# CS4223
# Data Parallelism

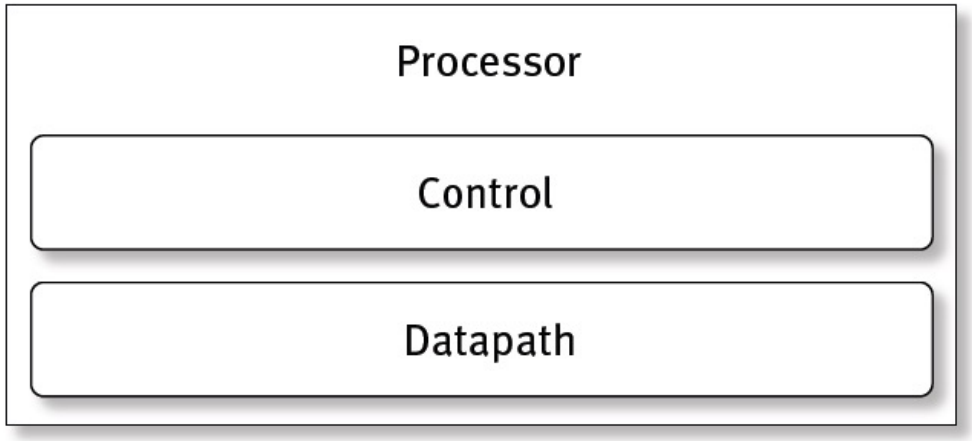
Trevor E. Carlson
National University of Singapore
tcarlson@comp.nus.edu.sg

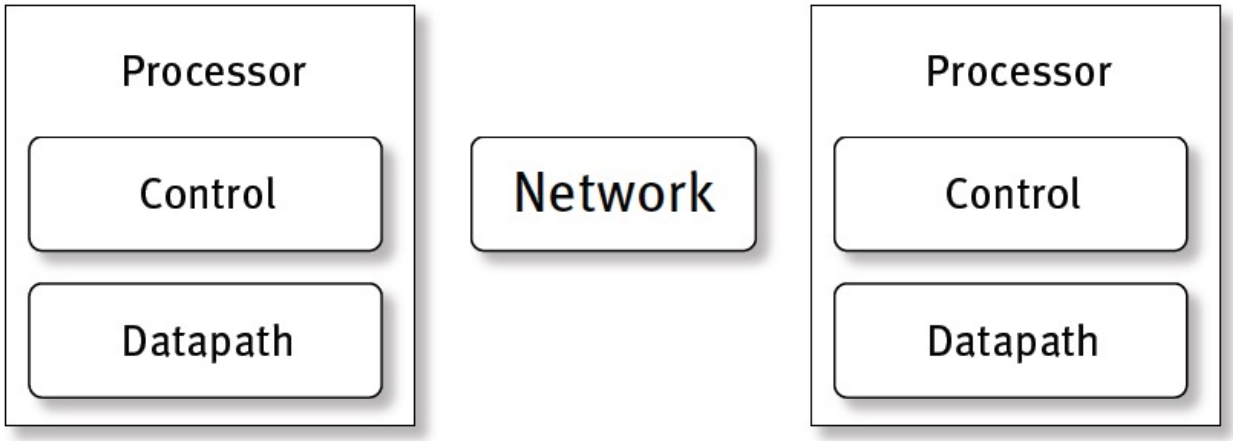
[Slides from Tulika Mitra]
[Partially Adapted from John Owens' Lecture]

# Learning Objectives

- What is DLP?
- Vector Architecture
  - Vector programming model
  - Vector execution
  - Lanes and Chaining
  - Vector memory system
  - Strip-mining
  - Conditional execution
- SIMD extension

# What have we learnt?



# TLP versus DLP

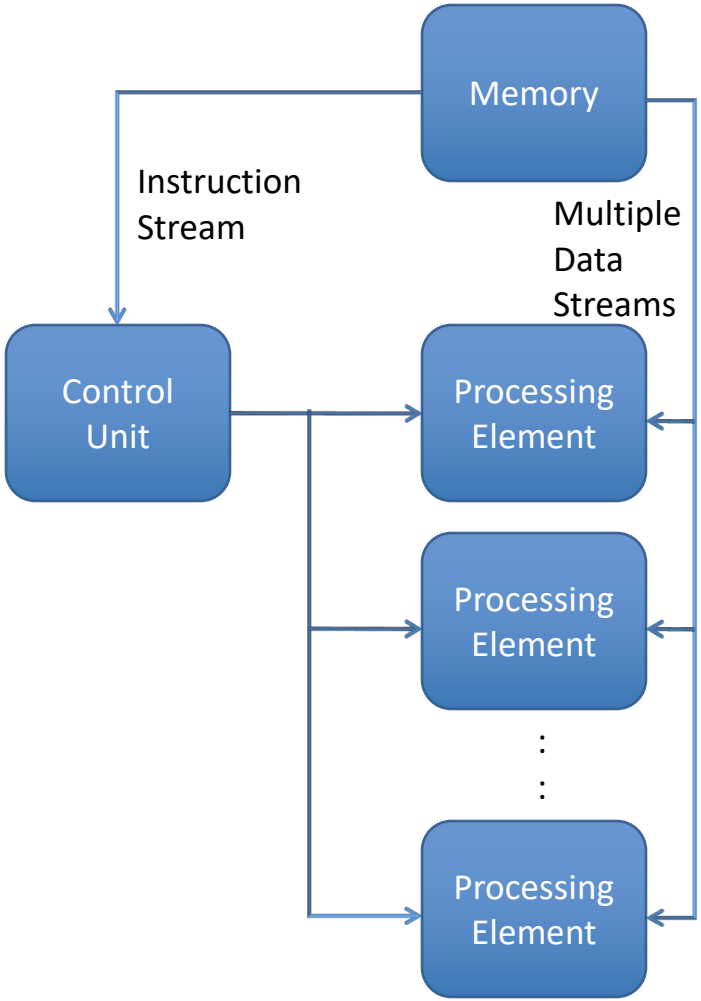
- Thread: process with own instructions and data
  - Thread may be a subpart of a parallel program (“thread”) or it may be an independent program (“process”)
  - Each thread has all the state (instructions, data, PC, register state, etc.) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and (possibly) lots of data

# Data Level Parallelism (DLP)

- DLP is parallelism inherent in program loops where similar operation sequences are performed on elements of a large data structure
- Need compiler/programmer's help in extracting DLP

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

# SIMD: Single Instruction Multiple Data

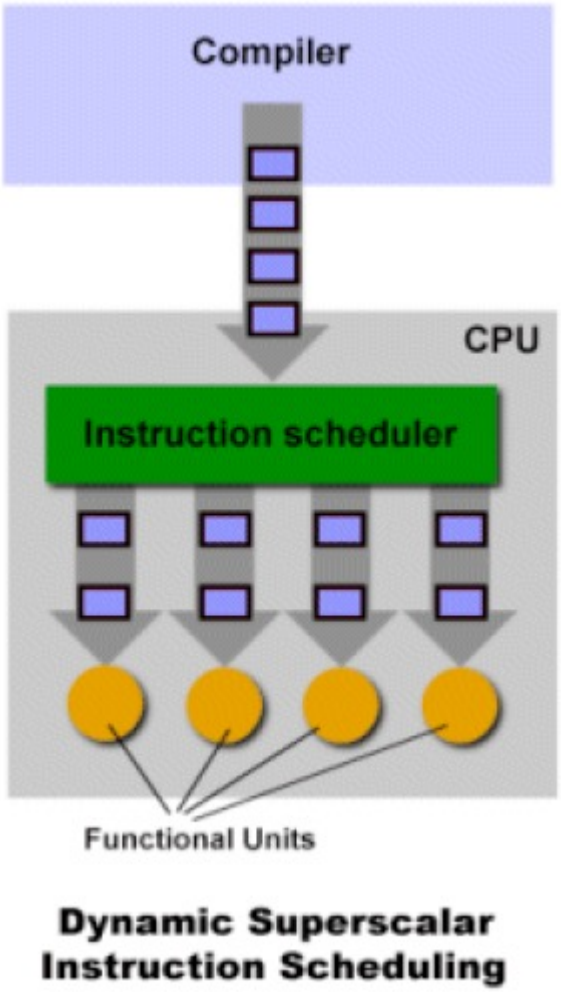


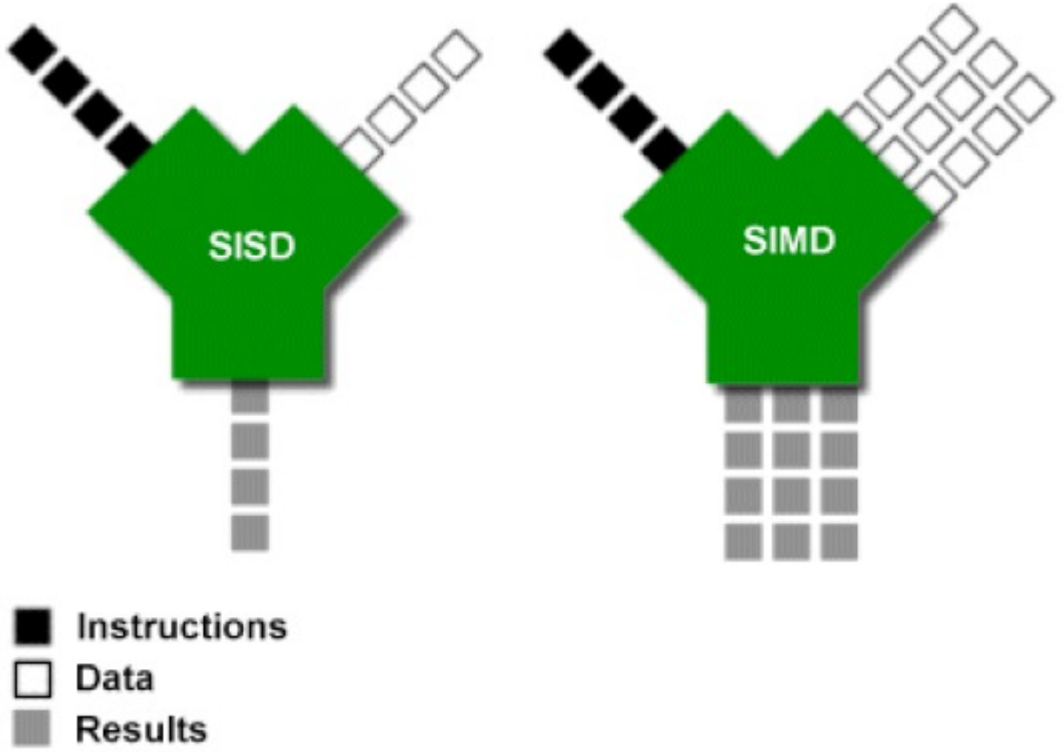

- A single instruction operates on multiple data to exploit data parallelism
- Vector processors and GPUs are excellent examples of SIMD architecture
- More efficient in terms of instruction count and loop control overhead

