

# EEL 5764 Computer Architecture

## Lecture 30-31:

- Multithreading and Advanced ILP Techniques
- Introduction to Vector Processing

**Sandip Ray**

Department of Electrical and Computer Engineering  
University of Florida

# Limitations of ILP

- Program structure – limited ILP
- WAW/WAR hazards through memory
- Memory bandwidth and latency
  - Pipeline cannot hide latency to access off-chip cache/mem.
  - Remember “memory wall”?
- Impacts of wide issue width
  - Logic complexity ↑
  - Clock rate ↓
  - Power ↑
- Size of ROB and RS

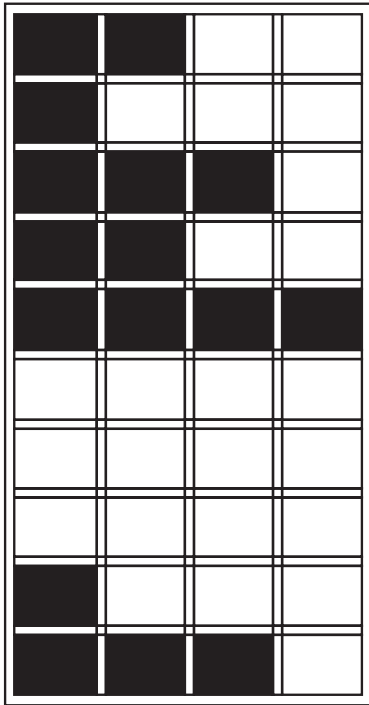
# Multithreading

- Exploiting thread-level parallelism to improve uniprocessor throughput
- Thread-level parallelism (TLP) exists abundantly
  - A property of applications,
  - eg. online transaction processing, SC, web applications,
- Multithreading allows multiple threads to share a processor
  - Each thread is independently controlled by OS
  - With duplicate private state (reg, PC, etc) for each thread
  - Share FUs and memory, etc

