Vector Architectures

Professor Hugh C. Lauer
CS-4515, System Programming Concepts

Overview

- Vector architecture outline
- Vector Execution Time
- Improvements to Vector Architectures
- Performance summary

The chapter is much larger than this….

Intro to Data-Level Parallelism

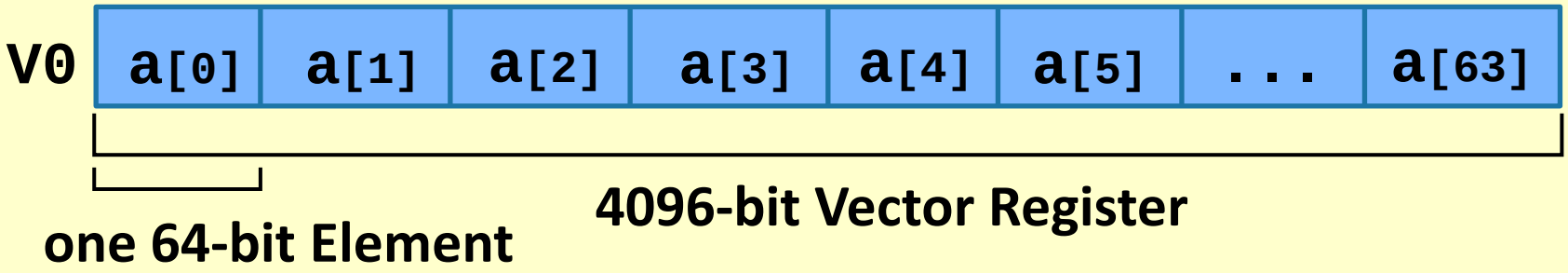
- **The goal: simultaneous operations on large sets of data**
  - SIMD: Single Instruction, Multiple Data
- **Many implementations have developed for these kind of operations**
  - **Vector architectures**
  - SIMD Multimedia Instructions
  - GPUs

Applications of Data Parallelism

- **Any application that involves number crunching on a lot of similar data:**
  - Graphics and image processing
  - Digital Signal Processing (DSP)
  - Physics Simulations
  - Searching and Sorting
  - Financial Simulations
  - Etc.

Vector Architectures: The Basics

- **Vector architectures provide pipelined execution of many data operations**
- **Vector Register: register file containing multiple elements of a set of data stored sequentially**
  - One instruction performs an operation on an entire vector of data
  - Operations are performed in parallel on independent elements



The VMIPS Architecture

- **Textbook model of vector architecture (stylized)**
  - ISA is based on MIPS
  - Architecture is based on the Cray-1
  - Idealized example of how a vector architecture *might* work
- **VMIPS Vector registers**
  - 8 registers, each with 64 64-bit elements
  - 16 read ports and 8 write ports for communication with other units
  - Connected with crossbar switches (expensive)

The VMIPS Architecture (cont'd)