# ILP versus DLP Processors

- SIMD is about exploiting parallelism in the data stream, while superscalar out-of-order SISD is about exploiting parallelism in the instruction stream

# ILP versus DLP Processors (contd.)



# Benefit

- 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
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

# SIMD Architectures

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

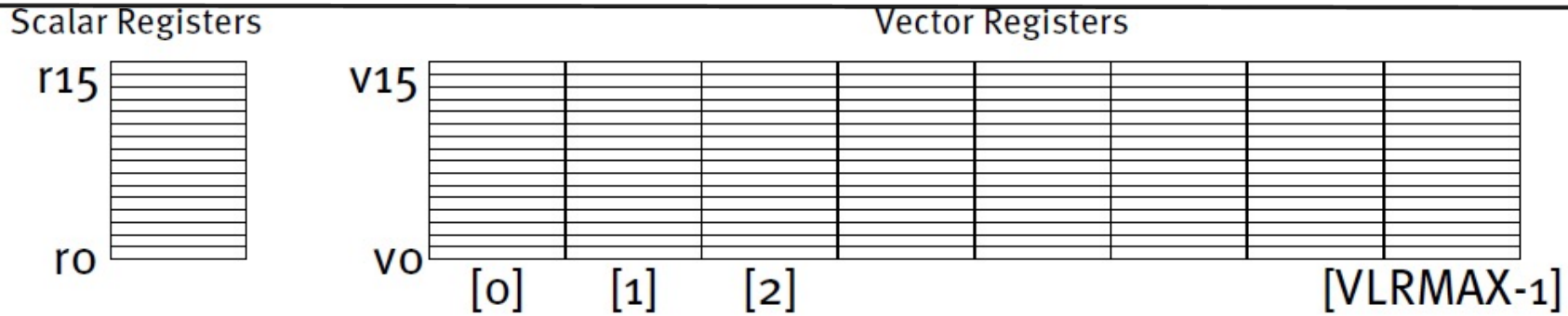
# Scalar versus Vector

- Basic unit of SIMD processing is the vector
- A vector is nothing more than a row of individual numbers or scalars
- Vector representation:
  - Multiple item within same data word
  - Multiple data words

# Vector Supercomputers

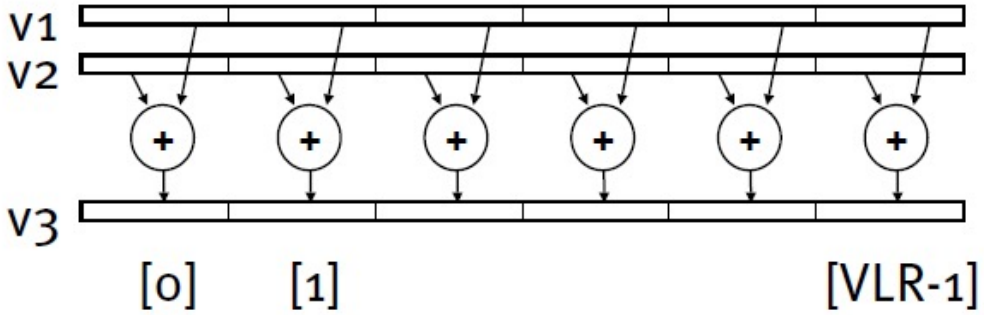
- Epitomized by Cray-1, 1976
- 4 chip types
  - 16x4 bit bipolar registers
  - 1024x1 bit SRAM
  - 4/5 input NAND gates
- 138 MFLOPS sustained,
  250 MFLOPS peak

# Vector Programming Model



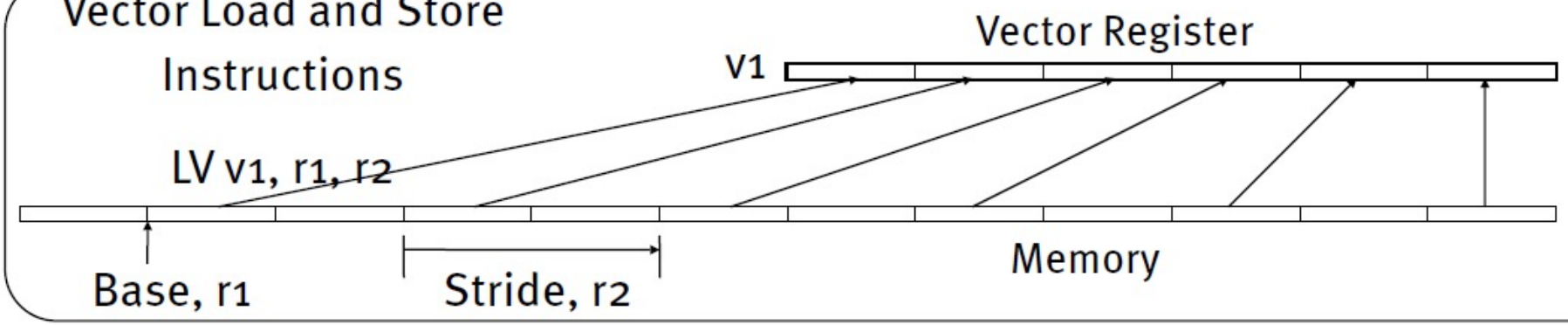

Vector Arithmetic Instructions

ADDV v3, v1, v2




Vector Load and Store Instructions

LV v1, r1, r2



# Vector Programming Model: Basic Idea

- Read sets of data elements into “vector registers”
- Operate on those registers
- Disperse the results back into memory

