High Performance Programming
SIMD – Vector Processors

FISE2-INFO2

Based on "Lecture 14: SIMD Processing"

Onur Mutlu, 2015

Guillaume MULLER

April 11, 2021

Readings

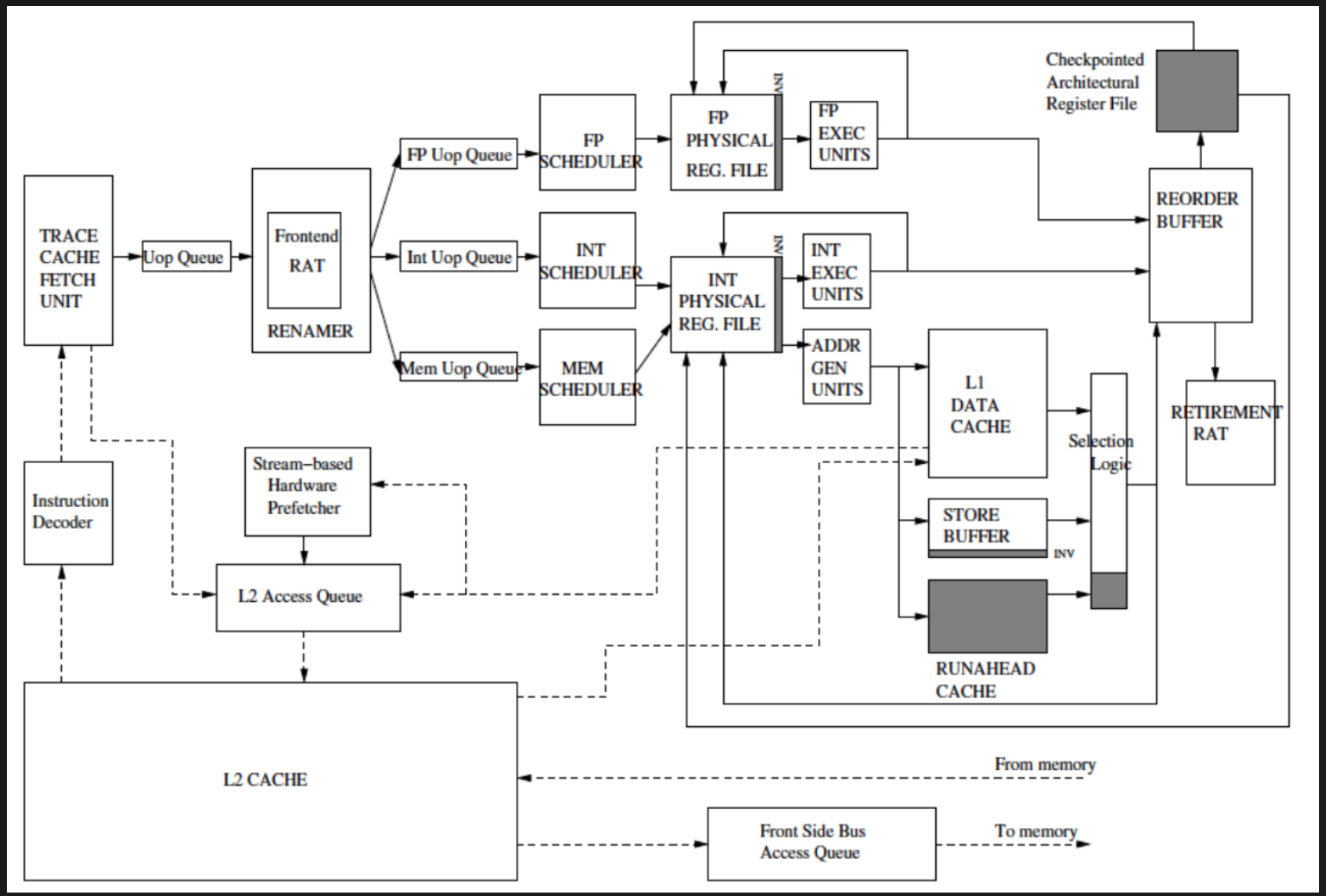
Paper1

- Lindholm *et al.*, "NVIDIA Tesla: A Unified Graphics and Computing Architecture", IEEE Micro 2008.

Paper2

- Fatahalian and Houston, "A Closer Look at GPUs", CACM 2008.

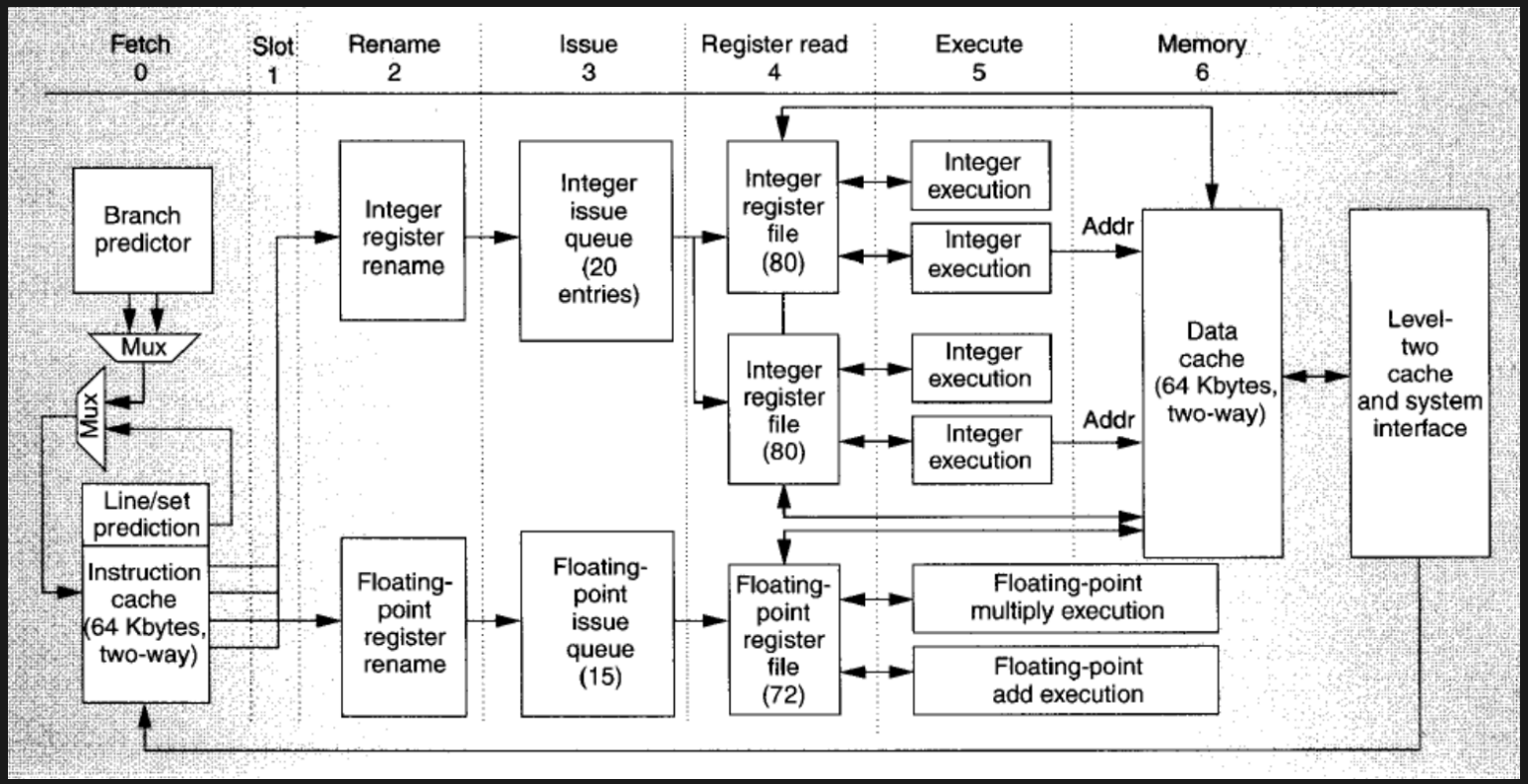
Reminder Intel Architecture



Reference

- Mutlu *et al.*, Runhead Execution, 2003

Reminder Alpha Architecture



Reference

- Kessler, "The Alpha 21264 Microprocessor", IEEE Micro, March-April 1999.

