16
Vector Architecture

18-548/15-548 Memory System Architecture
Philip Koopman
November 4, 1998

Required Reading:	**Cragon 11.0-11.2.2, 11.4-11.6.3**
Supplemental Reading:	**Hennessy & Patterson B.1-B.5**
	Siewiorek & Koopman 5.0-5.9
	Siewiorek, Bell & Newell Chapter 44 (Cray 1)

Carnegie Mellon

Assignments

- **By next class read about vector performance:**
  - Cragon 11.3-11.3.5, 11.7
  - Siewiorek & Koopman 7.4 (available on-line)
  - Supplemental Reading:
    - Hennessy & Patterson B.6-B.9
    - Palacharla & Kessler; ISCA 1994
- **Homework #9 due November 11**
- **Lab #5 due November 20**

Where Are We Now?

- **Where we've been:**
  - Main memory
  - Disk drives
- **Where we're going today:**
  - Vector architecture
  - Vector chaining
- **Where we're going next:**
  - Vector Performance
  - Buses

Preview

- **Vector processing overview**
  - Why was it a good idea in the 60s?
  - (Next lecture -- why it might be a good idea today...)
- **Generic vector processor architecture**
  - It's about data pipelining instead of instruction pipelining
- **Data pipelining within vector execution -- extended example**
  - Vector loads
  - Vectors stores
  - Vector "chaining"

WHAT IS VECTOR PROCESSING?

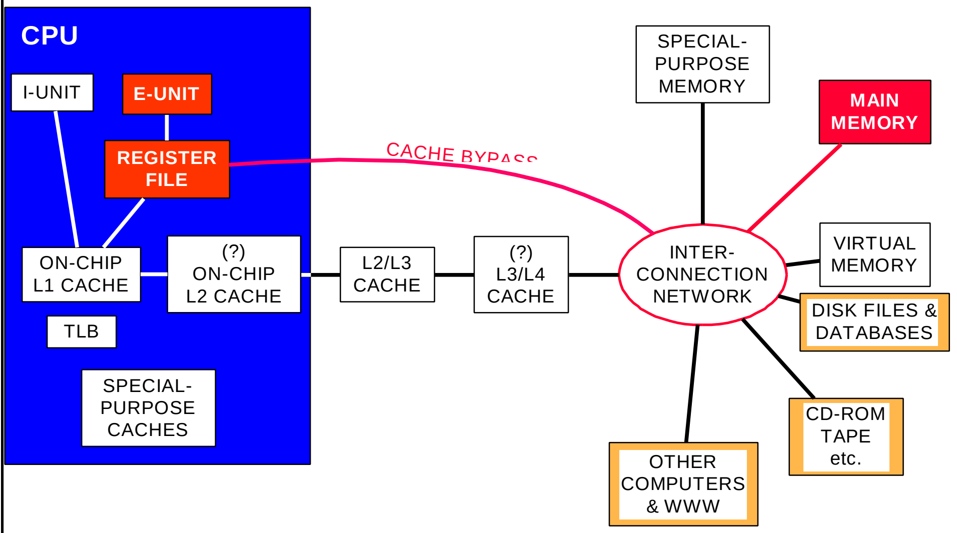
Vector Processing Definition

- **A vector instruction operates on a set of data elements**
  - Typically elements in a data array separated by fixed stride
  - Typically implemented as a load/store register architecture (but, “registers” contain many data elements each)

– Vector load	start address, stride, number of elements, register#
– Vector store	start address, stride, number of elements, register#
– Vector operation	source reg#1, source reg#2, dest reg

- **Traditional supercomputers are vector-based**
  - Scientific code typically operates on arrays and matrices
  - Provides hardware speedup for loop overhead and address generation
- **Vector techniques are useful, even outside of supercomputing**
  - Dealing with structured data that caches poorly, especially multimedia data streams

Classical Vector Processing Bypasses Cache



Historical View of Vector Processing

- **Scientific code speedup (Fortran heritage)**
  - Regular strided access patterns encouraged hardware speedup for loops
    - Single-issue machines didn't have to spend clocks on loop overhead
    - Vector data loads and stores could exploit interleaved memory capabilities
  - When caches became available, they weren't big enough to help
- **Reduced semantic gap between application and hardware -- allows compilers to make assumptions about data structures**
  - Assume no memory overlap on vector loads/stores (don't stall reads waiting for writes to complete)
  - Eliminate branch mis-prediction penalties (loops replaced by vector length)
  - Provided optimized performance in an era of immature compiler technology