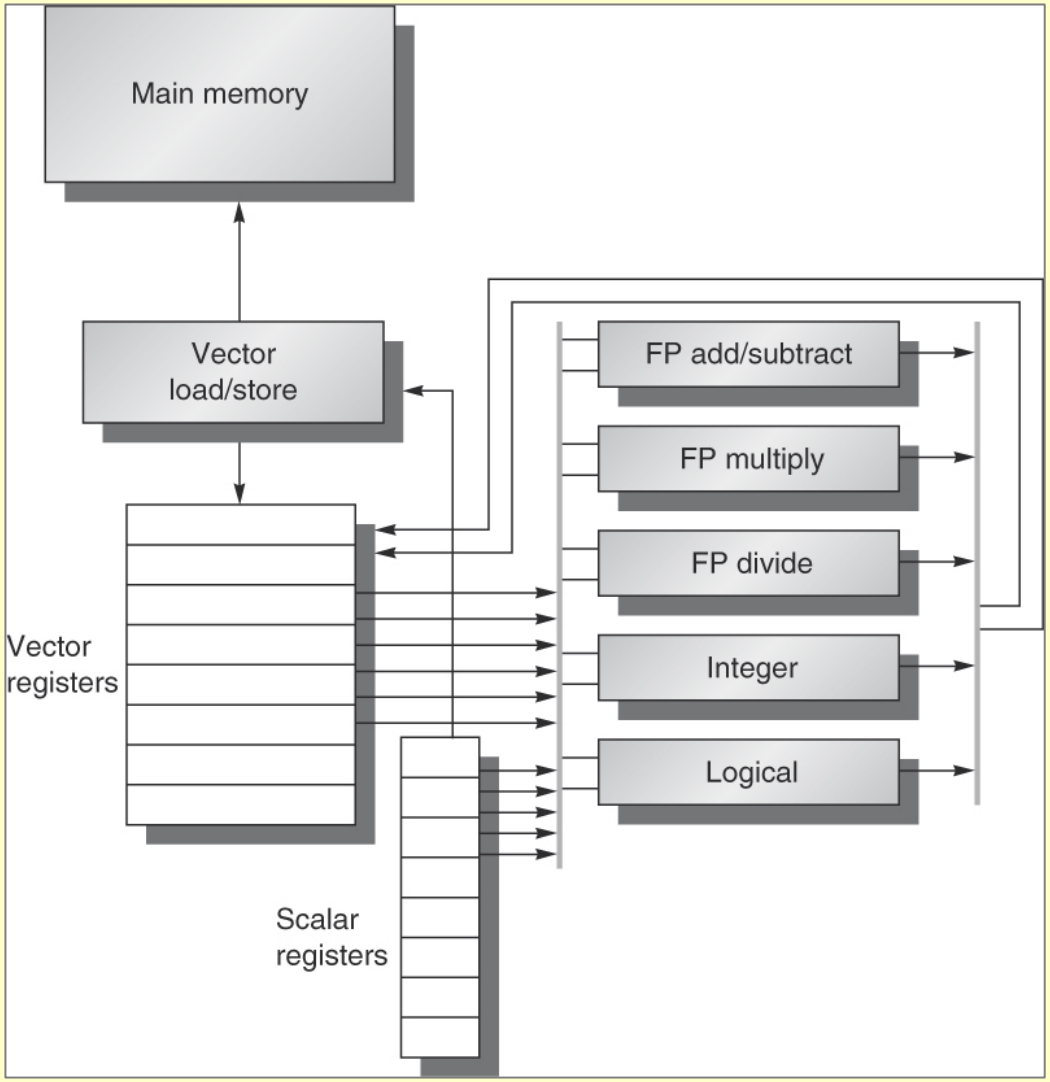
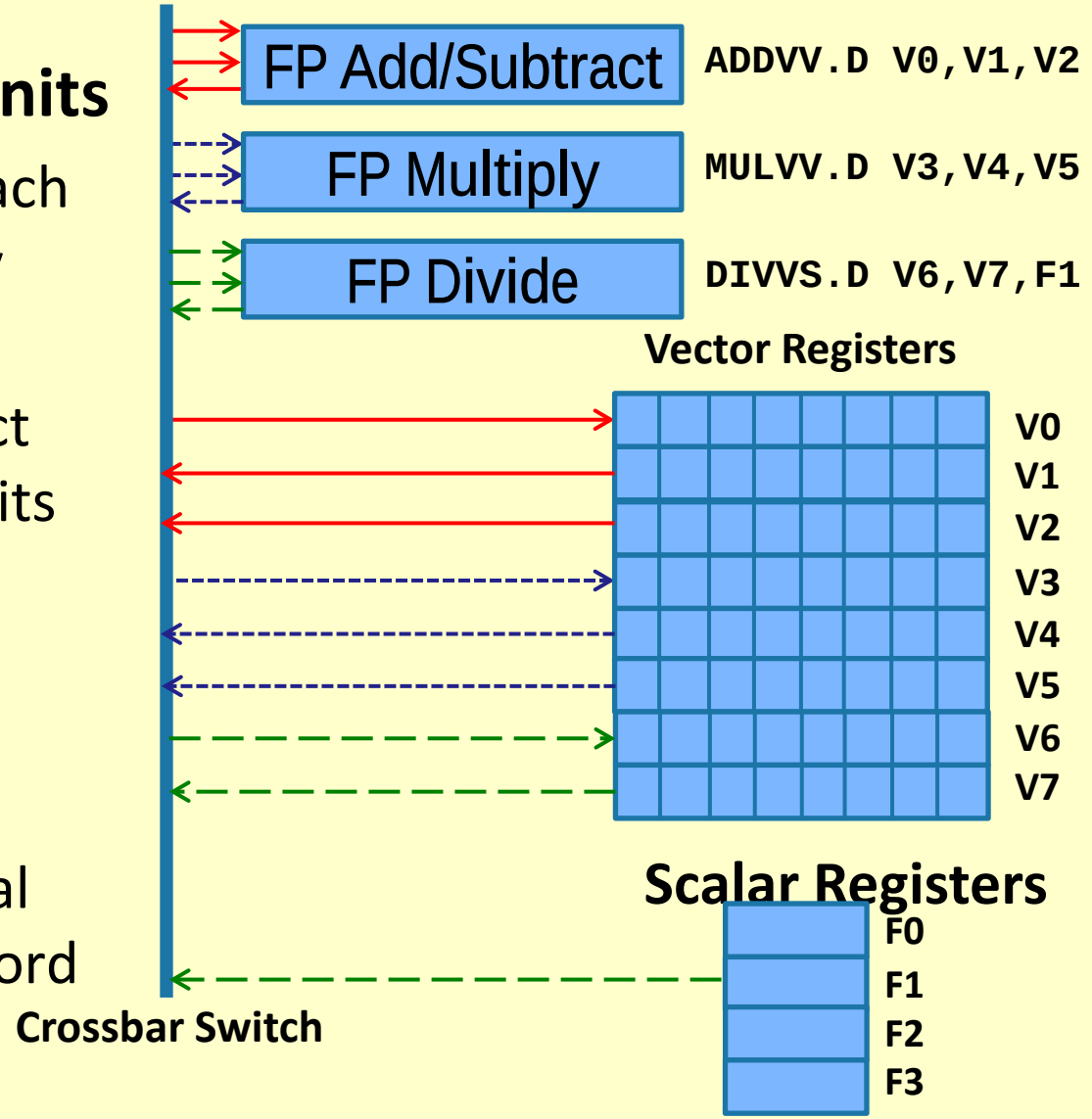


