Vector Processing and Architectures

Guang R. Gao

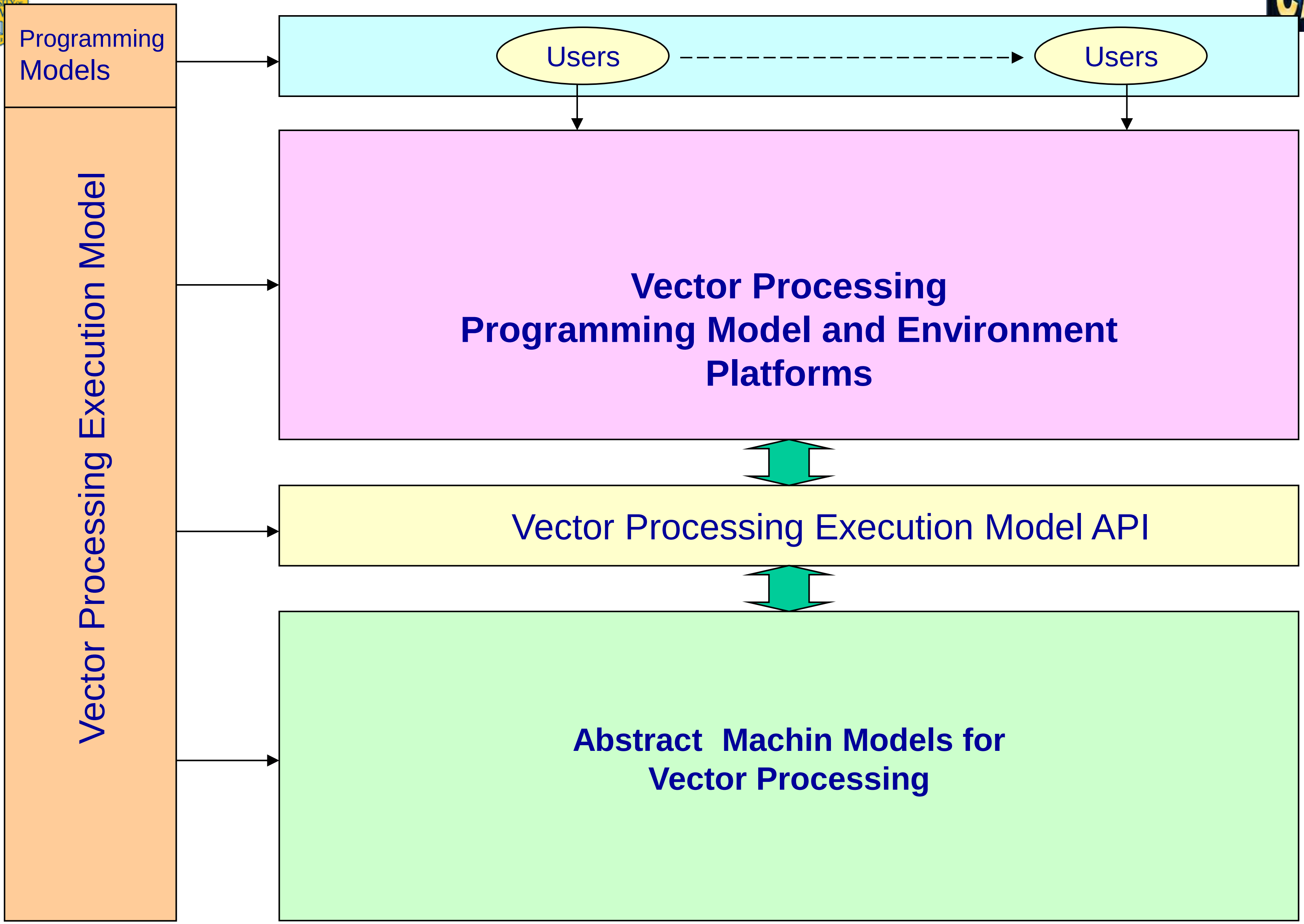
ACM Fellow and IEEE Fellow
Endowed Distinguished Professor
Electrical & Computer Engineering
University of Delaware
ggao@capsl.udel.edu

Reading List

- Slides.
- Henn&Patt: Chapter 4th, 5th Edition *(may change depending on your book's version).*
- Other assigned readings from homework and classes

Outline

- Introduction
- Vector Processing Model and Architectures
- Cray Example
- Performance Model
- Summary


Programming Models
Vector Processing Execution Model
Users
Users
Vector Processing
Programming Model and Environment
Platforms
Vector Processing Execution Model API
Abstract Machin Models for
Vector Processing

Execution Model and Abstract Machines

Vector Architectures
(a successful SIMD class architectures)

Types:

Register-Register Archs

Memory-Memory Archs

Vector Arch Components:

Vector Register Banks

Capable of holding a n number of vector elements. Two extra registers

Vector Functional Units

Fully pipelined, hazard detection (structural and data)

Vector Load-Store Unit

A Scalar Unit

A set of registers, FUs and CUs

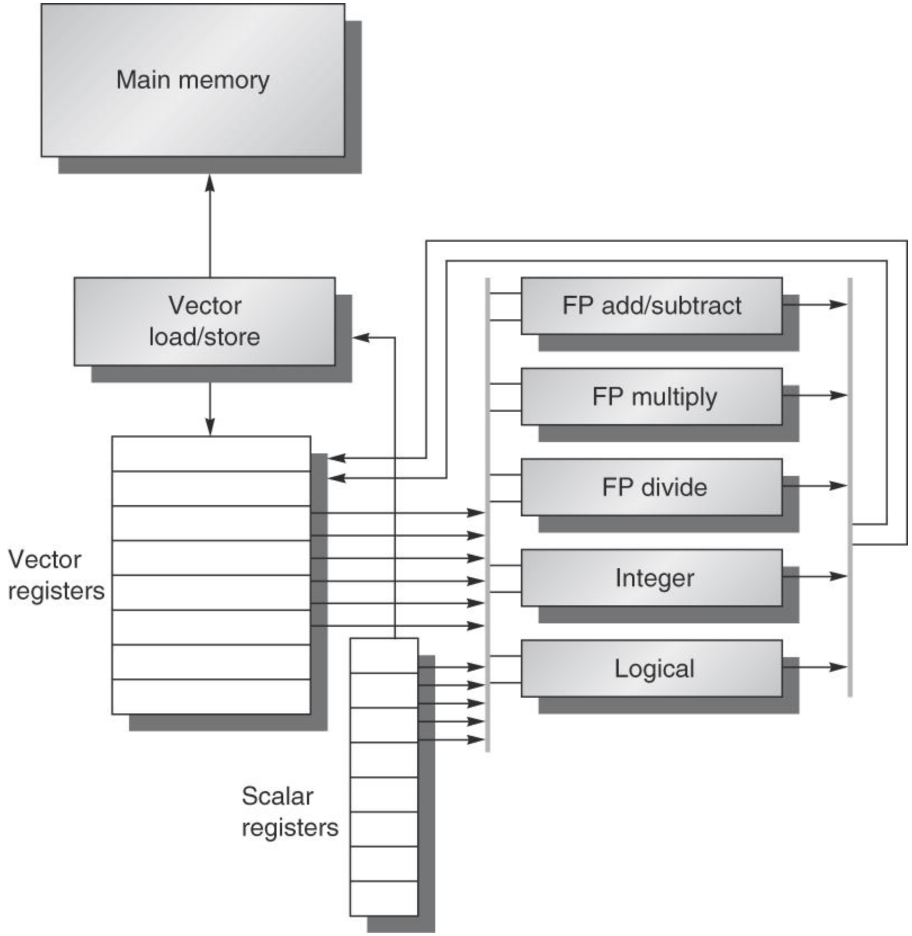


Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

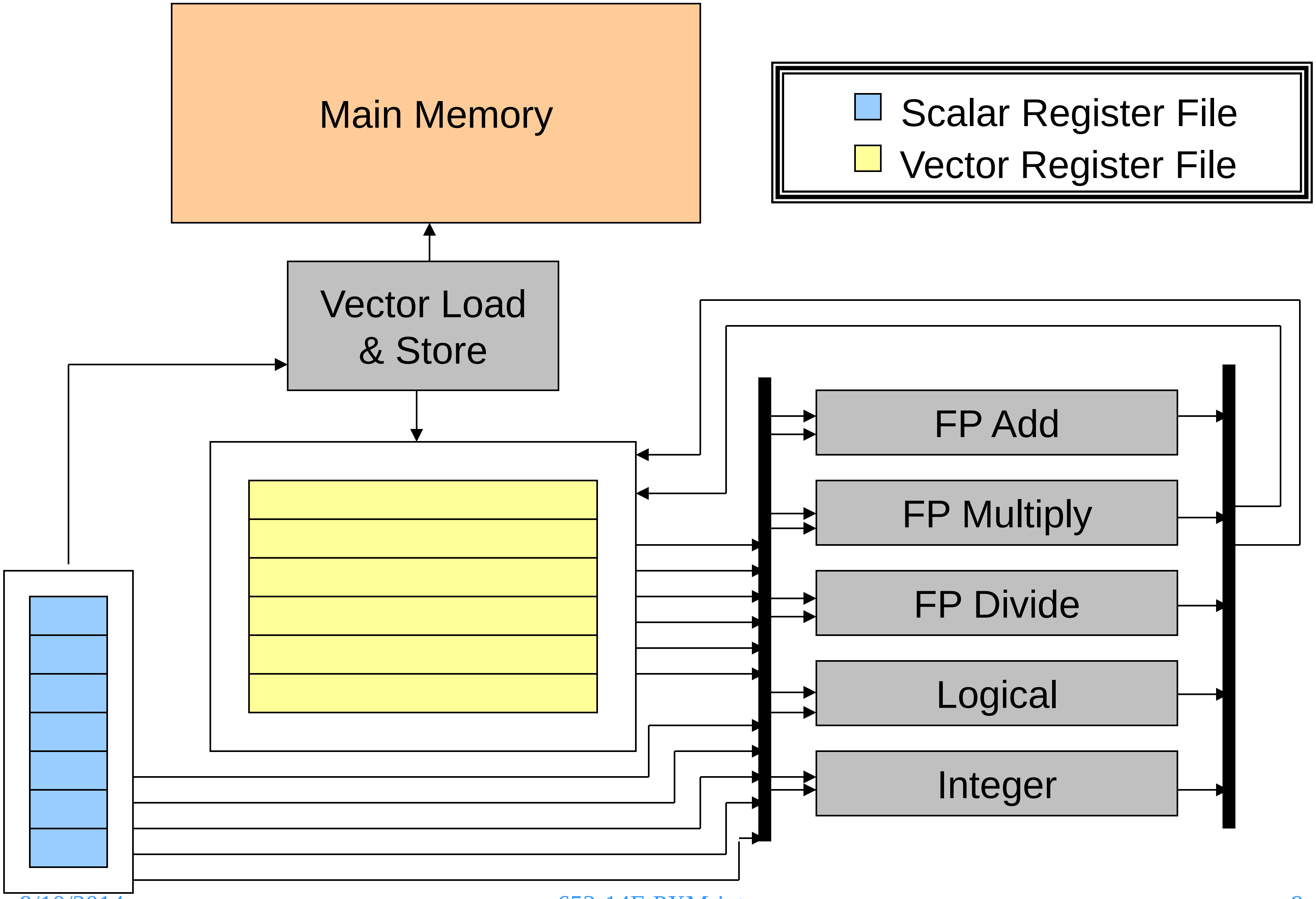
An Intro to DLXV

- A simplified vector architecture
- Consist of one *lane* per functional unit
  - ***Lane:*** The number of vector instructions that can be executed in parallel by a functional unit
- Loosely based on Cray 1 architecture and ISA
- Extension of DLX ISA for vector architecture