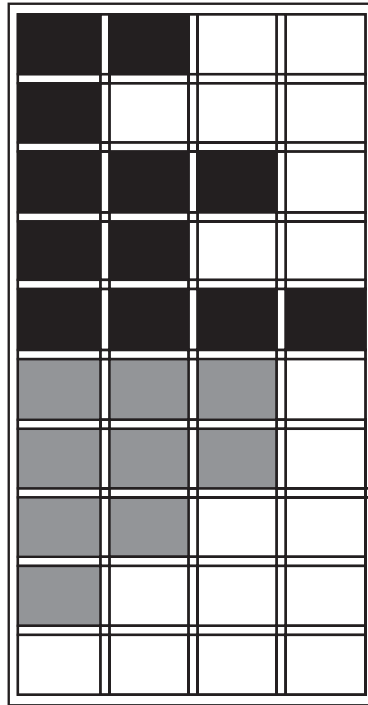
# Multithreading Hardware

Execution slots →

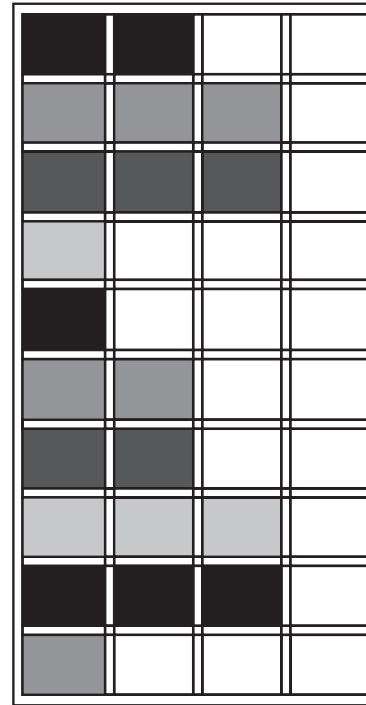
Superscalar



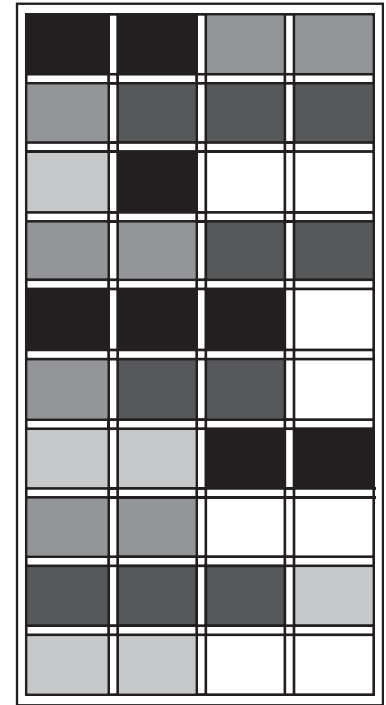
Coarse MT



Fine MT



SMT



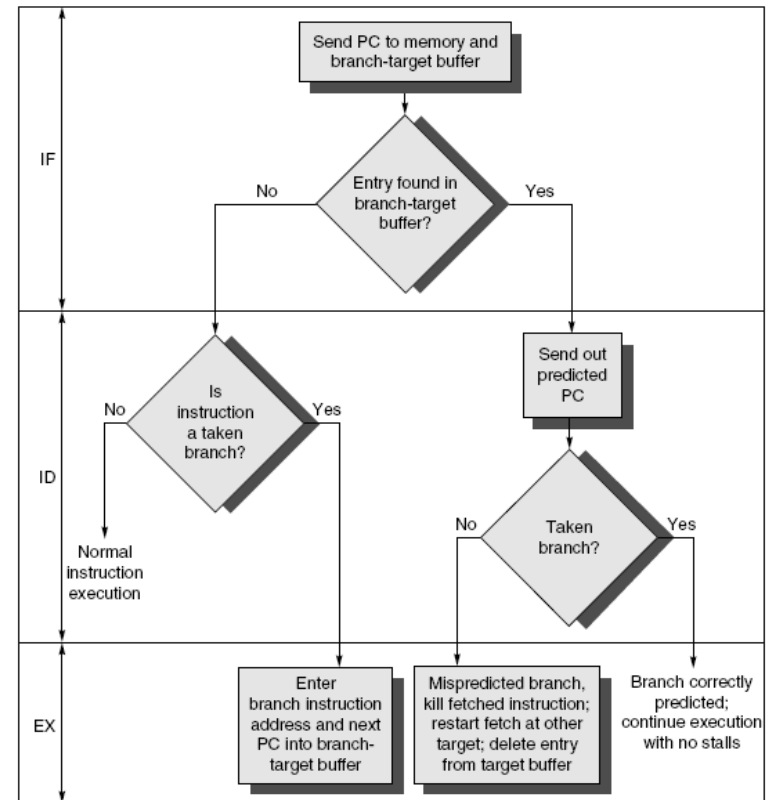
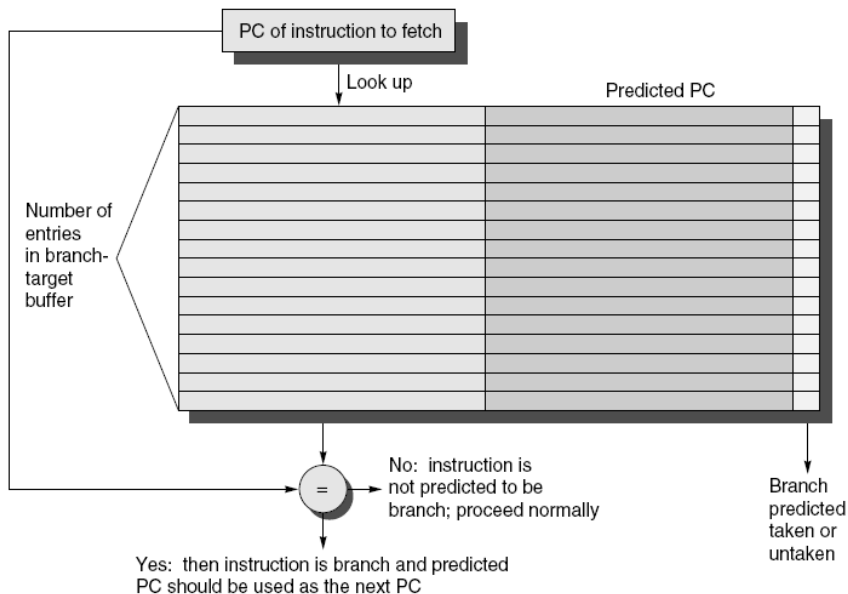
# Addressing ILP Bottlenecks

# Branch-Target Buffer

- Need high instruction bandwidth!

- Branch-Target buffers

- Next PC prediction buffer, indexed by current PC



# Branch Folding

- Optimization:
  - Larger branch-target buffer
  - Add target instruction into buffer to deal with longer decoding time required by larger buffer
  - “Branch folding”

# Return Address Predictor

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls
- Create return address buffer organized as a stack



# Integrated Instruction Fetch Unit

- Design monolithic unit that performs:
  - Branch prediction
  - Instruction prefetch
    - Fetch ahead
  - Instruction memory access and buffering
    - Deal with crossing cache lines

# Register Renaming

- Register renaming vs. reorder buffers
  - Instead of virtual registers from reservation stations and reorder buffer, create a single register pool
    - Contains visible registers and virtual registers
  - Use hardware-based map to rename registers during issue
  - WAW and WAR hazards are avoided
  - Speculation recovery occurs by copying during commit
  - Still need a ROB-like queue to update table in order
  - Simplifies commit:
    - Record that mapping between architectural register and physical register is no longer speculative
    - Free up physical register used to hold older value
    - In other words: SWAP physical registers on commit
  - Physical register de-allocation is more difficult

# Integrated Issue and Renaming

- Combining instruction issue with register renaming:
  - Issue logic pre-reserves enough physical registers for the bundle (fixed number?)
  - Issue logic finds dependencies within bundle, maps registers as necessary
  - Issue logic finds dependencies between current bundle and already in-flight bundles, maps registers as necessary

# How Much?

- How much to speculate
  - Mis-speculation degrades performance and power relative to no speculation
    - May cause additional misses (cache, TLB)
  - Prevent speculative code from causing higher costing misses (e.g. L2)
- Speculating through multiple branches
  - Complicates speculation recovery
  - No processor can resolve multiple branches per cycle

# Energy Efficiency

- Speculation and energy efficiency
  - Note: speculation is only energy efficient when it significantly improves performance
- Value prediction
  - Uses:
    - Loads that load from a constant pool
    - Instruction that produces a value from a small set of values
  - Not been incorporated into modern processors
  - Similar idea--*address aliasing prediction*--is used on some processors

# Vector Processing

# Flynn's Classification

	Single Data	Multiple Data
Single Instruction	<b>SISD</b>	<b>SIMD</b>
Multiple Instructions	<b>Not Exists</b>	<b>MIMD</b>

# Data Parallelism

- Concurrency arises from performing the **same operations on different pieces of data**
  - Single instruction multiple data (SIMD)
  - E.g., dot product of two vectors
- Contrast with data flow
  - Concurrency arises from executing different operations in parallel (in a data driven manner)
- Contrast with thread (“control”) parallelism
  - Concurrency arises from executing different threads of control in parallel



# Introduction

- 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
  - Fetch one instruction for many data operation
  - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially
  - Unlike MIMD

# SIMD Processing

- Single instruction operates on multiple data elements
  - In time – PEs are pipelined
  - In space – Multiples PEs
- Time-Space Duality
  - **Array processor** – instruction operates on multiple data at the same time, needs duplicates of PEs
  - **Vector processor** – instruction operates on multiple data in consecutive time steps, PEs needs to be pipelined