SIMD: Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers (1966)

- SISD: Single Instruction operates on Single Data

SIMD: Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers (1966)

- SISD: Single Instruction operates on Single Data
- SIMD: Single Instruction operates on Multiple Data
  - Array processor
  - Vector processor

SIMD: Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers (1966)

- SISD: Single Instruction operates on Single Data
- SIMD: Single Instruction operates on Multiple Data
  - Array processor
  - Vector processor
- MISD: Multiple Instructions operate on Single Data
  - (rare)

SIMD: Exploiting Regular (Data) Parallelism

Flynn's Taxonomy of Computers (1966)

- SISD: Single Instruction operates on Single Data
- SIMD: Single Instruction operates on Multiple Data
  - Array processor
  - Vector processor
- MISD: Multiple Instructions operate on Single Data
  - (rare)
- MIMD: Multiple Instructions operate on Multiple Data
  - Multi-Processor
  - Multi-Threaded

Data Parallelism

Concurrency Model

- Same operations different pieces of data
  - SIMD
  - E.g. dot product of 2 vectors
  - Instruction (operation) level

Data Parallelism

Concurrency Model

- Same operations different pieces of data
  - SIMD
  - E.g. dot product of 2 vectors
  - Instruction (operation) level

$\neq$ Threads

- "Control" parallelism

Data Parallelism

Concurrency Model

- Same operations different pieces of data
  - SIMD
  - E.g. dot product of 2 vectors
  - Instruction (operation) level

$\neq$ Threads

- "Control" parallelism

$\neq$ Data Flow

- (pipelines, branch prediction...)
- $\neq$ operations in parallel

SIMD Parallelism

Parallelism in Time

- Array processors
  - 1 Instruction on multiple data
  - at the same time
  - using different spaces

SIMD Parallelism

Parallelism in Time

- Array processors
  - 1 Instruction on multiple data
  - at the same time
  - using different spaces

Parallelism in Space

- Vector processor
  - 1 Instruction on multiple data
  - using the same space
  - in consecutive time steps

Array vs. Vector Processors

Code

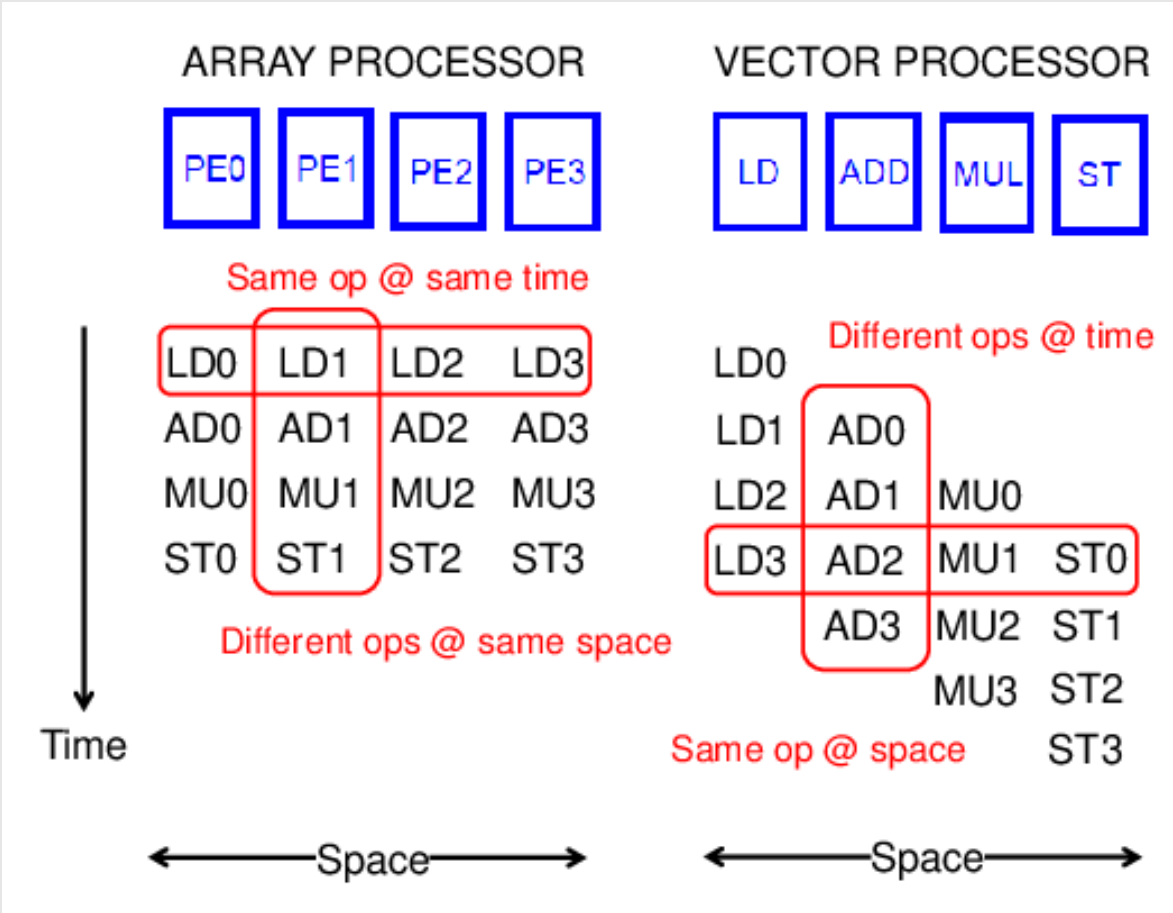
```
LD VR <- A[3:0]
ADD VR <- VR, 1
MUL VR <- VR, 2
ST  A[3:0] <- VR
```

Array vs. Vector Processors

Code

```
LD VR <- A[3:0]
ADD VR <- VR, 1
MUL VR <- VR, 2
ST  A[3:0] <- VR
```

Execution



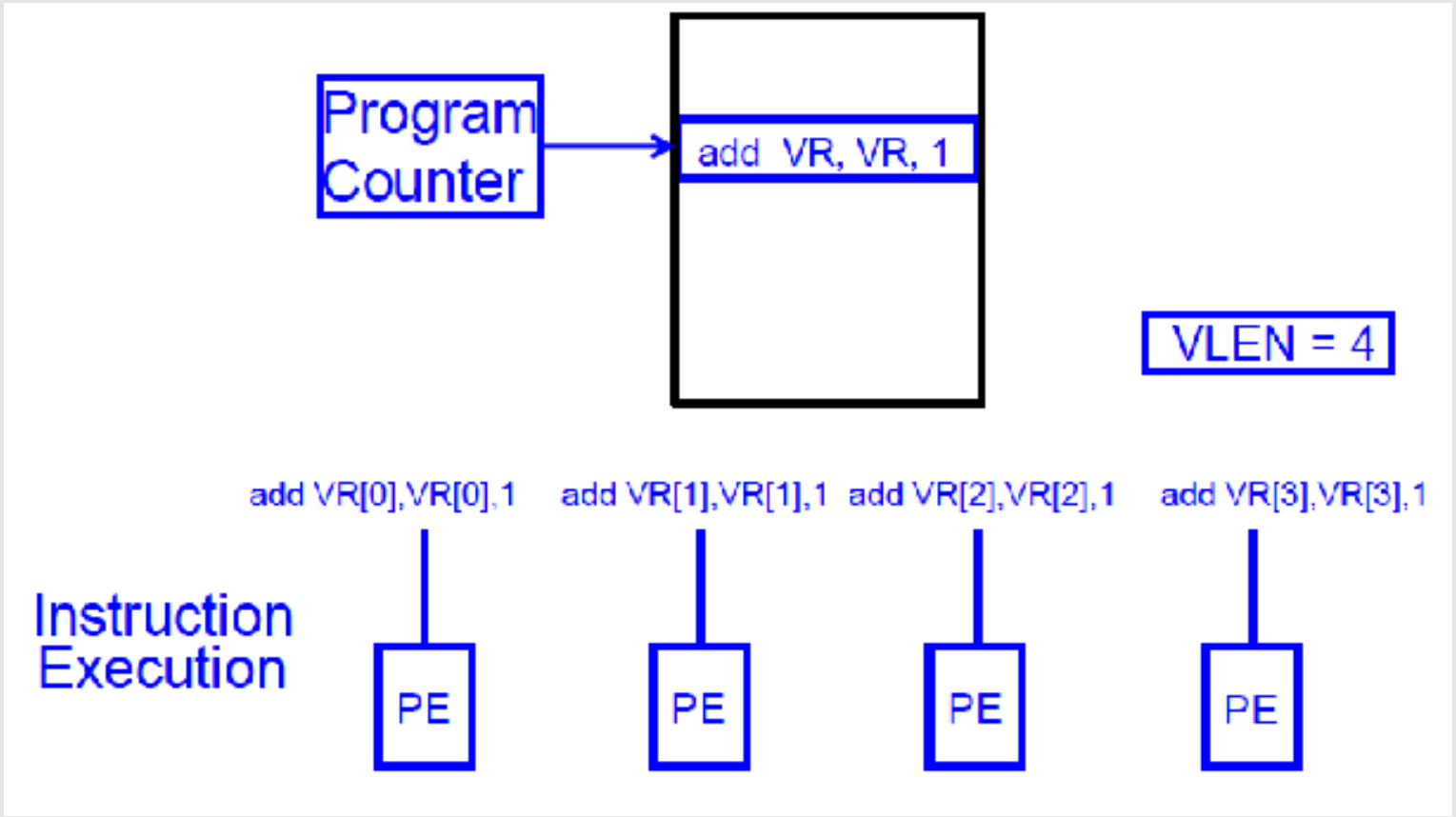
Array Processors

Array processors

- Single operation on multiple (different) data elements

Array processors

- Single operation on multiple (different) data elements



Vector Processors

Vector processor

- Operates on *vectors* not *scalars*
- Scalar = Single value
- Vector = 1D array of numbers
  - Used in many scientific apps

```
for (i = 0; i<=49; i++) {
  C[i] = (A[i] + B[i]) / 2
}
```

Vector Processors

Vector processor

- Operates on *vectors* not *scalars*
- Scalar = Single value
- Vector = 1D array of numbers
  - Used in many scientific apps

```
for (i = 0; i<=49; i++) {
  C[i] = (A[i] + B[i]) / 2
}
```

Basic requirements

- vector registers
- VLEN (vector *length* register)
- VSTR (vector *stride* register)
  - Stride: distance between two elements of a vector

Vector Processors

Vector Processors & Pipelines

- A vector instruction = each element in consecutive cycles
- Vector functional units are pipelined
- Each pipeline stage operates on a different data element

Vector Processors Advantages

- No intra-vector dependencies
  - $\Rightarrow$ no hardware interlocking
- No control flow within a vector
- Known stride
  - $\Rightarrow$ prefetching

Vector Processor Advantages

No dependencies within a vector

- Pipelining, parallelization work well
- Can have very deep pipelines, no dependencies!

