Introduction to Vector Processing

- **Motivation: Why vector Processing?** **Paper: VEC-1**
  - Limits to Conventional Exploitation of ILP
  - Flynn's 1972 Classification of Computer Architecture
  - Data Parallelism and Architectures
- **Vector Processing Fundamentals** **Paper: VEC-1**
  - Vectorizable Applications
  - Loop Level Parallelism (LLP) Review (From 551)
  - Vector vs. Single-Issue and Superscalar Processors
  - Properties of Vector Processors/ISAs
  - Vector MIPS (VMIPS) ISA
  - Vector Memory Operations Basic Addressing Modes
  - Vectorizing Example: DAXPY
  - Vector Execution Time Evaluation
  - Vector Load/Store Units (LSUs) and Multi-Banked Memory
  - Vector Loops (n > MVL): Strip Mining
  - More on Vector Memory Addressing Modes: Vector Stride Memory Access
  - Vector Operations Chaining **Vector element data forwarding**
  - Vector Conditional Execution & Gather-Scatter Operations
  - Vector Example with Dependency: Vectorizing Matrix Multiplication
  - Common Vector Performance Metrics & Examples
  - Limited Vector Processing: SIMD/vector or Multimedia Extensions to Scalar ISA
  - Summary: Vector Processing Advantages & Pitfalls
- **Vector Processing & VLSI: Vector Intelligent RAM (VIRAM)** **Papers: VEC-2, VEC-3**
  - Potential Advantages
  - VIRAM Architecture
  - Overview of VIRAM Prototype Generations: VIRAM-1, VIRAM-2

Papers: VEC-1, VEC-2, VEC-3

Motivation for Vector Processing:

Problems with Superscalar approach

- **Limits to conventional exploitation of ILP:**

1) ***Pipelined clock rate***: Increasing clock rate requires deeper pipelines with longer pipeline latency which increases the CPI increase (longer branch penalty , other hazards).

2) ***Instruction Issue Rate***: Limited instruction level parallelism (ILP) reduces actual instruction issue/completion rate. (vertical & horizontal waste) SMT fixes this one?

3) ***Cache hit rate***: Data-intensive scientific programs have very large data working sets accessed with poor locality; others have continuous data streams (multimedia) and hence poor locality. (poor memory latency hiding).

4) **Data Parallelism:** Poor exploitation of data parallelism present in many scientific and multimedia applications, where similar independent computations are performed on large arrays of data (Limited ISA, hardware support).

- As a result, actual achieved performance is much less than peak potential performance and low computational energy efficiency (computations/watt)

X86 CPU Cache/Memory Performance Example:
AMD Athlon T-Bird Vs. Intel PIII, Vs. P4

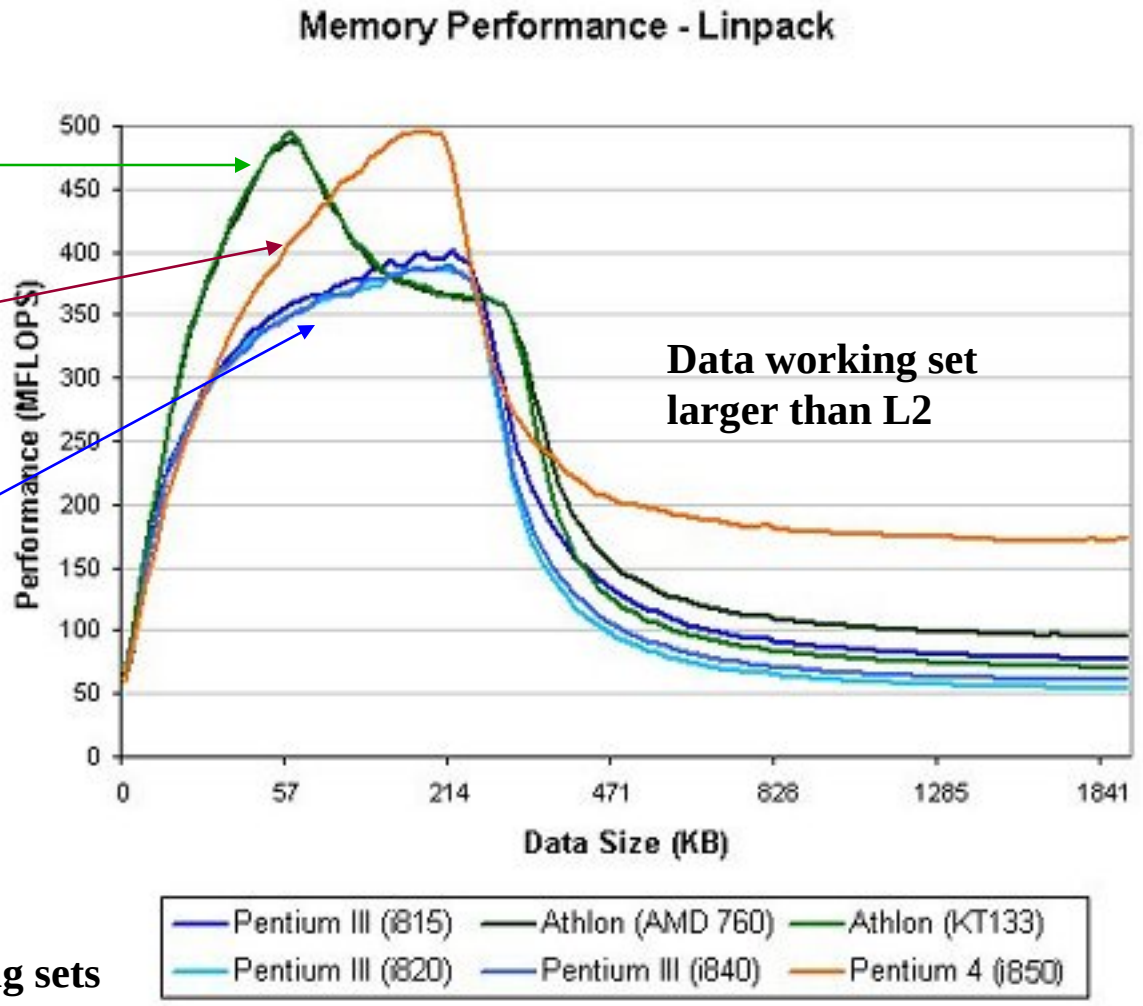
AMD Athlon T-Bird 1GHZ
L1: 64K INST, 64K DATA (3 cycle latency), both 2-way
L2: 256K 16-way 64 bit Latency: 7 cycles
L1,L2 on-chip

Intel P 4, 1.5 GHZ
L1: 8K INST, 8K DATA (2 cycle latency) both 4-way
96KB Execution Trace Cache
L2: 256K 8-way 256 bit , Latency: 7 cycles

L1,L2 on-chip

Intel PIII 1 GHZ
L1: 16K INST, 16K DATA (3 cycle latency) both 4-way
L2: 256K 8-way 256 bit , Latency: 7 cycles

L1,L2 on-chip



Impact of long memory latency for large data working sets

Source: http://www1.anandtech.com/showdoc.html?i=1360&p=15

From 551

Flynn's 1972 Classification of Computer Architecture

i.e single-threaded Uniprocessor

SISD
- **Single Instruction stream over a Single Data stream (SISD): Conventional sequential machines (Superscalar, VLIW).**

SIMD
- **Single Instruction stream over Multiple Data streams (SIMD): Vector computers, array of synchronized processing elements. (exploit data parallelism)**

AKA Data Parallel or Data Streaming Architectures

MISD
- **Multiple Instruction streams and a Single Data stream (MISD): Systolic arrays for pipelined execution.**

MIMD
- **Multiple Instruction streams over Multiple Data streams (MIMD): Parallel computers:**

Parallel Processor Systems: Exploit Thread Level Parallelism (TLP)

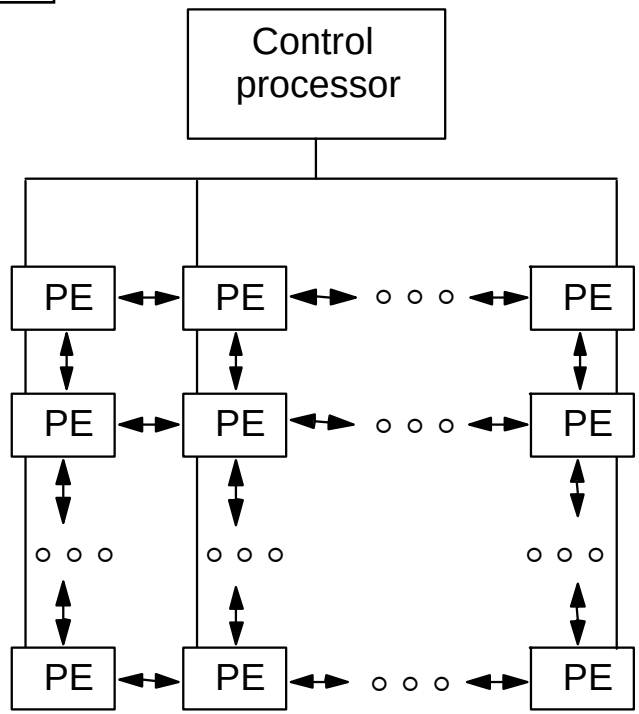
  - **Shared memory multiprocessors (e.g. SMP, CMP, NUMA, SMT)**
  - **Multicomputers: Unshared distributed memory, message-passing used instead (Clusters)**

Instruction stream = Hardware context or thread

From 756 Lecture 1

Data Parallel Systems SIMD in Flynn taxonomy

- **Programming model: Data Parallel**
  - **Operations performed in parallel on each element of data structure** i.e elements of arrays or vectors
  - **Logically single thread of control, performs sequential or parallel steps**
  - **Conceptually, a processing element (PE) or processor is associated with each data element**
- **Architectural model**
  - **Array of many simple, cheap processors each with little memory**
    - **Processors don't sequence through instructions** Control processor does that
  - **Attached to a control processor that issues instructions**
  - **Specialized and general communication, cheap global synchronization**
- **Example machines:**
  - **Thinking Machines CM-1, CM-2 (and CM-5)**
  - **Maspar MP-1 and MP-2,**
  - **Current Variations: IBM's Cell Architecture, Graphics Processor Units (GPUs)**
    - **Difference: PE's are full processors optimized for data parallel computations.**



PE = Processing Element

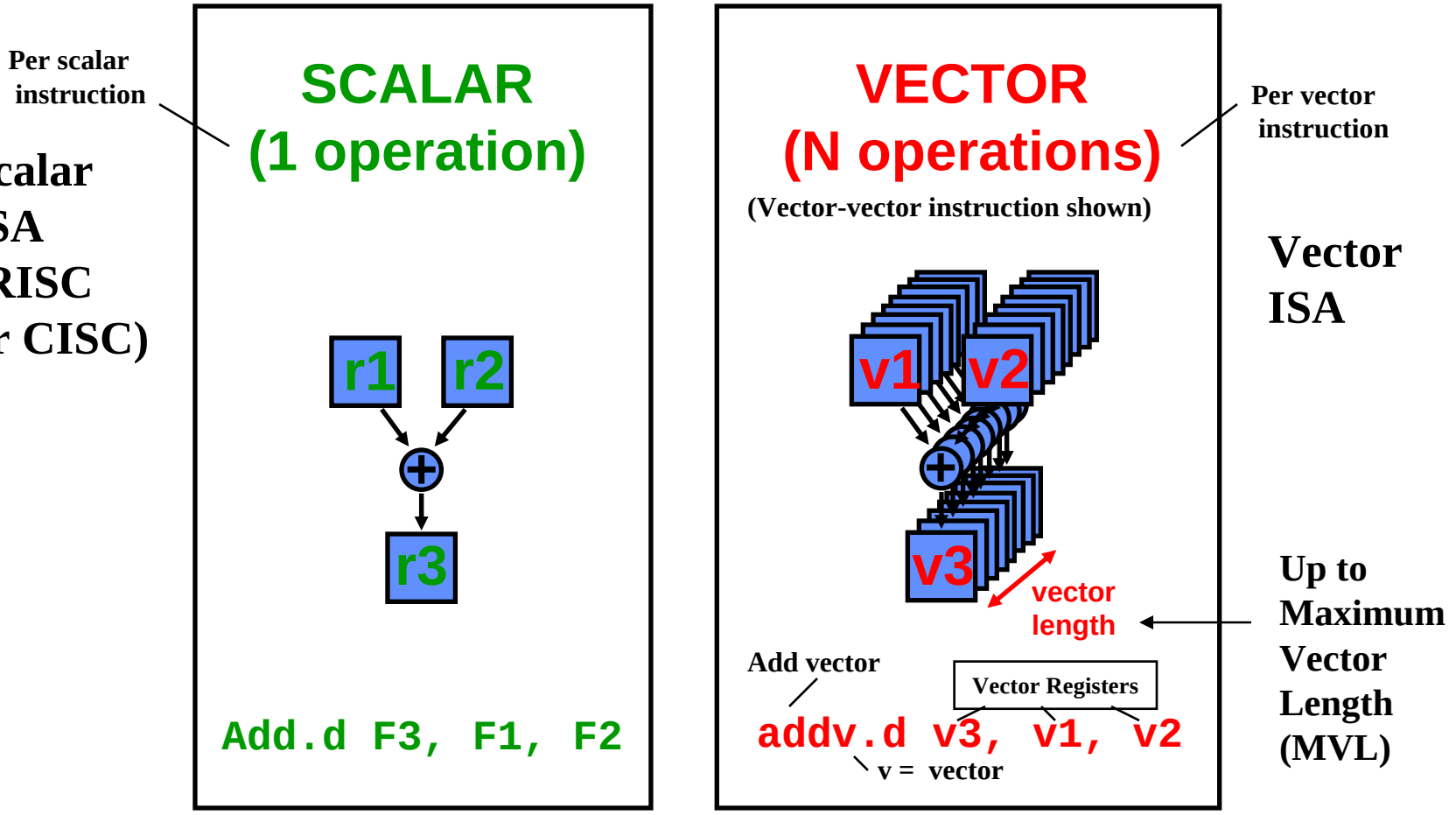
SIMD (array) Machine

From 756 Lecture 1

Alternative Model:Vector Processing

- Vector processing exploits data parallelism by performing the same computation on linear arrays of numbers "vectors" using one instruction.
- The maximum number of elements in a vector supported by a vector ISA is referred to as the Maximum Vector Length (MVL).

Typical MVL= 64-128 Range 64-4996



Vector (Vectorizable) Applications

Applications with high degree of data parallelism (loop-level parallelism), thus suitable for vector processing. Not Limited to scientific computing

- **Astrophysics**
- **Atmospheric and Ocean Modeling**
- **Bioinformatics**
- **Biomolecular simulation: Protein folding**
- **Computational Chemistry**
- **Computational Fluid Dynamics**
- **Computational Physics**
- **Computer vision and image understanding**
- **Data Mining and Data-intensive Computing**
- **Engineering analysis (CAD/CAM)**
- **Global climate modeling and forecasting**
- **Material Sciences**
- **Military applications**
- **Quantum chemistry**
- **VLSI design**
- **Multimedia Processing (compress., graphics, audio synth, image proc.)**
- **Standard benchmark kernels (Matrix Multiply, FFT, Convolution, Sort)**
- **Lossy Compression (JPEG, MPEG video and audio)**
- **Lossless Compression (Zero removal, RLE, Differencing, LZW)**
- **Cryptography (RSA, DES/IDEA, SHA/MD5)**
- **Speech and handwriting recognition**
- **Operating systems/Networking (memcpy, memset, parity, checksum)**
- **Databases (hash/join, data mining, image/video serving)**
- **Language run-time support (stdlib, garbage collection)**

Data Parallelism & Loop Level Parallelism (LLP)

- **Data Parallelism:** Similar independent/parallel computations on different elements of arrays that usually result in independent (or parallel) loop iterations when such computations are implemented as sequential programs.
- A common way to increase parallelism among instructions is to exploit data parallelism among independent iterations of a loop (e.g exploit Loop Level Parallelism, LLP).
  - One method covered earlier to accomplish this is by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present. This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler/hardware to eliminate more stall cycles.

In scalar code

Usually: Data Parallelism → LLP

- The following loop has parallel loop iterations since computations in each iterations are data parallel and are performed on different elements of the arrays.

Scalar Code

```
for (i=1; i<=1000; i=i+1;)
        x[i] = x[i] + y[i];
```

4 vector instructions:

Vector Code

```
LV
LV
ADDV
SV
```

Load Vector X
Load Vector Y
Add Vector X, X, Y
Store Vector X

- In supercomputing applications, data parallelism/LLP has been traditionally exploited by vector ISAs/processors, utilizing vector instructions
  - Vector instructions operate on a number of data items (vectors) producing a vector of elements not just a single result value. The above loop might require just four such instructions.