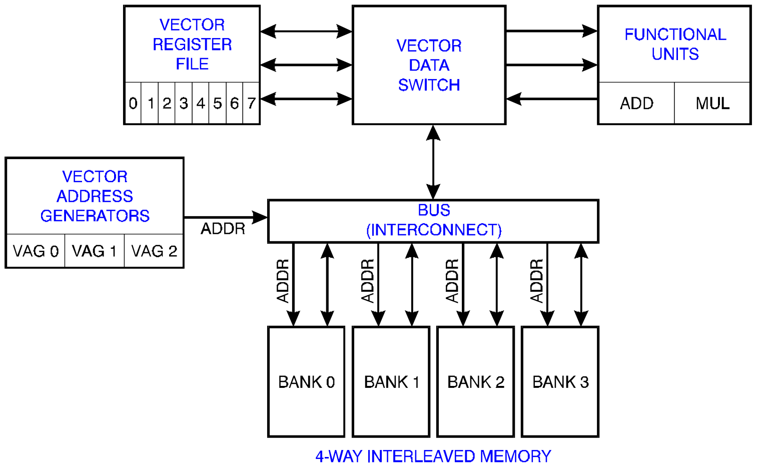
Vectorization As Pipelining

- ***Data* pipelining, rather than deep instruction execution pipelining**
  - Single instruction repeated multiple times on multiple data items
- **Single-issue machine could have multiple vector instructions active**
  - Even if one clock per result, vectors have many data elements
  - Concurrent instruction execution possible even with a single-issue machine

GENERIC VECTOR ARCHITECTURE

(Inspired by the Stardent Titan architecture; but this example is intentionally not balanced)

Generic Vector Architecture



- Vector Address Generators drive strided fetches/stores
- Interconnect/Bus supports pipelined+interleaved loads and stores
- Vector Register File holds collated data sets
- Functional units perform pipelined operations on vector register contents

Vector Register File

- **High-speed memory accessible as vector register set**
  - Each vector register holds multiple data elements forming a vector (*e.g.*, 8 vector registers, each with 64 data elements)
  - Length register may permit holding fewer than maximum elements in a vector register (*e.g.*, vector registers might have only 32 valid data elements, with the remainder of the vector register contents ignored)
- **Vector registers are a contiguous set of data**
  - Vector Registers always accessed in sequential data words (with “stride” of 1)
  - Memory arrays/vectors of stride > 1 are converted to/from contiguous data in VRF by load/store operations
- **Also need scalar registers**
  - Might be kept within VRF and accessed with special address form
  - Might be the scalar CPU floating point register set

Vector Address Generators

- **Coordinates data placement for load and store operations**
- **Generates strided memory addresses**
  - Seeded with vector information:
    - Start address
    - Stride
    - Number of elements
  - Generates addresses for memory load/store
- **Generates vector register addresses**
  - Vector register number
  - Offset within vector register
- **One VAG needed for each concurrent vector load/store**

Functional Units

- **Pipelined arithmetic and logic units, typically:**
  - Addition/logic functions
    1) Pre-normalization (align decimal points w.r.t. exponent values)
    2) Addition/subtraction
    3) Post-normalization (align resultant decimal point) & round
  - Multiplication
    1) Mantissa multiplication
    2) Partial product addition
    3) Post-normalization & round
  - Division/square root
- **Data may enter and leave functional unit every clock cycle for high throughput**
  - Vector execution model means software guarantees no data dependencies among vector elements

Vector Data Switch

- **Crossbar switch to connect system components**
  - Data rate must be one element per clock cycle per port
  - Number of ports determined by expected data flow rates
    - At least three VRF ports to support LOAD/LOAD/op/STORE functions
    - One or more memory ports to support concurrent Loads and Stores
    - At least three Functional Unit ports to support LOAD/LOAD/op/STORE functions

Multiple Memory Banks

- **Low end machine -- interleaved memory**
  - Memory banks take turns being connect to bus
  - Interleaved memory access improves available bandwidth and may reduce latency for concurrent accesses.
- **High end machine -- multiple concurrent banks**
  - Might use crossbar switch (instead of bus, not instead of VDS) to connect several memory banks to the VDS simultaneously
  - Might be interleaved and assume different subsets of banks connected each clock

Vector Instruction Forms

Result	**Operand1**	**Operand2**	**Example**	**Data Elements**
Vector	Memory	---	Load A(*i*)	*v*
Memory	Vector	---	Store A(*i*)	*v*
Vector	Scalar	Vector	C(*i*) Ü B + A(*i*)	2*v*+1
Vector	Vector	---	C(*i*) Ü abs(B(*i*))	2*v*
Vector	Vector	Mask	C(*i*) Ü B(*i*) Ç mask	up to 2*v*
Scalar	Vector	---	C Ü S B(*i*)	*v*+1
Scalar	Vector	Vector	C Ü S B(*i*) ´ A(*i*)	2*v*+1
Vector	Scalar	Vector	C(i) Ü A ´ B(*i*)+ C(*i*)	3*v*+1