Figure 4.2

The VMIPS Architecture (continued)

- **Vector Functional Units**
  - Separate units for each operation, each fully pipelined
  - Control unit to detect hazards between units
- **Scalar Registers**
  - As in ordinary MIPS
- **Load/Store Unit**
  - Fully pipelined—ideal bandwidth of one word per clock cycle



VMIPS Instruction Set

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 S--VS.D	V1,V2 V1,F0	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.
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 MFC1	VLR,R1 R1,VLR	Move contents of R1 to vector-length register VL. Move the contents of vector-length register VL to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to vector-mask register VM. Move contents of vector-mask register VM to F0.

Figure 4.3

Loading and Storing Vectors

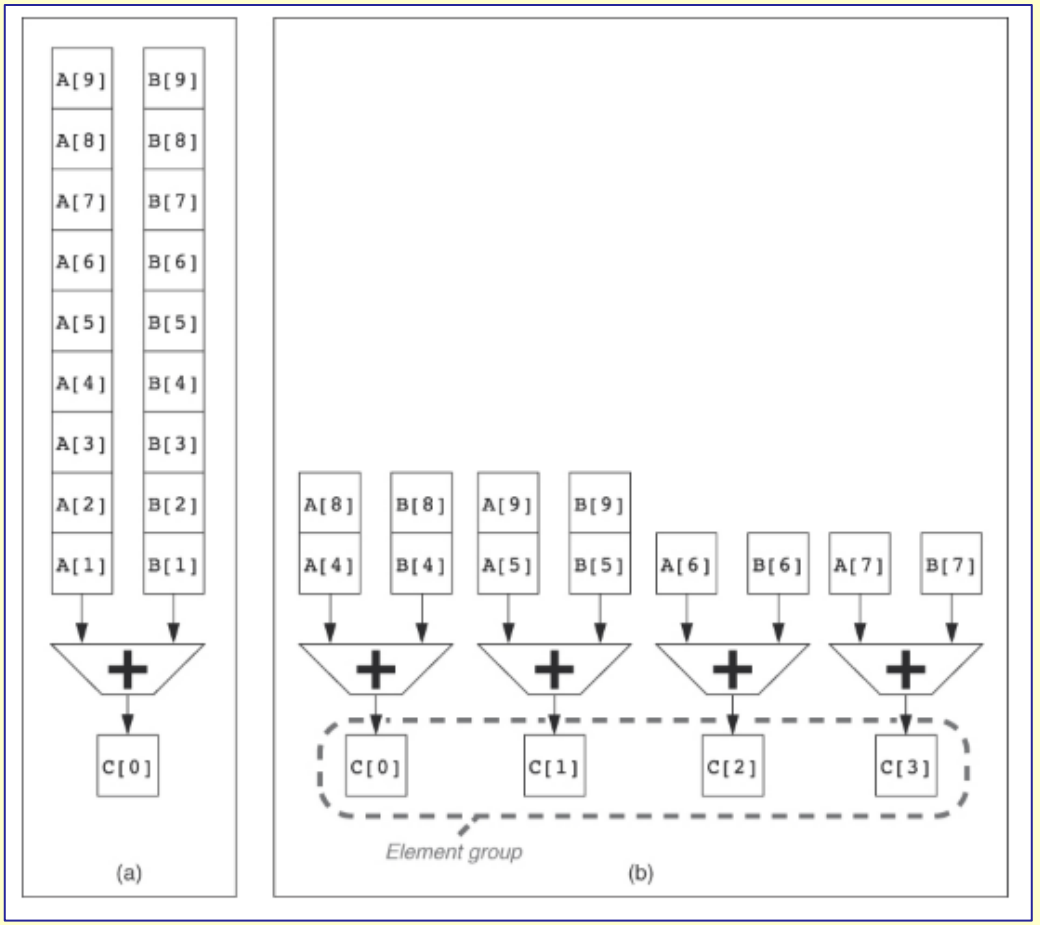
- **A vector load or store instruction reads or writes to an entire vector at once**
- **Long latency to fetch or store an entire vector, rather than a latency for each element**
  - Latency is amortized over each element in the vector
- **Memory operations are heavily pipelined**
  - "Hides" latency by taking advantage of memory bandwidth