DLXV Configuration

- Vector Registers
  - Eight Vector regs / 64 element each.
  - Two read ports and one write port per register
  - Sixteen read ports and eight write ports in total
- Vector Functional Unit
  - Five Functional Units
- Vector Load and Store Unit
  - A bandwidth of 1 word per cycle
  - Double as a scalar load / store unit
- A set of scalar registers
  - 32 general and 32 FP regs

A Vector / Register Arch



Advantages

- A single vector instruction ➔ A lot of work
- No data hazards
  - No need to check for data hazards inside vector instructions
  - Parallelism inside the vector operation
    - Deep pipeline or array of processing elements
- Known Access Pattern
  - Latency only paid once per vector (pipelined loading)
  - Memory address can be mapped to memory modules to reduce contentions
- Reduction in code size and simplification of hazards
  - Loop related control hazards from loop are eliminated.

DAXPY: DLX Code

Y = a * X + Y

```
           LD        F0, a
           ADDI      R4, Rx, #512    ; last address to load
Loop:      LD        F2, 0(Rx)       ; load X(i)
           MULTD     F2, F0, F2      ; a x X(i)
           LD        F4, 0 (Ry)      ; load Y(i)
           ADDD      F4, F2, F4      ; a  x  X(i) + Y(i)
           SD        F4, 0 (Ry)      ; store into Y(i)
           ADDI      Rx, Rx, #8      ; increment index to X
           ADDI      Ry, Ry, #8      ; increment index to Y
           SUB       R20, R4, Rx     ; compute bound
           BNZ       R20, loop       ; check if done
```

The bold instructions are part of the loop index calculation and branching

DAXPY: DLXV Code

Y = a * X + Y

```
LD        F0, a            ; load scalar a
LV        V1, Rx           ; load vector X
MULTSV    V2, F0, V1       ; vector-scalar multiply
LV        V3, Ry           ; load vector Y
ADDV      V4, V2, V3       ; add
SV        Ry, V4           ; store the result
```

Instruction Number [Bandwidth] for 64 elements

DLX Code	578 Instructions
DLXV Code	6 Instructions

Some Issues

- Vector Length Control
  - Vector lengths are not usually less or even a multiple of the hardware vector length
- Vector Stride
  - Access to vectors may not be consecutively.
- Solutions:
  - Two special registers
    - One for vector length up to a maximum vector length
    - One for vector mask

Vector Length Control

An Example:

```
for(i = 0; i < n; ++i)
   y[i] = a * x[i] +y[i]
```

Question: Assume the maximum hardware vector length is *MVL* which may be less than n. How should we do the above computation ?

Vector Length Control
Strip Mining

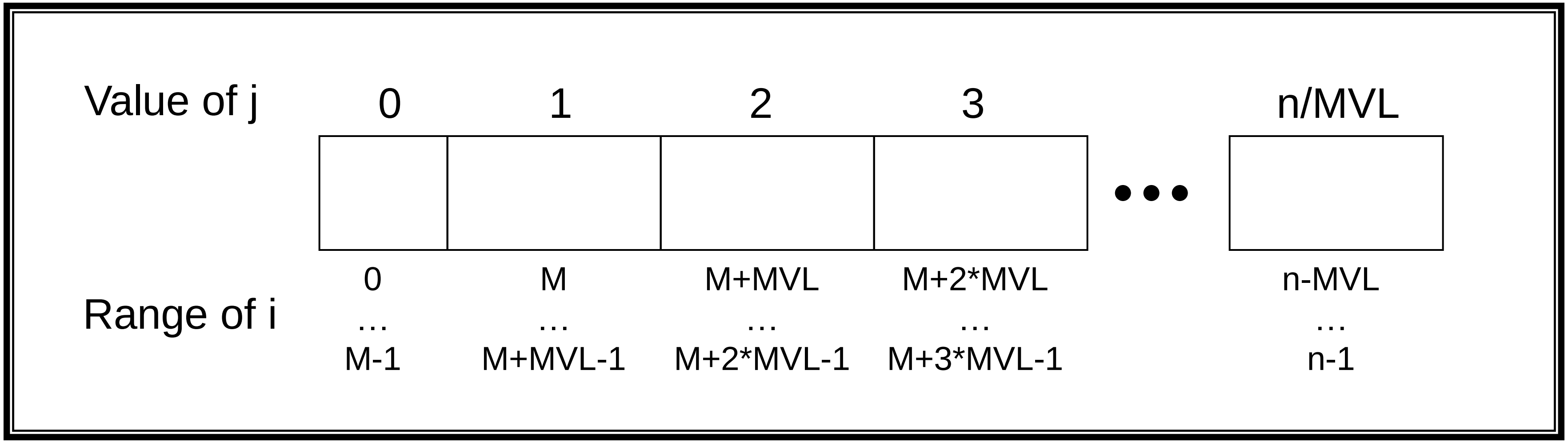
Original Code

```
for(i = 0; i < n; ++i)
   y[i] = a * x[i] +y[i]
```

Strip Mined Code

```
Low = 0
VL = n % MVL
for(j = 0; j <= (n /MVL); ++j){
   for(i = Low; i < Low + VL – 1; ++i)
       y[i] = a * x[i] + y[i];
   Low += VL;
   VL = MVL;
}
```

Vector Length Control
Strip Mining



Value of j	0	1	2	3		n/MVL
Range of i	0 … M-1	M … M+MVL-1	M+MVL … M+2*MVL-1	M+2*MVL … M+3*MVL-1	• • •	n-MVL … n-1

For a vector of arbitrary length VL = M = n % MVL

The vector length control register takes values similar to the Vector Length variable **(VL)** in the C code

Vector Stride

Matrix Multiply Code

```
for(i = 0; i < n; ++i)
   for(j = 0; j < n; ++j){
       c[i][j] = 0.0;
       for(k = 0; k < n; ++k)
            c[i][j] += a[i][k] * b[k][j];
   }
```

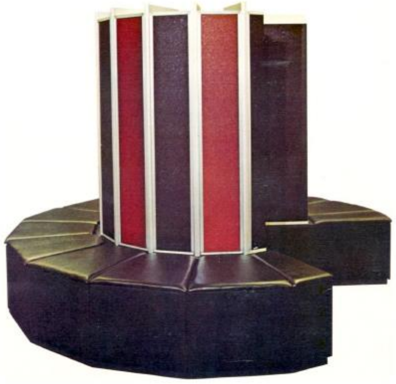
How to vectorize this code?

How stride works here?

Consider that in C the arrays are saved in memory row-wise. Therefore, a and c are loaded correctly. How about b?

Cray-1
“The World Most Expensive Love Seat...”

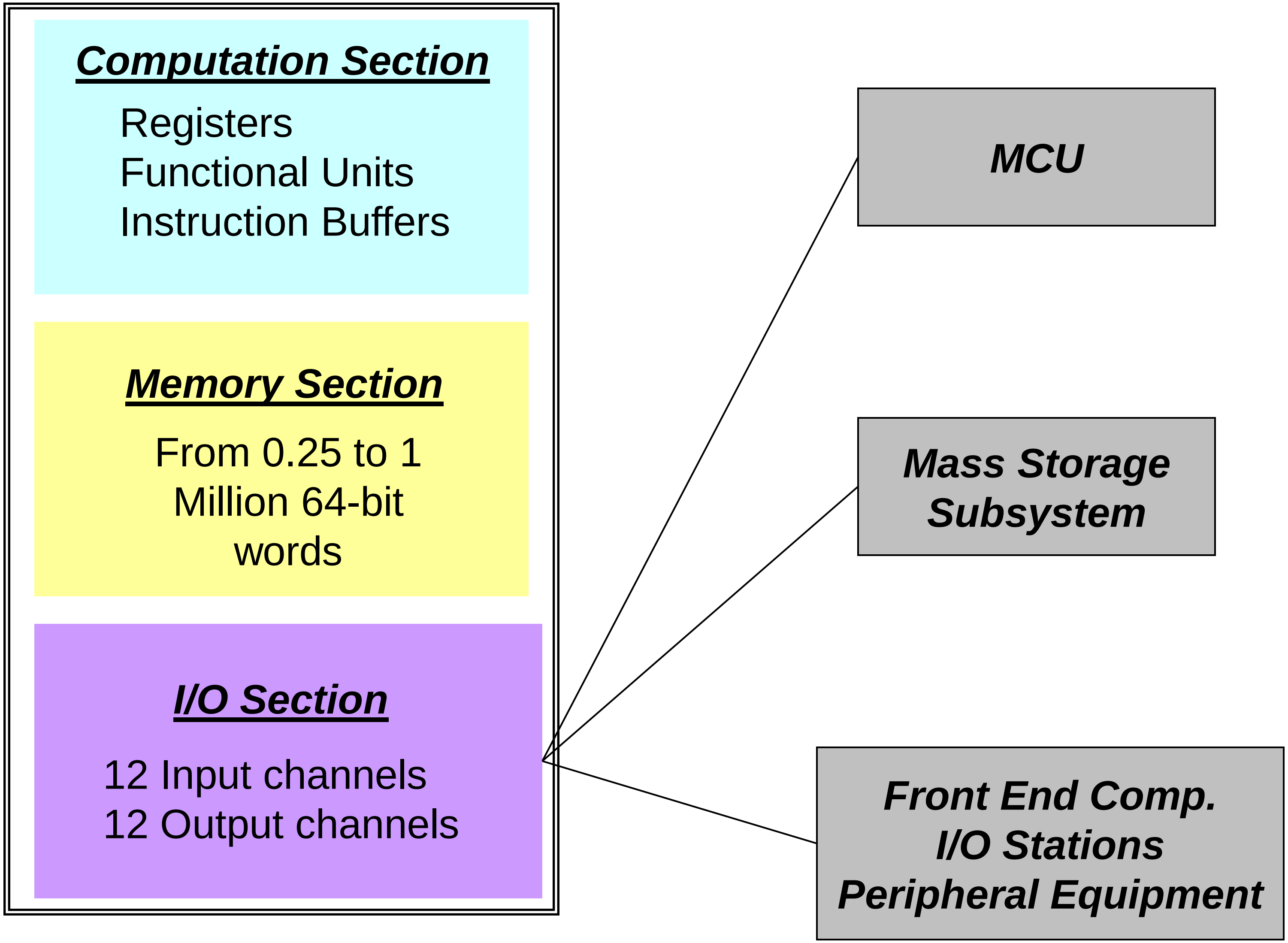
Picture courtesy of Cray
Original Source: Cray 1 Computer System Hardware Reference Manual



