Data-Level Parallelism in SIMD and Vector Architectures

Current Trends in Architecture

- Cannot continue to leverage Instruction-Level parallelism (ILP)

- **Beyond ILP:** new models for managing parallelism:
  - Multiprocessors (Multiple Instruction Multiple Data)
  - Thread-level parallelism (TLP)
  - Data-level parallelism (DLP) => *topic of this lecture*
  - Request-level parallelism (RLP)

ILP, DLP

- Instead of going in the direction of complex out-of-order **ILP** processors
- An in-order vector processor can achieve the same performance, or more, by exploiting **DLP (Data Level Parallelism):** same instruction to manage parallel data.
  - With more energy efficiency

Flynn's Taxonomy

- **SISD:** Single instruction stream, single data stream

  - Uniprocessors (including scalar processors like MIPS, but also ILP processors such as superscalars)

- **SIMD:** Single instruction stream, multiple data streams
  - Vector architectures
  - Multimedia extensions
  - Graphics processor units
- **MISD:** Multiple instruction streams, single data stream
  - *No practical usage => no commercial implementation*
- **MIMD:** Multiple instruction streams, multiple data streams
  - Tightly-coupled MIMD (with thread-level parallelism)
  - Loosely-coupled MIMD (with request-level parallelism)

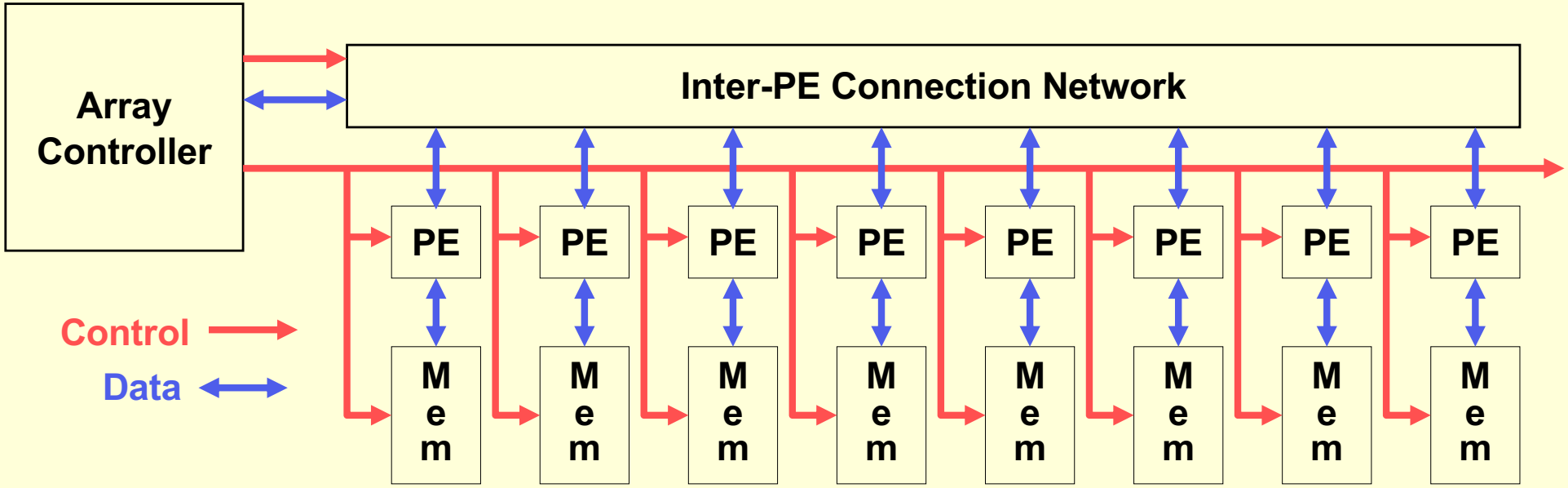
SIMD and Vector Architectures

Introduction to SIMD

- SIMD architectures can exploit significant **data-level parallelism** for:
  - Matrix-oriented scientific computing, machine learning and deep learning;
  - Multimedia such as 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*** and achieve ***parallel speedups*** (compared to MIMD that require parallel programming).

SIMD Architecture

- Radical architecture change w.r.t. multiprocessors.
- Central controller send the same instruction to multiple processing elements (PEs) for multiple data.



✓ Only requires one array controller
✓ Only requires storage for one copy of the sequential program
✓ All parallel computations are fully synchronized

SIMD model

- Synchronized PEs with **single Program Counter**
- Each **Processing Element (PE)** has its own set of data
  - Use different sets of register addresses
- Motivations for **SIMD:**
  - Cost of control unit shared by all execution units
  - Only one copy of the code in execution is necessary
- Practical use:
  - SIMD have a **mix of SISD and SIMD instructions;**
  - A **host processor** executes sequential instructions;
  - **SIMD instructions are sent to Processing Elements:** each PE has its own memory and registers and uses the interconnection network to exchange data.

Three variations of SIMD Machines

1. **Vector architectures**
2. **SIMD extensions:**
   - x86 multimedia SIMD extensions: MMX 1996,
     SSE (Streaming SIMD Extension),
     AVX (Advanced Vector Extension).
3. **Graphics Processor Units (GPUs) => *next lecture***

Vector Architectures

- Basic idea:
  - Load **sets** of data elements into ***vector registers***
  - Operate on ***vector registers***
  - Write the results back into memory

- A single instruction operates on ***vectors of data***
  - Synchronized processing units: ***single Program Counter***
  - Register-to-register operations
  - Used to hide memory latency (memory latency occurs one per vector load/store vs. one per element load/store).
  - Leverage memory bandwidth

