Lecture 20: Data Level Parallelism -- Introduction and Vector Architecture

CSE 564 Computer Architecture Summer 2017

Department of Computer Science and Engineering

Yonghong Yan

yan@oakland.edu

www.secs.oakland.edu/~yan

Very Important Terms

- Dynamic Scheduling → Out-of-order Execution
- Speculation → In-order Commit
- Superscalar → Multiple Issue

Techniques	Goals	Implementation	Addressing	Approaches
Dynamic Scheduling	Out-of-order execution	Reservation Stations, Load/Store Buffer and CDB	Data hazards (RAW, WAW, WAR)	Register renaming
Speculation	In-order commit	Branch Prediction (BHT/BTB) and Reorder Buffer	Control hazards (branch, func, exception)	Prediction and misprediction recovery
Superscalar/ VLIW	Multiple issue	Software and Hardware	To Increase CPI	By compiler or hardware

Last Lecture: Multithreading

CSE 564 Class Contents

- Introduction to Computer Architecture (CA)
- Quantitative Analysis, Trend and Performance of CA
  - Chapter 1
- Instruction Set Principles and Examples
  - Appendix A
- Pipelining and Implementation, RISC-V ISA and Implementation
  - Appendix C, RISC-V (riscv.org) and UCB RISC-V impl
- Memory System (Technology, Cache Organization and Optimization, Virtual Memory)
  - Appendix B and Chapter 2
  - Midterm covered till Memory Tech and Cache Organization
- Instruction Level Parallelism (Dynamic Scheduling, Branch Prediction, Hardware Speculation, Superscalar, VLIW and SMT)
  - Chapter 3
- Data Level Parallelism (Vector, SIMD, and GPU)
  - Chapter 4
- Thread Level Parallelism
  - Chapter 5

Topics for Data Level Parallelism (DLP)

- Parallelism (centered around …)
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

- DLP Introduction and Vector Architecture
  - 4.1, 4.2
- SIMD Instruction Set Extensions for Multimedia
  - 4.3
- Graphical Processing Units (GPU)
  - 4.4
- GPU and Loop-Level Parallelism and Others
  - 4.4, 4.5, 4.6, 4.7

Finish in two/three sessions

Acknowledge and Copyright

- Slides adapted from
  - UC Berkeley course “Computer Science 252: Graduate Computer Architecture” of David E. Culler Copyright(C) 2005 UCB
  - UC Berkeley course Computer Science 252, Graduate Computer Architecture Spring 2012 of John Kubiatowicz Copyright(C) 2012 UCB
  - Computer Science 152: Computer Architecture and Engineering, Spring 2016 by Dr. George Michelogiannakis from UC Berkeley
  - Arvind (MIT), Krste Asanovic (MIT/UCB), Joel Emer (Intel/MIT), James Hoe (CMU), John Kubiatowicz (UCB), and David Patterson (UCB)

- https://passlab.github.io/CSE564/copyrightack.html

Flynn's Classification (1966)

Broad classification of parallel computing systems

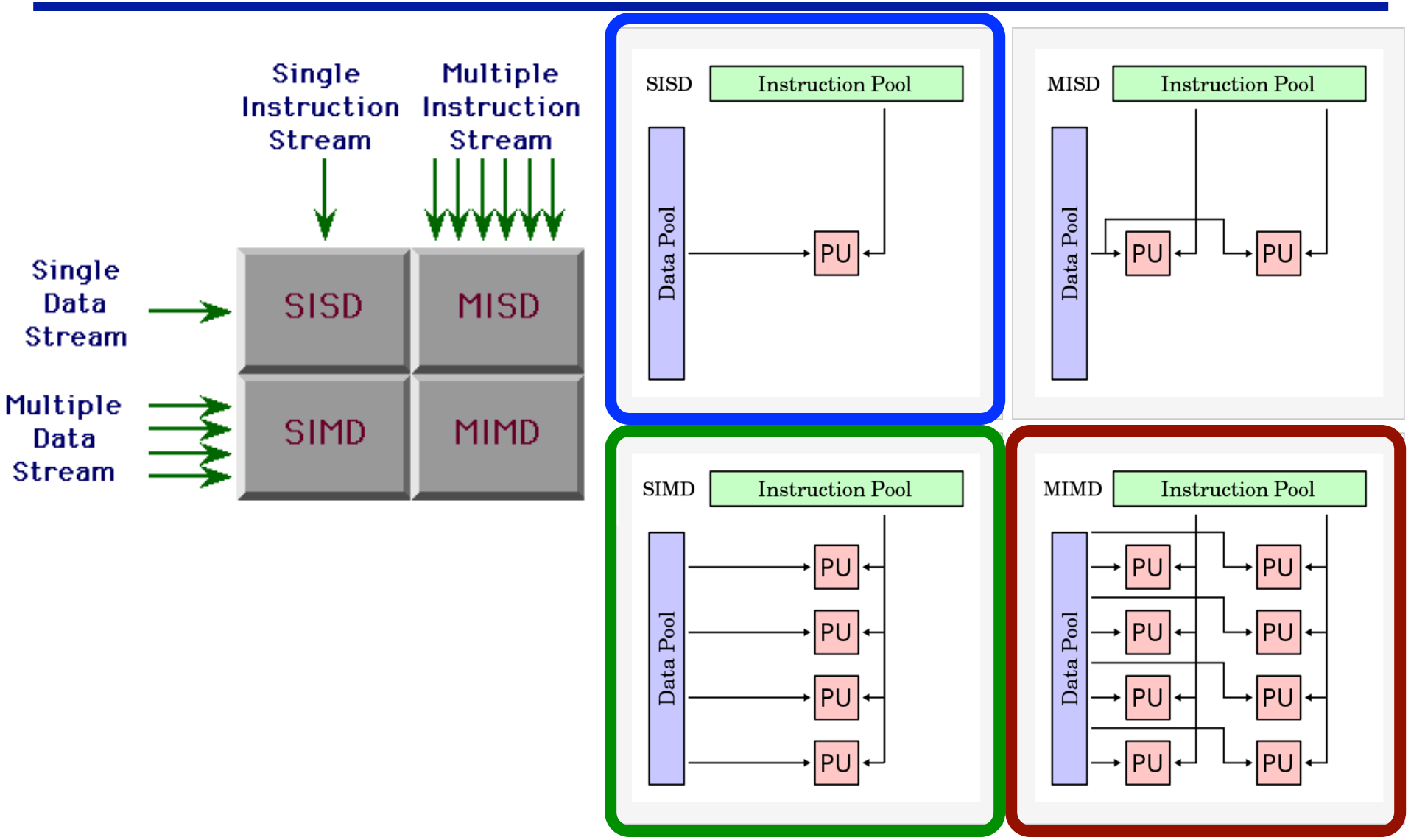
- **based upon the number of concurrent Instruction (or control) streams and Data streams**

Michael J. Flynn:

http://arith.stanford.edu/~flynn/

- **SISD: Single Instruction, Single Data**
  - **conventional uniprocessor**
- **SIMD: Single Instruction, Multiple Data**
  - **one instruction stream, multiple data paths**
  - **distributed memory SIMD (MPP, DAP, CM-1&2, Maspar)**
  - **shared memory SIMD (STARAN, vector computers)**
- **MIMD: Multiple Instruction, Multiple Data**
  - **message passing machines (Transputers, nCube, CM-5)**
  - **non-cache-coherent shared memory machines (BBN Butterfly, T3D)**
  - **cache-coherent shared memory machines (Sequent, Sun Starfire, SGI Origin)**
- **MISD: Multiple Instruction, Single Data**
  - **Not a practical configuration**

