高级计算机系统结构

沈海华
shenhh@ucas.ac.cn

第十五讲 Vector Computers

Daniel Sanchez
Computer Science and Artificial Intelligence Laboratory
M.I.T.

Supercomputers

Definition of a supercomputer:

- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing $30M+
- Any machine designed by Seymour Cray

CDC6600 (Cray, 1964) regarded as first supercomputer

Supercomputer Applications

Typical application areas

- Military research (nuclear weapons, cryptography)
- Scientific research
- Weather forecasting
- Oil exploration
- Industrial design (car crash simulation)
- Bioinformatics
- Cryptography

All involve huge computations on large data sets

In 70s-80s, Supercomputer ≡ Vector Machine

Loop Unrolled Code Schedule

```
loop:  ld f1, 0(r1)
       ld f2, 8(r1)
       ld f3, 16(r1)
       ld f4, 24(r1)
       add r1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       sd f5, 0(r2)
       sd f6, 8(r2)
       sd f7, 16(r2)
       sd f8, 24(r2)
       add r2, 32
       bne r1, r3, loop
```

Schedule

loop:

Int1	Int 2	M1	M2	FP+	FPx
		ld f1			
		ld f2			
		ld f3			
add r1		ld f4		fadd f5	
				fadd f6	
				fadd f7	
				fadd f8	
		sd f5			
		sd f6			
		sd f7			
add r2	bne	sd f8			

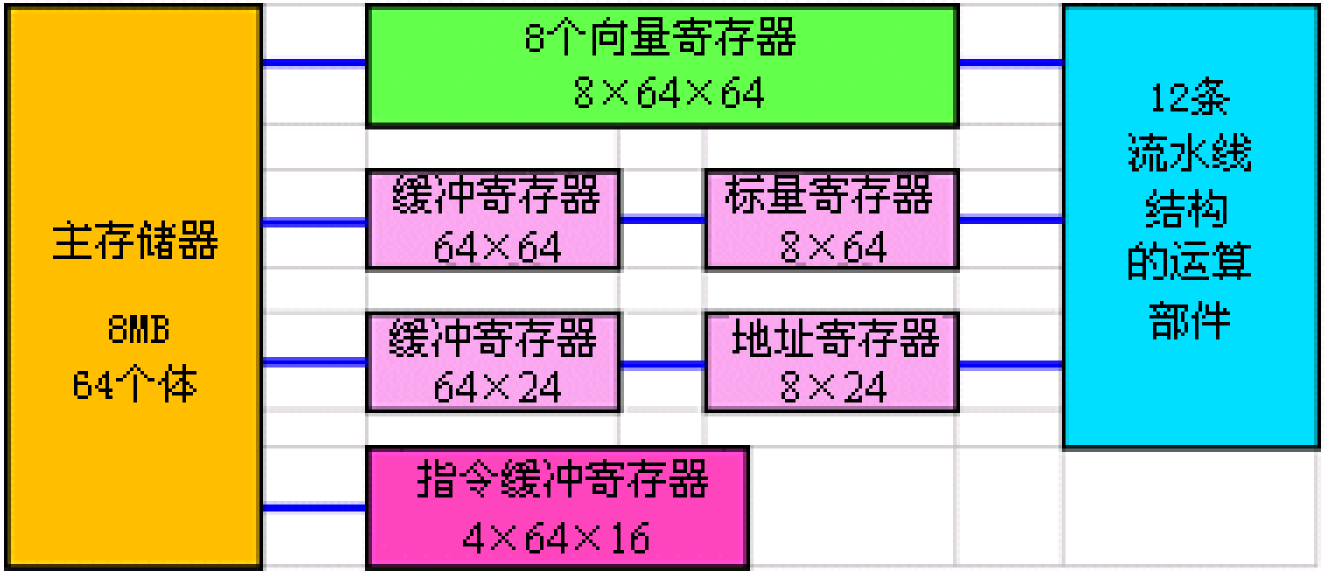
Vector Supercomputers

Epitomized by Cray-1, 1976:

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

Cray-1 (1976)

Cray-1 (1976)



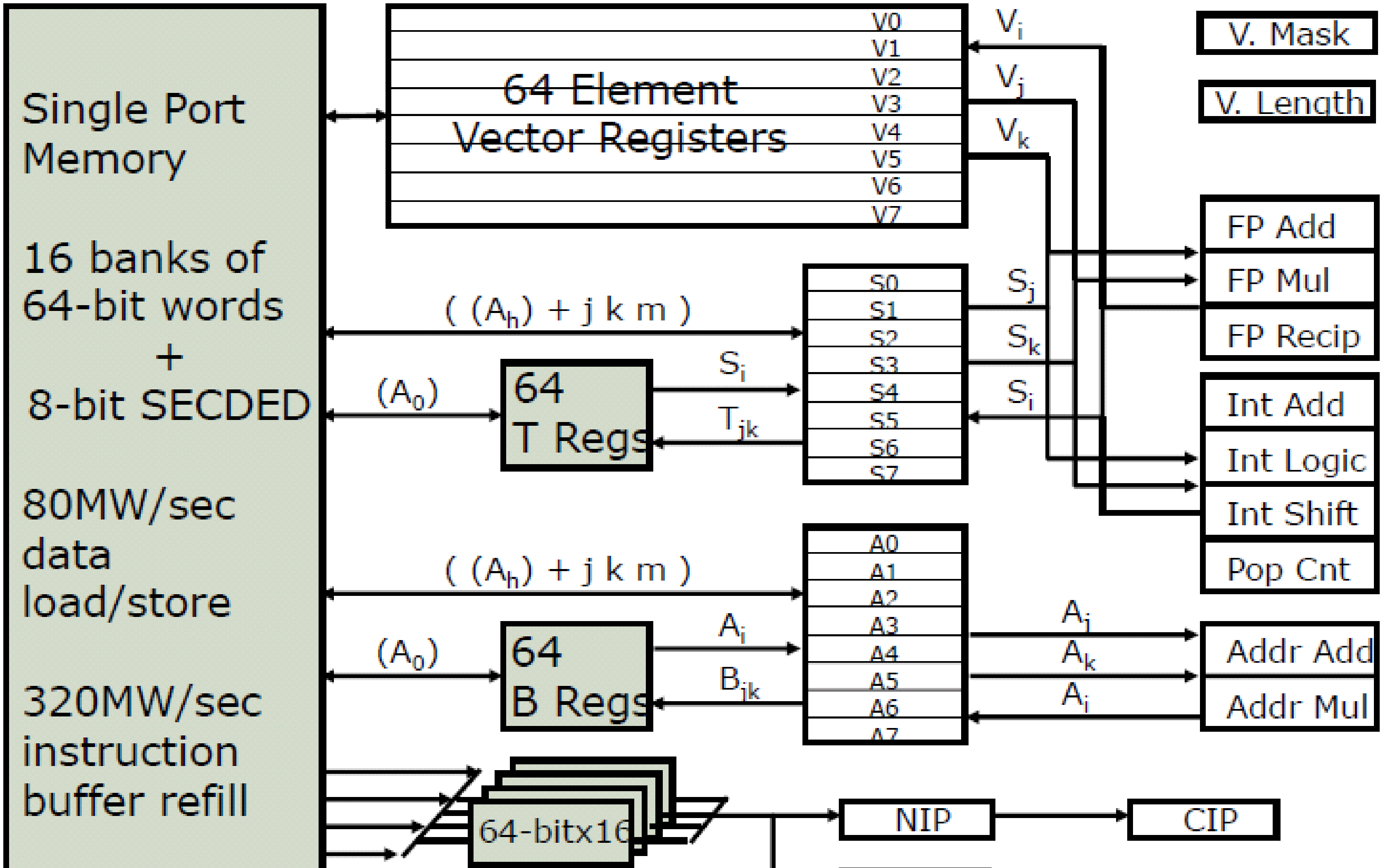
将存储器-存储器结构优化为寄存器-寄存器结构，运算部件需要的操作数从向量寄存器中读取，运算的中间结果也写到向量寄存器中。