Vector Architectures

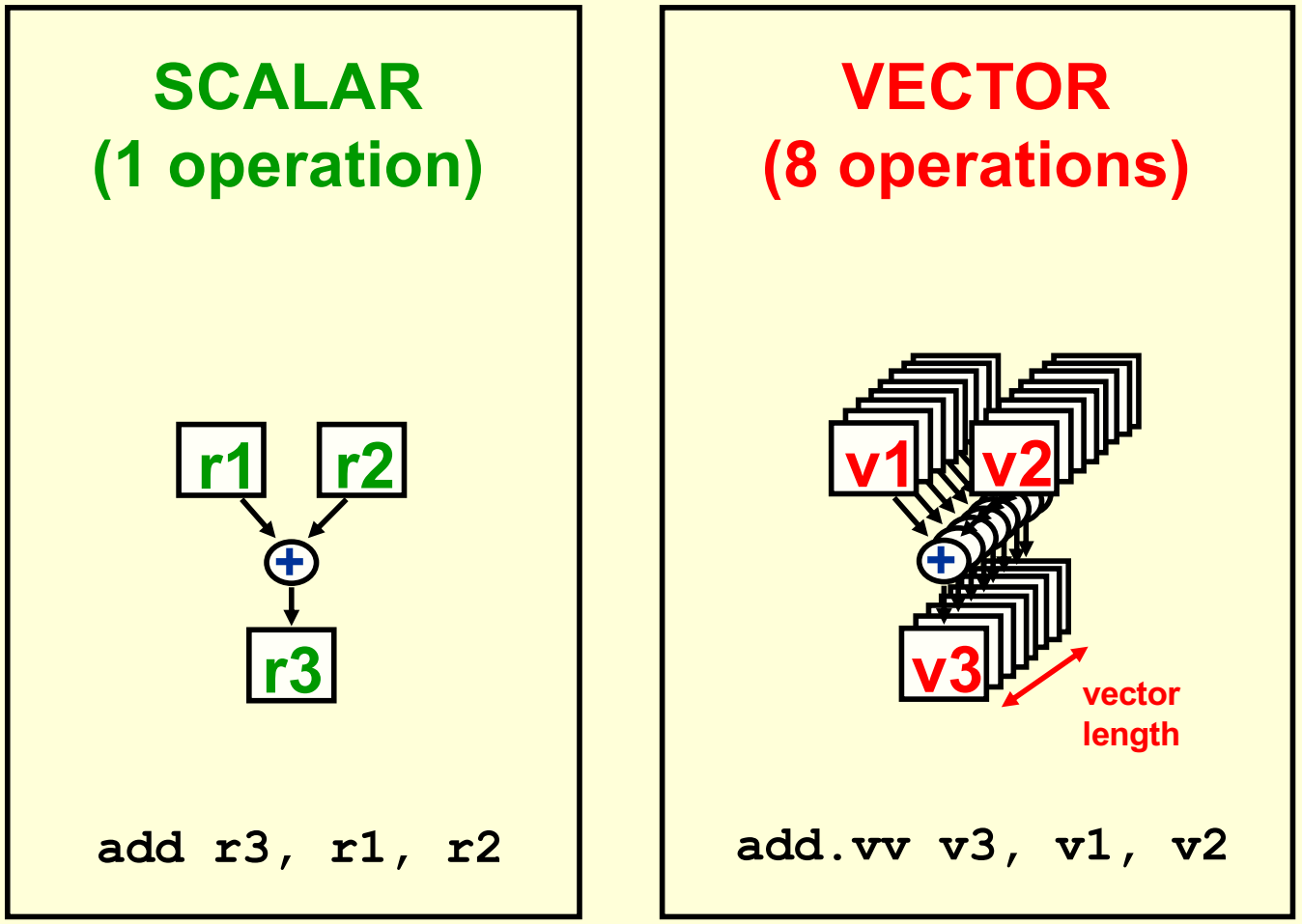
From Cray-1, 1976:

Scalar Unit + Vector Extensions

- Load/Store Architecture
- Vector Registers
- Vector Instructions
- Hardwired Control
- Highly Pipelined Functional Units
- Interleaved Memory System
- No Data Caches
- No Virtual Memory

Vector Processing

- Vector processors have operations that work on linear arrays of data: ***"vectors"***

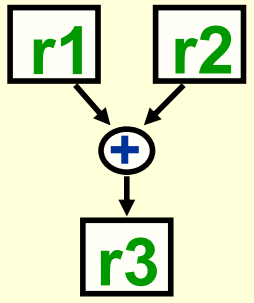


Vector Processing

- Vector processors have operations that work on linear arrays of data: ***"vectors"***

- Each register has n-bits

SCALAR
(1 operation:
1 n-bit ADDER
n-bit registers)



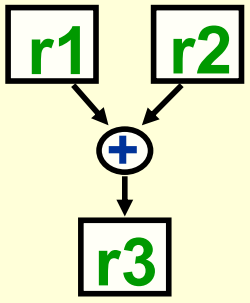

```
add r3, r1, r2
```

Vector Processing

- Vector processors have operations that work on linear arrays of data: ***"vectors"***

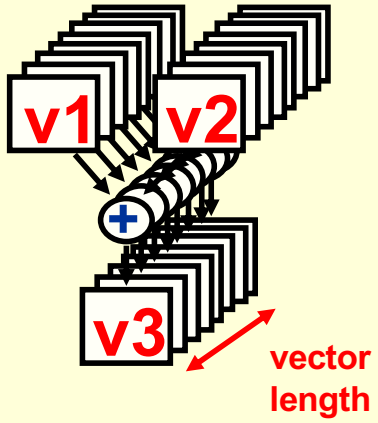
- Each register has n-bits

SCALAR
(1 operation:
1 n-bit ADDER
n-bit registers)




```
add r3, r1, r2
```

VECTOR
(8 operations:
8 n-bit adders
8-elem. n-bit vectors)




```
add.vv v3, v1, v2
```

- Each vector has 8-elements with n-bits/element
- Each element is independent to any other

Properties of Vector Processors

- Each result **independent** of previous result
  => long pipeline, compiler ensures no dependencies
  => high clock rate

- Vector instructions access memory with known pattern
  => highly interleaved memory
  => amortize memory latency of over 64 elements
  => no (data) caches required! (Do use instruction cache)

- Reduces the number of branches and branch problems in pipelines

- Single vector instruction implies lots of work (loop)
  => fewer instruction fetches

Styles of Vector Architectures

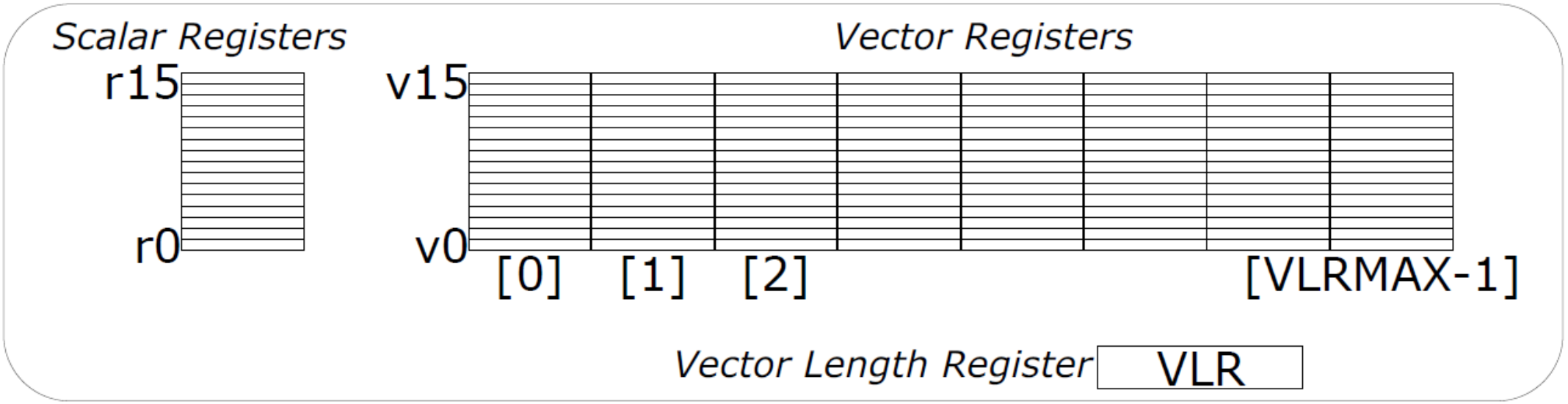
- ***Memory-memory vector processors:*** all vector operations are memory to memory

