

# Computer Architecture

ELE 475 / COS 475

## Slide Deck 11: Vector, SIMD, and GPUs

David Wentzlaff

Department of Electrical Engineering

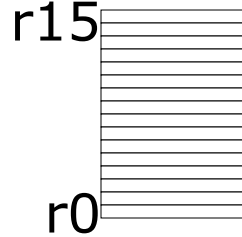
Princeton University

# Agenda

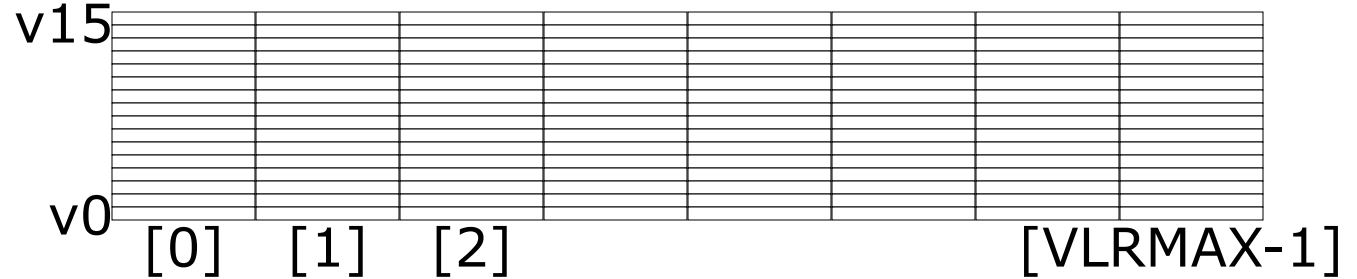
- Vector Processors
- Single Instruction Multiple Data (SIMD)  
Instruction Set Extensions
- Graphics Processing Units (GPU)

# Vector Programming Model

*Scalar Registers*



*Vector Registers*

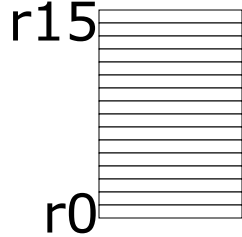


*Vector Length Register*

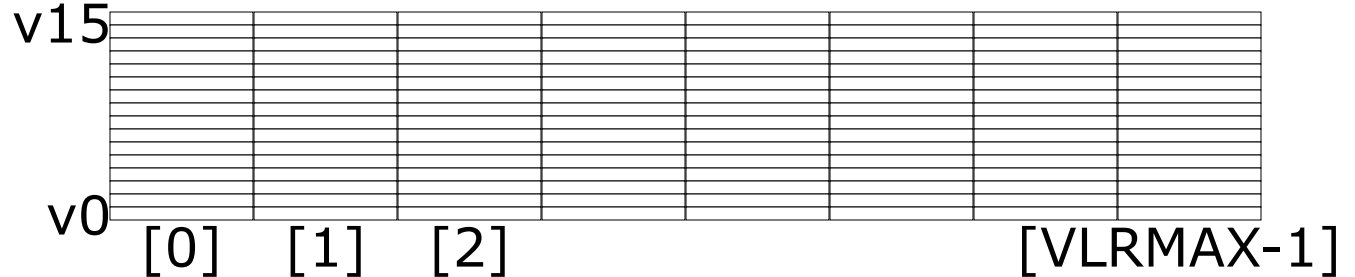
VLR

# Vector Programming Model

*Scalar Registers*

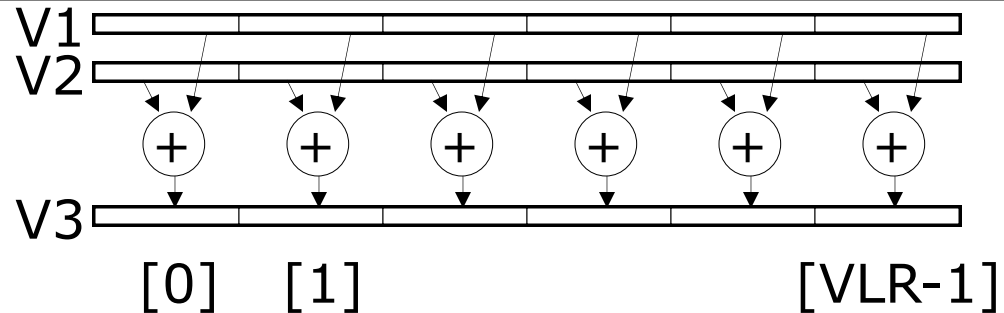


*Vector Registers*



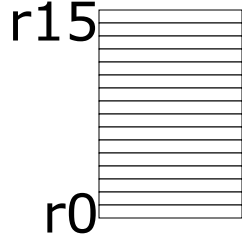
*Vector Length Register* VLR

Vector Arithmetic  
Instructions  
ADDVV V3, V1, V2

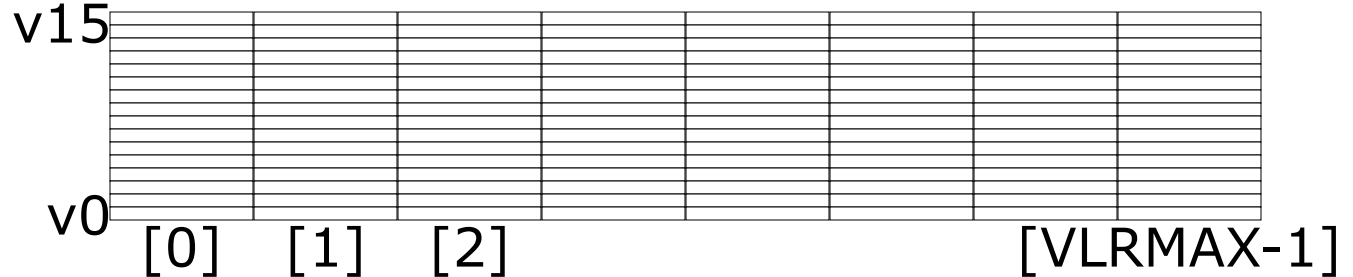


# Vector Programming Model

*Scalar Registers*

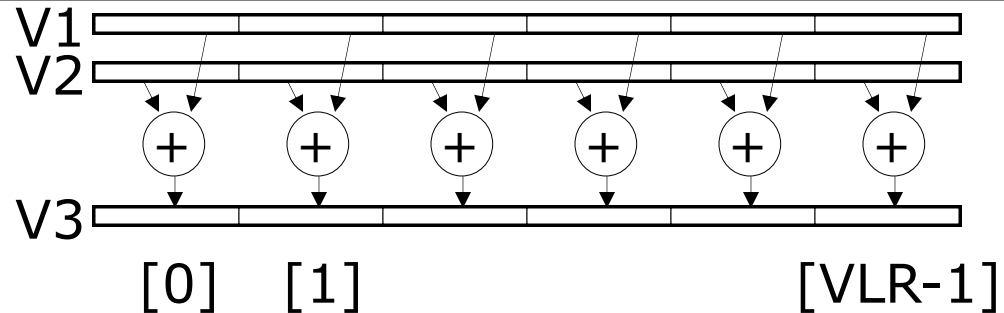


*Vector Registers*

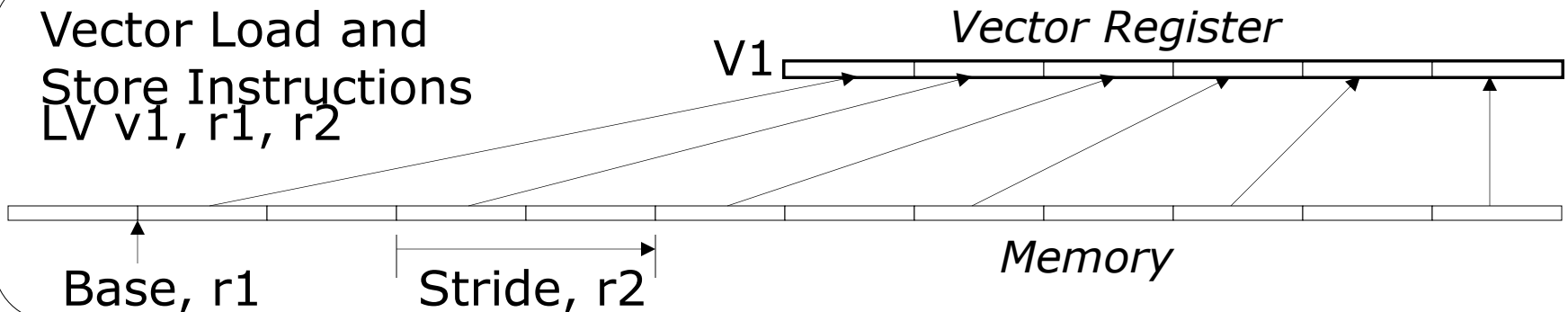


*Vector Length Register* VLR

Vector Arithmetic  
Instructions  
ADDVV V3, V1, V2



Vector Load and  
Store Instructions  
LV v1, r1, r2



# Vector Code Element-by-Element Multiplication

## # C code

```
for (i=0; i<64; i++)  
    C[i] = A[i] * B[i];
```

## # Scalar Assembly Code

```
    LI R4, 64  
loop:  
    L.D F0, 0(R1)  
    L.D F2, 0(R2)  
    MUL.D F4, F2, F0  
    S.D F4, 0(R3)  
    DADDIU R1, 8  
    DADDIU R2, 8  
    DADDIU R3, 8  
    DSUBIU R4, 1  
    BNEZ R4, loop
```