Execution Time

Vector Architectures

3 Factors Affect Execution Time

1. Structural Hazards
2. Data Dependences
3. **Length of Vectors**



Adapted from Figure 4.4

Measuring Vector Operations

- **Single Vector Instruction (Execution Time)**
  - *Initiation Rate:* Rate at which Vector Unit consumes vector elements
  - [Execution Time]=[vector length]/[Initiation Rate]

- **Most Vector processors implement pipelining and multiple *lanes***
  - Higher initiation rate
  - Typically *n* elements per cycle

Measuring Vector Operations

- ***Convoy***
  - *Convoy:* Set of vector instructions that could potentially execute together
    - (w/o structural hazards)
  - Unit by which long instruction sequences are measured

- ***Chaining:***
  - Allows vector operations to start as soon as individual elements of its operands become available
    - I.e., as output of other operands

Measuring Vector Operations

- ***Chime:*** **execution time for one Convoy**
  - Ignores vector-length dependent calculation overhead
  - Better for measuring longer vectors
  - VMIPS
    - **[Execution Time] = [# Chimes] × [Length of Vector]**

Example

```
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 two vectors
SV        Ry,V4       ;store the sum
```

Convoys:

1	LV — MULVS.D
2	LV — ADDVV.D
3	SV

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 64 x 3 = 192 clock cycles

Questions?

Vector Benefits — DAXPY Loop

- **DAXPY: "Double-precision A × X Plus Y"**
  - Y = a × X + Y
    - a is scalar; X & Y are vectors
  - Used for benchmarking performance
- **Vector multiplication requires extra overhead in ordinary (non-vectorized) MIPS-like processors**
- **How would you do this in MIPS? In VIMPS?**

DAXPY Loop — (unvectorized) MIPS

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

```
F0: a
```

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

```
F0: a
R4: last address
```

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: value at X[Rx]

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: X[Rx] * a

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: X[Rx] * a

F4: Y[Ry]

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: X[Rx] * a

F4: X[Rx] * a + Y[Ry]

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: X[Rx] * a

F4: X[Rx] * a + Y[Ry]

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: X[Rx] * a

F4: X[Rx] * a + Y[Ry]

Rx: Rx + [cell size]

Ry: Ry + [cell size]

From Example on pg. 267 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```

Registers

F0: a

R4: last address

F2: X[Rx] * a

F4: X[Rx] * a + Y[Ry]

Rx: Rx + [cell size]

Ry: Ry + [cell size]

R20: boundary check

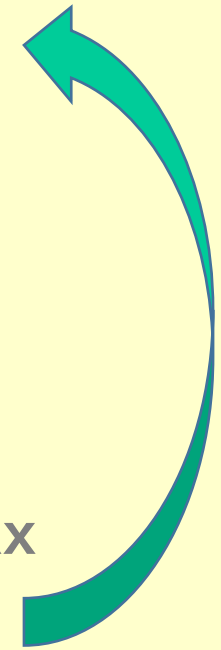
From Example on pg. 267 (4.2)