- ***Vector-register processors:*** all vector operations are done between vector registers (except load and store)
  - Vector-equivalent of load-store scalar architectures
  - Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC

Components of Vector Processors

- ***Vector Register File*:** fixed length register bank holding multiple-element vectors
  - Has at least 2 read and 1 write ports
  - Typically 8-32 vector registers, each holding 64-128 bit elements

Scalar Registers vs Vector Registers



16 Scalar Registers:

each register holds
a 32-bit element

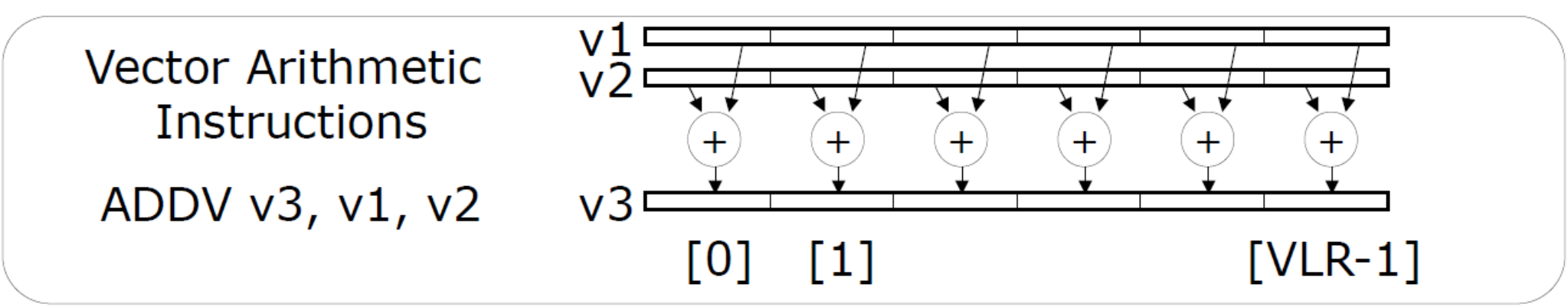
16 Vector Registers:

each vector register holds
VLRMAX elements, 32-bit per
element

Components of Vector Processors

- ***Vector Register File*:**
  - has at least 2 read and 1 write ports
  - typically 8-32 vector registers, each holding 64-128 bit elements
- ***Vector Functional Units (FUs)*: fully pipelined, start new operation every clock**
  - Typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- ***Vector Load-Store Units** (LSUs)*: fully pipelined unit to load or store a vector; may have multiple LSUs
- ***Scalar Registers:*** single element for FP scalar or address
- ***Cross-bar*** to connect FUs , LSUs, registers

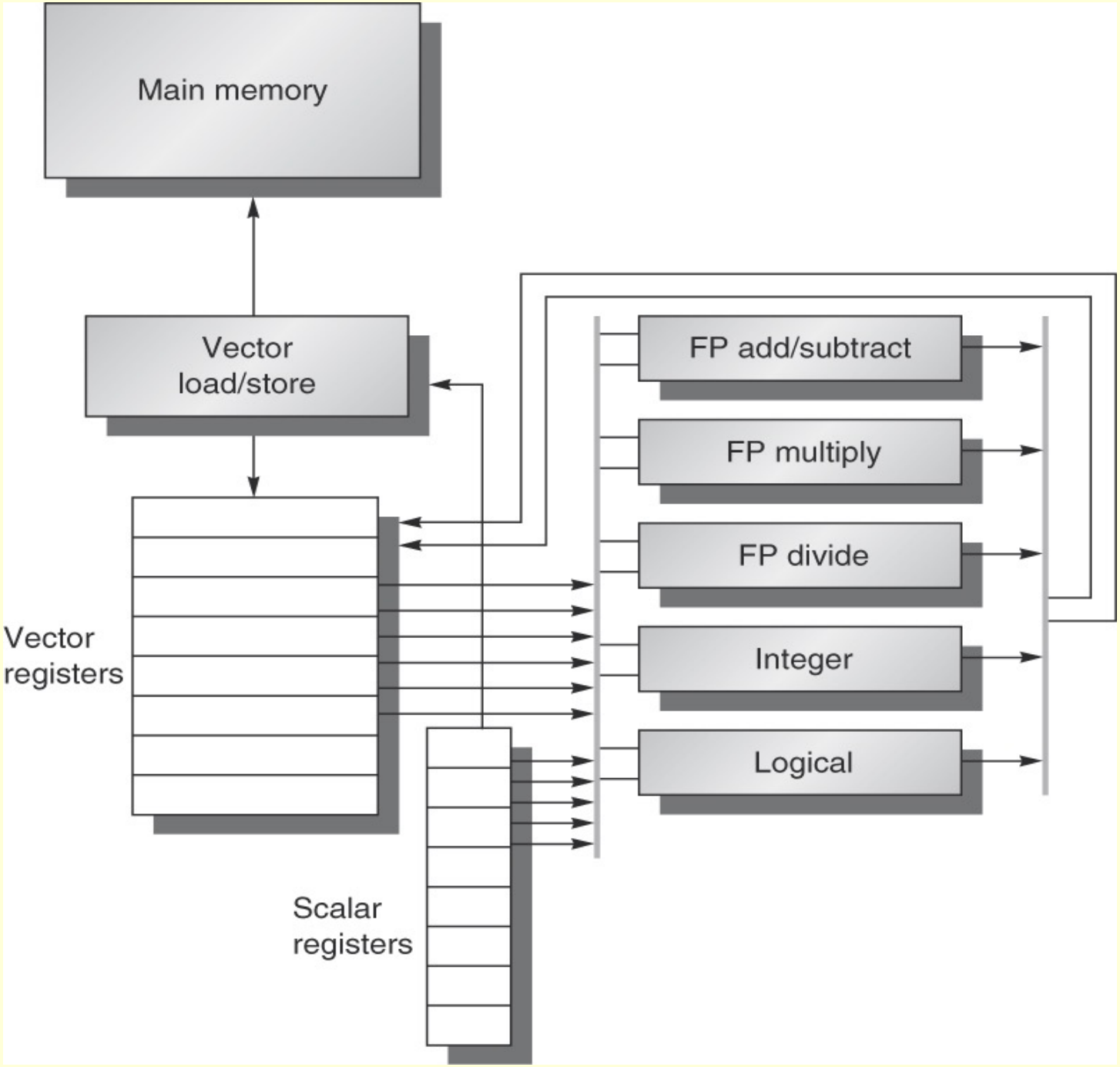
Vector Arithmetic Instructions



- There are 6-element vectors and 6 adders
  => Each vector instruction takes one time unit.
- If there were 6-element vectors and only 3 adders
  => Each vector instruction takes 2 time units.