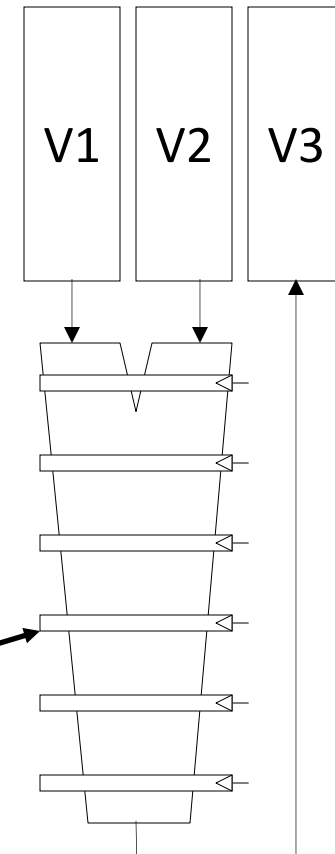
## # Vector Assembly Code

```
    LI VLR, 64  
    LV V1, R1  
    LV V2, R2  
    MULVV.D V3, V1, V2  
    SV V3, R3
```

# Vector Arithmetic Execution

- Use deep pipeline ( $\Rightarrow$  fast clock) to execute element operations
- Simplifies control of deep pipeline because elements in vector are independent
  - no data hazards!
  - no bypassing needed

Six stage multiply pipeline

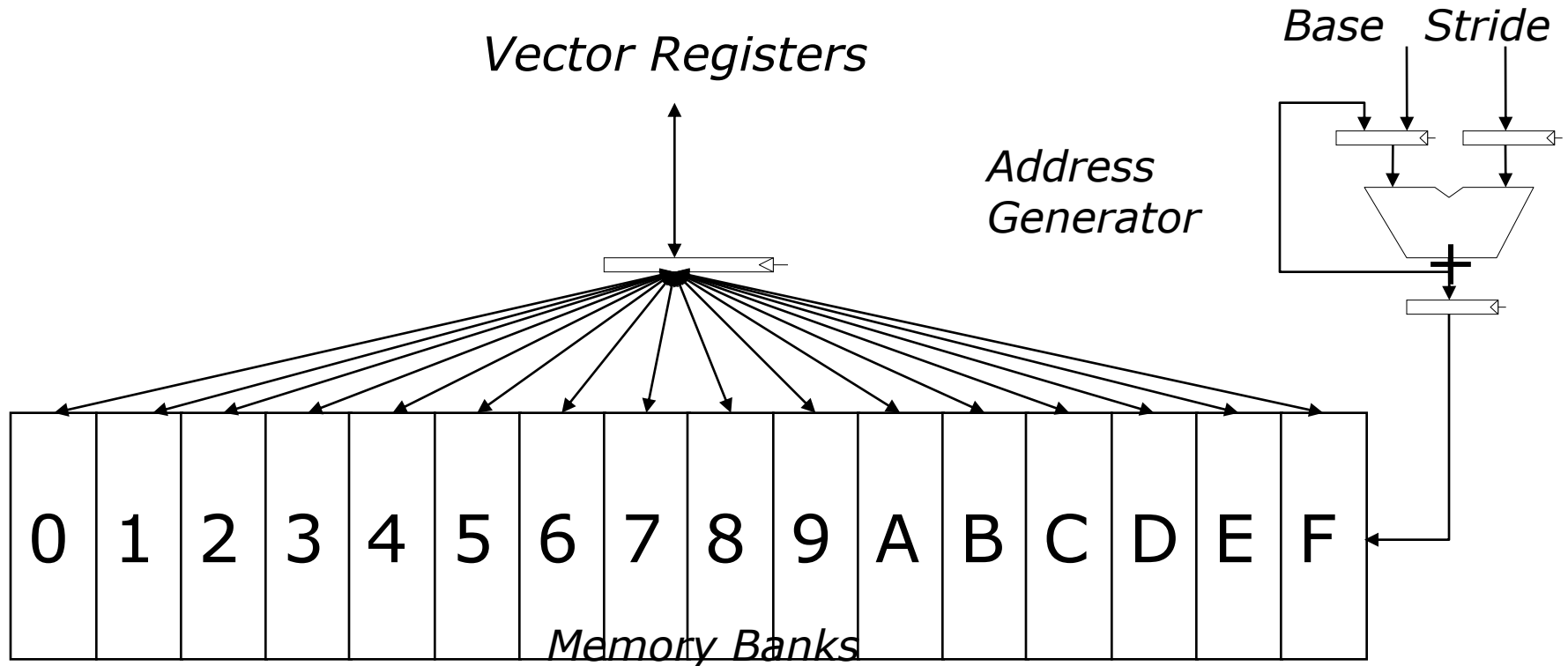


$$V3 \leftarrow V1 * V2$$

# Interleaved Vector Memory System

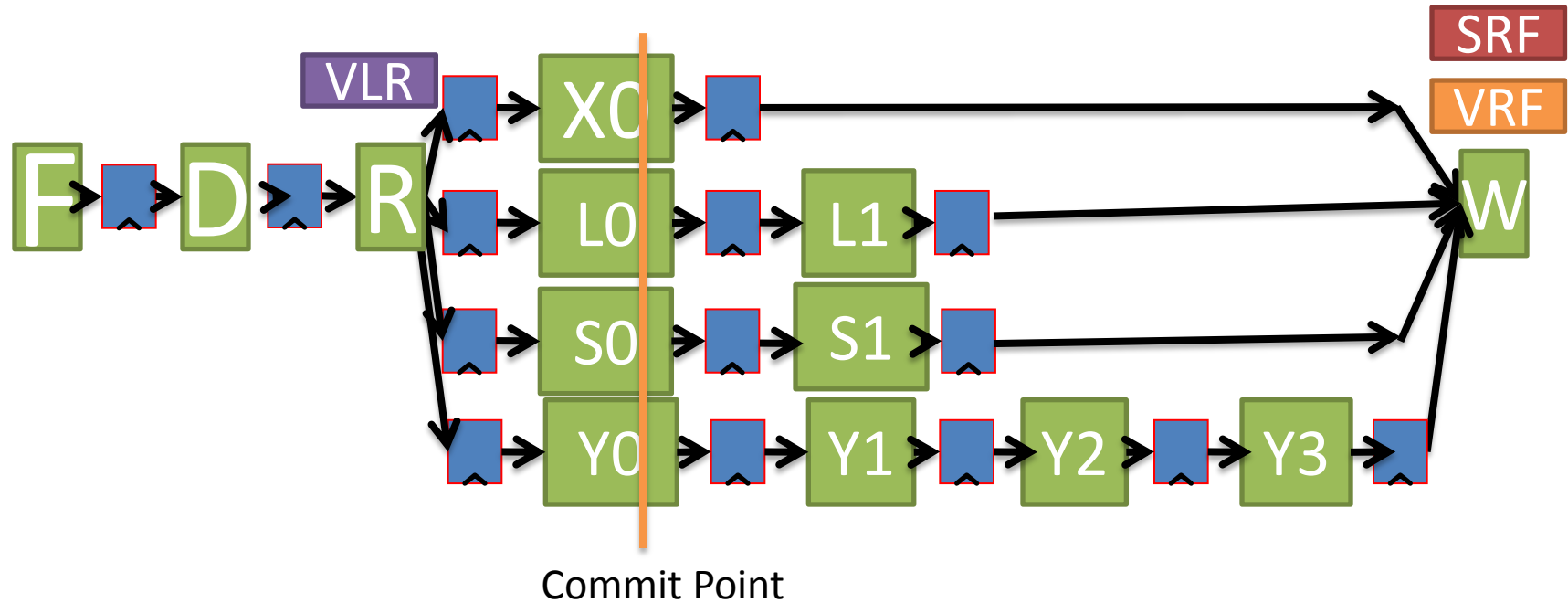
Cray-1, 16 banks, 4 cycle bank busy time, 12 cycle latency