优点：降低主存储器的流量。

寄存器-寄存器结构的CRAY-1运算速度是同时代存储器-存储器结构的STAR-100的3倍多，主存储器流量低2.5倍。

- STAR-100的主存储器流量：200MW/S
- CRAY-1的主存储器流量：80MW/S。

Cray-1 (1976)

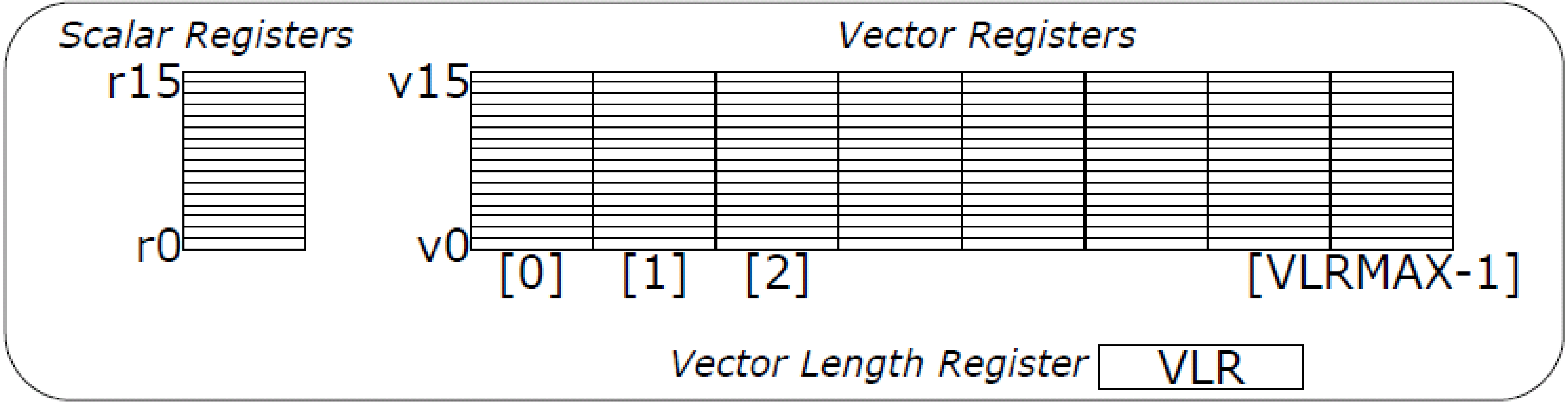


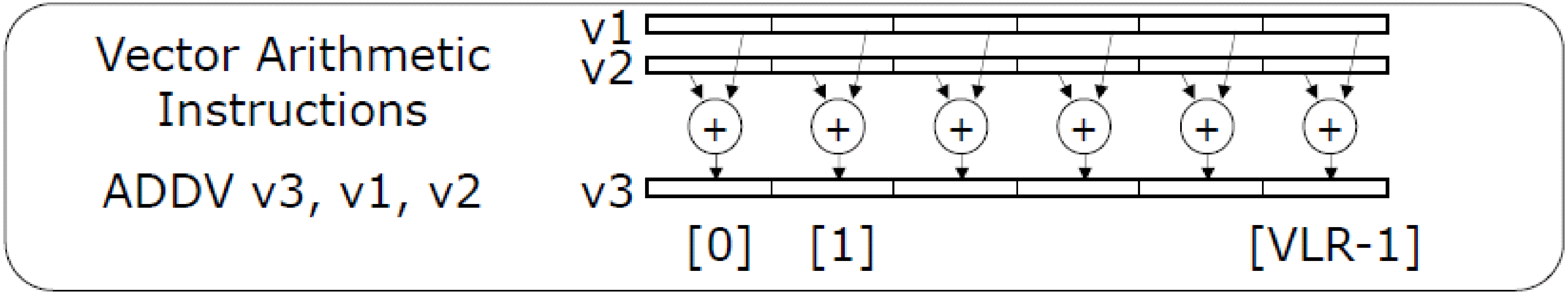
memory bank cycle 50 ns

4 Instruction Buffers

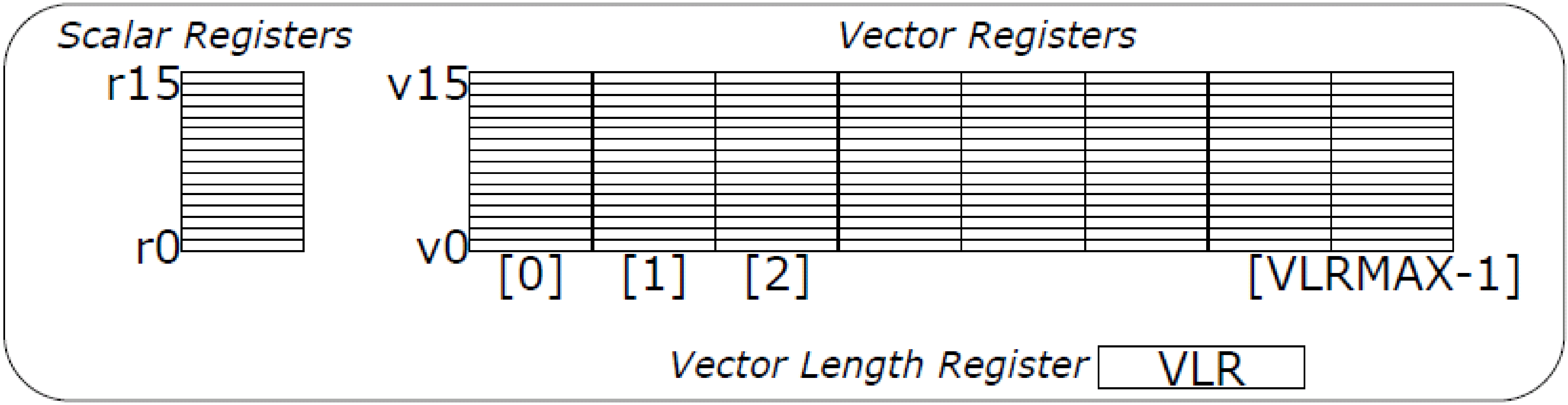
processor cycle 12.5 ns (80MHz)