- **Key instruction: DAXPY**
  - Y = aX + Y   ("double precision a-X plus Y")
  - Inner loop of LINPACK

DAXPY -- A Key Operation

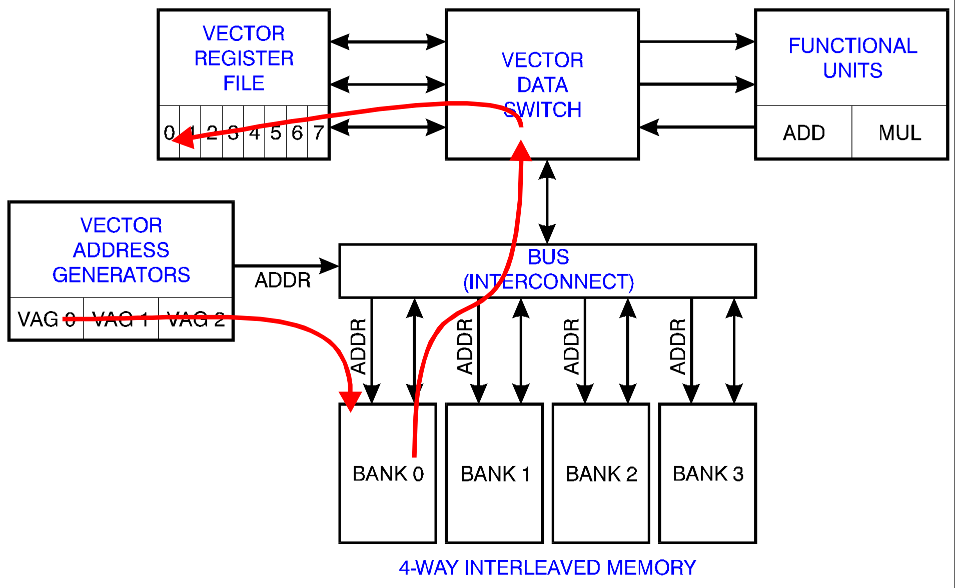
- **Used in Gaussian Elimination; "most used" supercomputer instruction**
  - LOAD -- LOAD -- OP -- STORE
- **for (i = 0; i < 64; i++) { Y(i) = a * X(i) + Y(i); }**

```
/* assume 64-element vector registers */
LD          F0, a           ; load scalar a
LV          V1, x           ; load vector register 1 with x
MULTSV      V2, F0, V1      ; vector-scalar multiply  F0 is a; V1 is X
LV          V3, y           ; load vector register 2 with y
ADDV        V4, V2, V3      ; add vector new Y (V4) = aX + old Y
SV          y, V4           ; Store result
```

- **Variations**
  - Vector length register could be set for varying array sizes
    - If vector exceeds vector register length, must use "strip mining"
  - Some systems have a DAXPY instruction: vLoad; vLoad; DAXPY; vStore

VECTOR EXAMPLE: Z(i) = X(i) + Y(i)

Vector Load

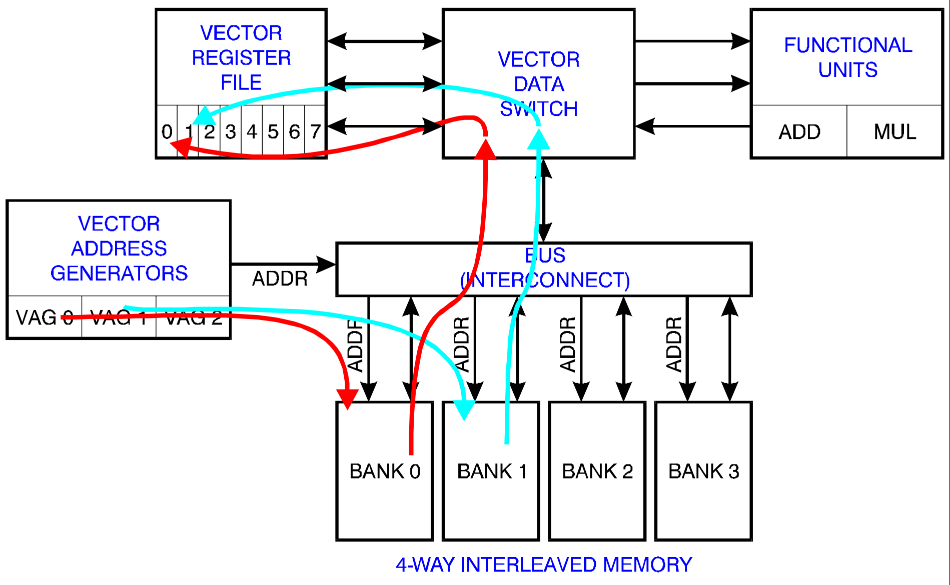


Vector Load Operation X(i)