- *Bank busy time*: Time before bank ready to accept next request





# Example Vector Microarchitecture



# Basic Vector Execution

## # C code

```
for (i=0; i<4; i++)
    C[i] = A[i] * B[i];
```

VLR = 4

LV        V2, R2    F   D   R   L0 L1 W

          R   L0 L1 W

          R   L0 L1 W

          R   L0 L1 W

MULVV.D V3, V1, V2 F   D   D   D   D   D   D   D   R   Y0 Y1 Y2 Y3 W

          R   Y0 Y1 Y2 Y3 W

          R   Y0 Y1 Y2 Y3 W

          R   Y0 Y1 Y2 Y3 W

SV        V3, R3            F   F   F   F   F   F   F   D   D   D   D   D   D   D   D   R   S0 S1 W

          R   S0 S1 W

          R   S0 S1 W

          R   S0 S1 W

## # Vector Assembly Code

LI VLR, 4

LV V1, R1

LV V2, R2

MULVV.D V3, V1, V2

SV V3, R3

# Vector Instruction Parallelism


- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes

Load Unit

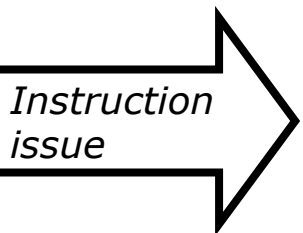
Multiply Unit

Add Unit

*time*

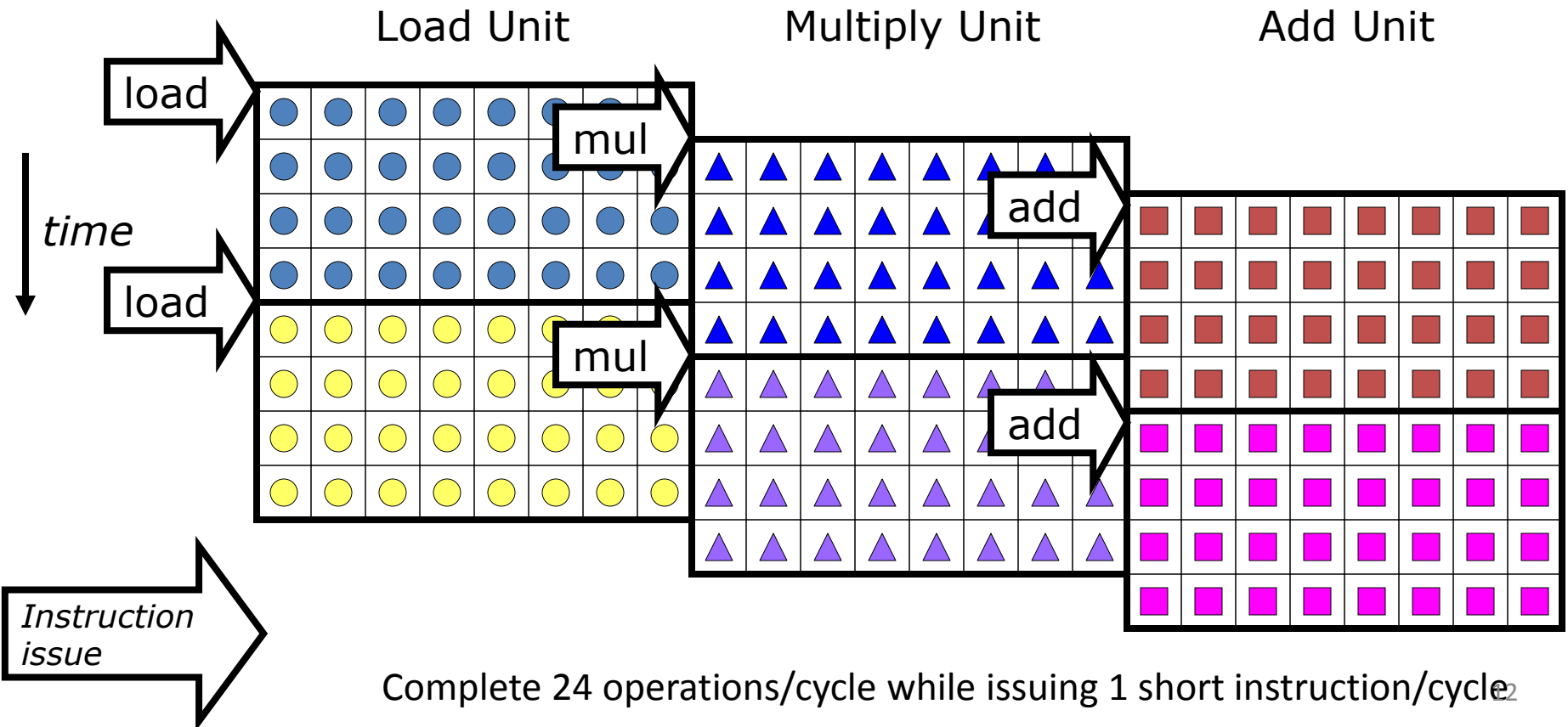


*Instruction  
issue*



# Vector Instruction Parallelism


- Can overlap execution of multiple vector instructions
  - example machine has 32 elements per vector register and 8 lanes



# Vector Chaining

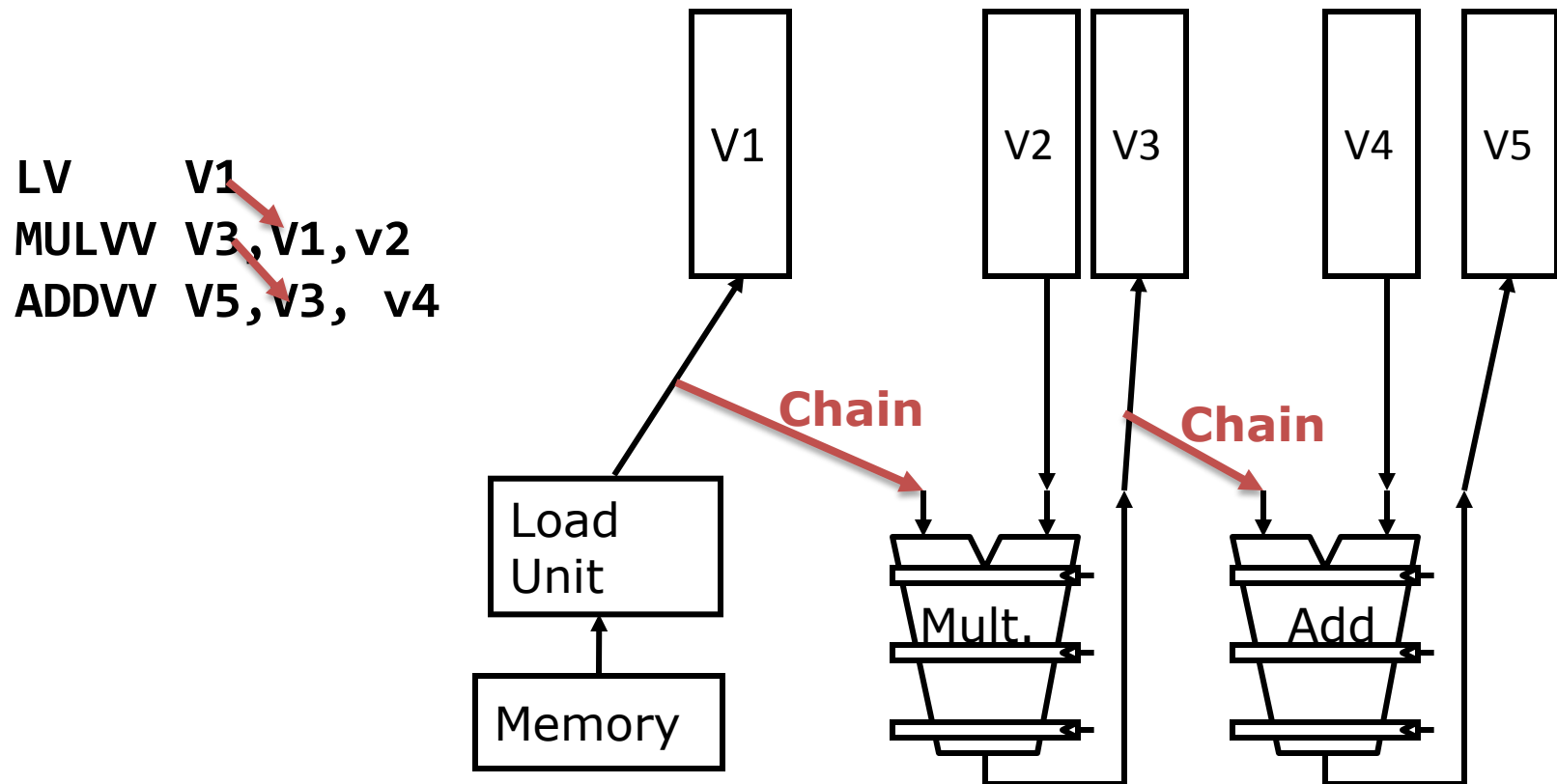
- Vector version of register bypassing
  - introduced with Cray-1

```
LV      V1
MULVV   V3, V1, v2
ADDVV   V5, V3, v4
```