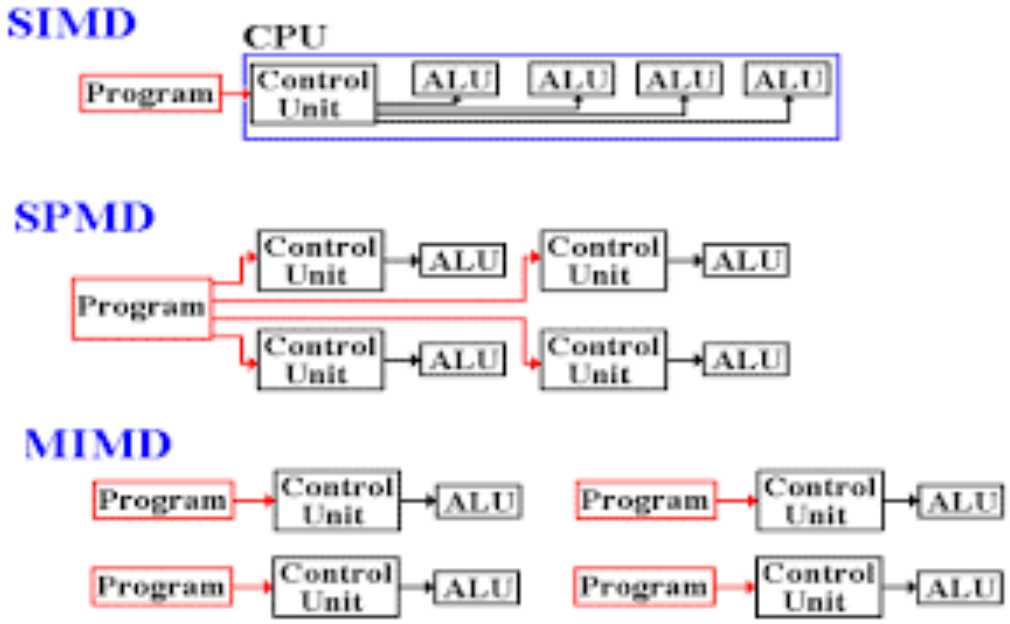
Flynn's Taxonomy



https://en.wikipedia.org/wiki/Flynn%27s_taxonomy

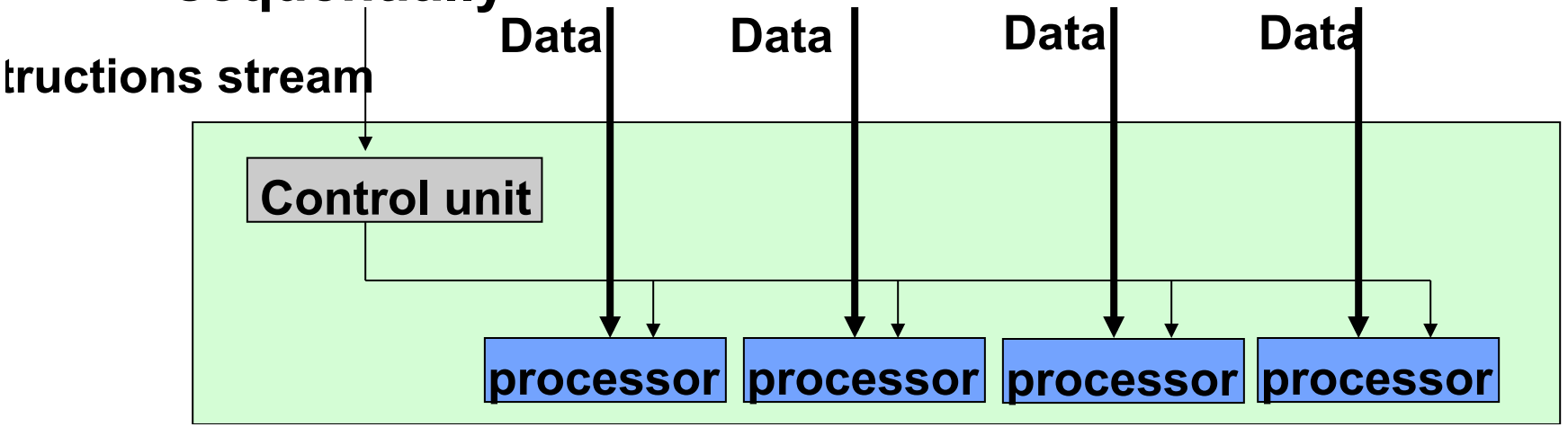
More Categories

- **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) ≈ MIMD**
  - Multiple autonomous processors simultaneously operating at least 2 independent programs



SIMD: Single Instruction, Multiple Data (Data Level Paralleism)

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation processing multiple data elements
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially



SIMD Parallelism

- Three variations
  - Vector architectures
  - SIMD extensions
  - Graphics Processor Units (GPUs)

- For x86 processors:
  - Expect two additional cores per chip per year (MIMD)
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

Vector Architecture

VLIW vs Vector

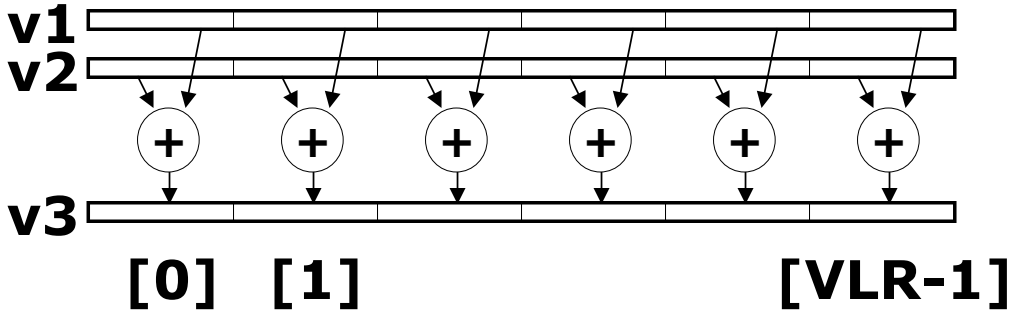
- VLIW takes advantage of instruction level parallelism (ILP) by specifying multiple instructions to execute in parallel

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

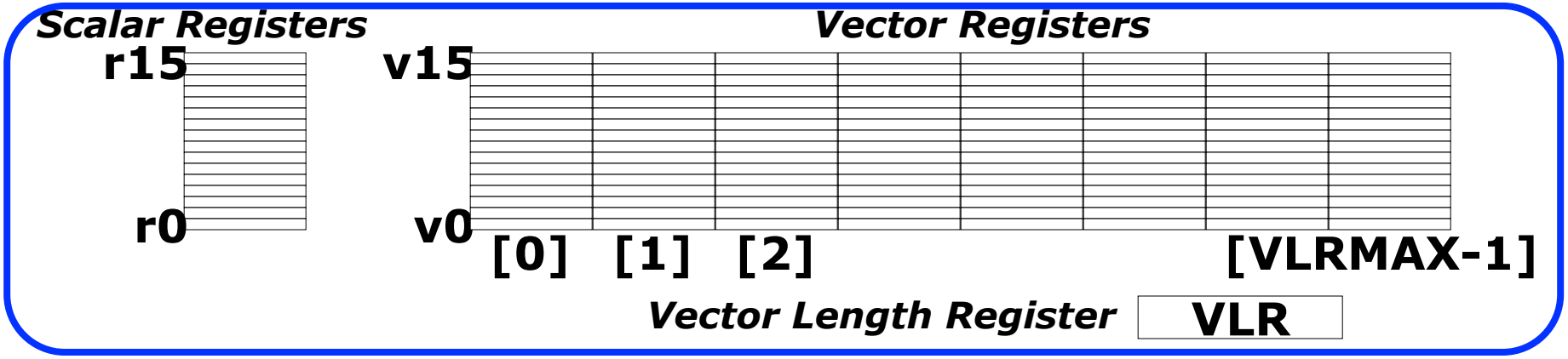
- Vector architectures perform the same operation on multiple data elements – single instruction
  - Data-level parallelism

Vector Arithmetic Instructions

ADDV v3, v1, v2

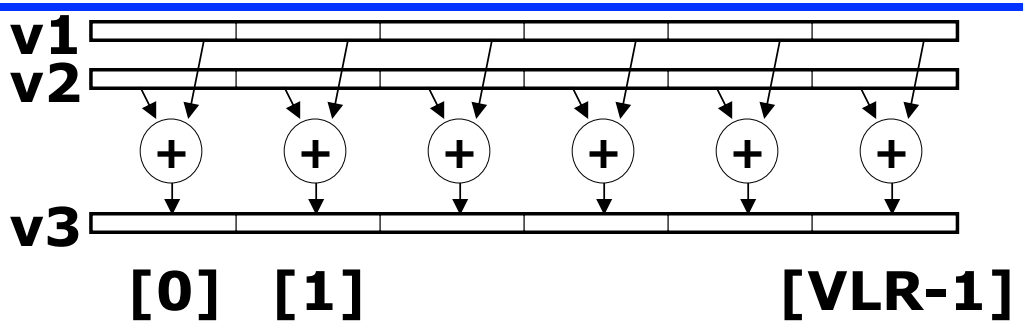


Vector Programming Model



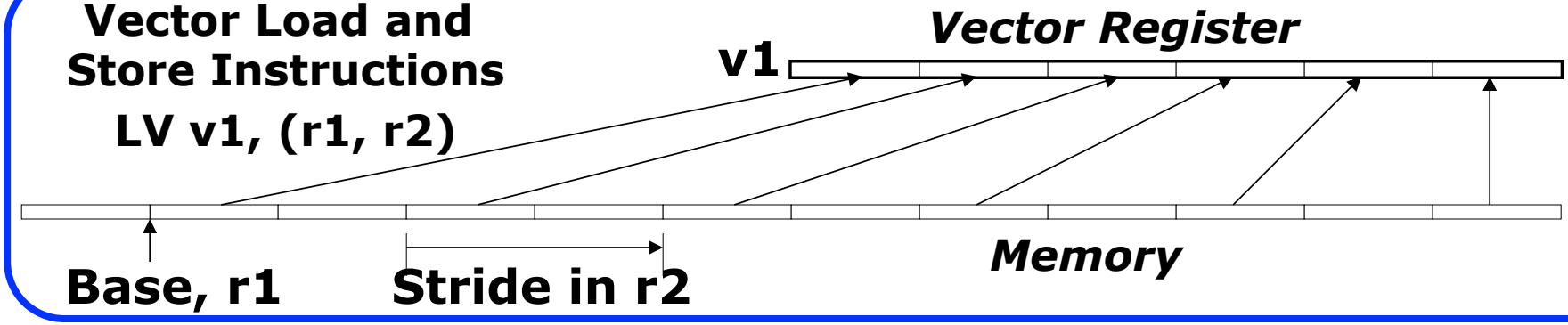
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

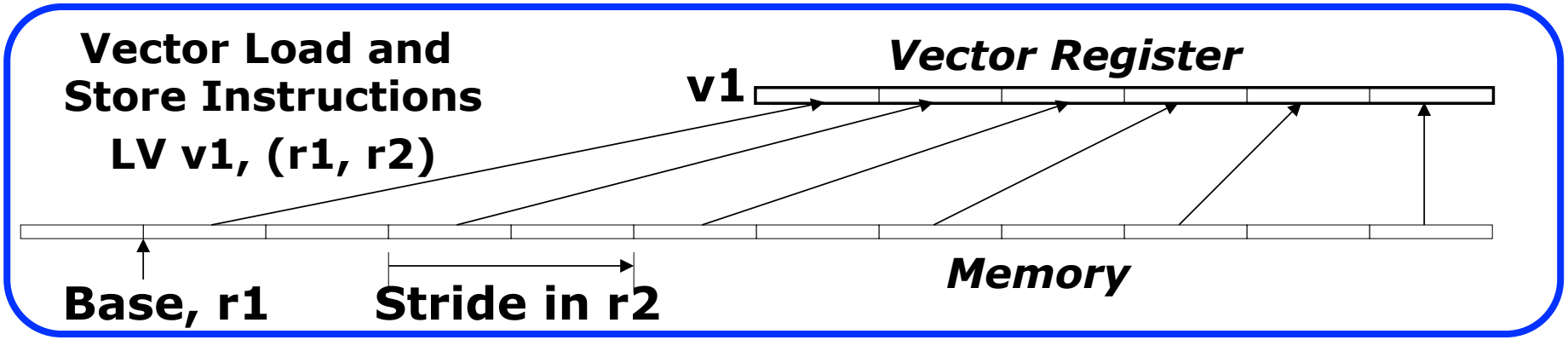
LV v1, (r1, r2)