Cray 1 Data Sheet

- Designed by: Seymour Cray
- Price: 5 to 8.8 Millions dollars
- Units Shipped: 85
- Technology: SIMD, deep pipelined functional units
- Performance: up to 160 MFLOPS
- Date Released: 1976
- Best Known for: The computer that made the term *supercomputer* mainstream

Architectural Components



Computation Section

Registers
Functional Units
Instruction Buffers

Memory Section

From 0.25 to 1 Million 64-bit words

I/O Section

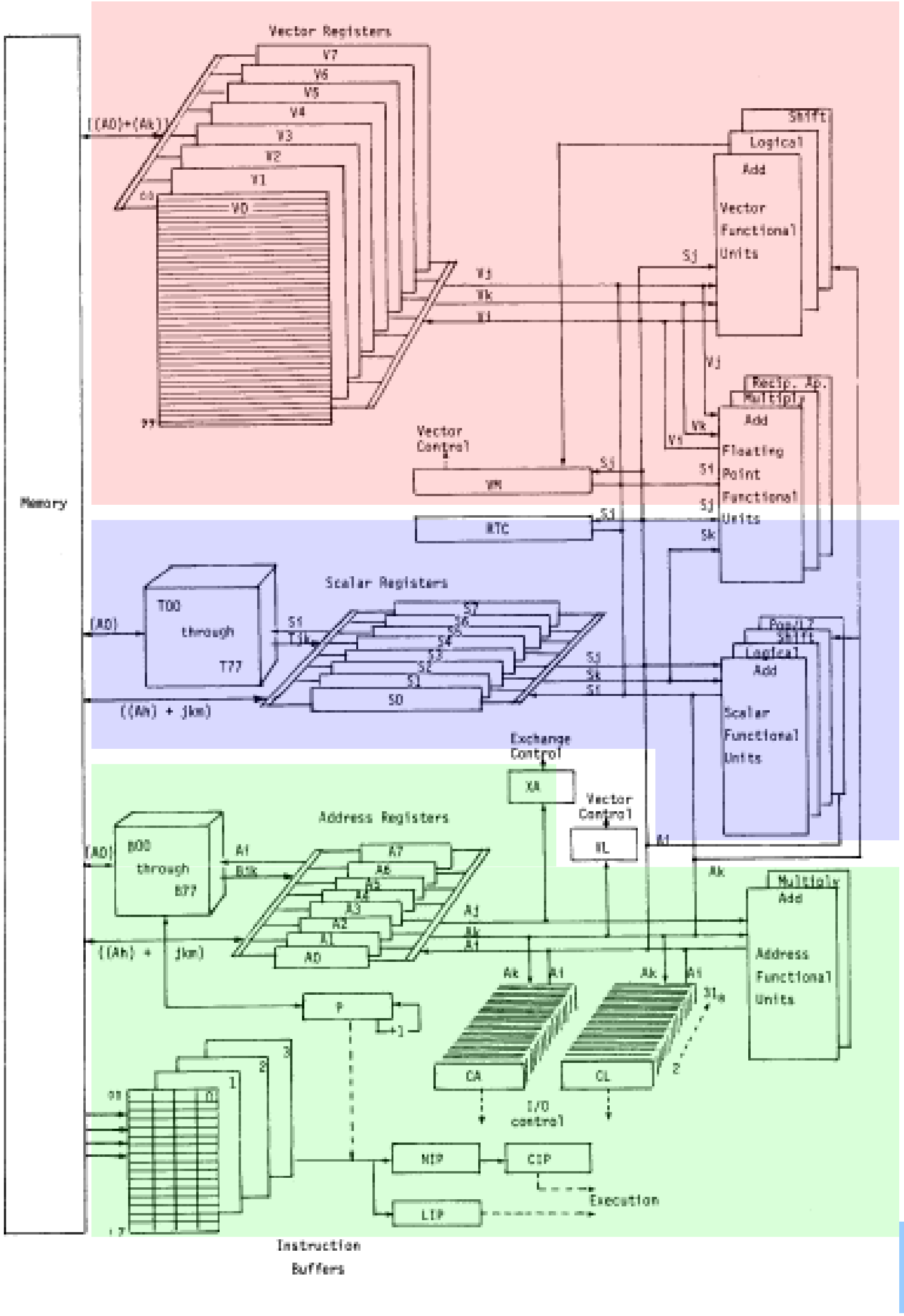
12 Input channels
12 Output channels

MCU

Mass Storage Subsystem

Front End Comp.
I/O Stations
Peripheral Equipment

The Cray-1 Architecture



Vector Components

Scalar Components

Address & Instruction Calculation Components

Computation Section

Register-Register Architecture

- All ALU operands are in registers
- Registers are specialized by function (A, B, T, etc) thus avoiding conflict
- Transfer between Memory and registers is treated differently than ALU
- RISC based idea
- Effective use of the Cray-1 requires careful planning to exploit its register resources
  - 4 Kbytes of high speed registers

Registers

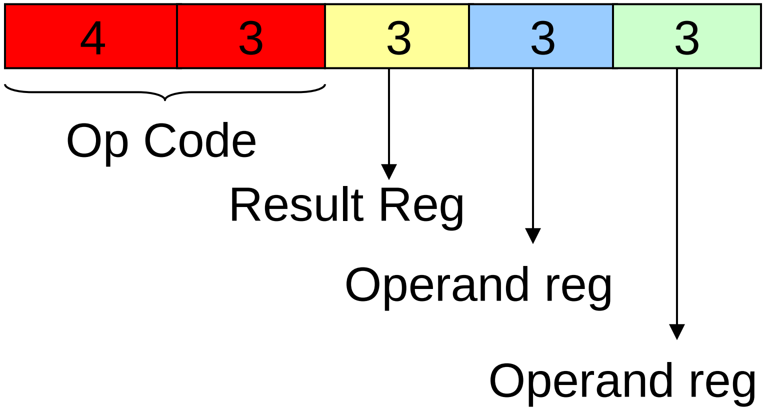
- ***Memory Access Time:*** 11 cycles
- ***Register Access Time:*** 1 ~ 2 cycles
- ***Primary Registers:***
  - Address Regs: 8 x 24 bits
  - Scalar Regs: 8 x 64 bits
  - Vector Regs: 8 x 64 words
- ***Intermediate Registers:***
  - B Regs: 64 x 24 bits
  - T Regs: 64 x 64 bits
- ***Special Registers:***
  - Vector Length Register: 0 <= VL <= 64
  - Vector Masks Register: 64 bits
- ***Total Size***: 4,888 bytes

Instruction Format

A ***parcel*** → 16-bit

Instruction word 16 (one parcel) or 32 (two parcels) according to type

A One Parcel Instruction:
Arithmetic Logical Instruction Word



4 3 3 3 3

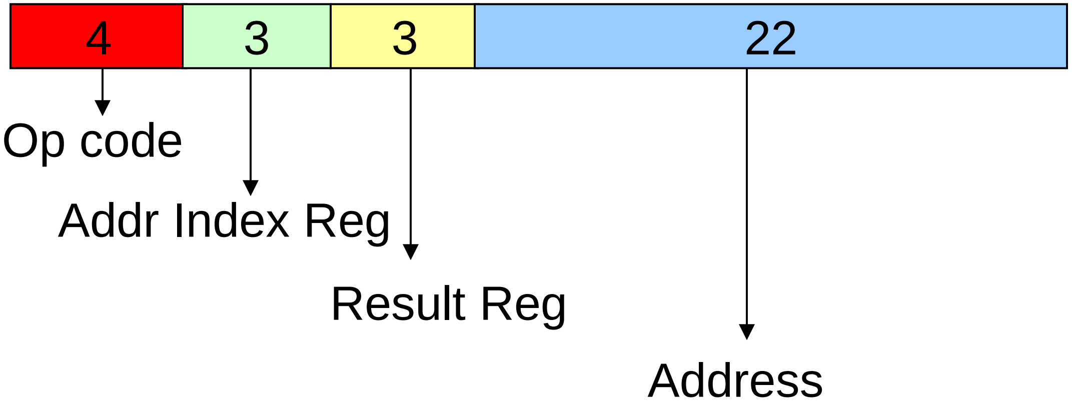
Op Code

Result Reg

Operand reg

Operand reg

A Two Parcels Instruction:
Memory Instruction Word



4 3 3 22

Op code

Addr Index Reg

Result Reg

Address

Functional Unit Pipelines

Functional pipelines	Register usage	Pipeline delays (clock periods)
Address functional units		
Address add unit	A	2
Address multiply unit	A	6
Scalar functional units		
Scalar add unit	S	3
Scalar shift unit	S	2 or 3
Scalar logical unit	S	1
Population/leading zero count unit	S	3
Vector functional units		
Vector add unit	V or S	3
Vector shift unit	V or S	4
Vector logical unit	V or S	2
Floating-point functional units		
Floating-point add unit	S and V	6
Floating-point multiply unit	S and V	7
Reciprocal approximation unit	S and V	14

Implementation Philosophy

- Instruction Processing
  - Instruction Buffering: Four Instructions buffers of 64 16-bit parcels each
- Memory Hierarchy
  - Memory Banks, T and B register banks
- Register and Function Unit Reservation
  - Example: Vector ops, register operands, register result and FU are checked as reserved
- Vector Processing

Instruction Processing

"Issue one instruction per cycle"

- 4 x 64 word
- 16 - 32 bit instructions
- Instruction parcel pre-fetch
- Branch in buffer
- 4 inst/cycle fetched to LRU I-buffer

