Friday 11/09 Class and Next Week

- Friday 11/09, 9:00AM, 2A31 (Same classroom)
  - I will post assignment 4 by today
- Next week:
  - Monday: No class
  - Wednesday: regular time via https://webconnect.sc.edu/csce513/
    - » Check https://passlab.github.io/CSCE513/OnlineAdobeConnect.html for details how to make your computer ready
- The week after next (11/19 week)
  - Monday 11/19 regular class
  - Wednesday 11/21, No class because of Thanksgiving
- Then there are two more weeks for this course.
- Final Exam: 9:00AM 12/12 Wednesday

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

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

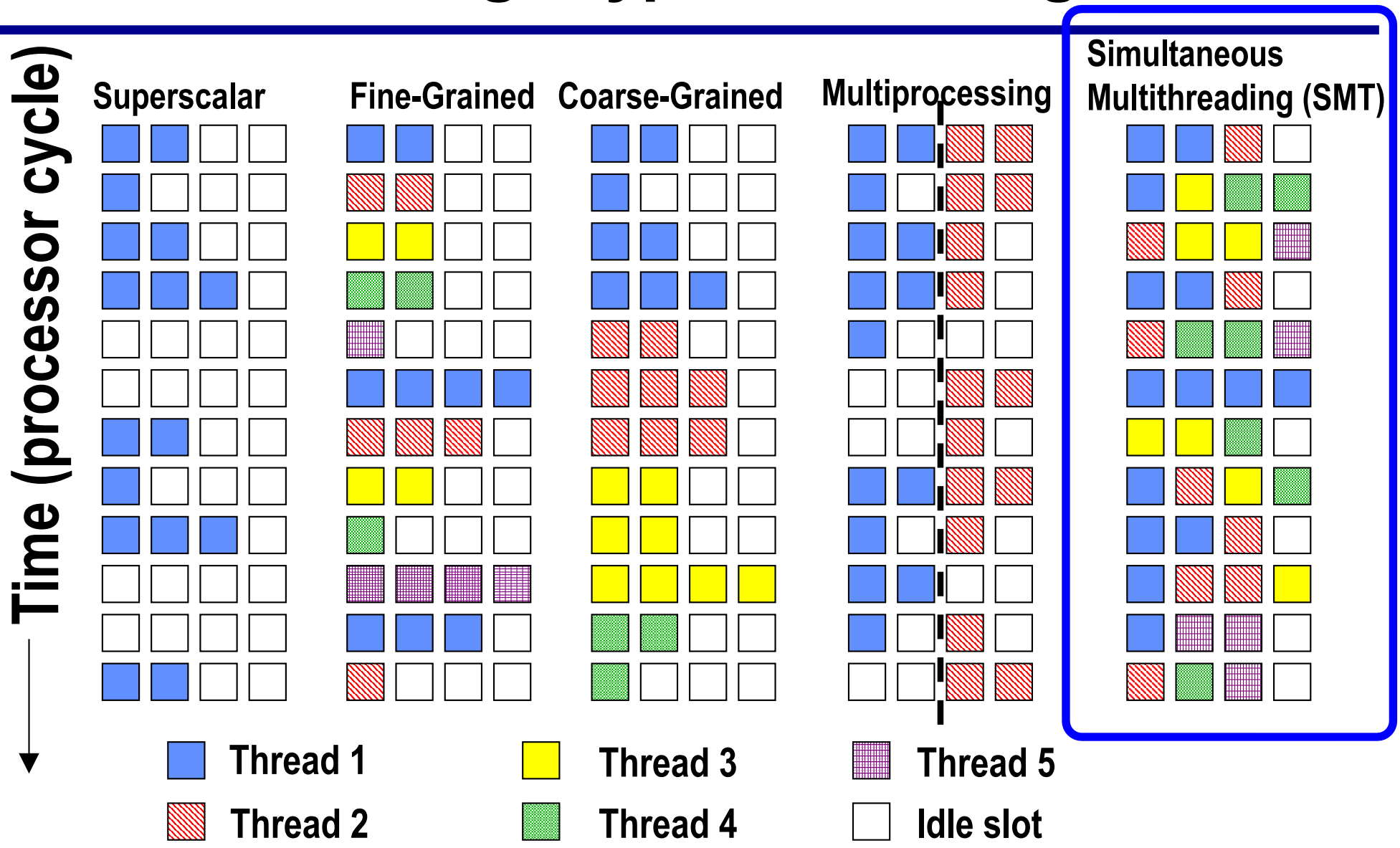
Very Important Terms for Instruction-Level Parallelism

- **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

Though mostly invented earlier, these techniques are still widely used today, in from embedded CPUs to server/desktop CPUs.

Multithreading: Hyper-Threading = SMT



SMT/HTT performance improvement is NOT significant!

https://en.wikipedia.org/wiki/Hyper-threading

CSE 564 Class Contents

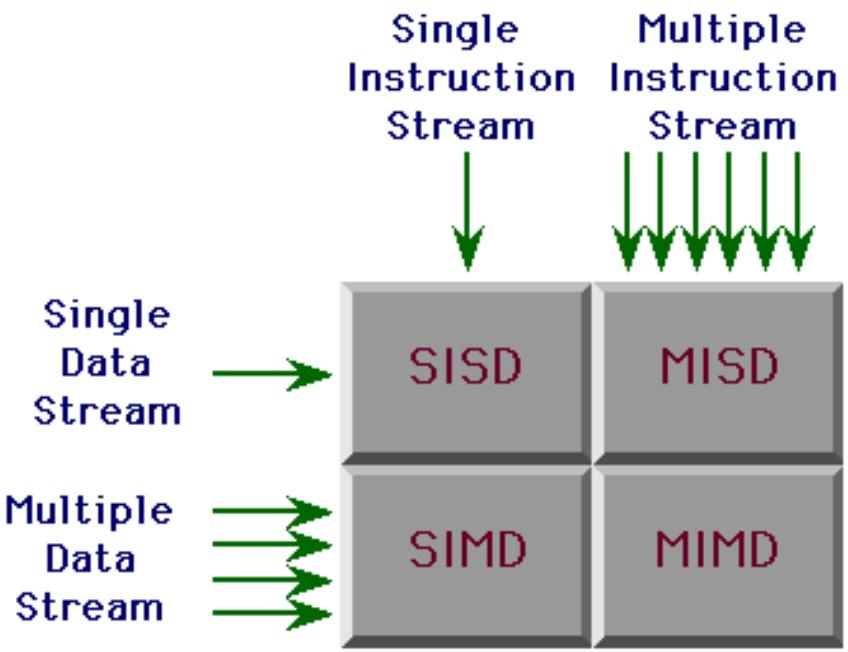
- Introduction to Computer Architecture (CA)
- Quantitative Analysis, Trend and Performance of CA
  - Chapter 1
- Instruction Set Principles and Examples
  - Appendix A
- Pipelining and Implementation, RISC-V ISA and Implementation
  - Appendix C, RISC-V (riscv.org) and UCB RISC-V impl
- Memory System (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
- Domain-specific architecture
  - Chapter 7

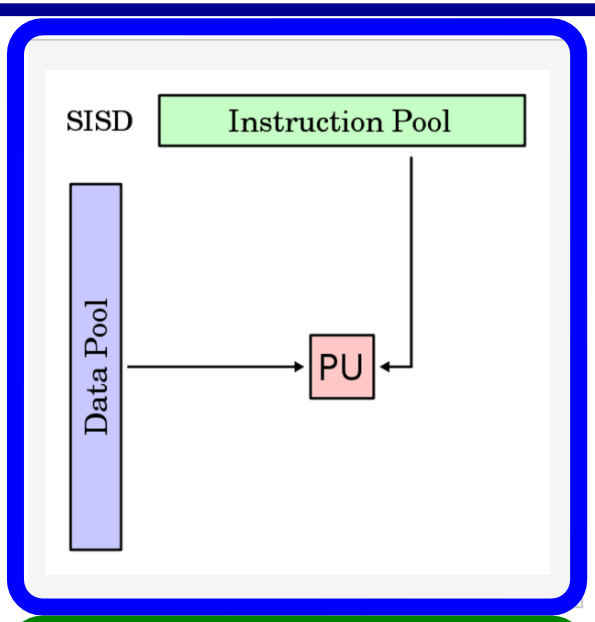
Topics for Data Level Parallelism (DLP)

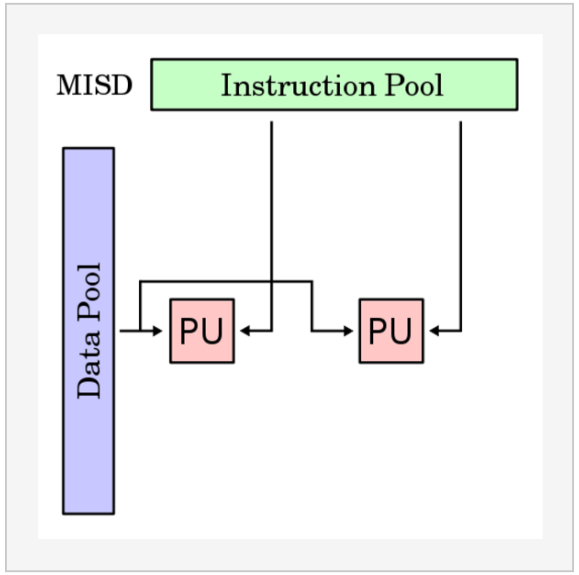
- Parallelism (centered around …)
  - Instruction Level Parallelism
  - Data Level Parallelism
  - Thread Level Parallelism