Each instruction generates a lot of work

- Reduces instruction fetch bandwidth requirements

Highly regular memory access pattern

- Can interleave vector data elements across multiple memory banks for higher memory bandwidth (to tolerate memory bank access latency)
- Prefetching a vector is relatively easy

No need to explicitly code loops

- Fewer branches in the instruction sequence

Vector Processor Disadvantages

Regular Parallelism

- Works (only) if parallelism is regular (data/SIMD parallelism)
  - = Vector operations
- Very inefficient if parallelism is irregular
  - How about searching for a key in a linked list?

> *"To program a vector machine, the compiler or hand coder must make the data structure in the code fit nearly exactly the regular structure built into the hardware. That's hard to do in first place, and just as hard to change. One tweak, and the low-level code has to be rewritten by a very smart and dedicated programmer who knows the hardware and often the subtelties of the application area."*
>
> Fisher, 1983.

Vector Processor Limitations

Memory

- Memory (bandwidth) can become a bottleneck if:
  1. compute/memory operation balance is not maintained
  2. data is not mapped appropriately to memory banks

Vector Registers

Vector control registers

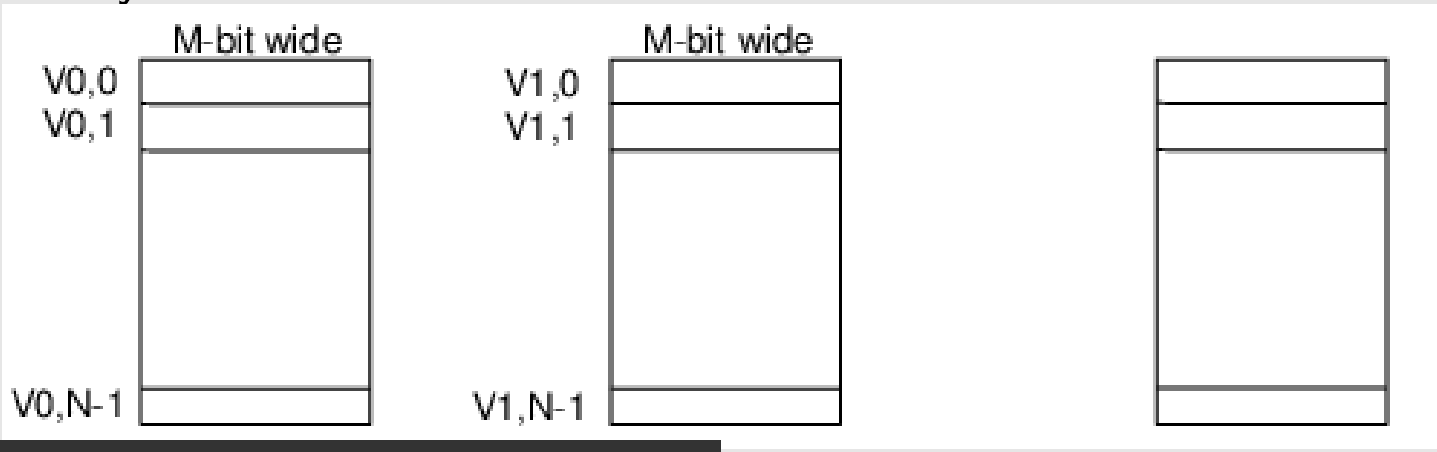
- VLEN, VSTR, VMASK

Vector Registers

- Each vector data register holds N $\times$ M-bit values
- Maximum VLEN can be N
  - Maximum number of elements stored in a vector register

Vector Mask Register (VMASK)

- Indicates which elements of vector to operate on
- Set by vector test instructions `VMASK[i] = (Vk[i] == 0)`



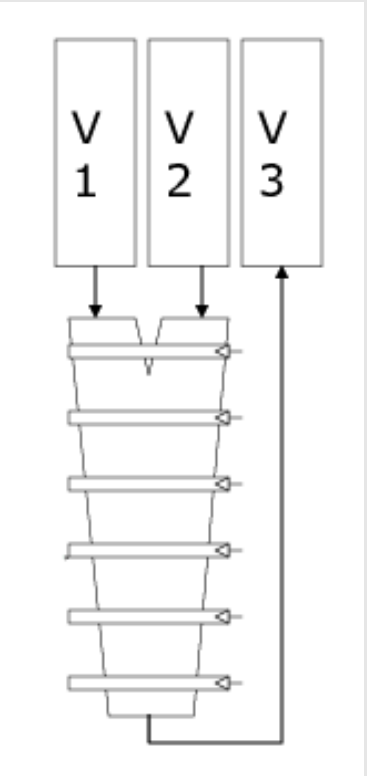
Vector Functional Units

Pipelines

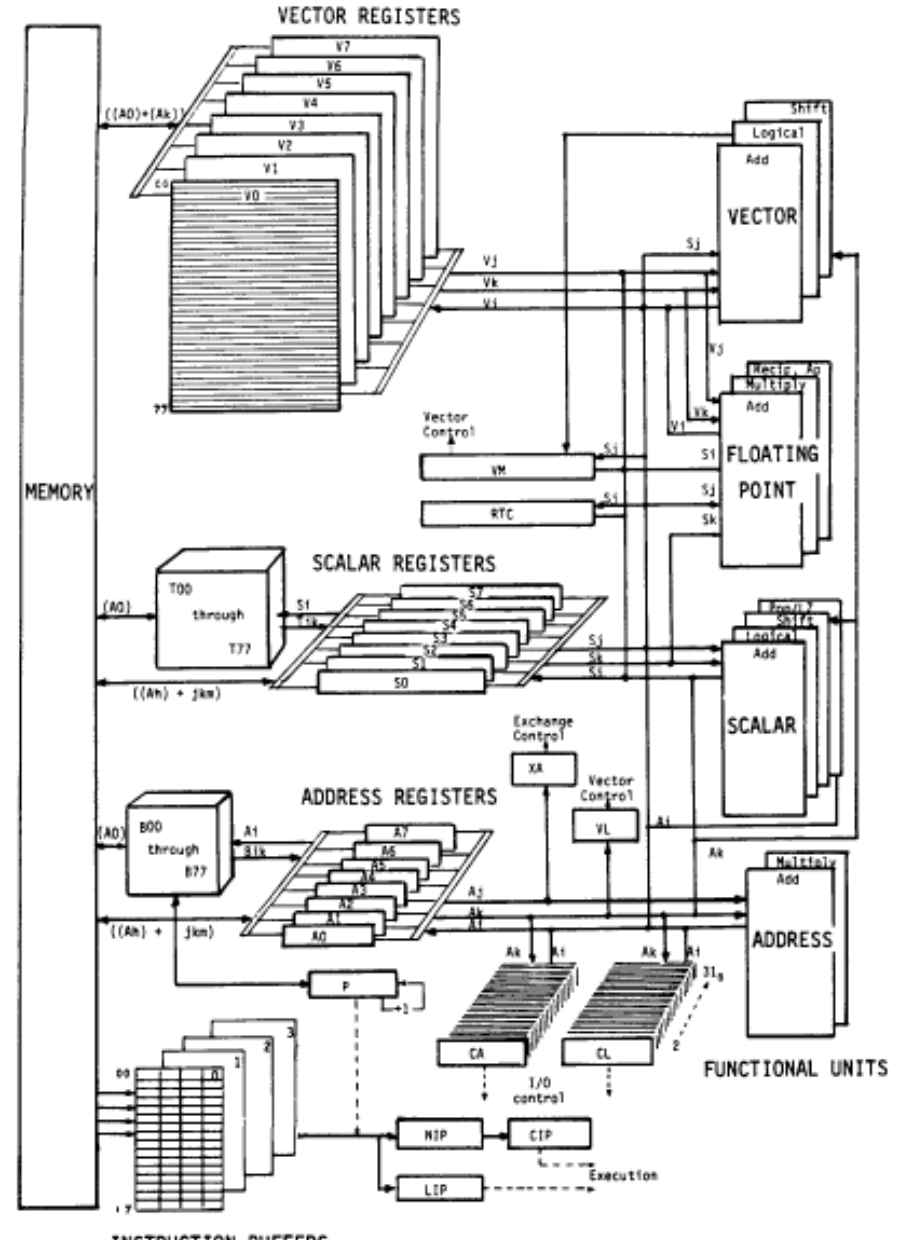
- Vector elements are independent
  - $\Rightarrow$ Deep pipeline control is easy
- Using deep pipelines
  - $\Rightarrow$ Fast clock cycle

Example

- 6 stages multiply
- `V1*V2 -> V3`



Vector Machine Organization (CRAY-1)



- CRAY-1, 1978
- scalar & vector regs.
- 8×64 elts / reg
- 64 bit / elt
- 16 memory banks
- 8×64b scalar re.
- 8×24b addr. reg.

Loading/Storing Vectors from/to Memory

- Requires loading/storing multiple elements
- Elements separated by a constant distance (stride)
  - Assume stride = 1 for now
- If we can start the load of one element per cycle
  - Elements can be loaded in consecutive cycles
  - $\Rightarrow$ Can sustain a throughput of 1 elt/cycle

Question

- How do we achieve this with a memory that takes more than 1 cycle to access?