Example architecture: VMIPS

- Loosely based on Cray-1
- ***Vector Registers File:***
  - **8 vector registers:** Each register holds a vector of **64-elements with 64 bits/element;**
  - Thera are (at least) 16 read ports and 8 write ports (to enable up to 8 simultaneous accesses)
- ***Vector functional units***
  - Fully pipelined so they can start a new operation every cycle
- ***Vector load-store unit***
  - Fully pipelined, one word per clock cycle after initial memory latency
- ***Scalar registers***
  - 32 general-purpose registers
  - 32 floating-point registers



Basic structure of the VMIPS vector architecture

•This processor has a **scalar** architecture just like MIPS.

• There are 8 64-element vector registers,
and 5 vector functional units.

•The vector and scalar registers have a significant number of read and write ports to enable simultaneous vector operations

•The control unit and the scalar integer functional units are not shown.

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 The VMIPS vector instructions, showing only the double-precision floating-point operations. In

ISBN: 978012383873

Accommodating varying data sizes

- Vector processors are good for several applications (scientific applications based on linear algebra, but also multimedia applications)
  - Because they can adapt to several width: the application vector size can be mapped to 64 64-bit elements, or 128 32-bit elements etc.

DAXPY operation

- DAXPY operation, in scalar vs 64-element vector VMIPS versions
- DAXPY stands for: double precision a X plus Y

$$Y = a * X + Y$$

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

Scalar version of DAXPY

Let's assume that Rx and Ry are holding the addresses of vectors X and 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, 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
```

Scalar version of DAXPY

(assume that Rx and Ry are holding the addresses of X and 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
        L.D     F4, 0(Ry)
        ADD.D   F4, F4, F2
        S.D     F4, 0(Ry)
        DADDIU  Rx, Rx, #8
        DADDIU  Ry, Ry, #8      ; increment index to Y
        DSUBU   R20, R4, Rx     ; compute bound
        BNEZ    R20, Loop       ; check if done
```

Scalar version of DAXPY:
9 instructions per iteration
=> (64 x 9) + 2 = 578
instructions per loop
plus stalls

Analysis of scalar version (1)

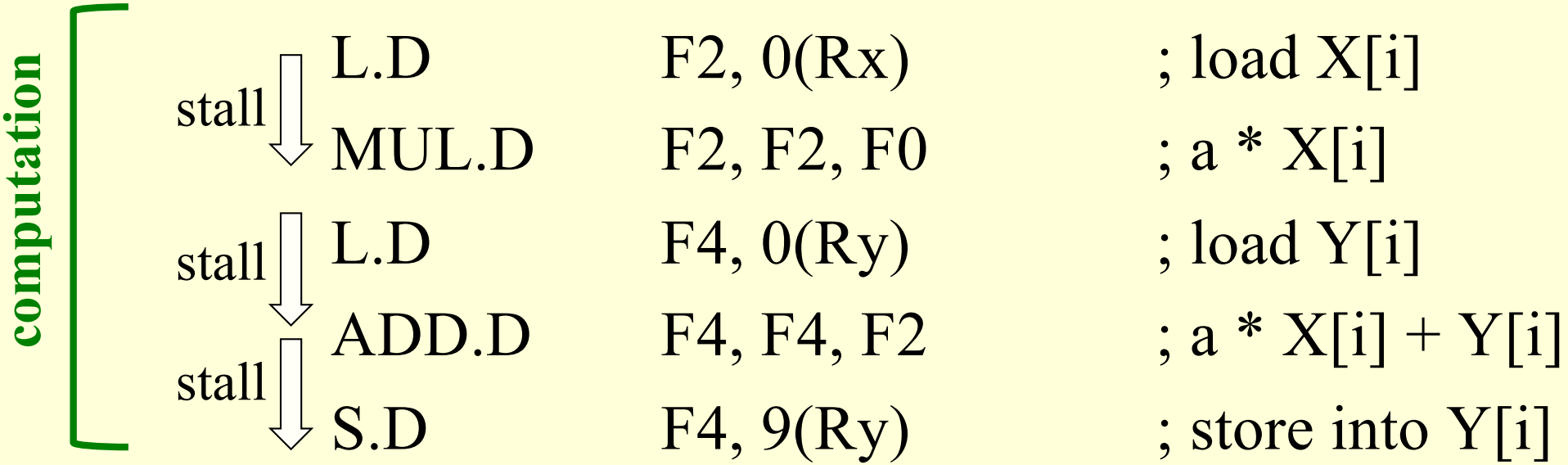
computation

```
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, 0(Ry)      ; store into Y[i]
```

overhead

```
      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
```

Analysis of scalar version (2)



Scalar version need some stalls at EVERY iteration

VMIPS Instructions

- **DAXPY: Y = a * X + Y**
- **ADDVV.D:** add two vectors
- **MULVS.D:** multiply vector to a scalar
- **LV/SV:** vector load and vector store from memory address
- **Vector processor version of DAXPY:**

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

VMIPS Instructions

- **Vector processor version of DAXPY: Y = a * X + Y**

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

- ***VMIPS requires 1 scalar and 5 vector instructions per loop vs. almost 600 scalar instructions for MIPS: greatly decreased!***
- ***But how many clock cycles?***
- ***It takes approximately 5 x 64 elements = 320 clock cycles (*)***

(*) Not considering vector start-up time and at least one clock to manage the load scalar.

Analysis of advantages vs. scalar version

- **Very compact code:** it requires only 6 vector instructions per loop!
- **No branches anymore (***no more loop overhead)*.
- The scalar version can try to get a similar effect by loop unrolling, but without the same instruction count decrease. Moreover, loop unrolling would require register renaming.
- **Pipeline stalls** greatly decreased in the vector version:
  - It must stall **ONLY** for **THE FIRST** vector element; after that, results can come out every clock cycle!