Assuming Maximum Vector Length(MVL) = 1000 is supported otherwise a vector loop (i.e strip mining) is needed, more on this later

From 551

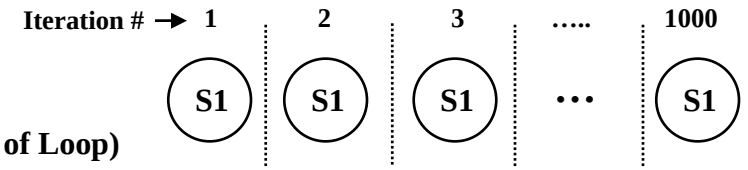
Loop-Level Parallelism (LLP) Analysis

- Loop-Level Parallelism (LLP) analysis focuses on whether data accesses in later iterations of a loop are data dependent on data values produced in earlier iterations and possibly making loop iterations independent (parallel).

e.g. in `for (i=1; i<=1000; i++)`

`x[i] = x[i] + s;` ← S1 (Body of Loop)

Usually: Data Parallelism → LLP in scalar code



Dependency Graph

the computation in each iteration is independent of the previous iterations and the loop is thus parallel. The use of X[i] twice is within a single iteration.

⇒ Thus loop iterations are parallel (or independent from each other).

Classification of Date Dependencies in Loops:

- Loop-carried Data Dependence: A data dependence between different loop iterations (data produced in an earlier iteration used in a later one).
- Not Loop-carried Data Dependence: Data dependence within the same loop iteration.
- LLP analysis is important in software optimizations such as loop unrolling since it usually requires loop iterations to be independent (and in vector processing).
- LLP analysis is normally done at the source code level or close to it since assembly language and target machine code generation introduces loop-carried name dependence in the registers used in the loop.
  - Instruction level parallelism (ILP) analysis, on the other hand, is usually done when instructions are generated by the compiler.

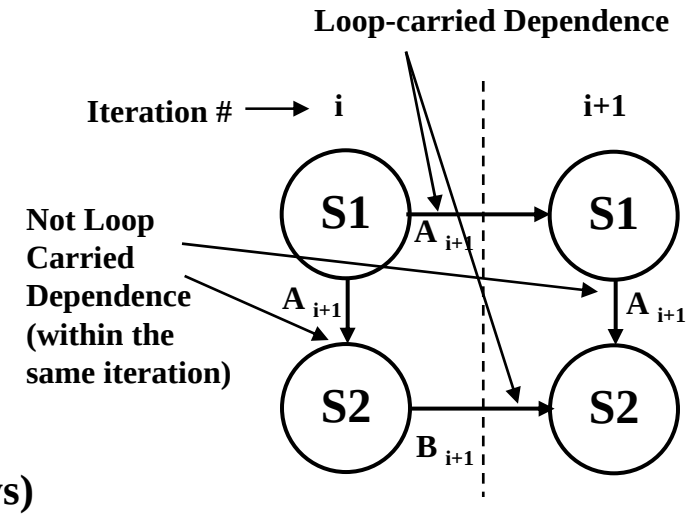
From 551

LLP Analysis Example 1

- **In the loop:**

```
for (i=1; i<=100; i=i+1) {
        A[i+1] = A[i] + C[i];  /* S1 */
        B[i+1] = B[i] + A[i+1];} /* S2 */
}
```

(Where A, B, C are distinct non-overlapping arrays)



Dependency Graph

– S2 uses the value A[i+1], computed by S1 in the same iteration. This data dependence is within the same iteration (not a loop-carried dependence).

⇒ **does not prevent loop iteration parallelism.**

i.e. S1 → S2 on A[i+1] Not loop-carried dependence

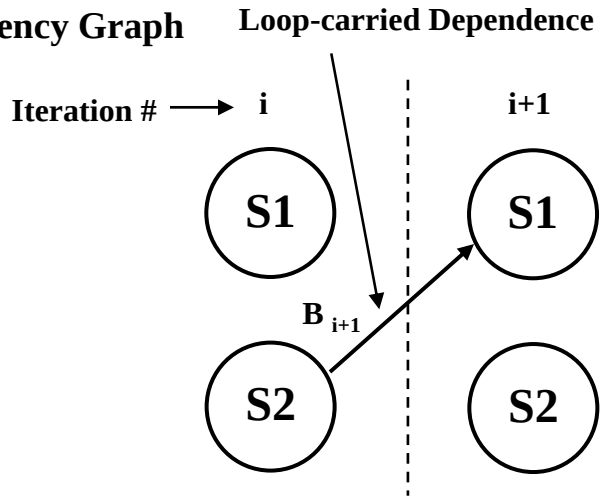
– S1 uses a value computed by S1 in the earlier iteration, since iteration i computes A[i+1] read in iteration i+1 (loop-carried dependence, prevents parallelism). The same applies for S2 for B[i] and B[i+1]

i.e. S1 → S1 on A[i] Loop-carried dependence
S2 → S2 on B[i] Loop-carried dependence

⇒ **These two data dependencies are loop-carried spanning more than one iteration (two iterations) preventing loop parallelism.**

In this example the loop carried dependencies form two dependency chains starting from the very first iteration and ending at the last iteration

LLP Analysis Example 2



- **In the loop:**

```
for (i=1; i<=100; i=i+1) {
    A[i] = A[i] + B[i];      /* S1 */
    B[i+1] = C[i] + D[i];    /* S2 */
}
```

  - S1 uses the value B[i] computed by S2 in the previous iteration (loop-carried dependence)

i.e. S2 → S1 on B[i] Loop-carried dependence

  - This dependence is not circular:
    - **S1 depends on S2 but S2 does not depend on S1.**
  - Can be made parallel by replacing the code with the following:

```
A[1] = A[1] + B[1];
for (i=1; i<=99; i=i+1)  {
    B[i+1] = C[i] + D[i];
    A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];
```

Scalar Code: Loop Start-up code

Vectorizable code

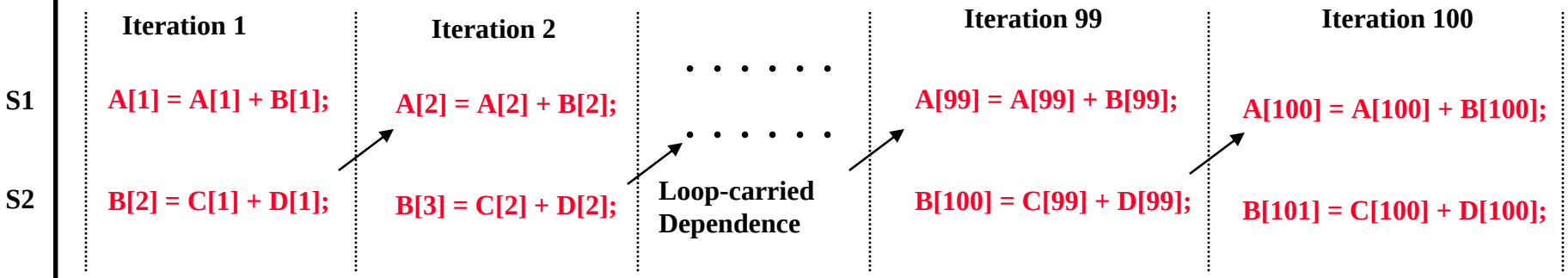
Parallel loop iterations (data parallelism in computation exposed in loop code)

Scalar Code: Loop Completion code

LLP Analysis Example 2

Original Loop:

```
for (i=1; i<=100; i=i+1) {
      A[i] = A[i] + B[i];      /* S1 */
      B[i+1] = C[i] + D[i];    /* S2 */
}
```

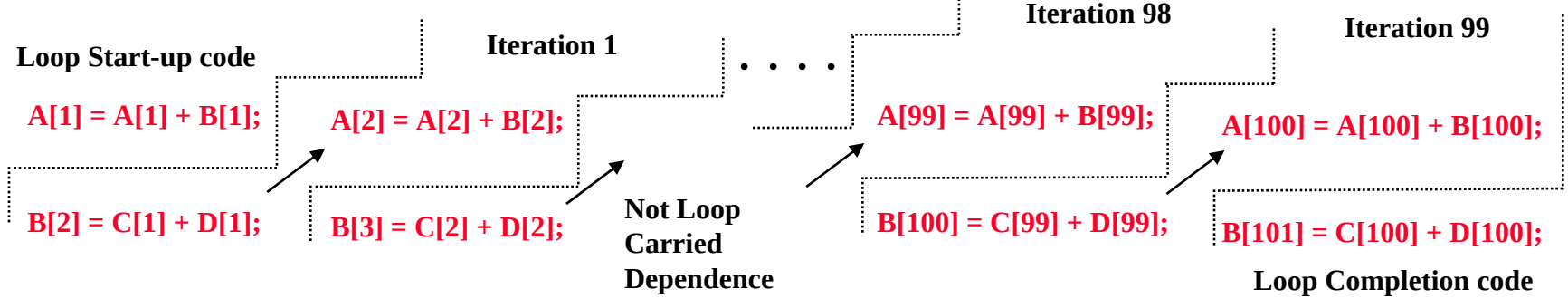


Modified Parallel Loop:

(one less iteration)

```
A[1] = A[1] + B[1];          Scalar Code: Loop Start-up code
for (i=1; i<=99; i=i+1) {
      B[i+1] = C[i] + D[i];
      A[i+1] = A[i+1] + B[i+1];
}
B[101] = C[100] + D[100];    Scalar Code: Loop Completion code
```

Vectorizable code



From 551

Properties of Vector Processors/ISAs

- **Each result (element) in a vector operation is independent of previous results (Data Parallelism, LLP exploited)**
  => Multiple pipelined Functional units (lanes) usually used, vector compiler ensures no dependencies between computations on elements of a single vector instruction
  => Higher clock rate (less complexity)
- **Vector instructions access memory with known patterns:**
  => Highly interleaved memory with multiple banks used to provide the high bandwidth needed and hide memory latency.
  => Amortize memory latency of over many vector elements.
  => No (data) caches usually used. (Do use instruction cache)

i.e. lower effective impact of memory latency

Thus more predictable performance

Up to MVL computations

- **A single vector instruction implies a large number of computations (replacing loops or reducing number of iterations needed)**

By a factor of MVL

  => Fewer instructions fetched/executed, TLB look-ups….
  => Reduces branches and branch problems (control hazards) in pipelines.

As if loop-unrolling by default MVL times?

Vector vs. Single-Issue Scalar Processor

Single-issue Scalar

- One instruction fetch, decode, dispatch per operation
- Arbitrary register accesses, adds area and power
- Loop unrolling and software pipelining for high performance increases instruction cache footprint
- All data passes through cache; waste power if no temporal locality
- One TLB lookup per load or store
- Off-chip access in whole cache lines

Vector

- One instruction fetch,decode, dispatch per vector (up to MVL elements)
- Structured register accesses
- Smaller code for high performance, less power in instruction cache misses
- Bypass cache (for data)
- One TLB lookup per group of loads or stores
- Move only necessary data across chip boundary

Vector vs. Superscalar Processors

Superscalar

- Control logic grows quad-ratically with issue width
- Control logic consumes energy regardless of available parallelism
  - Low Computational power efficiency (computations/watt)
- Dynamic nature makes real-time performance less predictable
- Speculation to increase visible parallelism wastes energy and adds complexity

Vector

- Control logic grows linearly with issue width
- Vector unit switches off when not in use - Higher energy efficiency
- More predictable real-time performance
- Vector instructions expose data parallelism without speculation
- Software control of speculation when desired:
  - Whether to use vector mask or compress/expand for conditionals

The above differences are <u>in addition</u> to the "Vector vs. Single-Issue Scalar Processor" differences (<u>from last slide</u>)

Changes to Scalar Processor to Run Vector Instructions

- **A vector processor typically consists of:**
  1- a pipelined scalar unit plus 2- a vector unit.

1 • **The scalar unit is basically not different than advanced pipelined CPUs, commercial vector machines have included both out-of-order scalar units (NEC SX/5) and VLIW scalar units (Fujitsu VPP5000).**

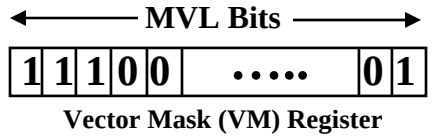
- **Computations that don't run in vector mode don't have high ILP, so can make scalar CPU simple.**

2 • **The vector unit supports a vector ISA including decoding of vector instructions which includes:**

  1 – **Vector functional units.** **Multiple Pipelined Functional Units (FUs) or lanes**

  2 – **ISA vector register bank.** **Each has MVL elements**

  3 – **Vector control registers**
    - **e,.g Vector Length Register (VLR), Vector Mask (VM)**



  4 – **Vector memory Load-Store Units (LSUs).**

  5 – **Multi-banked main memory** **To provide the very high data bandwidth needed**

- **Send scalar registers to vector unit (for vector-scalar ops).**
- **Synchronization for results back from vector register, including exceptions.**

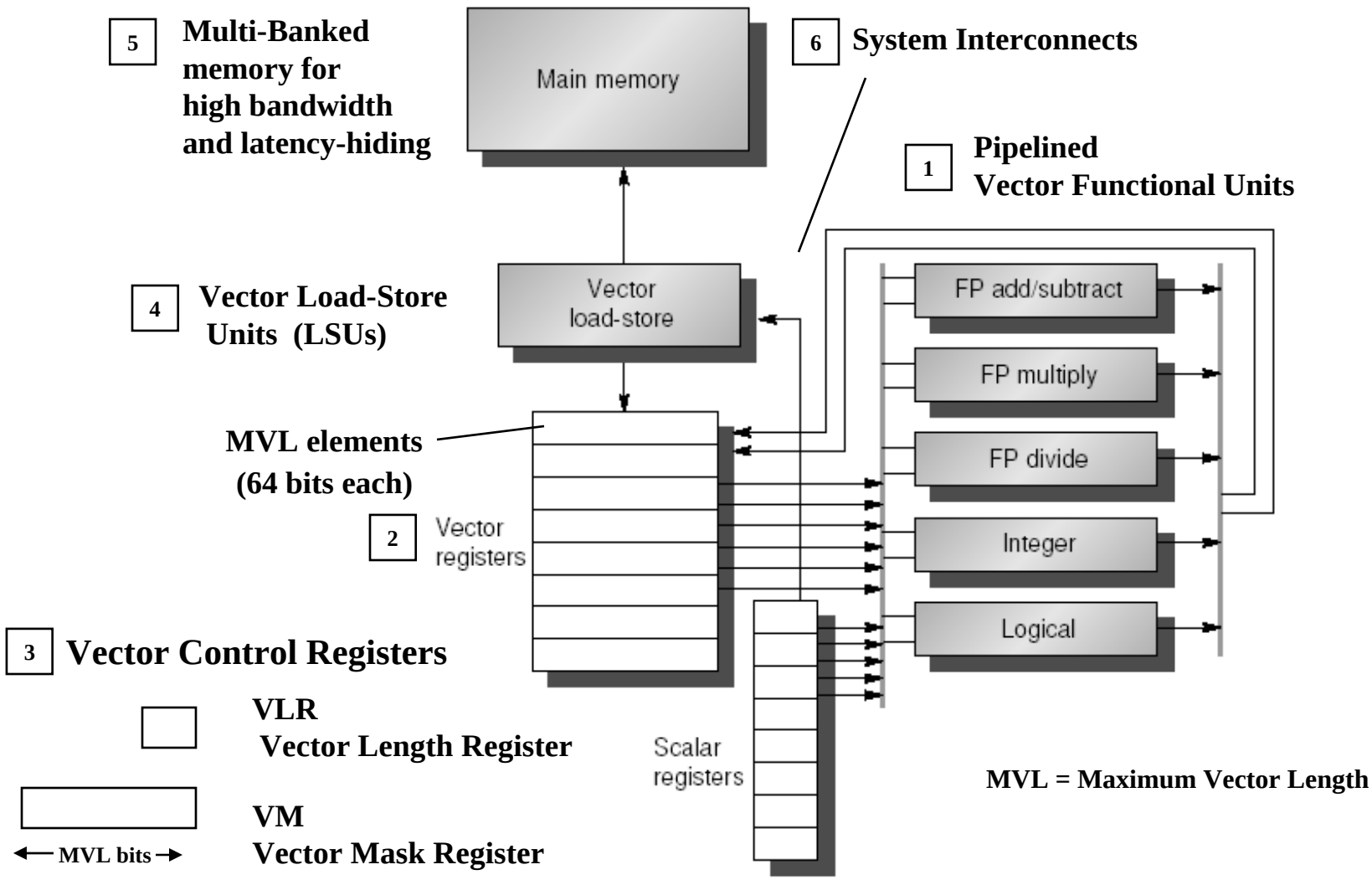
6 **+ System component interconnects**

Basic Types of Vector Architecture/ISAs

- **Types of architecture/ISA for vector processors:**
  - ***Memory-memory Vector ISAs/Processors*:**
    All vector operations are memory to memory
    (No vector ISA registers)
  - ***Vector-register ISAs/Processors*:**
    All vector operations between vector registers (except vector load and store)
    - **Vector equivalent of load-store scalar GPR architectures (ISAs)**
    - **Includes all vector machines since the late 1980 (Cray, Convex, Fujitsu, Hitachi, NEC)**

- **We assume vector-register for rest of the lecture.**

Basic Structure of Vector Register Architecture



Typical MVL = 64 (Cray)
MVL range 64-4096 (4K)

Components of Vector Processor

1 • ***Vector Functional Units (FUs)*: Fully pipelined, start new operation every clock**
  - **Typically 4 to 8 Fus (or lanes): FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit**

  (multiple lanes of the same type)

  More on lanes later

2 • ***ISA Vector Register Bank*: Fixed length bank holding vector ISA registers**
  - **Has at least 2 read and 1 write ports**
  - **Typically 8-32 vector registers, each holding MVL = 64-128 elements (typical, up to 4K possible) 64-bit elements.**

• ***ISA Scalar registers*: single element for FP scalar or address.**

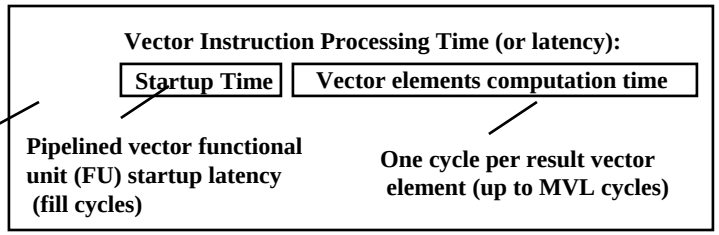
3 • ***Vector Control Registers*: Vector Length Register (VLR), Vector Mask Register (VM).**

4 • ***Vector Load-Store Units (LSUs)*: fully pipelined unit to load or store a vector; may have multiple LSUs.**

5 • **Multi-Banked memory: Multi-Banked memory for high throughput (bandwidth) and long latency-hiding.**

6 • **System Interconnects: Cross-bar to connect FUs , LSUs, registers, memory.**

Vector ISA Issues:
How To Pick Maximum Vector Length (MVL)?