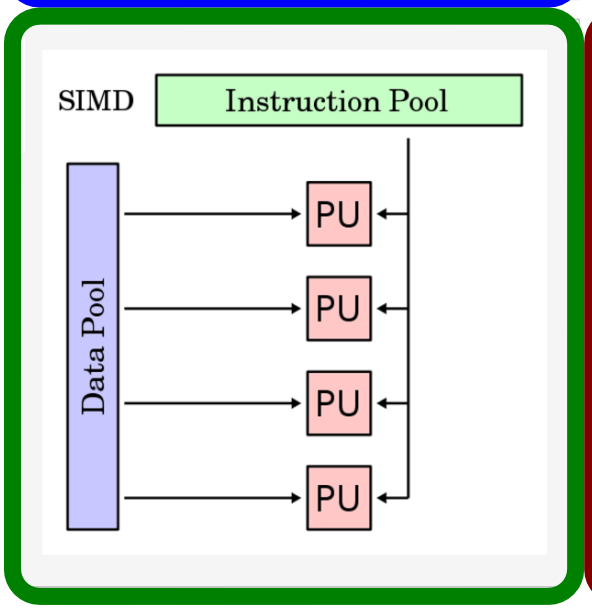
- DLP Introduction and Vector Architecture
  - 4.1, 4.2
- SIMD Instruction Set Extensions for Multimedia
  - 4.3
- Graphical Processing Units (GPU)
  - 4.4
- GPU and Loop-Level Parallelism and Others
  - 4.4, 4.5

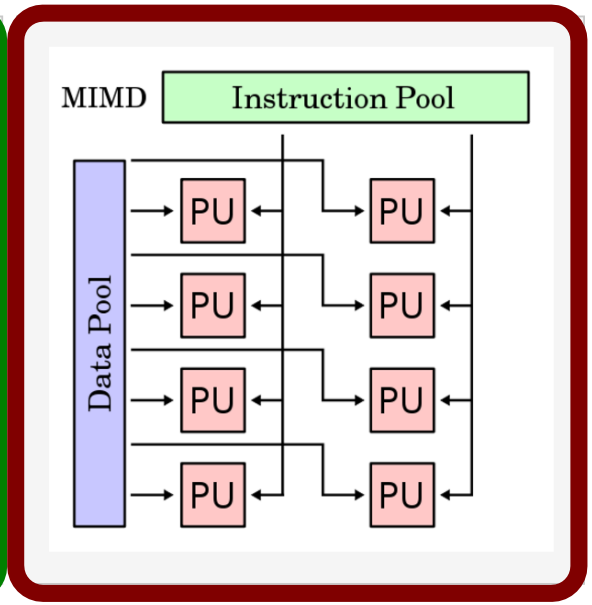
Flynn's Taxonomy for Classifying CA











Michael J. Flynn: http://arith.stanford.edu/~flynn/

Flynn's Classification (1966)

Broad classification of parallel computing systems

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

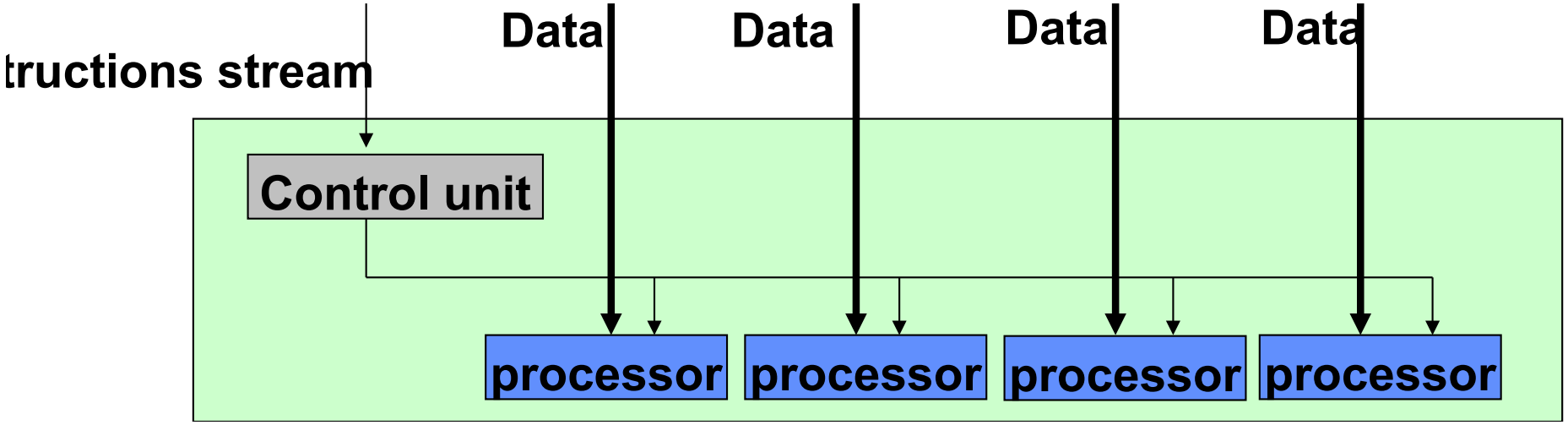
- **SISD: Single Instruction, Single Data**
  - conventional uniprocessor, a single core
- **SIMD: Single Instruction, Multiple Data**
  - one instruction stream, multiple data paths
  - distributed memory SIMD (MPP, DAP, CM-1&2, Maspar)
  - shared memory SIMD (STARAN, vector computers)
- **MIMD: Multiple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

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

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors

```
for (i=999; i>=0; i=i–1)
        x[i] = x[i] + s;
```

- 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



Hardware Implementation for SIMD|Data-Level Parallelism

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

- E.g. x86 processors → MIMD + SIMD
  - Expect two additional cores per chip per year (MIMD)
  - Each core has SIMD, and SIMD width double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

Vector Architecture

VLIW vs Vector

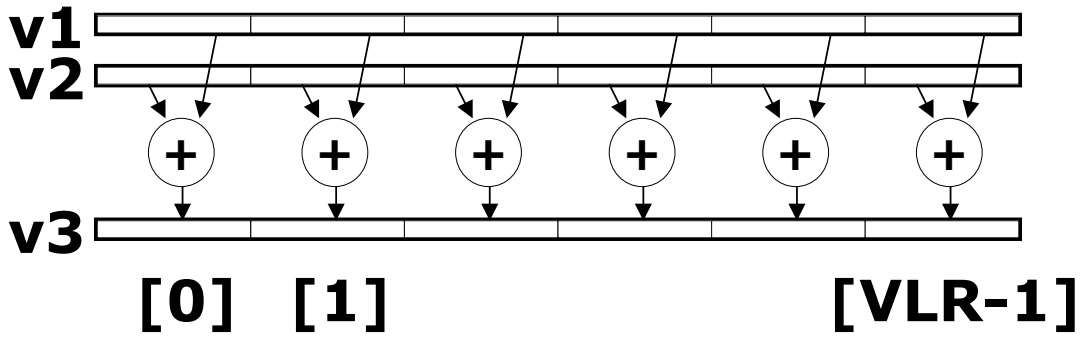
- VLIW takes advantage of instruction level parallelism (ILP) by specifying multiple (different) instructions to execute in parallel

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

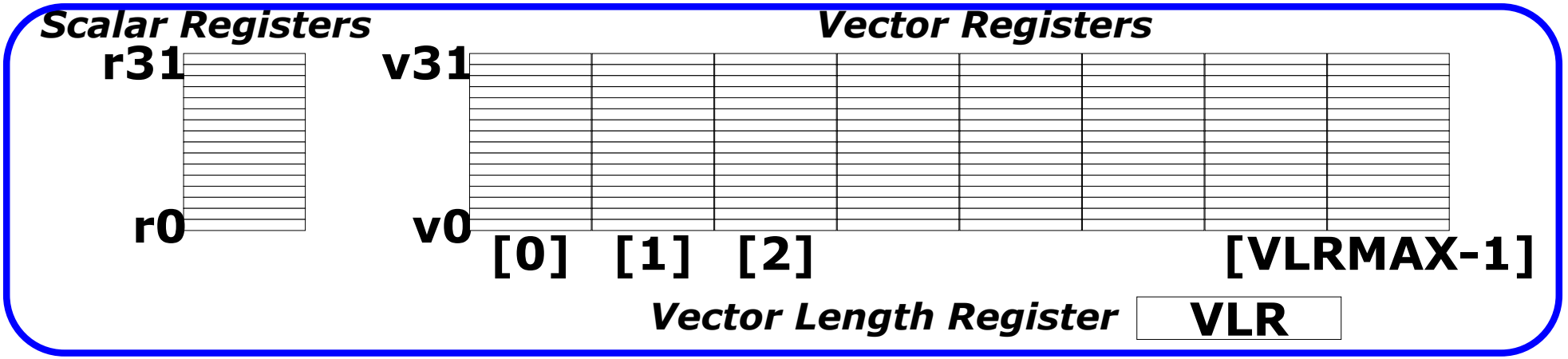
- Vector architectures perform the same operation on multiple data elements – single instruction
  - Data-level parallelism