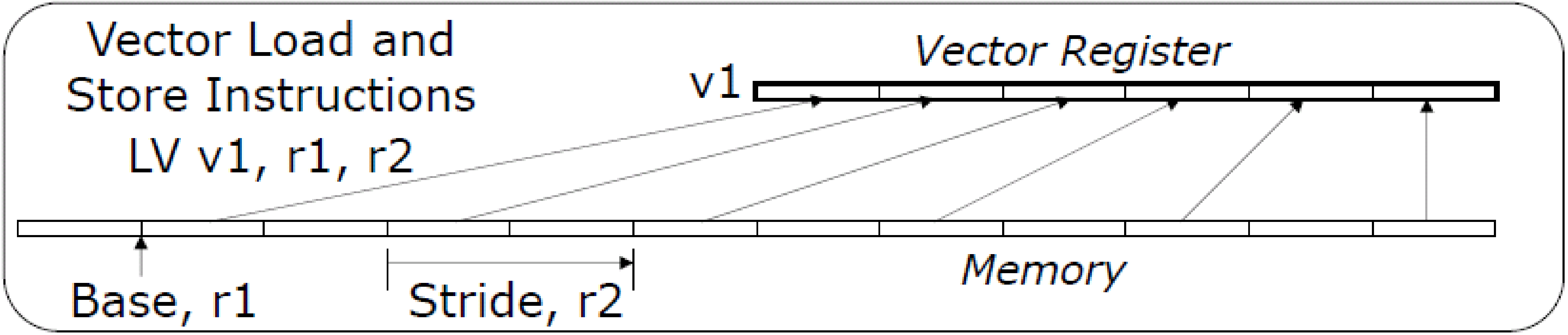
Vector Programming Model





Vector Programming Model

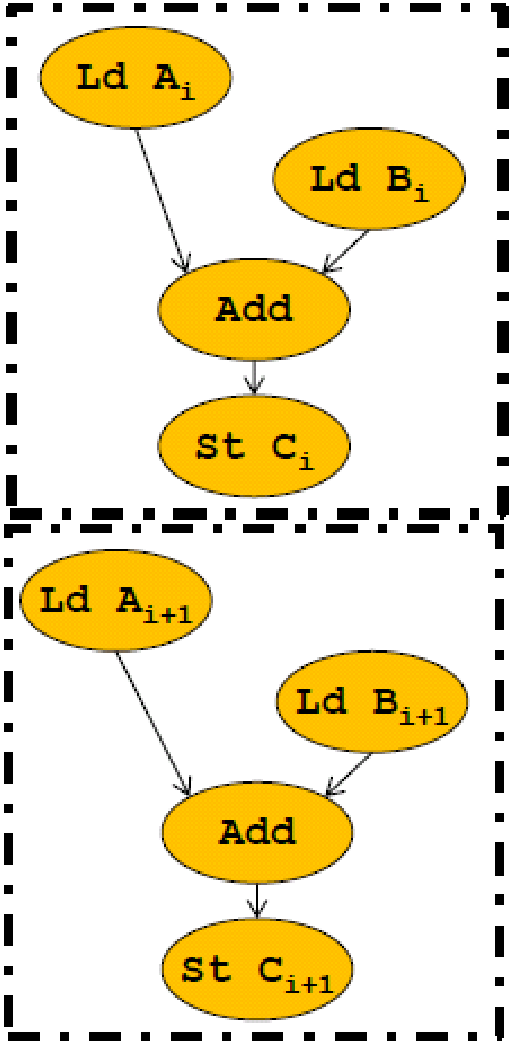




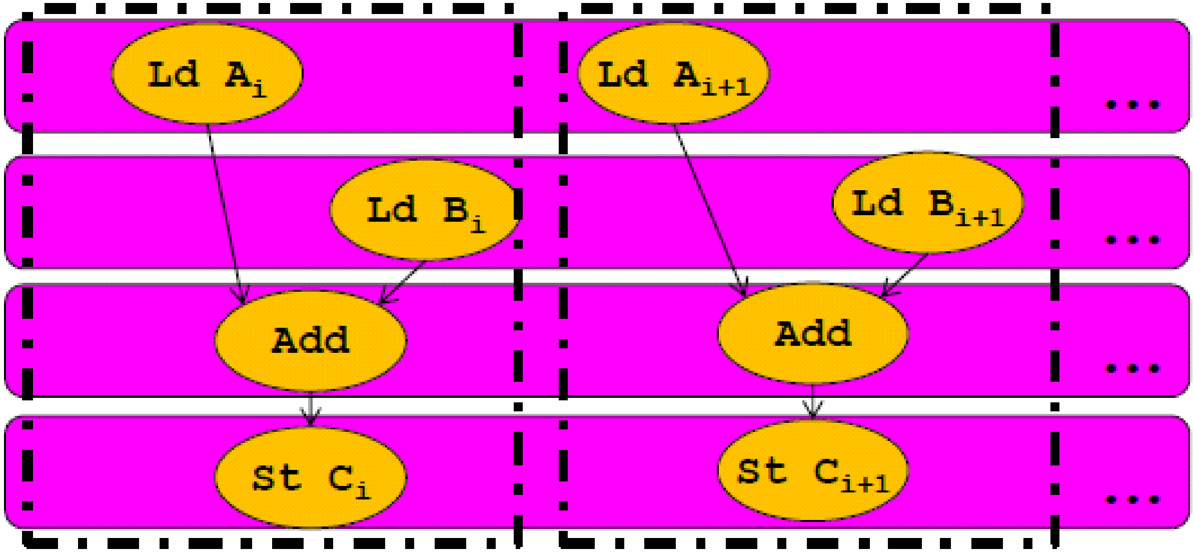
Compiler-based Vectorization

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

Compiler recognizes independent operations with loop dependence analysis



Scalar code



Vector code

Vector Code Example

# C code	# Scalar Code	# Vector Code
for (i=0; i<64; i++)	LI R4, 64	LI VLR, 64
C[i] = A[i] + B[i];	loop:	LV V1, R1
	L.D F0, 0(R1)	LV V2, R2
	L.D F2, 0(R2)	ADDV.D V3, V1, V2
	ADD.D F4, F2, F0	SV V3, R3
	S.D F4, 0(R3)	
	DADDIU R1, 8	
	DADDIU R2, 8	
	DADDIU R3, 8	
	DSUBIU R4, 1	
	BNEZ R4, loop	

Vector ISA Attributes

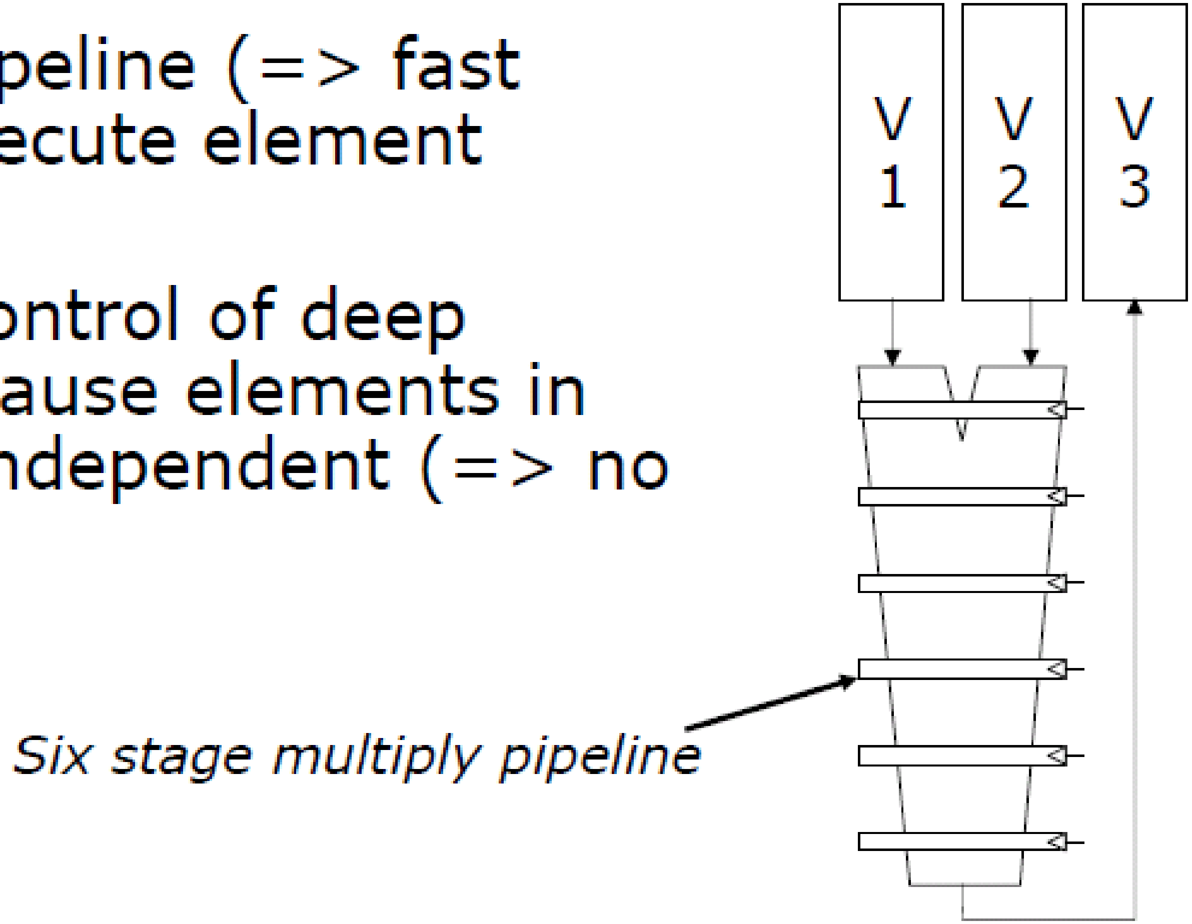
- Compact
  - one short instruction encodes N operations
  - many implicit bookkeeping/control operations
- Expressive, tells hardware that these N operations:
  - are independent
  - use the same functional unit
  - access disjoint registers
  - access registers in same pattern as previous instructions
  - access a contiguous block of memory (unit-stride load/store)
  - access memory in a known pattern (strided load/store)