- **Longer good because:**
  1) Hide vector startup time
  2) Lower instruction bandwidth — Fewer instructions fetched for a given computation
  3) Tiled access to memory reduce scalar processor memory bandwidth needs
  4) If known maximum length of app. is < MVL, no strip mining (vector loop) overhead is needed.
  5) Better spatial locality for memory access
- **Longer not much help because:**
  1) Diminishing returns on overhead savings as keep doubling number of elements.
  2) Need natural application vector length to match physical vector register length, or no help

i.e MVL

Media-Processing: Vectorizable? Vector Lengths?

Computational Kernel	Natural Application Vector length
• Matrix transpose/multiply	# vertices at once
• DCT (video, communication)	image width
• FFT (audio)	256-1024
• Motion estimation (video)	image width, iw/16
• Gamma correction (video)	image width
• Haar transform (media mining)	image width
• Median filter (image processing)	image width
• Separable convolution (img. proc.)	image width

MVL?

(from Pradeep Dubey - IBM, http://www.research.ibm.com/people/p/pradeep/tutor.html)

Vector Implementation

- **Vector register file:**
  - Each register is an array of MVL elements.
  - Size of each register is determined by the maximum vector length (MVL) supported by the implemented vector ISA.

Vector Control Registers

  - Vector Length Register (VLR) determines the actual vector length used for a particular vector operation or instruction.
  - Vector Mask Register (VM) determines which elements of a vector will be computed.

- **Multiple parallel execution units = "lanes" (sometimes called "pipelines" or "pipes") of the same type:**
  - Multiples pipelined functional units (lanes) are each assigned a number of computations of a single vector instruction.

Vector Lanes

$$\text{Vector Instruction Latency} = \text{Vector Startup Time} + \left\lceil \frac{\text{MVL}}{\text{N}} \right\rceil$$

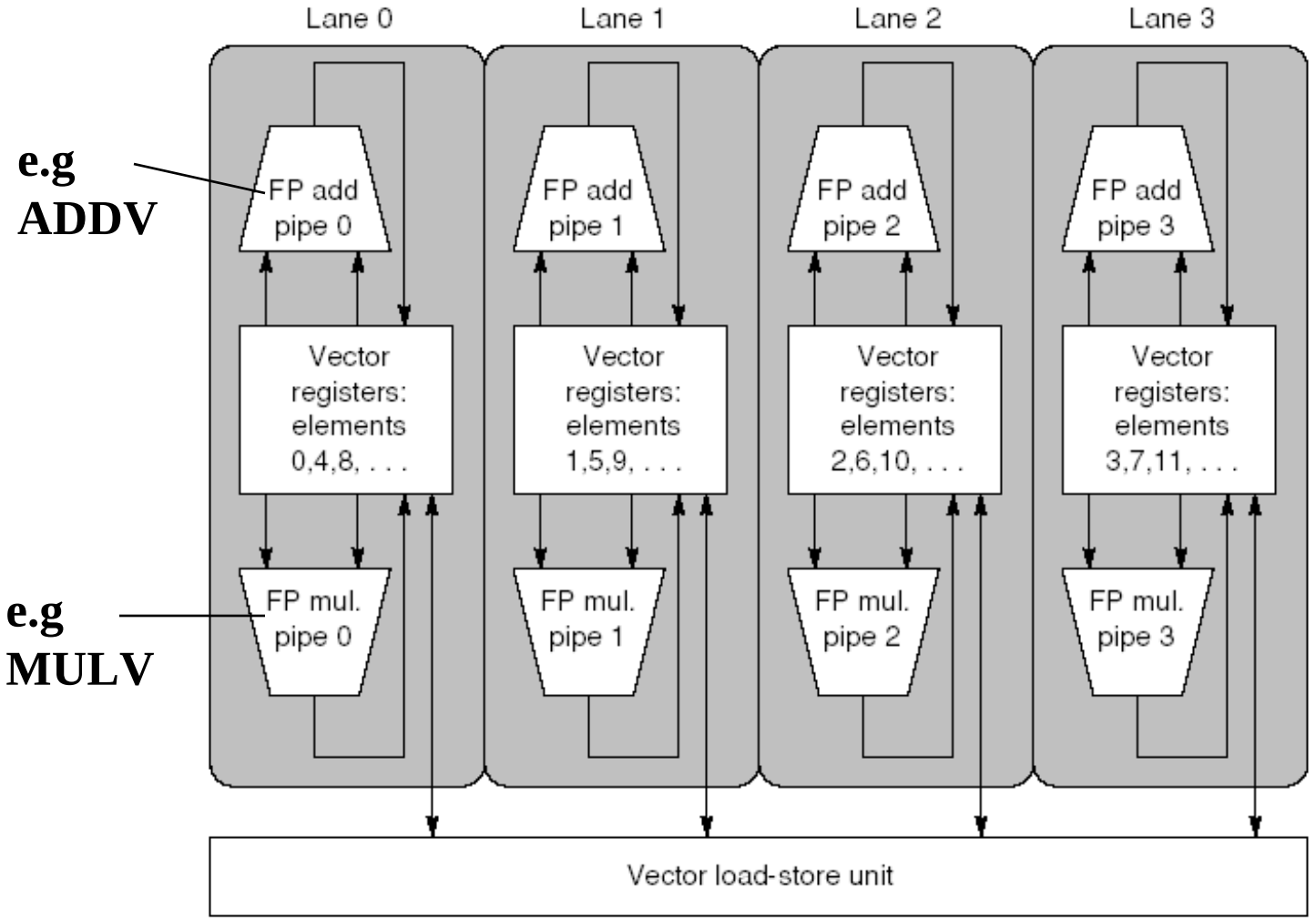
Where N is the number of lanes supported by the vector processor

- Thus, supporting multiple lanes in a vector processor reduces vector instruction latency by producing multiple elements of the result vector per cycle (after fill cycles).
- Having multiple lanes, however, does not reduce vector startup time (vector unit fill cycles).

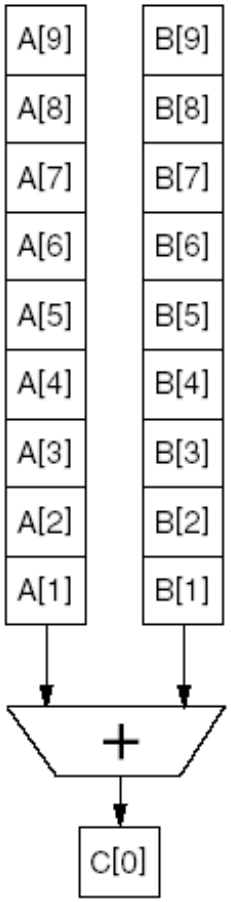
Processing time for a vector instruction in cycles

Structure of a Vector Unit Containing Four Lanes

Number of Lanes in a vector unit (processor): The number of vector functional units of the same type that are each assigned a number of computations of the same vector instruction



Using Multiple Lanes (Vector Functional Units) to Improve Performance of A Single Vector Add Instruction



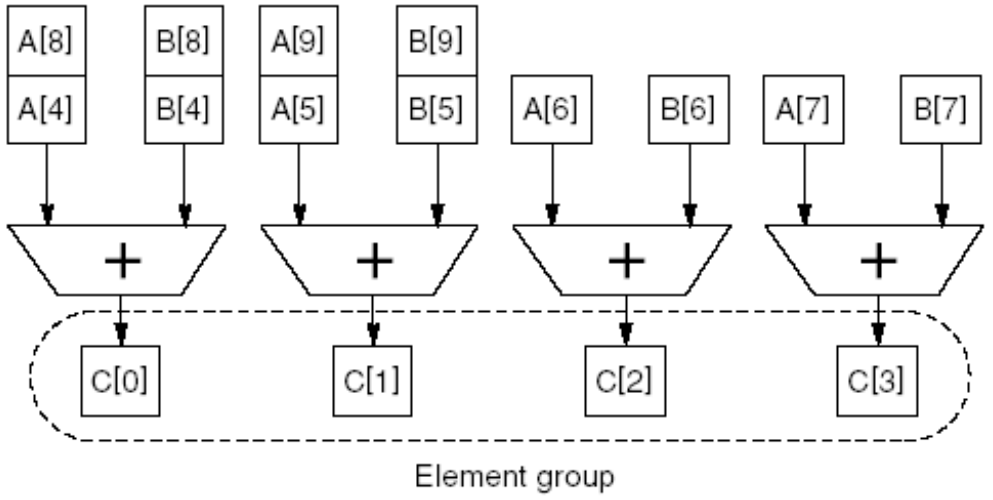
One Lane

(a)

Single Lane: **For vectors with nine elements (as shown)**
Time needed = 9 cycles + startup

(a) has a single add pipeline and can complete one addition per cycle. The machine shown in (b) has four add pipelines and can complete four additions per cycle.

Four Lanes: **For vectors with nine elements**
Time needed = 3 cycles + startup



(b)

Four Lanes

MVL lanes? Data parallel system, SIMD array?

Example Vector-Register Architectures

Processor (year)	Clock rate (MHz)	Vector registers	Elements per register (64-bit elements) (MVL)	Vector arithmetic units	Vector load-store units (LSUs)	Lanes
Cray-1 (1976)	80	8	64	6: FP add, FP multiply, FP reciprocal, integer add, logical, shift	1	1
Cray X-MP (1983) Cray Y-MP (1988)	118 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	1
Cray-2 (1985)	244	8	64	5: FP add, FP multiply, FP reciprocal/ sqrt, integer add/shift/population count, logical	1	1
Fujitsu VP100/ VP200 (1982)	133	8–256	32–1024	3: FP or integer add/logical, multiply, divide	2	1 (VP100) 2 (VP200)
Hitachi S810/ S820 (1983)	71	32	256	4: FP multiply-add, FP multiply/ divide-add unit, 2 integer add/logical	3 loads 1 store	1 (S810) 2 (S820)
Convex C-1 (1985)	10	8	128	2: FP or integer multiply/divide, add/ logical	1	1 (64 bit) 2 (32 bit)
NEC SX/2 (1985)	167	8 + 32	256	4: FP multiply/divide, FP add, integer add/logical, shift	1	4
Cray C90 (1991) Cray T90 (1995)	240 460	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store	2
NEC SX/5 (1998)	312	8 + 64	512	4: FP or integer add/shift, multiply, divide, logical	1	16
Fujitsu VPP5000 (1999)	300	8–256	128–4096	3: FP or integer multiply, add/logical, divide	1 load 1 store	16
Cray SV1 (1998) SV1ex (2001)	300 500	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	1 load-store 1 load	2 8 (MSP)
VMIPS (2001)	500	8	64	5: FP multiply, FP divide, FP add, integer add/shift, logical	1 load-store	1

Peak 133 MFLOPS

Vector processor family Used in Earth Simulator

VMIPS = Vector MIPS

The VMIPS Vector FP Instructions

8 Vector Registers
V0-V7
MVL = 64
(Similar to Cray)

Instruction	Operands	Function
ADDV.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.
SUBV.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.
MULV.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.
DIVV.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 from 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--V.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 the vector-mask register 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 the vector-length register. Move the contents of the vector-length register to R1.
MVTM MVFM	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

Vector FP

Vector Memory

Vector Index

Vector Mask (VM)

Vector Length (VLR)

Vector Memory Access Addressing Modes

1- Unit Stride Access

2- Constant Stride Access

3- Variable Stride Access (indexed)

Vector Control Registers: VM = Vector Mask
VLR = Vector Length Register

Basic Vector Memory Access Addressing Modes

- Load/store operations move groups of data between registers and memory
- Three types of addressing:

1 – **Unit stride** Fastest memory access — Sequential element access, i.e Stride = 1

LV (Load Vector), SV (Store Vector):

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.

2 – **Non-unit** (constant) **stride**

LVWS (Load Vector With Stride), SVWS (Store Vector With Stride):

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 from address at R1 with stride in R2, i.e., R1+i × R2.

3 – **Indexed** (gather-scatter) — Or Variable stride (i size of element)

- Vector equivalent of register indirect
- Good for sparse arrays of data
- Increases number of programs that vectorize

LVI (Load Vector Indexed or Gather), SVI (Store Vector Indexed or Scatter):

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.

Stride = Distance in elements between consecutive vector elements loaded or stored by a vector memory access instruction

Scalar Vs. Vector Code Example

DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64 =MVL

Scalar vs. Vector

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

VLR = 64
VM = (1,1,1,1 ..1)

```
      L.D      F0,a
      DADDIU   R4,Rx,#512      ;last address to load
loop: L.D      F2, 0(Rx)       ;load X(i)
      MUL.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
```

As if the scalar loop code was unrolled MVL = 64 times:
Every vector instruction replaces 64 scalar instructions.

Scalar Vs. Vector Code

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Unroll scalar loop code?
What does loop unrolling accomplish?

Vector Execution Time/Performance

In seconds or cycles

- Time = f(vector length, data dependencies, struct. hazards, C)
- ***Initiation rate***: rate that FU consumes vector elements. (= number of lanes; usually 1 or 2 on Cray T-90)
- ***Convoy***: a set of vector instructions that can begin execution in approximately the same clock cycle (no structural or data hazards).
- ***Chime***: approx. time in cycles to produce a vector element result (usually = number of convoys in vector code).

Assuming one lane is used/ignore startup

- ***m* convoys take T_{chime} = *m* cycles (or 1 chime)**; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead; one lane; good approximation for long vectors)

i.e vector functional unit startup time etc.

Assuming vector length, n ≤ MVL

Convoy

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

4 conveys, 1 lane, VL= n = 64
=> 4 x 64 = 256 cycles
(or m= 4 cycles per result vector element)

Ignoring vector startup time, n <= MVL

DAXPY

DAXPY (Y = a * X + Y) Timing

(One LSU, One Lane, No Vector Chaining, Ignoring Startup)

$n \leq MVL$

From Last Slide: Time in Cycles = Number of Convoys x Vector Length = T_{chime} x n = m x n

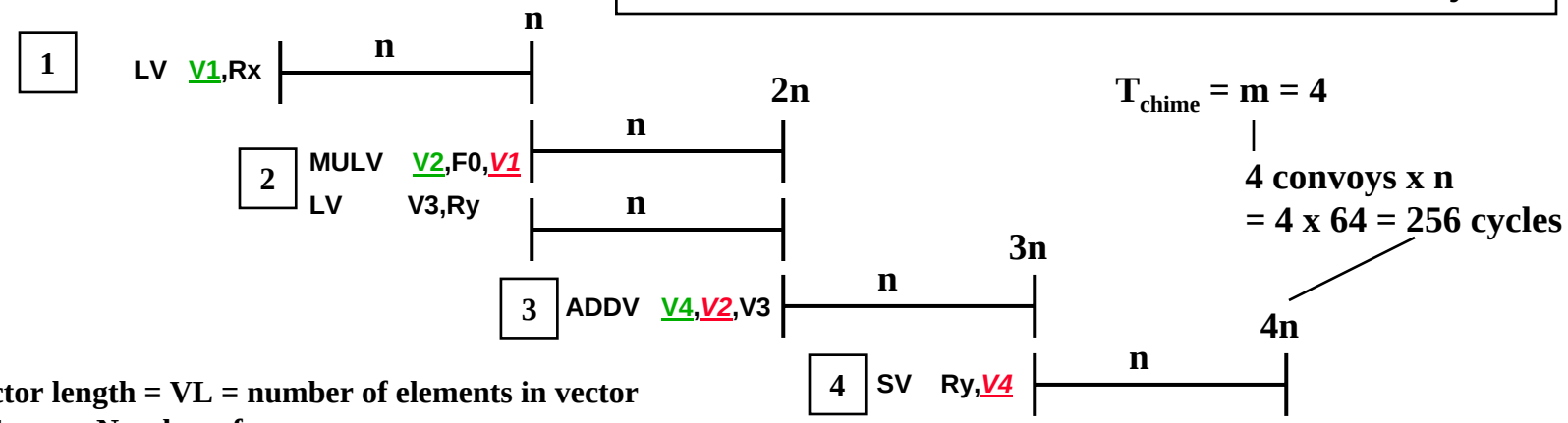
Convoy

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

m= 4 conveys, 1 lane,
VL = n = 64
=> 4 x 64 = 256 cycles
(or T_{chime}= 4 cycles per result vector element)

m x n

m = 4 Convoys or T_{chime} = 4 cycles per element
n elements take = m x n = 4 n cycles
For n = VL = MVL = 64 it takes 4x64 = 256 cycles



n = vector length = VL = number of elements in vector
m or T_{chime} = Number of convoys

What if multiple lanes are used?