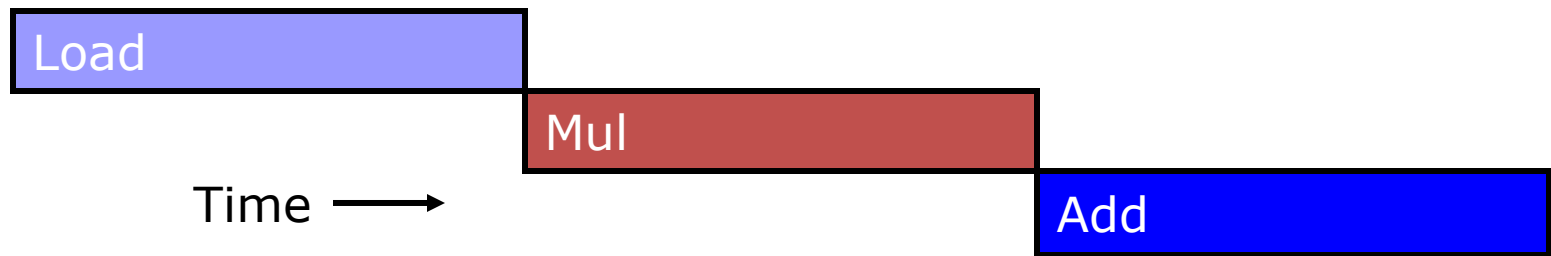
# Vector Chaining

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

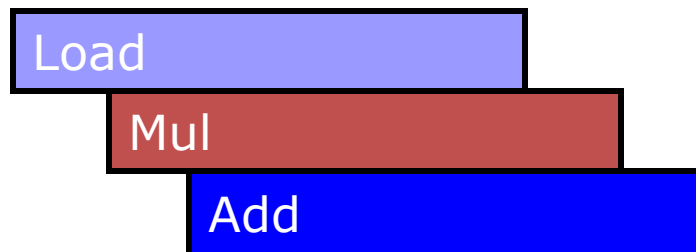


# Vector Chaining Advantage

- 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



# Chaining (Register File) Vector Execution

# C code	# Vector Assembly Code
for (i=0; i<4; i++)	LI VLR, 4
C[i] = A[i] * B[i];	LV V1, R1
	LV V2, R2
VLR = 4	MULVV.D V3, V1, V2
	SV V3, R3
LV           V2, R2   F   D   R   L0 L1 W	
	R   L0 L1 W
	R   L0 L1 W
	R   L0 L1 W
MULVV.D V3, V1, V2 F   D   D   D   D   R   Y0 Y1 Y2 Y3 W	
	R   Y0 Y1 Y2 Y3 W
	R   Y0 Y1 Y2 Y3 W
	R   Y0 Y1 Y2 Y3 W
SV           V3, R3           F   F   F   F   D   D   D   D   D   D   R   S0 S1 W	
	R   S0 S1 W
	R   S0 S1 W
	R   S0 S1 W



# Chaining (Bypass Network) Vector Execution

# C code	# Vector Assembly Code
for (i=0; i<4; i++)	LI VLR, 4
C[i] = A[i] * B[i];	LV V1, R1
	LV V2, R2
VLR = 4	MULVV.D V3, V1, V2
	SV V3, R3
LV           V2, R2   F   D   R   L0 L1 W	
	R   L0 L1 W
	R   L0 L1 W
	R   L0 L1 W
MULVV.D V3, V1, V2 F   D   D   D   D   R   Y0 Y1 Y2 Y3 W	
	R   Y0 Y1 Y2 Y3 W
	R   Y0 Y1 Y2 Y3 W
	R   Y0 Y1 Y2 Y3 W
SV           V3, R3           F   F   F   F   D   D   D   D   D   D   R   S0 S1 W	
	R   S0 S1 W
	R   S0 S1 W
	R   S0 S1 W

# Chaining (Bypass Network) Vector Execution and More RF Ports

# C code	# Vector Assembly Code
for (i=0; i<4; i++)	LI VLR, 4
C[i] = A[i] * B[i];	LV V1, R1
	LV V2, R2
VLR = 4	MULVV.D V3, V1, V2
	SV V3, R3
LV           V2, R2   F   D   R   L0 L1 W	
	R   L0 L1 W
	R   L0 L1 W
	R   L0 L1 W
MULVV.D V3, V1, V2 F   D   D   R   Y0 Y1 Y2 Y3 W	
	R   Y0 Y1 Y2 Y3 W
	R   Y0 Y1 Y2 Y3 W
	R   Y0 Y1 Y2 Y3 W
SV           V3, R3           F   F   D   D   D   D   R   S0 S1 W	
	R   S0 S1 W
	R   S0 S1 W
	R   S0 S1 W

# Chaining (Bypass Network) Vector Execution and More RF Ports

VLR = 8

```

LV      V2, R2  F  D  R  L0 L1 W
        R  L0 L1 W
        R  L0 L1 W
        R  L0 L1 W
        R  L0 L1 W
        R  L0 L1 W
        R  L0 L1 W
        R  L0 L1 W
MULVV.D V3, V1, V2 F  D  D  R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
        R  Y0 Y1 Y2 Y3 W
SV      V3, R3          F  F  D  D  D  D  R  S0 S1 W
        R  S0 S1 W
        R  S0 S1 W
        R  S0 S1 W
        R  S0 S1 W
        R  S0 S1 W
        R  S0 S1 W
    
```

# Vector Stripmining

Problem: Vector registers have finite length

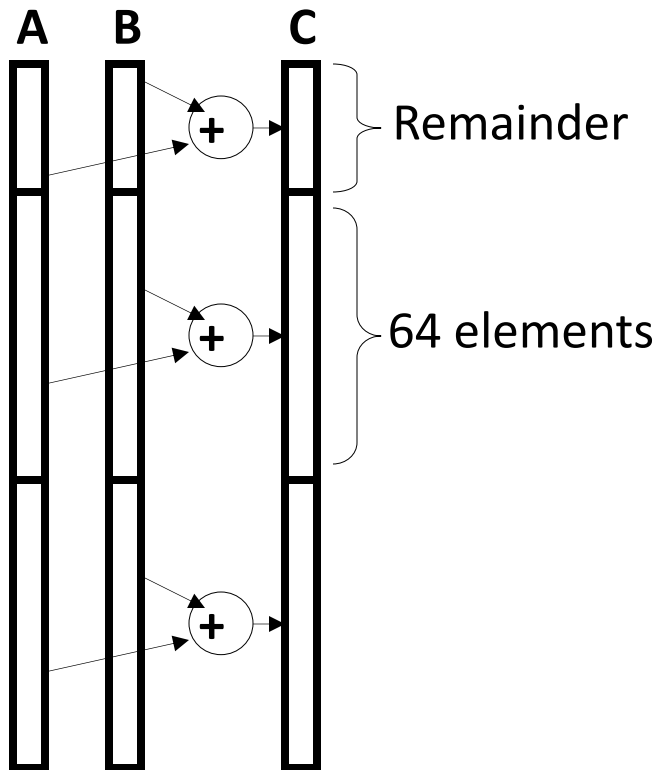
Solution: Break loops into pieces that fit in registers, *“Stripmining”*