# Vector Architecture

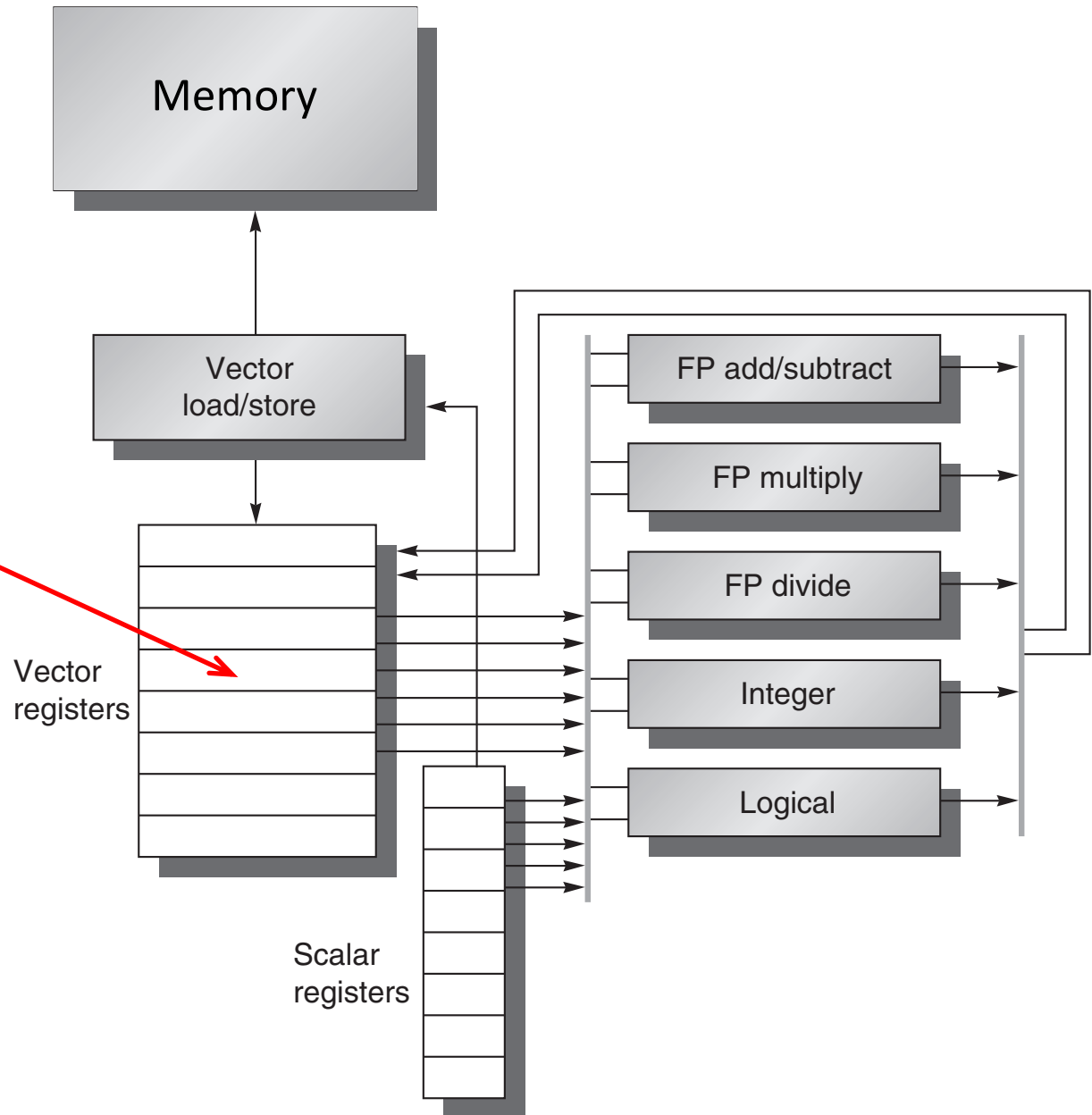
- A vector is a one-dimensional array of numbers
  - Many scientific/commercial applications use vector
- An instruction operates on vectors, instead of scalar values
  - Each instruction generates a lot of work
- Basic idea:
  - Read vectors of data elements into **vector registers**
  - Vector FUs operate on those registers in a pipelined manner one element at a time
  - Disperse the results back into memory

# Vector Architecture

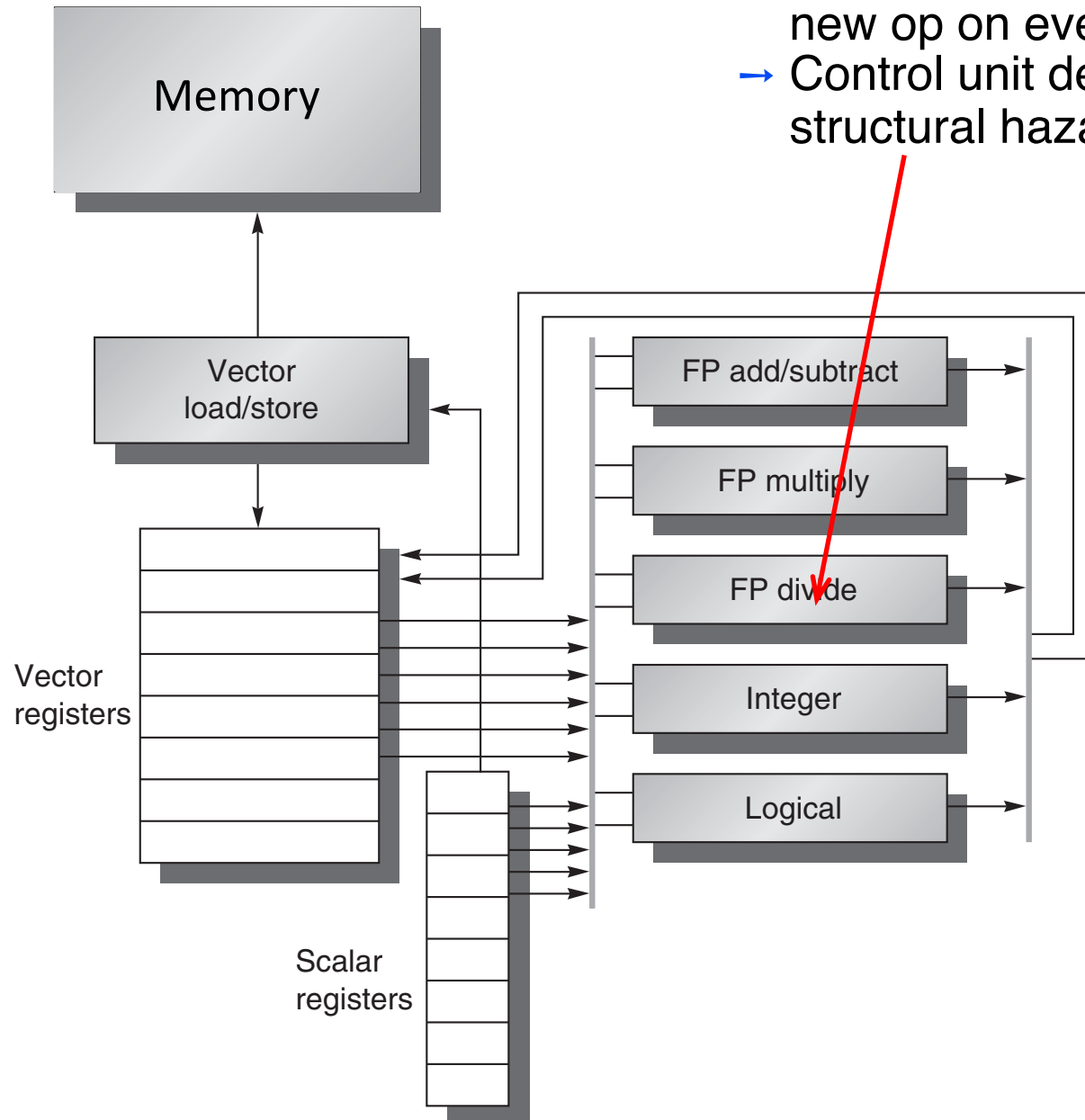
- FUs are deeply pipelined
  - No intra-vector dependence – no HW interlocking within a vector
  - No control-flow within a vector
- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth
- Deliver high performance without energy/design complexity of out-of-order superscalar processor
  - Increase performance of in-order simple scalar

# VMIPS

- Each register holds a 64-element, 64 bits/element vector
- Register file has 16 read ports and 8 write ports
- Other special purpose registers, i.e VLEN.