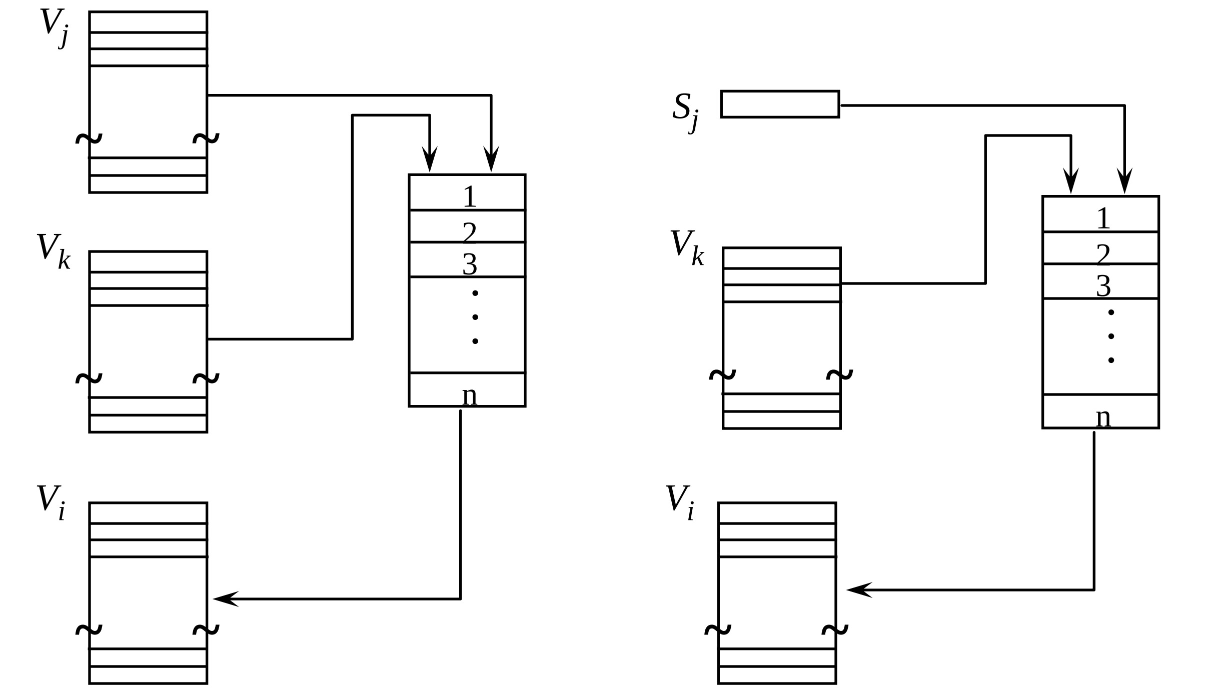
Reservations

- Vector operands, results and functional unit are marked reserved
- The vector result reservation is lifted when the chain slot time has passed
  - Chain Slot: Functional Unit delay plus two clock cycles

Examples:

V1 = V2 * V3	V1 = **V2** * V3	V1 = V2 ***** V3
V4 = V5 + V6	V4 = V5 + **V2**	V4 = V5 ***** V6
Independent	Second Instruction cannot begin until First is finished	Resource Dependency

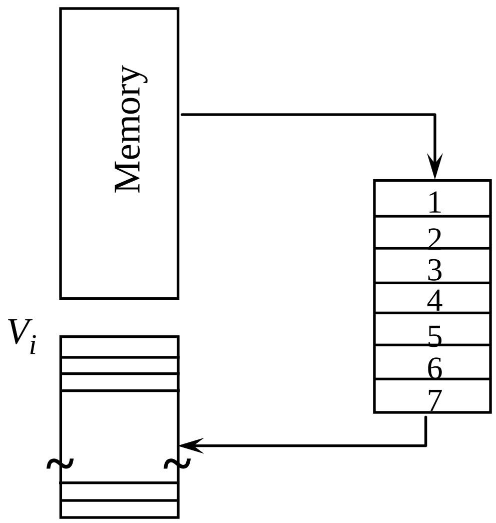
Vector Instructions in the Cray-1



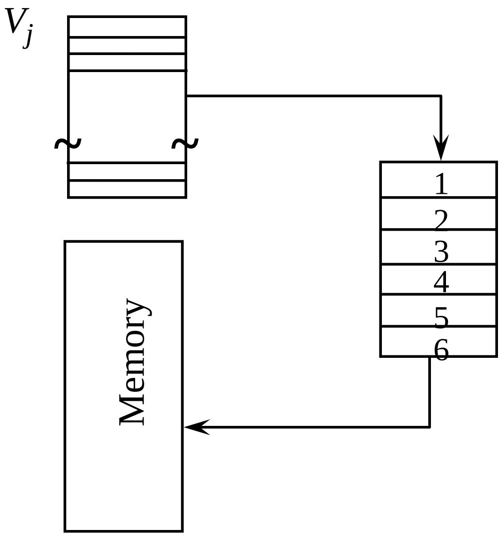
(a) Type 1 vector instruction

(b) Type 2 vector instruction

Vector Instructions in the Cray-1



(c) Type 3 vector instruction



(d) Type 4 vector instruction

Vector Loops

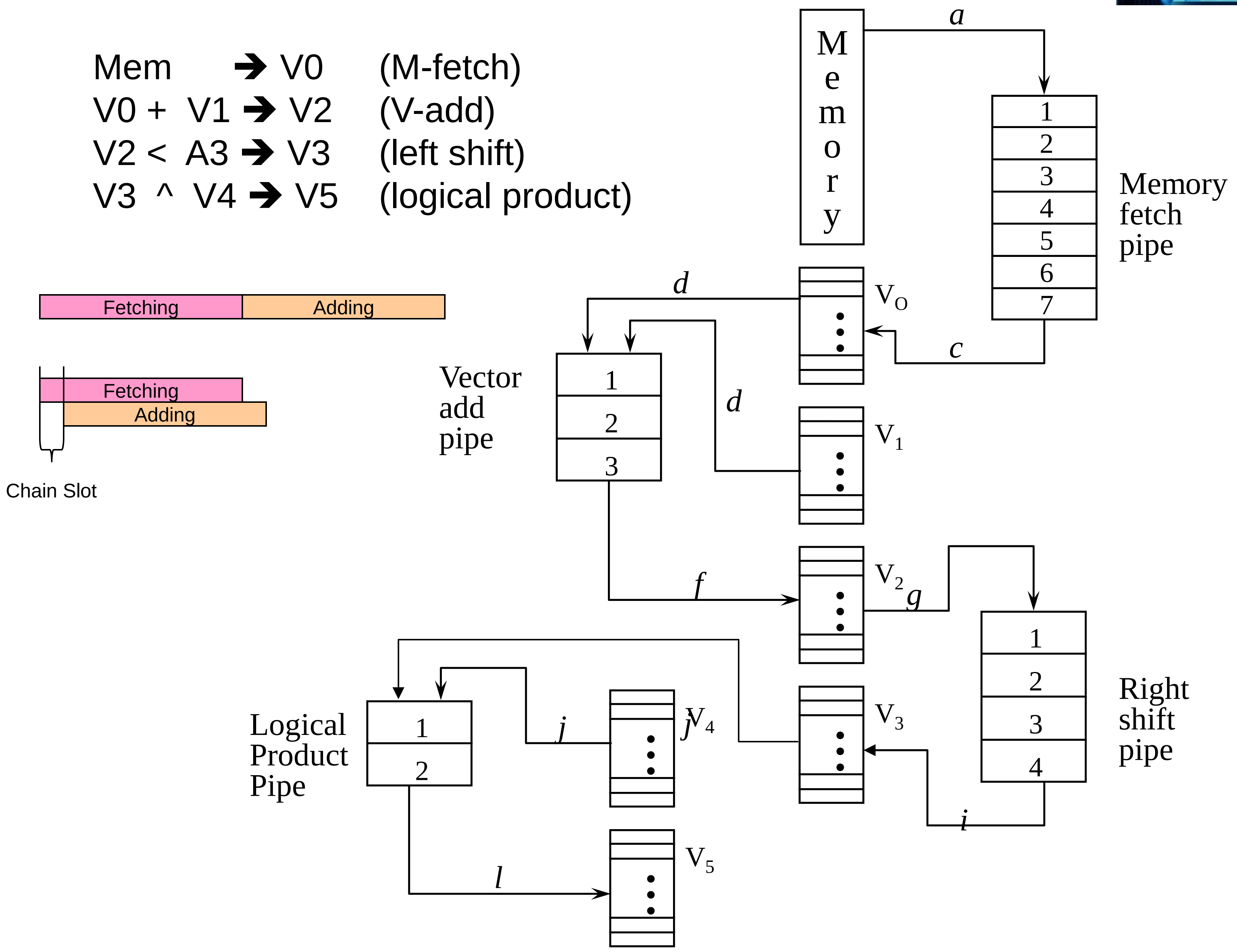
- Long vectors with N > 64 are sectioned
- Each time through a vector loop 64 elements are processed
- Remainder handling
- “transparent” to the programmer

Vector Chaining

- Internal forwarding techniques of IBM 360/91
- A “linking process” that occurs when results obtained from one pipeline unit are directly fed into the operand registers of another function pipe.
- Chaining allow operations to be issued as soon as the first result becomes available
- Registers/F-units must be properly reserved.
- Limited by the number of Vector Registers and Functional Units
- From 2 to 5

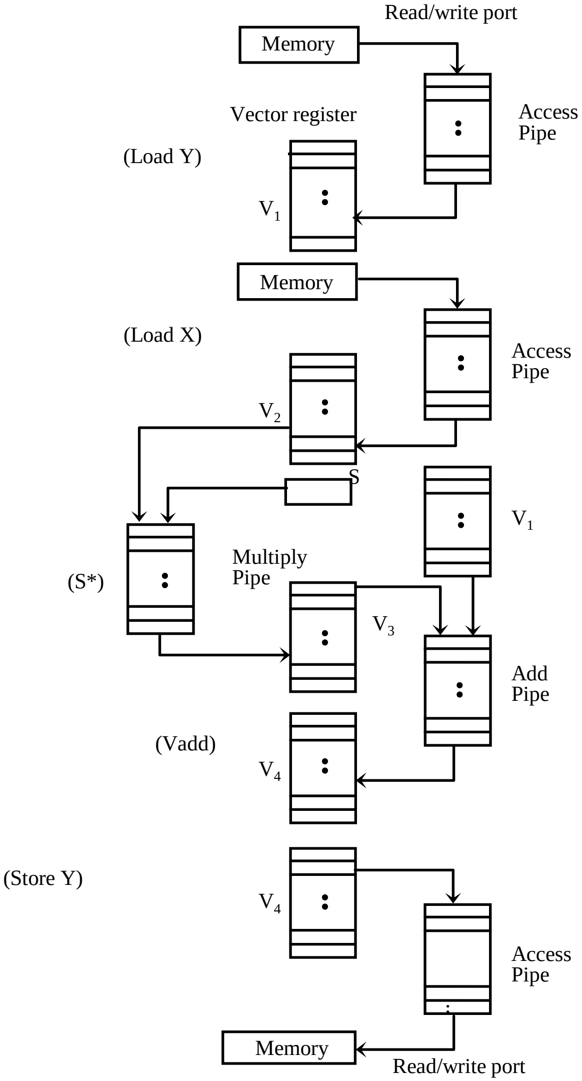
Chaining Example

Mem → V0 (M-fetch)
V0 + V1 → V2 (V-add)
V2 < A3 → V3 (left shift)
V3 ^ V4 → V5 (logical product)