Vector Arithmetic Instructions

ADDV v3, v1, v2

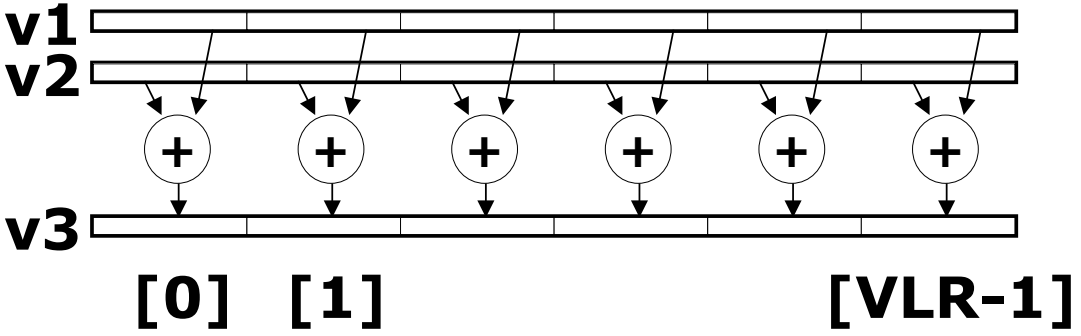


Vector Programming Model



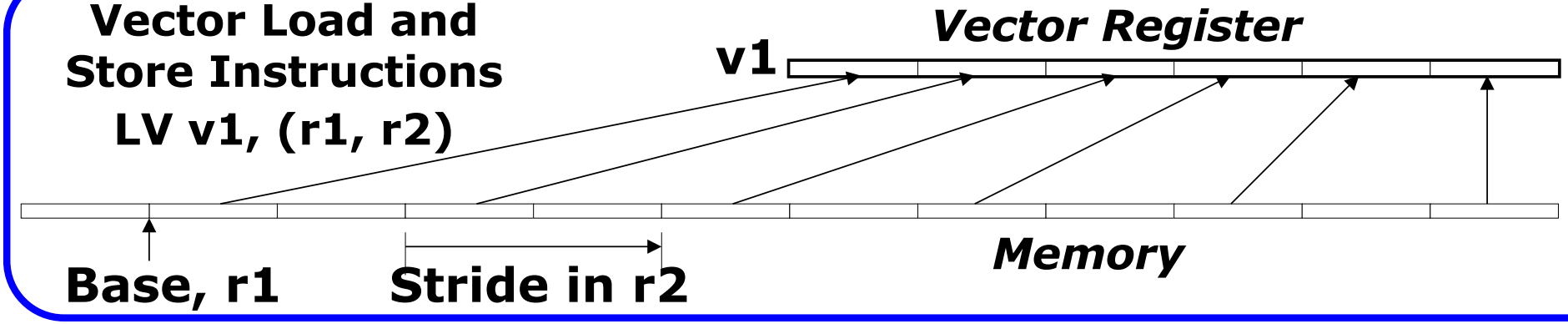
Vector Arithmetic Instructions

ADDV v3, v1, v2



Vector Load and Store Instructions

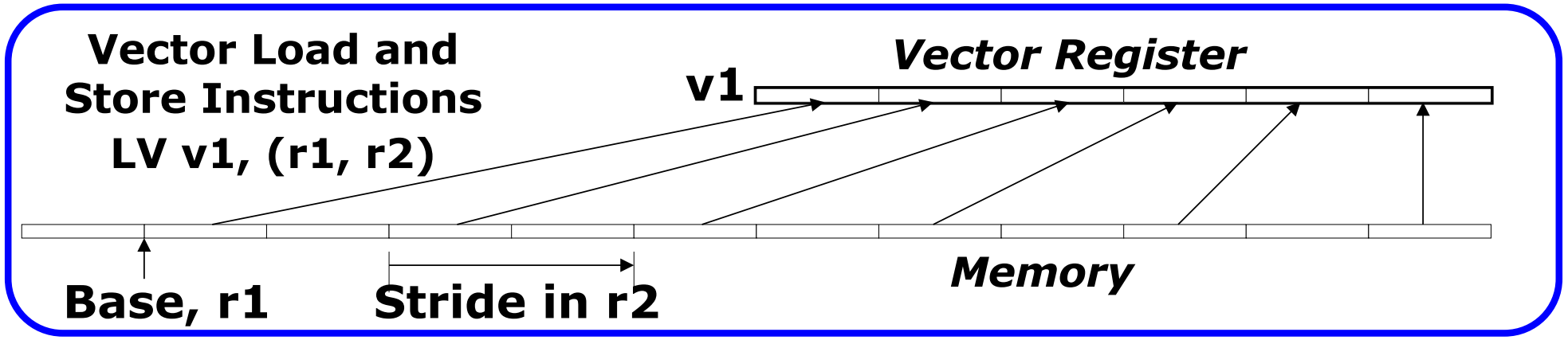
LV v1, (r1, r2)



Control Information

- **Vector length (VL) register limits the max number of elements to be processed by a vector instruction**
  - **VL 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

- RV64V
- 32 32x8-byte vector registers
- all the functional units are vector functional units.
- To sustain requests of high memory bandwidth
  - 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.

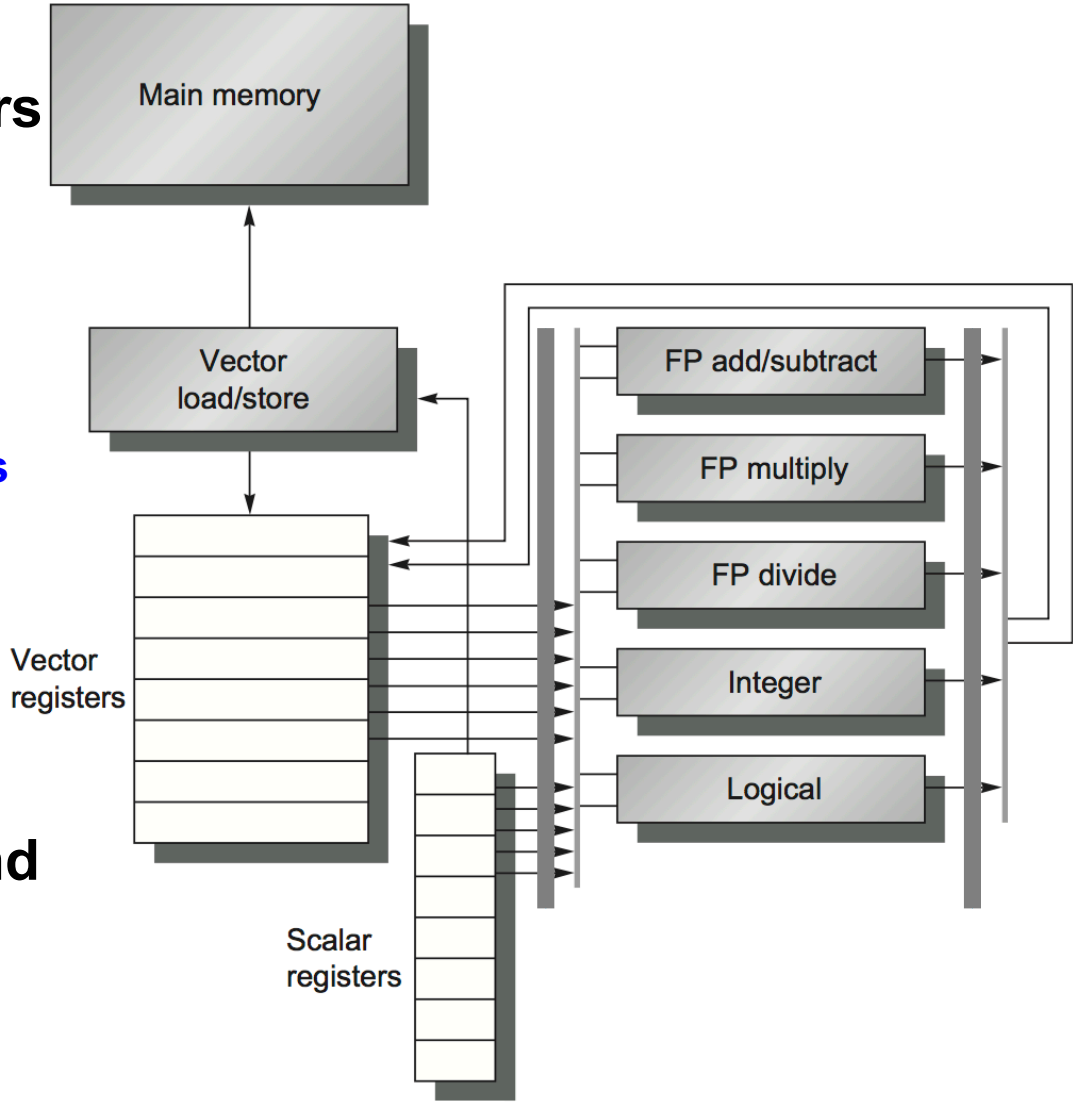


Figure 4.1 The basic structure of a vector architecture, RV64V, which includes RISC-V scalar architecture. There are also 32 vector registers, and all the functional ur are vector functional units. The vector and scalar registers have a significant number read and write ports to allow multiple simultaneous vector operations. A set of crossb

RV64V Vector Instructions

- **Suffix**
  - V suffix
  - VS suffix
- **Load/Store**
  - vld
  - vst
- **Registers**
  - V registers
  - VL (vector length register)
  - Predicate