# Vector Code Example

- DAXPY: Double-Precision a × X + Y

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

# Vector Code Example (contd.)

## Scalar Code

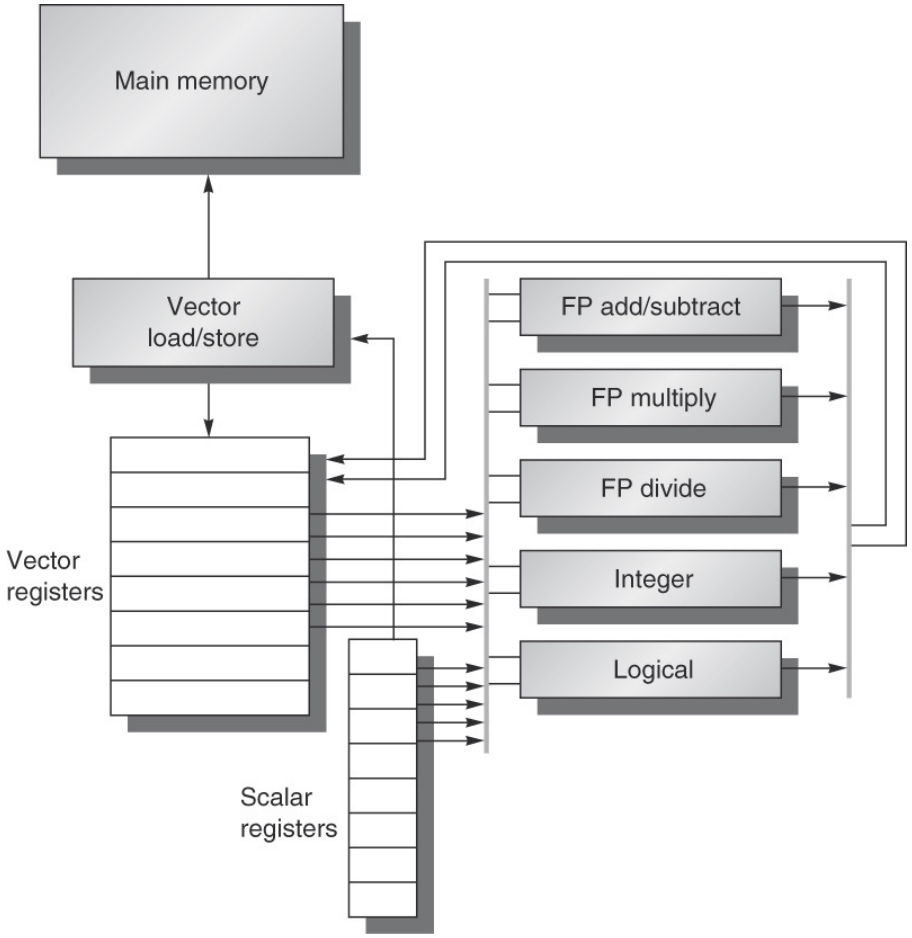
```
        L.D     F0,a
        DADDIU  R4,Rx,#512 ; last address to load
loop:   L.D     F2, 0(Rx)   ;load X(i)
        MULT.D  F2,F0,F2    ;a*X(i)
        L.D     F4, 0(Ry)   ;load Y(i)
        ADD.D   F4,F2,F4    ;a*X(i) + Y(i)
        S.D     F4,0(Ry)    ;store into Y(i)
        DADDIU  Rx,Rx,#8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

## Vector Code

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

Instructions: 578 (2 + 9*64 ) versus 6 (96X improvement)
Operations: 578 (2 + 9*64) versus 321 (1+5*64) (1.8X improvement)

# Vector Architecture

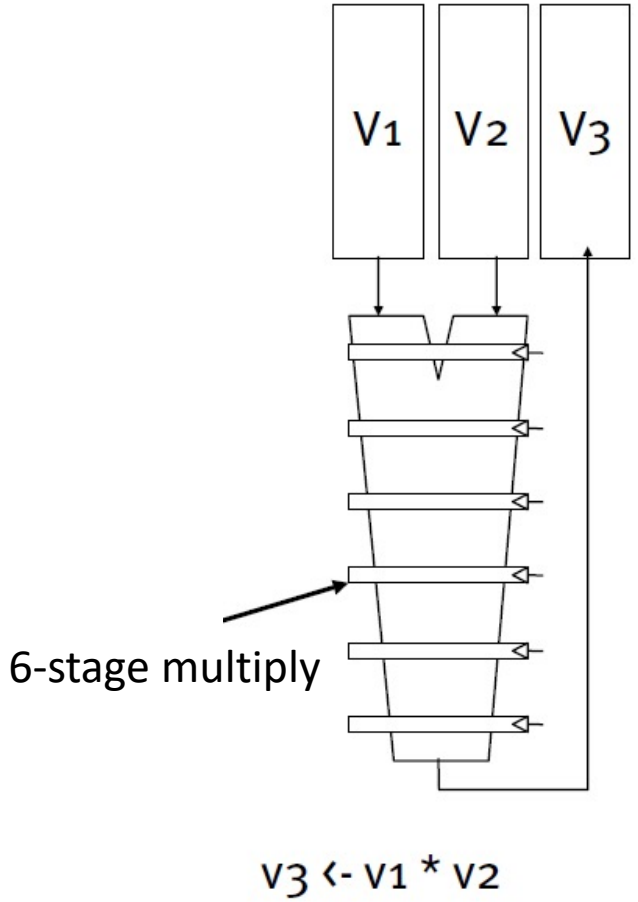


# Vector Arithmetic Execution

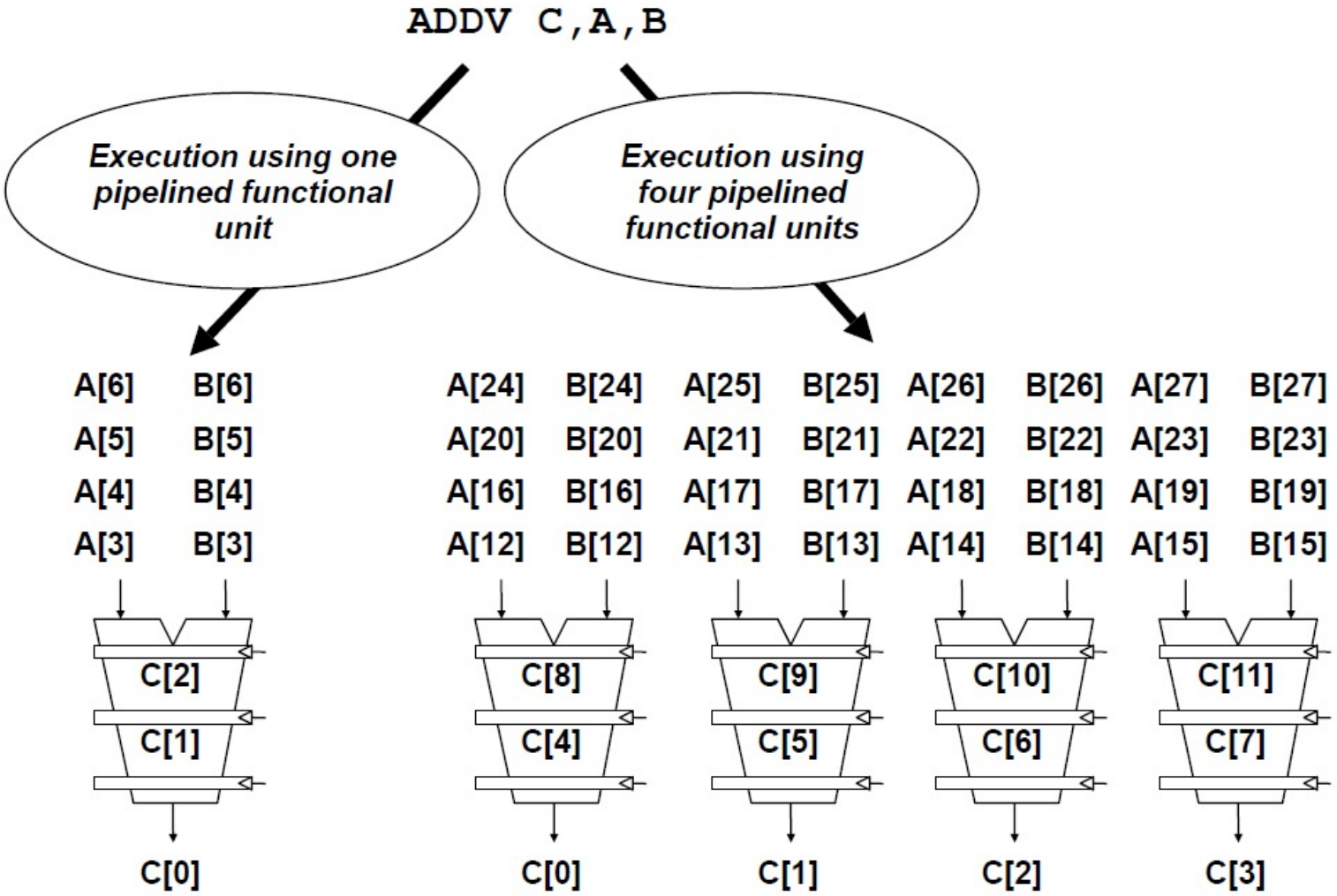
- Use deep pipeline

  (=> fast clock) to execute primitive operations

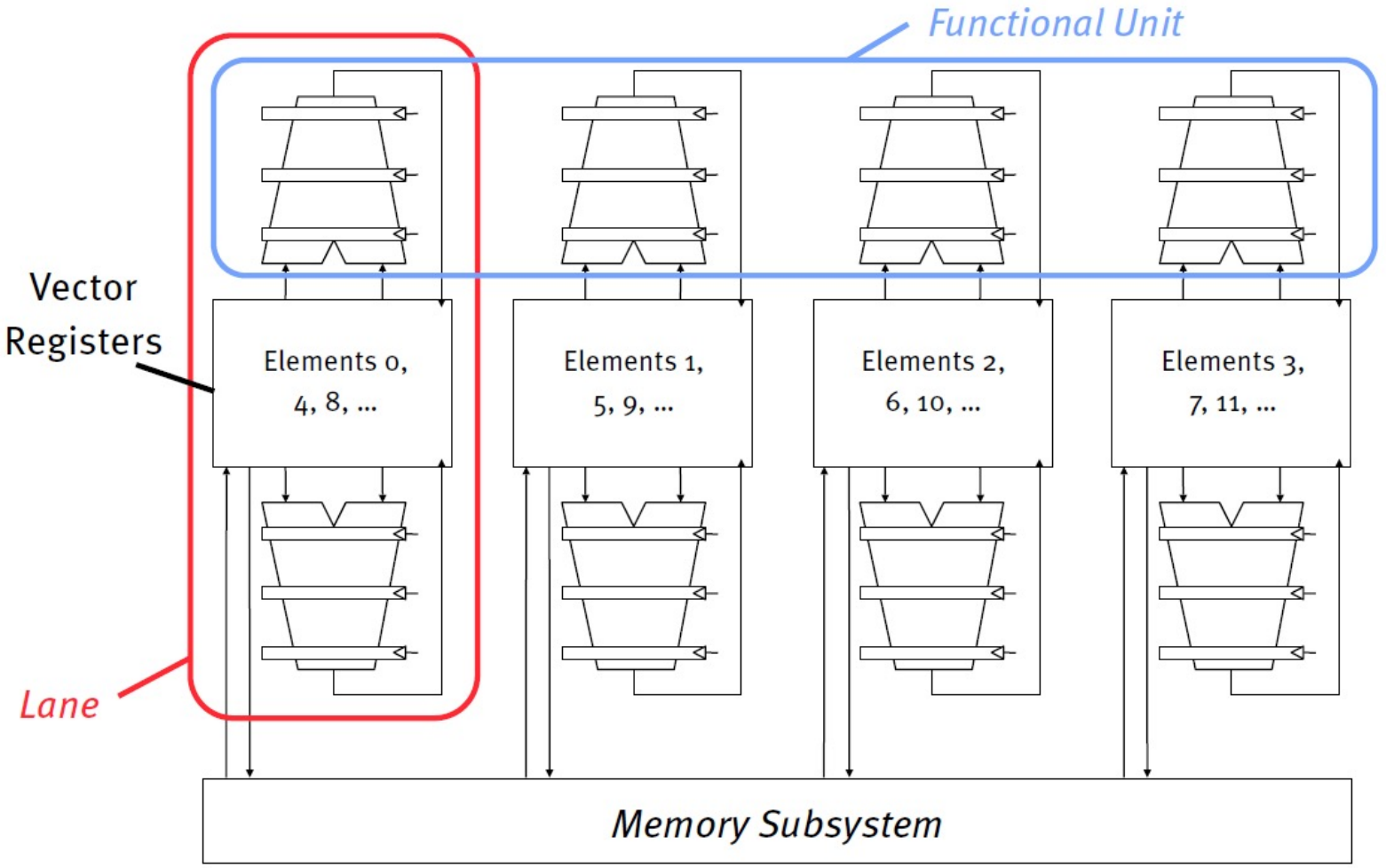
- Simplified deep pipeline control because elements in vector are independent (=> no hazards)



# Vector Instruction Execution



# Vector Unit Structure

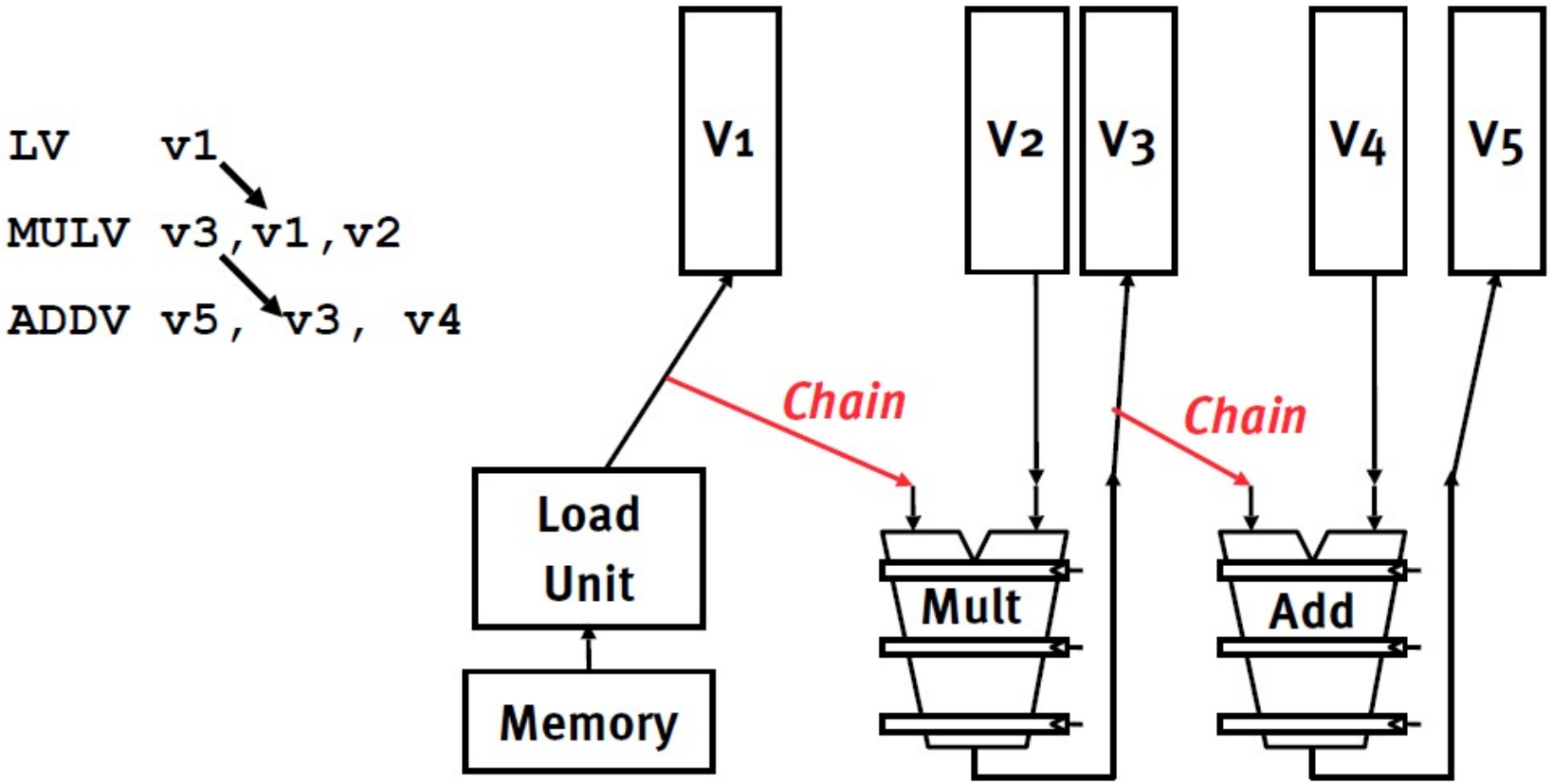