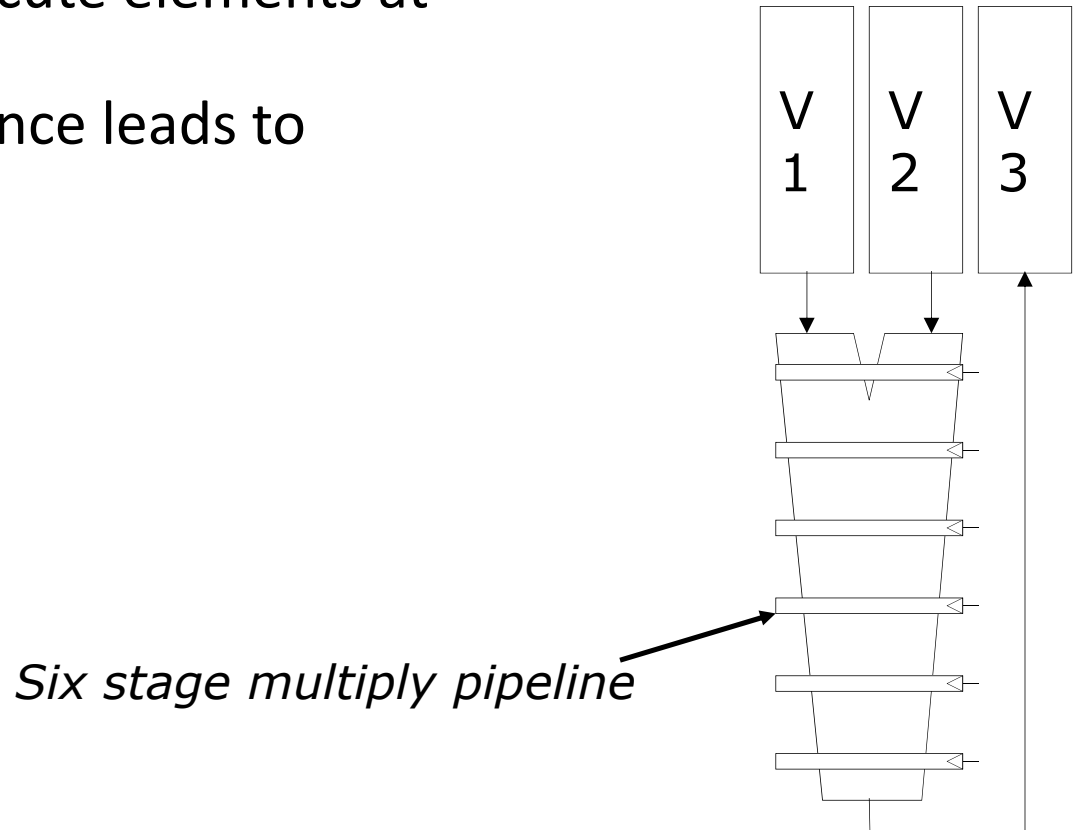
# VMIPS



- Fully pipelined – start a new op on every cycle
- Control unit detects data & structural hazards