Accounting For Vector FU Start-up Time

i.e Pipelined Vector Functional Unit Fill Cycles

- ***Start-up time*: pipeline latency time (depth of FU pipeline); another sources of overhead**
- **Operation Start-up penalty (from CRAY-1)**
  - **Vector load/store 12**
  - **Vector multiply 7**
  - **Vector add 6**

Time to get first result element (To account for pipeline fill cycles)

Assume convoys don't overlap (no vector chaining); vector length = n:

Accounting For Startup Time (for one lane):

n ≤ MVL

Time in Cycles = Total Startup + Number of Convoys x Vector Length = = Total Startup + m x n

Convoy	*Start*	*1st result*	*last result*
1. LV	0	12	11+n (12+n-1)
2. MULV, LV	12+n	12+n+12	23+2n *Load start-up*
3. ADDV	24+2n	24+2n+6	29+3n *Wait convoy 2*
4. SV	30+3n	30+3n+12	41+4n *Wait convoy 3*

DAXPY

Total Startup cycles

4 Convoys

DAXPY (Y = a * X + Y) Timing

(One LSU, One Lane, One LSU, No Vector Chaining, Including Startup)

Time in Cycles = Total Startup + Number of Convoys x Vector Length = = Total Startup + m x n

Convoy

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

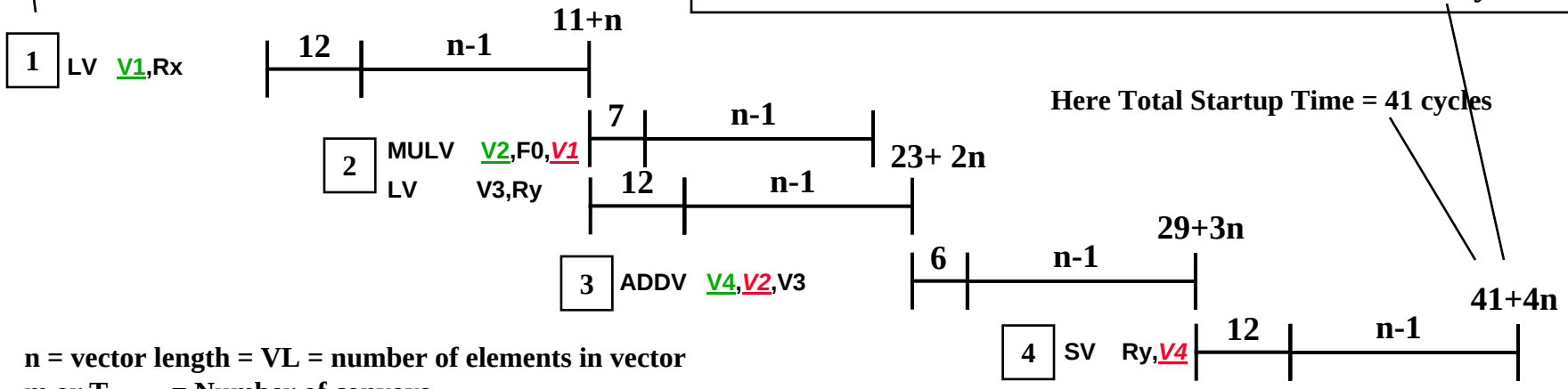
Time to get first result element

Operation Start-up penalty (from CRAY-1)

- –Vector load/store 12
- –Vector multiply 7
- –Vector add 6

297 cycles Vs. 256 cycles when ignoring startup time (slide 30)

m = 4 Convoys or T_{chime} = 4 cycles per element
n elements take = Startup + m x n = 41 + 4 n cycles
For n = VL = MVL = 64 it takes 41 + 4x64 = 297 cycles



n = vector length = VL = number of elements in vector
m or T_{chime} = Number of convoys

VEC-1

What if multiple lanes are used?

Vector Load/Store Units (LSUs) & Memories

- Start-up overheads usually longer for LSUs
- Memory system must sustain (# lanes x word) /clock cycle
- Many Vector Procs. use banks (vs. simple interleaving):

  1) support multiple loads/stores per cycle
  => multiple banks & address banks independently

  2) support non-sequential accesses (non unit stride)

- Note: No. memory banks > memory latency to avoid stalls
  - *m* banks => *m* words per memory lantecy *l* clocks
  - if *m* < *l*, then gap in memory pipeline:

clock: 0 … *l* *l*+1 *l*+2 … *l*+*m*- 1 l+m … 2 *l*

word: -- … 0 1 2 … *m*-1 -- … *m*

  - may have 1024 banks in SRAM

CPU

i.e to hide memory latency

i.e a large number of memory banks maybe needed

Vector Memory Requirements Example

- **The Cray T90 has a CPU clock cycle of 2.167 ns (460 MHz) and in its largest configuration (Cray T932) has 32 processors each capable of generating four loads and two stores per CPU clock cycle.** **i.e Each processor has 6 LSUs**
- **The CPU clock cycle is 2.167 ns, while the cycle time of the SRAMs used in the memory system is 15 ns.**
- **Calculate the minimum number of memory banks required to allow all CPUs to run at full memory bandwidth.**
- ***Answer:***
- **The maximum number of memory references each cycle is 192 (32 CPUs times 6 references per CPU).**
- **Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles, which we round up to 7 CPU clock cycles. Therefore we require a minimum of 192 × 7 = 1344 memory banks!** **CPU**
- **The Cray T932 actually has 1024 memory banks, and so the early models could not sustain full bandwidth to all CPUs simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth/latency.**

Note: No Data cache is used

Vector Memory Access Pattern Example

- **Suppose we want to fetch a vector of 64 elements (each element 8 bytes) starting at byte address 136, and a memory access takes 6 CPU clock cycles.**
  - **How many memory banks must we have to support one fetch per clock cycle?**
  - **With what addresses are the banks accessed?**
  - **When will the various elements arrive at the CPU?**

MVL

Assuming stride distance = 1 element = 8 bytes (sequential element access)

Answer

- **Six clocks per access require at least six banks, but because we want the number of banks to be a power of two, we choose to have eight banks as shown on next slide**

Not a requirement just assumed In this example

More than minimum number of memory banks needed to hide individual memory bank latency

Vector Memory Access Pattern Example

Unit Access Stride Shown (Access Stride = 1 element = 8 bytes)

LSU or memory bank startup latency (6 cycles)

8 Banks

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

What if access stride = 8 elements (which equals number of banks)?

Vector Length (VL or n) Needed Not Equal to MVL

- What to do when vector length is not exactly 64?
- *vector-length register* (VLR) controls the length of any vector operation, including a vector load or store. (cannot be > MVL = the length of vector registers)

n ≤ MVL

```
      do 10 i = 1, n
10    Y(i) = a * X(i) + Y(i)
```

Vector length = n

- Don't know n until runtime!

n > MVL

What if n > Max. Vector Length (MVL)?

→ • Vector Loop (Strip Mining)

What if n > MVL?

n = vector length = VL = number of elements in vector

Vector Loop :Strip Mining

- Suppose Vector Length > Max. Vector Length (MVL)?
- ***Strip mining*: generation of code such that each vector operation is done for a size Š to the MVL**
- 1st loop do short piece (n mod MVL), reset VL = MVL

Vector Loop

n

(For other iterations)

VL=Vector Length Control Register

```
low = 1
VL = (n mod MVL)  /*find the odd size piece*/
do 1 j = 0,(n / MVL)  /*outer loop*/
        do 10 i = low,low+VL-1  /*runs for length VL*/
                Y(i) = a*X(i) + Y(i)  /*main operation*/
10      continue
        low = low+VL  /*start of next vector*/
        VL = MVL  /*reset the length to max*/
1       continue
```

First iteration: < MVL elements

Other iterations: MVL elements

Time for Vector Loop:

One lane assumed here

Startup Time

Number of Convoys = m

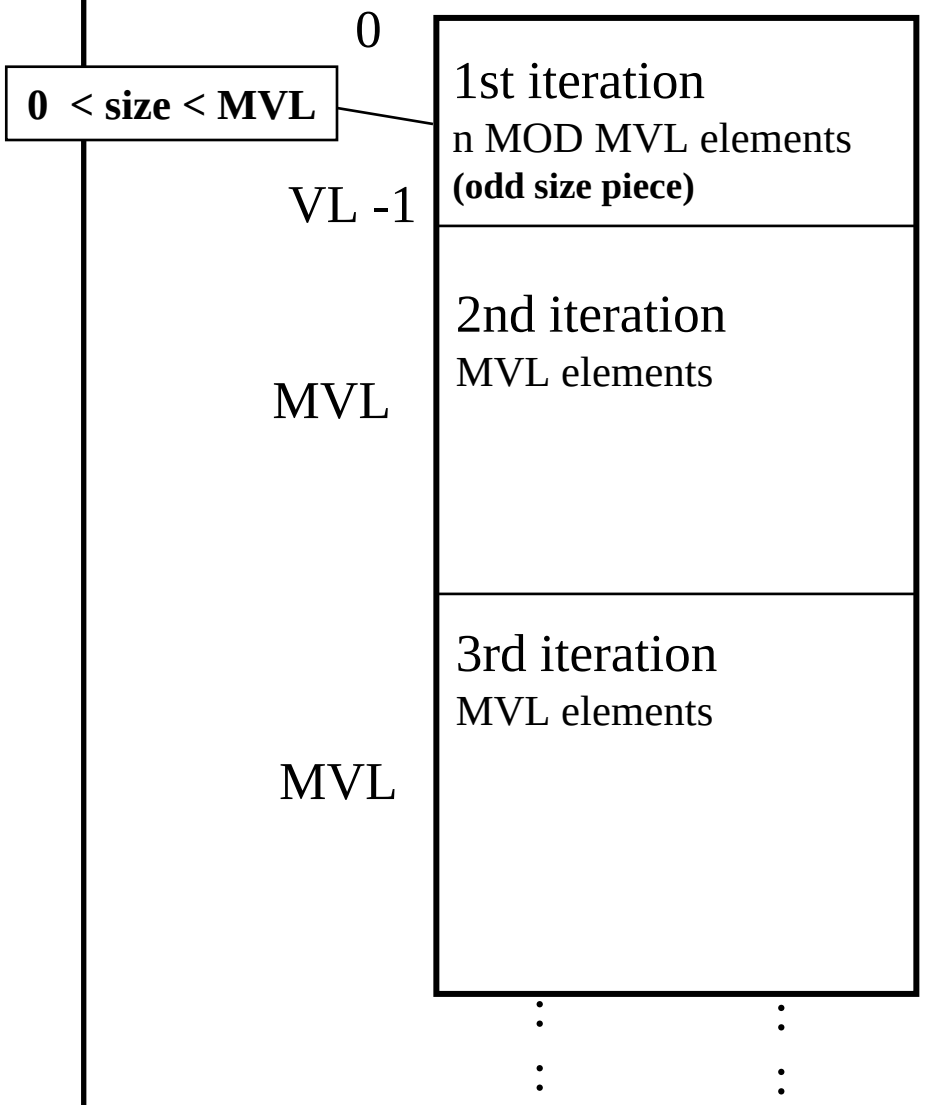
$$T_n = \left\lceil \frac{n}{\text{MVL}} \right\rceil \times (T_{\text{loop}} + T_{\text{start}}) + n \times T_{\text{chime}}$$

Vector loop iterations needed

Loop Overhead

Number of elements (i.e vector length)

Strip Mining Illustration



For First Iteration (shorter vector)
Set VL = n MOD MVL

For MVL = 64 VL = 1 - 63

For second Iteration onwards
Set VL = MVL

(e.g. VL = MVL = 64)

Number of Vector loop iterations:

$\lceil n/MVL \rceil$ vector loop iterations needed

n= Number of elements (i.e vector length)

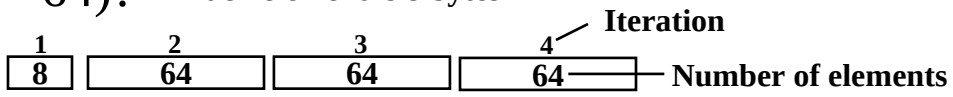
VL = Vector Length Control Register

Strip Mining Example

MVL =64
n = 200

- What is the execution time on VMIPS for the vector operation A = B × s, where s is a scalar and the length of the vectors A and B is 200 (MVL supported =64)? **Each element is 8 bytes** **n = vector length**

Answer



- Assume the addresses of A and B are initially in Ra and Rb, s is in Fs, and recall that for MIPS (and VMIPS) R0 always holds 0.
- Since (200 mod 64) = 8, the first iteration of the strip-mined loop will execute for a vector length of VL = 8 elements, and the following iterations will execute for a vector length = MVL = 64 elements.
- The starting byte addresses of the next segment of each vector is eight times the vector length. Since the vector length is either 8 or 64, we increment the address registers by 8 × 8 = 64 after the first segment and 8 × 64 = 512 for later segments.
- The total number of bytes in the vector is 8 × 200 = 1600, and we test for completion by comparing the address of the next vector segment to the initial address plus 1600.
- Here is the actual code follows: **(next)** ⟶

Strip Mining (Vector Loop) Example

4 vector loop iterations

VLR = n MOD 64 = 200 MOD 64 = 8
For first iteration only

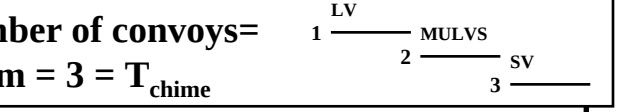
```
        DADDUI   R2,R0,#1600  ;total # bytes in vector
        DADDU    R2,R2,Ra     ;address of the end of A vector
        DADDUI   R1,R0,#8     ;loads length of 1st segment
        MTC1     VLR,R1       ;load vector length in VLR
        DADDUI   R1,R0,#64    ;length in bytes of 1st segment
        DADDUI   R3,R0,#64    ;vector length of other segments
Loop:   LV       V1,Rb        ;load B
        MULVS.D  V2,V1,Fs     ;vector * scalar
        SV       Ra,V2        ;store A
        DADDU    Ra,Ra,R1     ;address of next segment of A
        DADDU    Rb,Rb,R1     ;address of next segment of B
        DADDUI   R1,R0,#512   ;load byte offset next segment
        MTC1     VLR,R3       ;set length to 64 elements
        DSUBU    R4,R2,Ra     ;at the end of A?
        BNEZ     R4,Loop      ;if not, go back
```

VLR = 8 elements

= 64 bytes

= 64 elements

Number of convoys= m = 3 = T_{chime}



+ 64

+ 64

64 x 8 = 512 bytes

VLR = 64 elements

MTC1 VLR,R1 Move contents of R1 to the vector-length register.