Control Information

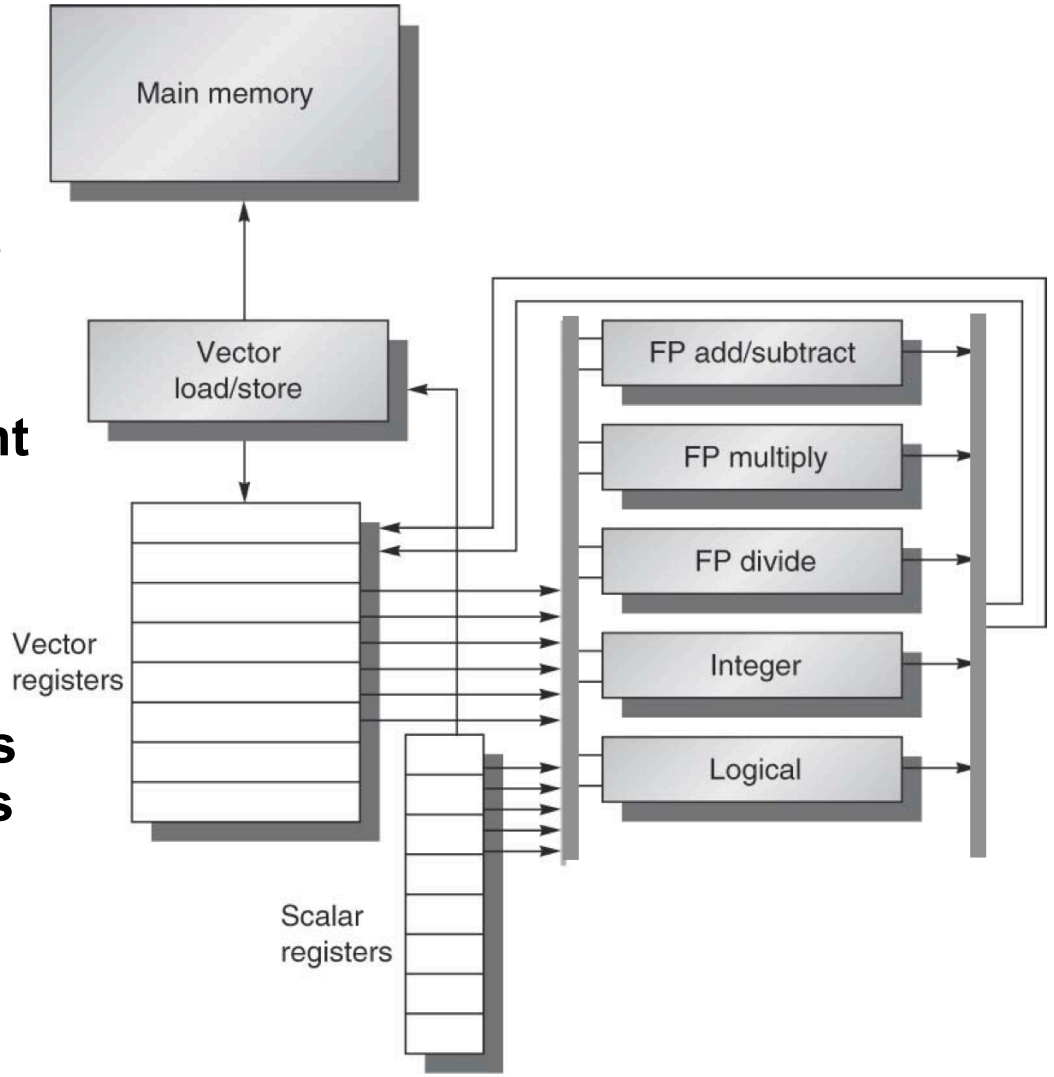
- VLR limits the highest vector element to be processed by a vector instruction
  - VLR is loaded prior to executing the vector instruction with a special instruction

- 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



Basic Structure of Vector Architecuture

- VMIPS
- eight 64-element vector registers
- all the functional units are vector functional units.
- The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations.
- A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.



VMIPS Vector Instructions

- **Suffix**
  - VV suffix
  - VS suffix
- **Load/Store**
  - LV/SV
  - LVWS/SVWS
- **Registers**
  - VLR (vector length register)
  - VM (vector mask)

Instruction	Operands	Function
ADDVV.D	V1,V2,V3	Add elements of V2 and V3, then put each result in V1.
ADDVS.D	V1,V2,F0	Add F0 to each element of V2, then put each result in V1.
SUBVV.D	V1,V2,V3	Subtract elements of V3 from V2, then put each result in V1.
SUBVS.D	V1,V2,F0	Subtract F0 from elements of V2, then put each result in V1.
SUBSV.D	V1,F0,V2	Subtract elements of V2 from F0, then put each result in V1.
MULVV.D	V1,V2,V3	Multiply elements of V2 and V3, then put each result in V1.
MULVS.D	V1,V2,F0	Multiply each element of V2 by F0, then put each result in V1.
DIVVV.D	V1,V2,V3	Divide elements of V2 by V3, then put each result in V1.
DIVVS.D	V1,V2,F0	Divide elements of V2 by F0, then put each result in V1.
DIVSV.D	V1,F0,V2	Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2).
SVWS	(R1,R2),V1	Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2).
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index).
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--VV.D	V1,V2	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand.
S--VS.D	V1,F0	
POP	R1,VM	Count the 1s in vector-mask register VM and store count in R1.
CVM		Set the vector-mask register to all 1s.
MTC1	VLR,R1	Move contents of R1 to vector-length register VL.
MFC1	R1,VLR	Move the contents of vector-length register VL to R1.
MVTM	VM,F0	Move contents of F0 to vector-mask register VM.
MVFM	F0,VM	Move contents of vector-mask register VM to F0.

Figure 4.3 The VMIPS vector instructions, showing only the double-precision floating-point operations. I
addition to the vector registers, there are two special registers, VLR and VM, discussed below. These special registe

Highlight of VMIPS Vector Instructions

- Vector operations have the letters "VV/VS" appended.
  - E.g. ADDVV.D is an addition of two double-precision vectors.
- Vector instructions input:
  - 1) a pair of vector registers (ADDVV.D) or
  - 2) a vector register and a scalar register (ADDVS.D).
    - » all operations use the same value in the scalar register as one input.
- LV/LVWS and SV/SVWS: vector load and vector store which load or store an entire vector of double-precision data.
  - One operand is the vector register to be loaded or stored;
  - The other operand, a GPR, is the starting address of the vector in memory.
  - LVWS/SVWS: For stride load/store
  - LVI/SVI: indexed load/store
- Two additional special-purpose registers:
  - Vector-length register: when the natural vector length is NOT 64
  - Vector-mask register: when loops involve IF statements.
- In-order scalar processor for vector architecture
  - Not out- of-order superscalar processors.
- Vectors naturally accommodate varying data sizes.
  - one view of a vector register size is 64 64-bit data elements, but 128 32-bit elements, 256 16-bit elements, and even 512 8-bit elements are equally valid views.

AXPY (64 elements) (Y = a * X + Y) in MIPS and VMIPS

```
for (i=0; i<64; i++)
  Y[i] = a* X[i] + Y[i];
```

The starting addresses of X and Y are in Rx and Ry, respectively

- **# instrs:**
  - **6 vs ~600**
- **Pipeline stalls**
  - **64x higher by MIPS**
- **Vector *chaining* (forwarding)**
  - **V1, V2, V3 and V4**

```
        L.D       F0,a          ;load scalar a
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.D       F2,0(Rx)      ;load X[i]
        MUL.D     F2,F2,F0      ;a × X[i]
        L.D       F4,0(Ry)      ;load Y[i]
        ADD.D     F4,F4,F2      ;a × X[i] + Y[i]
        S.D       F4,9(Ry)      ;store into Y[i]
        DADDIU    Rx,Rx,#8      ;increment index to X
        DADDIU    Ry,Ry,#8      ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ      R20,Loop      ;check if done
```

```
L.D       F0,a          ;load scalar a
LV        V1,Rx         ;load vector X
MULVS.D   V2,V1,F0      ;vector-scalar multiply
LV        V3,Ry         ;load vector Y
ADDVV.D   V4,V2,V3      ;add
SV        V4,Ry         ;store the result
```

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)

Vector Instruction Set Advantages

- **Compact**
  - one short instruction encodes N operations
- **Expressive and predictable, 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)

History: 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 (originally)**

- **CDC6600 (Cray, 1964) regarded as first supercomputer**
  - **A vector machine**

- **www.cray.com: The Supercomputer Company**

The Father of Supercomputing

Seymour Cray

Electrical engineer

Seymour Roger Cray was an American electrical engineer and supercomputer architect who designed a series of computers that were the fastest in the world for decades, and founded Cray Research which built many of these machines. Wikipedia

Born: September 28, 1925, Chippewa Falls, WI

Died: October 5, 1996, Colorado Springs, CO

Awards: Eckert–Mauchly Award

Parents: Seymour R. Cray, Lillian Cray

Education: University of Minnesota, Chippewa Falls High School

Fields: Applied mathematics, Computer Science, Electrical engineering

https://en.wikipedia.org/wiki/Seymour_Cray

http://www.cray.com/company/history/seymour-cray

Supercomputer Applications

- Typical application areas
  - Military research (nuclear weapons, cryptography)
  - Scientific research
  - Weather forecasting
  - Oil exploration
  - Industrial design (car crash simulation)
  - Bioinformatics
  - Cryptography