Vector ISA Hardware Implications

- Large amount of work per instruction
  -> Less instruction fetch bandwidth requirements
  -> Allows simplified instruction fetch design
- Implicit bookkeeping operations
  -> Bookkeeping can run in parallel with main compute
- Disjoint vector element accesses
  -> Banked rather than multi-ported register files
- No data dependence within a vector
  -> Amenable to deeply pipelined/parallel designs
- Known regular memory access pattern
  -> Allows for banked memory for higher bandwidth

Vector Arithmetic Execution

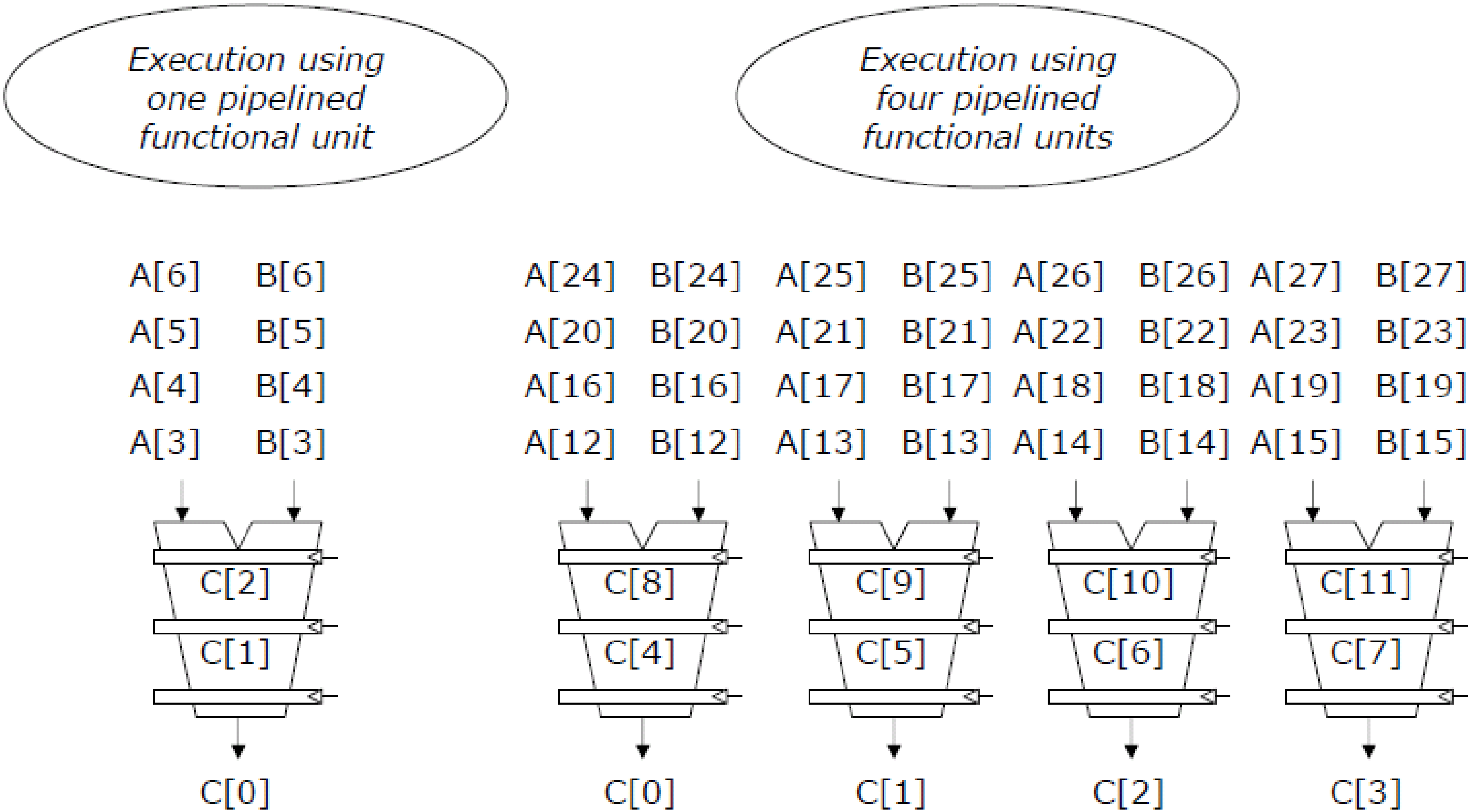
- Use deep pipeline (=> fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent (=> no hazards!)