Loading/Storing Vectors from/to Memory

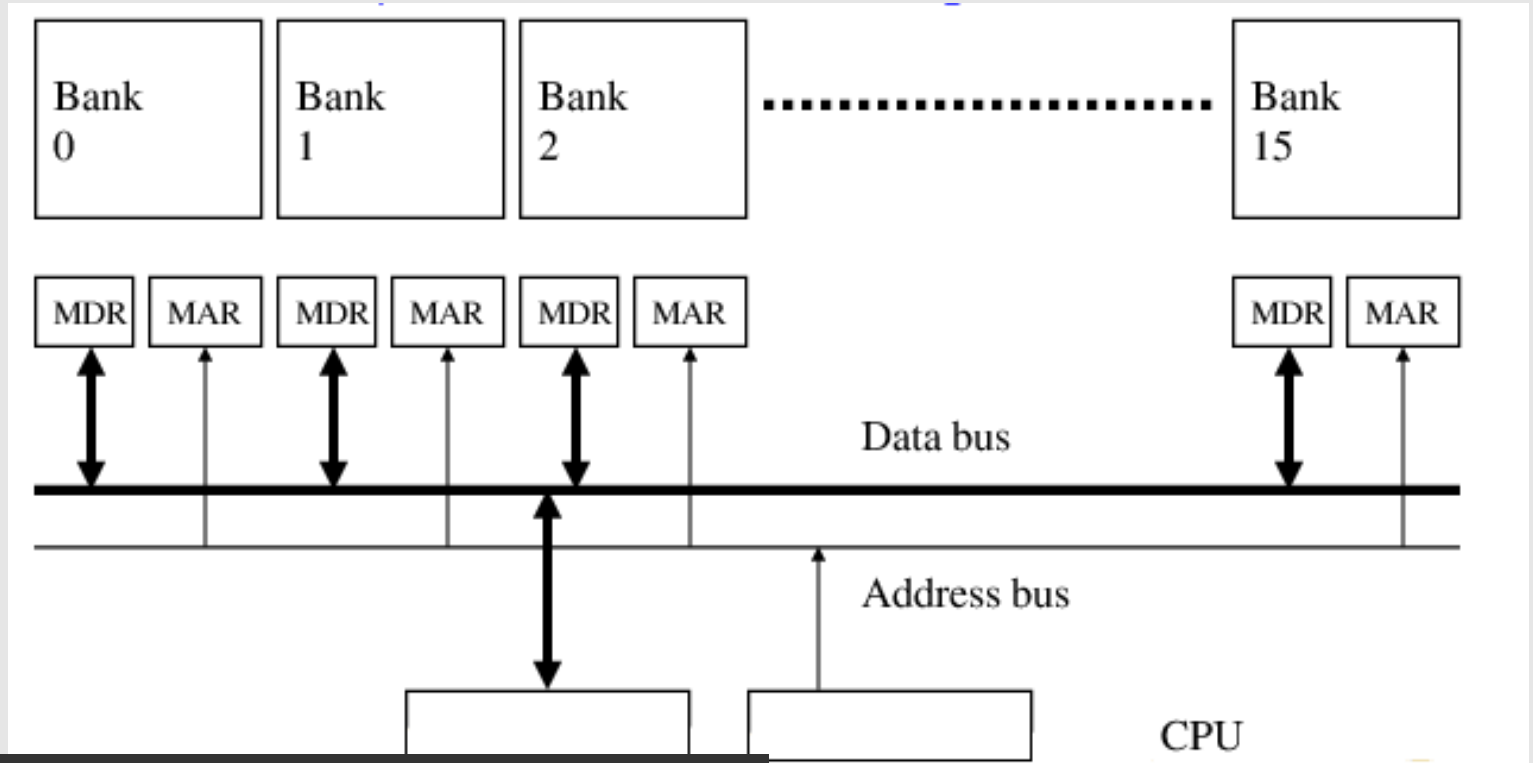
- Requires loading/storing multiple elements
- Elements separated by a constant distance (stride)
  - Assume stride = 1 for now
- If we can start the load of one element per cycle
  - Elements can be loaded in consecutive cycles
  - $\Rightarrow$ Can sustain a throughput of 1 elt/cycle

Question

- How do we achieve this with a memory that takes more than 1 cycle to access?
- $\Rightarrow$ Bank the memory
- $\Rightarrow$ Interleave elements across banks

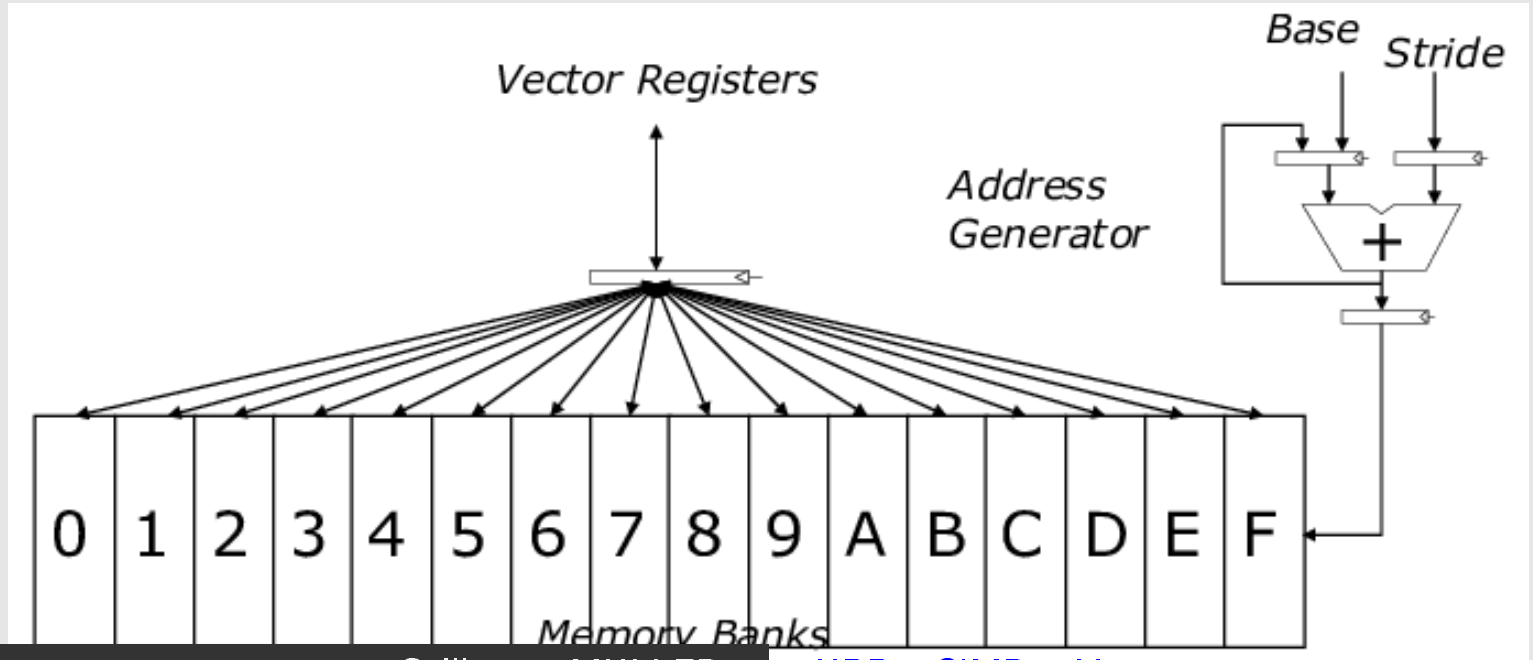
Memory Banking

- Memory is divided into independent banks
- But that share address and data buses
- Can start and complete 1 bank access/cycle
- Can sustain N parallel accesses if all N go to different banks



Vector Memory System

- Next address = Previous address + Stride
- If
  - stride = 1 &
  - consecutive elements interleaved across banks &
  - number of banks >= bank latency
- then can sustain 1 element/cycle



Scalar Code Example

C

```
for (i = 0; i<=49; i++) {
  C[i] = (A[i] + B[i]) / 2
}
```

ASM (*number = latency*)

```
MOVI R0 = 50              ; 1
MOVA R1 = A               ; 1
MOVA R2 = B               ; 1
MOVA R3 = C               ; 1
X: LD R4 = MEM[R1++]      ; 11 ; auto-inc
LD R5 = MEM[R2++]         ; 11
ADD R6 = R4 + R5          ; 4
SHFR R7 = R6 >> 1         ; 1
ST MEM[R3++] = R7         ; 11
DECBNZ --R0, X            ; 2 ; decr + brch non-0
```

- Number of instructions: 304
- Execution time: ??

Scalar Code Execution Time (In Order)

1 memory bank

- First two loads in the loop cannot be pipelined
  - $\Rightarrow$ 2*11 cycles
- $4 + 50 * 40 = 2004$ cycles

Scalar Code Execution Time (In Order)

1 memory bank

- First two loads in the loop cannot be pipelined
  - $\Rightarrow$ 2*11 cycles
- $4 + 50 * 40 = 2004$ cycles

16 memory bank

- First two loads in the loop can be pipelined
- $4 + 50 * 30 = 1504$ cycles

Scalar Code Execution Time (In Order)

1 memory bank

- First two loads in the loop cannot be pipelined
  - $\Rightarrow$ 2*11 cycles
- $4 + 50 * 40 = 2004$ cycles

16 memory bank

- First two loads in the loop can be pipelined
- $4 + 50 * 30 = 1504$ cycles

Why 16 banks?

Scalar Code Execution Time (In Order)

1 memory bank

- First two loads in the loop cannot be pipelined
  - $\Rightarrow$ 2*11 cycles
- $4 + 50 * 40 = 2004$ cycles

16 memory bank

- First two loads in the loop can be pipelined
- $4 + 50 * 30 = 1504$ cycles

Why 16 banks?