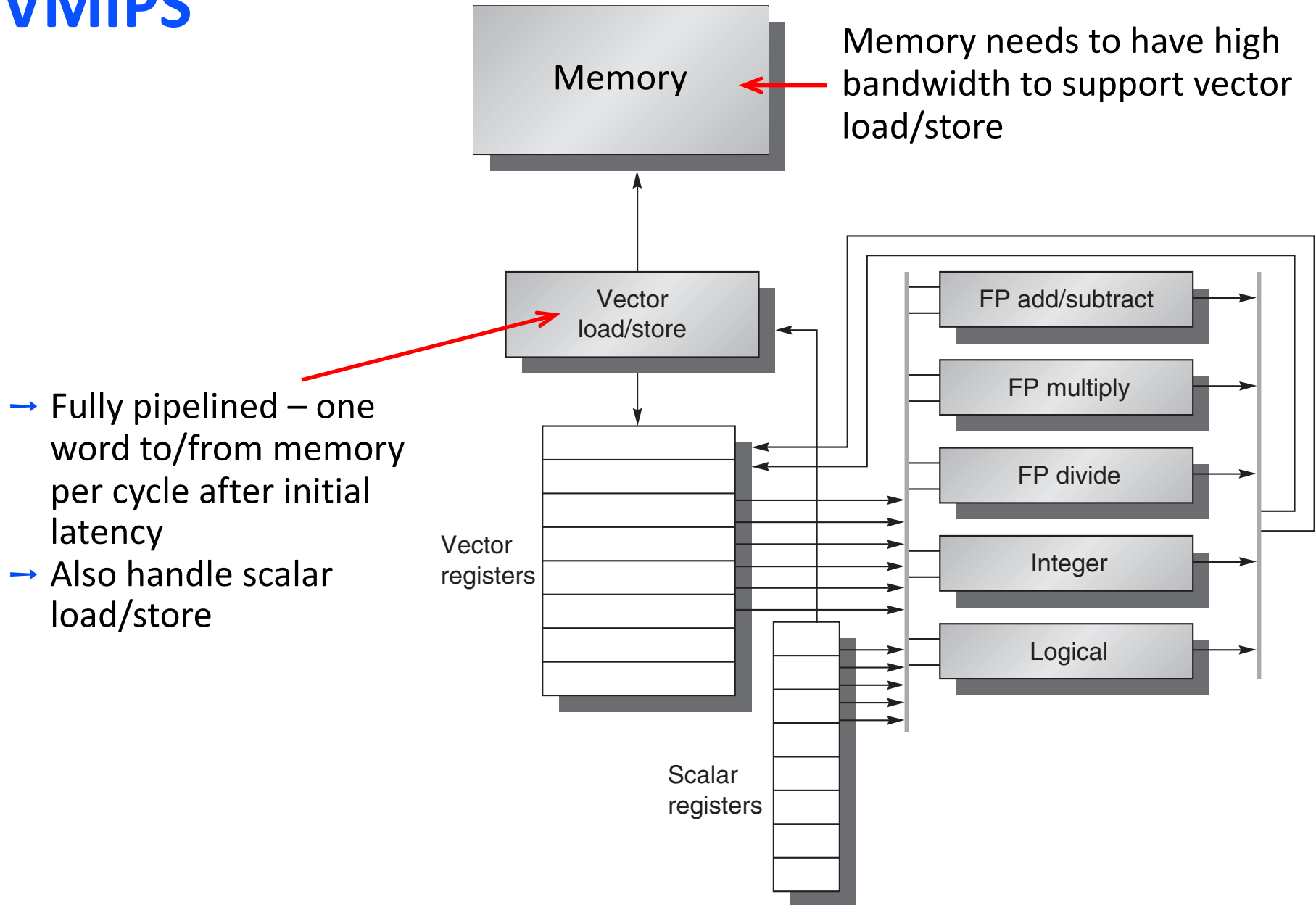
# Vector Functional Units

- Use deep pipeline to execute elements at fast speed
- No intra-vector dependence leads to simple pipeline control



$$V3 \leftarrow v1 * v2$$

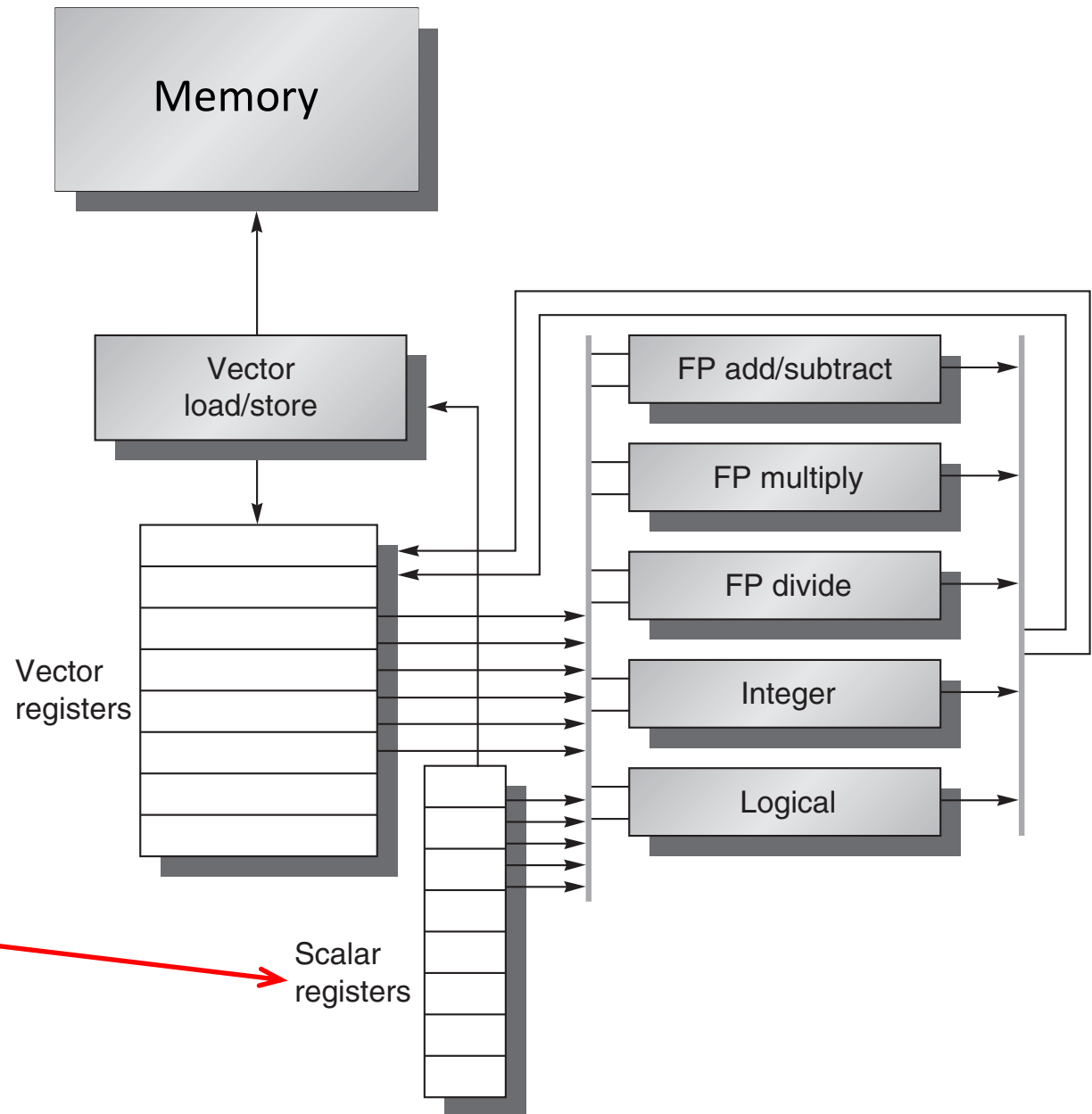
# VMIPS





# VMIPS

- 32 GP registers
- 32 FP registers



# VMIPS Vector Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address
- Example: DAXPY –  $Y = a * X + Y$  ; vector length = 64
  - 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        V4,V2,V3        ; add
  - SV            Ry,V4            ; store the result
- Requires 6 instructions vs. almost 600 for MIPS