Mnemonic	Name	Description
vadd	ADD	Add elements of V[rs1] and V[rs2], then put each result in V[rd]
vsub	SUBtract	Subtract elements of V[rs2] frpm V[rs1], then put each result in V[rd]
vmul	MULtiply	Multiply elements of V[rs1] and V[rs2], then put each result in V[rd]
vdiv	DIVide	Divide elements of V[rs1] by V[rs2], then put each result in V[rd]
vrem	REMainder	Take remainder of elements of V[rs1] by V[rs2], then put each result in V[rd]
vsqrt	SQuare RooT	Take square root of elements of V[rs1], then put each result in V[rd]
vsll	Shift Left	Shift elements of V[rs1] left by V[rs2], then put each result in V[rd]
vsrl	Shift Right	Shift elements of V[rs1] right by V[rs2], then put each result in V[rd]
vsra	Shift Right Arithmetic	Shift elements of V[rs1] right by V[rs2] while extending sign bit, then put each result in V[rd]
vxor	XOR	Exclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vor	OR	Inclusive OR elements of V[rs1] and V[rs2], then put each result in V[rd]
vand	AND	Logical AND elements of V[rs1] and V[rs2], then put each result in V[rd]
vsgnj	SiGN source	Replace sign bits of V[rs1] with sign bits of V[rs2], then put each result in V[rd]
vsgnjn	Negative SiGN source	Replace sign bits of V[rs1] with complemented sign bits of V[rs2], then put each result in V[rd]
vsgnjx	Xor SiGN source	Replace sign bits of V[rs1] with xor of sign bits of V[rs1] and V[rs2], then put each result in V[rd]
vld	Load	Load vector register V[rd] from memory starting at address R[rs1]
vlds	Strided Load	Load V[rd] from address at R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vldx	Indexed Load (Gather)	Load V[rs1] with vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vst	Store	Store vector register V[rd] into memory starting at address R[rs1]
vsts	Strided Store	Store V[rd] into memory at address R[rs1] with stride in R[rs2] (i.e., R[rs1]+i × R[rs2])
vstx	Indexed Store (Scatter)	Store V[rs1] into memory vector whose elements are at R[rs2]+V[rs2] (i.e., V[rs2] is an index)
vpeq	Compare =	Compare elements of V[rs1] and V[rs2]. When equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpne	Compare !=	Compare elements of V[rs1] and V[rs2]. When not equal, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vplt	Compare <	Compare elements of V[rs1] and V[rs2]. When less than, put a 1 in the corresponding 1-bit element of p[rd]; otherwise, put 0
vpxor	Predicate XOR	Exclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpor	Predicate OR	Inclusive OR 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
vpand	Predicate AND	Logical AND 1-bit elements of p[rs1] and p[rs2], then put each result in p[rd]
setvl	Set Vector Length	Set vl and the destination register to the smaller of mvl and the source regsiter

Highlight of RV64V Vector Instructions

- All are R-format instruction
- .vv and .vs|.sv operands
  - .vv: Vector-vector operands; .vs|.sv: Vector-scalar operands
- Vector load and store which loads or stores an entire vector
  - 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.
  - vlds/vsts: for stride load/store; vldx/vstx: indexed load/store
- Vector-length register vl is used when the natural vector length is not equal to mvl
- Vector-type register vctype records register types
- Predicate registers pi are used when loops involve IF statements.
- We'll see them in the following example:

Dynamic Register Typing in RV64V

- A vector register has 32 64-bit elements
  - Or 128 16- bit elements, and even 256 8-bit elements are equally valid views.

Integer	8, 16, 32, and 64 bits	Floating point	16, 32, and 64 bits

Figure 4.3 Data sizes supported for RV64V assuming it also has the single- and double-precision floating-point extensions RVS and RVD. Adding RVV to such a

- Associate a data type and data size with each vector register using vctype register
  - Existing and normal approach is that the instruction supplying the type information, but not in RV64V

DAXPY(Y = a * X + Y, 32 elements) in RV64G and RV64V

```
double a, X[], Y[]; // 8-byte per element
for (i=0; i<32; i++)
  Y[i] = a* X[i] + Y[i];
```

The starting addresses of X and Y are in X5 and X6, respectively

- # instrs:
  - 8 vs 258
- Pipeline stalls
  - 32x higher by RV64G
- Vector *chaining* (forwarding)
  - Per each vector element
  - v0→v1→v2→v3

```
        fld     f0,a              # Load scalar a
        addi    x28,x5,#256       # Last address to load
Loop:   fld     f1,0(x5)          # Load X[i]
        fmul.d  f1,f1,f0          # a × X[i]
        fld     f2,0(x6)          # Load Y[i]
        fadd.d  f2,f2,f1          # a × X[i] + Y[i]
        fsd     f2,0(x6)          # Store into Y[i]
        addi    x5,x5,#8          # Increment index to X
        addi    x6,x6,#8          # Increment index to Y
        bne     x28,x5,Loop       # Check if done
```

```
vsetdcfg  4*FP64          # Enable 4 DP FP vregs
fld       f0,a            # Load scalar a
vld       v0,x5           # Load vector X
vmul      v1,v0,f0        # Vector-scalar mult
vld       v2,x6           # Load vector Y
vadd      v3,v1,v2        # Vector-vector add
vst       v3,x6           # Store the sum
vdisable                  # Disable vector regs
```

DAXPY(Y = a * X + Y, 32 elements) in RV64G and RV64V

```
double a, X[], Y[]; // 8-byte per element
for (i=0; i<32; i++)
  Y[i] = a* X[i] + Y[i];
```

The starting addresses of X and Y are in X5 and X6, respectively

```
vsetdcfg   4*FP64          # Enable 4 DP FP vregs
fld        f0,a            # Load scalar a
vld        v0,x5           # Load vector X
vmul       v1,v0,f0        # Vector-scalar mult
vld        v2,x6           # Load vector Y
vadd       v3,v1,v2        # Vector-vector add
vst        v3,x6           # Store the sum
vdisable                   # Disable vector regs
```

```
float a, X[]; double Y[];
for (i=0; i<32; i++)
  Y[i] = a* X[i] + Y[i];
```

```
vsetdcfg 1*FP32,3*FP64  # 1 32b, 3 64b vregs
flw      f0,a           # Load scalar a
vld      v0,x5          # Load vector X
vmul     v1,v0,f0       # Vector-scalar mult
vld      v2,x6          # Load vector Y
vadd     v3,v1,v2       # Vector-vector add
vst      v3,x6          # Store the sum
vdisable                # Disable vector regs
```

DAXPY(Y = a * X + Y, 32 elements) in RV64G and RV64V

```
double a, X[], Y[]; // 8-byte per element
for (i=0; i<32; i++)
  Y[i] = a* X[i] + Y[i];
```

The starting addresses of X and Y are in X5 and X6, respectively

```
vsetdcfg   4*FP64          # Enable 4 DP FP vregs
fld        f0,a            # Load scalar a
vld        v0,x5           # Load vector X
vmul       v1,v0,f0        # Vector-scalar mult
vld        v2,x6           # Load vector Y
vadd       v3,v1,v2        # Vector-vector add
vst        v3,x6           # Store the sum
vdisable                   # Disable vector regs
```

```
int a, X[], Y[];
for (i=0; i<32; i++)
  Y[i] = a* X[i] + Y[i];
```

```
vsetdcfg 1*X32,3*X64  # 1 32b, 3 64b int reg
lw       x7,a         # Load scalar a
vld      v0,x5        # Load vector X
vmul     v1,v0,x7     # Vector-scalar mult
vld      v2,x6        # Load vector Y
vadd     v3,v1,v2     # Vector-vector add
vst      v3,x6        # Store the sum
vdisable              # Disable vector regs
```

Vector Instruction Set Advantages

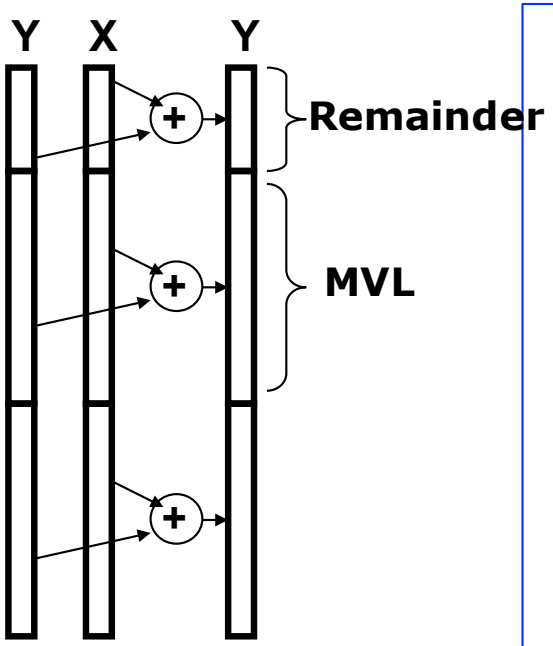
- **Compact**
  - one short instruction encodes N operations
- **Expressive and predictable, tells hardware that these N operations:**
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous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

Vector Length Register

- **Loop count not known at compile time?**

```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```

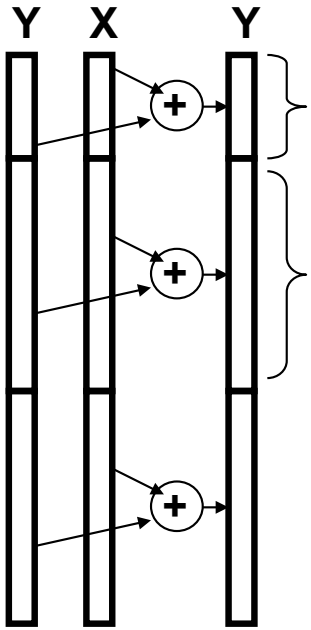
- **Use Vector Length (VL) and Max VL (MVL) Registers**
- **Use *strip mining* for vectors over the maximum length (serialized version before vectorization by compiler)**
  - **Break loops into pieces that fit in registers**