From Example on pg. 26 (4.2)

DAXPY Loop — (unvectorized) MIPS (con't)

MIPS

```
      L.D     F0,a
      DADDIU  R4,Rx,#512
Loop: L.D     F2,0(Rx)
      MUL.D   F2,F2,F0
      L.D     F4,0(Ry)
      ADD.D   F4,F4,F2
      S.D     F4,9(Ry)
      DADDIU  Rx,Rx,#8
      DADDIU  Ry,Ry,#8
      DSUBU   R20,R4,Rx
      BNEZ    R20,Loop
```



Registers

F0: a

R4: last address

F2: X[Rx] * a

F4: X[Rx] * a + Y[Ry]

Rx: Rx + [cell size]

Ry: Ry + [cell size]

R20: boundary check

From Example on pg. 267 (4.2)

DAXPY Loop — VMIPS

VMIPS

```
L.D      F0,a
LV       V1,Rx
MULVS.D  V2,V1,F0
LV       V3,Ry
ADDVV.D  V4,V2,V3
SV       V4,Ry
```

Benefits

- No looping
- Straightforward

From Example on pg. 267 (4.2)

Questions?

Vectorizing Compiler

- Able to extract vector operations from loop
- … in existing code
- Widely used in “number-crunching” organizations

Beyond One Element per Clock Cycle

- **Vector instruction sets allow software to pass a lot of (parallelizable) work to the hardware using one instruction**
- **Allows an implementation to use parallel functional units**
- **Simplified in VMIPS by only letting element *N* of one vector register to take part in operations with element *N* from other vector registers**
  - The set of elements that move through a pipeline together is called an *element group*