VLR = MVL= 64
for second iteration
onwards

A = B x s n= 200 elements s in Fs
Start Addresses: A in Ra B in Rb

A = B × s

Strip Mining Example

Cycles Needed

The three vector instructions in the loop are dependent and must go into three convoys, hence $T_{chime} = 3$. Let's use our basic formula:

m=3

1 LV
2 MULVS
3 SV

Loop overhead

m = number of convoys

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

n = 200
MVL = 64

4 iterations

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

The value of T_{start} is the sum of

Startup time calculation

- The vector load start-up of 12 clock cycles
- A 7-clock-cycle start-up for the multiply
- A 12-clock-cycle start-up for the store

T_{loop} = loop overhead = 15 cycles **(assumed/given)**

Thus, the value of T_{start} is given by

Total Startup Time: $T_{start} = 12 + 7 + 12 = 31$

So, the overall value becomes

Total time in cycles: $T_{200} = 660 + 4 \times 31 = 784$

784 cycles / 200 elements = 3.9 cycles/element

3.9 vs. m =3

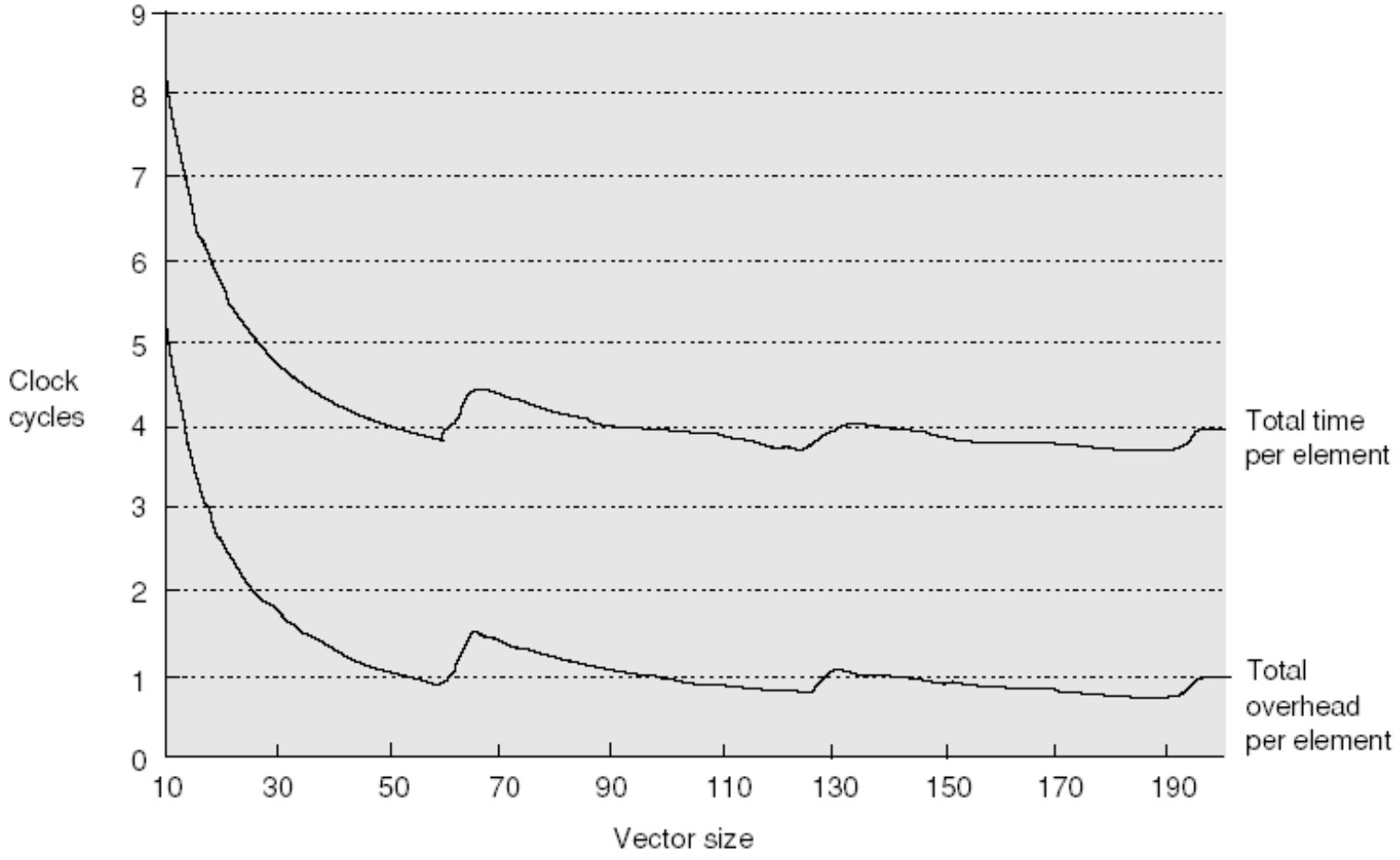
The execution time per element with all start-up costs is then 784/200 = 3.9, compared with a chime approximation of three.

m=3

Ideally 3.9/3 = 1.3 times faster (ignoring loop/startup overheads)

Strip Mining Example

The total execution time per element and the total overhead time per element versus the vector length for the strip mining example

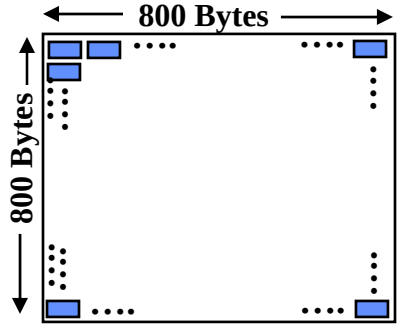


MVL supported = 64

Vector Memory Access Addressing: Constant Vector Stride

- **Suppose adjacent vector elements not sequential in memory**

Example: Matrix multiplication (each element size =8 bytes)




```
      do 10 i = 1,100
         do 10 j = 1,100
            A(i,j) = 0.0
            do 10 k = 1,100
10             A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

Depends if matrix is stored row-wise or column-wise

Vector dot product

Or 100 elements = stride

- **Either B or C accesses not adjacent (800 bytes between)**
- ***stride*: distance separating elements that are to be merged into a single vector (caches do unit stride)**
  => LVWS (load vector with stride) instruction

  In number of elements or in bytes

  LVWS V1,(R1,R2) Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.

  => SVWS (store vector with stride) instruction

  i = element size

  SVWS (R1,R2),V1 Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
- **Strides => can cause bank conflicts and stalls may occur.**

Here stride is constant > 1 element (100 elements)

Vector Stride Memory Access Example

- Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

element

Note: Multiple of memory banks number (32 = 4 x 8)

Answer

- Since the number of banks is larger than the bank busy time, for a stride of 1, the load will take 12 + 64 = 76 clock cycles, or 1.2 clocks per element.

Startup latency

- The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks.

Memory Bank Conflicts (collisions)

- Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time.
- The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clocks per element.

Stride = Multiple the number of banks ⟶ Bank Conflicts

Vector Operations Chaining

(AKA Vector Data Forwarding or Convoy Overlap)

- **Suppose:**

  MULV.D V1,V2,V3

  ADDV.D V4,*V1*,V5 ; separate convoys?

Vector version of result data forwarding

- ***chaining*: vector register (V1) is not treated as a single entity but as a group of individual registers, then 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 ports**

And reduces number of convoys = T_{chime} = m

- **As long as enough HW is available , increases convoy size**

Overlap convoys

- **With chaining, the above sequence is treated as a single convoy and the total running time becomes:**

$$\text{Vector length} + \text{Start-up time}_{ADDV} + \text{Start-up time}_{MULV}$$

Assuming n ≤ MVL i.e no strip mining (vector loop) is needed and also assuming one lane is used

Vector Chaining Example

For one lane

- **Timings for a sequence of dependent vector operations**

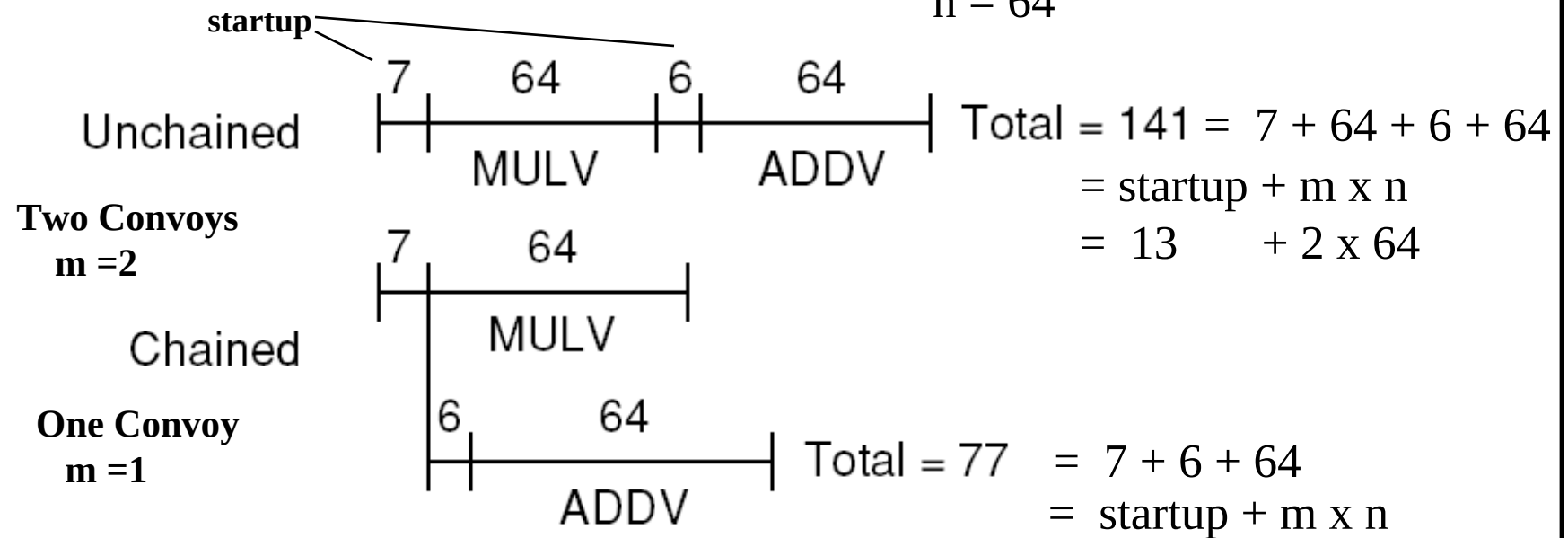
MULV.D V1,V2,V3

ADDV.D V4,V1,V5

both unchained and chained.

m convoys with n elements take:
startup + m x n cycles

Here startup = 7 + 6 = 13 cycles
n = 64



141 / 77 = 1.83 times faster with chaining

Assuming n ≤ MVL i.e no strip mining (vector loop) is needed and also assuming one lane is used

DAXPY (Y = a * X + Y) Timing

(One Lane, One LSU, With Vector Chaining, Including Startup)

With One LSU

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

Operation Start-up penalty (from CRAY-1)

- –Vector load/store 12
- –Vector multiply 7
- –Vector add 6

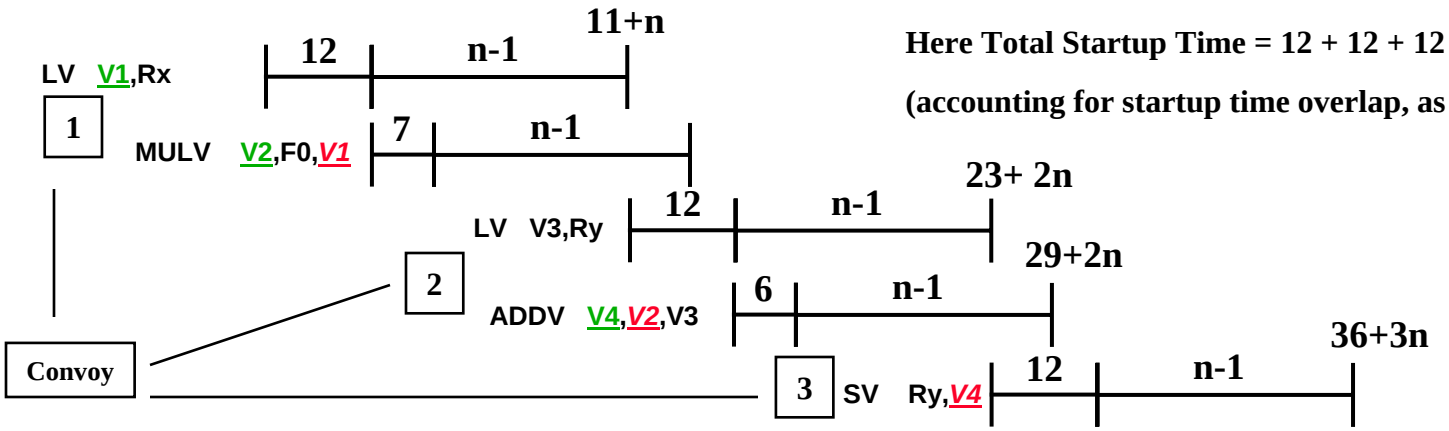
DAXPY With Chaining and One LSU (Load/Store Unit)

Was 4 convoys without chaining

228 cycles vs. 297 (no chaining slide 32)

m = 3 Convoys or T_{chime} = 3 cycles per element
n elements take = Startup + m x n = 36 + 3 n cycles
For n = VL = MVL = 64 it takes 36 + 3x64 = 228 cycles

3 Convoys:
m = T_{chime} = 3



Here Total Startup Time = 12 + 12 + 12 = 36 cycles

(accounting for startup time overlap, as shown)

n = vector length = VL = number of elements in vector

Time = Startup + m x n

DAXPY (Y = a * X + Y) Timing

(One Lane, With Vector Chaining, Including Startup)

With Three LSUs

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

Operation Start-up penalty (from CRAY-1)

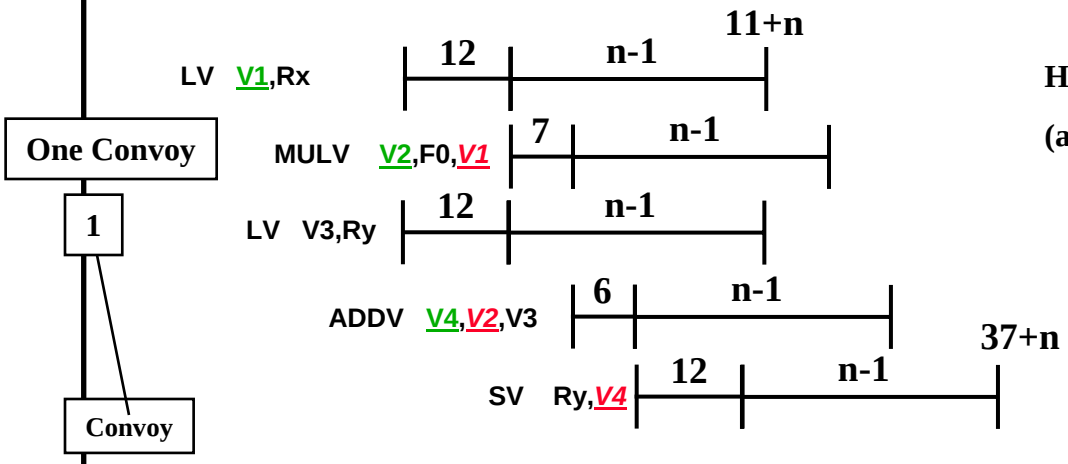
- –Vector load/store 12
- –Vector multiply 7
- –Vector add 6

DAXPY With Chaining and Three LSUs (Load/Store Units)

Still one lane

m = 1 Convoy or T_{chime} = 1 cycle per element
n elements take = Startup + m x n = 37 + n cycles
For n = VL = MVL = 64 it takes 37 + 1 x64 = 101 cycles

One Convoy: m = T_{chime} = 1



Here Total Startup Time = 12 + 7 + 6 + 12 = 37 cycles

(accounting for startup time overlap, as shown)

101 cycles vs. 228 cycles For one LSU (228/101 = 2.57 times faster)

Why not 3 times faster?

n = vector length = VL = number of elements in vector

Time = Startup + m x n

Vector Conditional Execution Using Vector Mask (VM)

Or Vector Element Masking

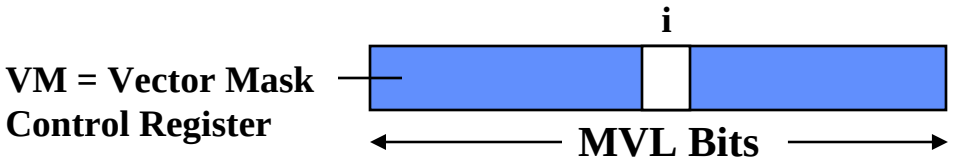
- Suppose:

```
      do 100 i = 1, 64
            if (A(i) .ne. 0) then
                  A(i) = A(i) - B(i)
            endif
100   continue
```

Vector Element test

Not equal

VM(i) =
1 = compute element i
0 = mask element i
(i.e do not compute element i)



- *vector-mask control* takes a Boolean vector: when *vector-mask (VM) register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.

i.e vector mask, VM bits = 1

- Still requires a clock cycle or more per element even if result not stored or computed.

Vector Conditional Execution Example

```
      do 100 i = 1, 64
            if (A(i).ne. 0) then
                  A(i) = A(i) – B(i)
            endif
100   continue
```

```
LV        V1,Ra        ;load vector A into V1
LV        V2,Rb        ;load vector B into V2
L.D       F0,#0        ;load FP zero into F0
SNEVS.D   V1,F0        ;sets VM(i) to 1 if V1(i)!=F0
SUBV.D    V1,V1,V2     ;subtract under vector mask
CVM                    ;set the vector mask to all 1s
SV        Ra,V1        ;store the result in A
```

Unit Stride Vector Load

Set mask

Set Mask

Clear mask

Clear Mask

S--V.D V1, V2
S--VS.D V1, F0

scalar

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.

Here in F0

Vector element test and set Vector Mask (VM) instructions

LV, SV Load/Store vector with stride 1
VM = Vector Mask Control Register

Vector Memory Operations/Addressing: Gather, Scatter

- **Suppose:** Variable Stride Vector Memory Access (or Indexed LVI, SVI)

```
        do   100 i = 1,n
100     A(K(i)) = A(K(i)) + C(M(i))
```

- ***gather*** **(`LVI`, load vector indexed), operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector =>** i.e dense **a nonsparse vector in a vector register**

LVI V1,(R1+V2) Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.

- **After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store (`SVI`, store vector indexed), using the same or different index vector**

SVI (R1+V2),V1 Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.

- **Can't be done by compiler since can't know K(i), M(i) elements**
- **Use `CVI` (create vector index) to create index 0, 1xm, 2xm, ..., 63xm**

Very useful for sparse matrix operations (few non-zero elements to be computed)

Gather, Scatter Example

```
        do      100 i = 1,n
100             A(K(i)) = A(K(i)) + C(M(i))
```

For data vectors — For Index vectors

Assuming that Ra, Rc, Rk, and Rm contain the starting addresses of the vectors in the previous sequence, the inner loop of the sequence can be coded with vector instructions such as:

```
LV      Vk,Rk          ;load K  (get index vector K)
LVI     Va,(Ra+Vk)     ;load A(K(I))
LV      Vm,Rm          ;load M  (get index vector M)
LVI     Vc,(Rc+Vm)     ;load C(M(I))
ADDV.D  Va,Va,Vc       ;add them
SVI     (Ra+Vk),Va     ;store A(K(I))
```

Index Vectors

Gather elements — LVI

Compute on dense vector — ADDV.D

Scatter results — SVI

Index vectors computed earlier — Vk, Vm

LVI V1, (R1+V2) (Gather) Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.

LVI = Load vector indexed

SVI (R1+V2), V1 (Scatter) Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.

SVI = Store vector indexed

Assuming Index vectors Vk Vm already initialized

Vector Conditional Execution Using Masking + Gather, Scatter

- **The indexed loads-stores and the create an index vector CVI instruction provide an alternative method to support conditional vector execution.**

```
do 100 i = 1, 64
        if (A(i).ne. 0) then
                A(i) = A(i) – B(i)
        endif
100     continue
```

V2 Index Vector
VM Vector Mask
VLR = Vector Length Register

```
LV          V1,Ra        ;load vector A into V1
L.D         F0,#0        ;load FP zero into F0
SNEVS.D     V1,F0        ;sets the VM to 1 if V1(i)!=F0
CVI         V2,#8        ;generates indices in V2
POP         R1,VM        ;find the number of 1's in VM
MTC1        VLR,R1       ;load vector-length register
CVM                      ;clears the mask
LVI         V3,(Ra+V2)   ;load the nonzero A elements
LVI         V4,(Rb+V2)   ;load corresponding B elements
SUBV.D      V3,V3,V4     ;do the subtract
SVI         (Ra+V2),V3   ;store A back
```

Set mask

Get all elements of A (stride = 1)

Vector element size

Create Index Vector

Set mask

To get dense vector length

Gather Non-zero elements

Compute on dense vector

Scatter results

CVI V1,R1 Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.

R1 has Vector element size

Vector Example with Dependency: Matrix Multiplication

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```

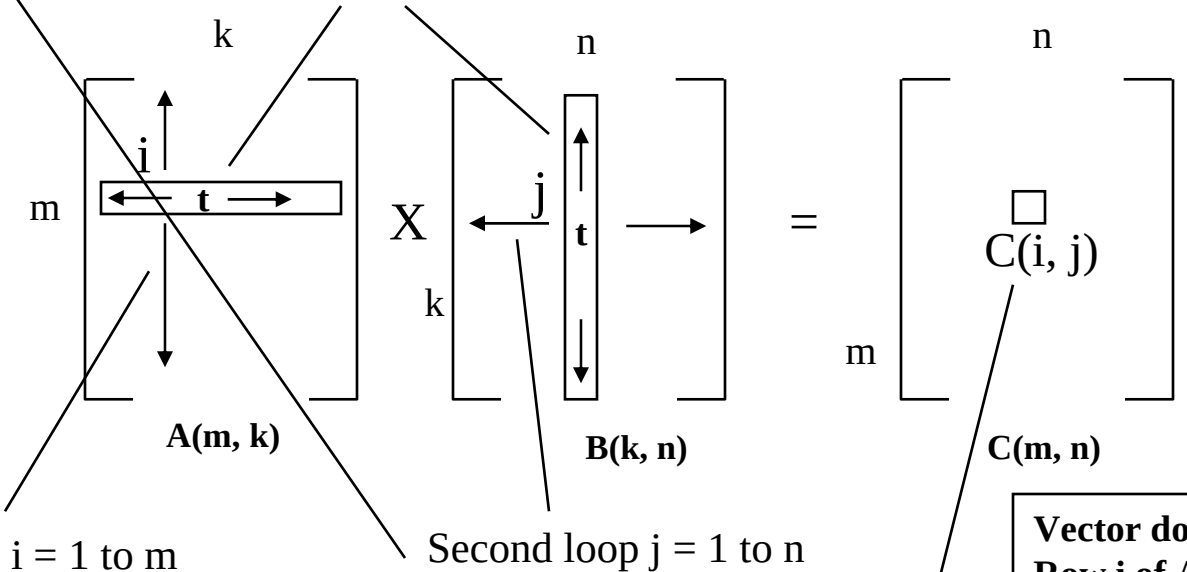
$C_{mxn} = A_{mxk} \; X \; B_{kxn}$

Dot product
(two vectors of size k)

Scalar Matrix Multiplication

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```

Inner loop t = 1 to k (vector dot product loop)
(for a given i, j produces one element C(i, j)



Outer loop i = 1 to m

Second loop j = 1 to n

Vector dot product:
Row i of A x Column j of B

For one iteration of outer loop (on i) and second loop (on j) inner loop (t = 1 to k) produces one element of C, C(i, j)

$$C(i, j) = \sum_{t=1}^{k} a(i,t) \times b(t, j)$$

Inner loop (one element of C, C(i, j) produced)

Vectorize inner loop "t"?

Straightforward Solution

Produce Partial Product Terms (vectorized)

- Vectorize most inner loop t (dot product) ?
  - MULV.D V1, V2, V3 — Produce partial product terms
- Must sum of all the elements of a vector to produce dot product besides grabbing one element at a time from a vector register and putting it in the scalar unit?
- e.g., shift all elements left 32 elements or collapse into a compact vector all elements not masked
- In T0, the vector extract instruction, vext.v. This shifts elements within a vector
- Called a "reduction"

Accumulate Partial Product Terms (Not vectorized)

$$C(i,j) = \sum_{t=1}^{k} a(i,t) \times b(t,j)$$

Assuming k= 32

Vectorized: Partial product terms using MULV.D V1, V2, V3

A More Optimal Vector Matrix Multiplication Solution

- **You don't need to do reductions for vector matrix multiplication.**
- **You can calculate multiple independent sums within one vector register** / Instead on most inner loop t
- **You can vectorize the j loop to perform 32 dot-products at the same time** **Or MVL**
- **Or you can think of each 32 Virtual Processor doing one of the dot products each.** **Or MVL**
- **(Assume Maximum Vector Length MVL = 32 and n is a multiple of MVL)**
- **Shown in "vector" C source code, but can create the assembly vector instructions from it.**

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
 for (j=1; j<n; j+=32)/* Step j 32 at a time. */
    {
        sum[0:31] = 0; /* Initialize a vector
                          register to zeros. */
        for (t=1; t<k; t++)
        {
         a_scalar = a[i][t]; /* Get scalar from
                                a matrix. */
         b_vector[0:31] = b[t][j:j+31];
                                    /* Get vector from
                                       b matrix. */
        prod[0:31] = b_vector[0:31]*a_scalar;
          /* Do a vector-scalar multiply. */

          /* Vector-vector add into results. */
        sum[0:31] += prod[0:31];
        }

        /* Unit-stride store of vector of
   results. */
        c[i][j:j+31] = sum[0:31];
    }
}
```

MVL

MVL

Each iteration of j Loop produces MVL result elements (here MVL =32)

Vectorize j Loop (how many Iterations?)

Vector Scalar Multiply MULVS

Vector Add ADDV

Work on MVL elements (here MVL = 32)

32 = MVL elements done (one j loop iteration)

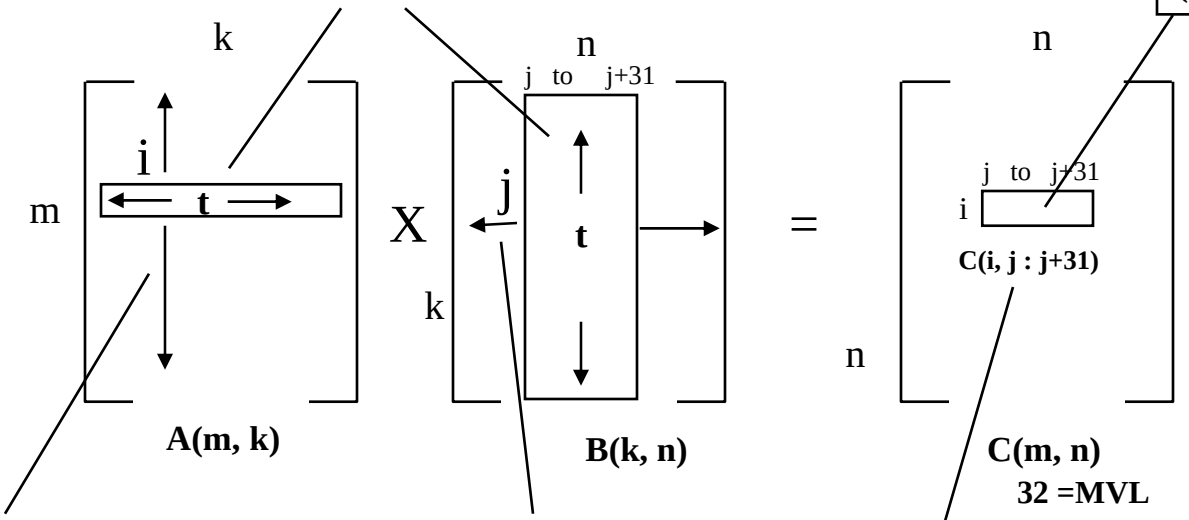
Optimized Vector Solution

Here we assume MVL = 32

Optimal Vector Matrix Multiplication

Inner loop t = 1 to k (vector dot product loop for MVL =32 elements)
(for a given i, j produces a 32-element vector C(i, j : j+31)

Each iteration of j Loop produces MVL result elements (here MVL =32)



Outer loop i = 1 to m
Not vectorized

Second loop j = 1 to n/32
(vectorized in steps of 32)

For one iteration of outer loop (on i) and vectorized second loop (on j) inner loop (t = 1 to k) produces 32 elements of C, C(i, j : j+31)

ADDV **MULVS**

$$C(i, j : j+31) = \sum_{t=1}^{k} a(i,t) \times b(t, j : j+31)$$

Inner loop (32 element vector of C produced)

Assume MVL =32
and n multiple of 32 (no odd size vector)

Common Vector Performance Metrics

For a given benchmark or program running on a given vector machine:

- **R_∞: MFLOPS rate on an infinite-length vector for this benchmark**
  - **Vector "speed of light" or peak vector performance.** For the given benchmark
  - **Real problems do not have unlimited vector lengths, and the effective start-up penalties encountered in real problems will be larger**
  - **(R_n is the MFLOPS rate for a vector of length n)**
- **$N_{1/2}$: The vector length needed to reach one-half of R_∞** Half power point?
  - **a good measure of the impact of start-up + other overheads**
- **N_V: The vector length needed to make vector mode performance equal to scalar mode**
  - **Break-even vector length, i.e:** **Break Even Vector Length N_V**
    - **For vector length = Nv**
      - **Vector performance = Scalar performance**
    - **For Vector length > Nv**
      - **Vector performance > Scalar performance**
  - **Measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit, etc.**

The Peak Performance R_∞ of VMIPS for DAXPY

With vector chaining and one LSU

$T_{start} = 12 + 7 + 12 + 6 + 12 = 49$

See slide 48

Using MVL = 64, $T_{loop} = 15$, $T_{start} = 49$, and $T_{chime} = 3$ in the performance equation, and assuming that n is not an exact multiple of 64, the time for an n-element operation is

Loop Overhead = 15 **Startup Time = 49** **Number of Convoys = m =3**

$$T_n = \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n$$

$$\leq (n + 64) + 3n$$

$$= 4n + 64$$

Number of elements n (i.e vector length)

From vector loop (strip mining) cycles equation (slide 38)

See slide 48

The sustained rate is actually over 4 clock cycles per iteration, rather than the theoretical rate of 3 chimes, which ignores overhead. The major part of the difference is the cost of the start-up overhead for each block of 64 elements (49 cycles versus 15 for the loop overhead).

We can now compute R_∞ for a 500 MHz clock as

64x2

$$R_\infty = \lim_{n \to \infty} \left(\frac{\text{Operations per iteration} \times \text{Clock rate}}{\text{Clock cycles per iteration}} \right)$$

The numerator is independent of n, hence

2 FP operations per result element

$$R_\infty = \frac{\text{Operations per iteration} \times \text{Clock rate}}{\lim_{n \to \infty} (\text{Clock cycles per iteration})}$$

$$\lim_{n \to \infty} (\text{Clock cycles per iteration}) = \lim_{n \to \infty} \left(\frac{T_n}{n} \right) = \lim_{n \to \infty} \left(\frac{4n + 64}{n} \right) = 4$$

Cycles per result element

2 FP operations every 4 cycles

$$R_\infty = \frac{2 \times 500 \text{ MHz}}{4} = 250 \text{ MFLOPS}$$

One LSU thus needs 3 convoys T_{chime} = m= 3

Sustained Performance of VMIPS on the Linpack Benchmark

The Linpack benchmark is a Gaussian elimination on a 100×100 matrix. Thus, the vector element lengths range from 99 down to 1. A vector of length k is used k times. Thus, the average vector length is given by

Note: DAXPY is the core computation of Linpack with vector length 99 down to 1

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Average Vector Length

Now we can obtain an accurate estimate of the performance of DAXPY using a vector length of 66.

$$T_n = \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n$$

From last slide

2 x 66 = 132 FP operations in 326 cycles

$$T_{66} = 2 \times (15 + 49) + 66 \times 3 = 128 + 198 = 326 \text{ cycles}$$

$$R_{66} = \frac{2 \times 66 \times 500}{326} \text{ MFLOPS} = 202 \text{ MFLOPS}$$

R_{66} / = 202 MFLOPS vs. R_∞ = 250 MFLOPS

$R_{66} / R_\infty = 202/250 = 0.808 = 80.8\ \%$

Larger versions of Linpack 1000x1000 or more

VMIPS DAXPY $N_{1/2}$

Example What is $N_{1/2}$ for just the inner loop of DAXPY for VMIPS with a 500 MHz clock?

250 MFLOPS

$N_{1/2}$ = vector length needed to reach half of R_∞

Answer Using R_∞ as the peak rate, we want to know the vector length that will achieve about 125 MFLOPS. We start with the formula for MFLOPS assuming that the measurement is made for $N_{1/2}$ elements:

$$\text{MFLOPS} = \frac{\text{FLOPS executed in } N_{1/2} \text{ iterations}}{\text{Clock cycles to execute } N_{1/2} \text{ iterations}} \times \frac{\text{Clock cycles}}{\text{Second}} \times 10^{-6}$$

$$125 = \frac{2 \times N_{1/2}}{T_N} \times 500$$

Simplifying this and then assuming $N_{1/2} \leq 64$, so that $T_{n \leq 64} = 1 \times 64 + 3 \times n$, yields

i.e. 125 MFLOPS = 8 cycles per result element (2 FP operations)

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$1 \times 64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

Thus: $N_{1/2} = 13$

So $N_{1/2} = 13$; that is, a vector of length 13 gives approximately one-half the peak performance for the DAXPY loop on VMIPS.

VMIPS DAXPY N_v — Break-even Vector Length

N_v = Vector length needed to make vector mode performance equal to scalar performance or break-even vector length (For n > N_v vector mode is faster)

Example What is the vector length, N_v, such that the vector operation runs faster than the scalar?

One element

Answer Again, we know that $N_v < 64$. The time to do one iteration in scalar mode can be estimated as 10 + 12 + 12 + 7 + 6 +12 = 59 clocks, where 10 is the estimate of the loop overhead, known to be somewhat less than the strip-mining loop overhead. In the last problem, we showed that this vector loop runs in vector mode in time $T_{n \le 64} = 64 + 3 \times n$ clock cycles. Therefore,

Scalar Processor: 59 cycles per result (element)

$$64 + 3N_v = 59N_v$$

$$N_v = \left\lceil \frac{64}{56} \right\rceil$$

$$N_v = 2$$

For the DAXPY loop, vector mode is faster than scalar as long as the vector has at least two elements. This number is surprisingly small,

i.e for vector length = VL = n > 2 vector is faster than scalar mode

Is MVL = 4 potentially useful?

DAXPY Performance on an Enhanced VMIPS

Vector Chained DAXPY With 3 LSUs

See slide 49

Here 3 LSUs

DAXPY, like many vector problems, is memory limited. Consequently, performance could be improved by adding more memory access pipelines. This is the major architectural difference between the Cray X-MP (and later processors) and the Cray-1. The Cray X-MP has three memory pipelines, compared with the Cray-1's single memory pipeline, and the X-MP has more flexible chaining. How does this affect performance?

See slide 49

For chained DAXPY with 3 LSUs number of convoys = m = T_{chime} = 1 (as opposed to 3 with one LSU)

Example What would be the value of T_{66} for DAXPY on VMIPS if we added two more memory pipelines?

3 LSUs total

Answer With three memory pipelines all the instructions fit in one convoy and take one chime. The start-up overheads are the same, so

m = 1 convoy Not 3

$$T_{66} = \left\lceil \frac{66}{64} \right\rceil \times (T_{loop} + T_{start}) + 66 \times T_{chime}$$

m=1

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

194 cycles vs 326 with one LSU

With three memory pipelines, we have reduced the clock cycle count for sustained performance from 326 to 194, a factor of 1.7. Note the effect of Amdahl's Law: We improved the theoretical peak rate as measured by the number of chimes by a factor of 3, but only achieved an overall improvement of a factor of 1.7 in sustained performance.

Speedup = 1.7 (going from m=3 to m=1) Not 3 (Why?)