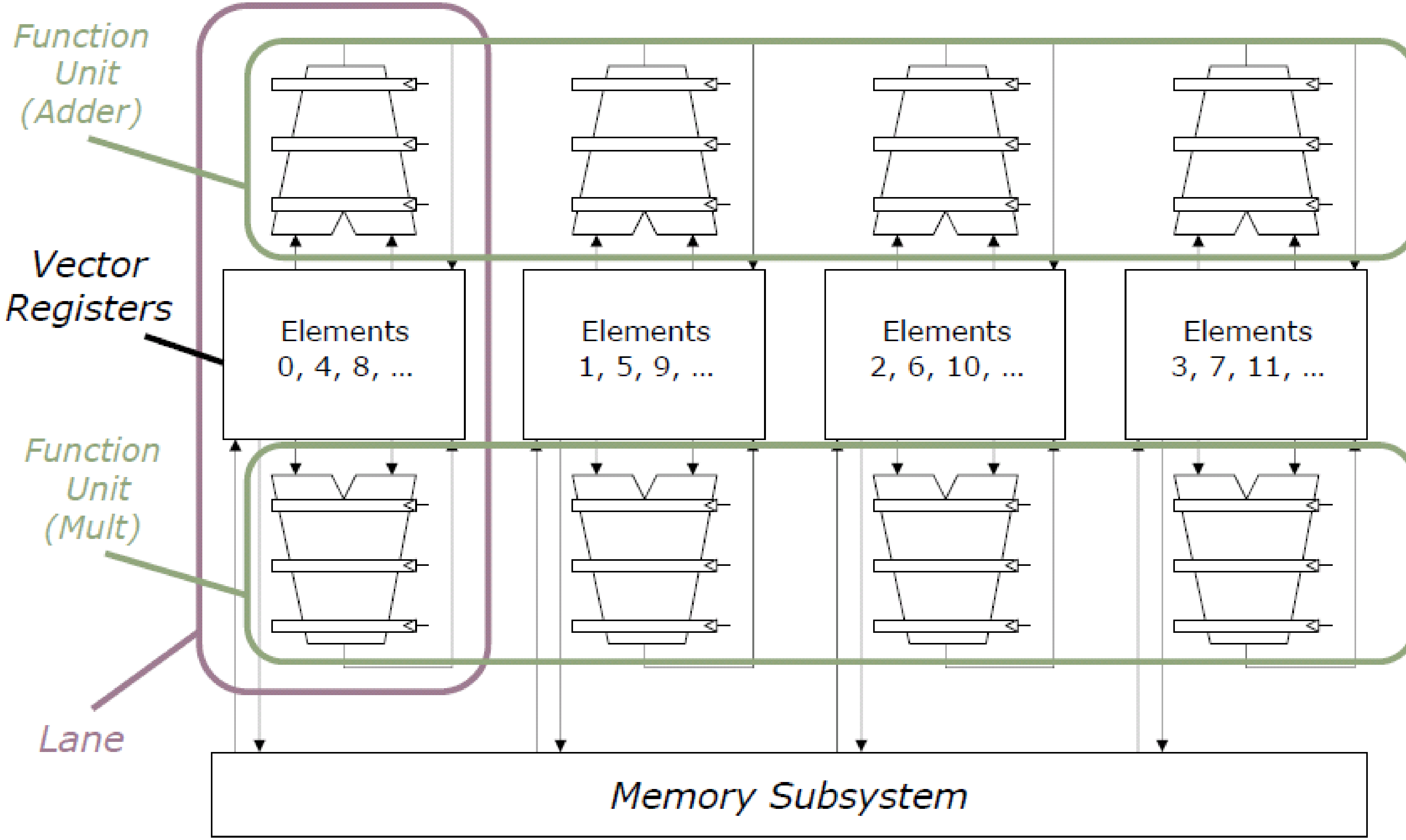
V3 <- v1 * v2

Vector Instruction Execution

ADDV C,A,B



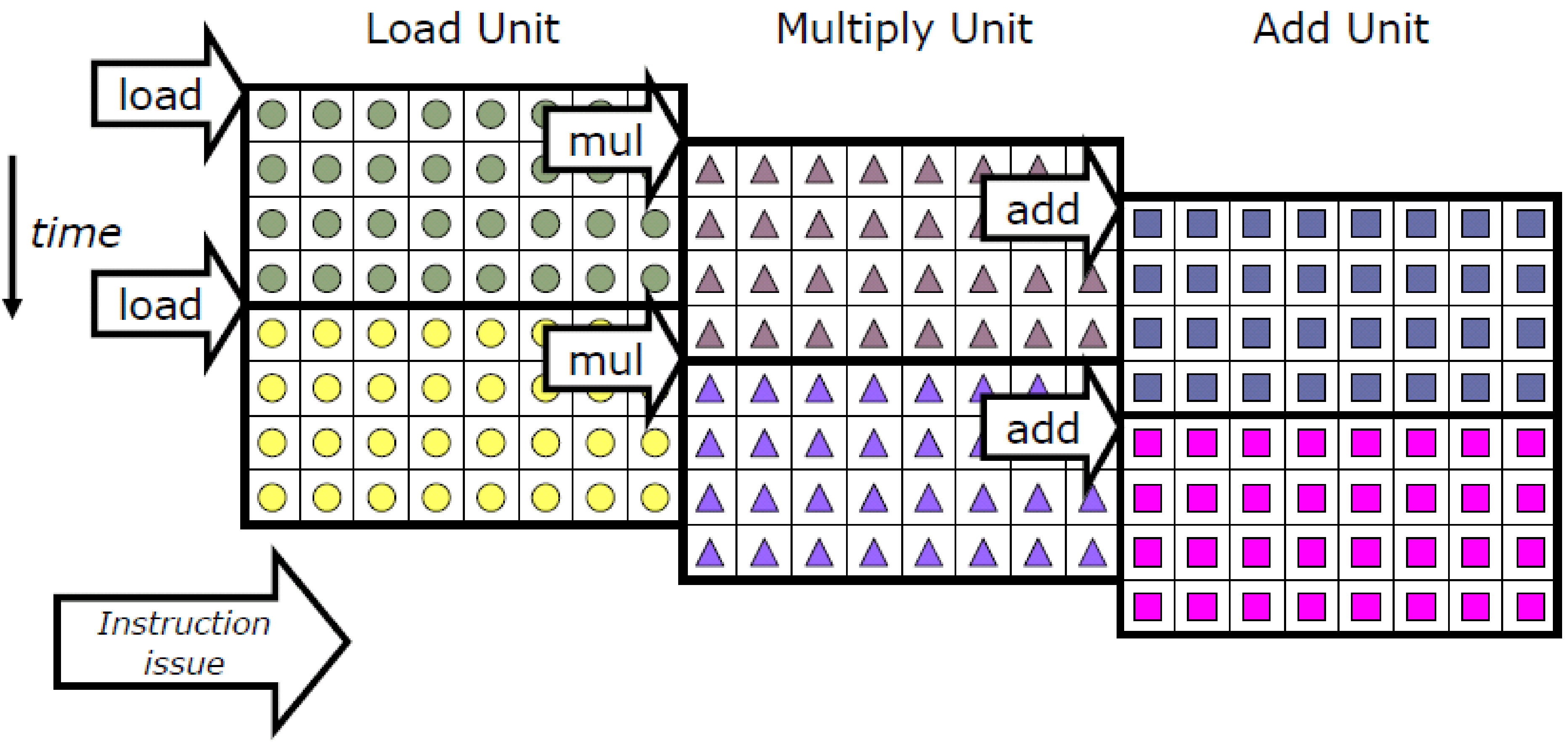
Vector Unit Structure



Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

- example machine has 32 elements per vector register and 8 lanes

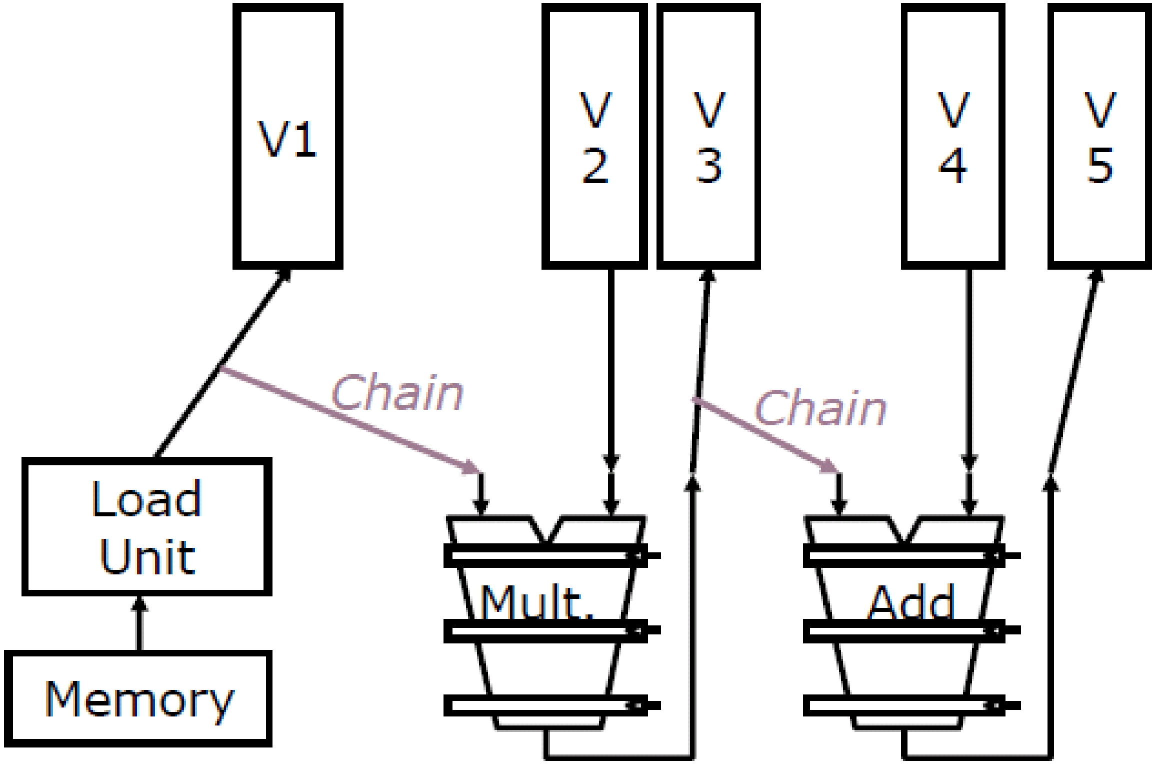


Complete 24 operations/cycle while issuing 1 short instruction/cycle

Vector Chaining

Problem: Long latency for RAW register dependencies

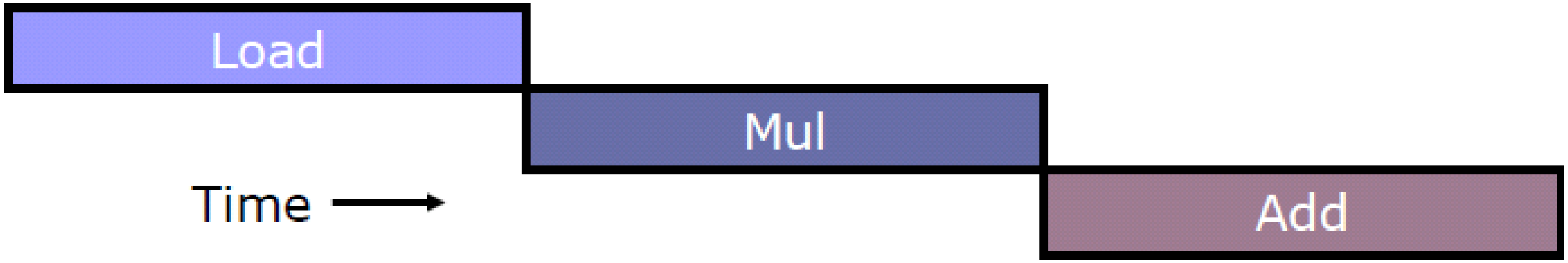
```
LV   v1
MULV v3,v1,v2
ADDV v5, v3, v4