# 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  
    LV V2, RB  
    MULVV.D V3, V1, V2  
    SV V3, RC  
    DSLL R2, R1, 3    # Multiply by 8  
    DADDU RA, RA, R2  # Bump pointer  
    DADDU RB, RB, R2  
    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 Stripmining

```

VLR = 4
LV  F  D  R  L0 L1 W
      R  L0 L1 W
      R  L0 L1 W
      R  L0 L1 W
LV      V2, R2  F  D  R  L0 L1 W
      R  L0 L1 W
      R  L0 L1 W
      R  L0 L1 W
MULVV.D V3, V1, V2 F  D  D  R  Y0 Y1 Y2 Y3 W
      R  Y0 Y1 Y2 Y3 W
      R  Y0 Y1 Y2 Y3 W
      R  Y0 Y1 Y2 Y3 W
SV      V3, R3      F  F  D  D  D  D  R  S0 S1 W
      R  S0 S1 W
      R  S0 S1 W
      R  S0 S1 W
DSLL R2, R1, 3      F  F  F  F  D  R  X  W
DADDU RA, RA, R2      F  D  R  X  W
DADDU RB, RB, R2      F  D  R  X  W
DADDU RC, RC, R2      F  D  R  X  W
DSUBU N, N, R1      F  D  R  X  W
LI R1, 64      F  D  R  X  W
MTC1 VLR, R1      F  D  R  X  W
BGTZ N, loop      F  D  R  X  W
  
```

# Vector Instruction Execution

MULVV C,A,B

# Vector Instruction Execution

MULVV C,A,B

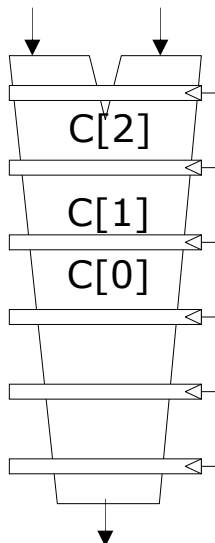
Execution using  
one pipelined  
functional unit

A[6] B[6]

A[5] B[5]

A[4] B[4]

A[3] B[3]



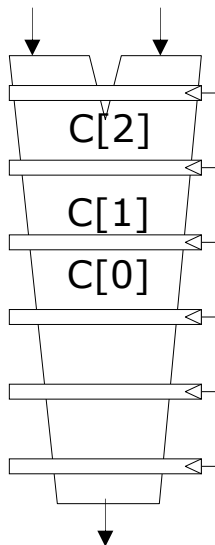


# Vector Instruction Execution

MULVV C,A,B

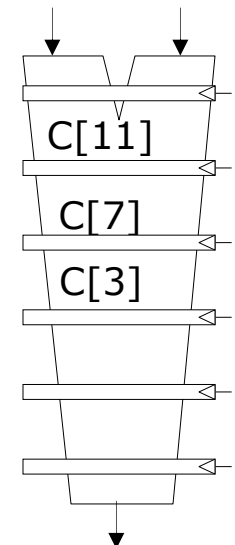
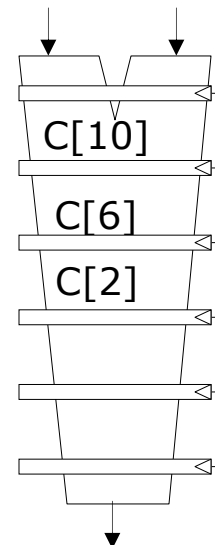
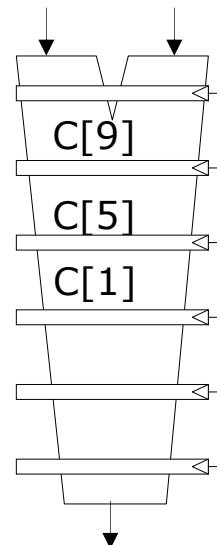
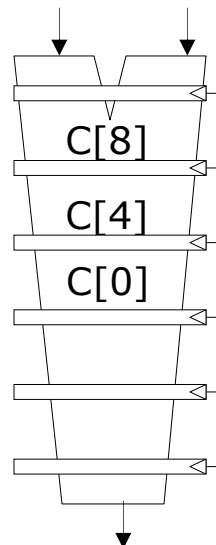
Execution using  
one pipelined  
functional unit

A[6] B[6]  
A[5] B[5]  
A[4] B[4]  
A[3] B[3]

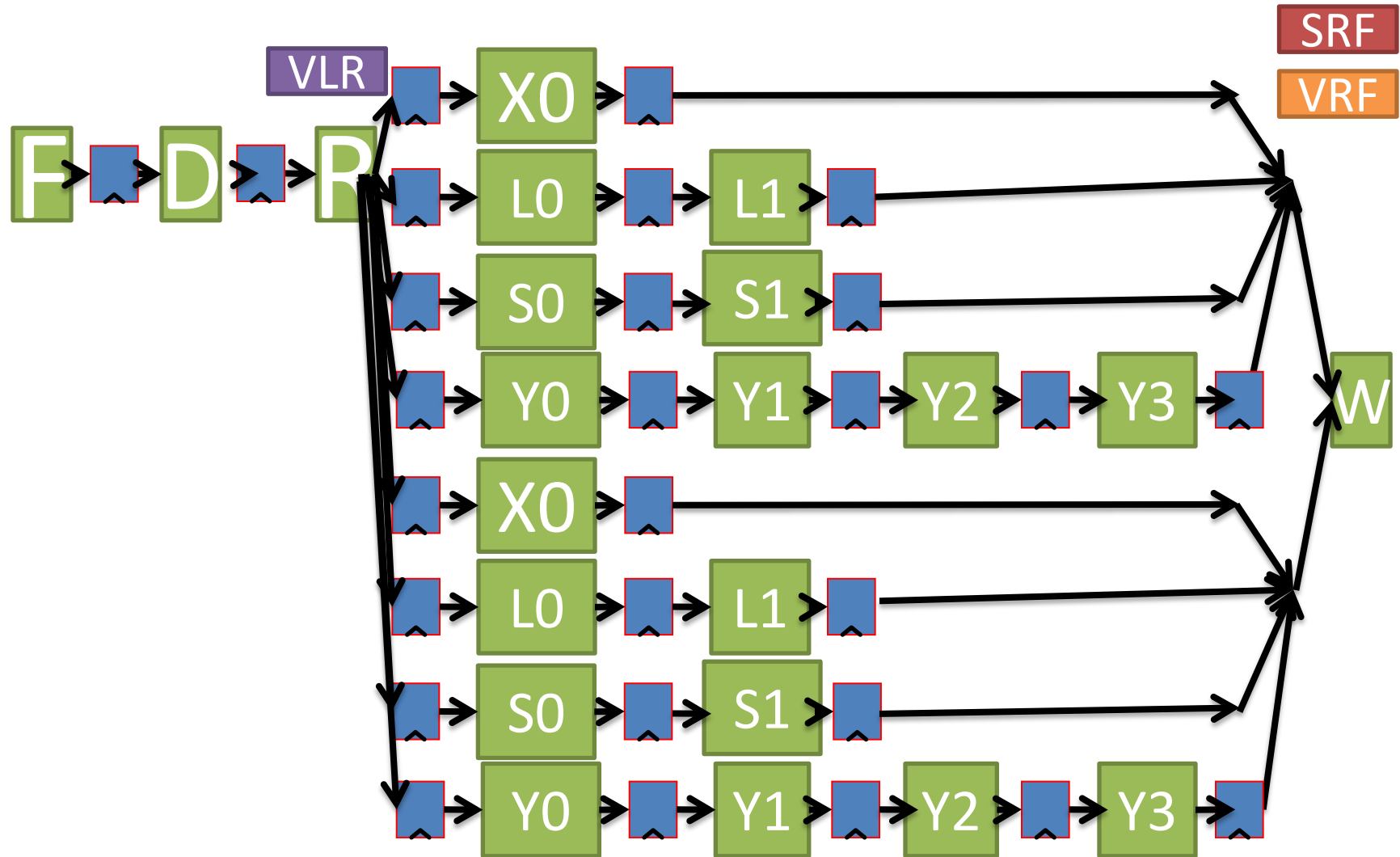


Execution using  
four pipelined  
functional units

A[24] B[24] A[25] B[25] A[26] B[26] A[27] B[27]  
A[20] B[20] A[21] B[21] A[22] B[22] A[23] B[23]  
A[16] B[16] A[17] B[17] A[18] B[18] A[19] B[19]  
A[12] B[12] A[13] B[13] A[14] B[14] A[15] B[15]