# VMIPS Vector Instructions

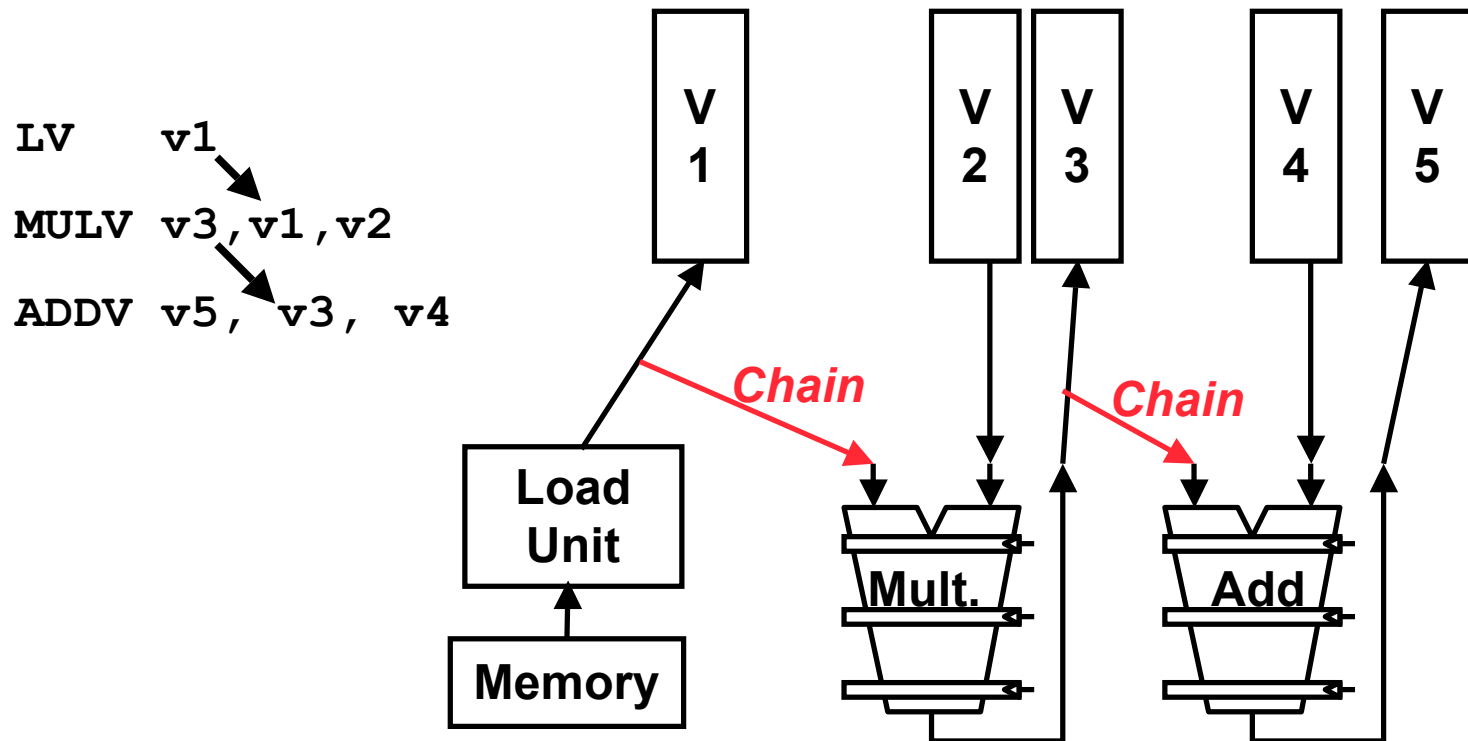
- Example: DAXPY –  $\mathbf{Y} = a * \mathbf{X} + \mathbf{Y}$

	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

- Almost 600 MIPS instructions

# Vector Chaining – Handling D-Dependence

- Vector version of forwarding
- Next vector operation starts when the result from previous operation on the 1<sup>st</sup> element is finished.

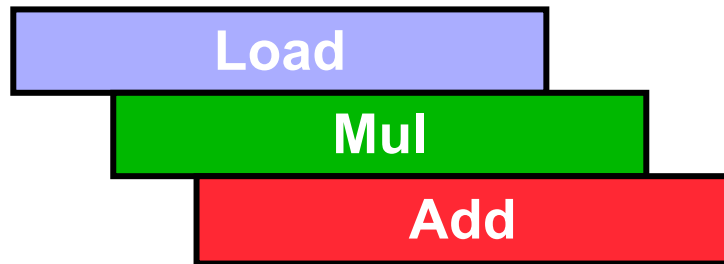


# Vector Chaining

- Without chaining



- With Chaining



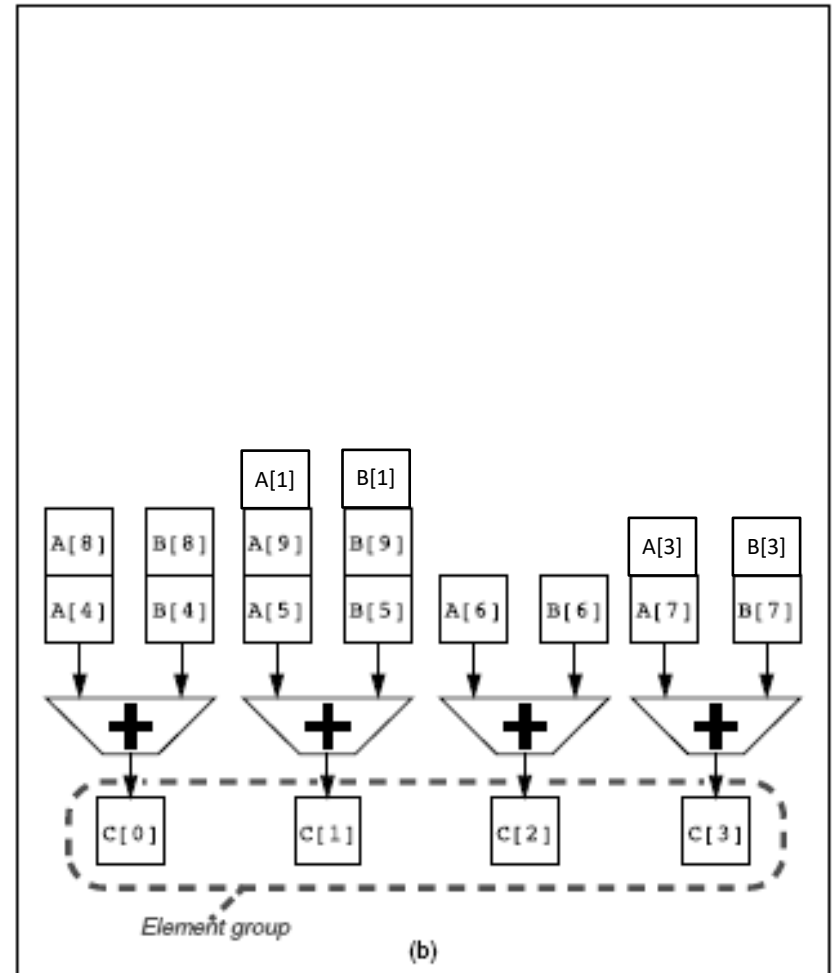
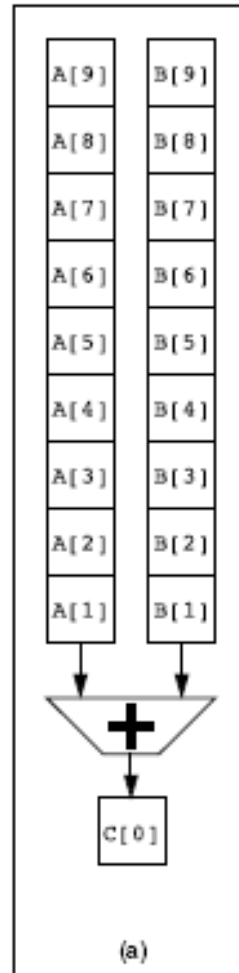
# Multiple Lanes

- How can a vector processor execute a single vector faster than one element in a clock cycle?