```
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 in RV64V

```
for (i=0; i <n; i=i+1)
    Y[i] = a * X[i] + Y[i];
```




```
        vsetdcfg  2 DP FP     # Enable 2 64b Fl.Pt. registers
        fld       f0,a        # Load scalar a
loop:   setvl     t0,a0       # vl = t0 = min(mvl,n)
        vld       v0,x5       # Load vector X
        slli      t1,t0,3     # t1 = vl * 8 (in bytes)
        add       x5,x5,t1    # Increment pointer to X by vl*8
        vmul      v0,v0,f0    # Vector-scalar mult
        vld       v1,x6       # Load vector Y
        vadd      v1,v0,v1    # Vector-vector add
        sub       a0,a0,t0    # n -= vl (t0)
        vst       v1,x6       # Store the sum into Y
        add       x6,x6,t1    # Increment pointer to Y by vl*8
        bnez      a0,loop     # Repeat if n != 0
        vdisable              # Disable vector regs
```

Using Predictate Register for Vector Mask

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

Use predicate register to "disable" elements (1 bit per element):

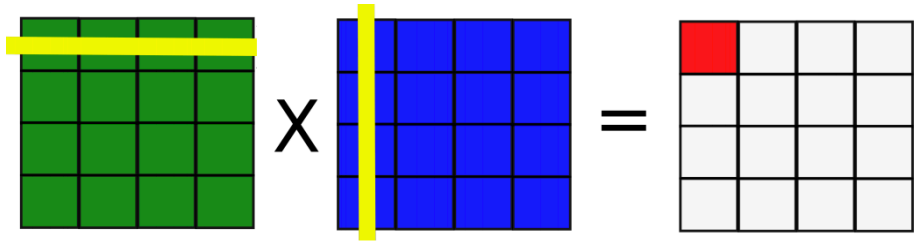
```
vsetdcfg    2*FP64      # Enable 2 64b FP vector regs
vsetpcfgi   1           # Enable 1 predicate register
vld         v0,x5       # Load vector X into v0
vld         v1,x6       # Load vector Y into v1
fmv.d.x     f0,x0       # Put (FP) zero into f0
vpne        p0,v0,f0    # Set p0(i) to 1 if v0(i)!=f0
vsub        v0,v0,v1    # Subtract under vector mask
vst         v0,x5       # Store the result in X
vdisable                # Disable vector registers
vpdisable               # Disable predicate registers
```

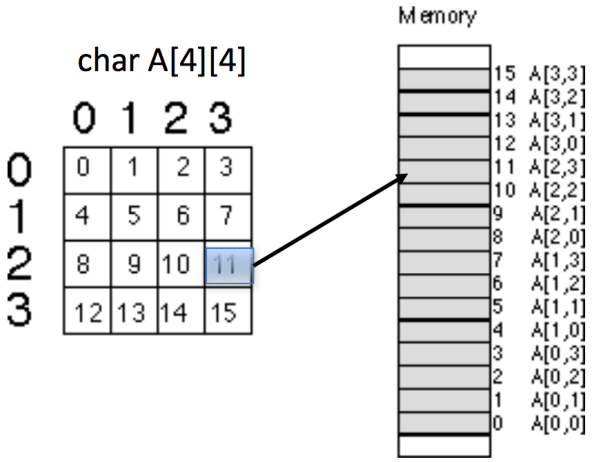
- **GFLOPS rate decreases!**
  - **Vector operation becomes bubble ("NOP") at elements where mask bit is clear**

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***
  - **vlds and vsts: strided load and store**
- **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[ ])**

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

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

- **Use index vector:**

```
vsetdcfg   4*FP64          # 4  64b  FP vector registers
vld        v0, x7          # Load K[]
vldx       v1,(x5, v0)     # Load A[K[]]
vld        v2, x28         # Load M[]
vldi       v3,(x6, v2)     # Load C[M[]]
vadd       v1, v1, v3      # Add them
vstx       v1,(x5, v0)     # Store A[K[]]
vdisable                   # Disable vector registers
```

Memory Operations (vld and vst)

- 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

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**

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

- **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**
- **Today's supercomputer**
  - **https://www.top500.org/**

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

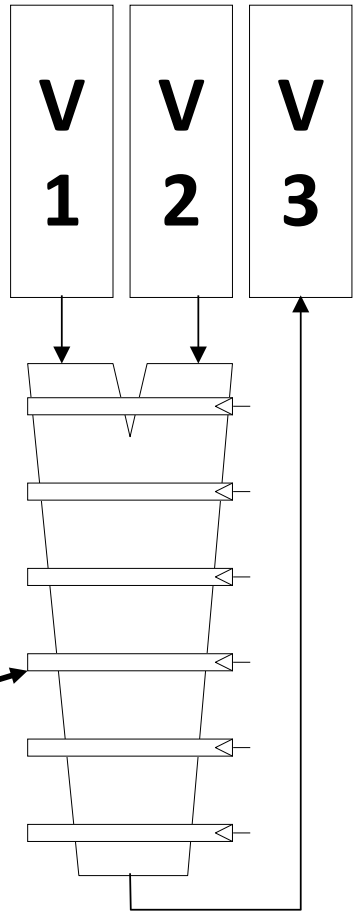
- All involve huge computations on large data sets

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

Vector Arithmetic Execution

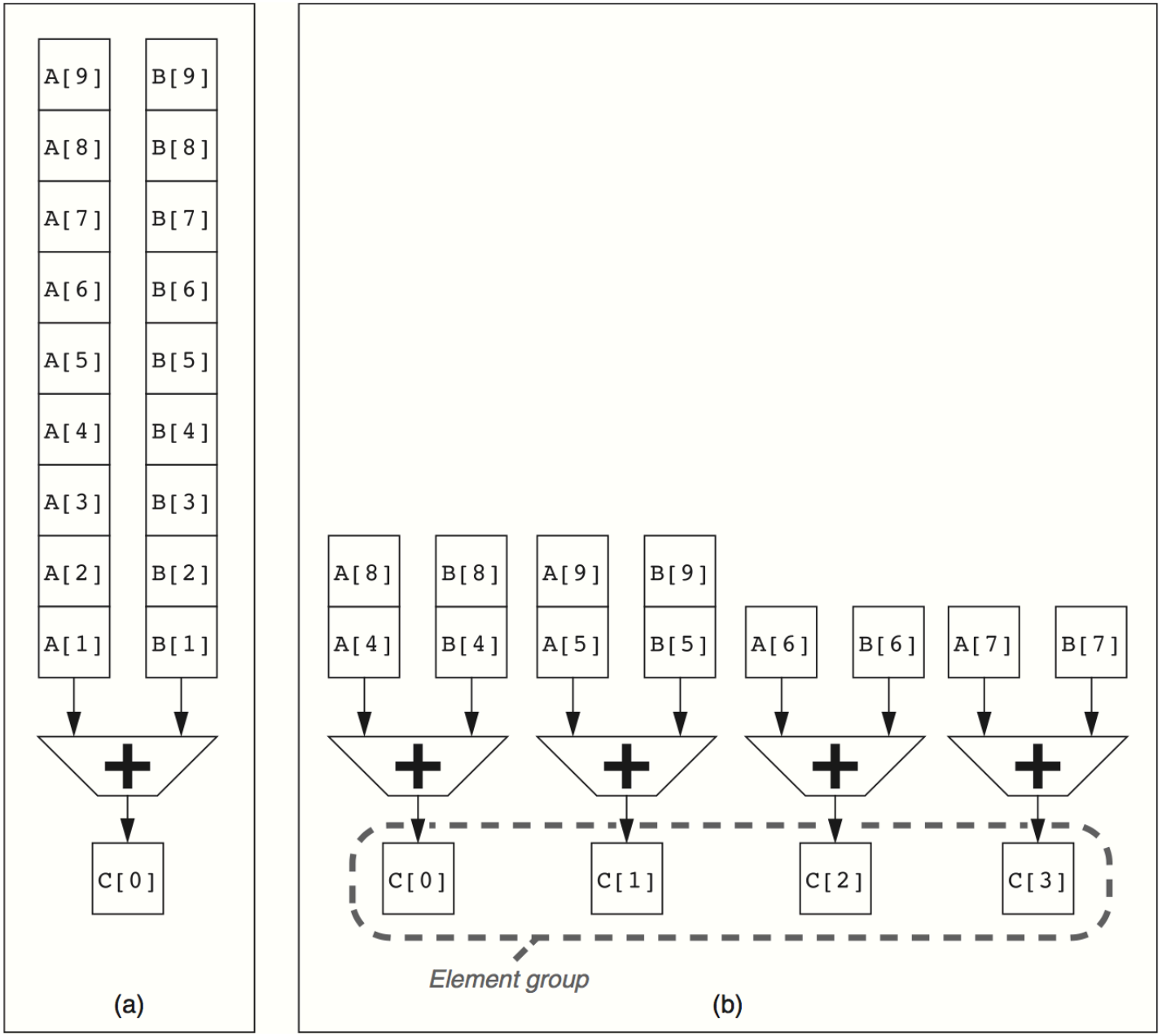
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