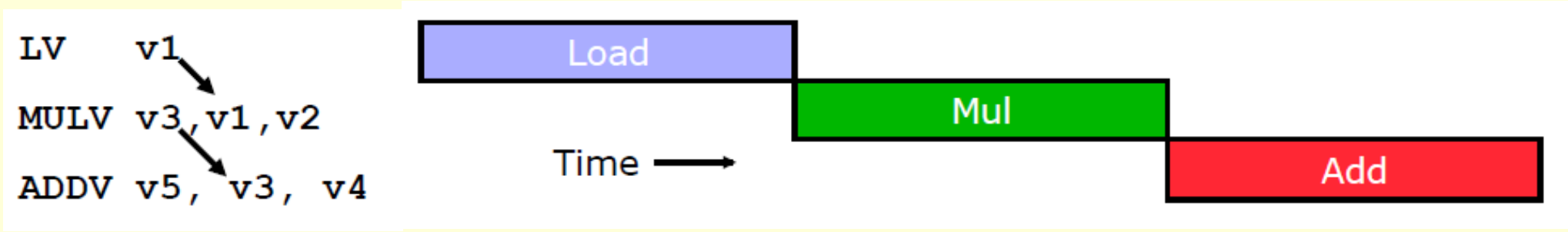
- ***How can we get a further speedup?***

Operation Chaining

- **Results from each FU forwarded to next FU in the chain**
- Concept of *forwarding* extended to vector registers:
  - A vector operation can start as soon as each element of its vector source operand become available
  - Even though a pair of operations depend on one another, *chaining allows the operations to proceed in parallel on separate elements of the vector.*
  - In this way, we don't need anymore to wait for the last element of a load to start the next dependent instruction.

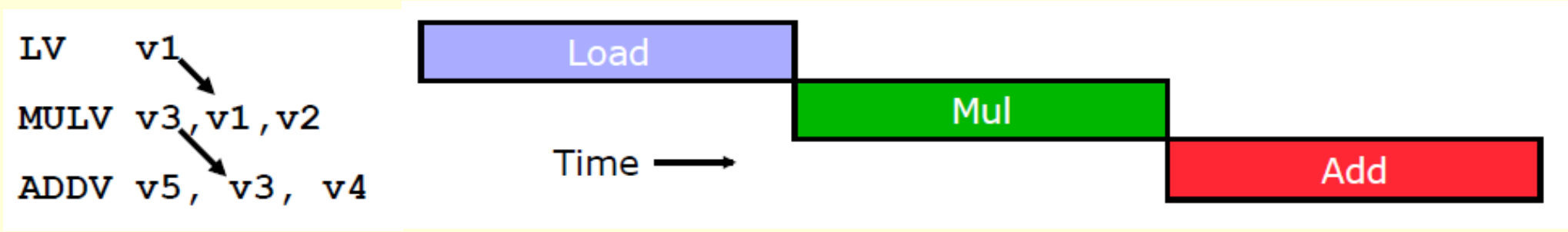
Operation Chaining (2)

- **Without chaining:** must wait for last element of one instruction to be written before starting the next dependent instruction

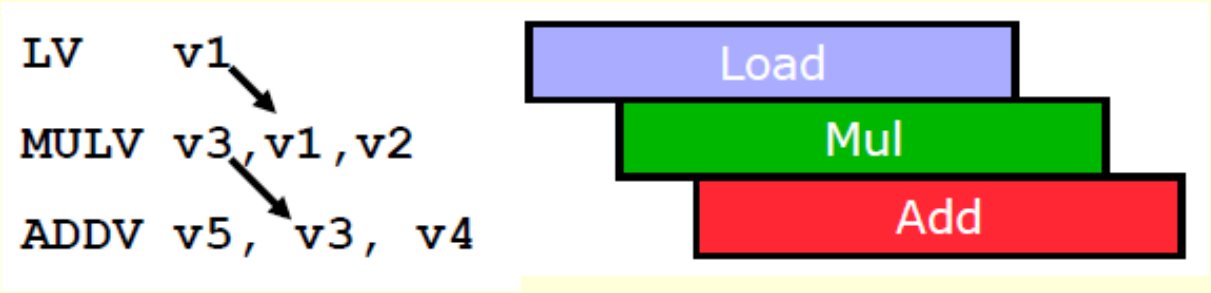


Operation Chaining (2)

- **Without chaining:** must wait for last element of one instruction to be written before starting the next dependent instruction



- **With chaining:** a dependent operation can start as soon as each element of its vector source operand become available (*):



(*) We assumed to have 1 LV/SV Unit, 1 MULV Unit and 1 ADDV Unit

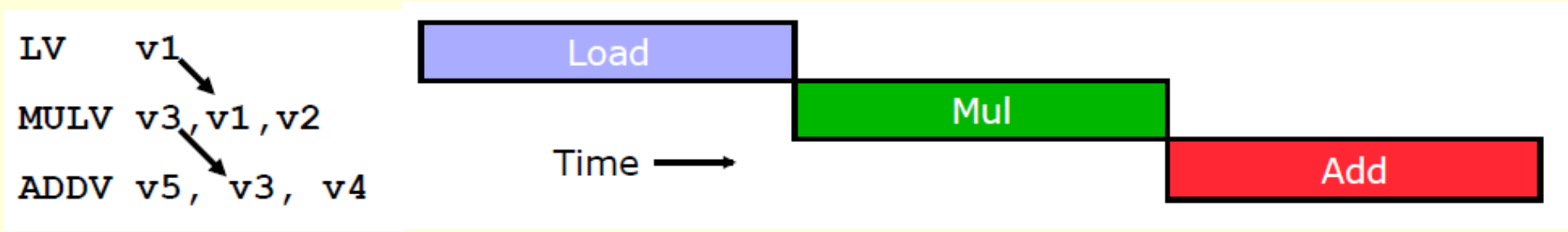
http://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-T14-vector-processors.pdf

Vector Execution Time

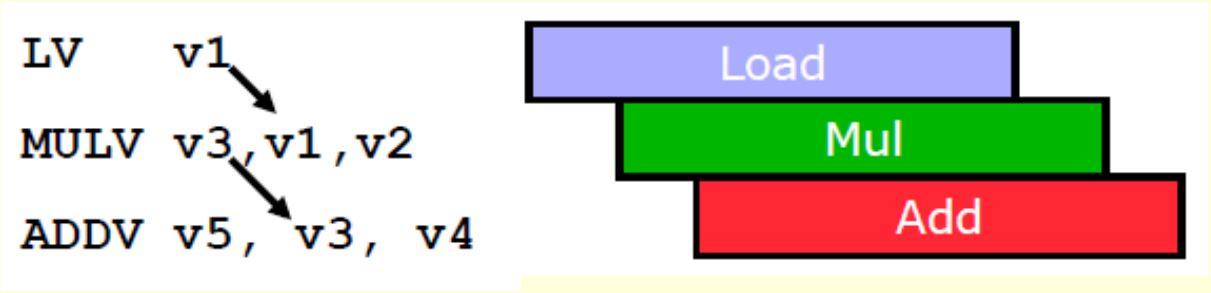
- Execution time depends on three factors:
  - Length of operand vectors *(number of elements)*
  - Structural hazards *(how many vector functional units)*
  - Data dependencies *(need to introduce operation chaining)*
- VMIPS functional units consume ***one element per clock cycle***
  - So, the execution time of one vector instruction is approximately given by the vector length

Operation Chaining (2)

- **Without chaining: (3 x 64) = 192 clock cycles**



- **With chaining: (64 +2) = 66 clock cycles**



(*) We assumed to have 1 LV/SV Unit, 1 MULV Unit and 1 ADDV Unit

http://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-T14-vector-processors.pdf

