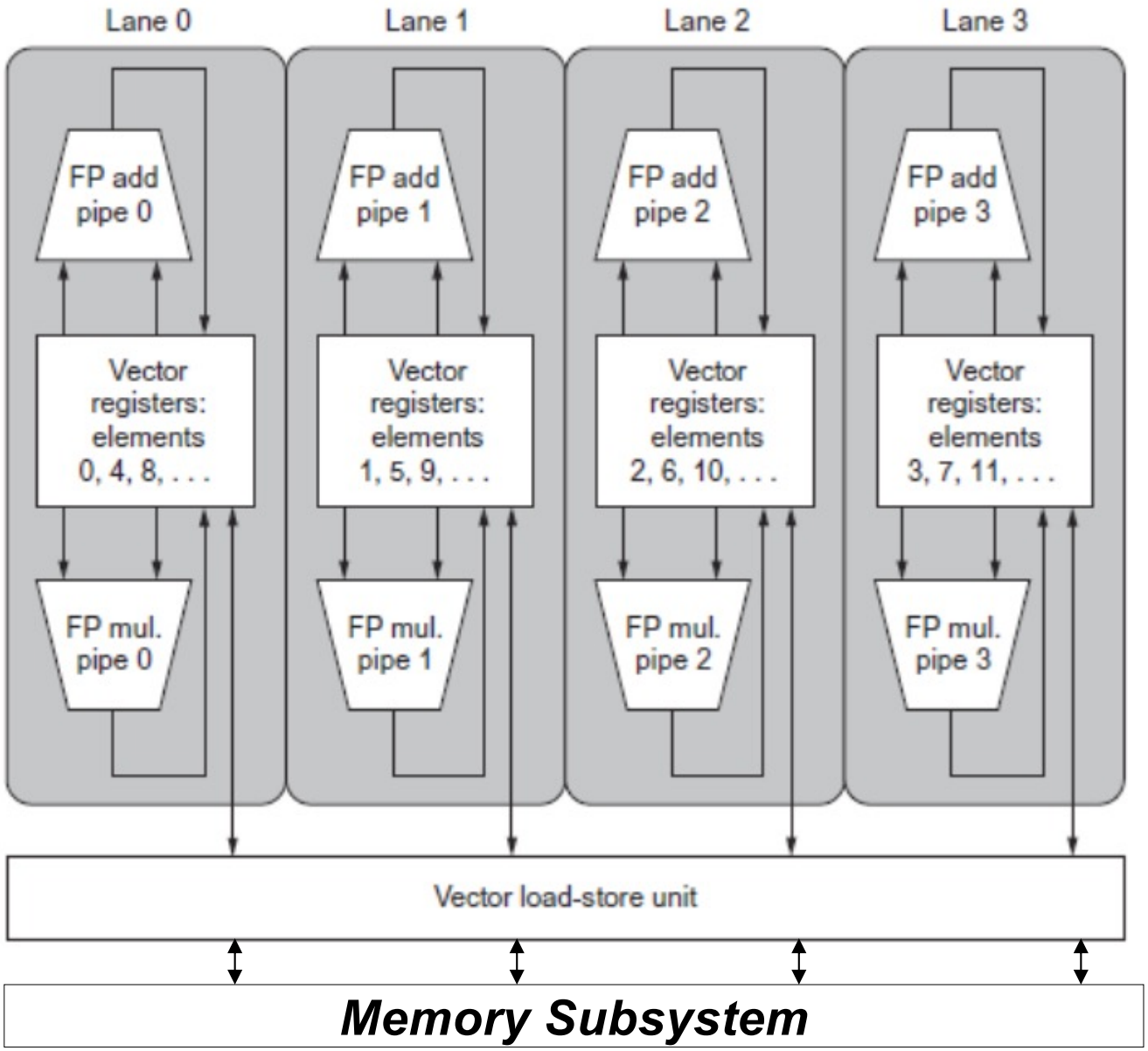

```
V3 <- v1 * v2
```

Vector Instruction Execution

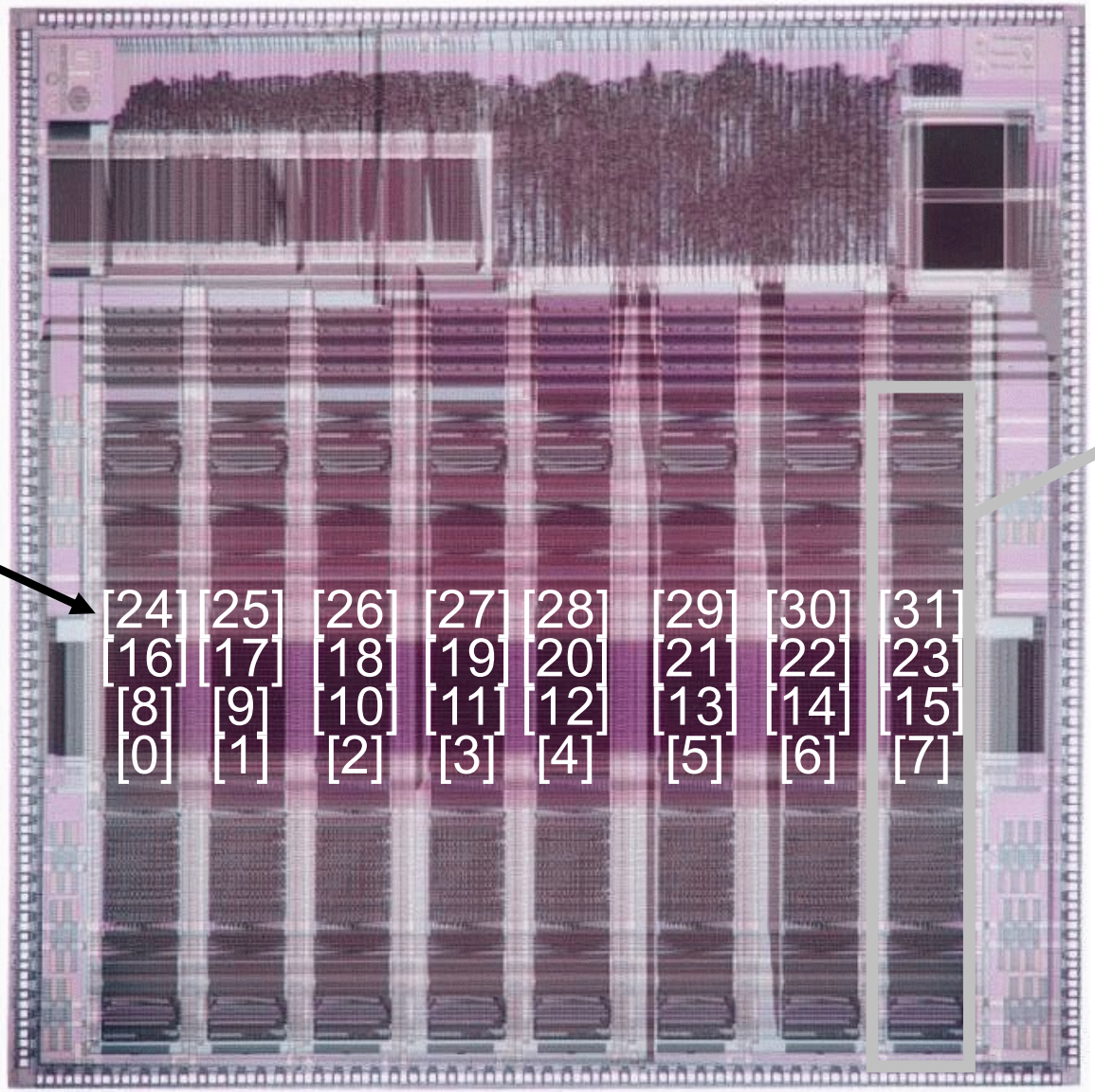


Multiple Lanes

The same set of functional units in each lane



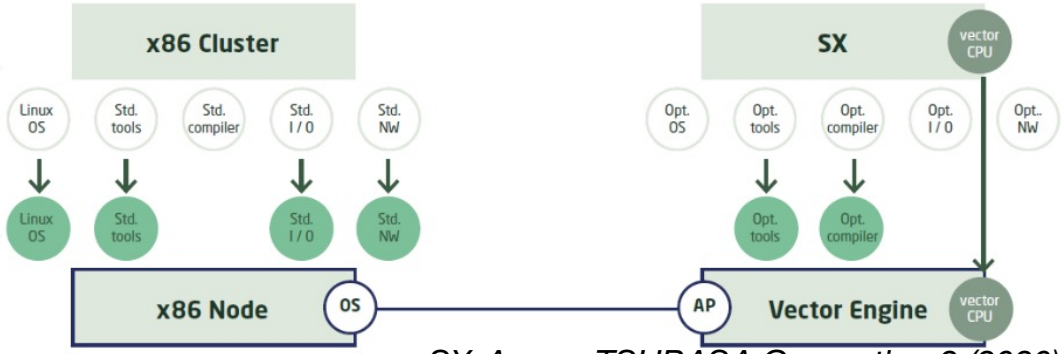
T0 Vector Microprocessor (1995)



NEC SX-Aurora TSUBASA (2020)

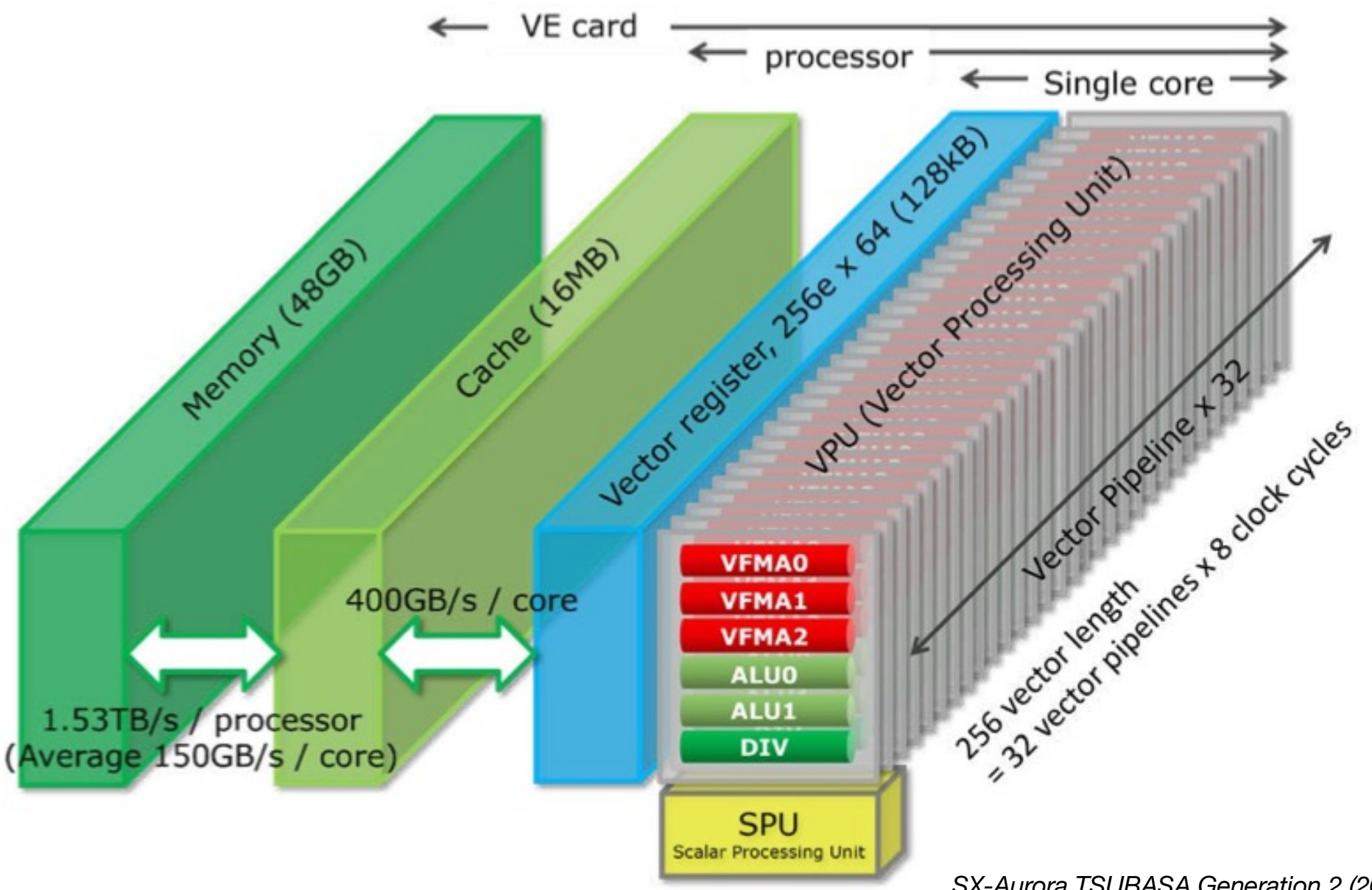
Vector Engine 20A processor

- integrated transparently into the Linux software environment
- 16 nm FinFET technology
- 1.6 GHz clock
- 10 vector cores
- 48 GB HBM2 (6 modules)
- 64 x 256 x 64b vector registers/core
- 32 vector lanes/core
- 3 FP DP FMA units/lane
- 192 FLOPs per cycle/core or up to 307 GFLOPS (DP)
- **3.07 TFLOPS** peak w/ 10 cores
- 200 W per Vector Engine
- 1.53 TB/s memory BW/VE
- 16 MB shared cache



SX-Aurora TSUBASA Generation 2 (2020)