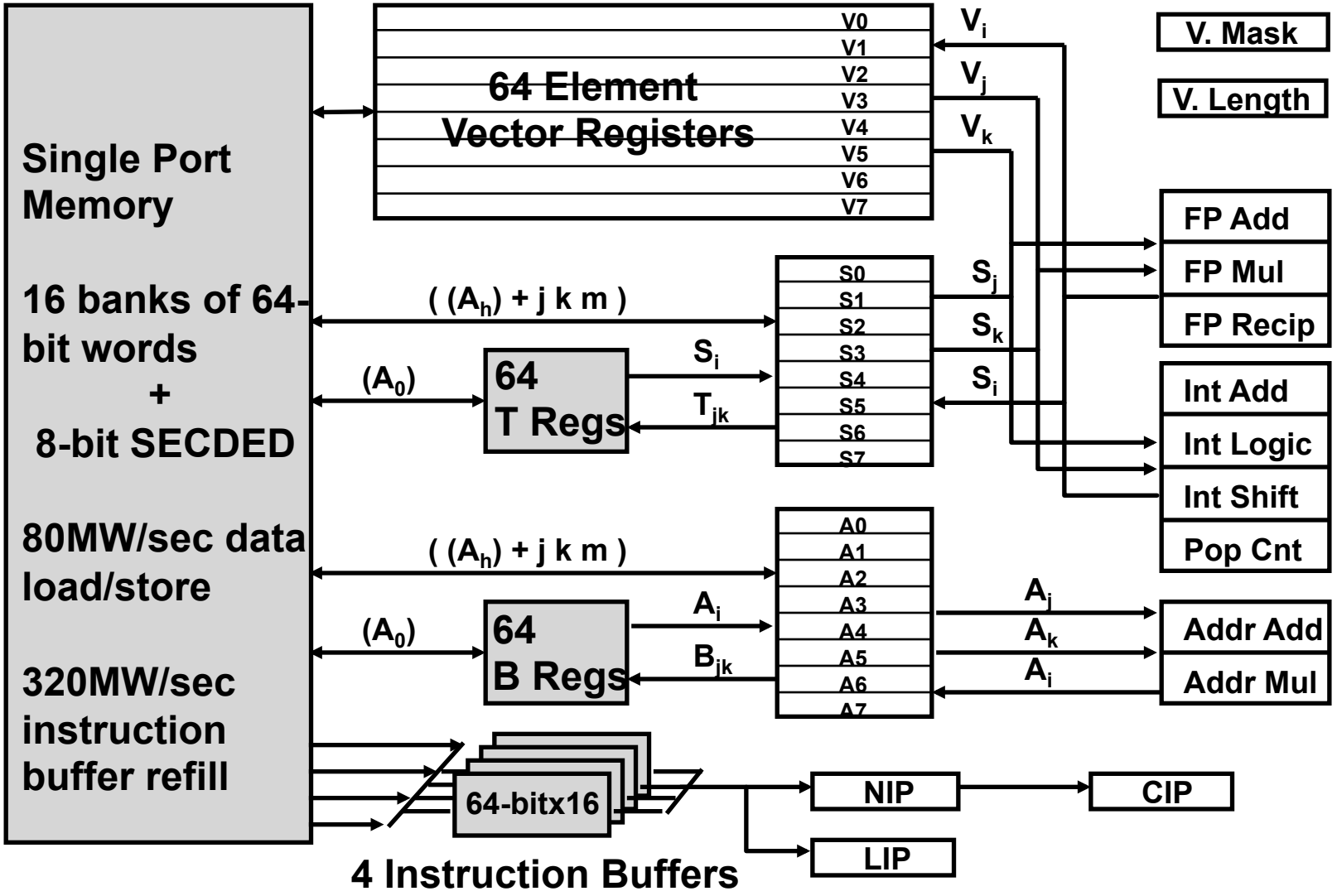
- All involve huge computations on large data sets

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

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)

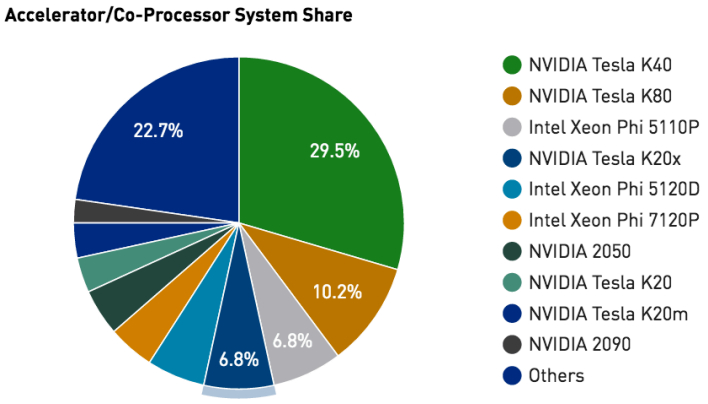


memory bank cycle 50 ns *processor cycle* 12.5 ns (80MHz)

Today's Supercomputers (www.top500.org)

- **MIMD and Hybrid**

MIMD/vector or MIMD/GPU



TOP 10 Sites for November 2016

For more information about the sites and systems in the list, click on the links or view the complete list.

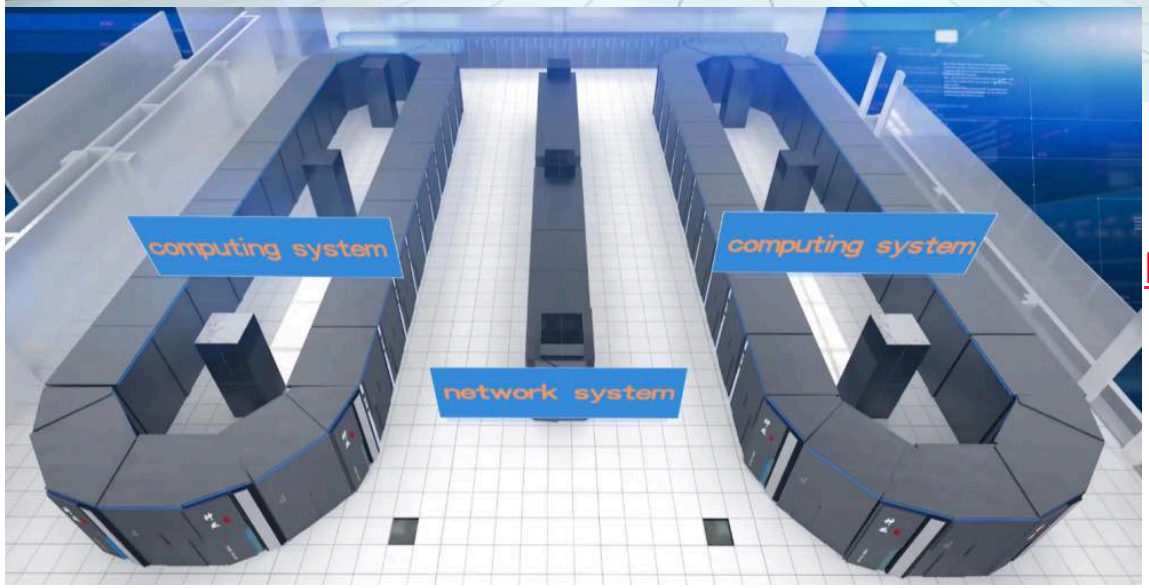
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
2	National Super Computer Center in Guangzhou China	**Tianhe-2 (MilkyWay-2)** - TH-IVB-FEP Cluster, Intel Xeon E5-2692 12C 2.200GHz, TH Express-2, Intel Xeon Phi 31S1P NUDT	3,120,000	33,862.7	54,902.4	17,808
3	DOE/SC/Oak Ridge National Laboratory United States	**Titan** - Cray XK7 , Opteron 6274 16C 2.200GHz, Cray Gemini interconnect, NVIDIA K20x Cray Inc.	560,640	17,590.0	27,112.5	8,209
	DOE/NNSA/LLNL United States	**Sequoia** - BlueGene/Q, Power BQC 16C 1.60 GHz, Custom IBM	1,572,864	17,173.2	20,132.7	7,890
	DOE/SC/LBNL/NERSC United States	**Cori** - Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries interconnect Cray Inc.	622,336	14,014.7	27,880.7	3,939
	Joint Center for Advanced High Performance Computing Japan	**Oakforest-PACS** - PRIMERGY CX1640 M1, Intel Xeon Phi 7250 68C 1.4GHz, Intel Omni-Path Fujitsu	556,104	13,554.6	24,913.5	2,719
	RIKEN Advanced Institute for Computational Science (AICS) Japan	K computer, SPARC64 VIIIfx 2.0GHz, Tofu interconnect Fujitsu	705,024	10,510.0	11,280.4	12,660
8	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	206,720	9,779.0	15,988.0	1,312
9	DOE/SC/Argonne National	**Mira** - BlueGene/Q, Power BQC 16C 1.60GHz,	786,432	8,586.6	10,066.3	3,945

#1 of TOP500 as of Nov 2016

TOP 10 Sites for November 2016

For more information about the sites and systems in the list, click on the links or view the complete list.

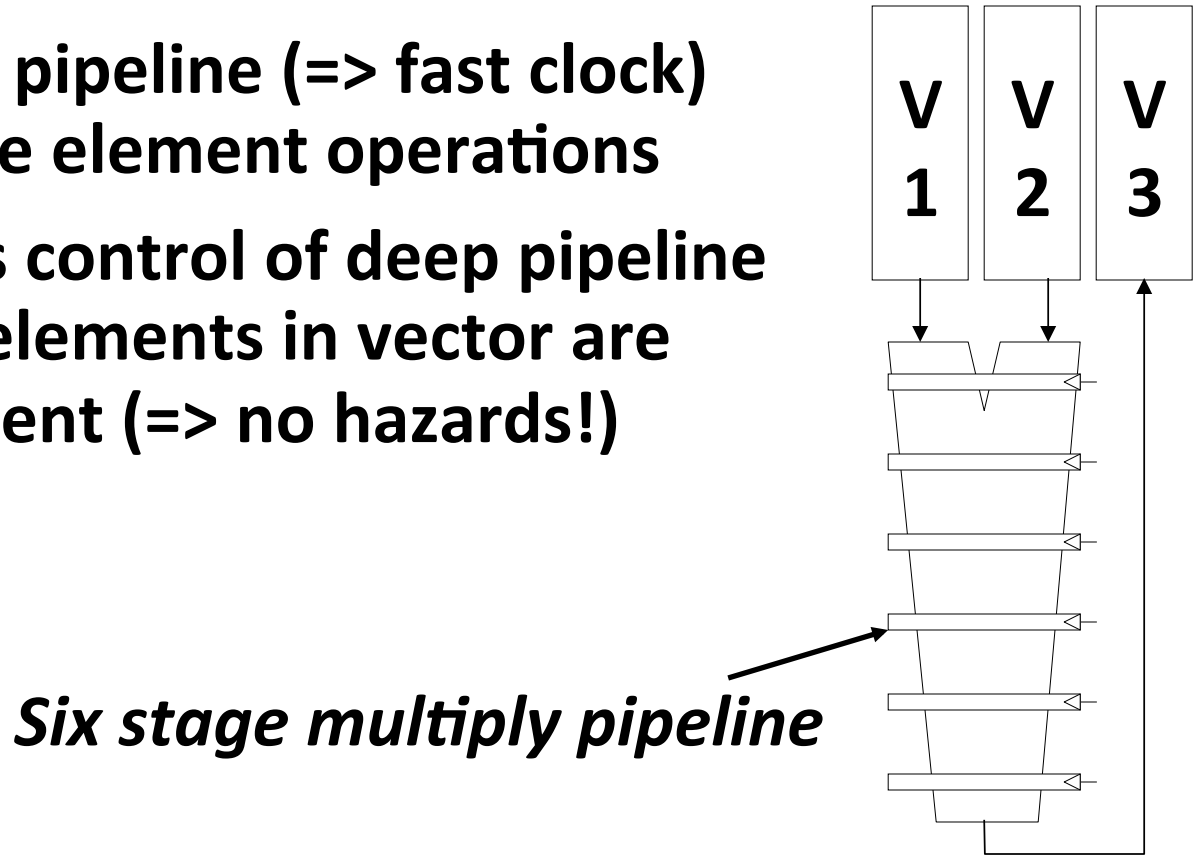
Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371