Beat	VAG	ADDR BUS	BANK 0	BANK 1	BANK 2	BANK 3	DATA BUS	DATA SWITCH	VRF
1	X[0]								
2	X[1]	X[0]							
3	X[2]	X[1]	RAS X[0]						
4	X[3]	X[2]	CAS X[0]	RAS X[1]					
5	X[4]	X[3]	DATA X[0]	CAS X[1]	RAS X[2]				
6	X[5]	X[4]	CYCLE	DATA X[1]	CAS X[2]	RAS X[3]	X[0]		
7	X[6]	X[5]	RAS X[4]	CYCLE	DATA X[2]	CAS X[3]	X[1]	X[0]	
8	X[7]	X[6]	CAS X[4]	RAS X[5]	CYCLE	DATA X[3]	X[2]	X[1]	X[0]
9		X[7]	DATA X[4]	CAS X[5]	RAS X[6]	CYCLE	X[3]	X[2]	X[1]
10			CYCLE	DATA X[5]	CAS X[6]	RAS X[7]	X[4]	X[3]	X[2]
11				CYCLE	DATA X[6]	CAS X[7]	X[5]	X[4]	X[3]
12					CYCLE	DATA X[7]	X[6]	X[5]	X[4]
13						CYCLE	X[7]	X[6]	X[5]
14								X[7]	X[6]
15									X[7]

- **8-element vector load**
  - 8 beat latency to first load
  - 8 elements loaded in 15 beats latency
  - BUT, could load other data starting at beat 9...

Concurrent Vector Loads



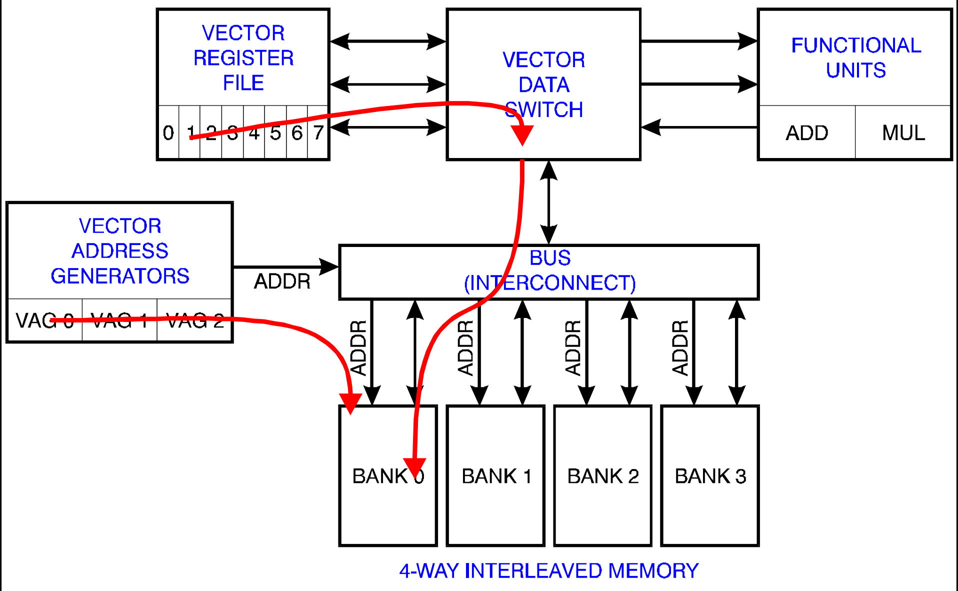
Concurrent Vector Load Operations X(i) Y(i)