Vector Engine Single Core & Memory

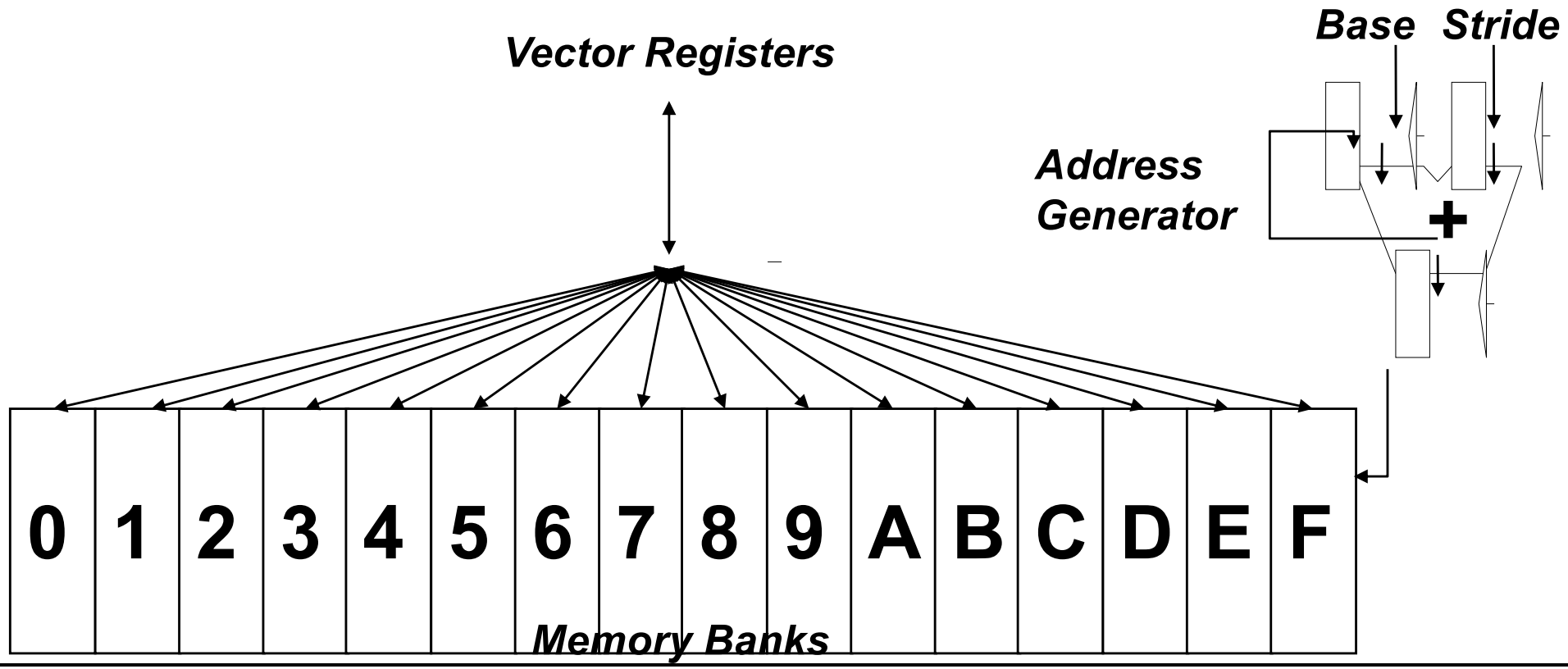


SX-Aurora TSUBASA Generation 2 (2020)

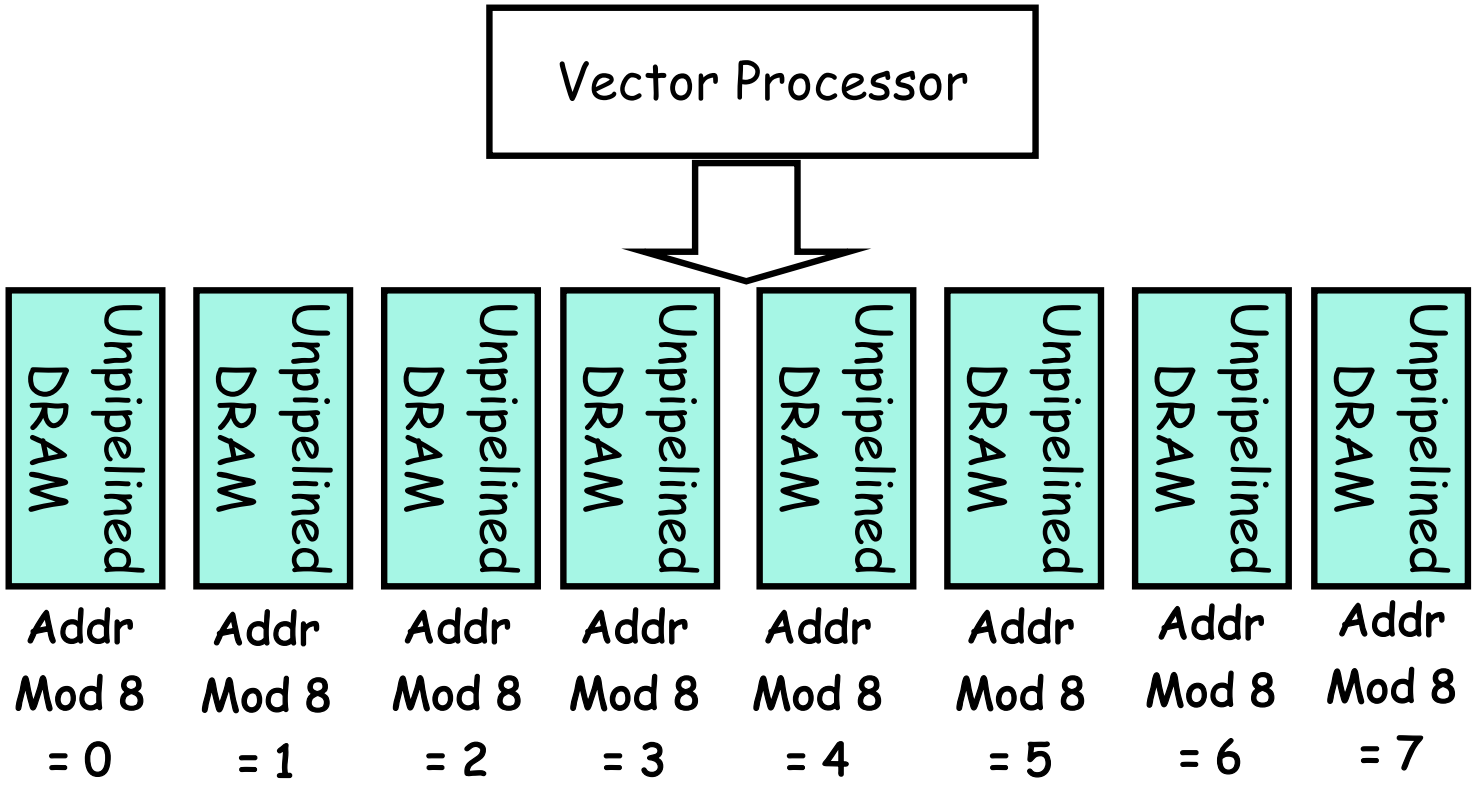
Vector Memory System

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

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



Interleaved Memory Layout



- Great for unit stride:
  - Contiguous elements in different DRAMs
  - Startup time for vector operation is latency of single read
- What about non-unit stride?
  - Above good for strides that are relatively prime to 8
  - Bad for: 2, 4
  - Better: prime number of banks…!

Memory Operations

- Load/store operations move groups of data between registers and memory
- Three types of addressing
  - Unit stride
    - Contiguous block of information in memory
    - Fastest: always possible to optimize this
  - Non-unit (constant) stride
    - Harder to optimize memory system for all possible strides
    - Prime number of data banks makes it easier to support different strides at full bandwidth
  - Indexed (gather-scatter)
    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

How to get full bandwidth for Unit Stride?

- Memory system must sustain (# lanes x word) /clock
- No. memory banks > memory latency to avoid stalls
  - *m* banks ⇒ *m* words per memory latency *l* clocks
  - if *m* < *l*, then gap in memory pipeline:

clock:	0	…	*l*	*l*+1	*l*+2	…	*l*+*m*- 1	l+m	…	2 *l*
word:	--	…	0	1	2	…	*m*-1	--	…	*m*

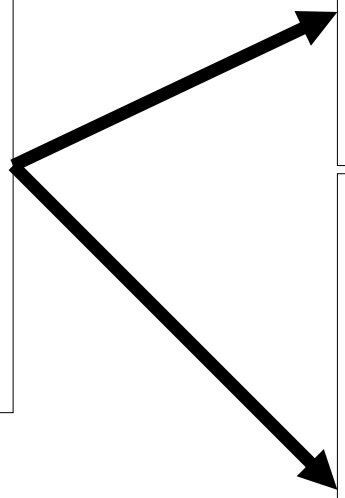
  - may have 1024 banks in SRAM
- If desired throughput greater than one word per cycle
  - Either more banks (start multiple requests simultaneously)
  - Or wider DRAMS. Only good for unit stride or large data types
- More banks/weird numbers of banks good to support more strides at full bandwidth