```



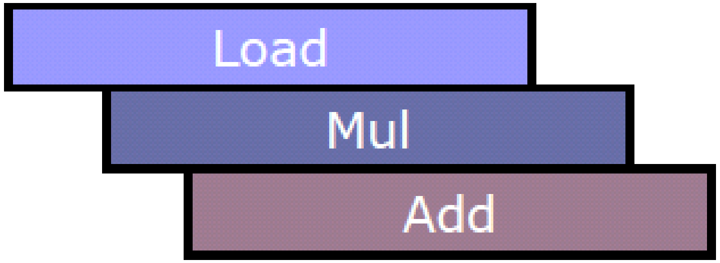
- Vector version of register bypassing
  - introduced with Cray-1

Vector Chaining Advantage

- Without chaining, must wait for last element of result to be written before starting dependent instruction



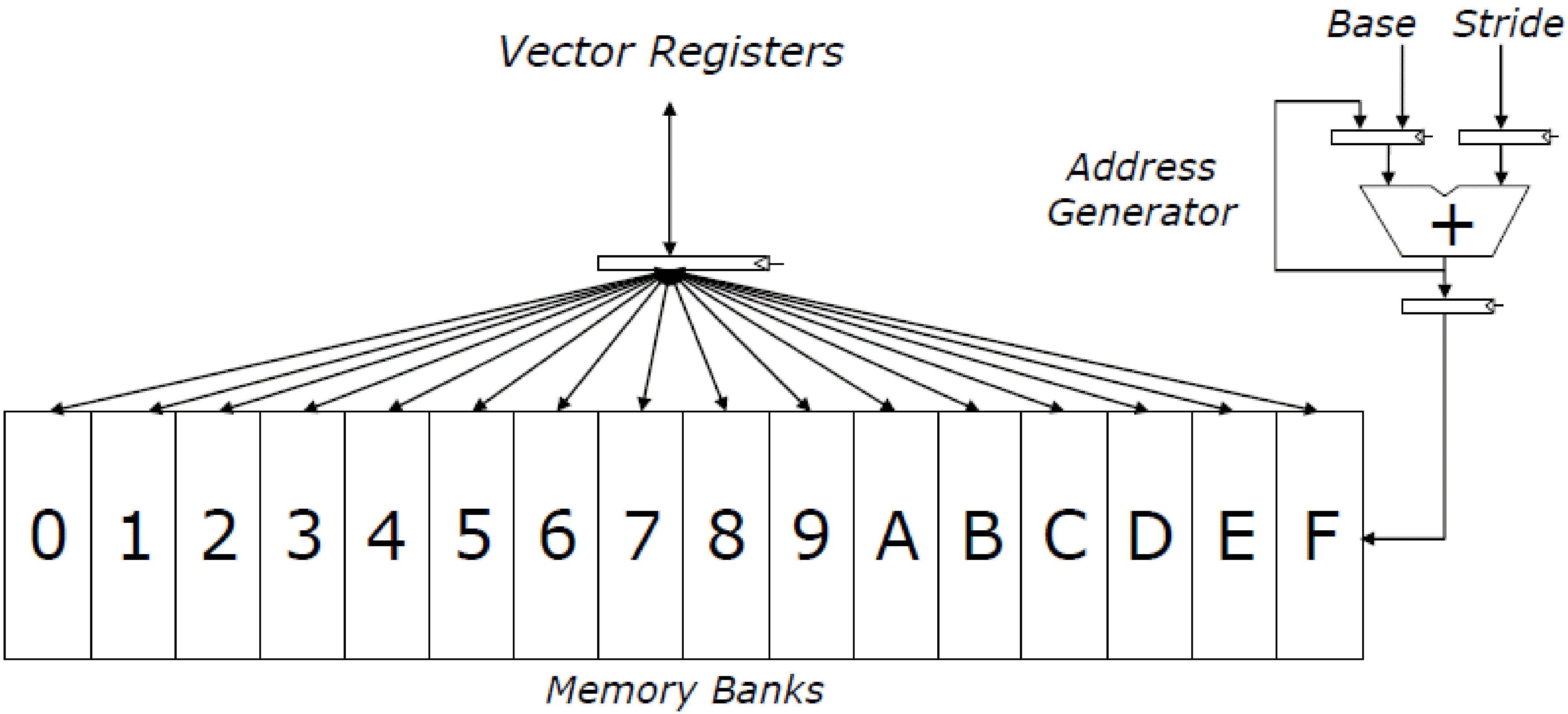
- With chaining, can start dependent instruction as soon as first result appears



Vector Memory System

Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Cycles between accesses to same bank
- Allows 16 parallel accesses (if data in different banks)