Beat	VAG0	VAG1	ADDR BUS	BK 0	BK 1	BK 2	BK 3	DATA BUS	DATA SWITCH	VRF
1	X[0]									
2	X[1]	Y[0]	X[0]							
3		Y[1]	Y[0]	rX[0]						
4	X[2]		X[1]	cX[0]		rY[0]				
5		Y[2]	Y[1]	X[0]	rX[1]	cY[0]				
6		Y[3]	Y[2]	cyc	cX[1]	Y[0]	rY[1]	X[0]		
7	X[3]		X[2]	rY[2]	X[1]	cyc	cY[1]	Y[0]	X[0]	
8		Y[4]	Y[3]	cY[2]	cyc	rX[2]	Y[1]	X[1]	Y[0]	X[0]
9	X[4]		X[3]	Y[2]	rY[3]	cX[2]	cyc	Y[1]	X[1]	Y[0]
10	X[5]		X[4]	cyc	cY[3]	X[2]	rX[3]	Y[2]	Y[1]	X[1]
11		Y[5]	Y[4]	rX[4]	Y[3]	cyc	cX[3]	X[2]	Y[2]	Y[1]
12	X[6]		X[5]	cX[4]	cyc	rY[4]	X[3]	Y[3]	X[2]	Y[2]
13		Y[6]	Y[5]	X[4]	rX[5]	cY[4]	cyc	X[3]	Y[3]	X[2]
14		Y[7]	Y[6]	cyc	cX[5]	Y[4]	rY[5]	X[4]	X[3]	Y[3]
15	X[7]		X[6]	rY[6]	X[5]	cyc	cY[5]	Y[4]	X[4]	X[3]
16			Y[7]	cY[6]	cyc	rX[6]	Y[5]	X[5]	Y[4]	X[4]
17			X[7]	Y[6]	rY[7]	cX[6]	cyc	Y[5]	X[5]	Y[4]
18				cyc	cY[7]	X[6]	rX[7]	Y[6]	Y[5]	X[5]
19					Y[7]	cyc	cX[7]	X[6]	Y[6]	Y[5]
20					cyc		X[7]	Y[7]	X[6]	Y[6]
21							cyc	X[7]	Y[7]	X[6]
22									X[7]	Y[7]
23										X[7]

Notes on Concurrent Load Example

- **Single bus results in data bandwidth limit**
  - Could provide 2 paths (2 buses, or 2 ports on a crossbar switch)
  - Could run bus at 2x vector clock cycle (2 beats per clock cycle)
- **Two VAGs compete for bus**
  - Need arbitration logic to assure fair access
  - VAGs aren’t strictly alternating in order to exploit available memory banks
    - VAGs with addresses waiting might go in round-robin order
    - But, a VAG might have to skip a turn or wait if the memory bank it needs is busy

Vector Store

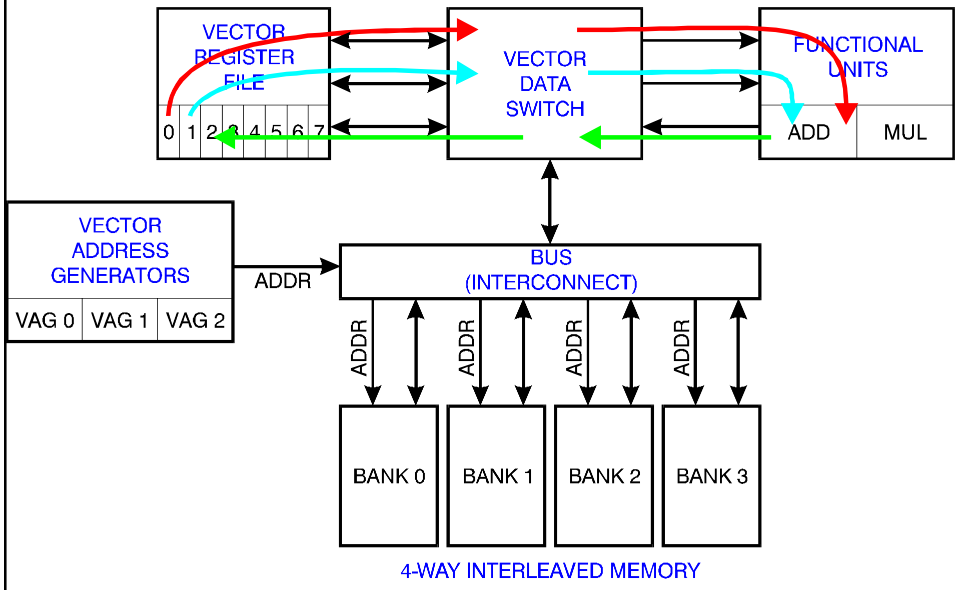


Vector Store Operation Z(i)

Beat	VAG	ADDR BUS	VRF	DATA SWITCH	DATA BUS	BANK 0	BANK 1	BANK 2	BANK 3
1	Z[0]		Z[0]						
2	Z[1]	Z[0]	Z[1]	Z[0]					
3	Z[2]	Z[1]	Z[2]	Z[1]	Z[0]	RAS Z[0]			
4	Z[3]	Z[2]	Z[3]	Z[2]	Z[1]	CAS Z[0]	RAS Z[1]		
5	Z[4]	Z[3]	Z[4]	Z[3]	Z[2]	CYCLE	CAS Z[1]	RAS Z[2]	
6	Z[5]	Z[4]	Z[5]	Z[4]	Z[3]		CYCLE	CAS Z[2]	RAS Z[3]
7	Z[6]	Z[5]	Z[6]	Z[5]	Z[4]	RAS Z[4]		CYCLE	CAS Z[3]
8	Z[7]	Z[6]	Z[7]	Z[6]	Z[5]	CAS Z[4]	RAS Z[5]		CYCLE
9		Z[7]		Z[7]	Z[6]	CYCLE	CAS Z[5]	RAS Z[6]	
10					Z[7]		CYCLE	CAS Z[6]	RAS Z[7]
11								CYCLE	CAS Z[7]
12									CYCLE