- 11 cycle memory access latency
- 16 banks (>11 cycles)
  - Enough to overlap enough mem. op. to cover mem. latency

Vectorizable Loops

- Loop is vectorizable if each iteration independent of others

C

```
for (i = 0; i<=49; i++) {
  C[i] = (A[i] + B[i]) / 2
}
```

Vectorized ASM (*number = latency*)

```
MOVI VLEN = 50       ; 1
MOVI VSTR = 1        ; 1
VLD V0 = A           ; 11 + VLN-1
VLD V1 = B           ; 11 + VLN-1
VADD V2 = V0 + V1    ; 4  + VLN-1
VSHFR V3 = V2 >> 1   ; 1  + VLN-1
VST C = V3           ; 11 + VLN-1
```

- Number of instructions: 7
- Execution time: ??

Vectorized Code Performance – No Chaining

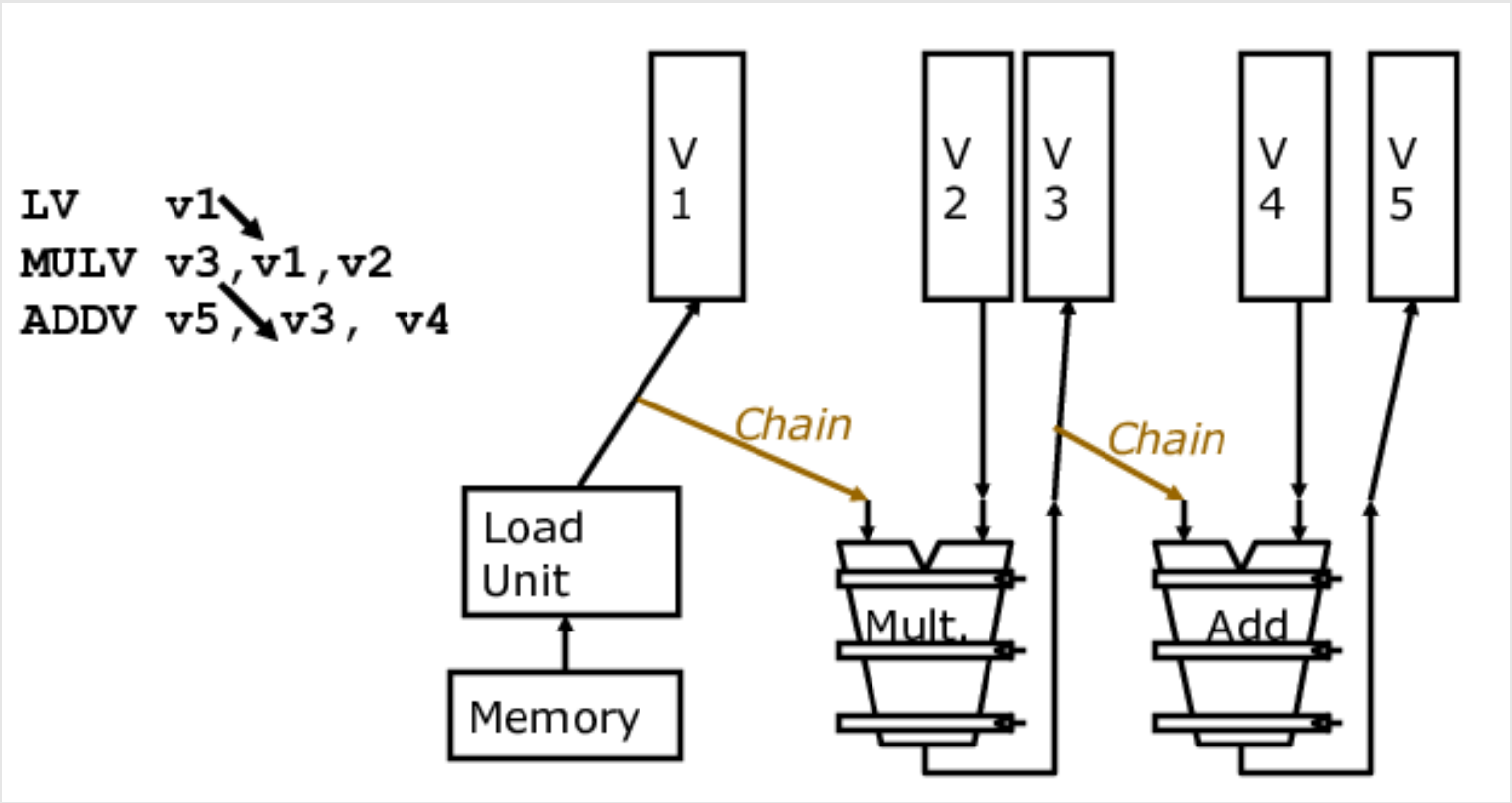
- Assume no chaining (no vector data forwarding)
  - Output of a vector functional unit cannot be used as the direct input of another
  - The entire vector register needs to be ready before any element of it can be used as part of another operation
- One memory port (one address generator)
- 16 memory banks (word-interleaved)