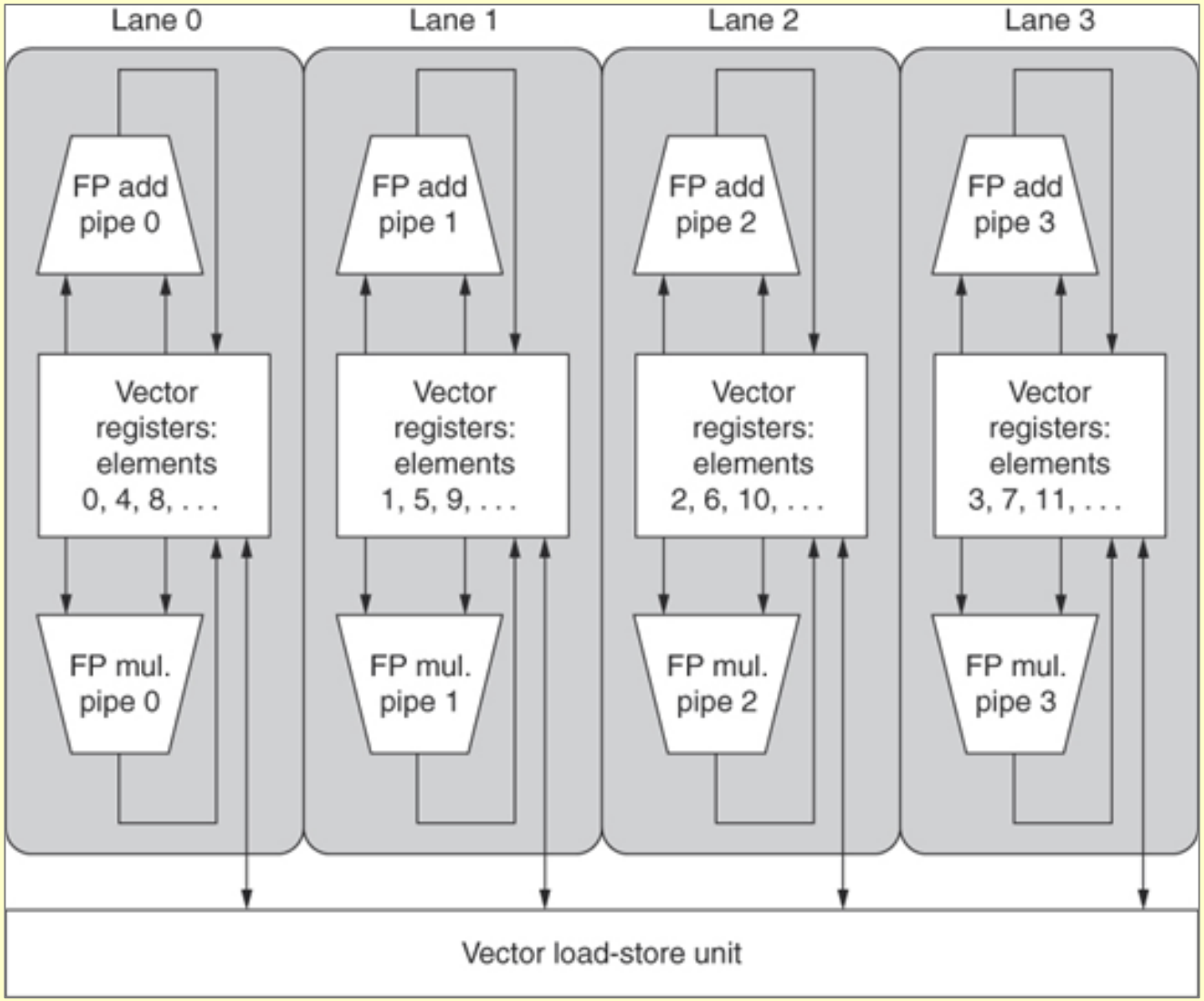
Using Multiple Lanes

- **A highly parallel vector unit can be structured as multiple parallel lanes**
- **Adding more lanes increases the peak throughput of a vector unit**
  - E.g. Going to four lanes from one lane reduces number of cycles for a chime from 64 to 16
  - $\Rightarrow$ halving the clock rate
  - but doubling number of lanes gives same performance
- **To get the most out of lanes, applications and architecture must *both* support long vectors**
  - Otherwise risk running out of instruction bandwidth

Structure of a Lane

- **Each lane contains part of the vector register file and an execution pipeline for each vector unit**
- **Each lane can complete its operation without communicating with the other lanes**
  - This reduces wiring cost and the number of required register file ports

Structure of Vector Unit Containing Four Lanes



(Figure 4.5)

Natural Vector Length

- **Each vector architecture has a natural vector length**
  - Natural vector length for VMIPS is 64
- **Determined by number of elements in each vector register**
- **This usually has nothing to do with the real vector length in a program**

Vector-Length Registers

- **The Vector-Length Register controls the length of any vector operation**
  - Including loads and stores
- ***MVL* — the Maximum Vector Length**
  - Cannot be greater than the length of the vector registers
- **MVL as a parameter**
  - $\Rightarrow$ length of the vector registers can change in later generations and the instruction set can stay the same