Convoys

- Simplification: to introduce the notion o*f* ***convoy***
  - Set of vector instructions that could potentially execute together partially overlapped (no structural hazards)
- Sequences with read-after-write dependency hazards can be in the same convoy via ***chaining***

Chimes

- ***Chime*** is a timing metric corresponding to the unit of time to execute one convoy
  - ***m convoys*** execute in ***m chimes (or time units)***
  - Simply stated: for a vector length of ***n,*** and ***m*** convoys in a program, ***n x m clock cycles are required***
  - Chime approximation ignores some processor-specific overheads

DAXPY 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
```

Let assume to have 1 LV/SV Unit & 1 ALU V Unit with chaining:

DAXPY 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
```

Let assume to have 1 LV/SV Unit & 1 ALU V Unit with chaining:

3 convoys:

1) LV MULVS.D (chaining)

2) LV ADDVV.D (chaining)

3) SV

3 convoys (3 chimes): 2 FP ops per element => 1.5 cycles per FP ops per element;

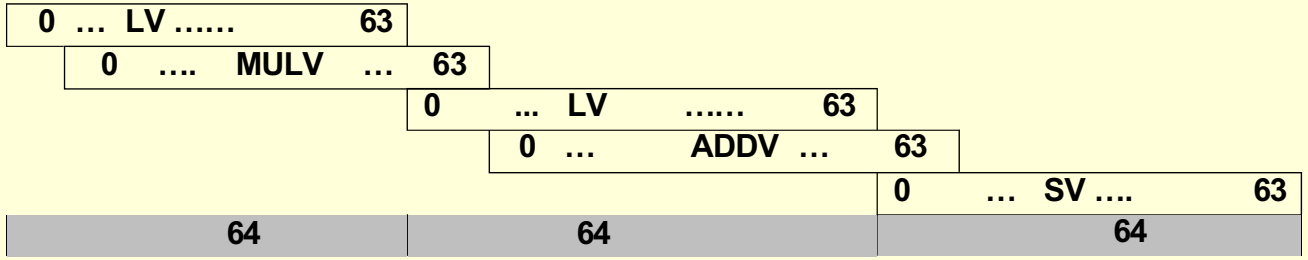
DAXPY Example

3 convoys:

1) LV MULVS.D (chaining)

2) LV ADDVV.D (chaining)

3) SV

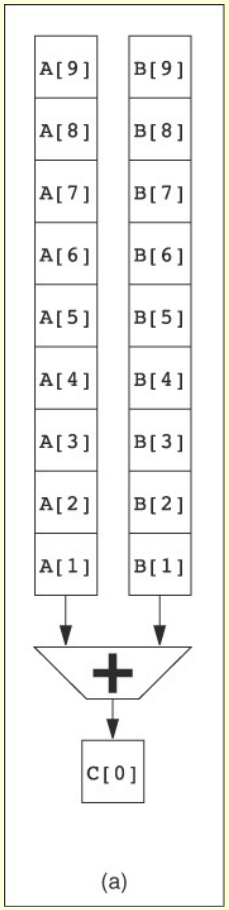


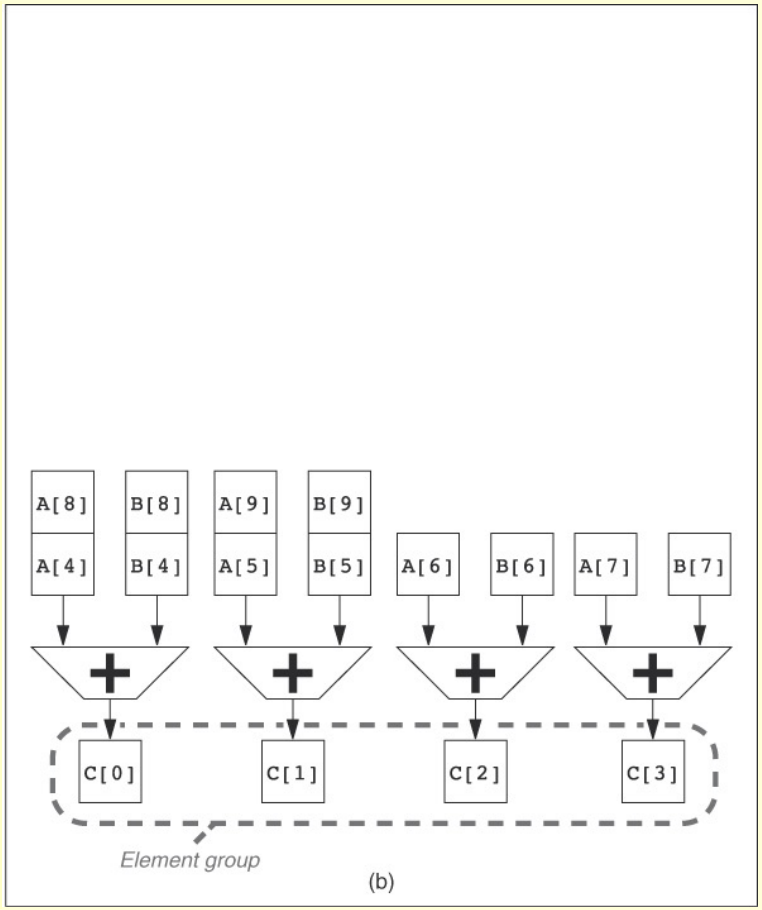
- **3 convoys (3 chimes)**
- For 64 element vectors => **3 x 64 = 192 clock cycles** instead of:
  - about 320 clock cycles without chaining;
  - about 600 cycles (plus stalls) for scalar MIPS;

- ***How can a vector processor execute a vector faster than one element per clock cycle?***

Introducing Multiple Lanes

- Instead of generating an element per cycle in one lane, spread the elements of the two vector operands into multiple lanes to improve vector performance





SINGLE ADD PIPELINE:

1 add per cycle

64 cycles for a vector of 64 elements

FOUR ADDs PIPELINE:

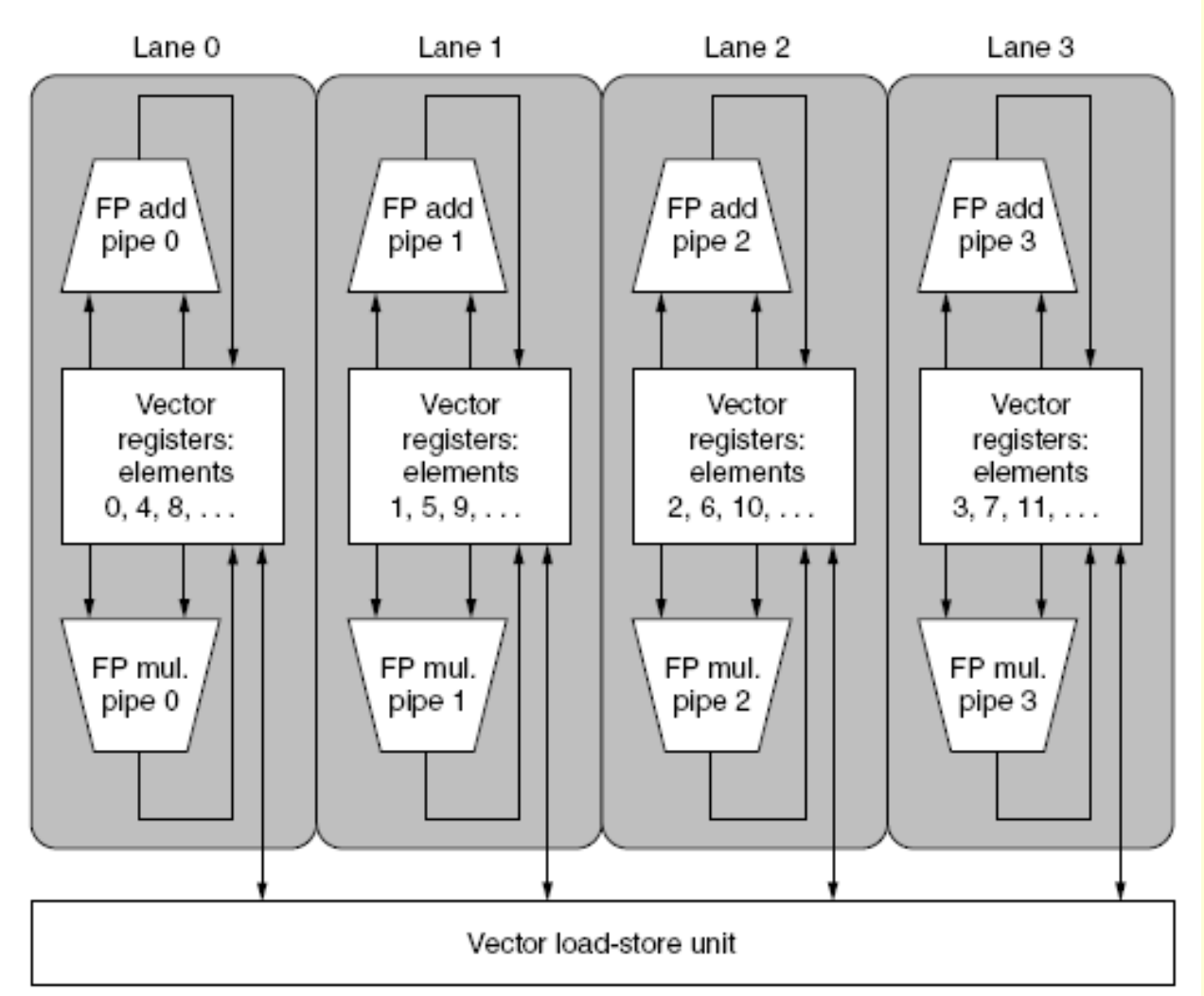
4 adds per cycle

16 cycles for a vector of 64 elements

Multiple Lanes

Vector Unit with 4 lanes

Vector registers are divided across 4 lanes



- ***How to handle a number of loop iterations not equal to the vector length?***

Vector Length Control

- The ***Maximum Vector Length (MVL)*** is the physical length of vector registers in a machine (64 in VMIPS)

- *What do you do when the vector length in a program is not exactly 64?*
  1. Vector length **smaller** than 64
  2. Vector length **unknown** at compile time and maybe **greater** than MVL

1) Vector length smaller than 64

```
for (i=0; i<63; i=i+1)
    Y[i]=a * X[i] + Y[i];
```

```
for (i=0; i<31; i=i+1)
    Y[i]=a * X[i] + Y[i];
```

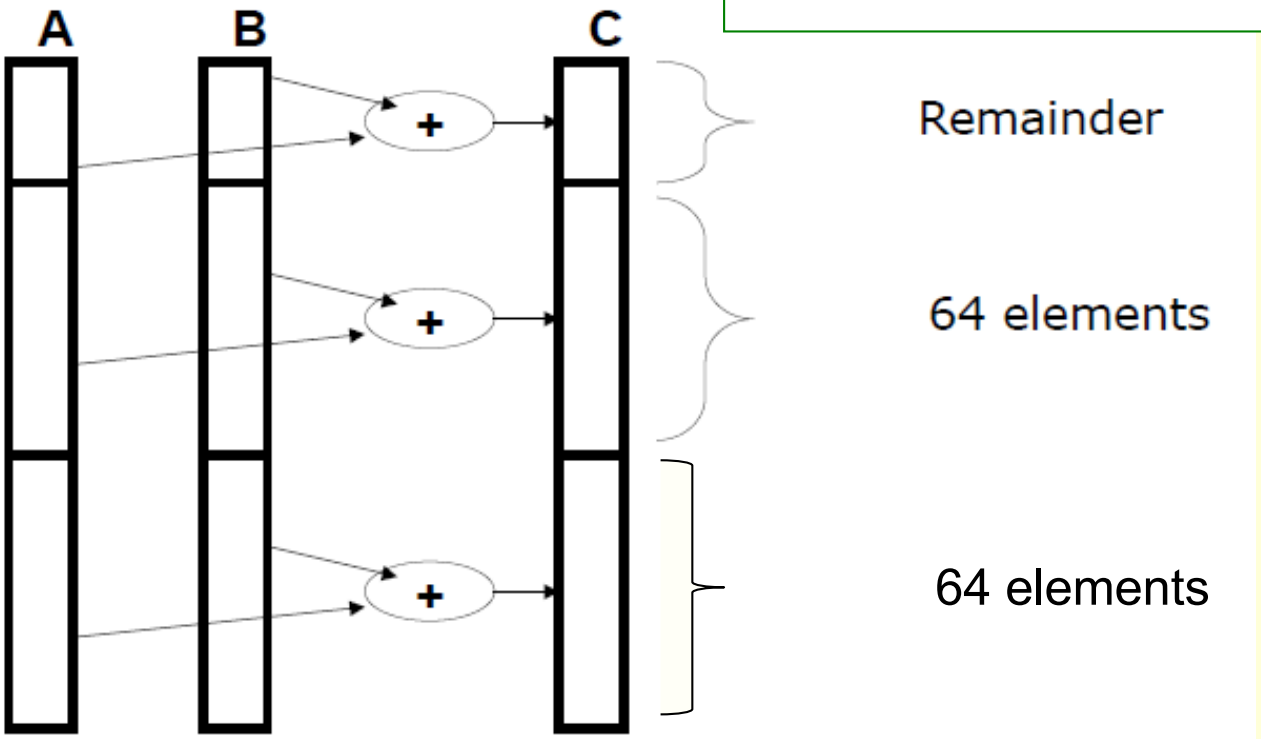
- There is a special register, called vector-length register (**VLR**)
- The VLR controls the length of any vector operation (including vector load/store).
- It can be set to any value ***smaller*** than the MVL (64)