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

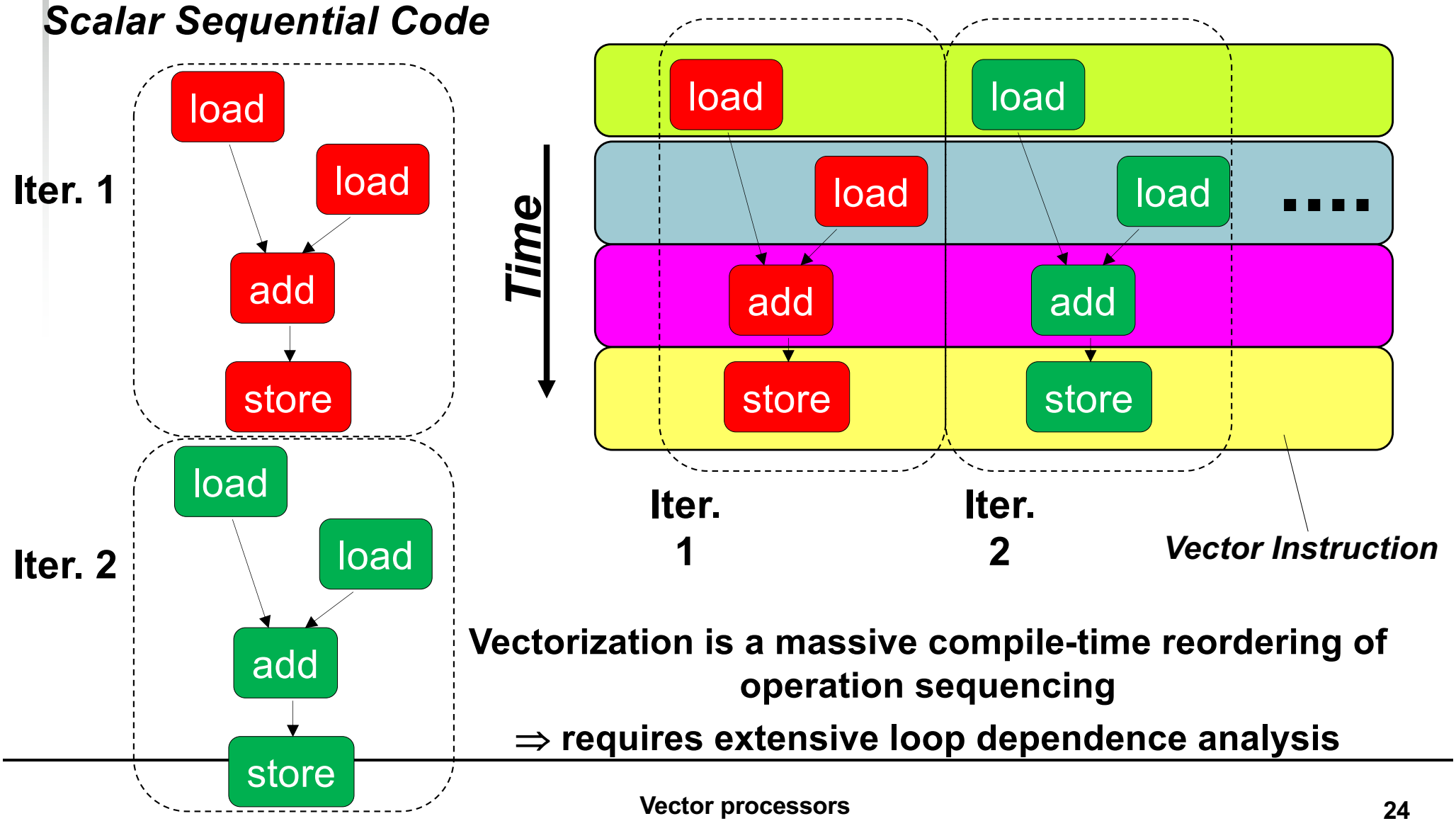
- Vector memory-memory architectures (VMMA) require greater main memory bandwidth, why?
  - All operands must be read in and out of memory
- VMMAs make if difficult to overlap execution of multiple vector operations, why?
  - Must check dependencies on memory addresses
- VMMAs incur greater startup latency
  - Scalar code was faster on CDC Star-100 for vectors < 100 elements
  - For Cray-1, vector/scalar breakeven point was around 2 elements

⇒ *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

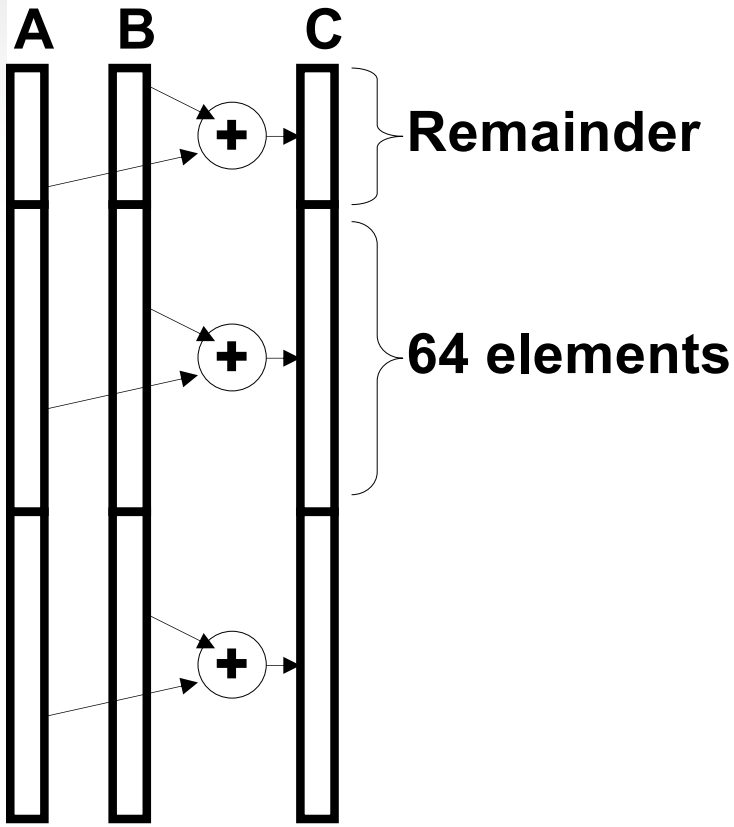
⇒ requires extensive loop dependence analysis