# Two Lane Vector Microarchitecture



# Vector Stripmining 2-Lanes

VLR = 4

LV    F    D    R    L0 L1 W

          R    L0 L1 W

              R    L0 L1 W

              R    L0 L1 W

LV            F    D    D    R    L0 L1 W

              R    L0 L1 W

                  R    L0 L1 W

                  R    L0 L1 W

MULVV.D        F    F    D    D    R    Y0 Y1 Y2 Y3 W

                  R    Y0 Y1 Y2 Y3 W

                      R    Y0 Y1 Y2 Y3 W

                      R    Y0 Y1 Y2 Y3 W

SV                    F    F    D    D    D    D    R    S0 S1 W

                          R    S0 S1 W

                              R    S0 S1 W

                              R    S0 S1 W

DSLL R2, R1, 3                    F    F    F    F    D    R    X    W

DADDU RA, RA, R2                    F    D    R    X    W

DADDU RB, RB, R2                    F    D    R    X    W

DADDU RC, RC, R2                    F    D    R    X    W

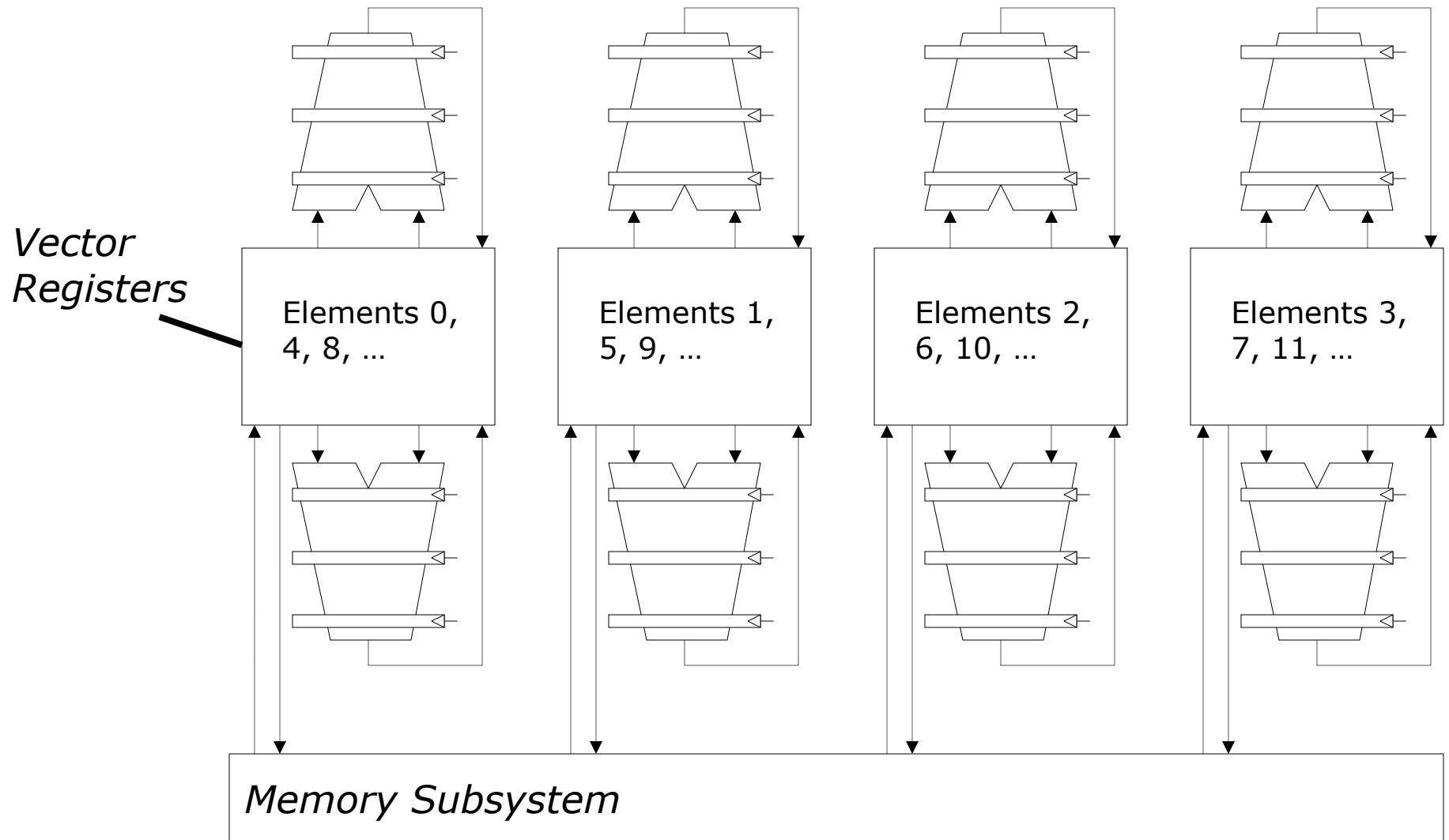
DSUBU N, N, R1                    F    D    R    X    W

LI R1, 64                    F    D    R    X    W

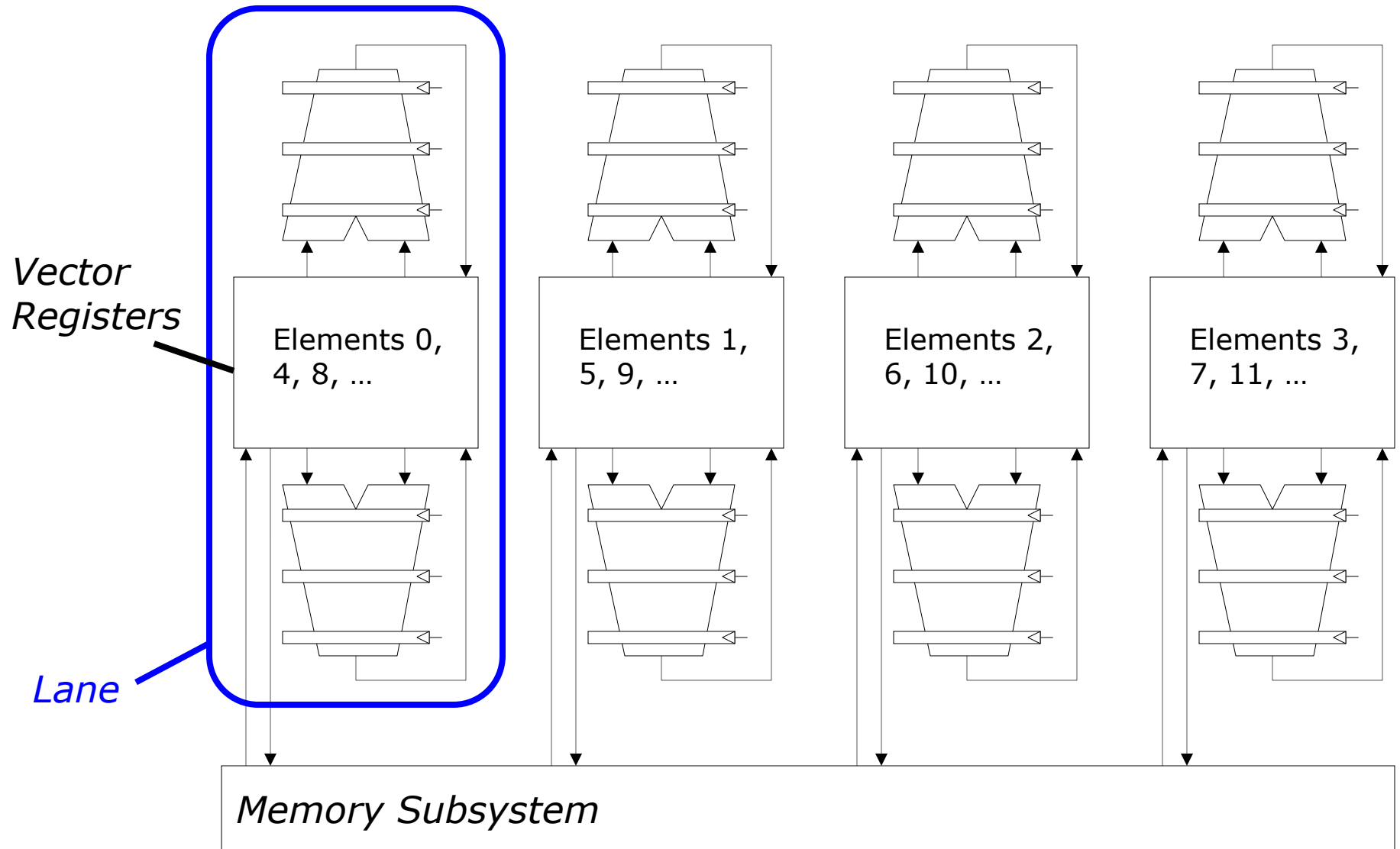
MTC1 VLR, R1                    F    D    R    X    W