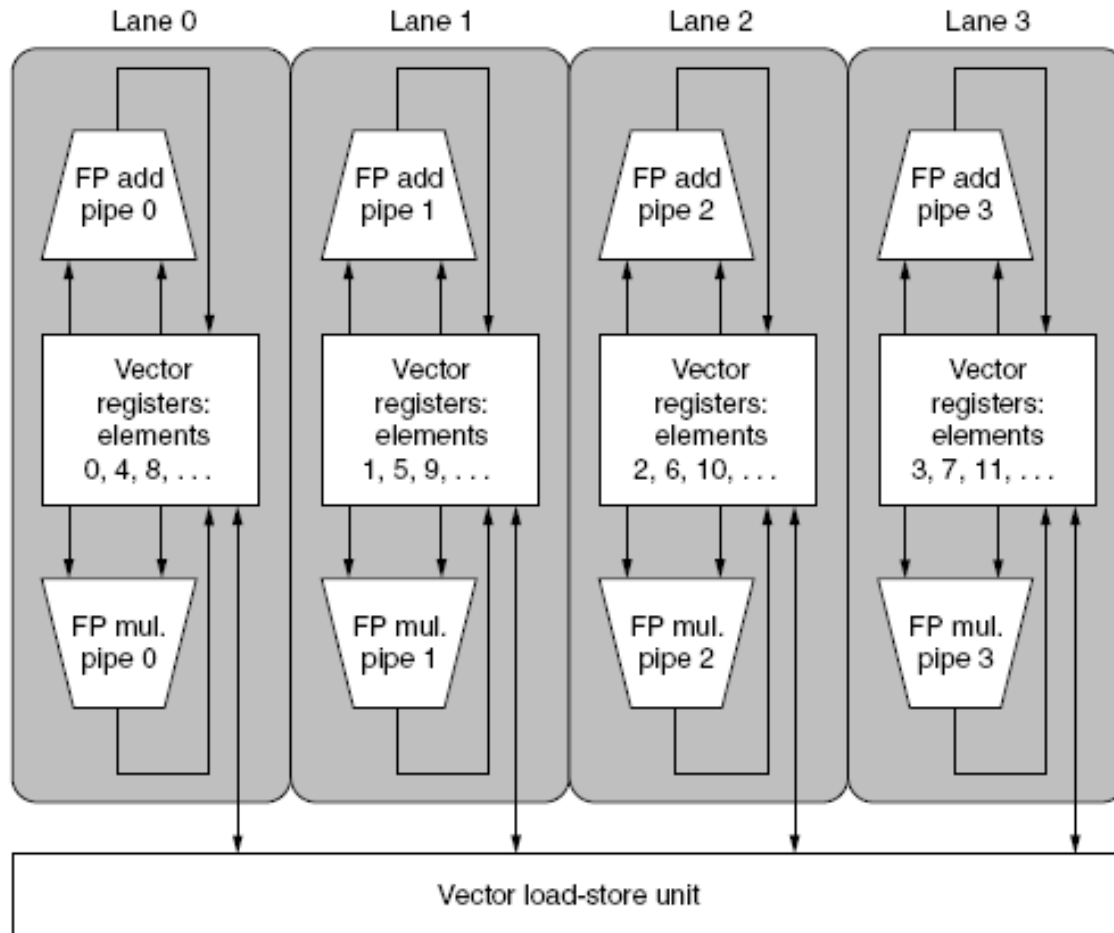
# Multiple Lanes

- Goal: Execute >1 element per cycle.

- Use multiple FUs to improve performance of a single vector add operation  $C = A + B$
- Elements are interleaved across FUs
- Increase performance to 4 elements per cycle from 1 per cycle



# Multiple Lanes



Both applications and HW architecture must support long vectors to take advantage of the higher processing power of multiple lanes

Doubling performance with the *same* power – by reducing clock rate by half, and doubling lanes

- Element *n* of vector register *A* is “hardwired” to element *n* of vector register *B*
- Each lane handles a portion of vector registers




# Vector Length Register

- What if vector length is different from the length of vector registers? Or
- The length is known only at runtime?

Known only at runtime

**VLR** controls the length of every vector operation



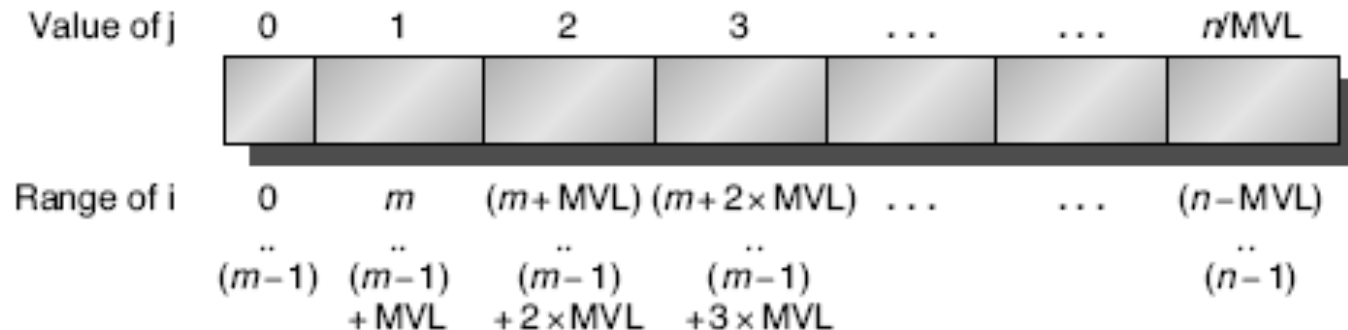
```
for (i=0; i < n; i = i + 1)  
    Y[i] = a * X[i] + Y[i]
```

- What if *n* is not known at compile time, but may be larger than the *maximal vector length (MVL)*?