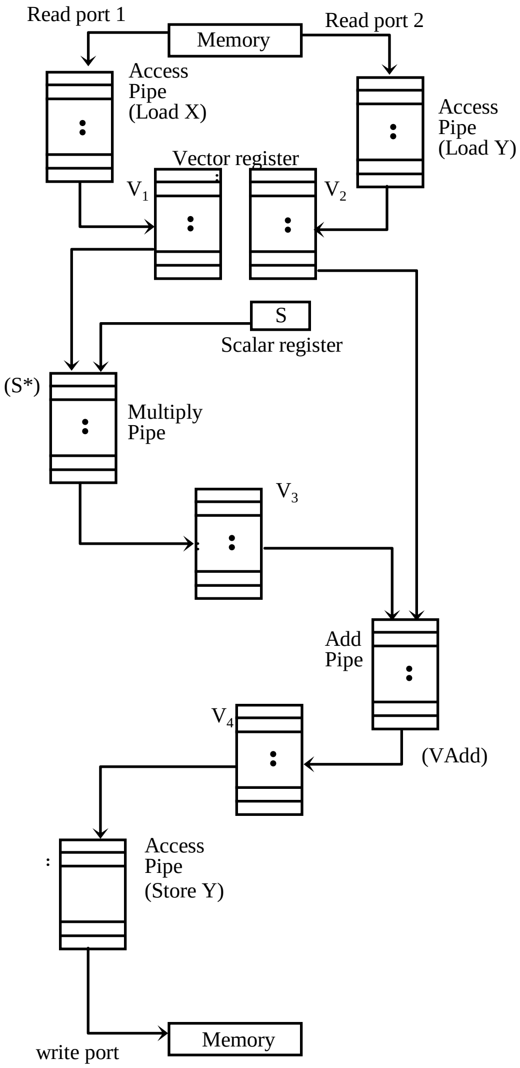


Y(1:N) = S x X(1:N) + Y(1:N)

Multipipeline chaining SAXPY code



Limited chaining using only one memory-access pipe in the Gray 1



Complete chaining using three memory-access pipes in the Cray X-MP

Cray 1 Performance

- 3 to 160 MFLOPS
  - Application and Programming Skills
- Scalar Performance: 12 MFLOPS
- Vector Dot Product: 22 MFLOPS
- Peak Performance: 153 MFLOPS

Irregular Vector Ops

- Scatter: Use a vector to scatter another vector elements across Memory
  - X[A[i]] = B[i]
- Gather: The reverse operation of scatter
  - X[i] = B[C[i]]
- Compress
  - Using a Vector Mask, compress a vector
- No Single instruction to do these before 1984
  - Poor Performance: 2.5 MFLOPS

Gather Operation

V1[i] = A[V0[i]]

VL	4
A0	100

V0	V1
4	
2	
7	
0	
⋮	⋮

Memory Contents / Addresses

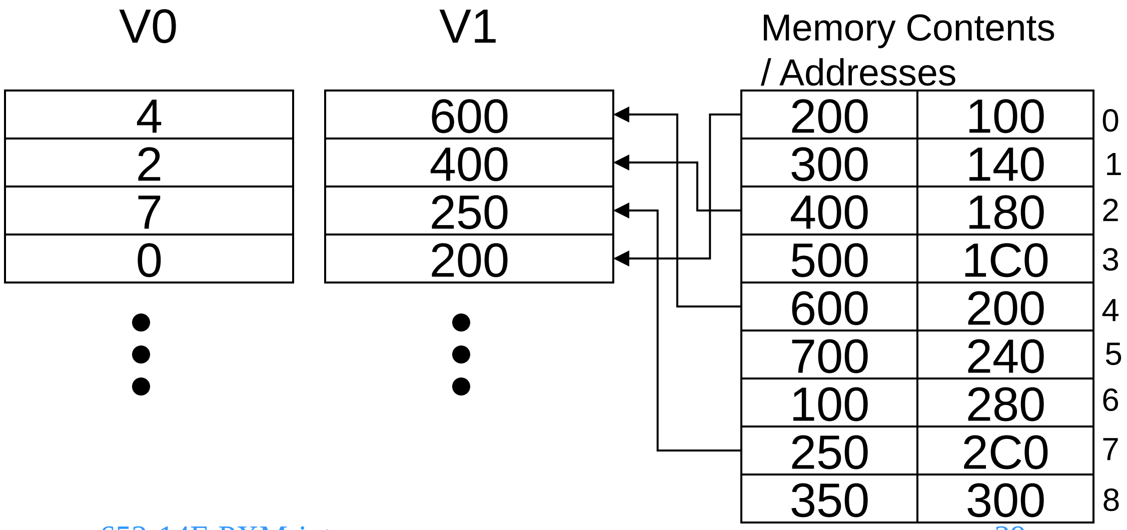
200	100
300	140
400	180
500	1C0
600	200
700	240
100	280
250	2C0
350	300

Example:

V1[2] = A[V0[2]]

= A[7]

= 250



V0	V1
4	600
2	400
7	250
0	200
⋮	⋮

Memory Contents / Addresses

200	100	0
300	140	1
400	180	2
500	1C0	3
600	200	4
700	240	5
100	280	6
250	2C0	7
350	300	8

Scatter Operation

A[V0[i]] = V1[i]

VL	4
A0	100

V0	V1
4	200
2	300
7	400
0	500
⋮	⋮

Memory Contents
/ Addresses

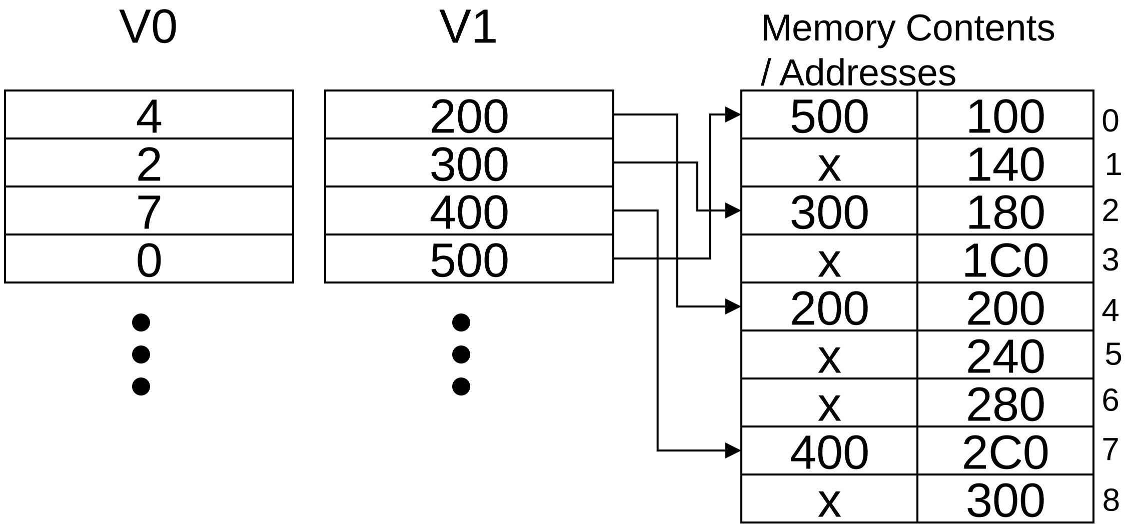
x	100
x	140
x	180
x	1C0
x	200
x	240
x	280
x	2C0
x	300

Example:

A[V0[0]] = V1[0]

A[4]   = 200

*(0x200)= 200



V0	V1
4	200
2	300
7	400
0	500
⋮	⋮

Memory Contents
/ Addresses

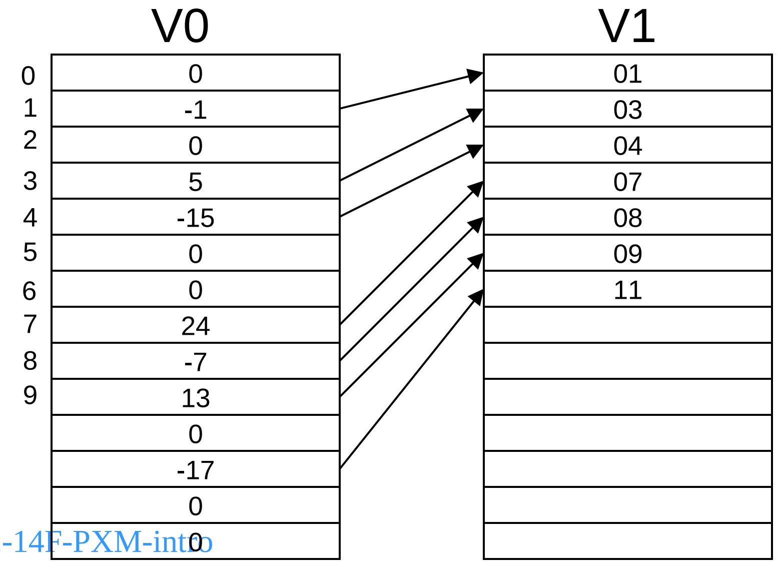
500	100	0
x	140	1
300	180	2
x	1C0	3
200	200	4
x	240	5
x	280	6
400	2C0	7
x	300	8