- **8-element vector store**
  - 4 beat latency to first store -- data transmission in parallel with addressing
  - Assume writing one clock faster than reading because don’t have to wait for data to get back out of DRAM -- address and data arrive concurrently

Vector Add



Vector Add Operation registers: Z = X + Y

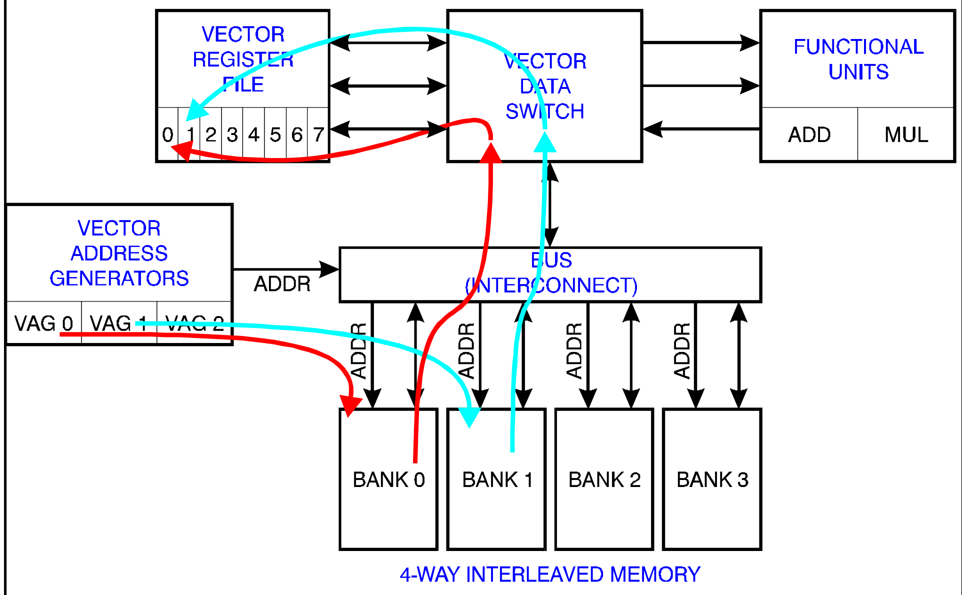
Beat	VRF			DATA SWITCH			ADDER (3 STAGES)		
1	X[0]	Y[0]							
2	X[1]	Y[1]		X[0]	Y[0]				
3	X[2]	Y[2]		X[1]	Y[1]		Z[0]		
4	X[3]	Y[3]		X[2]	Y[2]		Z[1]	Z[0]	
5	X[4]	Y[4]		X[3]	Y[3]		Z[2]	Z[1]	Z[0]
6	X[5]	Y[5]		X[4]	Y[4]	Z[0]	Z[3]	Z[2]	Z[1]
7	X[6]	Y[6]	Z[0]	X[5]	Y[5]	Z[1]	Z[4]	Z[3]	Z[2]
8	X[7]	Y[7]	Z[1]	X[6]	Y[6]	Z[2]	Z[5]	Z[4]	Z[3]
9			Z[2]	X[7]	Y[7]	Z[3]	Z[6]	Z[5]	Z[4]
10			Z[3]			Z[4]	Z[7]	Z[6]	Z[5]
11			Z[4]			Z[5]		Z[7]	Z[6]
12			Z[5]			Z[6]			Z[7]
13			Z[6]			Z[7]			
14			Z[7]						

- **8-element vector add**
  - Data pipelining:
    - 7 beats to first result
    - 1 beat per result steady state