# Strip Mining

```
for (i=0; i < n; i = i + 1)
    Y[i] = a * X[i] + Y[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 * X[i] + Y[i];          /*main operation*/
    low = low + VL;                          /*start of next vector*/
    VL = MVL;    /*reset the length to maximum vector length*/
}
```



# Vector Mask Register

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

IF statement introduces control dependence,  
reduces the level of parallelism

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 VMR(i) to 1 if V1(i)≠F0
SUBVV.D	V1,V2	;subtract under vector mask
SV	Rx,V1	;store the result in X

# Vector Mask Register

## Simple Implementation

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

M[7]=1 A[7] B[7]

M[6]=0 A[6] B[6]

M[5]=1 A[5] B[5]

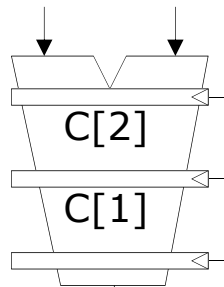
M[4]=1 A[4] B[4]

M[3]=0 A[3] B[3]

M[2]=0

M[1]=1

M[0]=0



Write Enable

Write data port

## Density-Time Implementation

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

M[7]=1

M[6]=0

M[5]=1

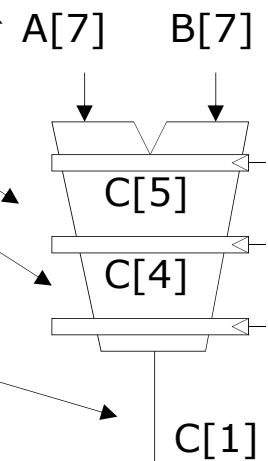
M[4]=1

M[3]=0

M[2]=0

M[1]=1

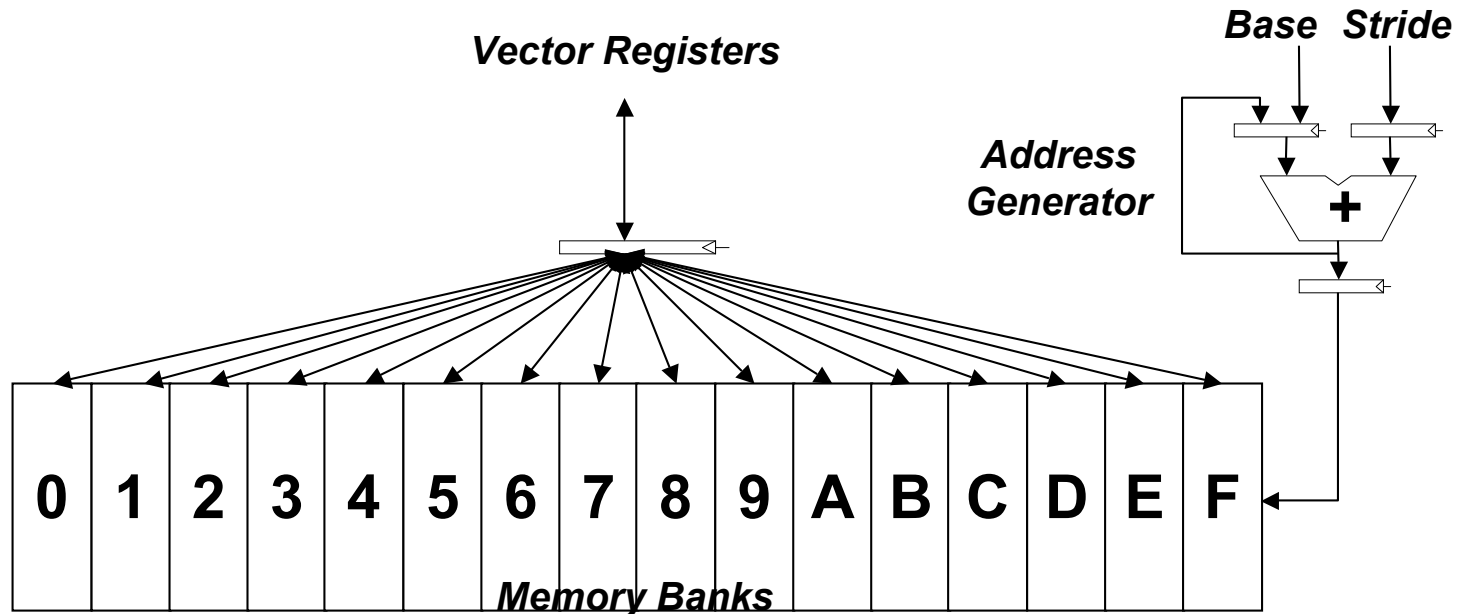
M[0]=0



Write data port

# Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Load or store non sequential words
  - Support multiple vector processors sharing the same memory



# Memory Banks – Example

**Question:** A vector machine has 32 processors, each can generate 4 loads and 2 stores per cycle. SRAM cycle time is 7 cycles. What is the minimum number of memory banks needed to allow all processors to run at full bandwidth?

**Answer:** the max number of memory references per cycle is  $32 * 6 = 192$ .

The min number of memory banks is  $192 * 7 = 1344$ .

# Vector Stride – Handling Multi-D Arrays

```
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];
    }
```

→ Assume all A, B, D are 100x100 matrices

→ How are matrices stored in memory?

# Memory Layout of Matrices

→ Consider  $M[i][j]$

Row-major

$M[0][0]$
$M[0][1]$
...
$M[0][j-1]$
$M[1][0]$
...
$M[1][j-1]$
$M[2][0]$
...
$M[2][j-1]$
...

Column-major

$M[0][0]$
$M[1][0]$
...
$M[i-1][0]$
$M[0][1]$
...
$M[i-1][1]$
$M[0][2]$
...
$M[i-1][2]$
...



# Vector Stride

```
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];
    }
```

- Assume all A, B, D are 100x100 matrices
- **Stride** is the distance that separate elements in a array to be gathered into a single register.
- For D, in row-major layout, the stride is 100 double words, or 800 bytes

# Vector Stride

- Vector base address and stride are stored in GP registers.
- Memory bank conflicts due to Mem access with stride

**Question:** 8 Mem banks, with busy time of 6 cycles for each bank.  
Mem latency is 12 cycles. Time to access 64 data with stride of 1 and 32.

**Answer:** When stride = 1, access time =  $12 + 64 = 76$ .  
When stride = 32, access time =  $12 + 1 + 6 * 63 = 391$

# Gather and Scatter

- In Sparse matrices most elements are 0

1	0	0	3	0	0	0	0	0	6
---	---	---	---	---	---	---	---	---	---

- Sparse matrices are stored in a compact format, and accessed indirectly.

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

- A and C have the same number of non-zero elements  
→ **Index vectors** are K and M are the same size.

# Gather and Scatter

- Access sparse matrices using index vectors.
- **Gather load** takes an index vector and fetches vector elements whose addresses are  $\text{base} + \text{offset}$ 
  - Offsets are stored in the index vector
- **Scatter store** values in vector registers to Mem addresses computed as above.
- Allows sparse matrix operations to run in vector mode