Strip Mining

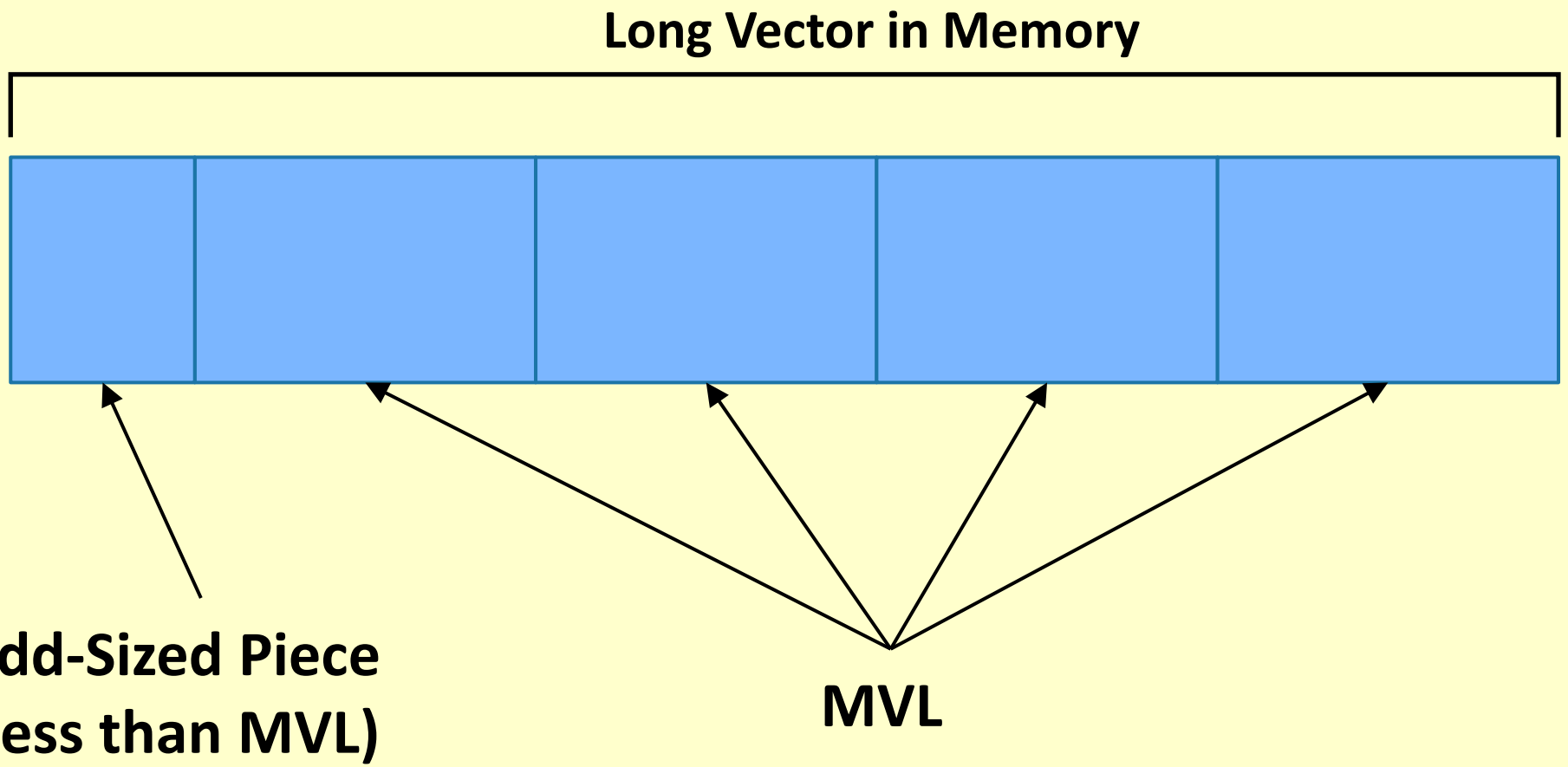
- **Strip Mining: technique to make sure that each vector operation is done for a size ≤ MVL**

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

```
low = 0;
VL = (n % MVL)
for (j = 0; j <= (n/MVL); j = j + 1) {
        for (i = low; i < (low + VL); i = i + 1)
                Y[i] = a * X[i] + Y[i];
        low = low + VL;
        VL = MVL;
}
```

p.274

Strip Mining: A Visual Guide



IF Statements in Vector Loops

- **Conditionals (IF statements) introduce control dependencies into loops**
  - Cannot be run in vector mode using techniques previously discussed

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

Vector-Mask Control

- **Vector-Mask Control uses a Boolean vector to control the execution of a vector instruction**
  - Similar to using a Boolean condition to determine whether to execute a scalar instruction
- **The Boolean vector is called the Vector-Mask Register**
  - Entries in the destination vector that correspond to zeros in the mask register are not affected by the vector operation
  - Clearing the vector mask sets all entries to ones, so later vector instructions operate on all elements