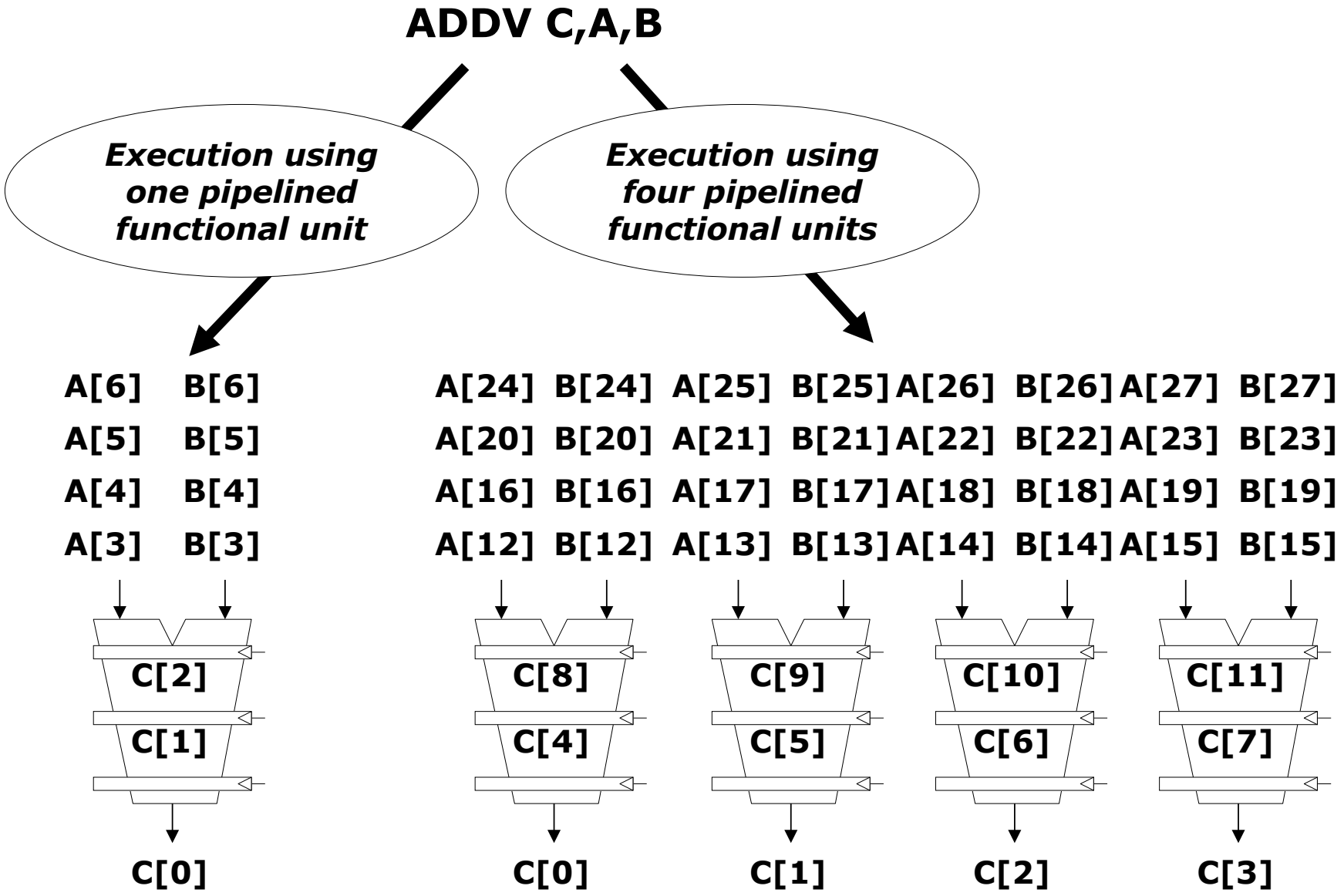
Six stage multiply pipeline

V3 <- v1 * v2

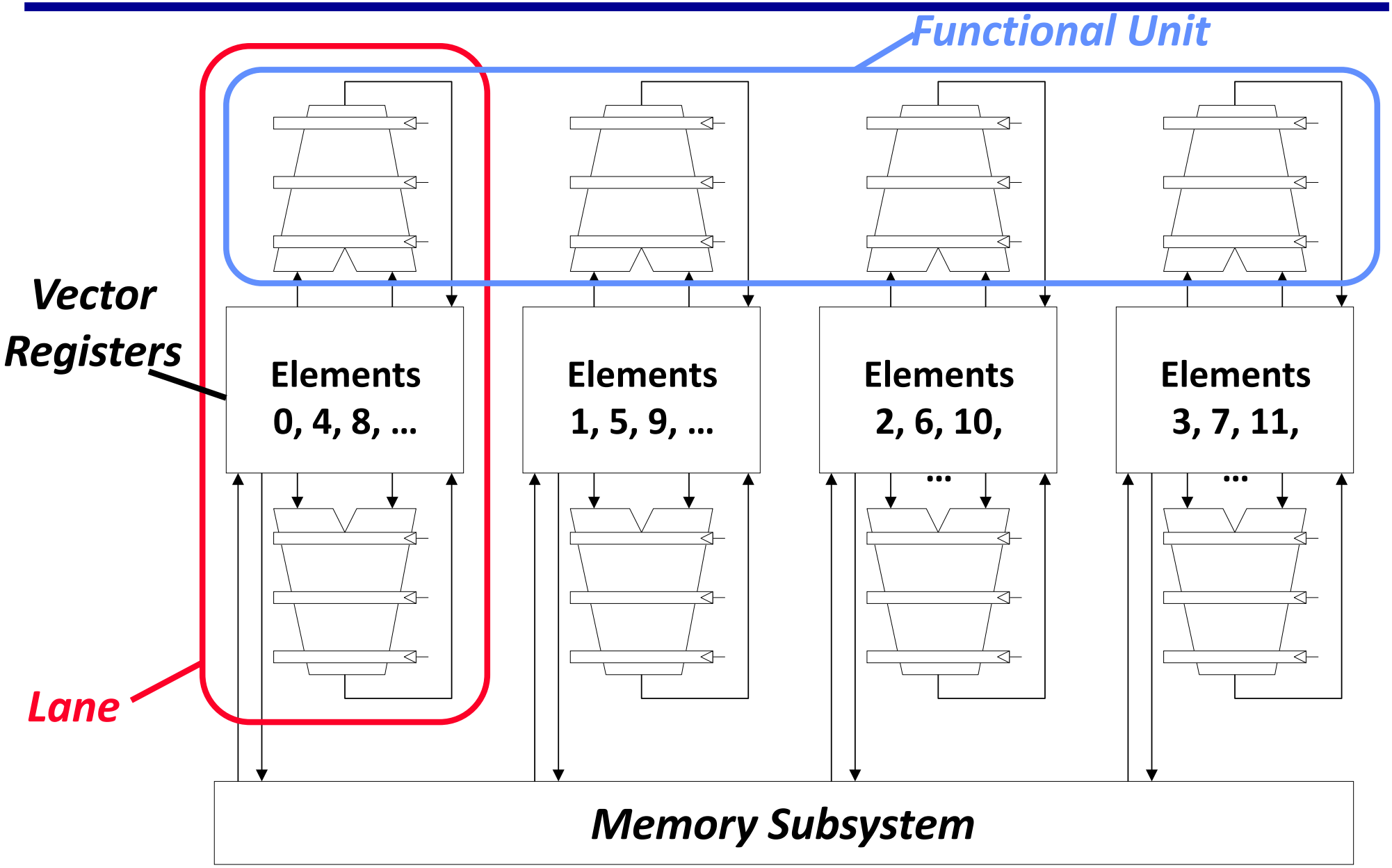
Vector Execution: Element Group



Vector Instruction Execution with Pipelined Functional Units

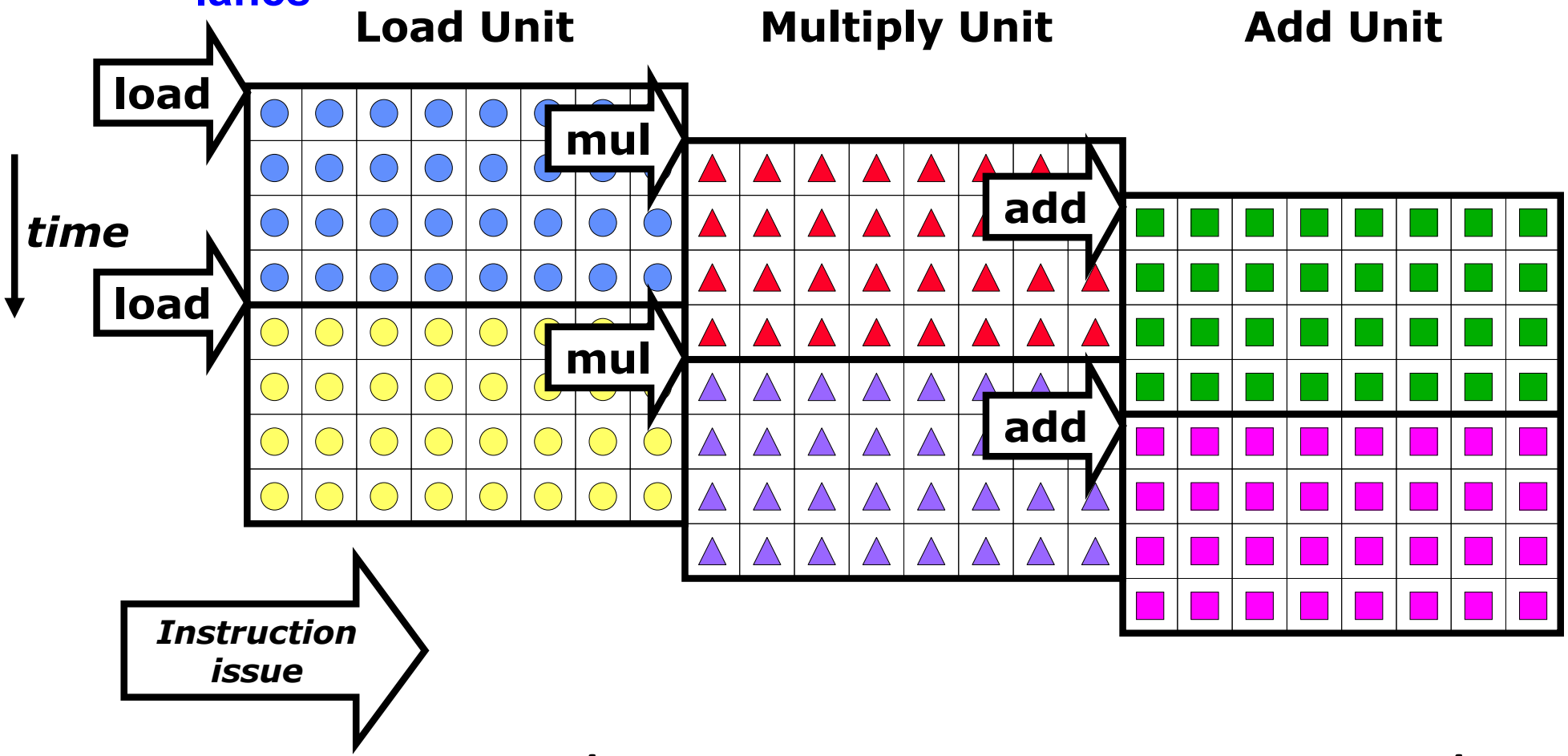


Vector Unit Structure (4 Lanes)