# Vector Inefficiency

- Must wait for the last element of result to be written before starting dependent instructions

# Vector Chaining

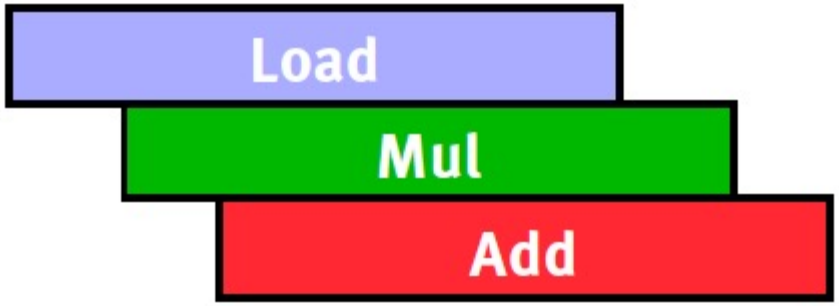
- Vector version of register bypassing



# Vector Chaining Advantage

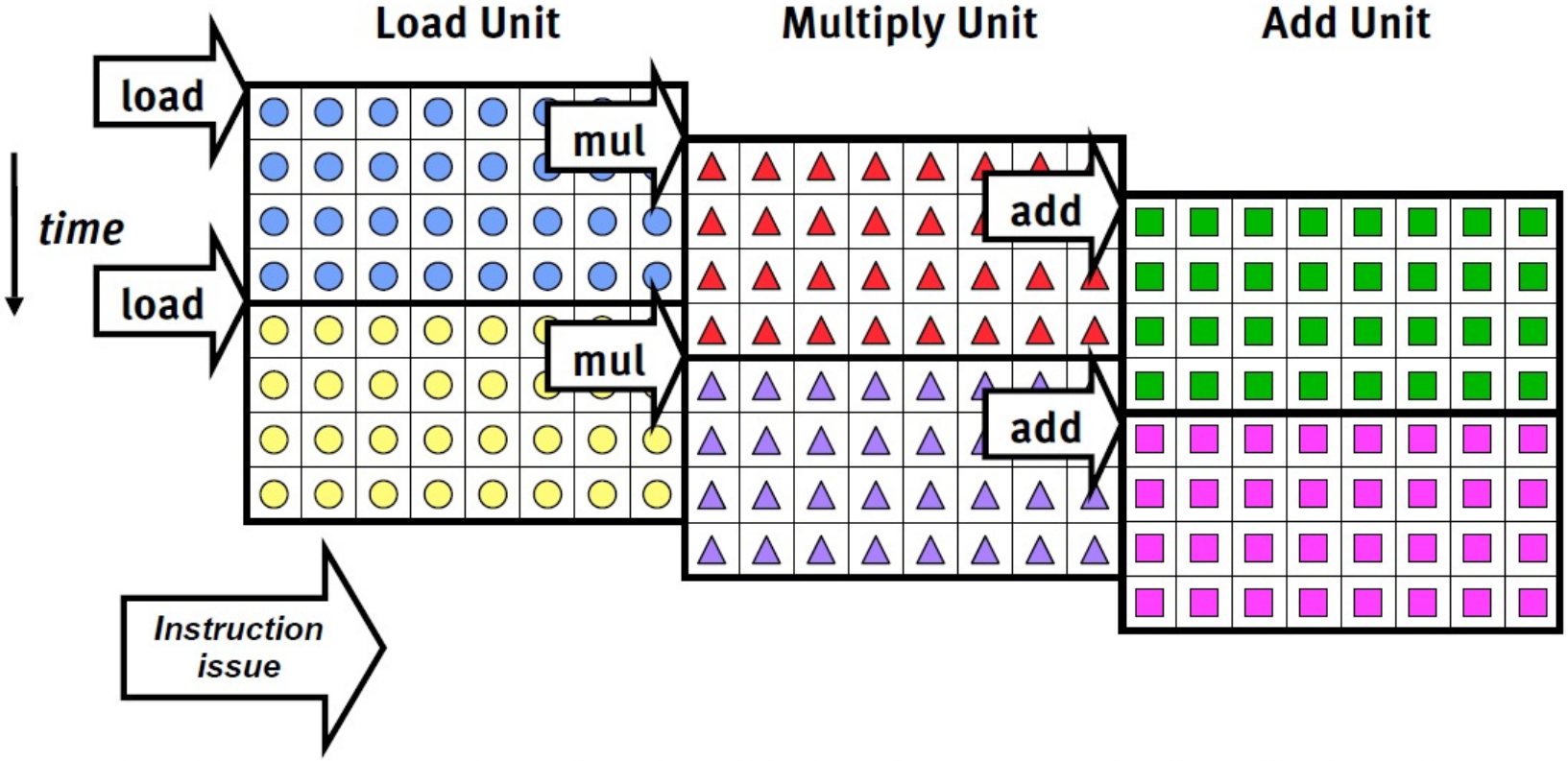
- Without chaining, must wait for last element of result to be written before starting dependent instructions


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



# Vector Instruction Parallelism

- Overlap execution of multiple vector instructions
  - Assume 32 elements per vector register and 8 lanes



# Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies

# Approximate Vector Execution Time

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

1 lane, no chaining, Vector Length =64
Assuming 1 clock cycle per group of instructions, 4 cycles per result
Total cycles = 4x64

# Vector Start-up Time

- Start-up time: pipeline latency time (depth of FU pipeline)

| Operation | Start-up penalty |
|---|---|
| Vector load/store | 12 |
| Vector multiply | 7 |
| Vector add | 6 |

Assume one lane, no chaining, vector length = n