Vector Compression Operation

VL	14
VM	010110011101 …

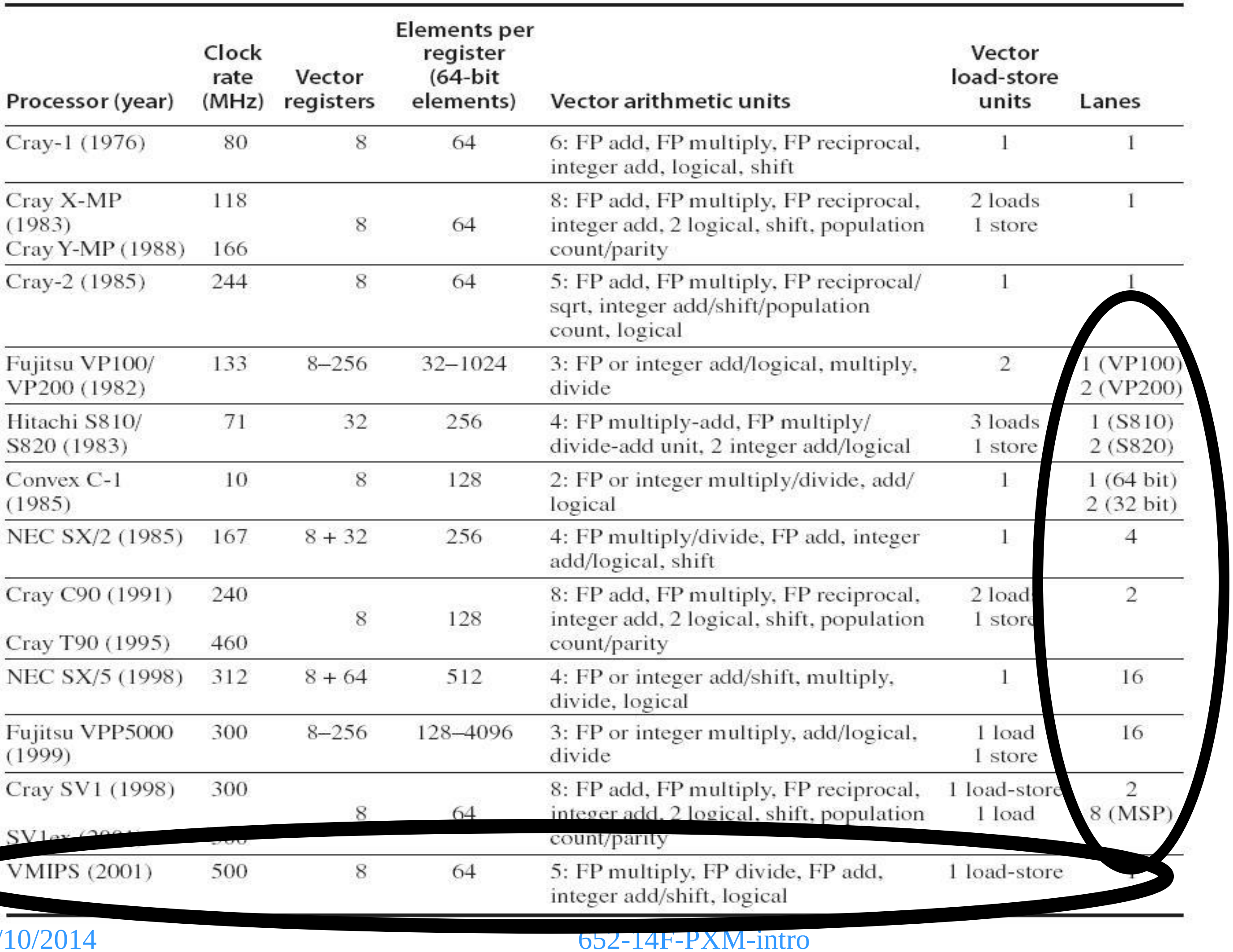
V1 = Compress(V0, VM, Z)

	V0	V1
0	0	
1	-1	
2	0	
3	5	
4	-15	
5	0	
6	0	
7	24	
8	-7	
9	13	
10	0	
11	-17	
12	0	
13	0	



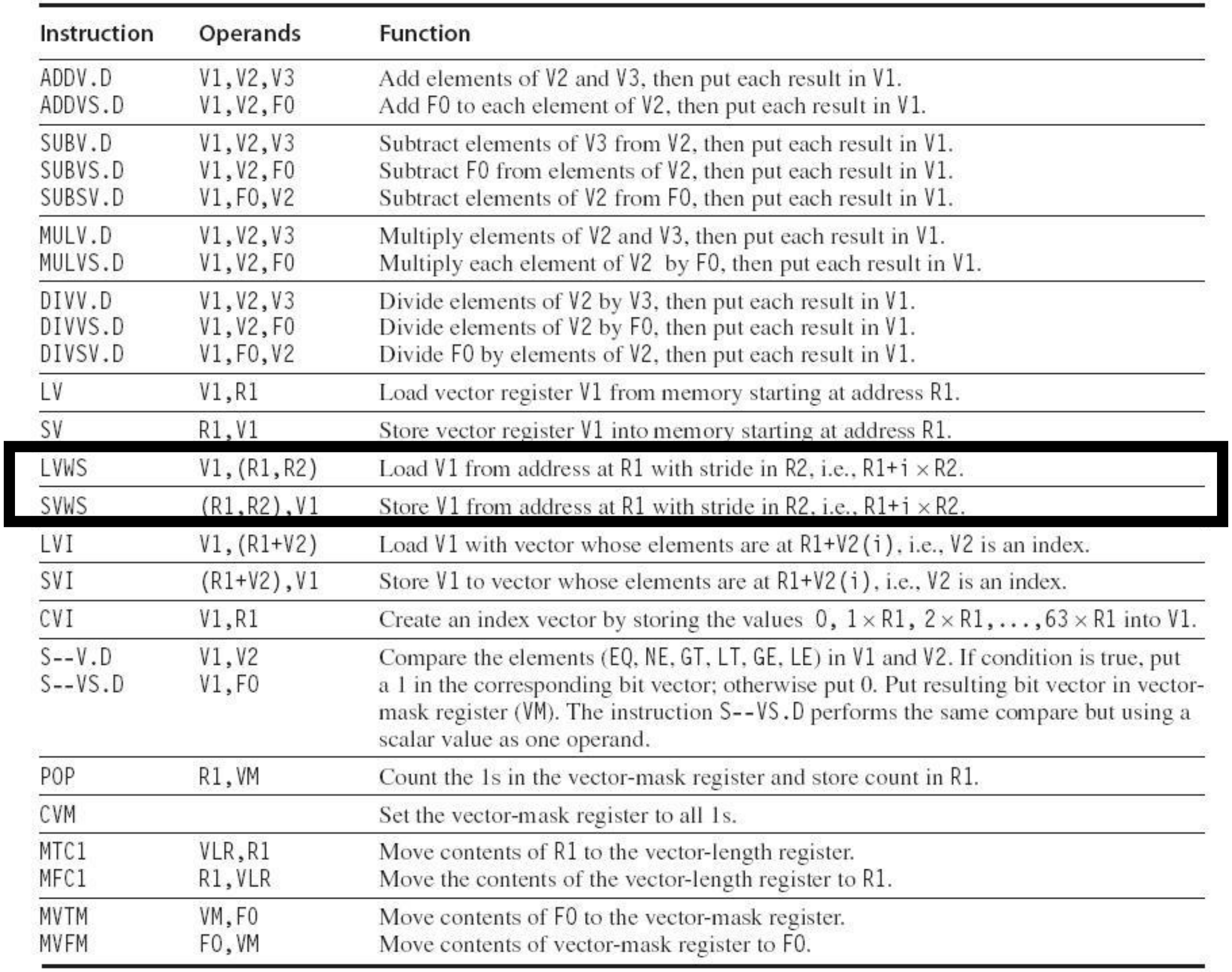
	V0	V1
0	0	01
1	-1	03
2	0	04
3	5	07
4	-15	08
5	0	09
6	0	11
7	24	
8	-7	
9	13	
	0	
	-17	
	0	
	0	

Characteristics of Several Vector Architectures



Processor (year)	Clock rate (MHz)	Vector registers	Elements per register (64-bit elements)	Vector arithmetic units	Vector load-store units	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

The VMIPS Vector Instructions



Instruction	Operands	Function
ADDV.D ADDVS.D	V1,V2,V3 V1,V2,F0	Add elements of V2 and V3, then put each result in V1. Add F0 to each element of V2, then put each result in V1.
SUBV.D SUBVS.D SUBSV.D	V1,V2,V3 V1,V2,F0 V1,F0,V2	Subtract elements of V3 from V2, then put each result in V1. Subtract F0 from elements of V2, then put each result in V1. Subtract elements of V2 from F0, then put each result in V1.
MULV.D MULVS.D	V1,V2,V3 V1,V2,F0	Multiply elements of V2 and V3, then put each result in V1. Multiply each element of V2 by F0, then put each result in V1.
DIVV.D DIVVS.D DIVSV.D	V1,V2,V3 V1,V2,F0 V1,F0,V2	Divide elements of V2 by V3, then put each result in V1. Divide elements of V2 by F0, then put each result in V1. 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.

A MIPS ISA extended to support Vector Instructions. The same as DLXV

Multiple Lanes

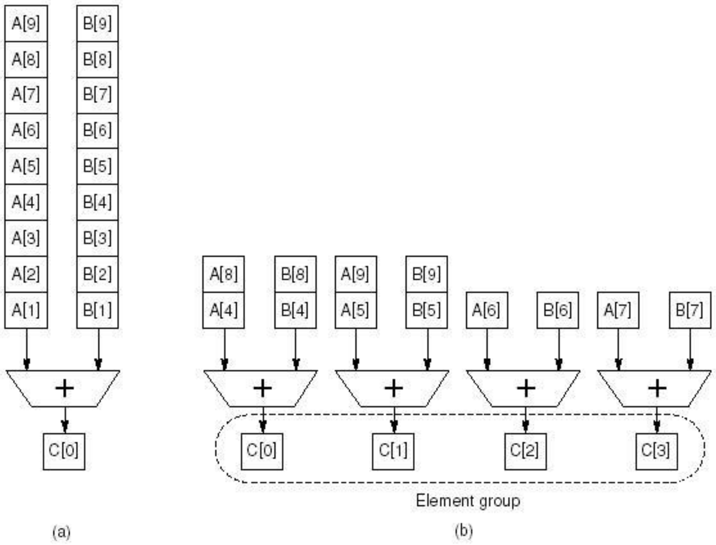

A[9] A[8] A[7] A[6] A[5] A[4] A[3] A[2] A[1]
B[9] B[8] B[7] B[6] B[5] B[4] B[3] B[2] B[1]
+
C[0]
(a)
A[8] A[4] B[8] B[4] A[9] A[5] B[9] B[5] A[6] B[6] A[7] B[7]
+ + + +
C[0] C[1] C[2] C[3]
Element group
(b)

Figure G.11 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B. The machine shown in (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. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*. (Reproduced with permission from Asanovic [1998].)