- $2 + 2 * (11 + 49) + 4 + 49 + 1 + 49 + 11 + 49 = 285$ cycles

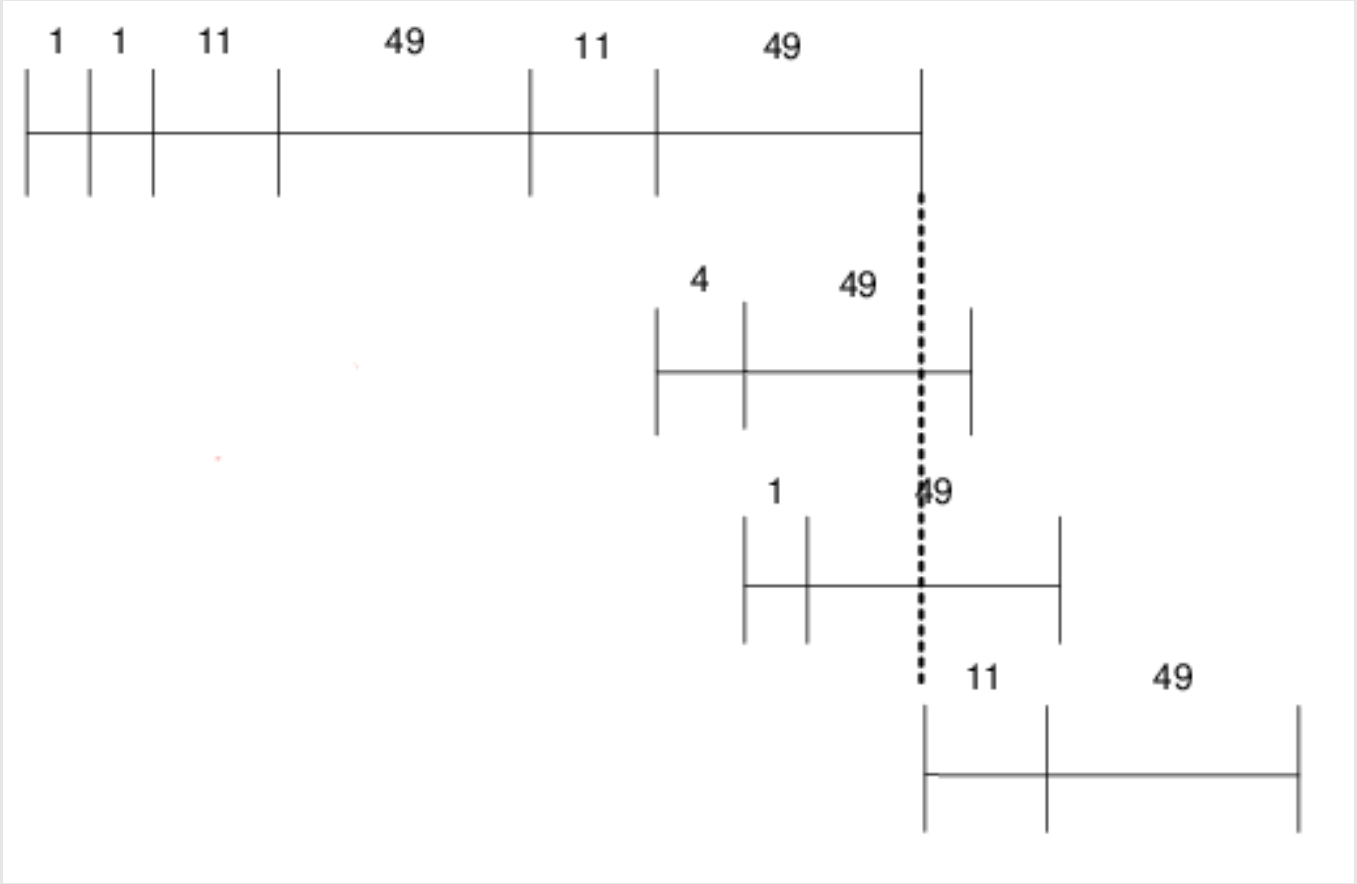
Vector Chaining

Vector chaining

- Data forwarding from one vector functional unit to another

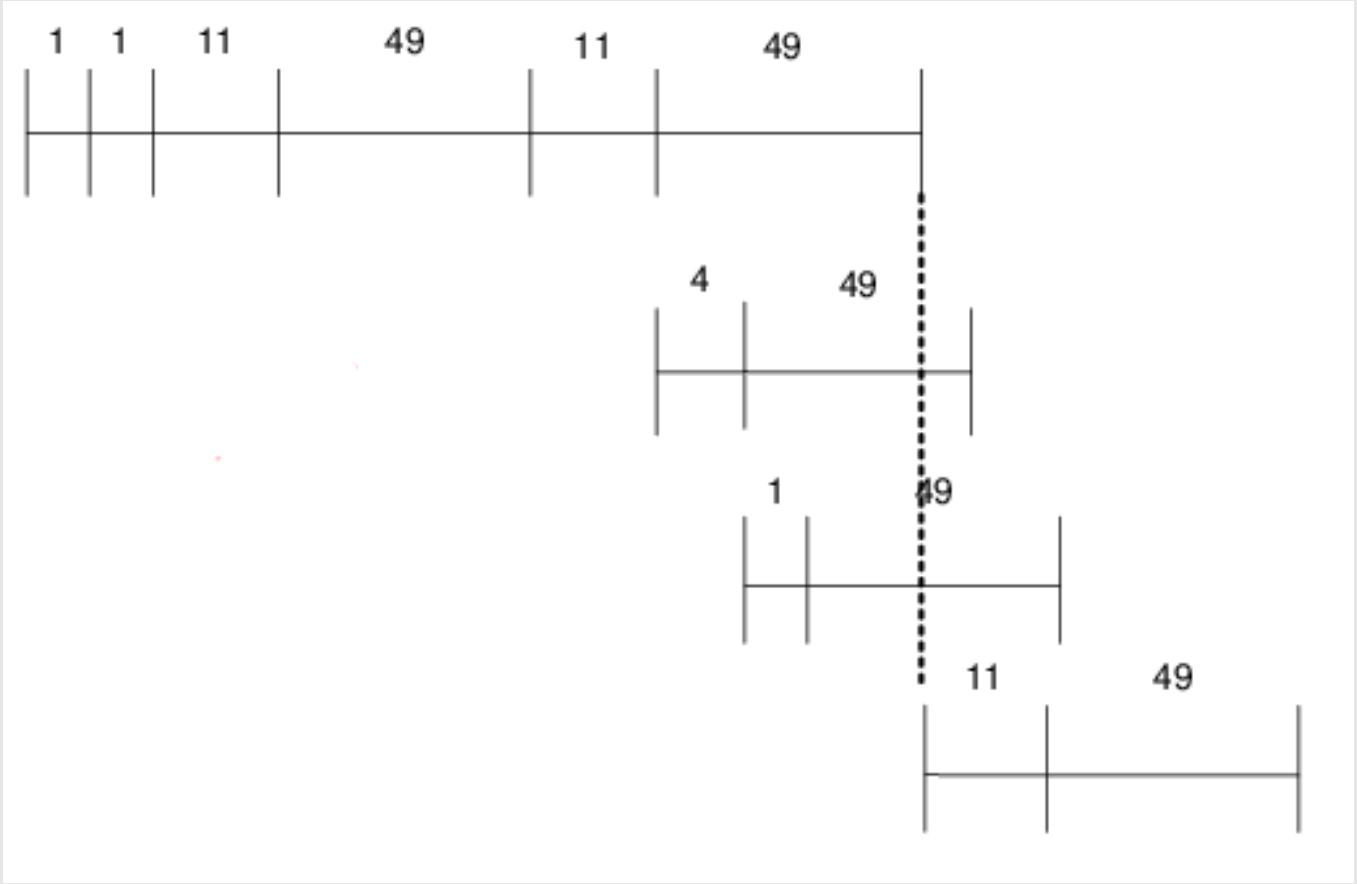


Vectorized Code Performance – With Chaining



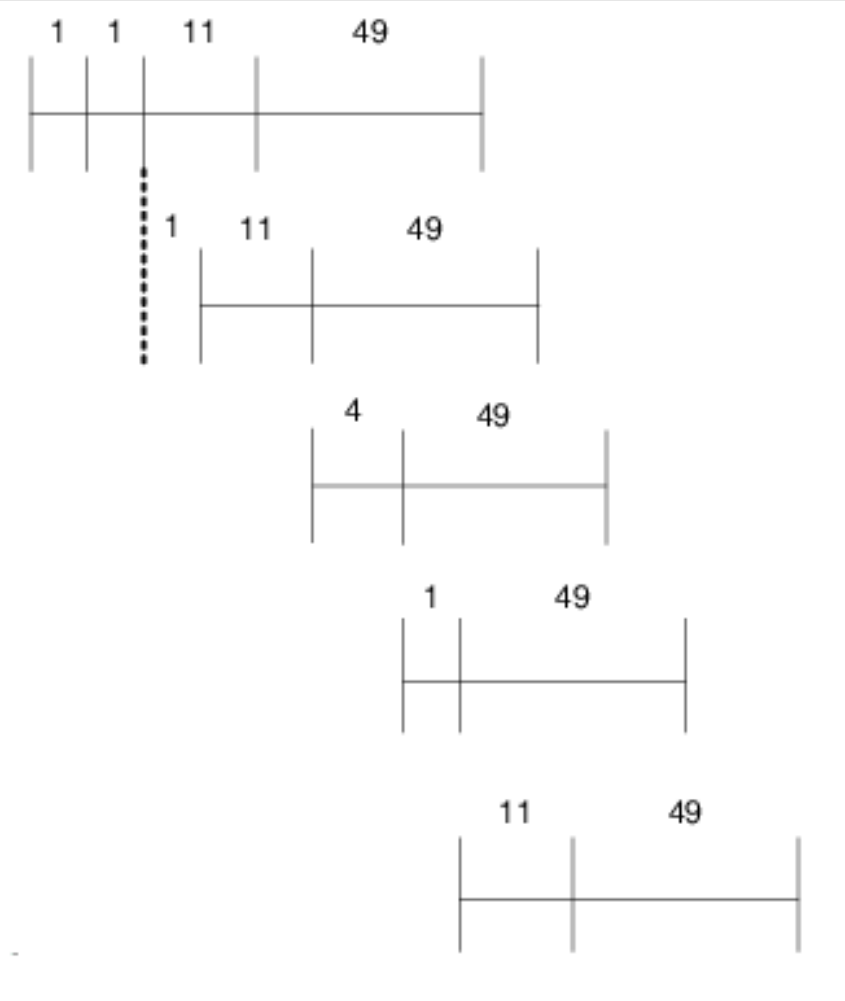
- $1 + 1 + 11 + 49 + 11 + 49 + 11 + 49 = 182$ cycles

Vectorized Code Performance – With Chaining



- $1 + 1 + 11 + 49 + 11 + 49 + 11 + 49 = 182$ cycles
- 2 first VLD cannot be pipelined?
- VLD & VST cannot be pipelined?

Vectorized Code Performance – Multiple Ports



- $1 + 1 + 1 + 11 + 4 + 1 + 11 + 49 = 79$ cycles
- $19\times$ improvment!

Discussions

What if #data elements > #elements in a vector register?

Discussions

What if #data elements > #elements in a vector register?

- Idea: Break loops
  - Each iteration operates on #elements vector register
  - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)

Discussions

What if #data elements > #elements in a vector register?

- Idea: Break loops
  - Each iteration operates on #elements vector register
  - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
- Called "vector stripmining"

Discussions

What if #data elements > #elements in a vector register?

- Idea: Break loops
  - Each iteration operates on #elements vector register
  - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
- Called "vector stripmining"

What if vector data is not stored in a strided fashion in memory?

Discussions

What if #data elements > #elements in a vector register?

- Idea: Break loops
  - Each iteration operates on #elements vector register
  - E.g., 527 data elements, 64-element VREGs
    - 8 iterations where VLEN = 64
    - 1 iteration where VLEN = 15 (need to change value of VLEN)
- Called "vector stripmining"

What if vector data is not stored in a strided fashion in memory?

- Idea: Use indirection
  - Combine/pack elements in vector registers
- Called "scatter/gather operations"

Gather/Scatter Operations

Vectorize loops with indirect accesses

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

Indexed load intruction (Gather)

```
LV vD, rD        ; Load indices in D vector
LVI vC, rC, vD   ; Load indirect from rC base
LV vB, rB        ; Load B vector
ADDV.D vA,vB,vC  ; Do add
SV vA, rA        ; Store result
```

Gather/Scatter Operations

- Gather/scatter often implemented in hardware
- Vector load/store addresses = base register + index vector

Example

Index Vector	Data Vector (to store)		Stored Vector (in memory)
0	3.14	Base+0	3.14
2	6.5	Base+1	x
6	71.2	Base+2	6.5
7	2.71	Base+3	x
		Base+4	x
		Base+5	x
		Base+7	71.2
		Base+7	2.71

Conditional Operations in a Loop

- What if some operations should not be executed on a vector?

```
loop: if (a[i] != 0) {
   b[i] = a[i]*b[i];
}
goto loop
```

Conditional Operations in a Loop

- What if some operations should not be executed on a vector?

```
loop: if (a[i] != 0) {
   b[i] = a[i]*b[i];
}
goto loop
```

Masked Operations

- VMASK register is a bit mask determining which data element should not be acted upon

```
VLD V0 = A
VLD V1 = B
VMASK = (V0 != 0)
VMUL V1 = V0 * V1
VST B = V1
```

Another Example with Masking

C Code

```
for (i = 0; i < 64; ++i)
     {
  if (a[i] >= b[i]) {
    c[i] = a[i];
  } else {
    c[i] = b[i];
  }
}
```