| | Start | 1st result | last result | |
|---|---|---|---|---|
| 1. LV | 0 | 12 | 11+n (12-1+n) | |
| 2. MULVS.D, LV | 12+n | 12+n+12 | 23+2n | load start-up |
| 3. ADDV.D | 24+2n | 24+2n+6 | 29+3n | wait for group 2 |
| 4. SV | 30+3n | 30+3n+12 | 41+4n | wait for group 3 |

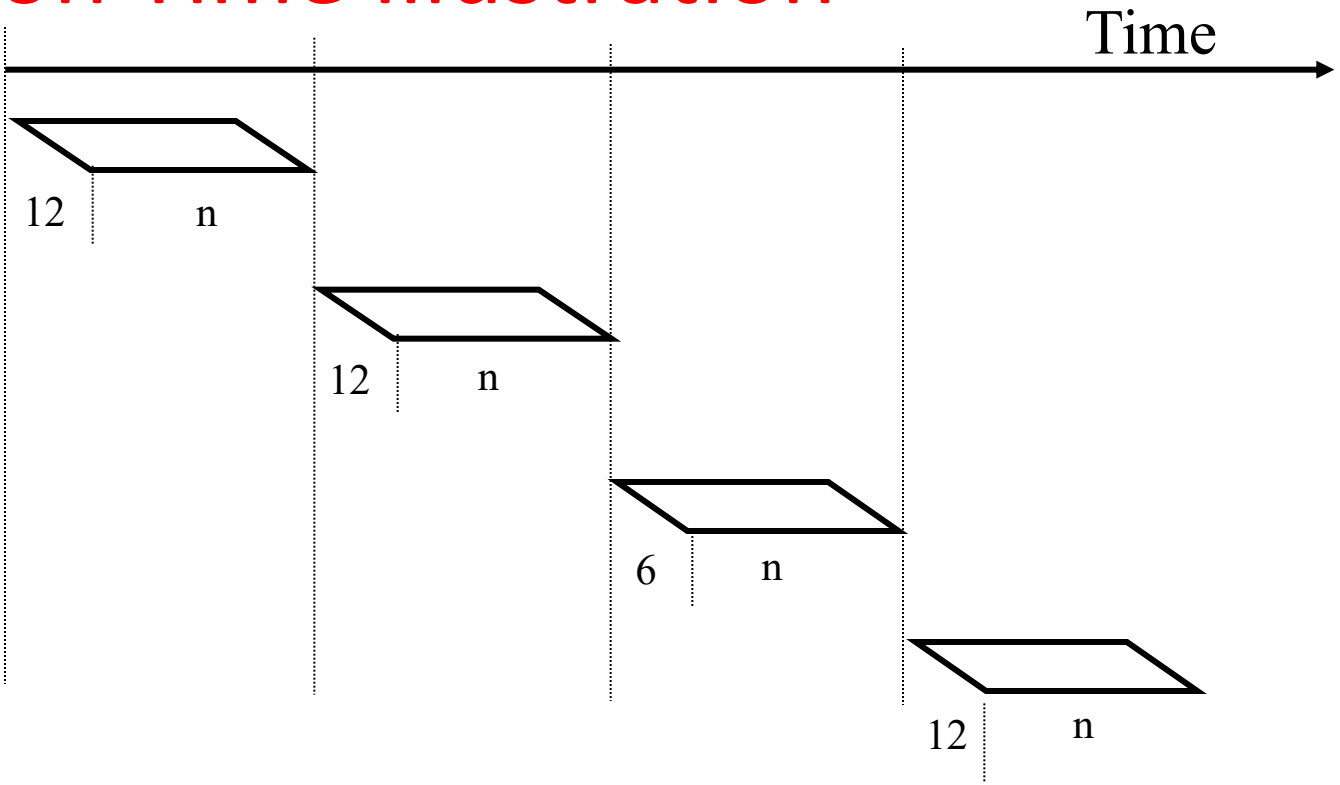
# Execution Time Illustration

1: LV V1,Rx

2: MULV V2,F0,V1
LV V3,Ry

3: ADDV V4,V2,V3

4: SV Ry,V4

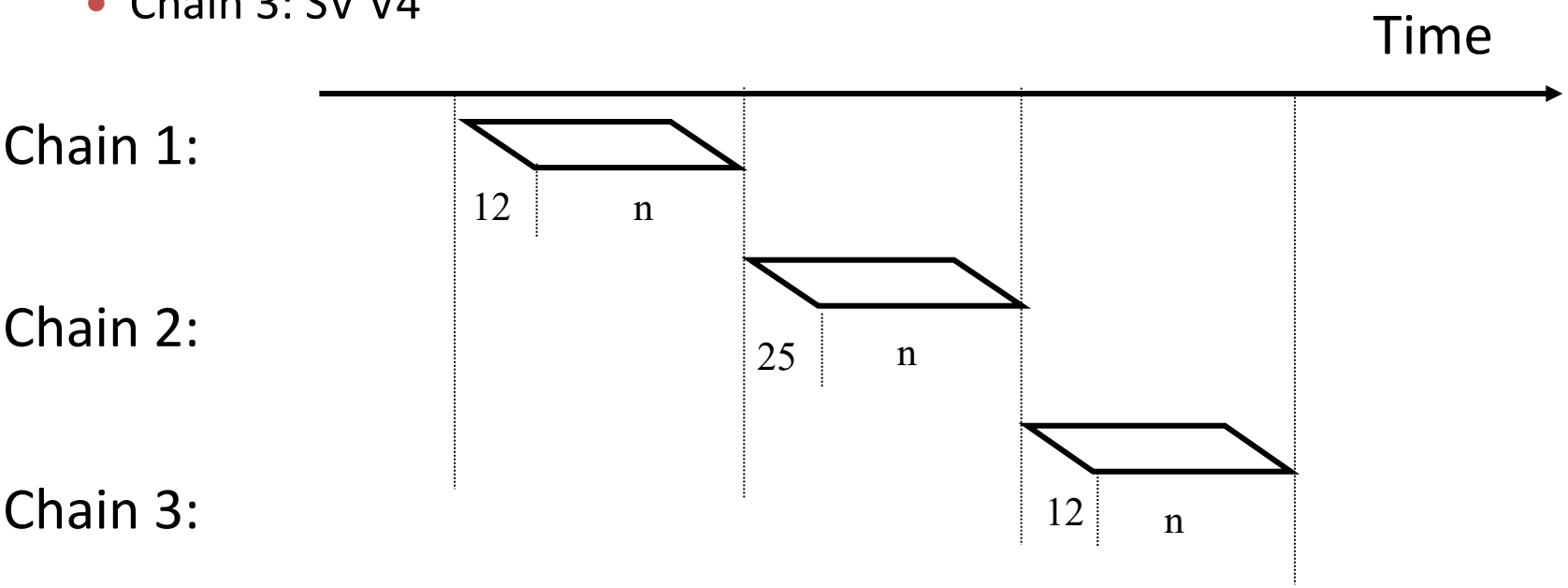


# Vector Chaining

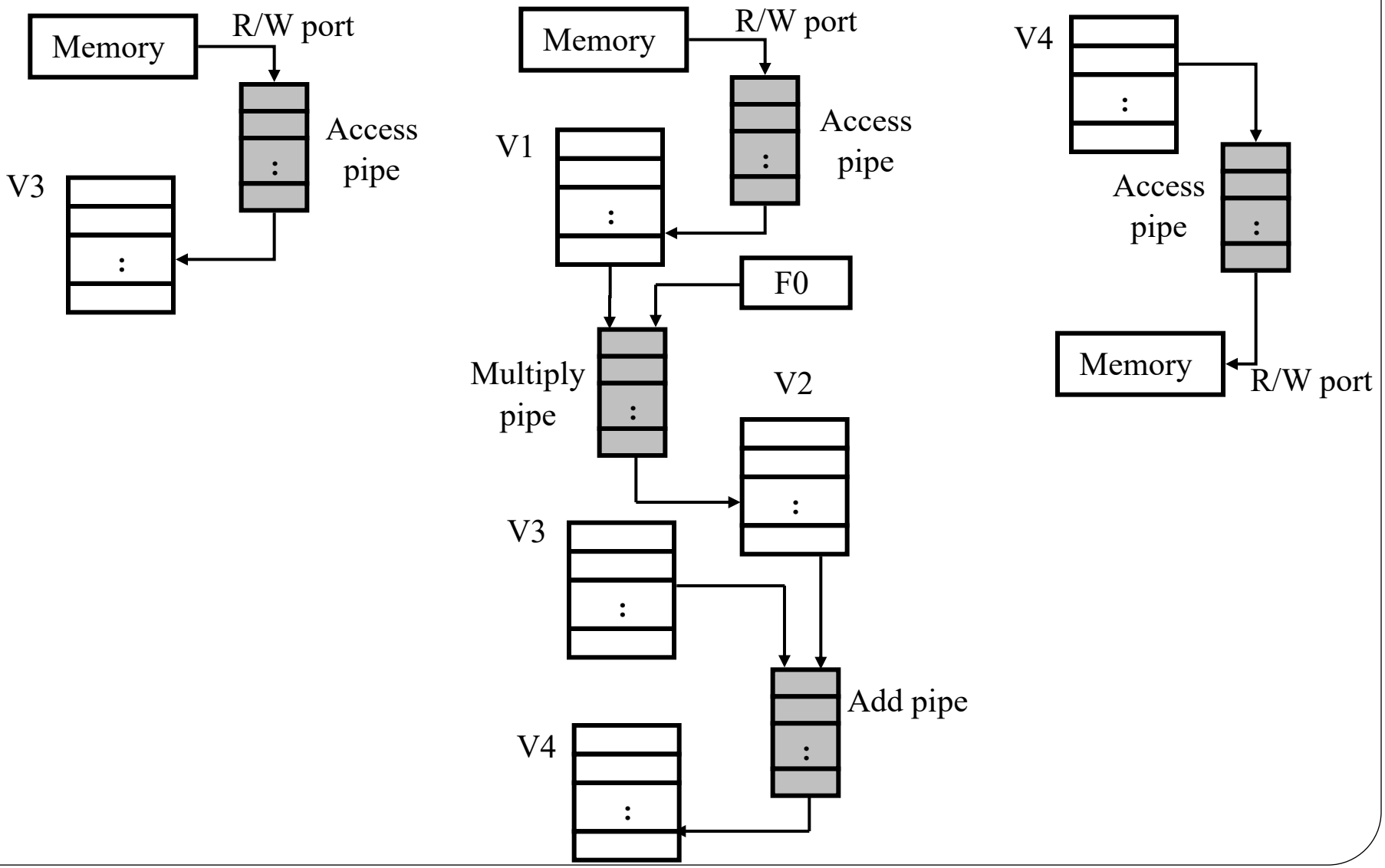
- Suppose:
  MULV.D V1,V2,V3
  ADDV.D V4,V1,V5
- Chaining: vector register is not as a single entity but as a group of individual registers; pipeline forwarding can work on individual elements of a vector
- Flexible chaining: allow vector to chain to any other active vector operation => more read/write port
- As long as enough HW, increases parallelism

# DAXPY Chaining on Cray-1

- One memory access pipe either for load or store (not for both at the same time)
- 3 chains
  - Chain 1: LV V3
  - Chain 2: (LV V1) + (MULV V2,F0,V1) + (ADDV V4,V2,V3)
  - Chain 3: SV V4

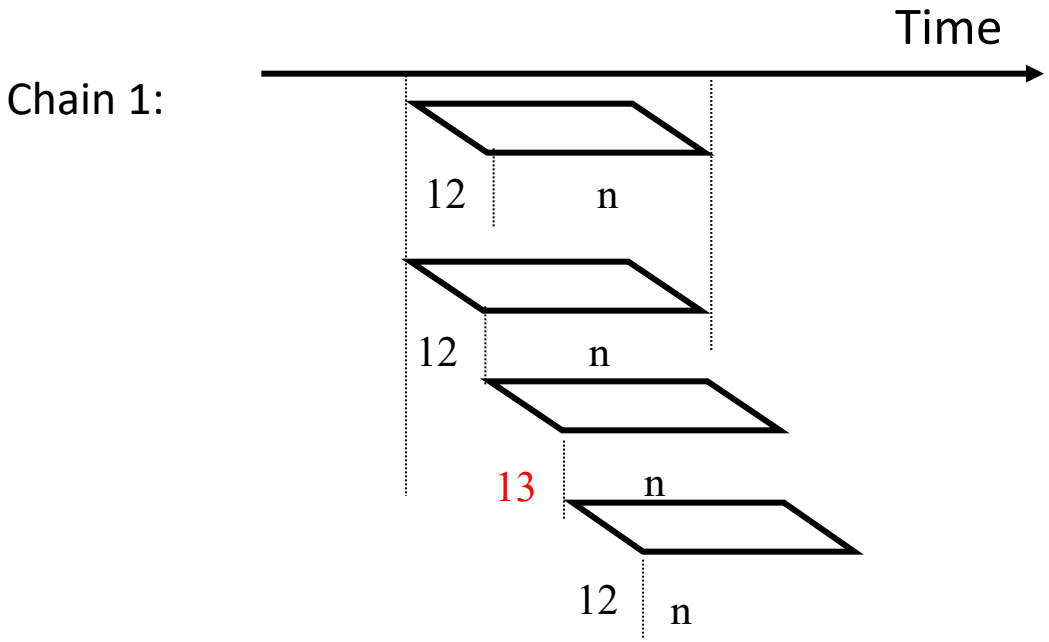