BGTZ N, loop                    F    D    R    X    W

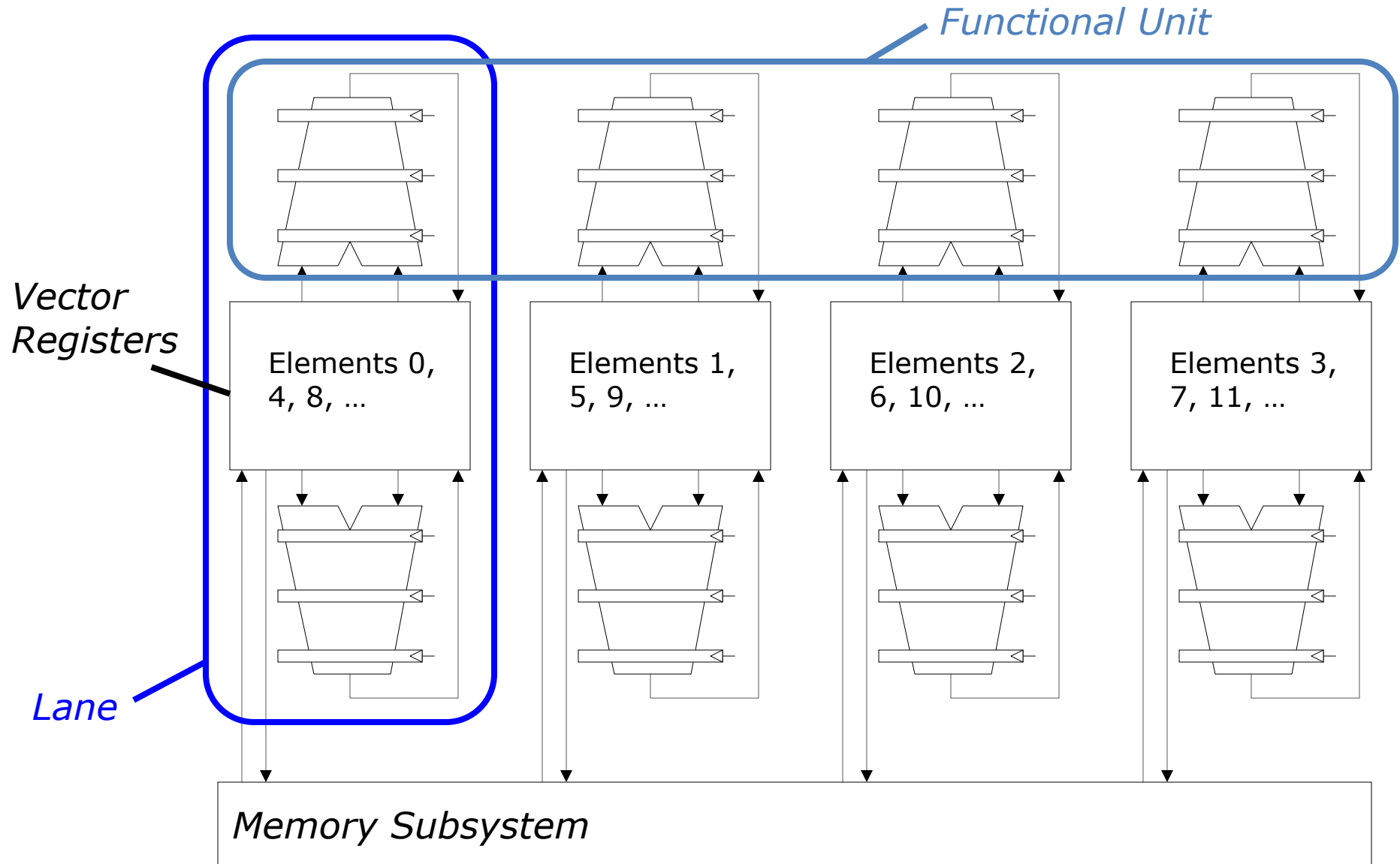
# Vector Unit Structure



# Vector Unit Structure



# Vector Unit Structure



# T0 Vector Microprocessor (UCB/ICSI, 1995)

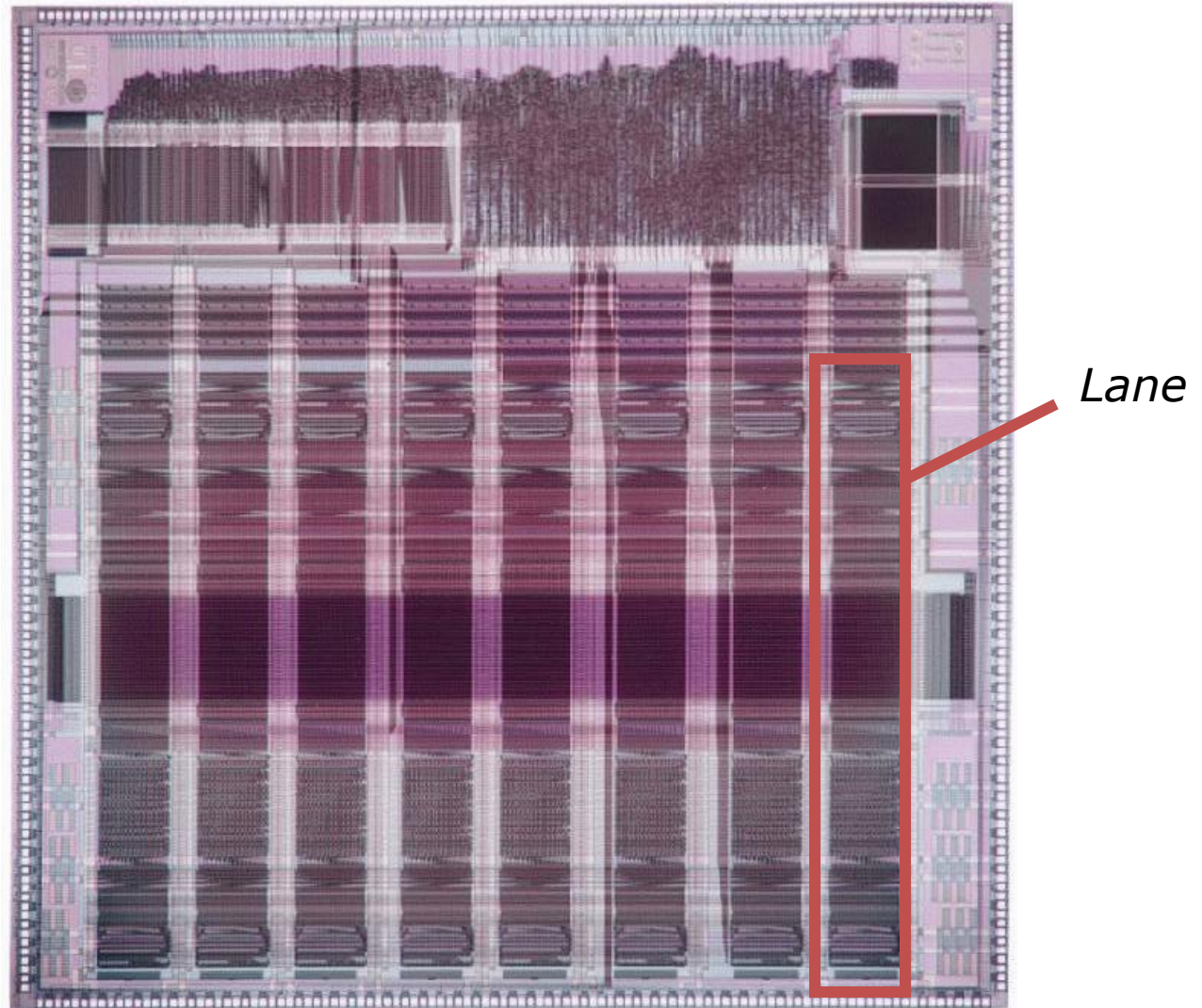


Photo of Berkeley T0, © University of California (Berkeley)  
<http://www1.icsi.berkeley.edu/Speech/spert/t0die.jpg>