Vector Instruction Parallelism

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



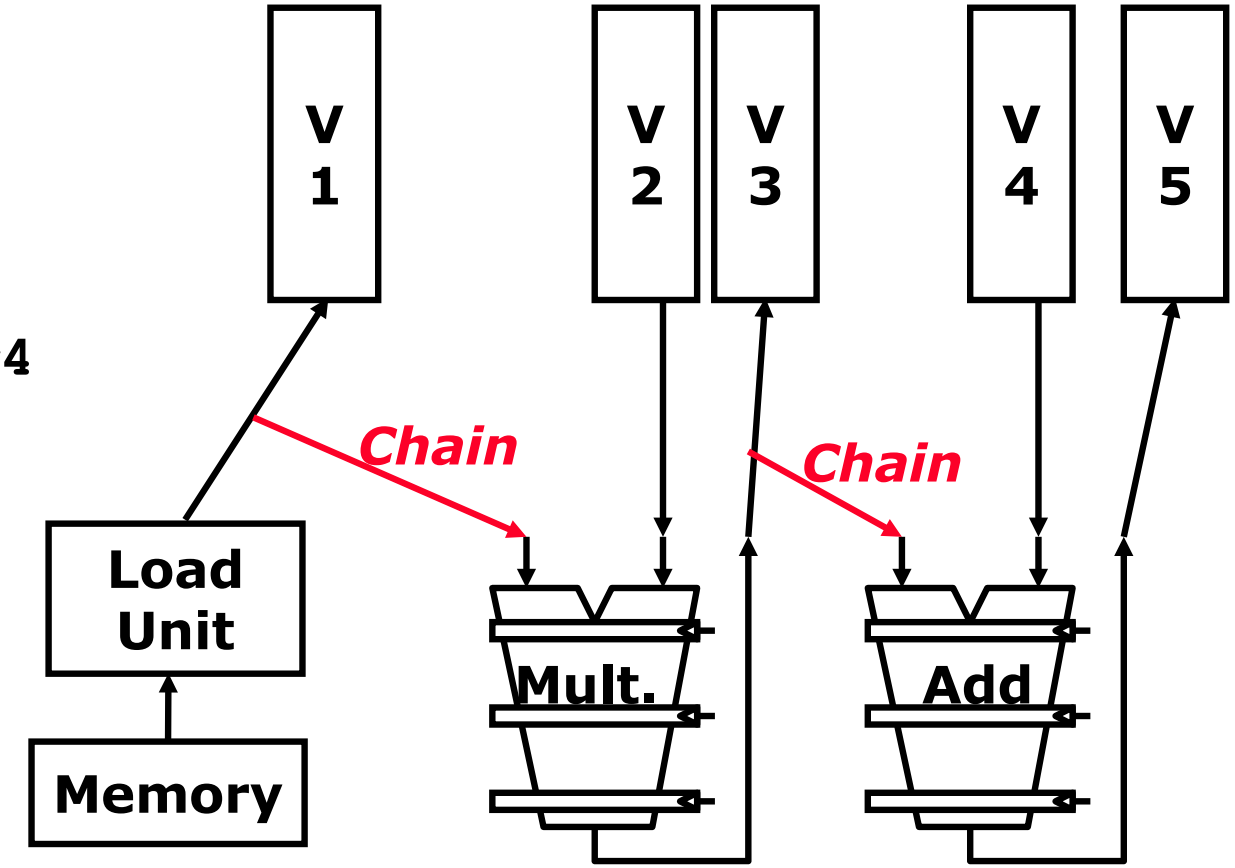
Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