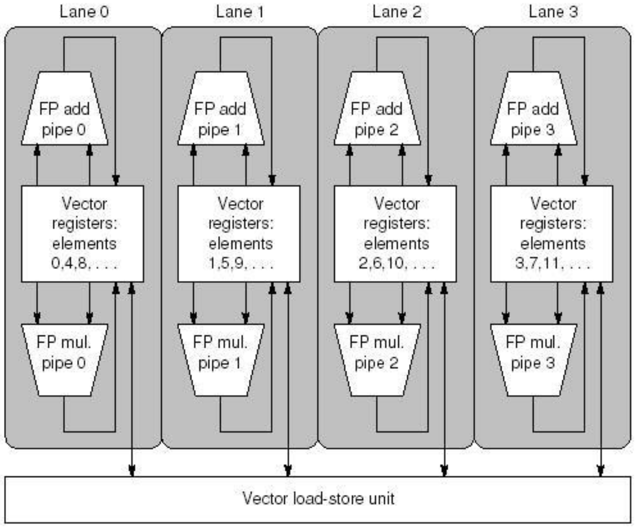
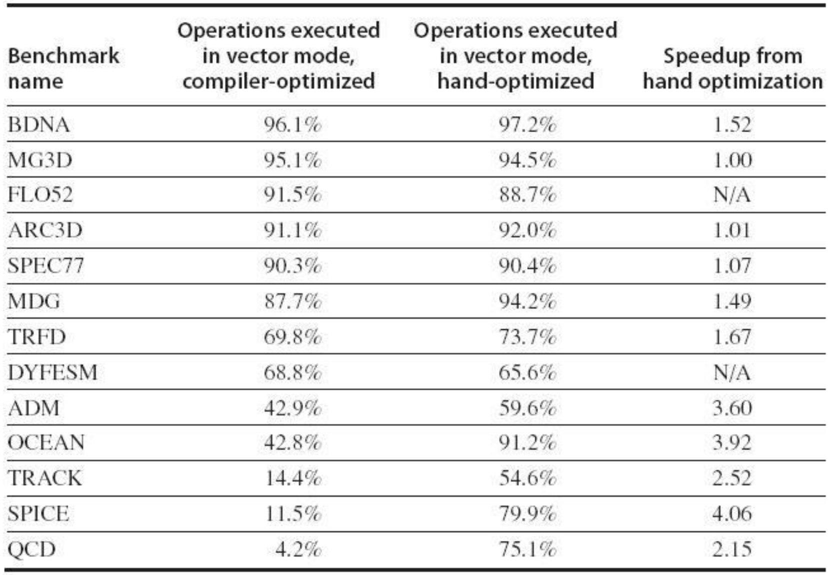

Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0,4,8, . . .
Vector registers: elements 1,5,9, . . .
Vector registers: elements 2,6,10, . . .
Vector registers: elements 3,7,11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

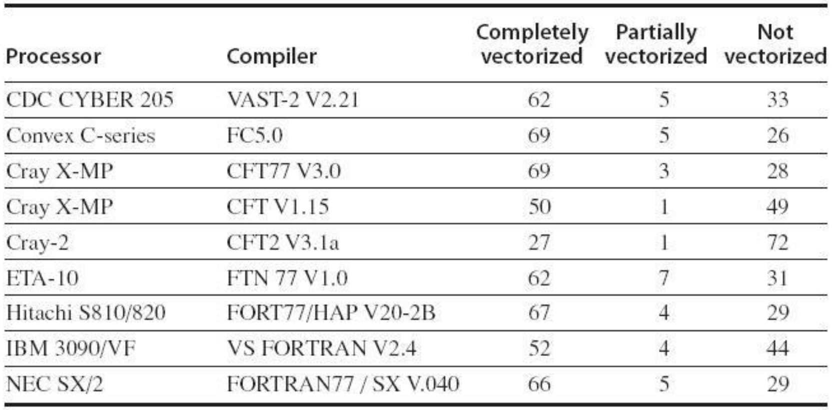
Figure G.12 Structure of a vector unit containing four lanes. The vector-register storage is divided across the lanes, with each lane holding every fourth element of each vector register. There are three vector functional units shown, an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, that act in concert to complete a single vector instruction. Note how each section of the vector-register file only needs to provide enough ports for pipelines local to its lane; this dramatically reduces the cost of providing multiple ports to the vector registers. The path to provide the scalar operand for vector-scalar instructions is not shown in this figure, but the scalar value must be broadcast to all lanes.

Vectorizing Compilers



Benchmark name	Operations executed in vector mode, compiler-optimized	Operations executed in vector mode, hand-optimized	Speedup from hand optimization
BDNA	96.1%	97.2%	1.52
MG3D	95.1%	94.5%	1.00
FLO52	91.5%	88.7%	N/A
ARC3D	91.1%	92.0%	1.01
SPEC77	90.3%	90.4%	1.07
MDG	87.7%	94.2%	1.49
TRFD	69.8%	73.7%	1.67
DYFESM	68.8%	65.6%	N/A
ADM	42.9%	59.6%	3.60
OCEAN	42.8%	91.2%	3.92
TRACK	14.4%	54.6%	2.52
SPICE	11.5%	79.9%	4.06
QCD	4.2%	75.1%	2.15

Figure G.14 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP [Vajapeyam 1991]. The first column shows the vectorization level obtained with the compiler, while the second column shows the results after the codes have been hand-optimized by a team of Cray Research programmers. Speedup numbers are not available for FLO52 and DYFESM as the hand-optimized runs used larger data sets than the compiler-optimized runs.

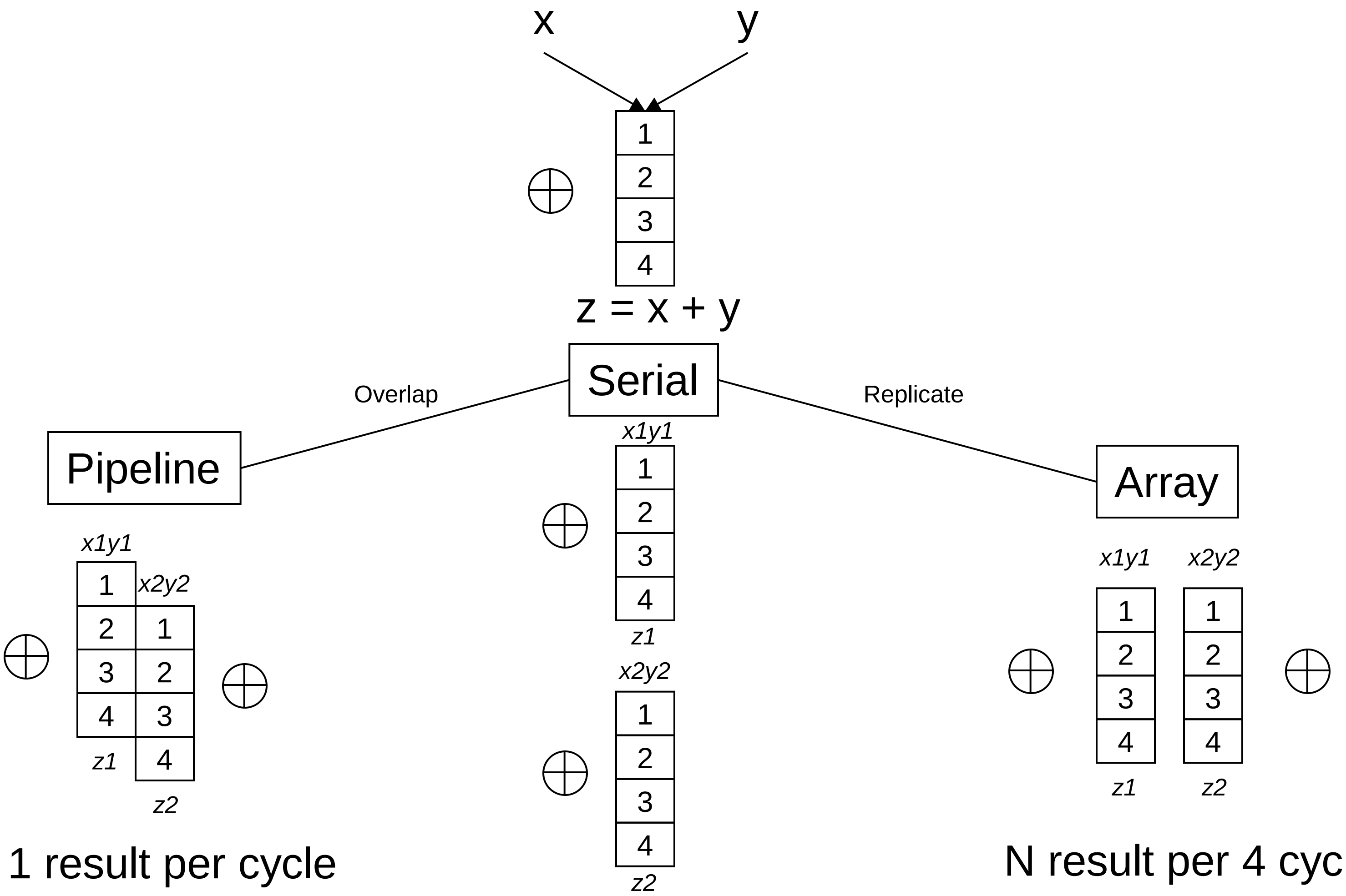


Processor	Compiler	Completely vectorized	Partially vectorized	Not vectorized
CDC CYBER 205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
Cray X-MP	CFT77 V3.0	69	3	28
Cray X-MP	CFT V1.15	50	1	49
Cray-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Figure G.15 Result of applying vectorizing compilers to the 100 FORTRAN test kernels. For each processor we indicate how many loops were completely vectorized, partially vectorized, and unvectorized. These loops were collected by Callahan, Dongarra, and Levine [1988]. Two different compilers for the Cray X-MP show the large dependence on compiler technology.