Vector Stripmining

Problem: Vector registers have finite length

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

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

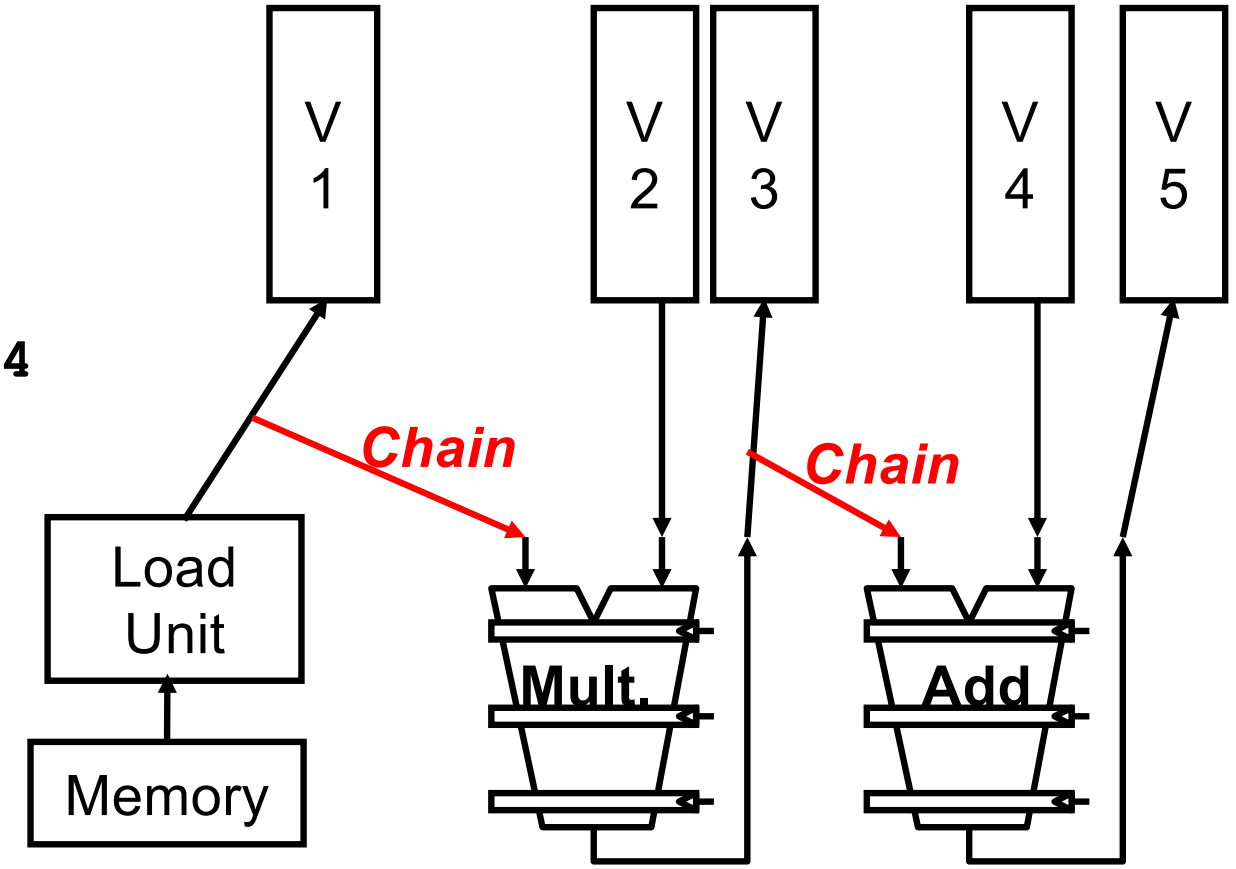



```
        ANDI R1, N, 63    # N mod 64
        MTC1 VLR, R1      # Do remainder
loop:
        LV V1, RA
        DSLL R2, R1, 3    # Multiply by 8
        DADDU RA, RA, R2  # Bump pointer
        LV V2, RB
        DADDU RB, RB, R2
        ADDV.D V3, V1, V2
        SV V3, RC
        DADDU RC, RC, R2
        DSUBU N, N, R1 # Subtract elements
        LI R1, 64
        MTC1 VLR, R1    # Reset full length
        BGTZ N, loop    # Any more to do?
```