# 3 Chains DAXPY on Cray-1

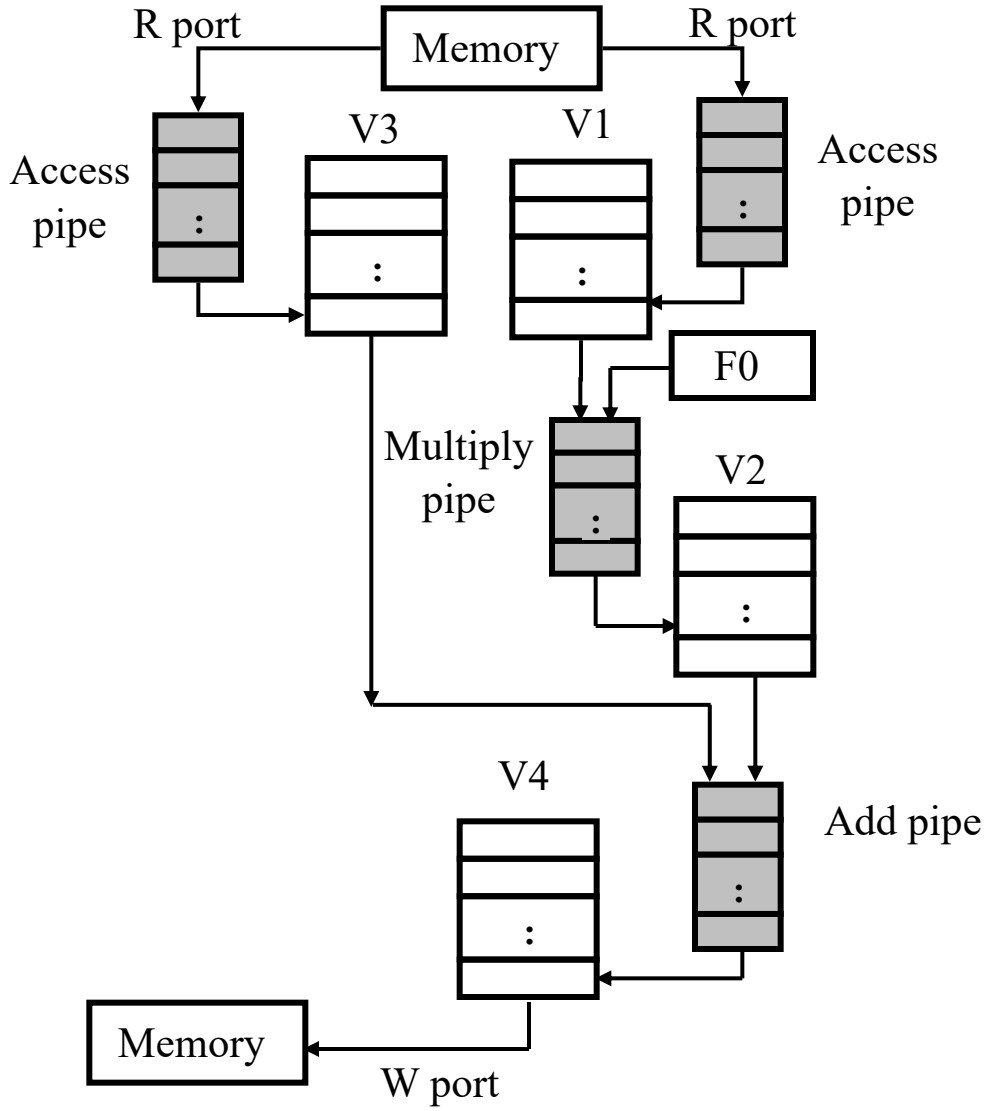


# DAXPY Chaining: CRAY X-MP

- CRAY X-MP has 3 memory access pipes, two for vector load and one for vector store
- 1 chain: (LV V3, LV V1) + (MULV V2,F0,V1) + (ADDV V4,V2,V3) + (SV V4)



# One Chain DAXPY for CRAY X-MP

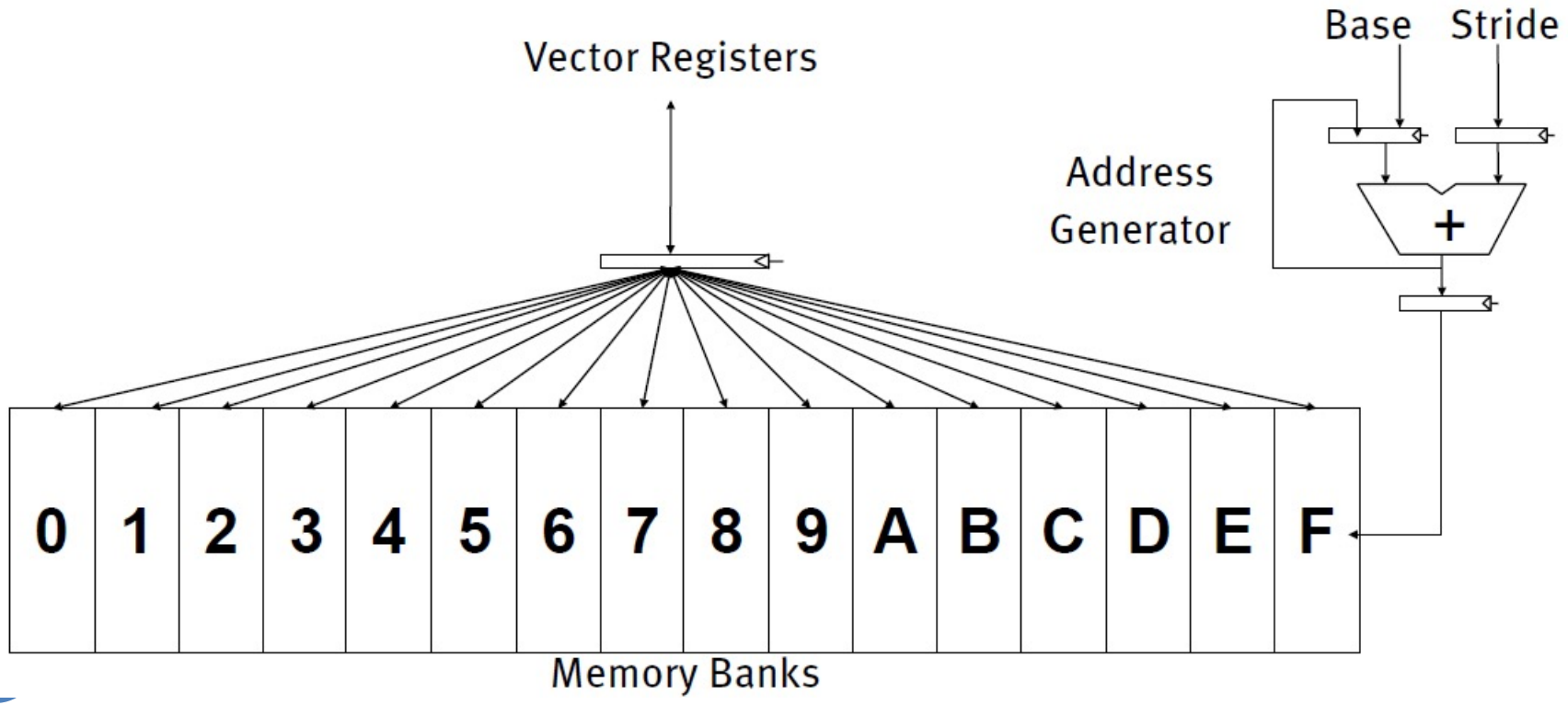


# Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
- 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?

# Vector Memory System

- Cray-1: 16 banks, 64bit wide per bank, 4-cycle bank busy time, 12-cycle latency
- Bank busy time: Cycles between accesses to same bank



# Stride

- Consider:

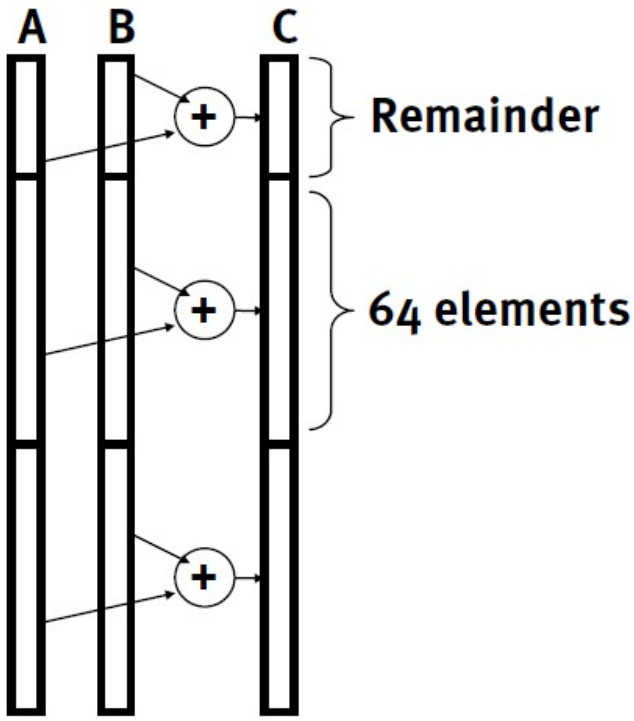
```
for (i = 0; i < 100; i=i+1)
        for (j = 0; j < 100; j=j+1) {
                A[i][j] = 0.0;
                for (k = 0; k < 100; k=k+1)
                A[i][j] = A[i][j] + B[i][k] * D[k][j];
        }
```

- 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 / GCD(stride,#banks) < bank busy time

# Vector Strip-mining

- Problem: Vector register have finite length
- Solution: Break loop into strips that fit into vector registers

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