PERFORMANCE ANALYSIS OF VECTOR ARCHITECTURES

Serial, Parallel and Pipelines



1 result per cycle

1 result per 4 cycles

N result per 4 cycles

Generic Performance Formula

(R. Hockney & C. Jesshope 81)

$$t = r_{\infty}^{-1}(n + n_{1/2})$$

r_{∞}^{-1} **Asymptotic Performance:** Maximum rate of computation in floating point operations per second. Performance of the architecture with an infinite length vector

$n_{1/2}$ **Half Performance Length:** The vector length needed to achieve half of the peak performance

n **Vector length**

Serial Architecture

Generic Formula:

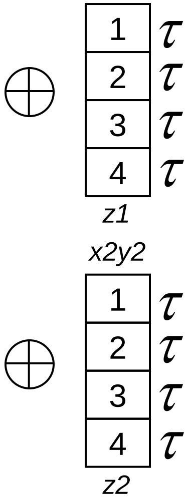
$$t = r_{\infty}^{-1}(n + n_{1/2})$$

$$t_{serial} = l * \tau * n$$

Parameters:

$$r_{\infty}^{-1} = l * \tau$$

$$n_{1/2} = 0$$



l	Number of stages
τ	Time per stage
s	Start up time

Pipeline Architecture

The number of elements that will come out of the pipeline after the initial penalty has been paid

Generic Formula: $t = r_\infty^{-1}(n + n_{1/2})$

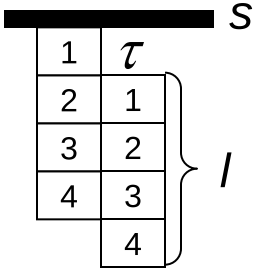
Initial Penalty

$$t_{pipeline} = \tau(\boxed{s+l} + \boxed{(n-1)})$$

$$t_{pipeline} = \tau(n+s+l-1)$$

Parameters: $r_\infty^{-1} = \tau$

$$n_{1/2} = s+l-1$$

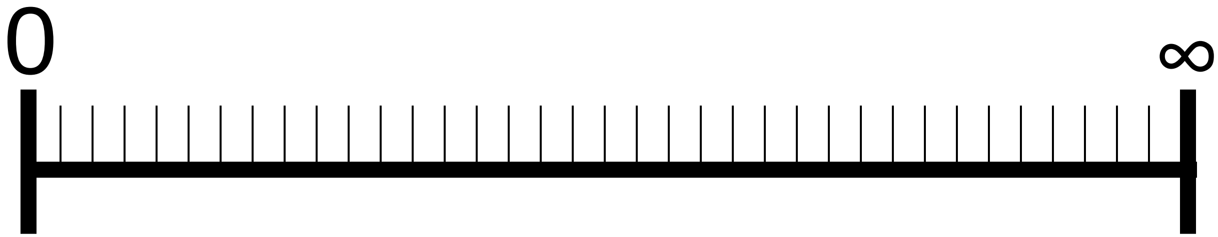


l	Number of stages
τ	Time per stage
s	Start up time

Thus …

- The Asymptotic Performance Parameter
  - It is primarily a characteristic of the computer technology used.
  - It is a scale factor applied to the performance of a particular computer architecture reflecting the technology in which particular implementation of that architecture is built.
- The N half Parameter
  - The amount of parallelism that is presented in a given architecture.
  - Determined by a combination of vector unit startup and vector unit latency

The N Half Range



0

∞

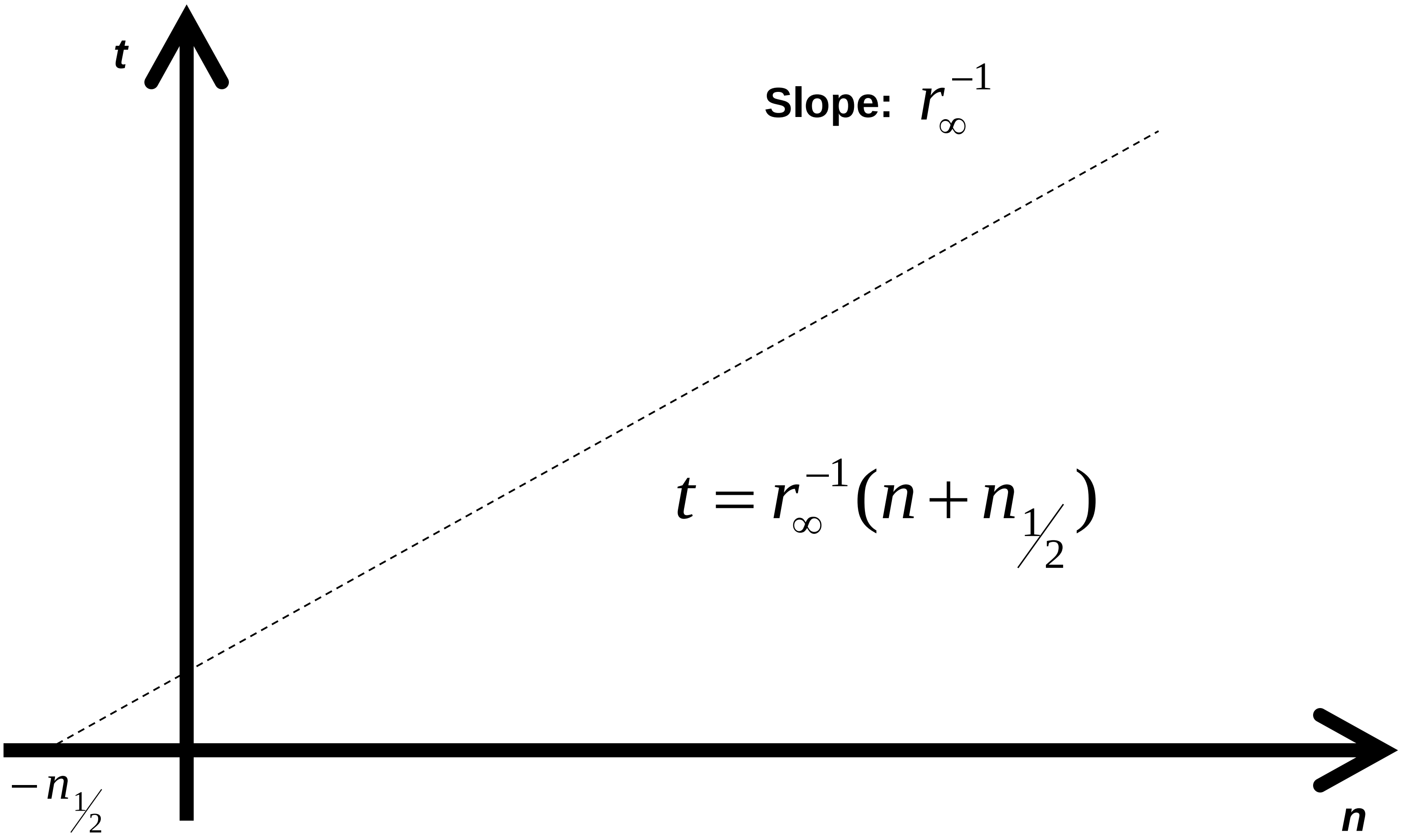
Serial Machine

$n_{1/2}$

Infinite array of processor

The relative performance of different algorithms on a computer is determined by the value of *N half* (matching problem parallelism with architecture parallelism)

Vector Length v.s. Vector Performance



Calculation of r∞

$$t = r_{\infty}^{-1}(n + n_{1/2})$$

Measure t *for two or more* n *values*

n	t
n0	t0
n1	t1
…	…

Then, determine the slope of the line in the form of:

$$r_{\infty}^{-1} = \frac{t1 - t0}{n1 - n0}$$

Parameters of Several Parallel Architectures

Computer	***N half***	***R infinity***
CRAY-1	10-20	80
BSP	25-150	50
2-pipe CDC CYBER 205	100	100
1-pipe TIASC	30	12
CDC STAR 100	150	25
(64 x 64) ICL DAP	2048	16

Another Example
The Chaining Effect

Assume m vector operations unchained

$$t_m = \sum_{i=1}^{m} [s_i + l_i + (n-1)]\tau$$

$$t_m = \sum_{i=1}^{m} [(s_i + l_i - 1) + n]\tau$$

Thus $$t = {}^{1}\!/_{m}\, t_m$$

$$t = {}^{1}\!/_{m} * \sum_{i=1}^{m} [(s_i + l_i - 1) + n]\tau$$

So $$n_{1/2} = s + l - 1$$

$$r_\infty^{-1} = \tau$$

Assume that all **s**'s are the same. The same goes for the **l**'s

Another Example
The Chaining Effect

Assume m vector operations chained

$$t_m = [\sum_{i=1}^{m}(s_i + l_i) + (n-1)]\tau$$

$$t_m = [m*(s+l) - 1 + n]\tau$$

Thus
$$t = {}^{1}\!/_{m}\, t_m$$

$$t = {}^{1}\!/_{m} *[m*(s+l) - 1 + n]\tau$$

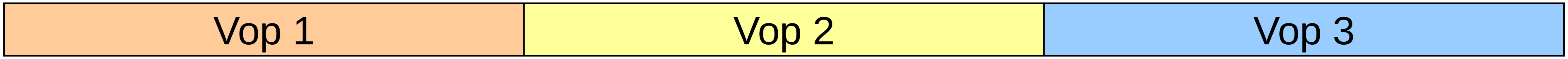
So
$$n_{1/2} = m(s+l) - 1$$

$$r_{\infty}^{-1} = {}^{\tau}\!/_{m}$$

Summary

Explanation

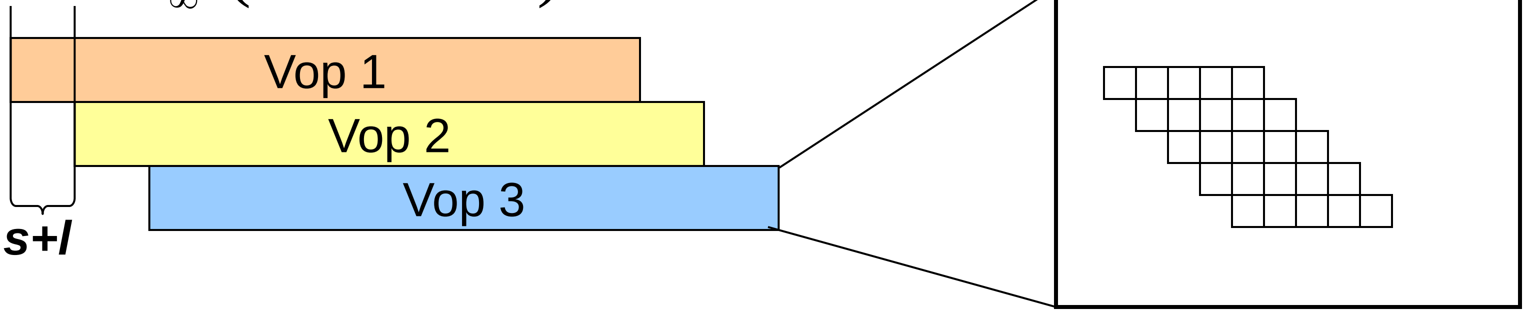
Unchained



$r_{\infty}^{-1}(n+s+l-1)$ $r_{\infty}^{-1}(n+s+l-1)$ $r_{\infty}^{-1}(n+s+l-1)$

Chained

$r_{\infty}^{-1}(n+s+l-1)$


Vop 1
Vop 2
Vop 3
s+l