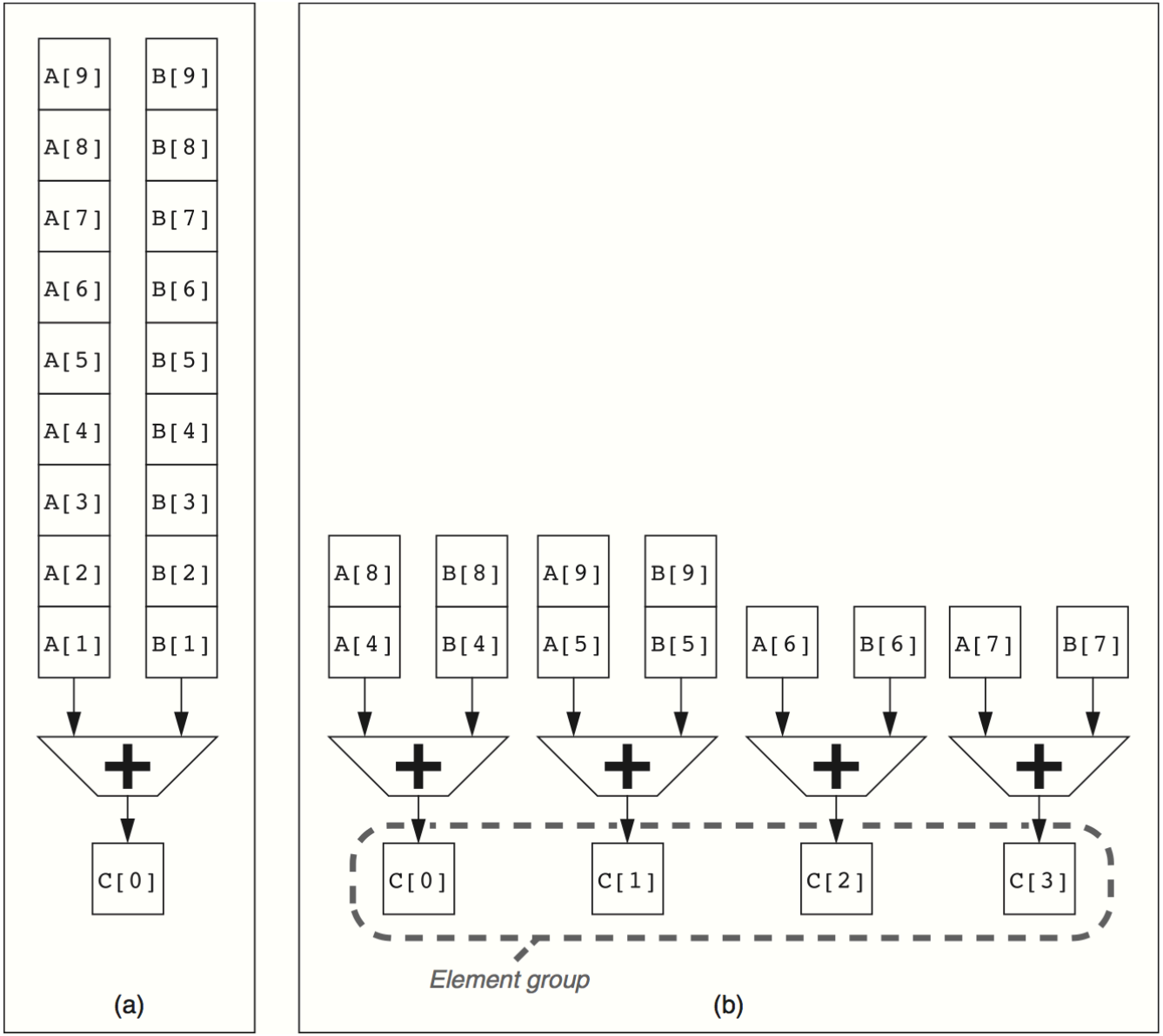
http://sc16.supercomputing.org/

Vector Arithmetic Execution

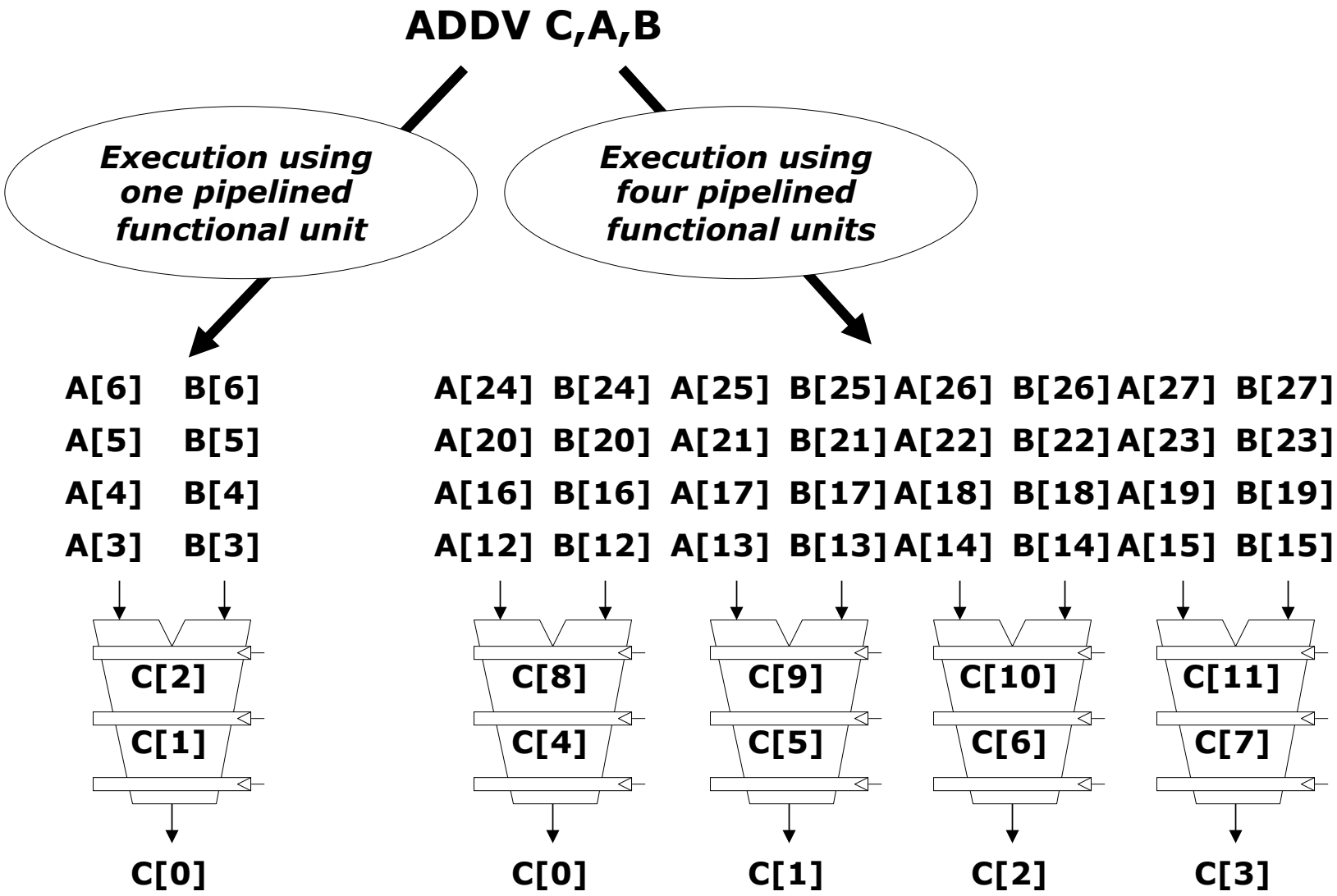
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