Vector Chaining

- Vector version of register bypassing (forwarding)
  - introduced with Cray-1

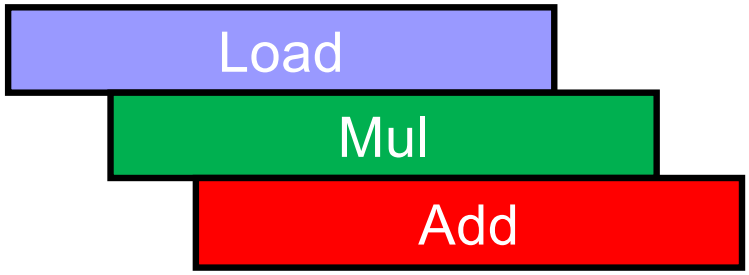
```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



Vector Chaining Advantage

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

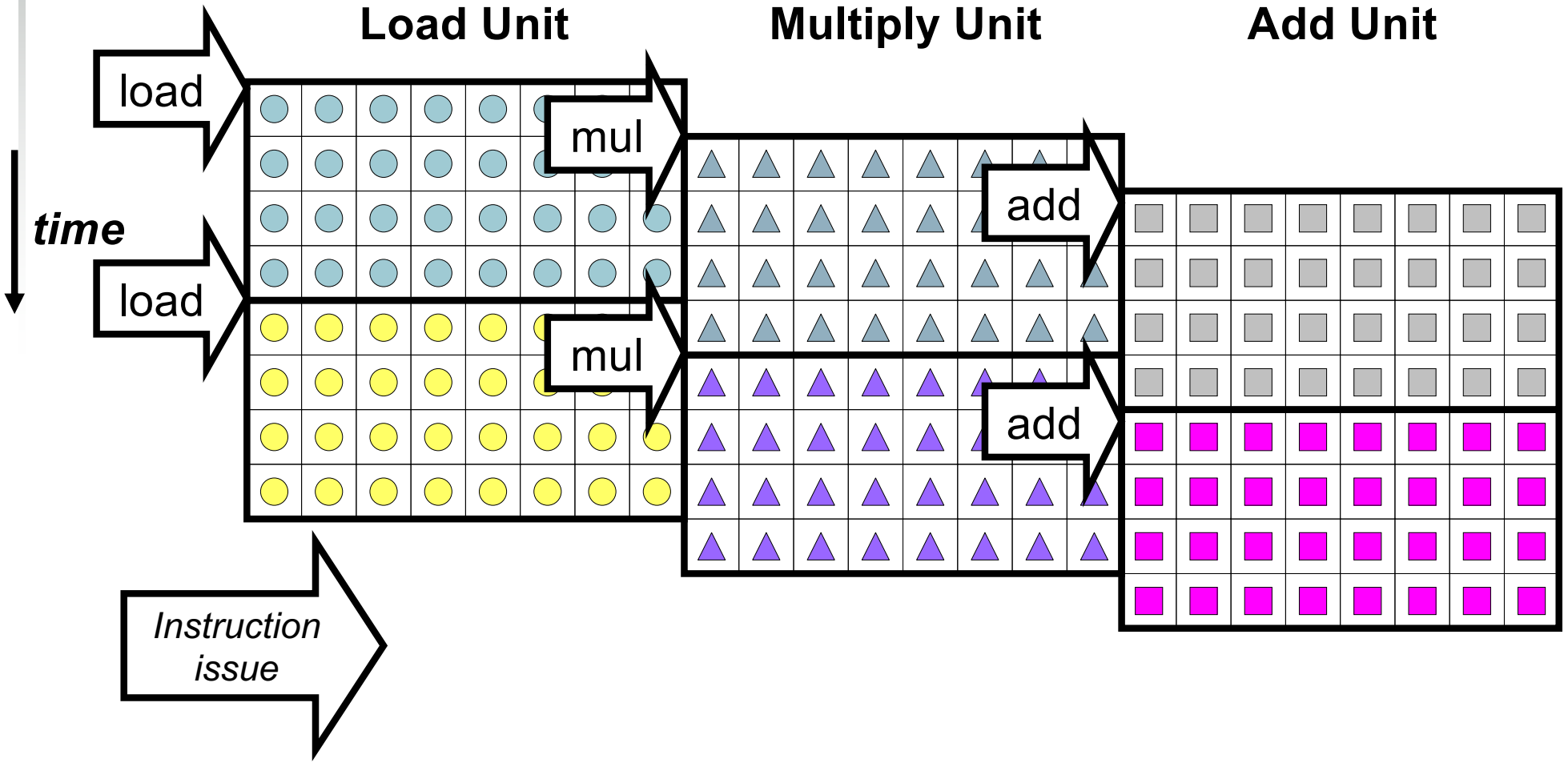
- With chaining, can start dependent instruction as soon as first result appears



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions with **vector chaining**

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

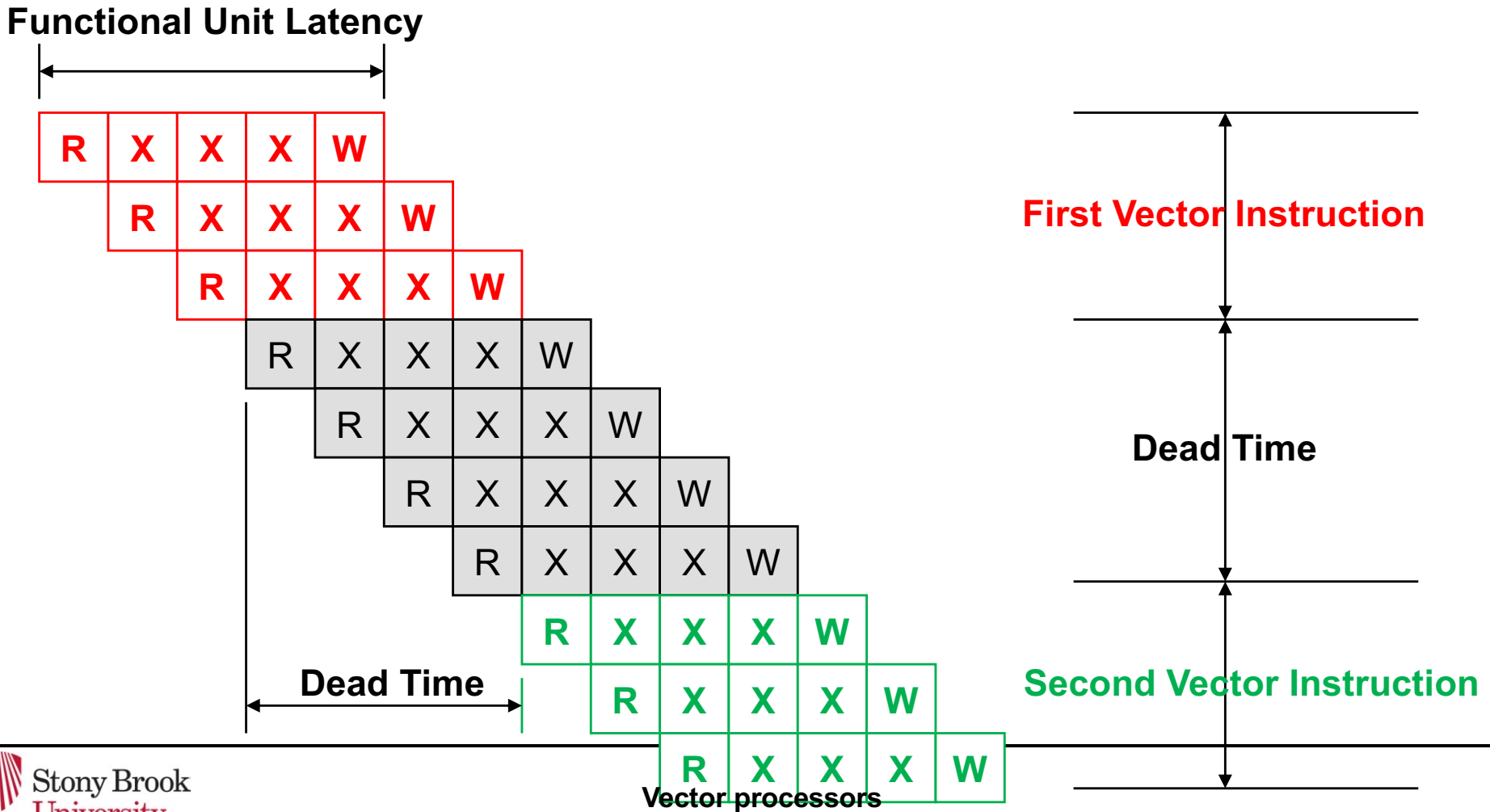


Complete 24 operations/cycle while issuing 1 short instruction/cycle

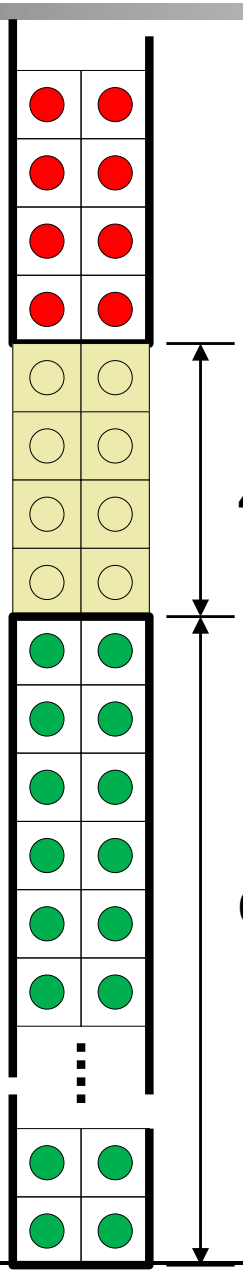
Vector Startup

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

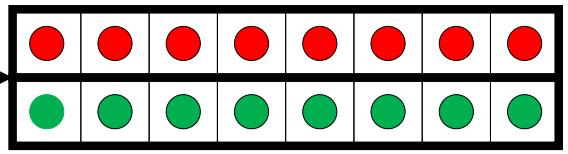
64 cycles active

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 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

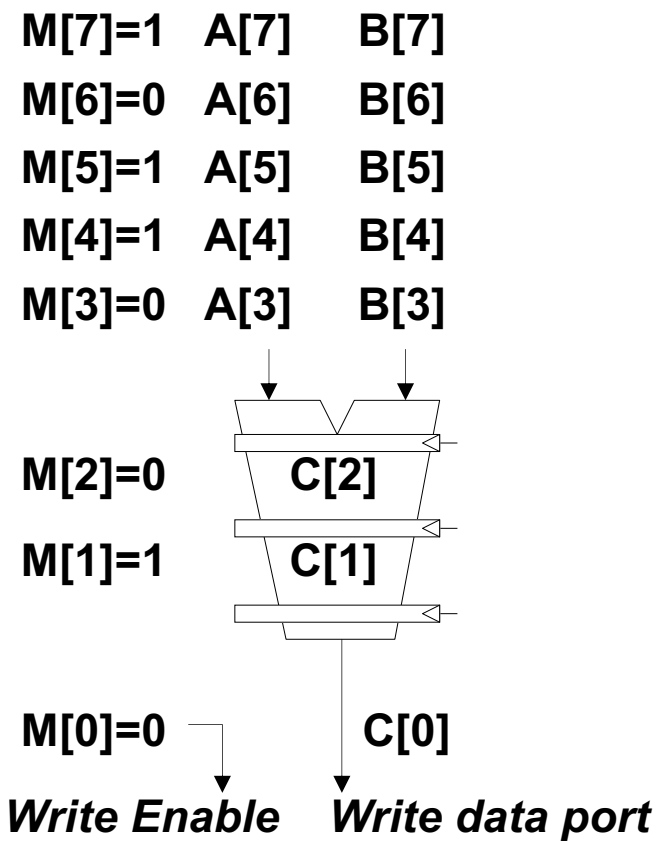
Code example:

```
CVM              # Clear vector mask
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