```
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[i] + B[i] ; /*main operation*/
        low = low + VL; /*start of next vector*/
        VL = MVL; /*reset the length to maximum vector length*/
}
```

# Vector Scatter-Gather

- Vectorize loops with indirect accesses:

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

# Vector Conditional Execution

- Problem: Vectorize loops with conditional codes

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

- Solution: Use vector mask register to “disable” elements

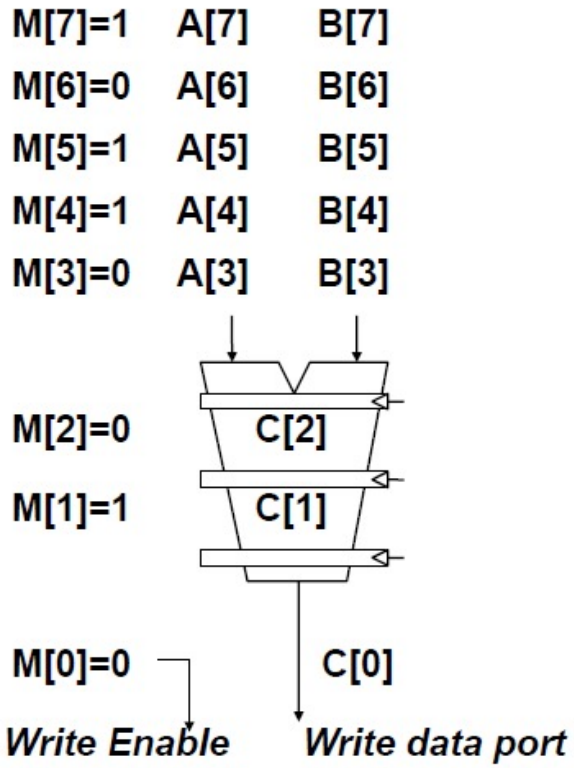
```
LV          V1,Rx       ;load vector X into V1
LV          V2,Ry       ;load vector Y
L.D         F0,#0       ;load FP zero into F0
SNEVS.D     V1,F0       ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2    ;subtract under vector mask
SV          Rx,V1       ;store the result in X
```

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

# Masked Vector Instructions

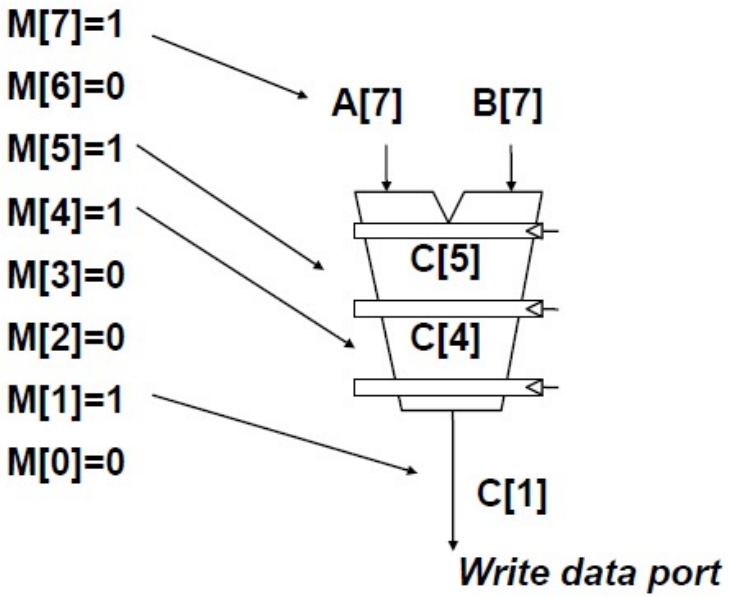
## Simple Implementation

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



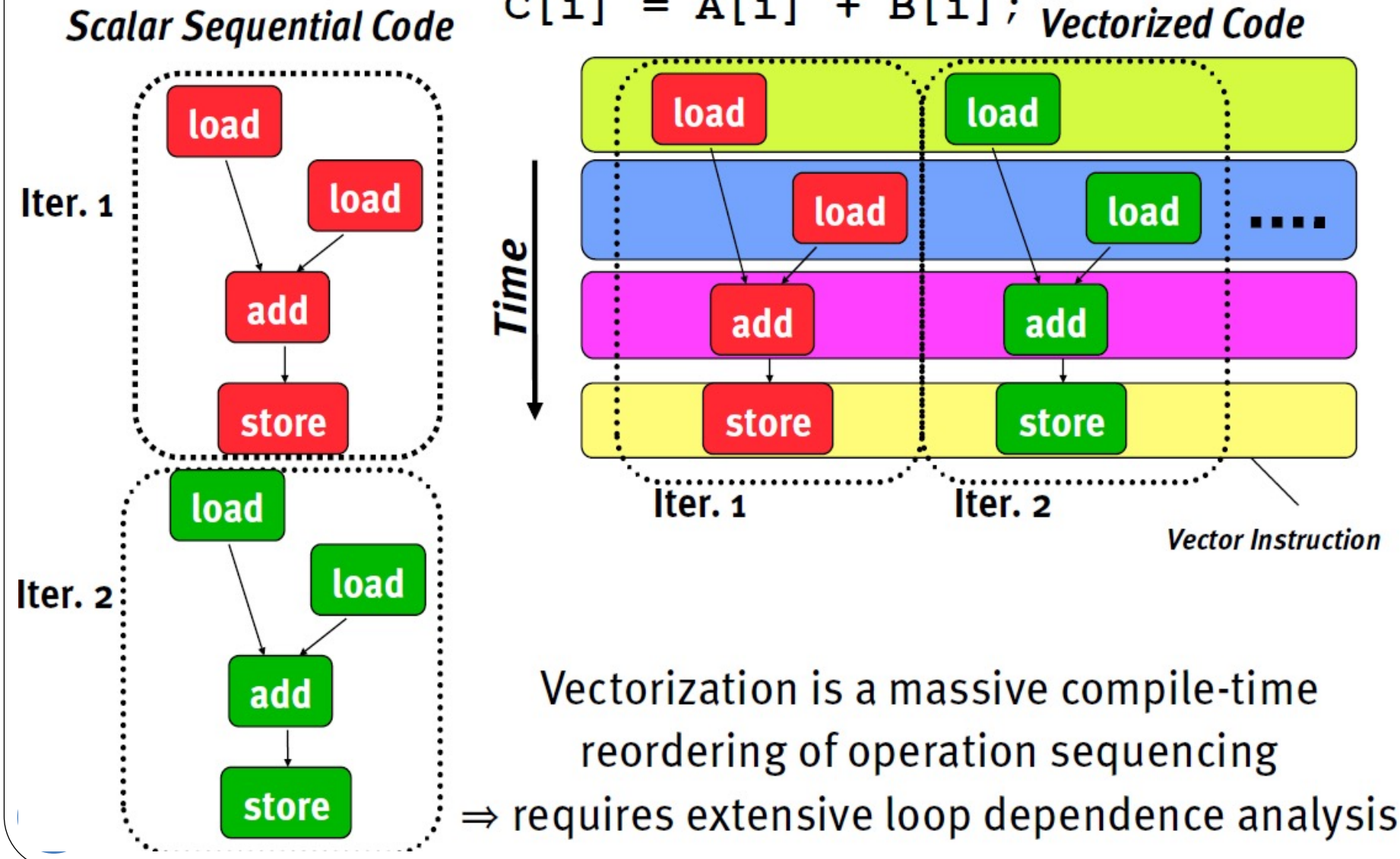

## Density-Time Implementation

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



# 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

# SIMD Extensions

- Media applications operate on data types narrower than the native word size

- One instruction can operate on multiple data elements
  - Example:  disconnect carry chains to “partition” adder

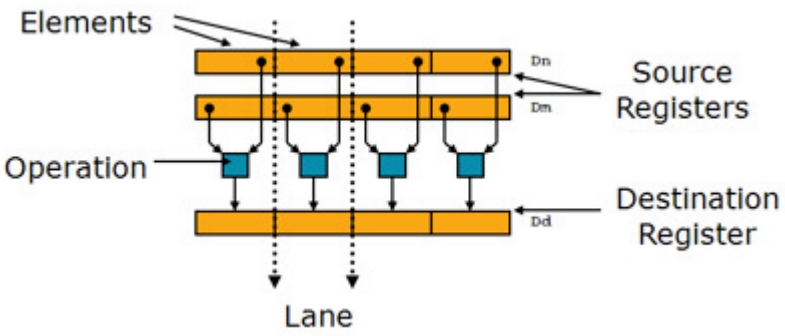
- Limitations, compared to vector instructions:
  - Number of data operands encoded into opcode
  - No sophisticated addressing modes (strided, scatter-gather)
  - No mask registers

# SIMD Implementations

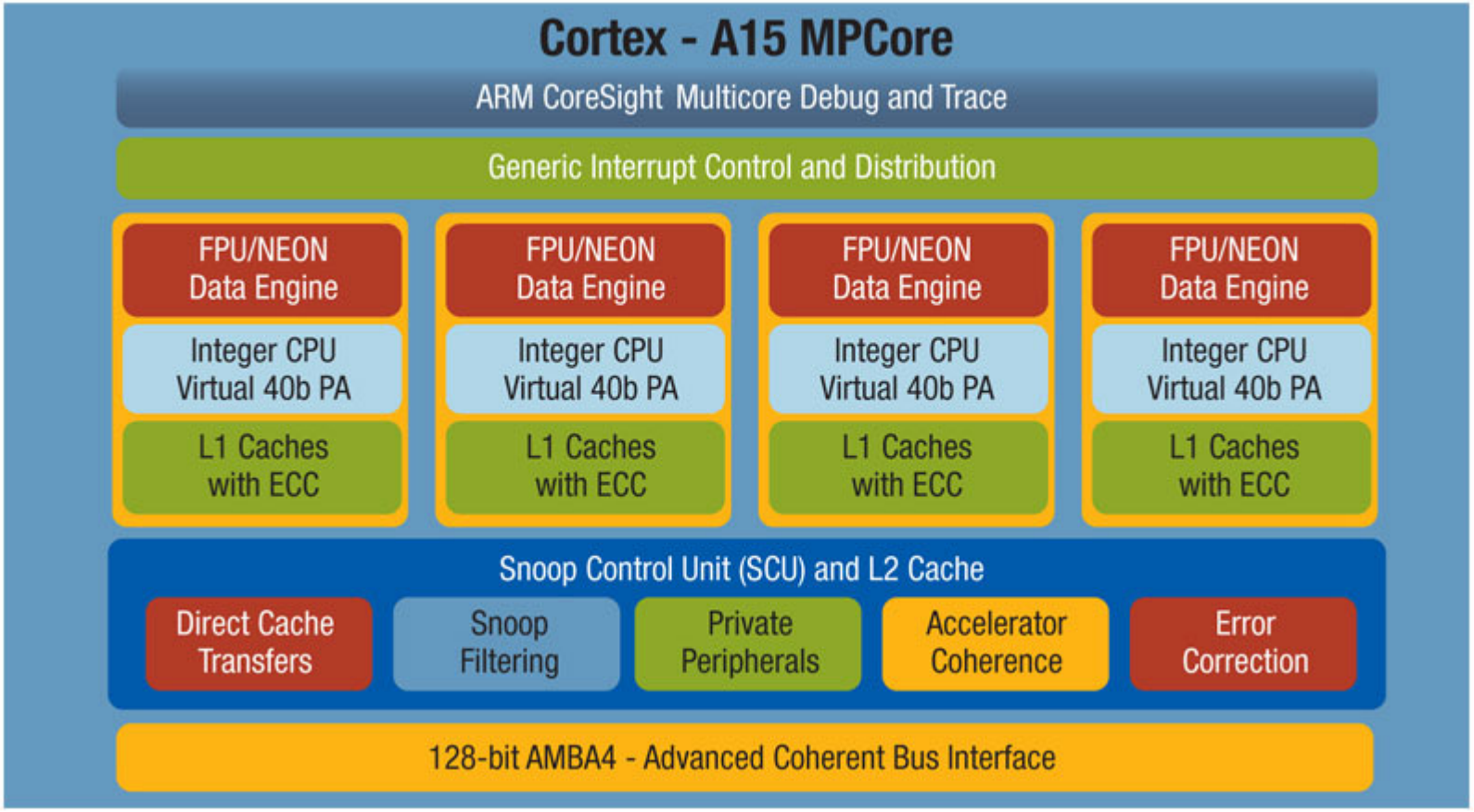
- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
  - ARM Neon

# ARM Neon

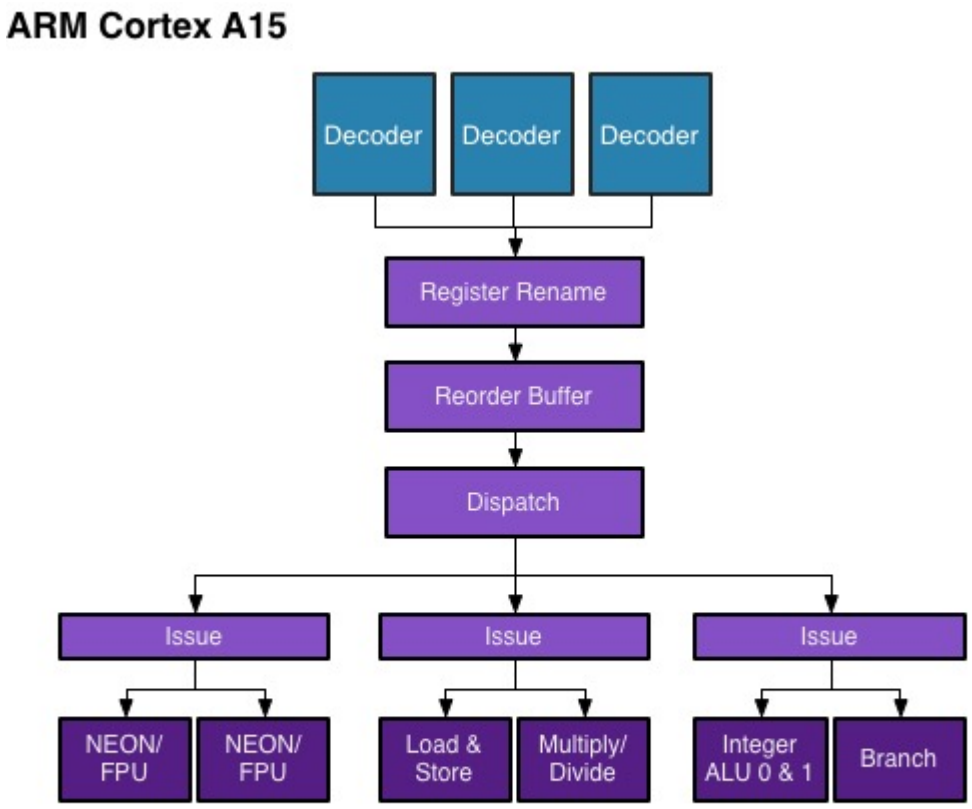
- Wide SIMD data processing architecture
  - 32 register, 64-bits wide (dual view 16 registers, 128-bit)
- Packed SIMD processing
  - Registers are vectors of elements of same data type
  - Data types: 8-bit, 16-bit, 32-bit, 64-bit, 32-bit float
  - Instructions perform the same operation on all lanes



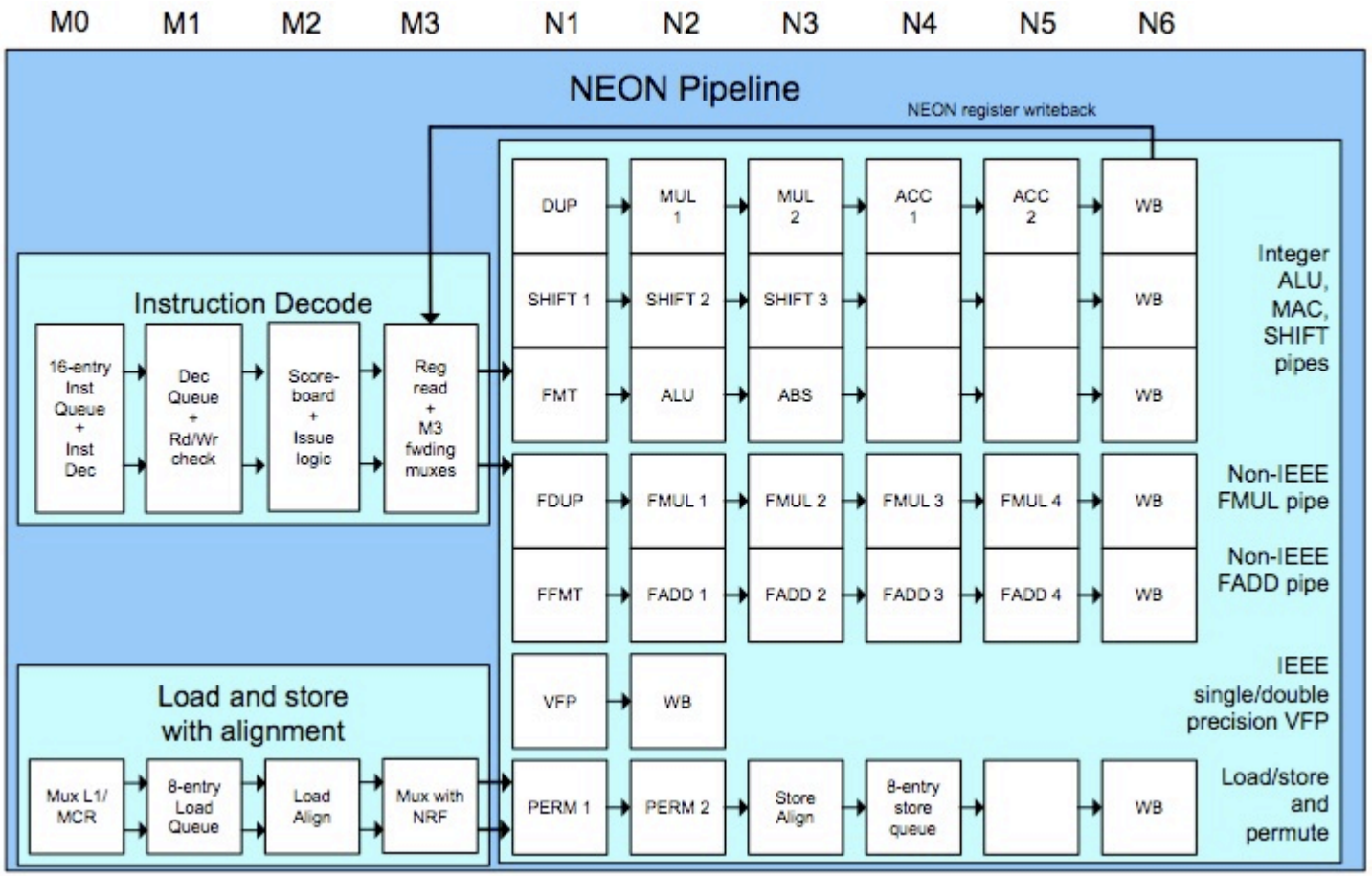
# Neon: Big Picture



# Cortex A15: A Deeper Look



# Neon Pipeline



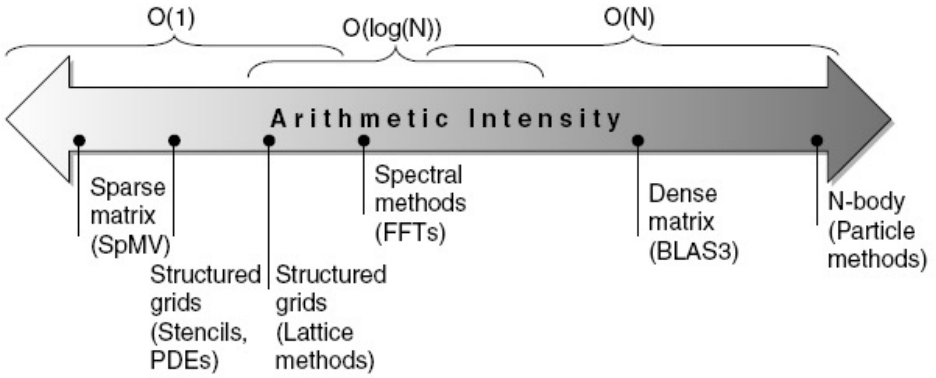
# Example SIMD Code

- Example DAXPY:

```
        L.D       F0,a          ;load scalar a
        MOV       F1, F0        ;copy a into F1 for SIMD MUL
        MOV       F2, F0        ;copy a into F2 for SIMD MUL
        MOV       F3, F0        ;copy a into F3 for SIMD MUL
        DADDIU    R4,Rx,#512    ;last address to load
Loop:   L.4D      F4,0[Rx]      ;load X[i], X[i+1], X[i+2], X[i+3]
        MUL.4D    F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
        L.4D      F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
        ADD.4D    F8,F8,F4       ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
        S.4D      0[Ry],F8      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
        DADDIU    Rx,Rx,#32     ;increment index to X
        DADDIU    Ry,Ry,#32     ;increment index to Y
        DSUBU     R20,R4,Rx     ;compute bound
        BNEZ      R20,Loop      ;check if done
```

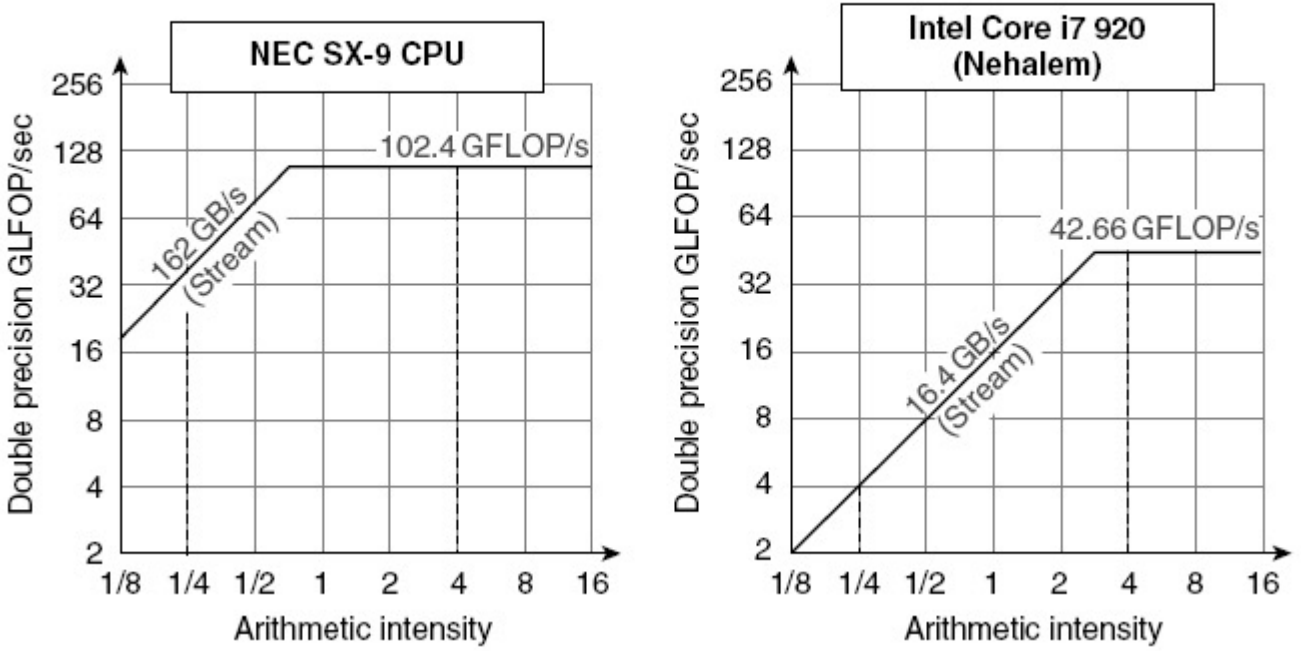
# Roofline Performance Model

- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Floating-point operations per byte read



# Examples

- Attainable GFLOPs/sec Min = (Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)



# Rising Power of Data Parallelism