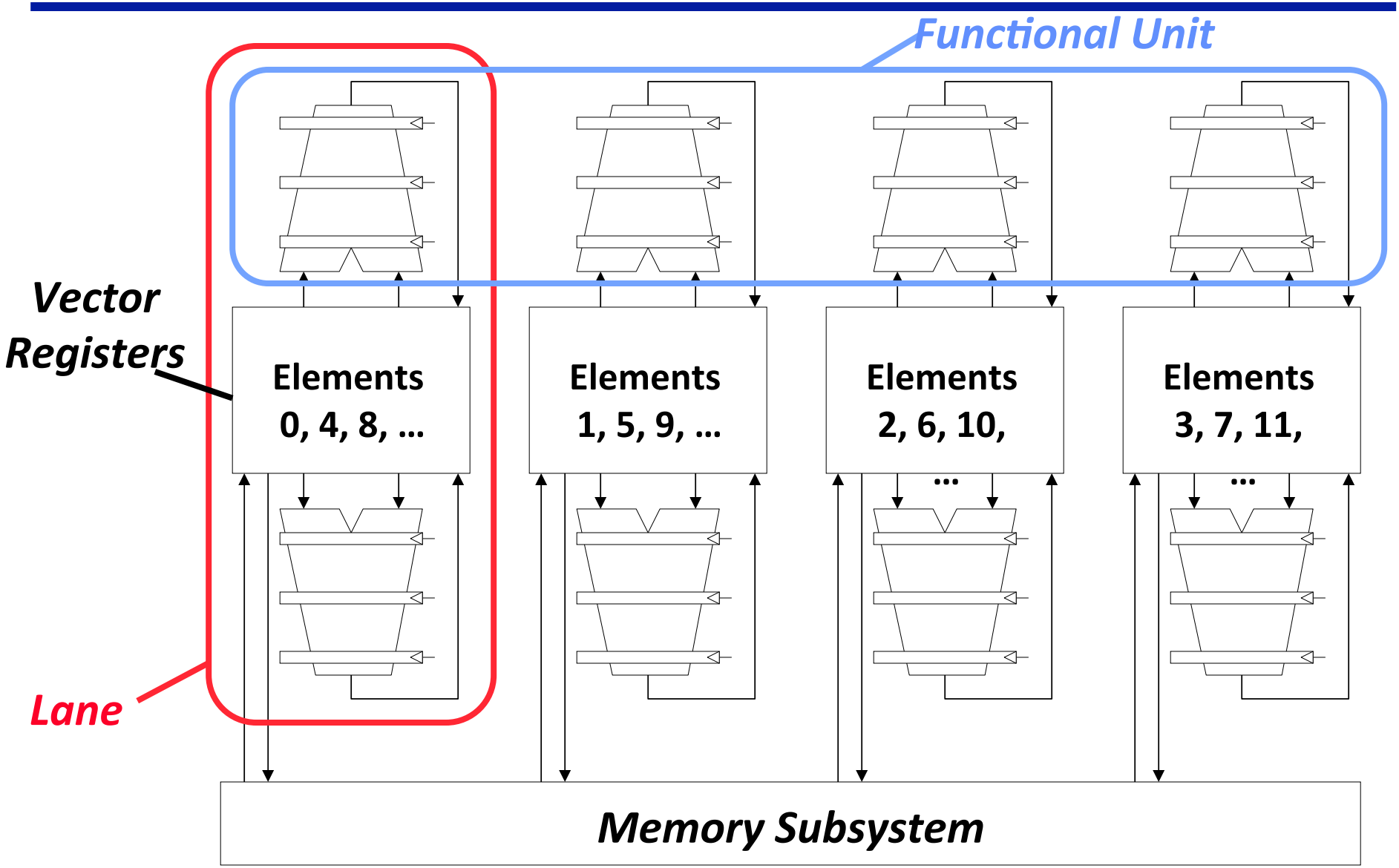
Vector Execution: Element Group



Vector Instruction Execution with Pipelined Functional Units

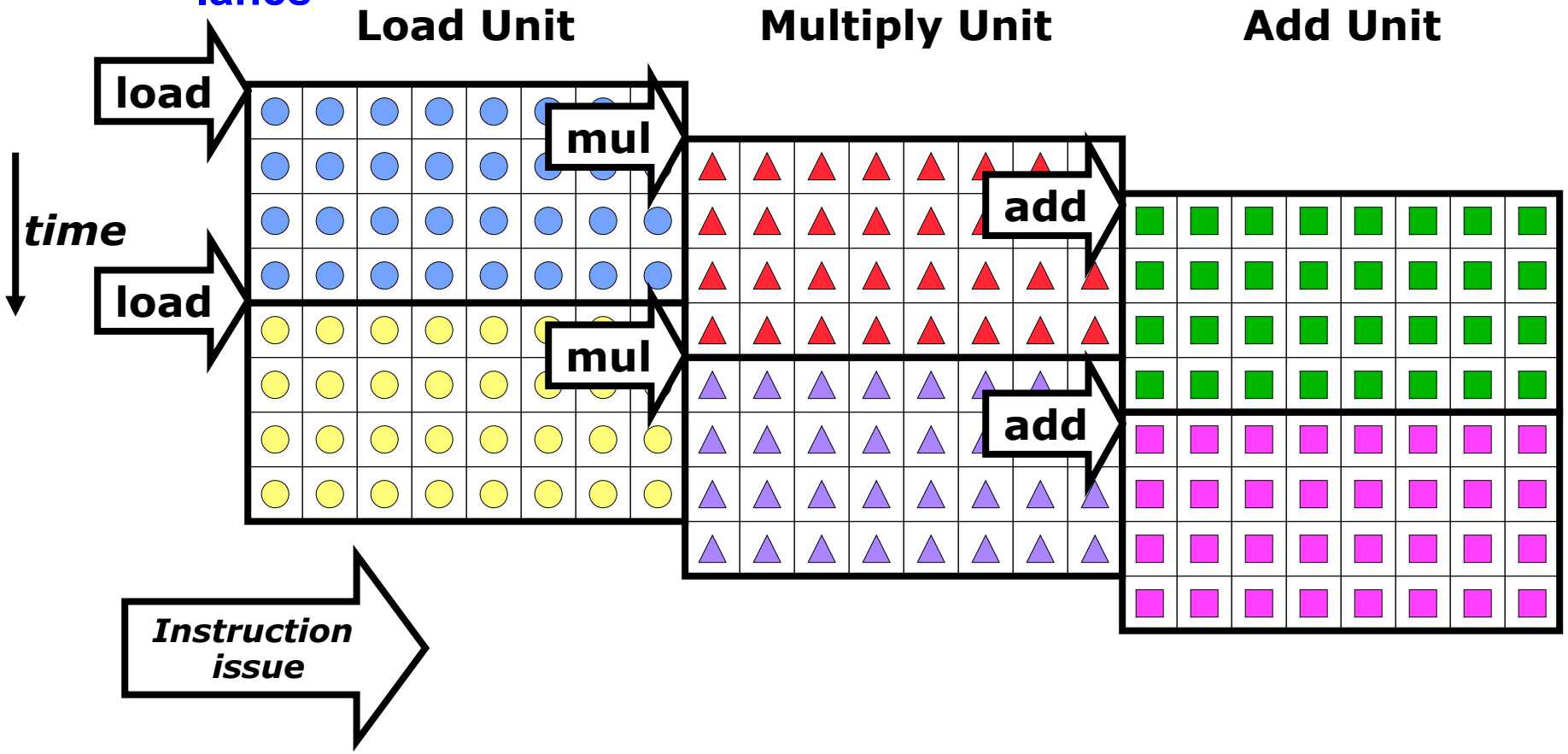


Vector Unit Structure (4 Lanes)



Vector Instruction Parallelism

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



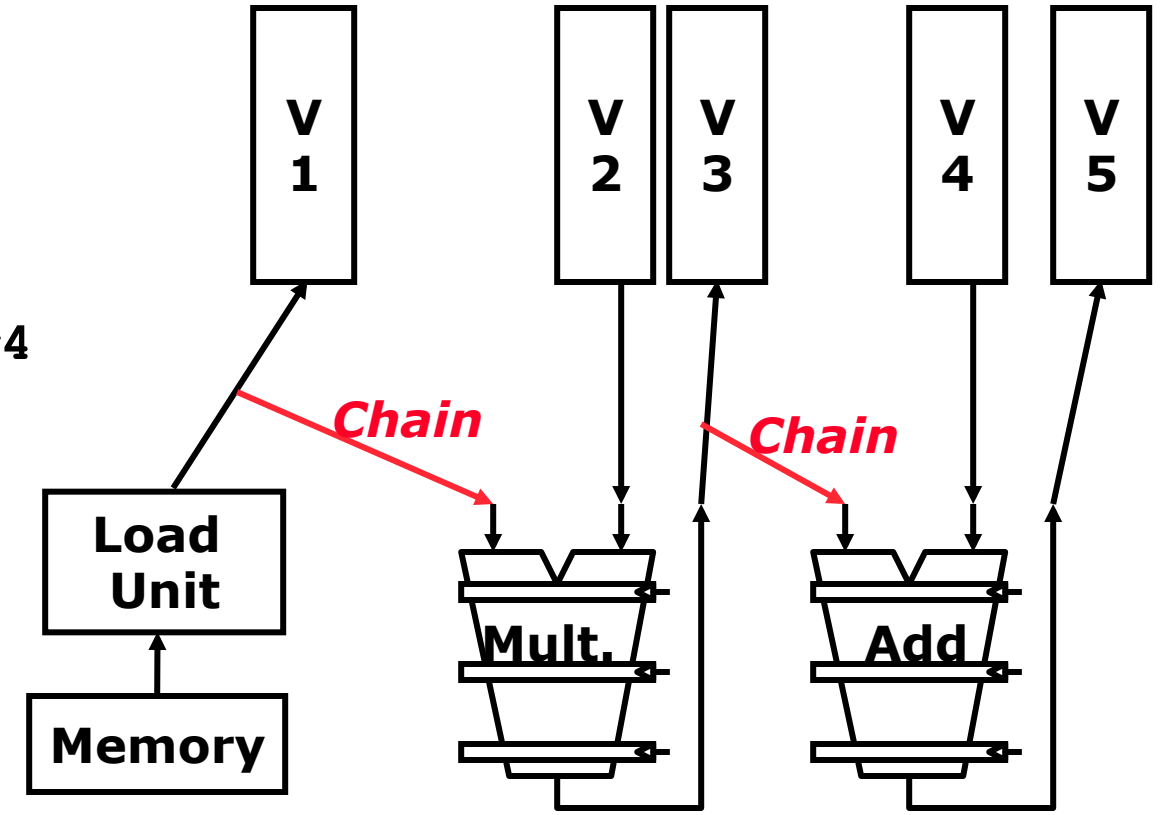
Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