Another Example with Masking

C Code

```
for (i = 0; i < 64; ++i)
     {
  if (a[i] >= b[i]) {
    c[i] = a[i];
  } else {
    c[i] = b[i];
  }
}
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

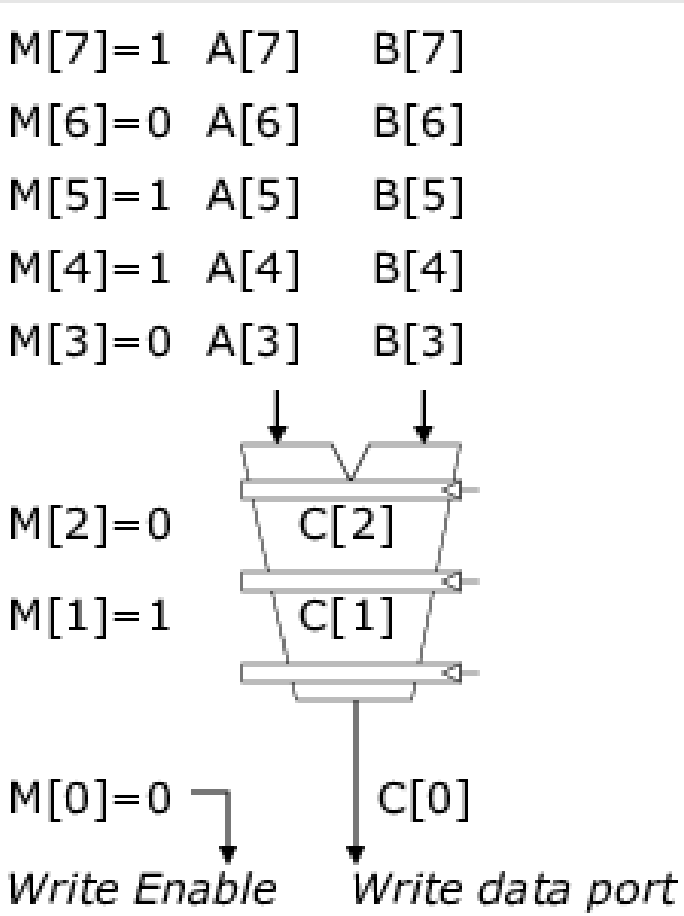
Steps to execute the loop in SIMD code

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

Masked Vector Instructions

Simple Implementation

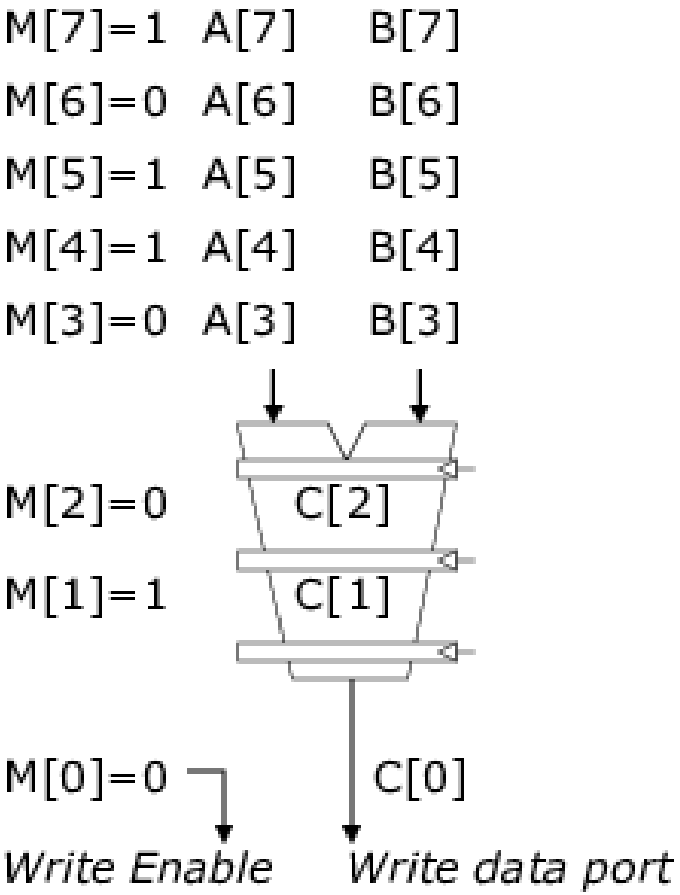
- Do all computations
- Prevent writing of output



Masked Vector Instructions

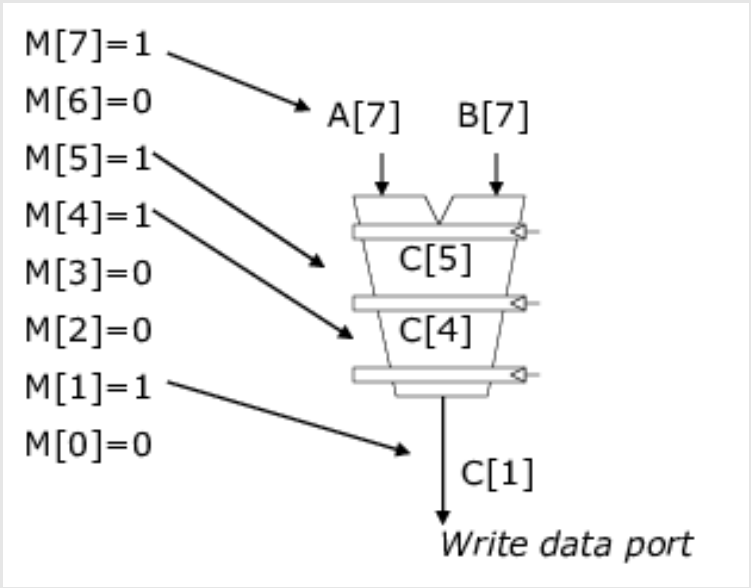
Simple Implementation

- Do all computations
- Prevent writing of output



Density-Time Implementation

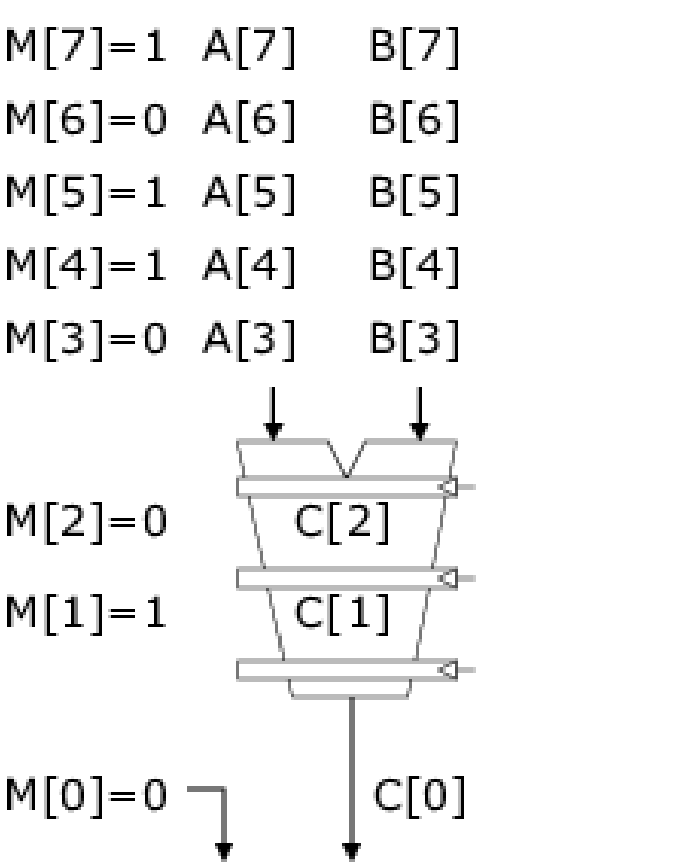
- Scan mask vector
- Only execute non-zero op



Masked Vector Instructions

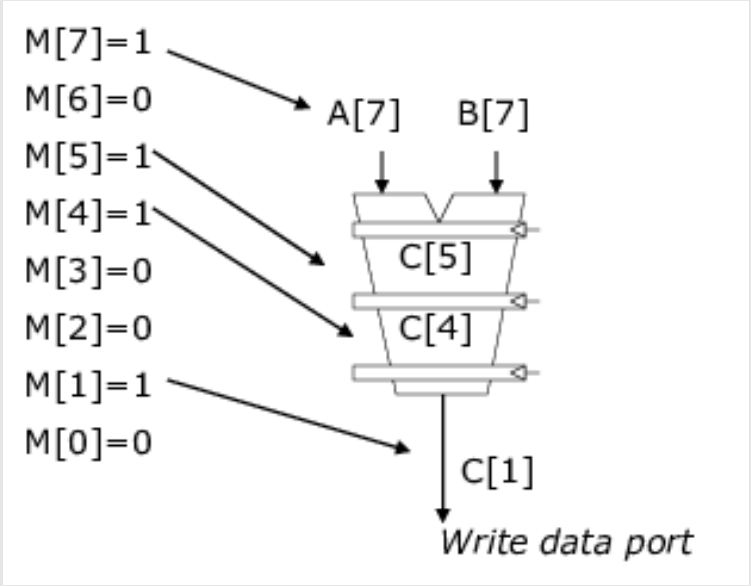
Simple Implementation

- Do all computations
- Prevent writing of output



Density-Time Implementation

- Scan mask vector
- Only execute non-zero op



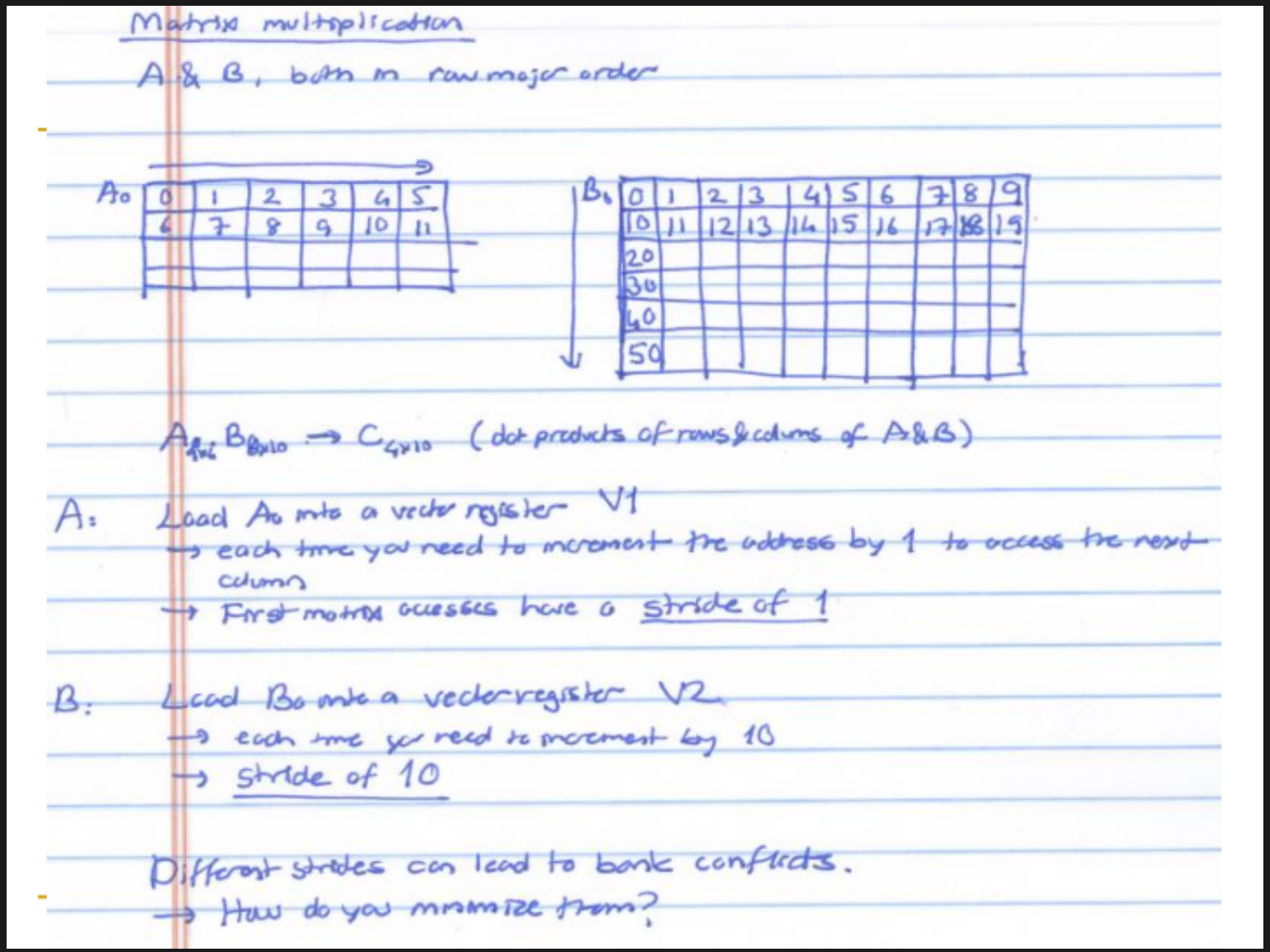
Comparison

- Which one is better?
- Trade-Offs?

Stride vs. Banking

- To sustain 1 element/cycle throughput
  - #Banks & Bank latency must be relatively prime
  - Requires enough banks to cover bank access latency
- Storage of a matrix
  - Row major vs. Column major (see next slide)
  - row $\rightarrow$ column $\Rightarrow$ change the stride

Issues – 2


Matrix multiplication
A & B, both in row major order
A0
0 1 2 3 4 5
6 7 8 9 10 11
B0
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20
30
40
50
A4x2 B2x10 → C4x10 (dot products of rows & colums of A & B)
A: Load A0 into a vector register V1