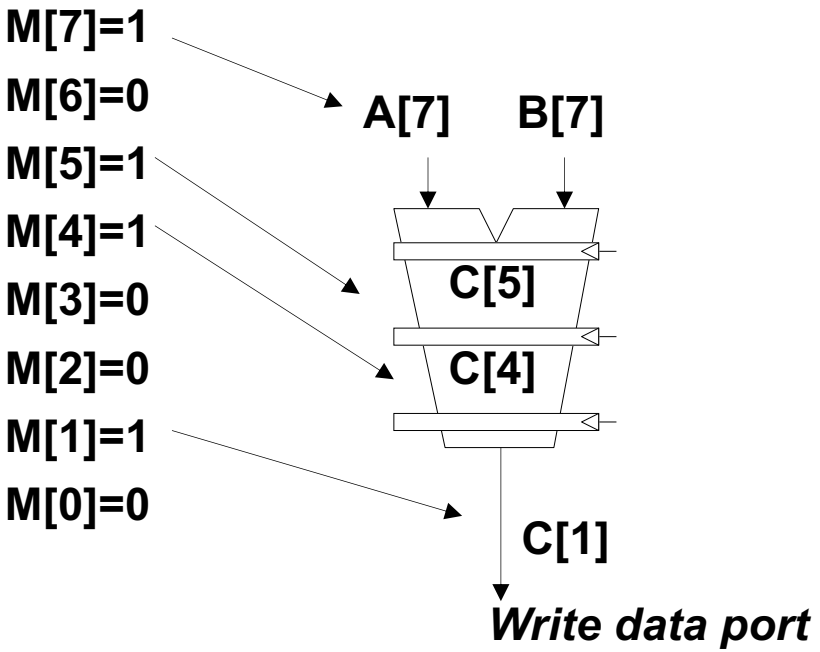
Simple Implementation

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



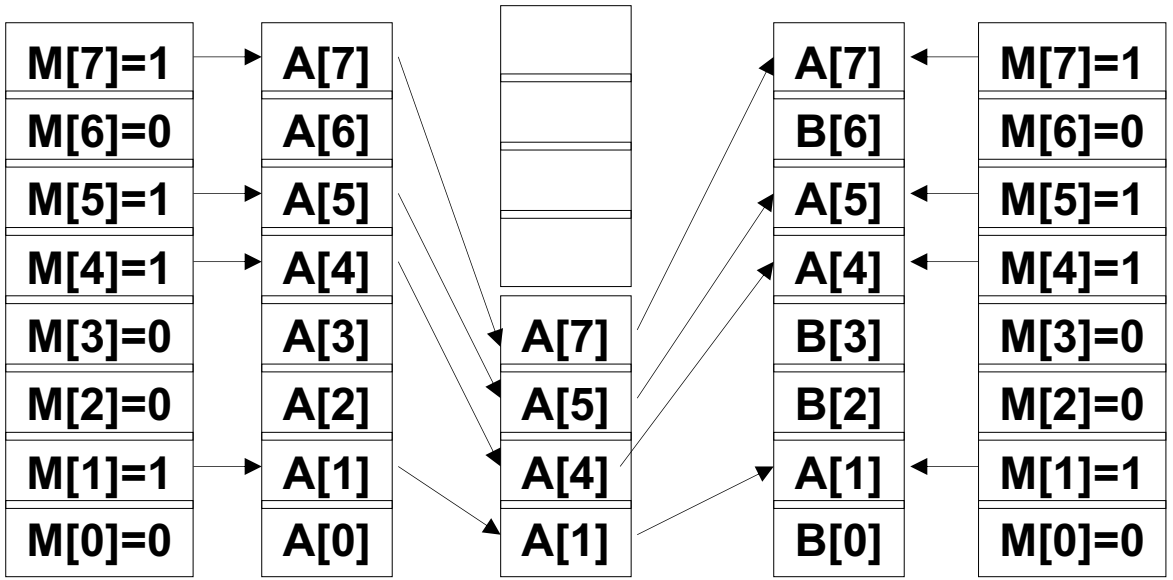
Density-Time Implementation

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



Compress/Expand Operations

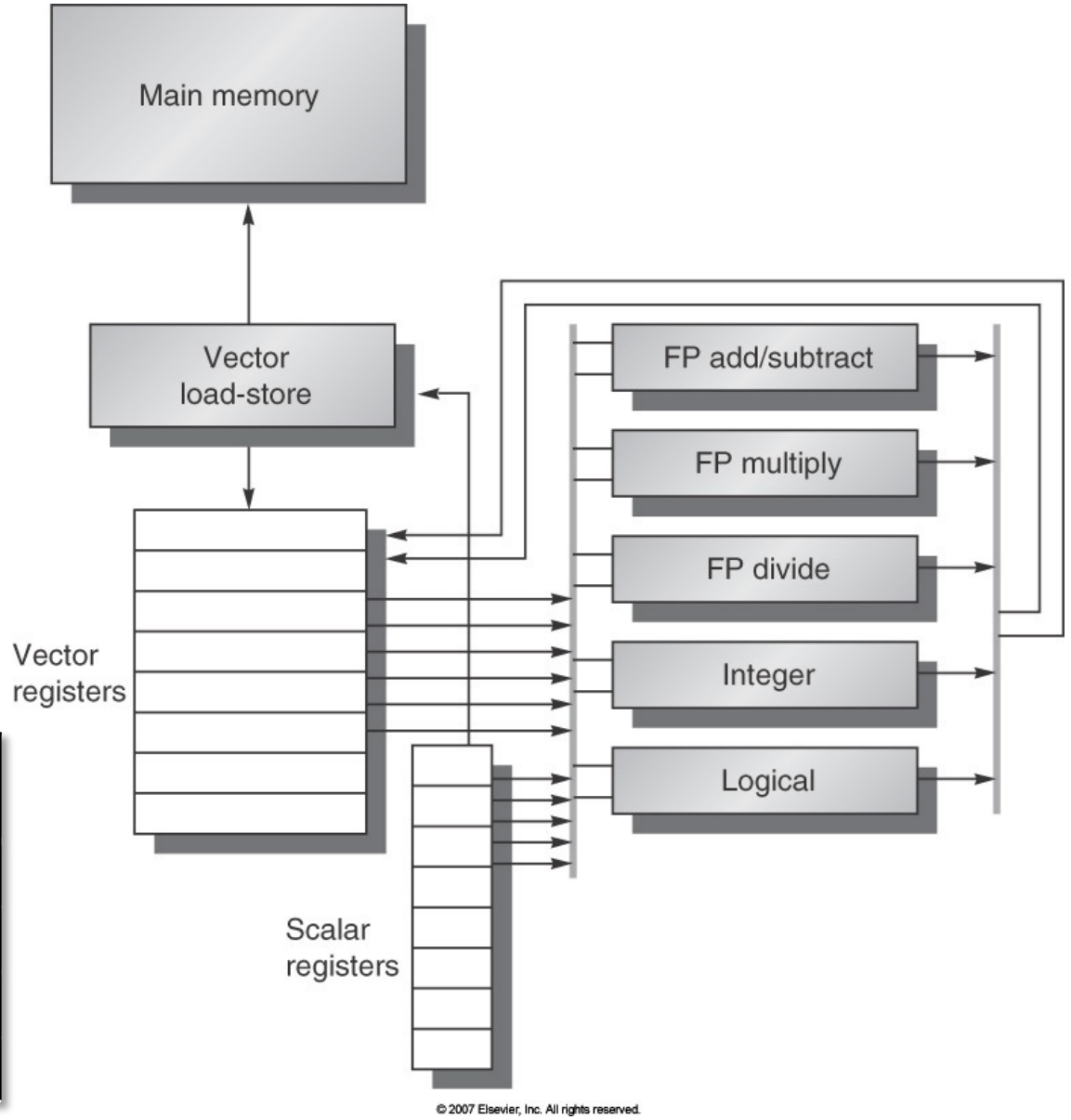
- **Compress** packs non-masked elements from one vector register contiguously at start of destination vector register
  - population count of mask vector gives packed vector length
- **Expand** performs inverse operation



Compress ***Expand***

Used for density-time conditionals and also for general selection operations

RV64V(aka VMIPS) Vector-Register Architecture



Vector operation	Start-up penalty
Load/store	12 cycles
FP Divide	20 cycles
FP Multiply	7 cycles
FP Add/Subtract	6 cycles

Vector Execution Time

- Time = f(vector length, data dependencies, structural hazards)
- ***Initiation rate***: rate that FU consumes vector elements (= number of lanes)
- ***Convoy***: set of vector instructions that can begin execution in same clock and continue their execution in parallel if there are:
  - Case 1 (*when vector chaining is supported*) : no structural hazards, i.e., there are no two or more vector instructions using the same unit (data hazards are OK)
  - **Case 2** (*when vector chaining is not supported like in the example below*)**:** no structural nor data hazards
- ***Chime***: approx. time to execute one convoy (using vector length ***n*** only)
- ***m* convoys take *m* chimes**; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead; good approximation for long vectors)

```
1:  LV    V1,Rx      ;load vector X
2:  MULV  V2,F0,V1   ;vector-scalar mult.
    LV    V3,Ry      ;load vector Y
3:  ADDV  V4,V2,V3   ;add
4:  SV    Ry,V4      ;store the result
```

Case 2: vector chaining is not supported

4 convoys, 1 lane, n=64, MVL=64

=> 4 x 64 = 256 clocks (w/o startup)

(or 4 clocks per result)

How many convoys are here when *vector chaining IS supported?*

Running Time of a Strip-Mined Loop

- Two key factors that contribute to the running time of a strip-mined loop consisting of
  - a sequence of convoys
  - scalar non-vector (non-convoy!) instructions for loop control
  - The number of convoys in the loop, which determines the number of chimes.
    - We use the notation **T*chime*** = number of convoys for the execution time in chimes.
  - The overhead for each strip-mined sequence of convoys.
    - This overhead consists of the cost of executing the scalar code for strip-mining each block, **T*loop***, plus the vector start-up cost for each convoy, **T*start***.
    - There may also be a fixed overhead associated with setting up the vector sequence the first time. In recent vector processors this overhead has become quite small, so we ignore it.
- The total running time for a vector sequence operating on a vector of length ***n,*** which we will call T_n:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times \left(T_{loop} + T_{start}\right) + n \times T_{chime}$$

The values of T*start*, T*loop*, and T*chime* are compiler and processor dependent.