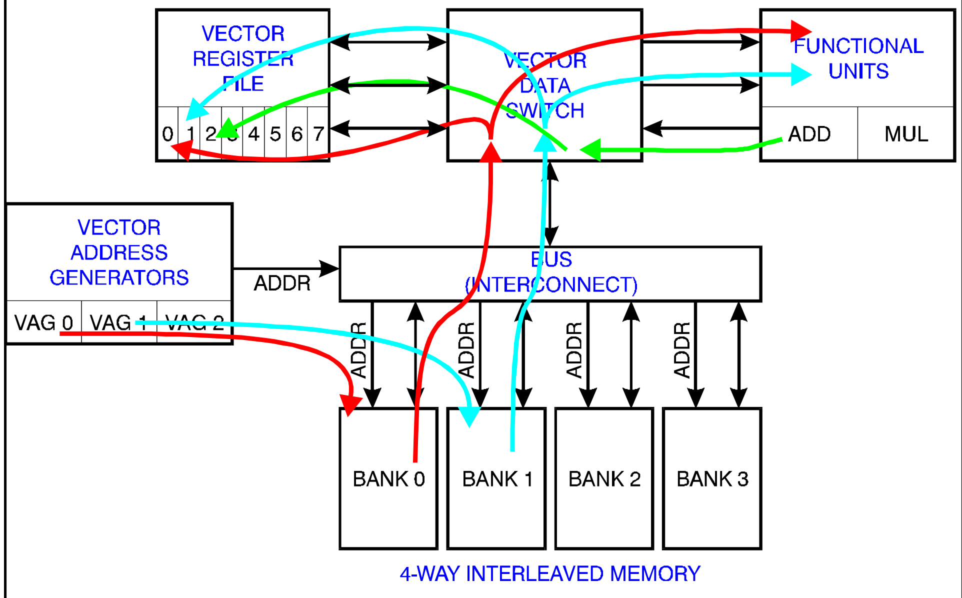
Vector Chaining

- **Vector chaining is analogous to data forwarding in a scalar processor**
  - Accomplished by hardware
    - Semantics of vector operations assume no implicit data hazards among vectors
  - Load/op chaining:
    - Vectors loaded via data switch from memory
    - Subsequent arithmetic operations fed from data switch as data arrives
  - Op/op chaining:
    - Outputs from arithmetic op fed back into functional units
  - Op/store chaining:
    - Outputs from arithmetic op fed straight to memory from data switch

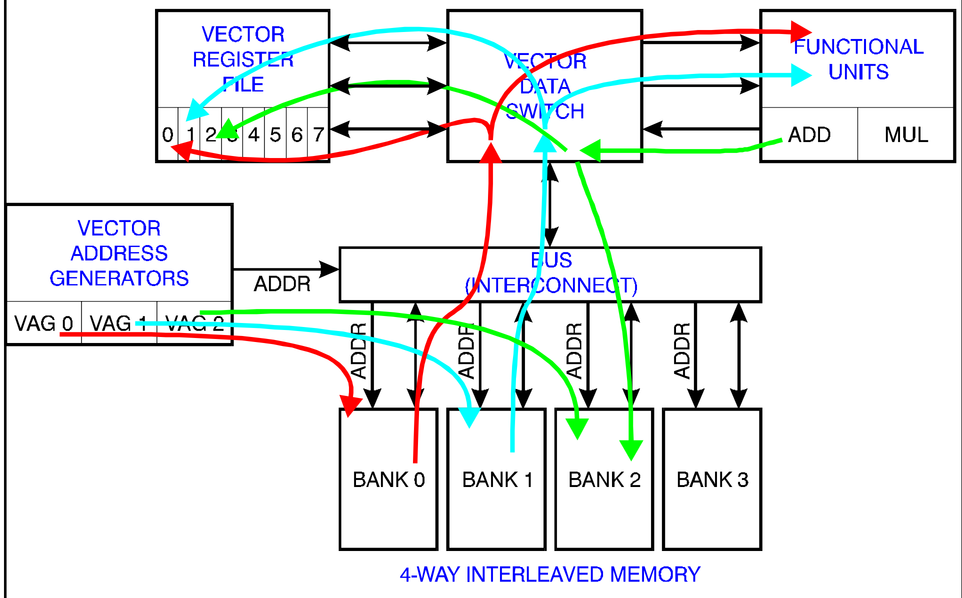
Vector Chaining -- loads (before chaining starts)