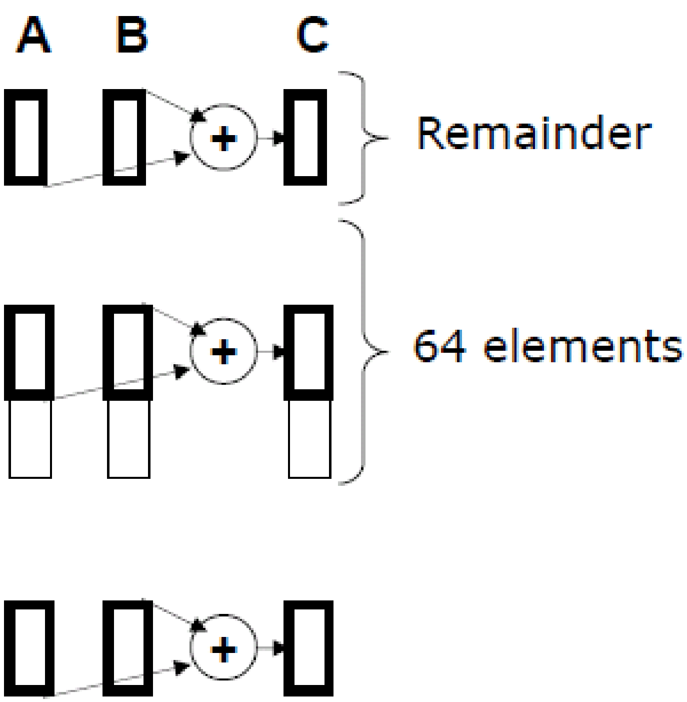


Vector Stripmining

Problem: Vector registers have finite length

Solution: Break loops into pieces that fit in registers, *"Stripmining"*

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




```
 ANDI R1, N, 63    # N mod 64
 MTC1 VLR, R1      # Do remainder
loop:
 LV V1, RA
 DSLL R2, R1, 3        # Multiply by 8
 DADDU RA, RA, R2 # Bump pointer
 LV V2, RB
 DADDU RB, RB, R2
 ADDV.D V3, V1, V2
 SV V3, RC
 DADDU RC, RC, R2
 DSUBU N, N, R1 # Subtract elements
 LI R1, 64
 MTC1 VLR, R1    # Reset full length
 BGTZ N, loop    # Any more to do?
```

Vector Conditional Execution

Problem: Want to vectorize loops with conditional code:

```
for (i=0; i<N; i++)
    if (A[i]>0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

...and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

Code example:

```
CVM             # Turn on all elements
LV vA, rA       # Load entire A vector
SGTVS.D vA, F0  # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

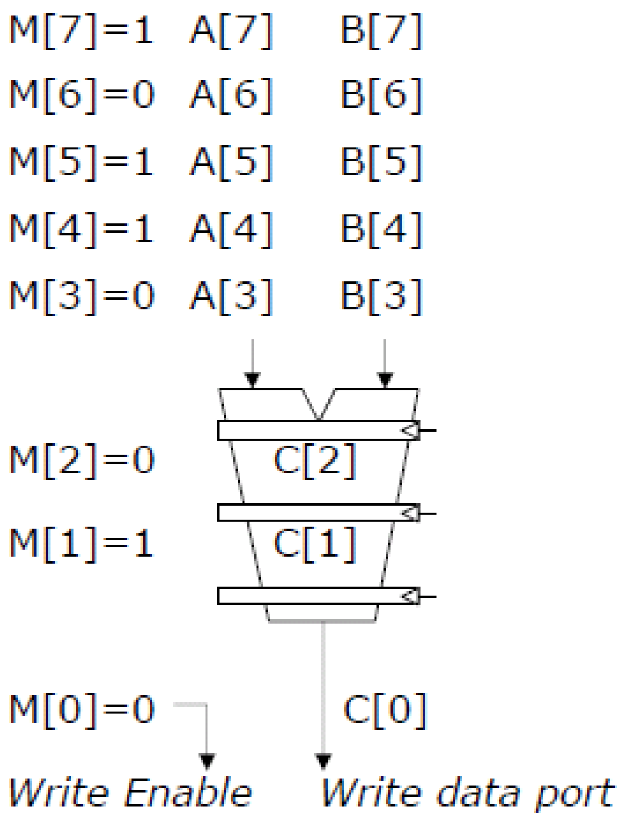
Masked Vector Instructions

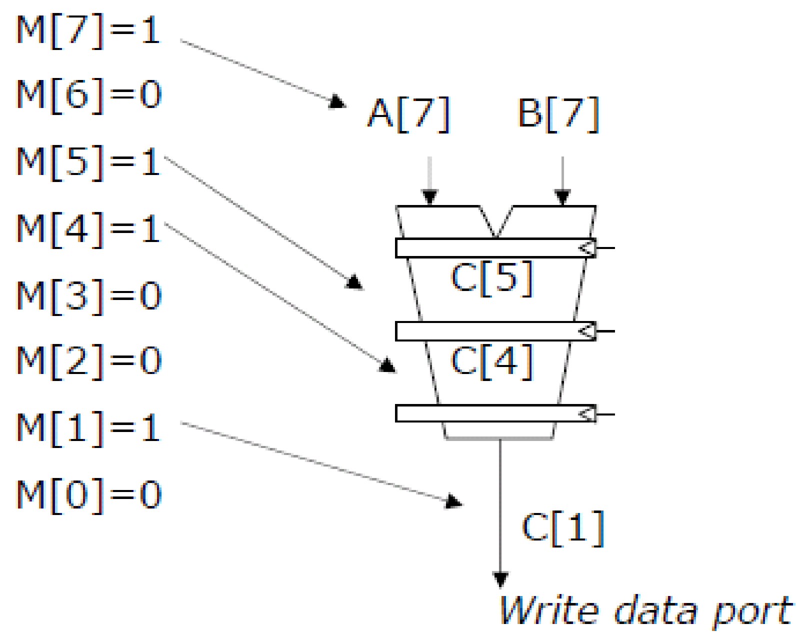
Simple Implementation

– execute all N operations, turn off result writeback according to mask

Density-Time Implementation

– scan mask vector and only execute elements with non-zero masks





Vector Scatter/Gather

Want to vectorize loops with indirect accesses:

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

Indexed load instruction (*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
```

Vector Scatter/Gather

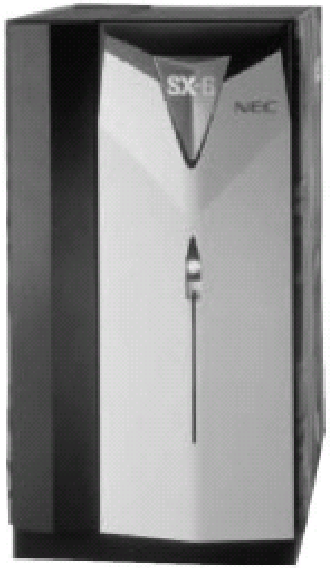
Scatter example:

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

Is following a correct translation?