Vector version of register bypassing

– introduced with Cray-1

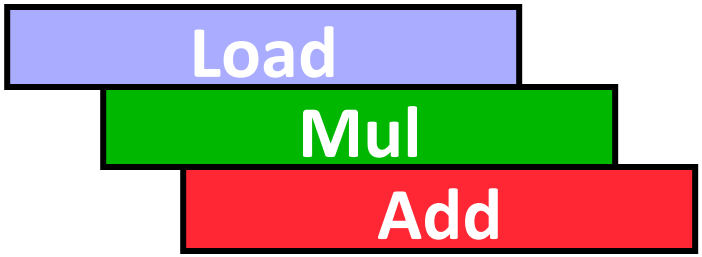
```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



Vector Chaining Advantage

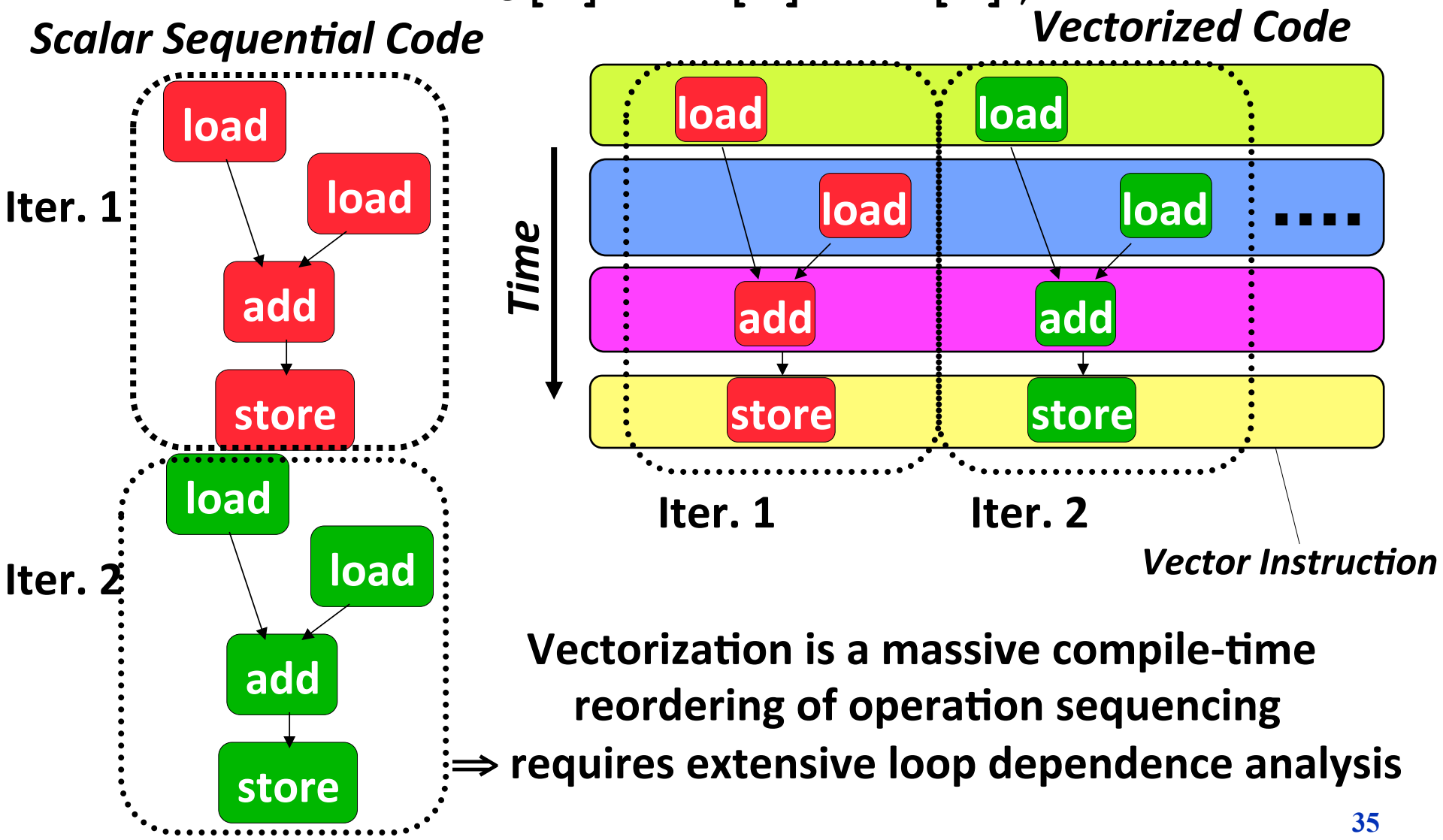
- Without chaining, must wait for last element of result to be written before starting dependent instruction

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



Automatic Code Vectorization

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



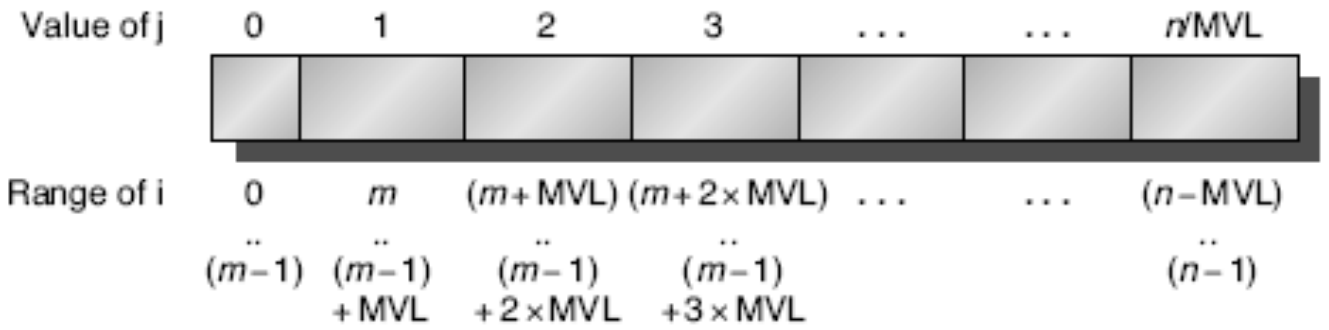
Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- Use *strip mining* for vectors over the maximum length (serialized version before vectorization by compiler)

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
    for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
        Y[i] = a * X[i] + Y[i] ; /*main operation*/
    low = low + VL; /*start of next vector*/
    VL = MVL; /*reset the length to maximum vector length*/
}
```

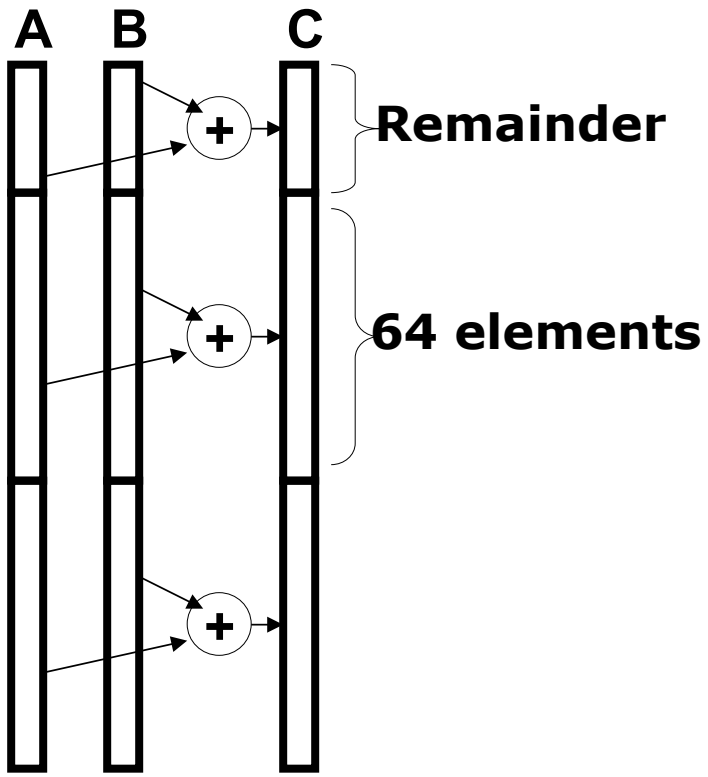


Vector Stripmining

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 Mask Registers

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

- Use vector mask register to "disable" elements (1 bit per element):

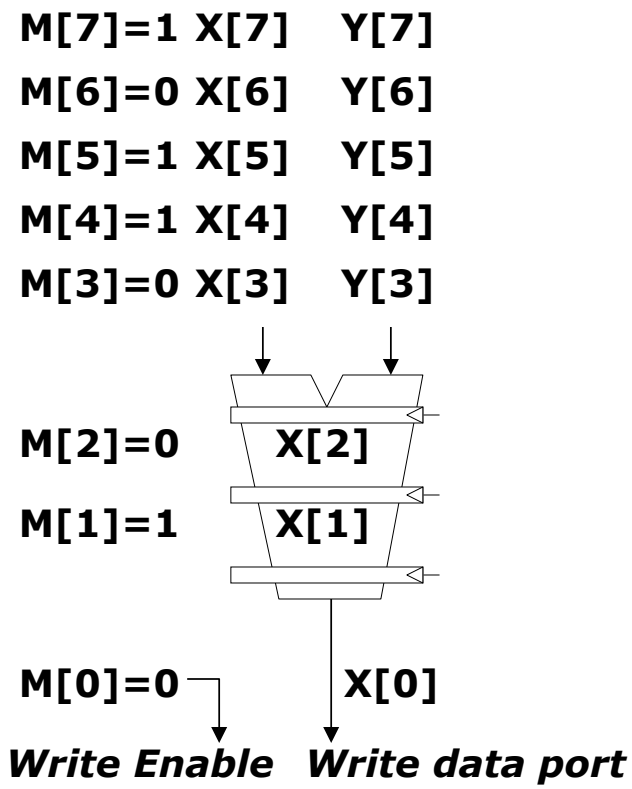
```
LV          V1,Rx       ;load vector X into V1
LV          V2,Ry       ;load vector Y
L.D         F0,#0       ;load FP zero into F0
SNEVS.D     V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2    ;subtract under vector mask
SV          Rx,V1       ;store the result in X
```

- GFLOPS rate decreases!
  - Vector operation becomes bubble ("NOP") at elements where mask bit is clear