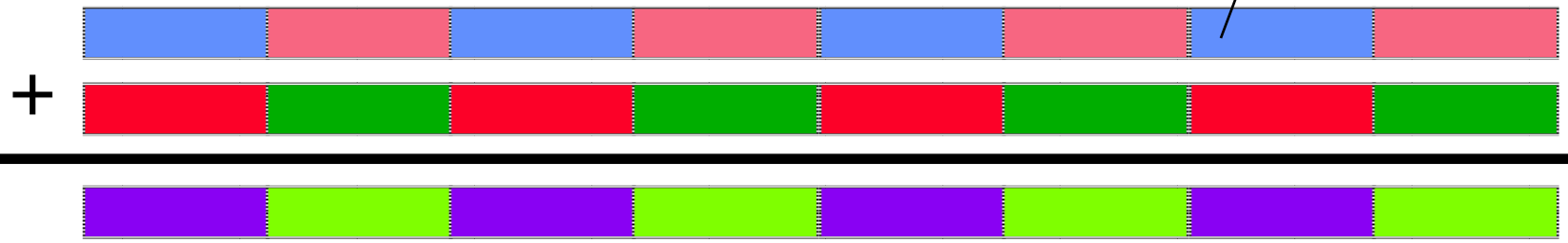
SIMD/Vector or Multimedia Extensions to Scalar ISAs

- **Vector or Multimedia ISA Extensions: Limited vector instructions added to scalar RISC/CISC ISAs with MVL = 2-8**

Why? Improved exploitation of data parallelism in scalar ISAs/processors

- **Example: Intel MMX: 57 new x86 instructions (1st since 386)**
  - **similar to Intel 860, Mot. 88110, HP PA-71000LC, UltraSPARC**
  - **3 integer vector element types: 8 8-bit (MVL =8), 4 16-bit (MVL =4) , 2 32-bit (MVL =2) in packed in 64 bit registers**
    - **reuse 8 FP registers (FP and MMX cannot mix)**

- short vector: load, add, store 8 8-bit operands



  - **Claim: overall speedup 1.5 to 2X for multimedia applications (2D/3D graphics, audio, video, speech …)**

- **Intel SSE (Streaming SIMD Extensions) adds support for FP with MVL =2 to MMX**
- **SSE2 Adds support of FP with MVL = 4 (4 single FP in 128 bit registers), 2 double FP MVL = 2, to SSE**

Major Issue: Efficiently meeting the increased data memory bandwidth requirements of such instructions

MMX Instructions

- Move 32b, 64b
- Add, Subtract in parallel: 8 8b, 4 16b, 2 32b
  - opt. signed/unsigned saturate (set to max) if overflow
- Shifts (sll,srl, sra), And, And Not, Or, Xor
  in parallel: 8 8b, 4 16b, 2 32b
- Multiply, Multiply-Add in parallel: 4 16b
- Compare = , > in parallel: 8 8b, 4 16b, 2 32b
  - sets field to 0s (false) or 1s (true); removes branches
- Pack/Unpack
  - Convert 32b<–> 16b, 16b <–> 8b
  - Pack saturates (set to max) if number is too large

Vector Processing Advantages

- Easy to get high performance; N operations:
  - are independent
  - use same functional unit (similar operations)
  - access disjoint registers
  - access registers in same order as previous instructions
  - access contiguous memory words or known patterns (normally)
  - can exploit large memory bandwidth
  - hide memory latency (and any other latency)

Data parallel within a vector instruction

- Scalable get higher performance as more HW resources available (e.g. more vector lanes/FUs)
- Compact: Describe N operations with 1 short instruction (v. VLIW)
- Predictable (real-time) performance vs. statistical performance (cache)
- Multimedia ready: choose N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
- Mature, developed vectorizing compiler technology

Vector Processing Pitfalls

- **Pitfall: Concentrating on peak performance and ignoring start-up/strip mining/other overheads: N_V (length faster than scalar) > 100!**
- **Pitfall: Increasing vector performance, without comparable increases in scalar (strip mining overhead ..) performance (Amdahl's Law).** As shown in example
- **Pitfall: High-cost of traditional vector processor implementations (Supercomputers).**
- **Pitfall: Adding vector instruction support without providing the needed memory bandwidth/low latency**
  - **MMX? Other vector media extensions, SSE, SSE2, SSE3..?**
- **One More Vector Disadvantage: Out of fashion in high performance computing due to rise of lower-cost commodity supercomputing/clusters utilizing multiple off-the-shelf GPPs.**

Vector Processing & VLSI: Vector Intelligent RAM (VIRAM)

Effort towards a full-vector processor on a chip:

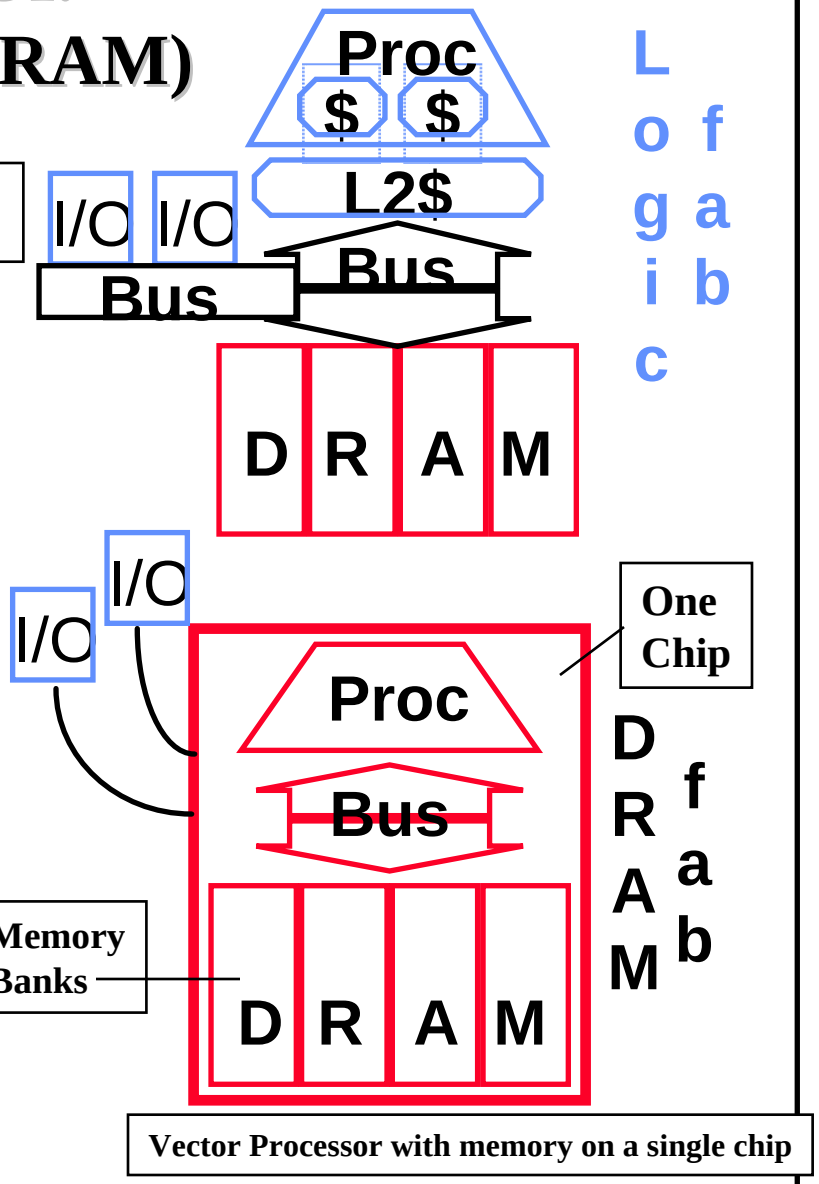
Capitalize on increasing VLSI chip density

How to meet vector processing high memory bandwidth and low latency requirements?

Full Vector Microprocessor & DRAM on a single chip:

Why?

- On-chip memory latency 5-10X lower, bandwidth 50-100X higher
- Improve energy efficiency 2X-4X (no off-chip bus)
- Serial I/O 5-10X v. buses
- Smaller board area/volume
- Adjustable memory size/width
- Much lower cost/power than traditional vector supercomputers



Vector Processor with memory on a single chip

VEC-2, VEC-3

Potential VIRAM Latency Reduction: 5 - 10X

- No parallel DRAMs, memory controller, bus to turn around, SIMM module, pins…
- New focus: Latency oriented DRAM?
  - Dominant delay = RC of the word lines
  - keep wire length short & block sizes small?
- 10-30 ns for 64b-256b IRAM "RAS/CAS"?
- AlphaSta. 600: 180 ns=128b, 270 ns= 512b
  Next generation (21264): 180 ns for 512b?

Now about 70 ns

Potential VIRAM Bandwidth Increase: 100X

- 1024 1Mbit modules(1Gb), each 256b wide
  - 20% @ 20 ns RAS/CAS = 320 GBytes/sec
- If cross bar switch delivers 1/3 to 2/3 of BW of 20% of modules
  ⇒ 100 - 200 GBytes/sec
- FYI: AlphaServer 8400 = 1.2 GBytes/sec (now ~ 6.4 GB/sec)
  - 75 MHz, 256-bit memory bus, 4 banks

VS.

Characterizing V-IRAM Cost/Performance

- **Low Cost - VMIPS vector processor + memory banks/interconnects integrated on one chip**
- **Small memory on-chip (25 - 100 MB)**
- **High vector performance (2 -16 GFLOPS)**
- **High multimedia performance (4 - 64 GOPS)**
- **Low latency main memory (15 - 30ns)**
- **High BW main memory (50 - 200 GB/sec)**
- **High BW I/O (0.5 - 2 GB/sec via N serial lines)**
  - **Integrated CPU/cache/memory with high memory BW ideal for fast serial I/O**

Cray 1
133 MFLOPS
Peak

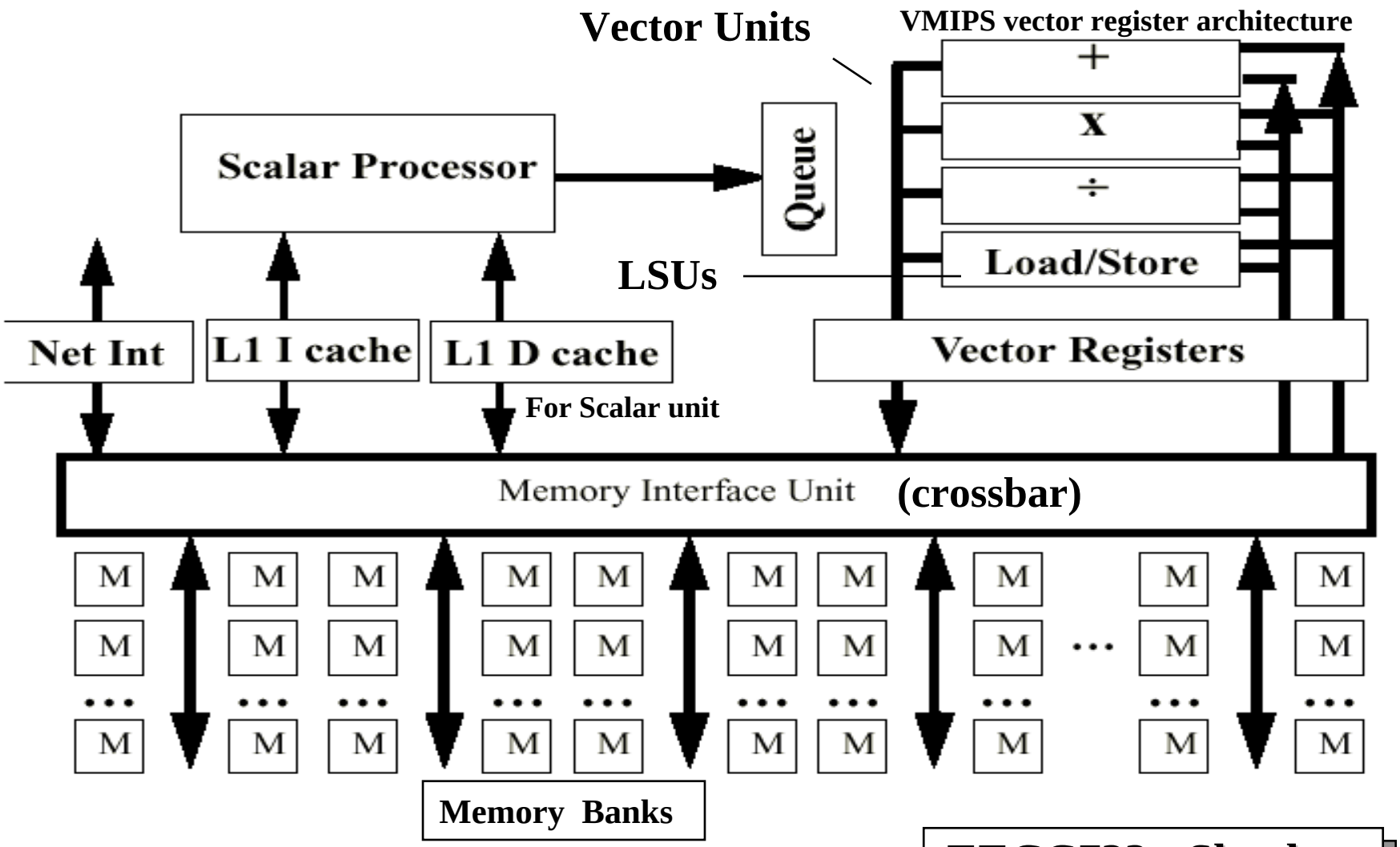
~ 5-10X Lower latency

~ 100X Higher Bandwidth

Estimated power ~ 2 watts

Vector IRAM Organization

VMIPS vector processor + memory banks/interconnects integrated on one chip



V-IRAM1 Instruction Set (VMIPS)

Scalar — **Standard scalar instruction set (e.g., ARM, MIPS)**

Vector IRAM (V-IRAM) ISA : VMIPS
(covered earlier)

MVL =64

Vector ALU

{ + | – | x | ÷ | & | \| | shl | shr } { s.int | u.int | s.fp | d.fp } { 8 | 16 | 32 | 64 } { .vv | .vs | .sv } { saturate | overflow } { masked | unmasked }

Vector Memory

{ load | store } { s.int | u.int } { 8 | 16 | 32 | 64 } { 8 | 16 | 32 | 64 } { unit | constant | indexed } { masked | unmasked }

Vector Registers

32 x 32 x 64b (or 32 x 64 x 32b or 32 x 128 x 16b)
+ 32 x128 x 1b flag

Plus: **flag**, **convert**, **DSP**, and **transfer** operations

Goal for Vector IRAM Generations

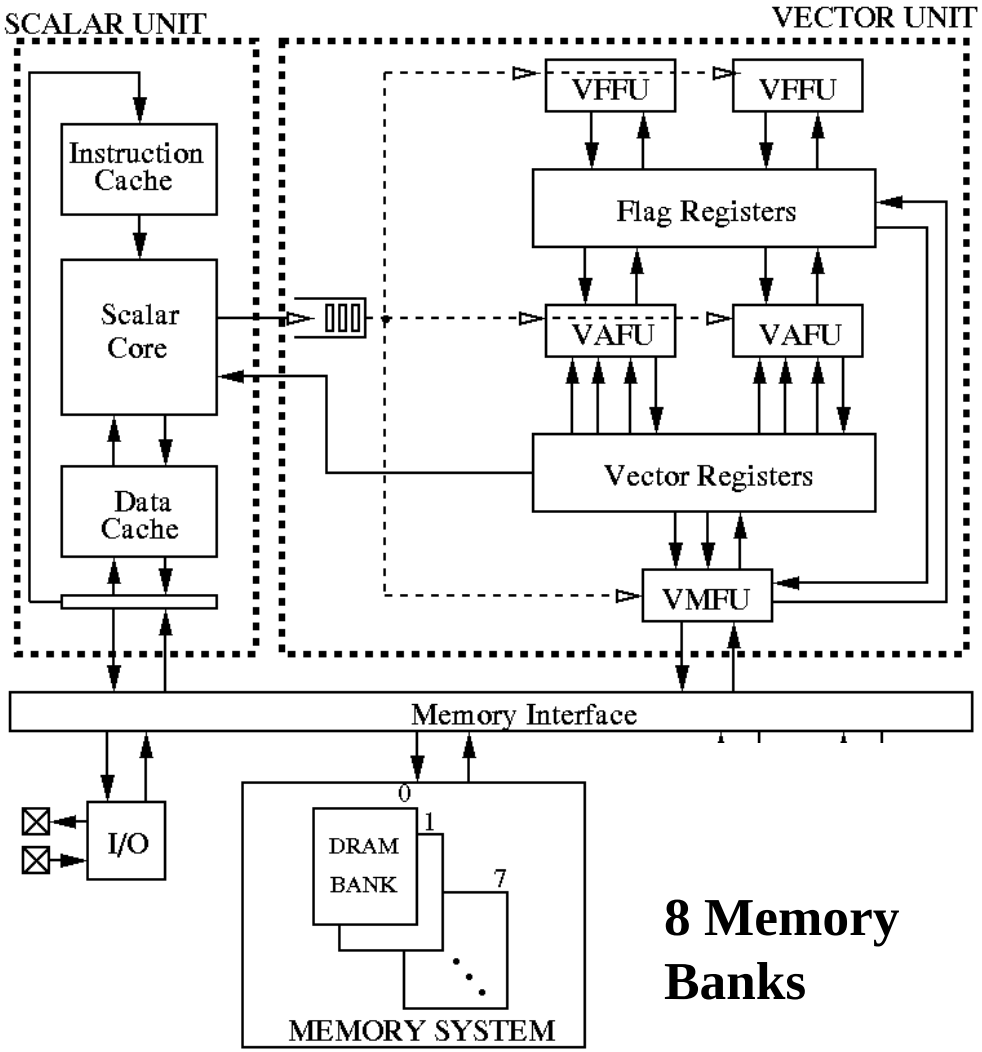
- V-IRAM-1 (-2000)
- 256 Mbit generation (0.20)
- Die size = 1.5X 256 Mb die
- 1.5 - 2.0 v logic, 2-10 watts
- 100 - 500 MHz
- 4 64-bit pipes/lanes
- 1-4 GFLOPS(64b)/6-16G (16b)
- 30 - 50 GB/sec Mem. BW
- 32 MB capacity + DRAM bus
- Several fast serial I/O
- 16 memory banks