2) Vector length unknown at compile time

Restructure the code using a technique called ***strip mining:***

- Sort of loop unrolling where the length of first segment is the **remainder (N mod MVL)** and all subsequent segments are of length **MVL**

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

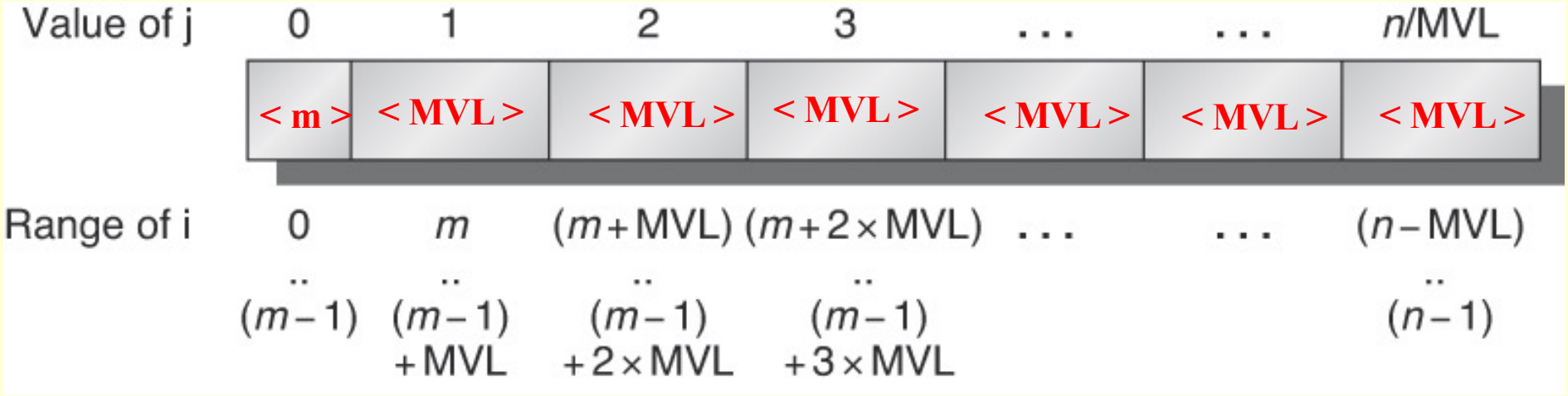


Vector length unknown at compile time

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

Restructure the code using a technique called ***strip mining:***

- Code generation technique such that each vector operation is done for a size less than or equal to MVL
- Sort of loop unrolling where the length of the first segment is (n mod MVL) and all subsequent segments are of length MVL



- ***What to do when there is an IF statement inside the FOR loop code to be vectorized?***

Vector Mask Registers

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

Control Dependence in a loop

This loop cannot normally be vectorized because of the **if** clause inside it

- Use a ***vector mask register*** to "disable" some elements:
- The vector mask uses a Boolean vector of length MVL to control the execution of a vector instruction.
- When vector mask registers are enabled, any vector instruction operates ONLY on the vector elements whose corresponding masks bits are set to 1.

Vector Mask Registers

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

This loop cannot normally be vectorized because of the **if** clause inside it

Use vector mask register to "disable" elements:

```
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        Rx,V1      ;store the result in X
```

The cycles for non-executed operation elements are lost

But the loop can still be vectorized!

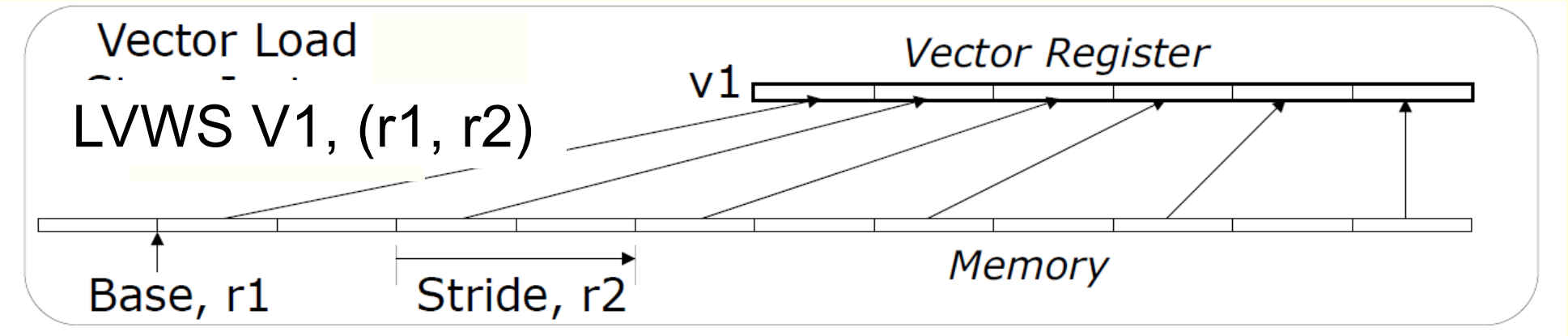
Memory Banks

- 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
  - Startup time: time to get first word from memory to registers
- 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?

- ***How to handle bi-dimensional matrices?***

Stride

- When a ***matrix*** is allocated in memory, it is linearized and laid out in row-major order in C => the elements in the columns are not-adjacent in memory
- *How to handle non-adjacent memory elements?*
- Managing the ***stride:*** the distance separating memory elements that are to be gathered into a single register.



Stride

- When the elements of a matrix in the inner loop are accessed by column => they are separated in memory by a ***stride*** equal to the row size times 8 bytes per entry
- We need an instruction ***LVWS*** to load elements of a vector that are non-adjacent in memory from address R1 with stride R2:

```
LVWS V1, (R1, R2)        ; V1 <= M[R1 + i*R2]
```

- Example:
```
LVWS V1, (C, 100)        ; V1 <= M[C + i*100]
while   LV V2, B         ; V2 <= M[B]
```

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 for columns of D*
- Bank conflict (stall) occurs when the same bank is hit faster than bank busy time:
  - #banks / LCM(stride,#banks) < bank busy time

- ***How to handle sparse matrices in vector processors?***

Gather-Scatter

- Primary mechanism to support ***sparse matrices*** by using index vectors. Consider:

  for (i = 0; i < n; i=i+1)

  A[K[i]] = A[K[i]] + C[M[i]];

- Use index vector K and M to indicate the nonzero elements of A and C (A and C must have the same number of nonzero elements).

```
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[]]
```

SIMD Instruction Set Extensions

- Multimedia applications operate on data types narrower than the native word size
  - Example: disconnect carry chains to "partition" adder

- Limitations, compared to vector instructions:
  - Number of data operands encoded into op code
  - 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
  - Operands must be consecutive and aligned memory locations