Masked Vector Instructions

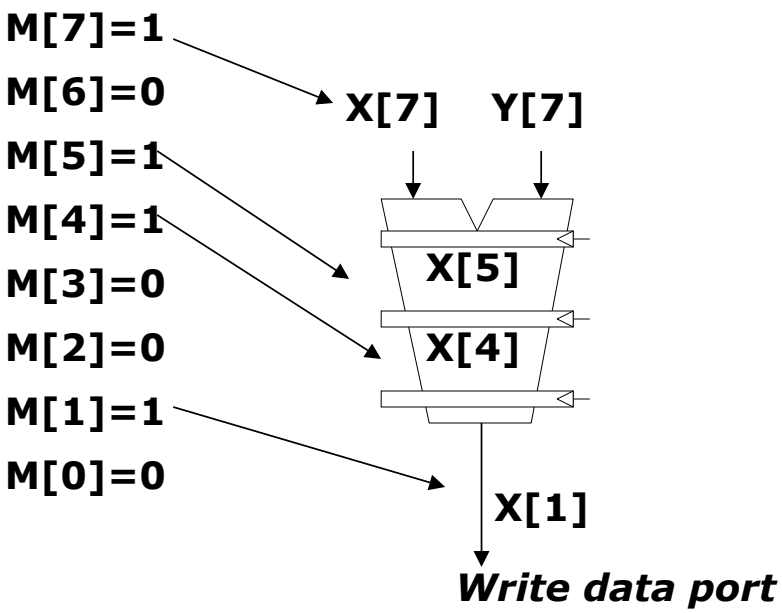
Simple Implementation

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



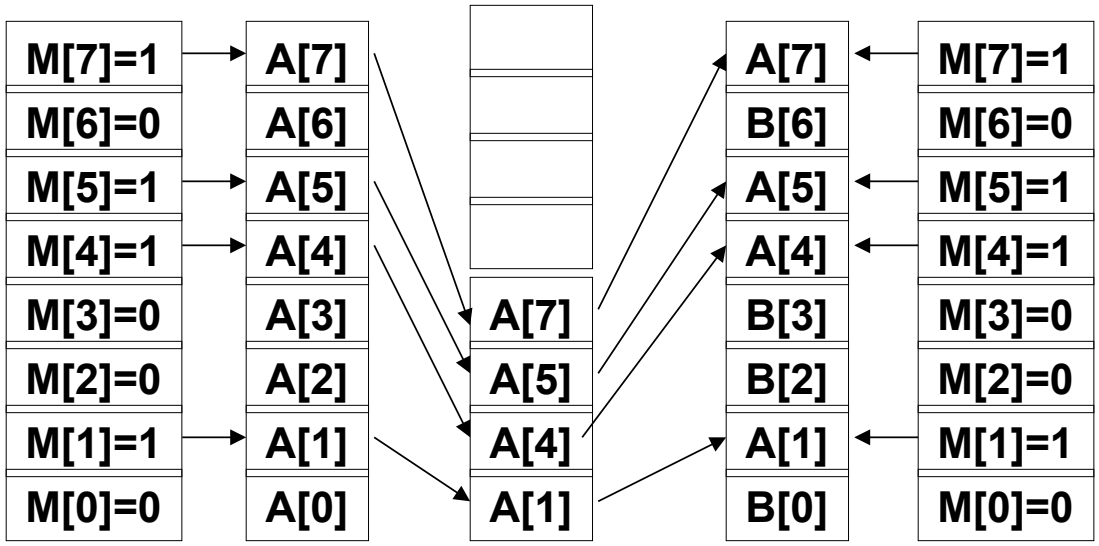
Density-Time Implementation

- 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

Stride

DGEMM (Double-Precision Matrix Multiplication)

```
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
  - Row-major: B: 1 double (8 bytes), and D: 100 doubles (800 bytes)
- Use *non-unit stride*
  - LDWS
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time

Scatter-Gather

- **Sparse matrix:**
  - **Non-zero values are compacted to a smaller value array (A[ ])**
  - **indirect array indexing, i.e. use an array to store the index to value array (K[ ])**

$$\begin{pmatrix} 1.0 & 0 & 5.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 3.0 & 0 & 0 & 0 & 0 & 11.0 & 0 \\ 0 & 0 & 0 & 0 & 9.0 & 0 & 0 & 0 \\ 0 & 0 & 6.0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 7.0 & 0 & 0 & 0 & 0 \\ 2.0 & 0 & 0 & 0 & 0 & 10.0 & 0 & 0 \\ 0 & 0 & 0 & 8.0 & 0 & 0 & 0 & 0 \\ 0 & 4.0 & 0 & 0 & 0 & 0 & 0 & 12.0 \end{pmatrix}$$

```
for (i = 0; i < n; i=i+1)
        A[K[i]] = A[K[i]] + C[M[i]];
```

- **Use index vector:**

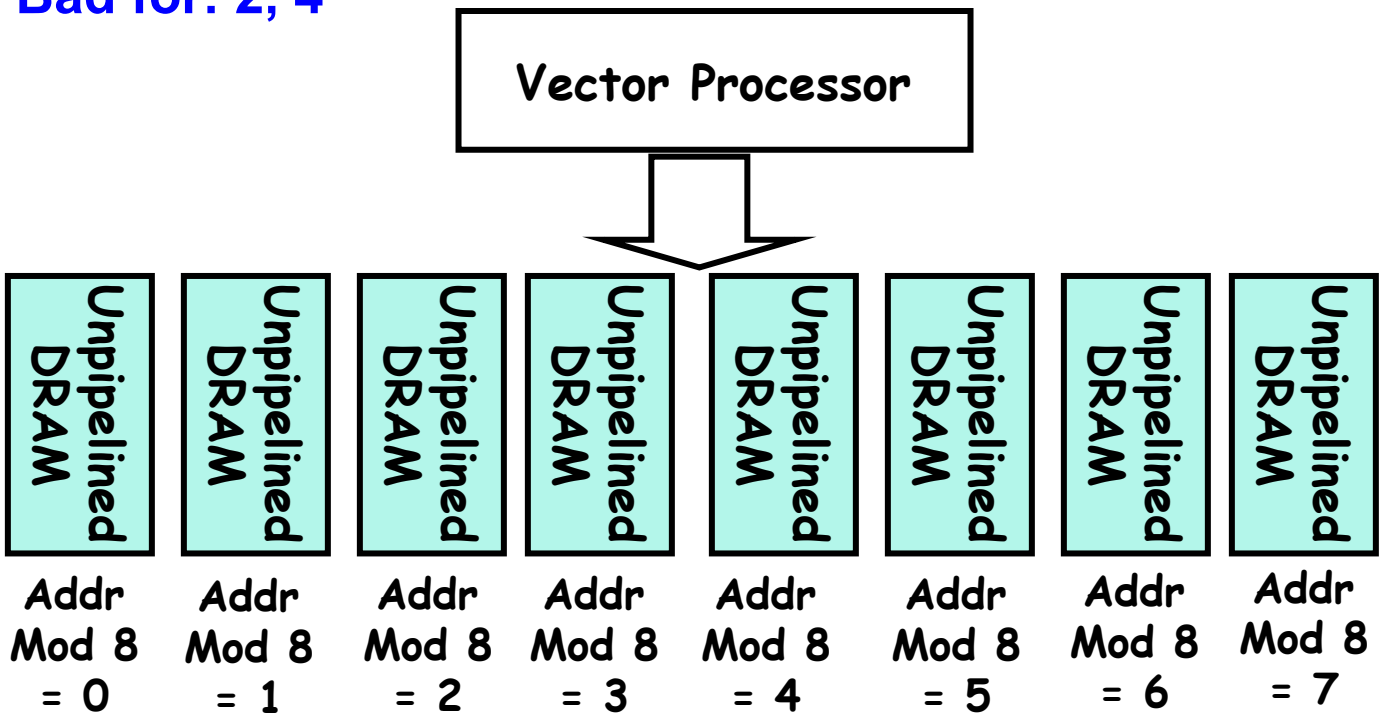
```
LV        Vk, Rk          ;load K
LVI       Va, (Ra+Vk)     ;load A[K[]]
LV        Vm, Rm          ;load M
LVI       Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D   Va, Va, Vc      ;add them
SVI       (Ra+Vk), Va     ;store A[K[]]
```

Memory operations

- Load/store operations move groups of data between registers and memory
  - Increased mem/instr ratio (intensity)
- 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

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



Avoiding Bank Conflicts

- Lots of banks

```
int x[256][512];
    for (j = 0; j < 512; j = j+1)
         for (i = 0; i < 256; i = i+1)
              x[i][j] = 2 * x[i][j];
```

- Even with 128 banks, since 512 is multiple of 128, conflict on word accesses
- SW: loop interchange or declaring array not power of 2 ("array padding")
- HW: Prime number of banks
  - bank number = address mod number of banks
  - address within bank = address / number of words in bank
  - modulo & divide per memory access with prime no. banks?
  - address within bank = address mod number words in bank
  - bank number? easy if 2^N words per bank

Finding Bank Number and Address within a bank

- Problem: Determine the number of banks, N_b and the number of words in each bank, N_w, such that:
  - given address x, it is easy to find the bank where x will be found, B(x), and the address of x within the bank, A(x).
  - for any address x, B(x) and A(x) are unique
  - the number of bank conflicts is minimized
- Solution: Use the Chinese remainder theorem to determine B(x) and A(x):

  $B(x) = x \text{ MOD } N_b$
  $A(x) = x \text{ MOD } N_w$ where N_b and N_w are co-prime (no factors)
  - Chinese Remainder Theorem shows that B(x) and A(x) unique.
- Condition allows N_w to be power of two (typical) if N_b is prime of form 2^m-1.
- Simple (fast) circuit to compute (x mod N_b) when $N_b = 2^m$-1:
  - Since $2^k = 2^{k-m}(2^m-1) + 2^{k-m} \Rightarrow 2^k \text{ MOD } N_b = 2^{k-m} \text{ MOD } N_b = \ldots = 2^j$ with $j < m$
  - And, remember that: (A+B) MOD C = [(A MOD C)+(B MOD C)] MOD C
  - for every power of 2, compute single bit MOD (in advance)
  - B(x) = sum of these values MOD N_b
    (low complexity circuit, adder with ~ m bits)

Conclusion

- Vector is alternative model for exploiting ILP
  - 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