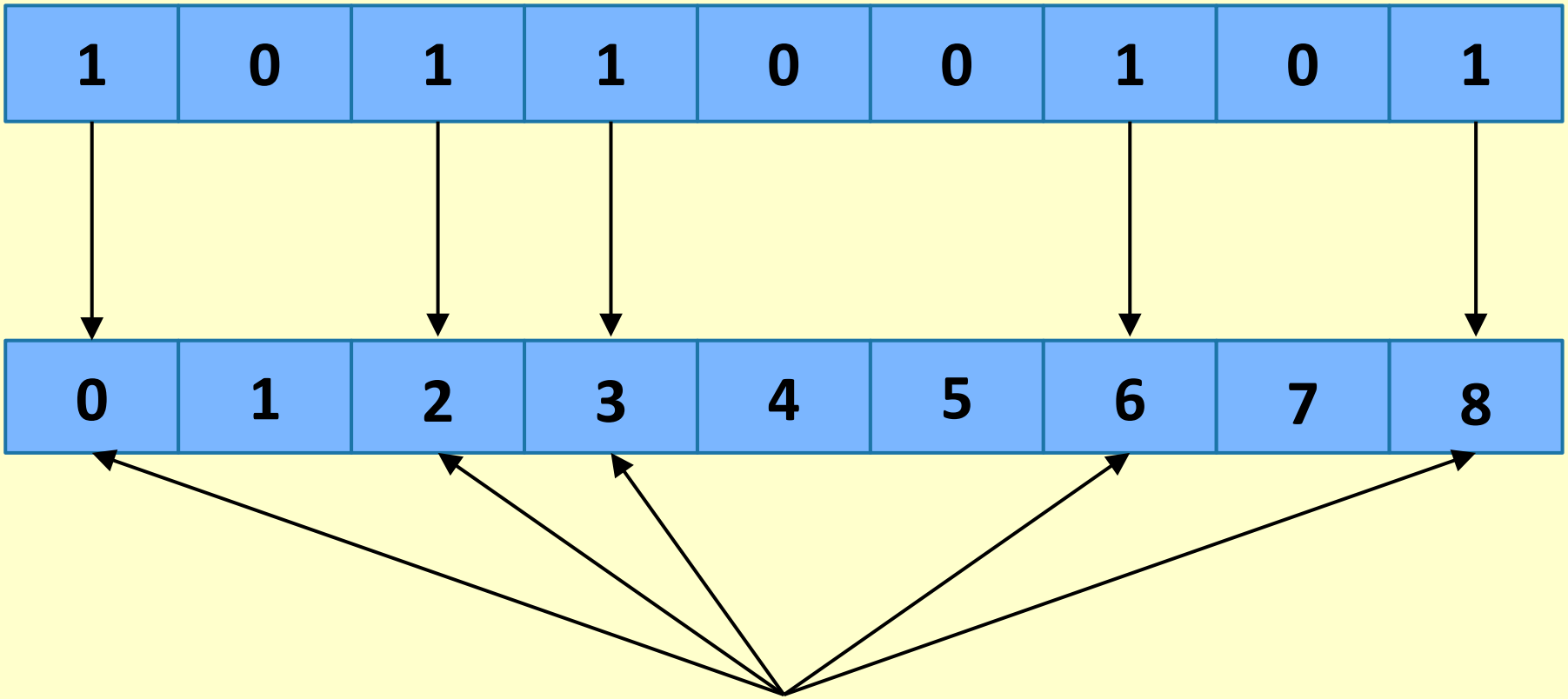
Using the Vector Mask for a Loop

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

```
LV          V1,Rx  ;load vector X into V1
LV          V2,Ry  ;load vector Y into V2
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          V1,Rx  ;store the result in X
```

Vector Masks: A Visual Guide

Vector Mask



Only these entries will be affected

Vector-Mask Performance

- **Vector instructions executed with a vector mask take the same execution time, even for elements where the mask is zero**
  - Similar to scalar architectures
  - Can still be faster than scalar mode, even with a significant number of zeros in the vector-mask
- **Mask registers are part of the architectural state and rely on compilers to manipulate mask registers explicitly**

Memory Banks

- **Vector processors are usually bottlenecked by memory bandwidth**
- **How do we improve memory bandwidth?**
  - Banked memory
  - Improved vector load/store unit
- **Memory is significantly slower than CPUs. We need a lot of banks to compensate.**
  - If we have multiple CPUs, they will likely share a single memory system as well

45

Example: Cray T932

Strides

- **The position in memory of adjacent memory elements in a vector may not be sequential**
  - Single element from each row or column of a 2D array
- **The distance separating elements in a single register is called stride**
  - By default, unit-stride – stride of 1 word
- **Some vector load/store instructions permit specifying a stride other than 1**

Gather-Scatter

- **Some vectors may have indirectly indexed elements**
  - For example: indexing an array with elements of another array
- **Gather and Scatter operations use an index vector loaded with offsets and a base address**
  - We Load (Gather) or Store (Scatter) from the base address plus the offset specified in the index vector

48

Gather-Scatter Example

Sample Code:

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

```
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 A[] and C[]
SVI (Ra+Vk), Va       ; store A[K[]]
```

Programming Vector Architectures

- **The compiler can easily determine at compile time whether a section of code will vectorize**
  - And if they will not, where the dependences are
- **The compiler must be given hints by the programmer in some cases**
  - We can tell it to vectorize operations it otherwise would not

Questions?