- V-IRAM-2 (-2005???)
- 1 Gbit generation (0.13)
- Die size = 1.5X 1 Gb die
- 1.0 - 1.5 v logic, 2-10 watts
- 200 - 1000 MHz
- 8 64-bit pipes/lanes
- 2-16 GFLOPS/24-64G
- 100 - 200 GB/sec Mem. BW
- 128 MB cap. + DRAM bus
- Many fast serial I/O
- 32 memory banks

VIRAM-1 Microarchitecture

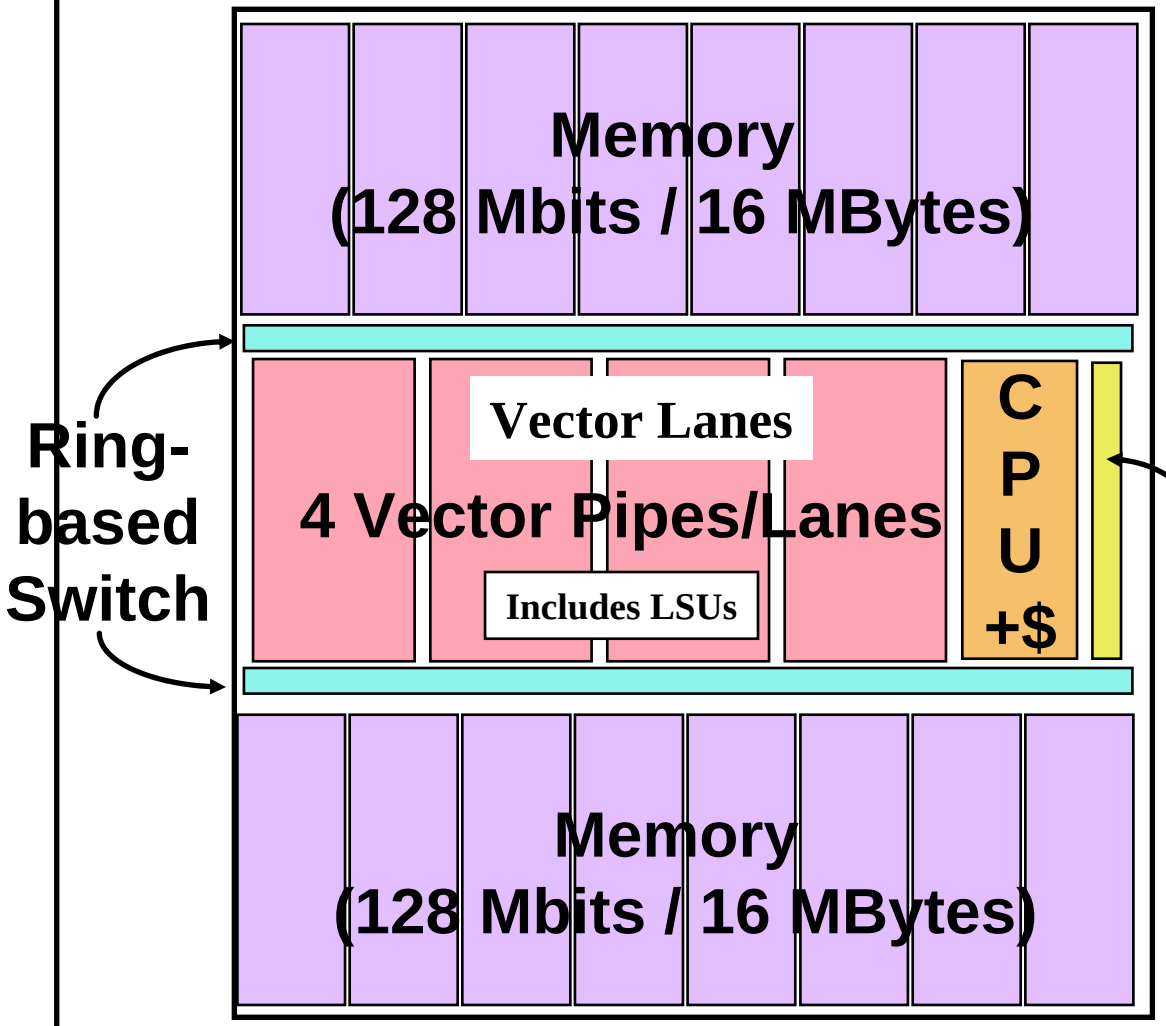
- 2 arithmetic units
  - both execute integer operations
  - one executes FP operations
  - 4 64-bit datapaths (lanes) per unit
- 2 flag processing units
  - for conditional execution and speculation support
- 1 load-store unit
  - optimized for strides 1,2,3, and 4
  - 4 addresses/cycle for indexed and strided operations
  - decoupled indexed and strided stores
- Memory system
  - 8 DRAM banks
  - 256-bit synchronous interface
  - 1 sub-bank per bank
  - 16 Mbytes total capacity
- Peak performance
  - 3.2 $GOPS_{64}$, 12.8 $GOPS_{16}$ (w. madd)
  - 1.6 $GOPS_{64}$, 6.4 $GOPS_{16}$ (wo. madd)
  - 0.8 $GFLOPS_{64}$, 1.6 $GFLOPS_{32}$
  - 6.4 Gbyte/s memory bandwidth comsumed by VU

VIRAM-1 block diagram



8 Memory Banks

VIRAM-1 Floorplan



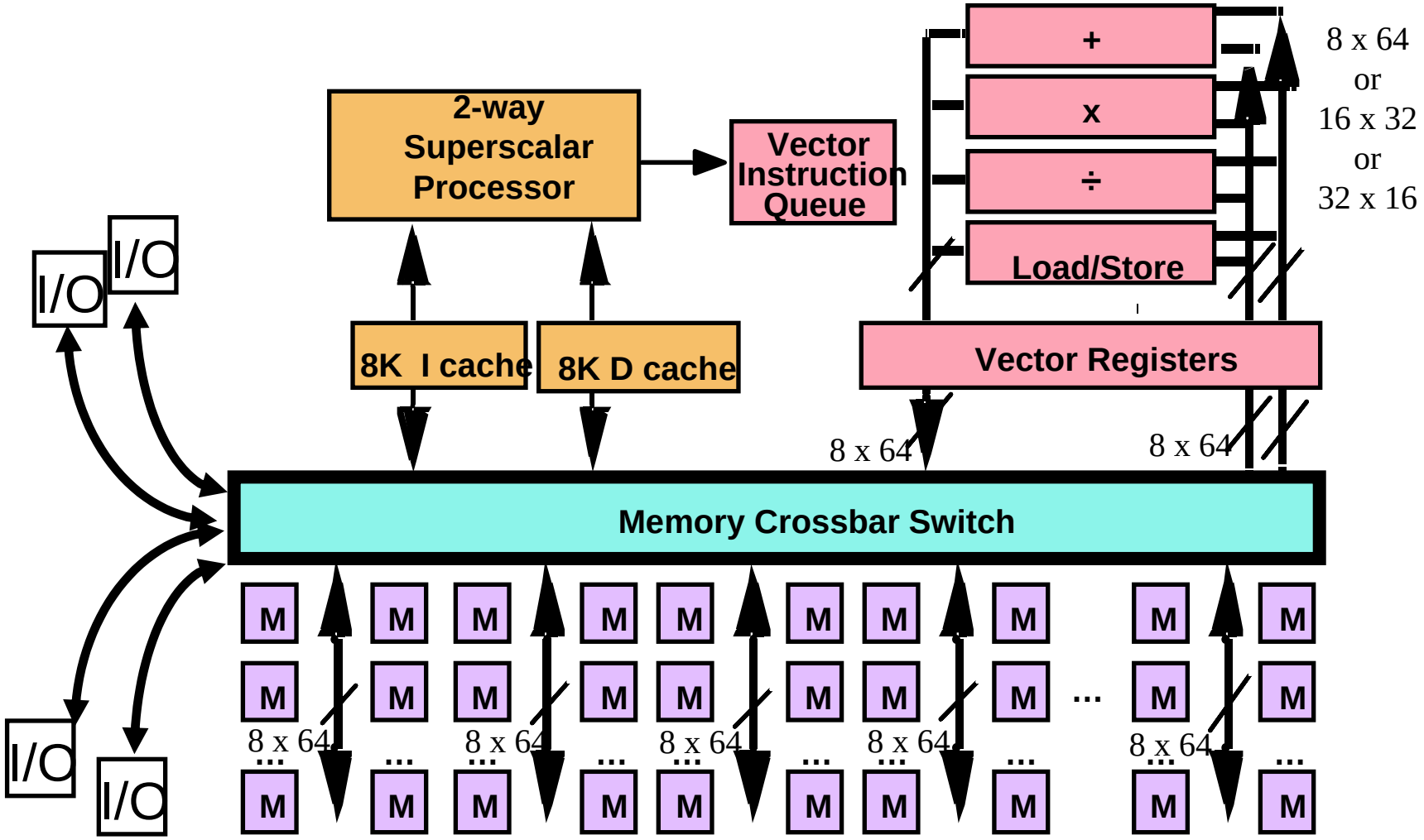
- 0.18 µm DRAM
  32 MB in 16 banks

banks x 256b, 128 subbanks

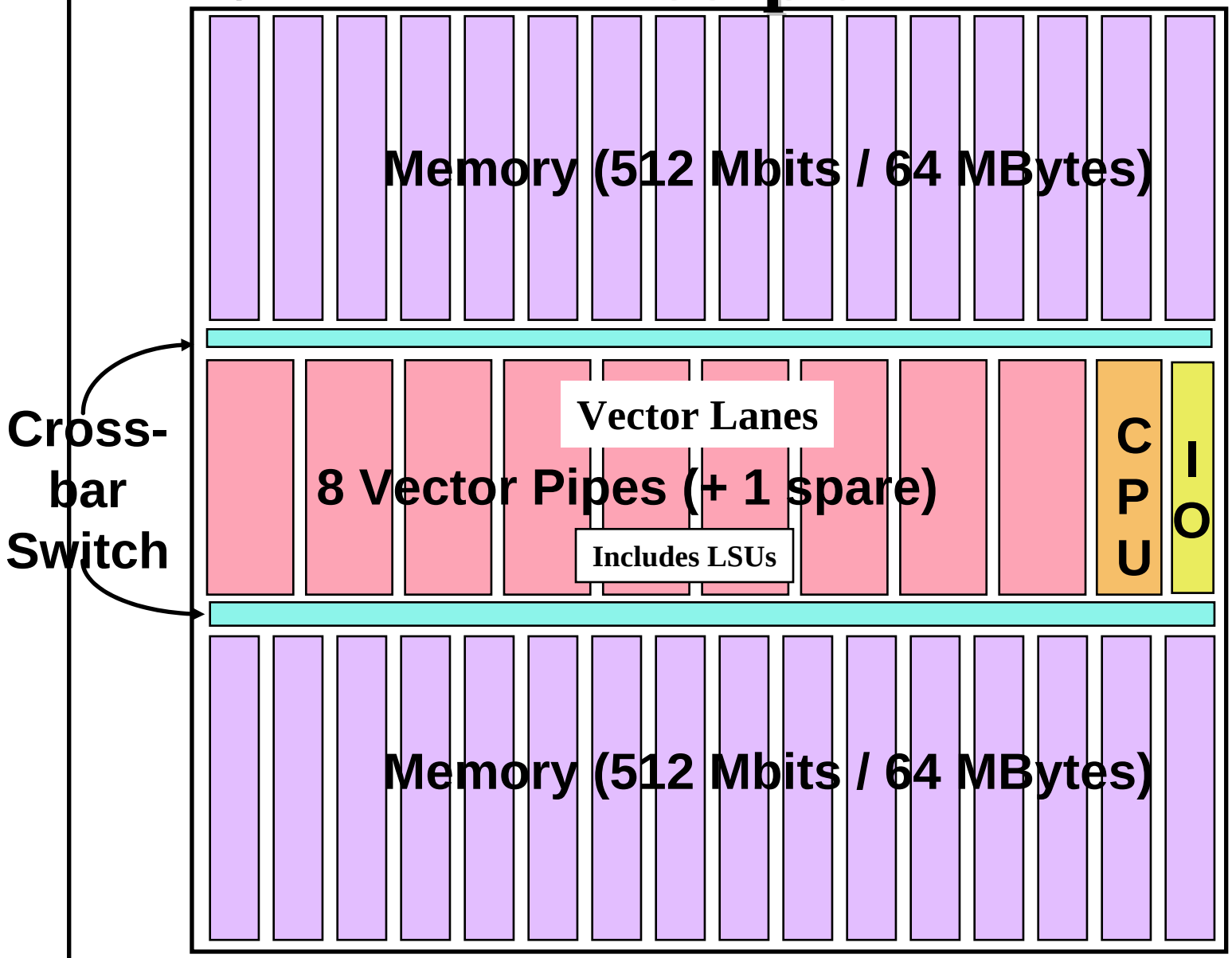
- 0.25 µm,
  5 Metal Logic
- - 200 MHz MIPS,
    16K I$, 16K D$
- - 4 200 MHz
  FP/int. vector units
- die: - 16x16 mm
- Transistors: - 270M
- power: -2 Watts
- Performance:

1-4 GFLOPS

V-IRAM-2: 0.13 µm, 1GHz
16 GFLOPS(64b)/64 GOPS(16b)/128MB

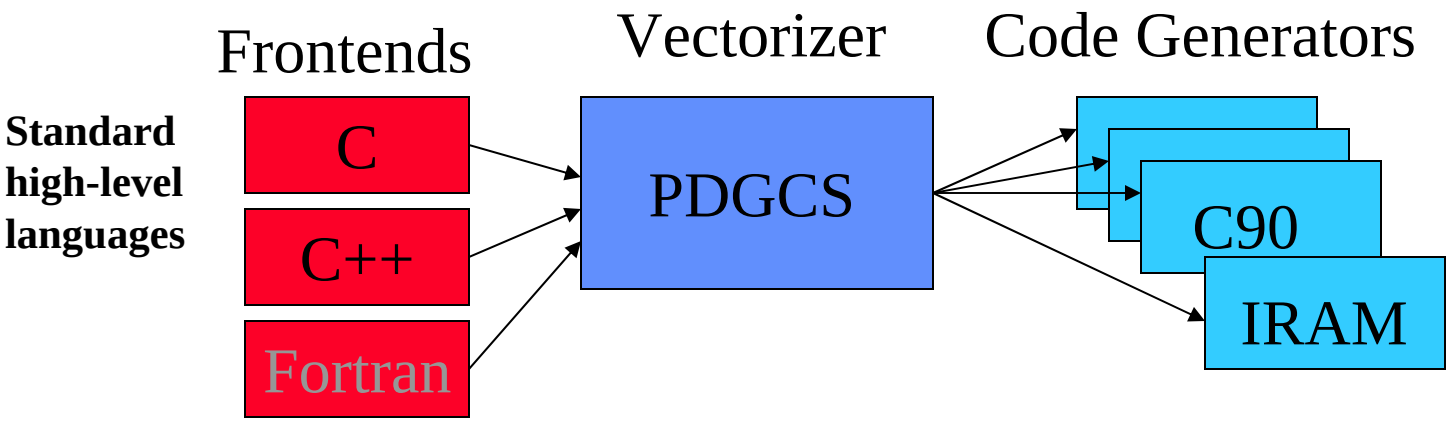


V-IRAM-2 Floorplan



- 0.13 µm, 1 Gbit DRAM
- >1B Xtors: 98% Memory, Xbar, Vector ⇒ regular design
- Spare Pipe & Memory ⇒ 90% die repairable
- Short signal distance ⇒ speed scales <0.1 µm

VIRAM Compiler



- **Retargeted Cray compiler to VMIPS**
- **Steps in compiler development**
  - **Build MIPS backend (done)**
  - **Build VIRAM backend for vectorized loops (done)**
  - **Instruction scheduling for VIRAM-1 (done)**
  - **Insertion of memory barriers (using Cray strategy, improving)**
  - **Additional optimizations (ongoing)**
  - **Feedback results to Cray, new version from Cray (ongoing)**