→ each time you need to increment the address by 1 to access the next colum
→ First matrix accesses have a stride of 1
B: Load B0 into a vector register V2
→ each time you need to increment by 10
→ stride of 10
Different strides can lead to bank conflicts.
→ How do you minimize them?

Minimizing Bank Conflicts

More banks

- Adding Banks solves the problem, but **$**

Minimizing Bank Conflicts

More banks

- Adding Banks solves the problem, but **$**

Better data layout to match the access pattern

- Is this always possible?

Minimizing Bank Conflicts

More banks

- Adding Banks solves the problem, but **$**

Better data layout to match the access pattern

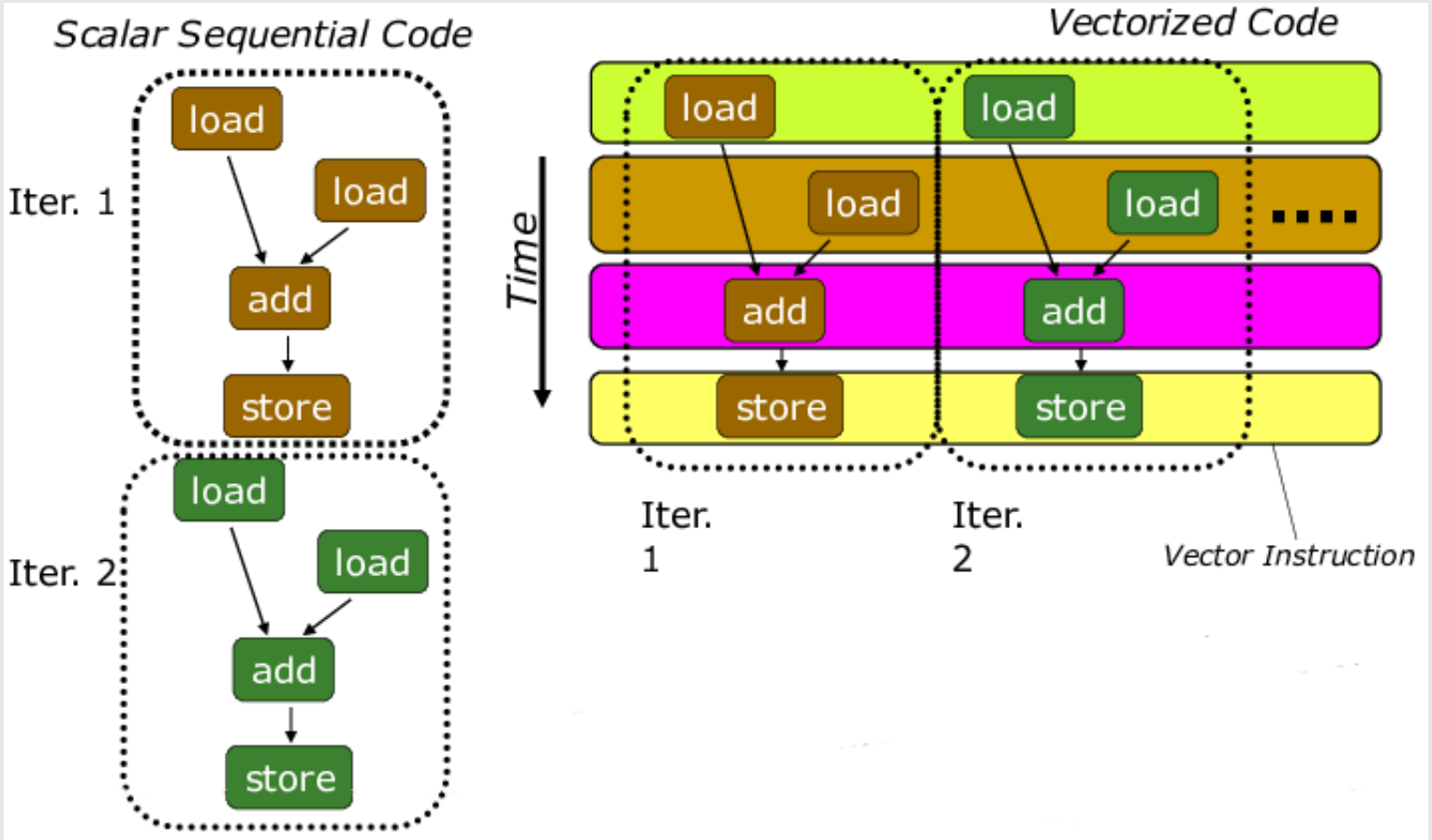
- Is this always possible?

Better mapping of address to bank

- E.g., randomized mapping

Automatic Code Vectorization

```
for (i = 0; i<=49; i++) {
  C[i] = (A[i] + B[i]) / 2
}
```



Vector/SIMD Processing Summary

Data-level parallelism

- Same operation performed on many data elements
- Improve performance, simplify design
- No intra-vector dependencies

Vectorizability is the limit

- Scalar operations limit vector machine performance
- Remember Amdahl's Law
- CRAY-1 was the fastest scalar machine at its time!

Current situation

- Many existing ISAs include (vector-like) SIMD operations
  - Intel MMX/SSE/AVX, ARM Advanced SIMD ...

Vector Processors vs. Array Processors

"purist's" distinction

- Array vs. vector processor distinction is a "purist's" distinction

Current situation

- Most "modern" SIMD processors are a combination of both
  - They exploit data parallelism in both time and space
  - GPUs are a prime example