Vector version of register bypassing

– introduced with Cray-1

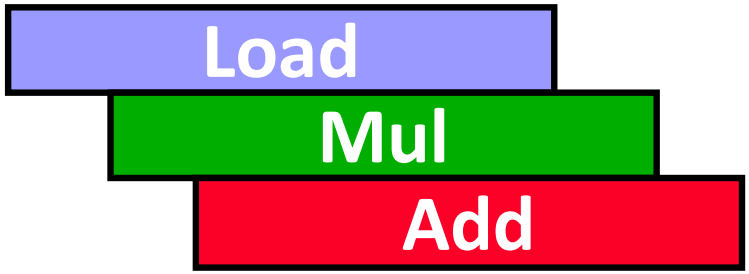
```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



Vector Chaining Advantage

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

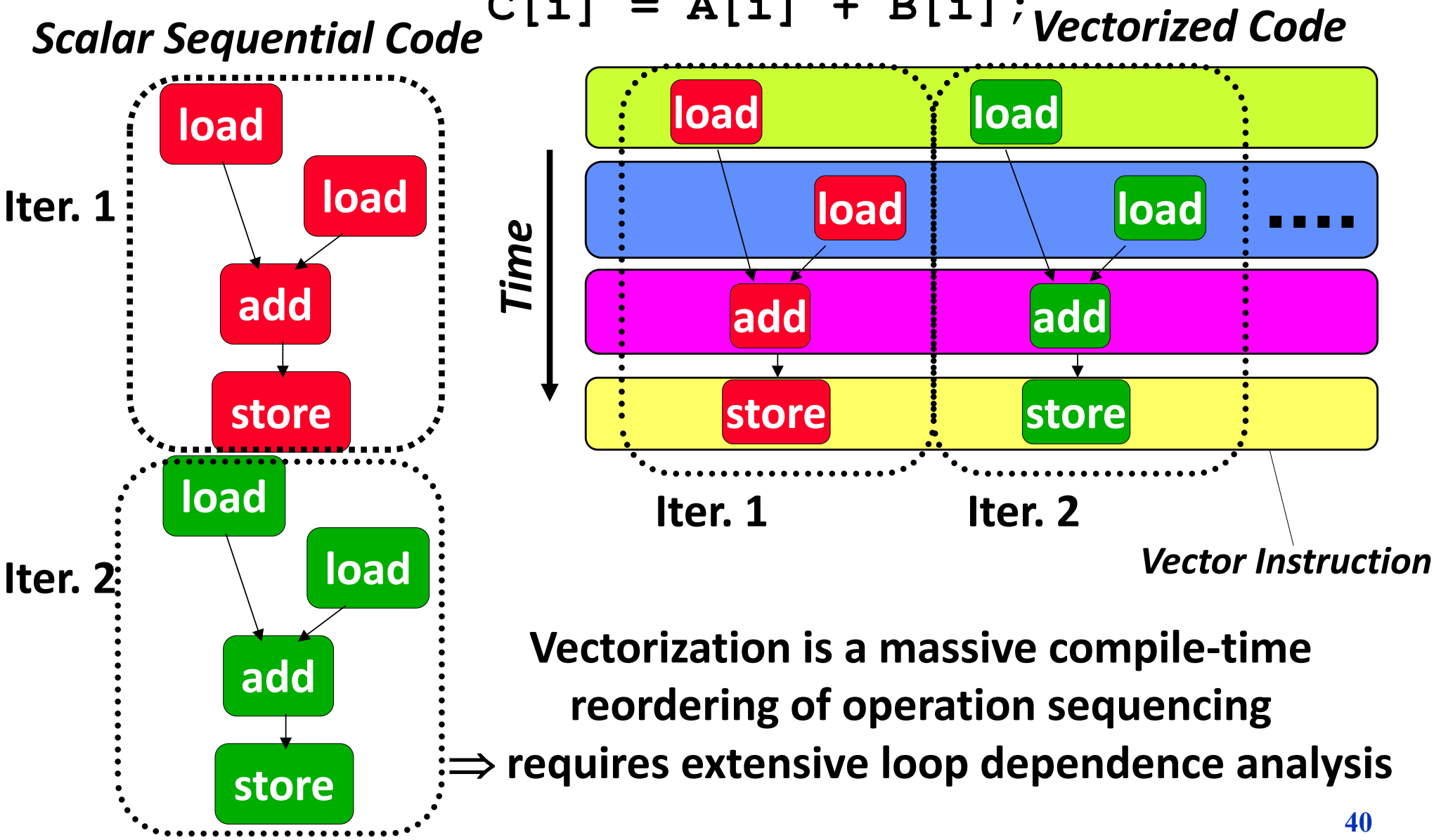
- With chaining, can start dependent instruction as soon as first result appears



Class Lecture Ends Here!

Automatic Code Vectorization

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



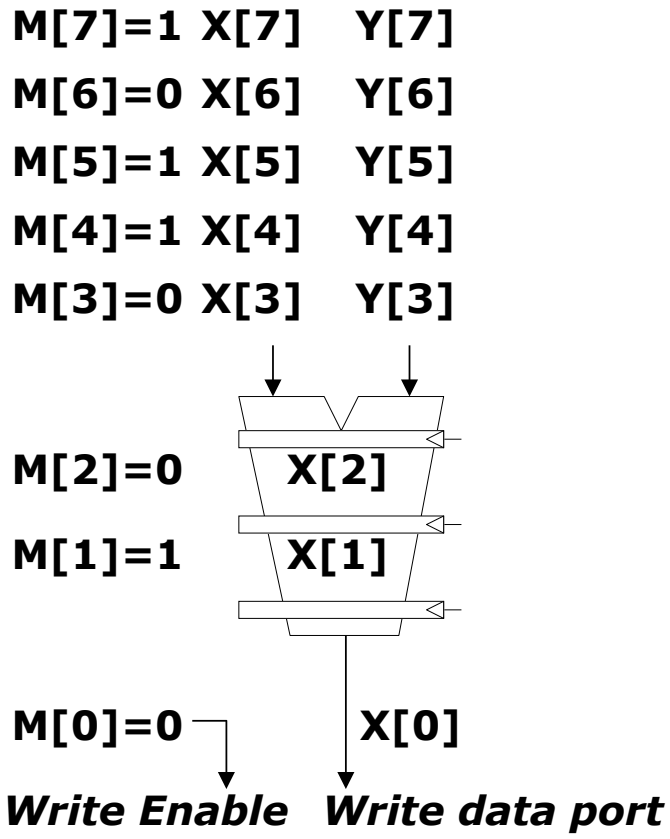
Vectorization is a massive compile-time reordering of operation sequencing

⇒ requires extensive loop dependence analysis

Masked Vector Instructions

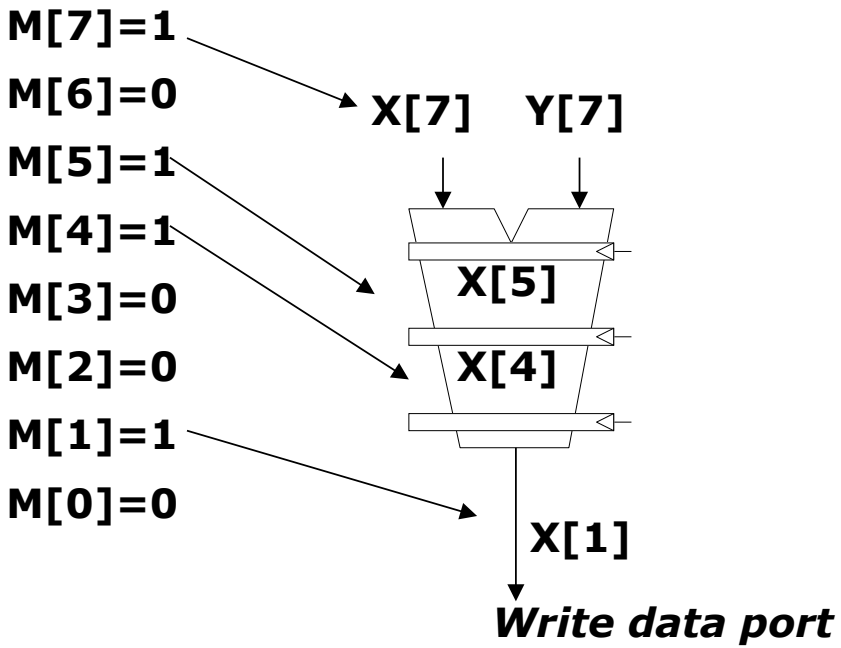
Simple Implementation

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



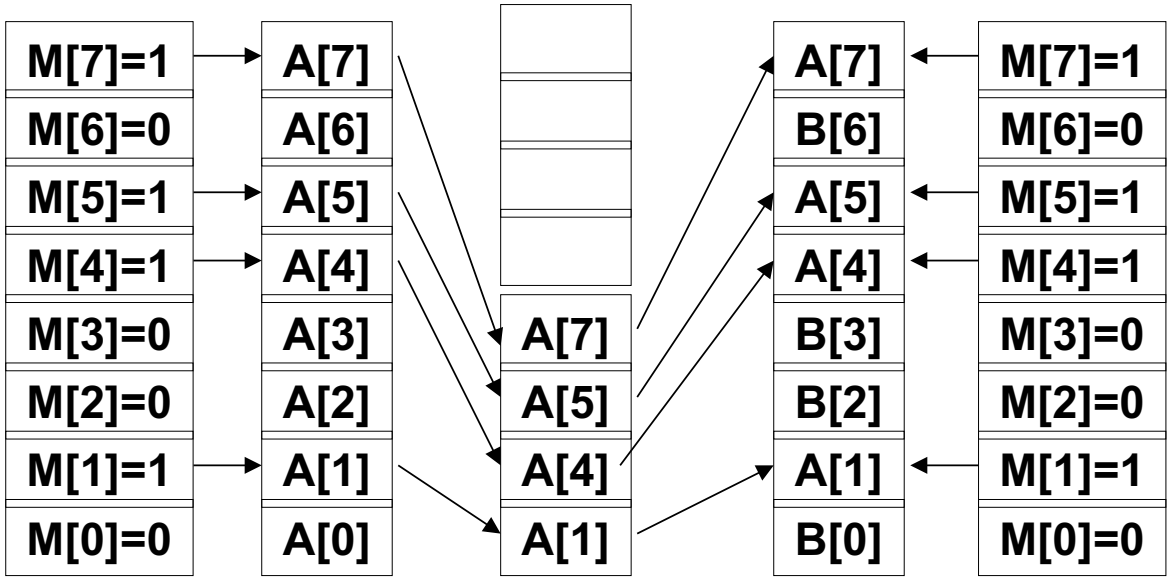
Density-Time Implementation

- 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

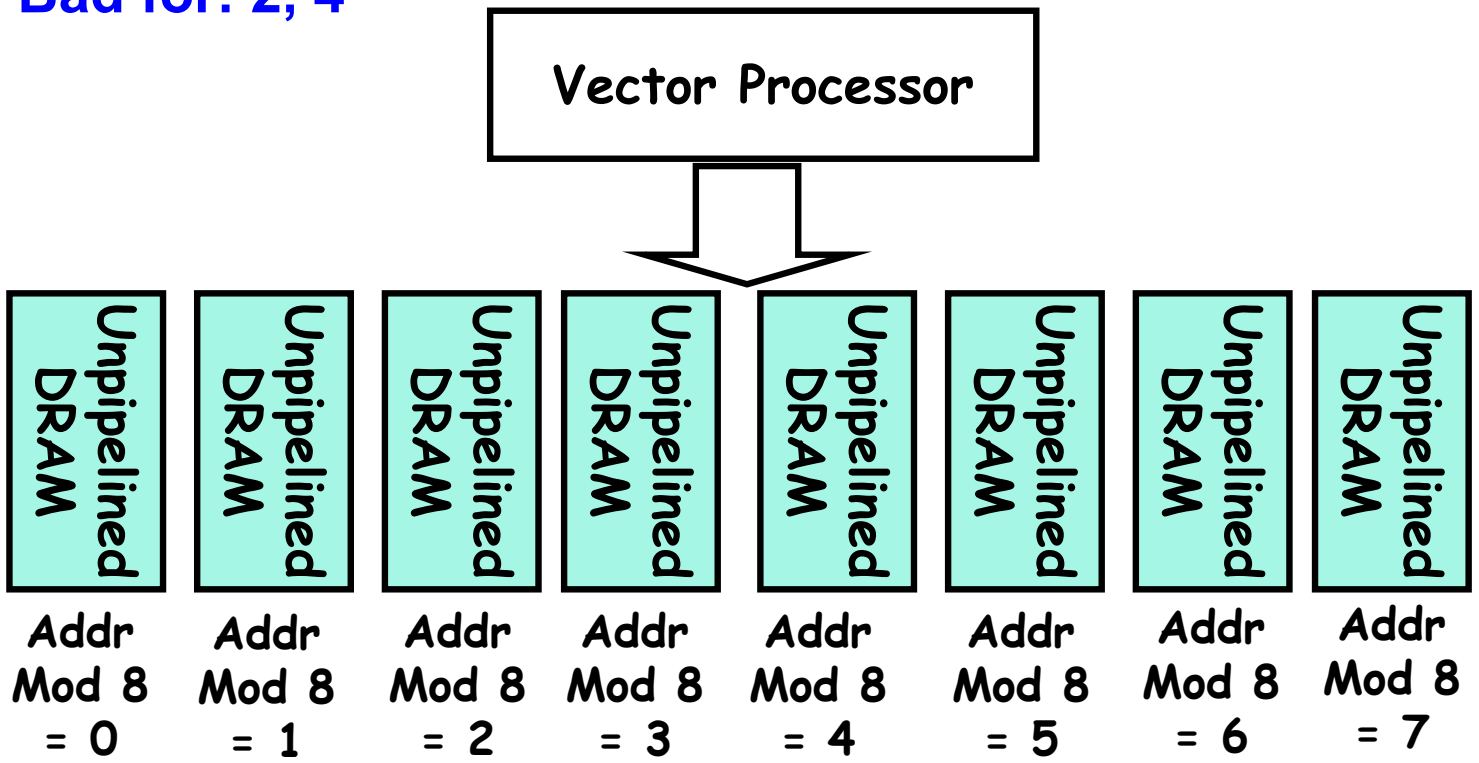


Compress *Expand*

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

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)