Vector Chaining -- loads + operation



Vector Chaining -- loads + operation + store



Vector Chaining: load / load / op / store

Beat	VAG0	VAG1	VAG1	ADR	BK 0	BK 1	BK 2	BK 3	BUS	VDS	ADD			VRF
1	X[0]													
2	X[1]	Y[0]		X[0]										
3		Y[1]		Y[0]	rX[0]									
4	X[2]		Z[0]	X[1]	cX[0]		rY[0]							
5		Y[2]		Y[1]	X[0]	rX[1]	cY[0]							
6		Y[3]		Y[2]	cyc	cX[1]	Y[0]	rY[1]	X[0]					
7	X[3]			X[2]	rY[2]	X[1]	cyc	cY[1]	Y[0]	X[0]				
8		Y[4]		Y[3]	cY[2]	cyc	rX[2]	Y[1]	X[1]	Y[0]				X[0]
9	X[4]			X[3]	Y[2]	rY[3]	cX[2]	cyc	Y[1]	X[1]	Z[0]			Y[0]
10	X[5]			X[4]	cyc	cY[3]	X[2]	rX[3]	Y[2]	Y[1]		Z[0]		X[1]
11		Y[5]		Y[4]	rX[4]	Y[3]	cyc	cX[3]	X[2]	Y[2]	Z[1]		Z[0]	Y[1]
12	X[6]			X[5]	cX[4]	cyc	rY[4]	X[3]	Y[3]	X[2],Z[0]		Z[1]		Y[2]
13		Y[6]		Y[5]	X[4]	rX[5]	cY[4]	cyc	X[3]	Y[3]	Z[2]		Z[1]	X[2],Z[0]
14		Y[7]		Y[6]	cyc	cX[5]	Y[4]	rY[5]	X[4]	X[3],Z[1]		Z[2]		Y[3]
15	X[7]			X[6]	rY[6]	X[5]	cyc	cY[5]	Y[4]	X[4]	Z[3]		Z[2]	X[3],Z[1]
16				Y[7]	cY[6]	cyc	rX[6]	Y[5]	X[5]	Y[4],Z[2]		Z[3]		X[4]
17				X[7]	Y[6]	rY[7]	cX[6]	cyc	Y[5]	X[5]	Z[4]		Z[3]	Y[4],Z[2]
18					cyc	cY[7]	X[6]	rX[7]	Y[6]	Y[5],Z[3]		Z[4]		X[5]
19						Y[7]	cyc	cX[7]	X[6]	Y[6]	Z[5]		Z[4]	Y[5],Z[3]
20						cyc		X[7]	Y[7]	X[6],Z[4]		Z[5]		Y[6],Z[0]
21			Z[1]	Z[0]				cyc	X[7]	Y[7],Z[0]	Z[6]		Z[5]	X[6],Z[4],Z[1]
22			Z[2]	Z[1]		rZ[0]			Z[0]	X[7],Z[5],Z[1]		Z[6]		Y[7],Z[2]
23			Z[3]	Z[2]		Z[0]	rZ[1]		Z[1]	Z[2]	Z[7]		Z[6]	X[7],Z[5],Z[3]
24			Z[4]	Z[3]		cyc	Z[1]	rZ[2]	Z[2]	Z[6],Z[3]		Z[7]		Z[4]
25			Z[5]	Z[4]	rZ[3]		cyc	Z[2]	Z[3]	Z[4]			Z[7]	Z[6],Z[5]
26			Z[6]	Z[5]	Z[3]	rZ[4]		cyc	Z[4]	Z[7],Z[5]				Z[6]
27			Z[7]	Z[6]	cyc	Z[4]	rZ[5]		Z[5]	Z[6]				Z[7],Z[7]
28				Z[7]		cyc	Z[5]	rZ[6]	Z[6]	Z[7]				
29					rZ[7]		cyc	Z[6]	Z[7]					
30					Z[7]			cyc						
					cyc									

Vector Chaining Analysis

- **Example: 8 adds in 30 beats**
  - 8 of 30 beats of ADDER consumed
  - 24 of 30 beats of bus consumed (both address and data)
  - 24 of 90 available beats of vector address generator consumed
  - 88 of 120 available beats of memory bandwidth consumed
- **Problem with this example architecture: lack of balance**
  - Bus a bottleneck
  - Memory bandwidth is close to being a bottleneck
  - Vector address generators under-used
  - Adder under-used

  - Lack of bus bandwidth is a common architectural problem
    - Bus bandwidth is more expensive getting a faster CPU
    - Toy benchmarks don’t use the bandwidth, so there may not be “obvious” benefit

REVIEW

Vector Microprocessors Today?

- **Vector computation model not as compelling as it once was**
  - Multi-issue, latency-tolerant architectures reduce cost of loop overhead
    - Instruction concurrency is available, and can substitute for data concurrency
  - Improved compiler technology reduces value of programmer using vectors to give hints to hardware
    - Improved algorithms to exploit cache
    - Smart pre-fetching hardware, cache bypass, latency tolerance
  - Commodity networked computing can often achieve comparable performance to a supercomputer
    - Single-chip CPUs now have very high clock rates
    - Improved infrastructure for parallel computing makes it accessible
- **But, desktop CPUs can benefit from supercomputer tricks**
  - Strided prefetching to reduce latency and better use memory bandwidth
  - Selective bypassing of cache to avoid cache pollution
  - Intel i860 was an experiment in this direction; but it was a poor compiler target

Review

- **Vector processing overview**
  - Exploits regular data access patterns to achieve data movement pipelining
- **Generic vector processor architecture**
  - VRF, VAG, VDS, functional units, memory banks
- **Data pipelining within vector execution**
  - Vector loads
  - Vectors stores
  - Vector chaining