```
LV vB, rB          # Load indices in B vector
LVI vA, rA, vB     # Gather initial A values
ADDV vA, vA, 1     # Increment
SVI vA, rA, vB     # Scatter incremented values
```

A Later Generation Vector Super:
NEC SX-6 (2003)



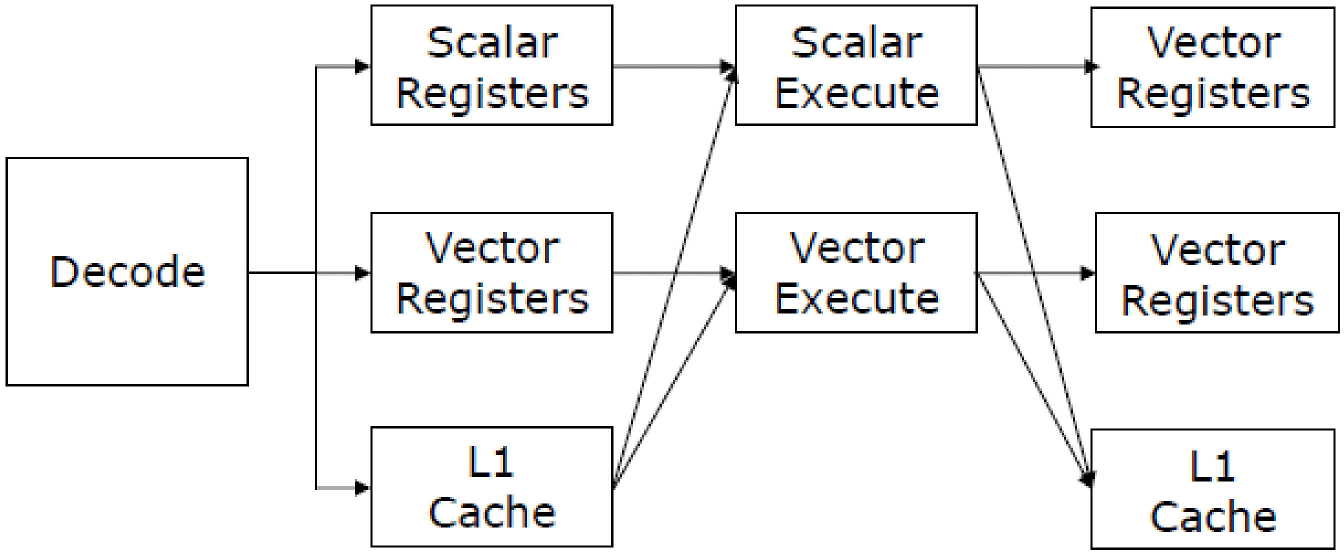
- CMOS Technology
  - 500 MHz CPU, fits on single chip
  - SDRAM main memory (up to 64GB)
- Scalar unit
  - 4-way superscalar
  - with out-of-order and speculative execution
  - 64KB I-cache and 64KB data cache
- Vector unit
  - 8 foreground VRegs + 64 background VRegs (256x64-bit elements/VReg)
  - 1 multiply unit, 1 divide unit, 1 add/shift unit, 1 logical unit, 1 mask unit
  - 8 lanes (8 GFLOPS peak, 16 FLOPS/cycle)
  - 1 load & store unit (32x8 byte accesses/cycle)
  - 32 GB/s memory bandwidth per processor
- SMP structure
  - 8 CPUs connected to memory through crossbar
  - 256 GB/s shared memory bandwidth (4096 interleaved banks)

Multimedia Extensions

- Short vectors added to existing general-purpose ISAs
- Initially, 64-bit registers split into 2x32b or 4x16b or 8x8b
- Limited instruction set:
  - No vector length control
  - No strided load/store or scatter/gather
  - Unit-stride loads must be aligned to 64-bit boundary
- Limited vector register length:
  - Requires superscalar dispatch to keep multiply/add/load units busy
  - Loop unrolling to hide latencies increases register pressure

- Trend towards fuller vector support in microprocessors
  - e.g. x86: MMX → SSEx (128 bits) → AVX (256 bits) → AVX-512 (512 bits)

Larrabee/Xeon Phi: x86 with vectors



- Short in-order instruction pipeline
- Separate scalar and vector units and register sets
  - Vector unit: 16 32-bit ops/clock
- Fast access to L1 cache
- L1 connects to core's portion of the L2 cache

Larrabee Vector Architecture

- Data types
  - Int32, Float32 and Float64 data
- Vector operations
  - Two input/one output operations
  - Full complement of arithmetic and media operations
    - Fused multiply-add (three input arguments)
  - Mask registers select lanes to write
  - Swizzle the vector elements on register read
- Memory access
  - Vector load/store including scatter/gather
  - Data replication on read from memory
  - Numeric type conversion on memory read

Larrabee Motivation

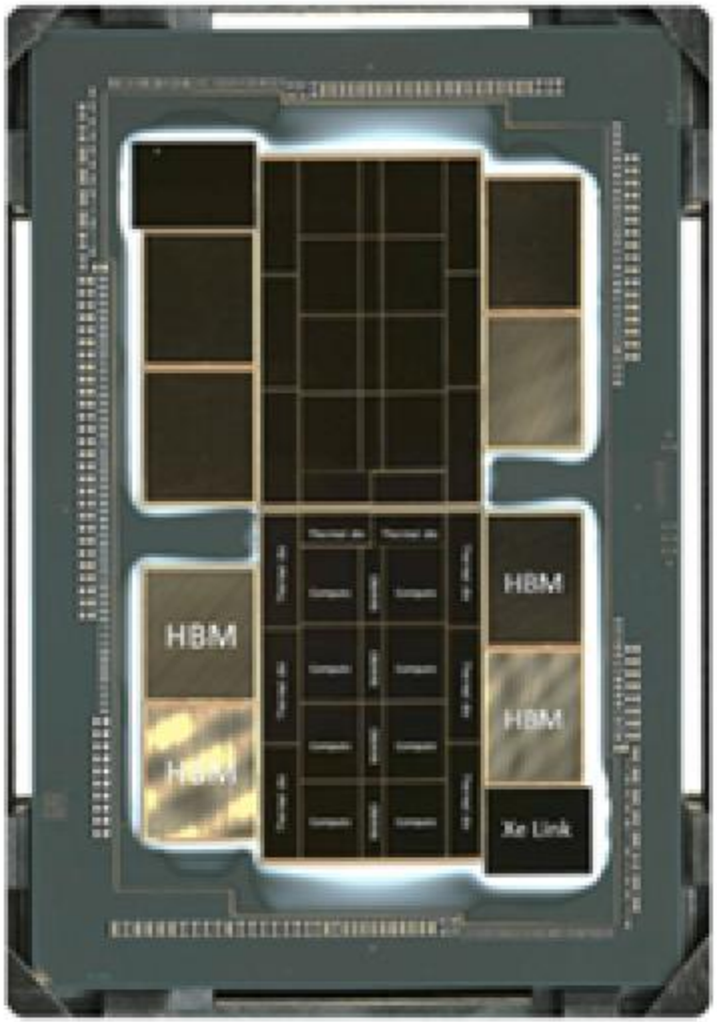
Design experiment: not a real 10-core chip!

# CPU cores	2 out of order	10 in-order
Instructions per issue	4 per clock	2 per clock
VPU lanes per core	4-wide SSE	16-wide
L2 cache size	4 MB	4 MB
Single-stream	4 per clock	2 per clock
Vector throughput	**8 per clock**	**160 per clock**

20 times the multiply-add operations per clock

Data in chart taken from Seiler, L., Carmean, D., et al. 2008. *Larrabee: A many-core x86 architecture for visual computing*.

向量计算机与GPU



ISSCC 2022国际固态电路会议上，Intel介绍了即将推出的面向超算的Ponte Vecchio计算加速卡，使用了5种不同的制造工艺，内部封装多达47个芯片/单元(Tile)，晶体管数量突破1000亿个，Ponte Vecchio整体面积4844mm^2，4468个引脚，整体功耗达到600W。

向量计算机与GPU

Nvidia下一代GPU架构"Hopper"和"ADA"，使用TSMC的4nm工艺，用于支持AI计算。ADA 102面积约611.3mm^2，功耗也是600W！