We will use T*loop* =15 and MVL (max. vector length) =64 on VMIPS.

Common Vector Metrics

- R_∞: MFLOPS rate on an infinite-length vector
  - Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
  - (R_n is the MFLOPS rate for a vector of length n)
- $N_{1/2}$: The vector length needed to reach one-half of R_∞
  - a good measure of the impact of start-up
- N_V: The vector length needed to make vector mode faster than scalar mode
  - measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit

$$R_\infty = \lim_{n\to\infty}\left(\frac{\text{Float.point arithmetic operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}}\right)$$

$$R_\infty = \frac{\text{Float. point arithmetic operations per iteration} \times \text{Clock rate}}{\lim_{n\to\infty}\left(\frac{T_n}{n}\right)}$$

Operation & Instruction Count:
RISC v. Vector Processor

Spec92fp	Operations (Millions)			Instructions (M)		
Program	RISC	Vector	R / V	RISC	Vector	R / V
swim256	115	95	1.1x	115	0.8	142x
hydro2d	58	40	1.4x	58	0.8	71x
nasa7	69	41	1.7x	69	2.2	31x
su2cor	51	35	1.4x	51	1.8	29x
tomcatv	15	10	1.4x	15	1.3	11x
wave5	27	25	1.1x	27	7.2	4x
mdljdp2	32	52	0.6x	32	15.8	2x

Vector reduces ops by 1.2X, instructions by 20X

Vectors Lower Power

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access in whole cache lines

Vector

- One inst fetch, decode, dispatch per vector
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary

Superscalar Energy Efficiency Even Worse

Superscalar

- Control logic grows quadratically with issue width
- Control logic consumes energy regardless of available parallelism
- Speculation to increase visible parallelism wastes energy

Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use
- Vector instructions expose parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals

Multimedia SIMD Extensions

- First very short vectors added to existing ISAs for micros
  - E.g., 64-bit registers split into 2x32b or 4x16b or 8x8b
- Newer designs have 128-bit (Altivec, SSE2) or 256/512-bit regs
- Limited instruction set at first:
  - no vector length control
  - no strided load/store or scatter/gather (added now)
  - unit-stride loads must be aligned to 64/128/256/512-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

Vector Applications

Limited to scientific computing? No!

- 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)
- Machine learning (Tensor units operating on small matrices)

Parallel Vector Architecture

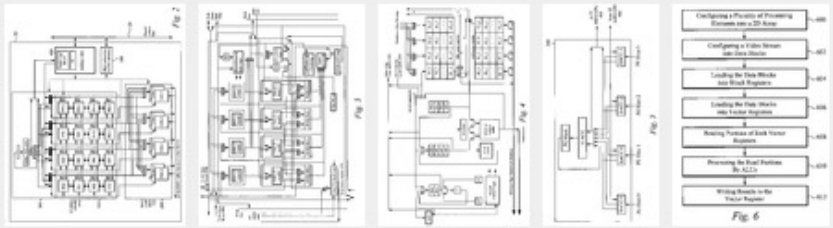
(M. Dorojevets, E. Ogura, SONY Electronics, 2004)

Parallel vector processing

Abstract

A video platform architecture provides video processing using parallel vector processing. The video platform architecture includes a plurality of video processing modules, each module including a plurality of processing elements (PEs). Each PE provides parallel vector processing. Specifically, means are provided to read all elements of one or two source vector registers in each PE simultaneously, process the read elements by a set of arithmetic-logical units (ALUs), and write back all results to one of the vector registers, all of which occurs in one PE cycle. To provide such parallel vector processing capabilities, the datapath of each PE is built as a set of identical PE processing slices, each of which includes an integer arithmetic-logical unit (ALU), a vector register bank, and a block register bank. A block/vector register bank holds all I elements of row J in a two-dimensional I×J data blocks for all block/vector registers provided by the architecture.

Images (5)



Classifications

G06F9/3885 Concurrent instruction execution, e.g. pipeline, look ahead using a plurality of independent parallel functional units

US7196708B2
United States

Download PDF | Find Prior Art | Similar

Inventor: Mikhail Dorojevets, Eiji Ogura

Current Assignee : Sony Corp , Sony Electronics Inc

Worldwide applications

2004 · US

Application US10/815,329 events

2004-03-31	Application filed by Sony Corp, Sony Electronics Inc
2004-03-31	Priority to US10/815,329
2004-08-13	Assigned to SONY CORPORATION, SONY ELECTRONICS INC.
2005-10-06	Publication of US20050219422A1
2007-03-27	Application granted
2007-03-27	Publication of US7196708B2
Status	Expired - Fee Related

NEC VE against Nvidia Ampere GPU (2020)

Vector Engine 20A processor

- 16 nm FinFET technology
- 1.6 GHz clock
- **10 vector cores**
- 48 GB HBM2 (6 modules)
- 1.53 TB/s memory BW/VE
- **200 W** per Vector Engine
- 64 x 256 x 64b vector registers/core
  - **128KB/vector core**
- 32 vector lanes/core
- 3 FP DP FMA units/core
- **192 FLOPs per cycle/core** or up to 307 GFLOPS (DP)
- **3.07 TFLOPS** peak w/ **10 cores**
- 16 MB shared cache

Nvidia Ampere GA100

- TSMC's 7 nm FinFET process
- 1.41 GHz boost clock
- **108 SM cores**
- 40 GB of HBM2 DRAM
- 1555 GB/s Memory BW
- TDP: **400 W**
- 65536 32b registers/SM (**256KB/SM core**)
- Each SM has 4 partitions of CUDA cores
  - 16 INT32 + 16 FP32 units + 8 FP64 + Tensor core per partition
  - parallel execution of int32 and fp32 instructions
  - 64 (16 x 4) FMA FP32 + 64 INT32 ops/clock
  - 32 FMA FP64 **(64 FP64 ops/clock)**
- 6912 (16 x 4 x 108) FP32 CUDA cores
- Peak FP32: 19.5 TFLOPS
- **Peak FP64: 9.7 TFLOPS** (w/ **108 SM cores**)
- 20736 KB (192KB x108) L1 mem/cache
- 40960 KB L2 cache
  - HW-controlled L2 cache coherency across GPU

Vector Summary

- Vector is alternative (DLP) model for exploiting patallelisim
- If code is vectorizable, then simpler hardware, more energy efficient, and better real-time model than Out-of-order machines
- Design issues include number of lanes, number of functional units, number of vector registers, length of vector registers, exception handling, conditional operations
- Fundamental design issue is memory bandwidth
  - With virtual address translation and caching
- Will multimedia popularity revive vector architectures?
- Or GPUs will beat them all?

Acknowledgements

- These slides contain material developed and copyright by:
  - Morgan Kauffmann (Elsevier, Inc.)
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Mikhail Dorojevets (SBU)