# T0 Vector Microprocessor (UCB/ICSI, 1995)

*Vector register  
elements striped  
over lanes*

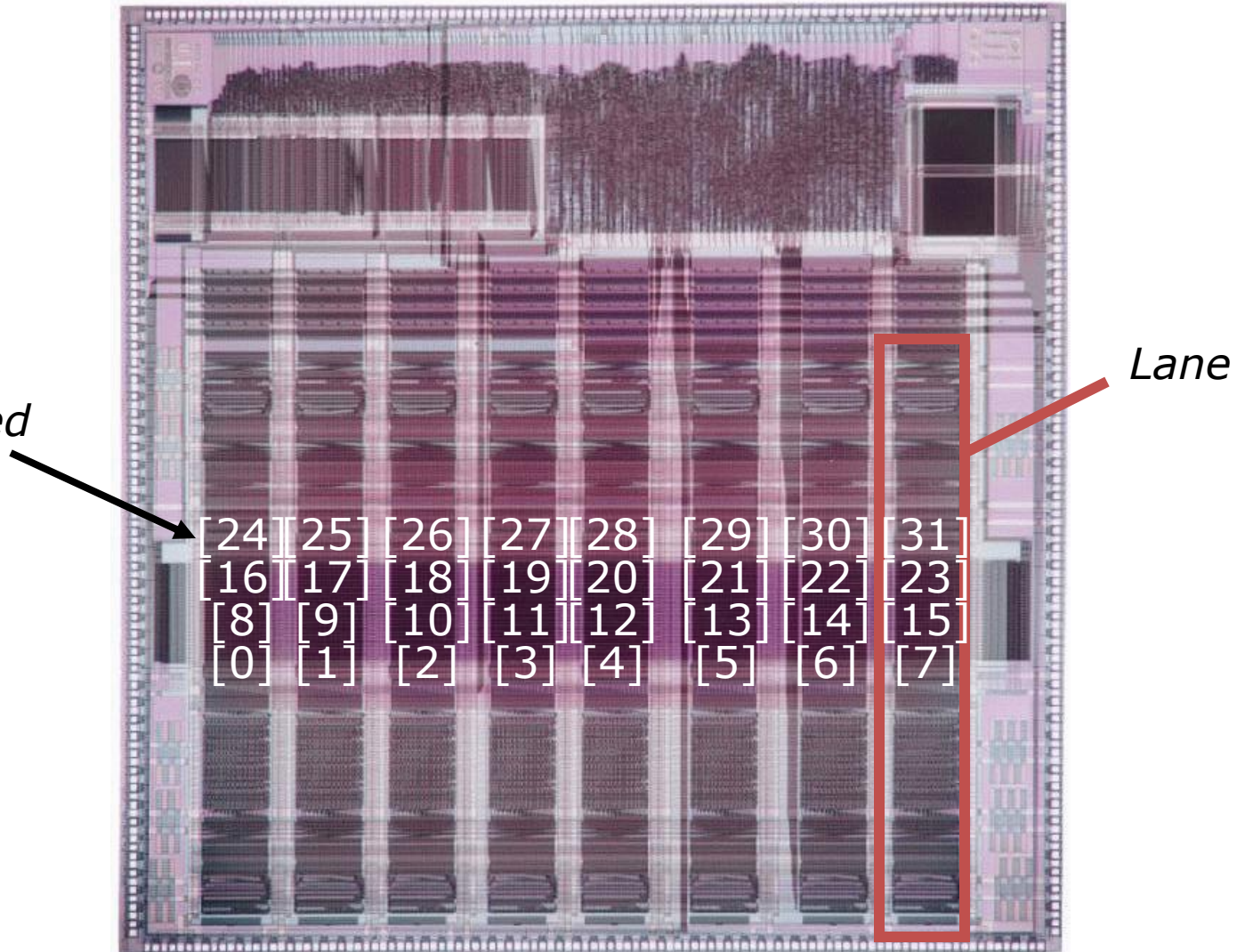


Photo of Berkeley T0, © University of California (Berkeley)  
<http://www1.icsi.berkeley.edu/Speech/spert/t0die.jpg>



# Vector Instruction Set Advantages

- Compact
  - one short instruction encodes N 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)
- Scalable
  - can run same code on more parallel pipelines (lanes)

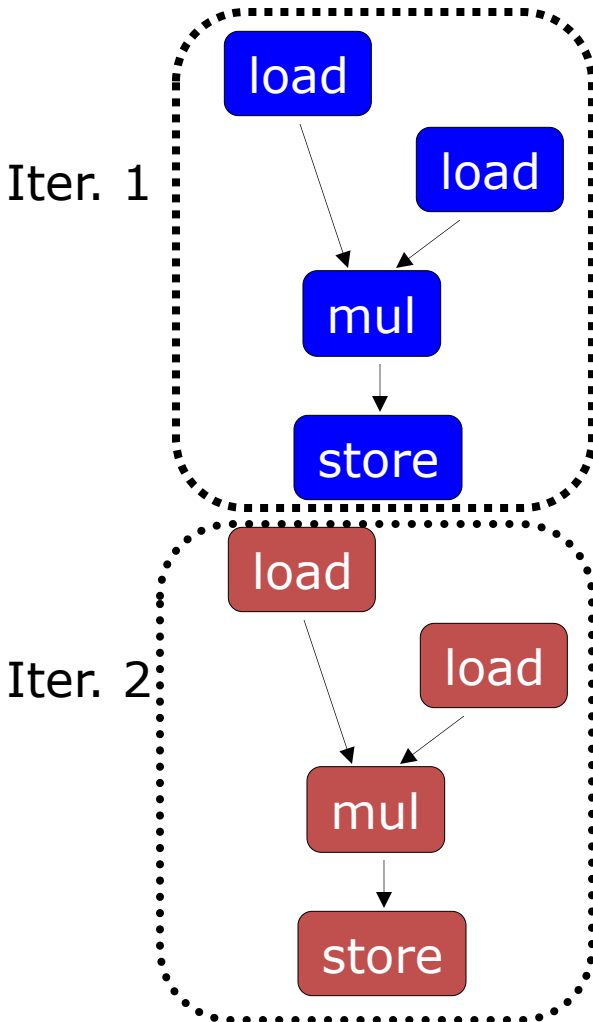
# Automatic Code Vectorization

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

# Automatic Code Vectorization

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

*Scalar Sequential Code*

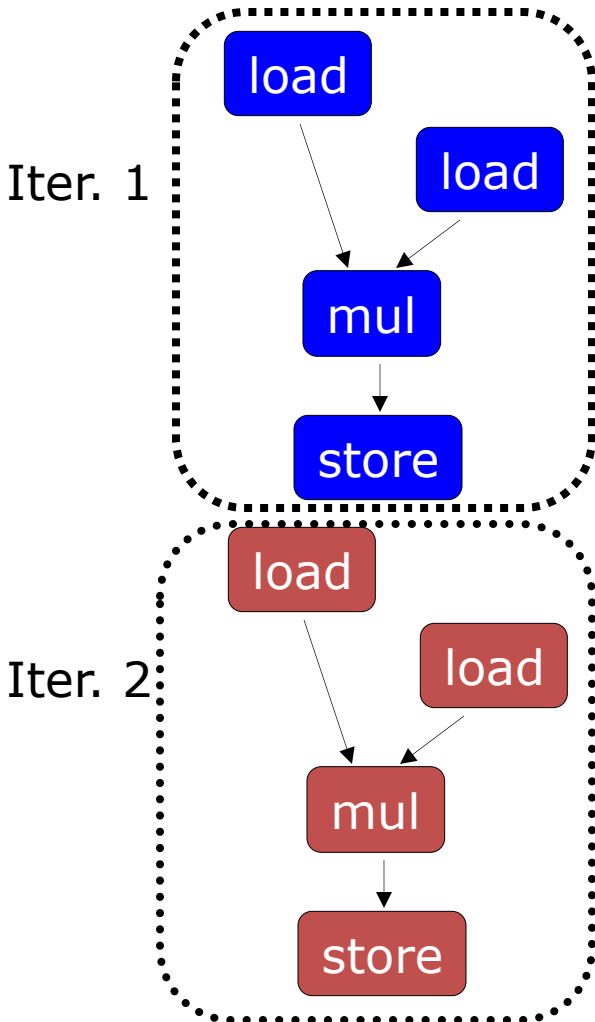


# Automatic Code Vectorization

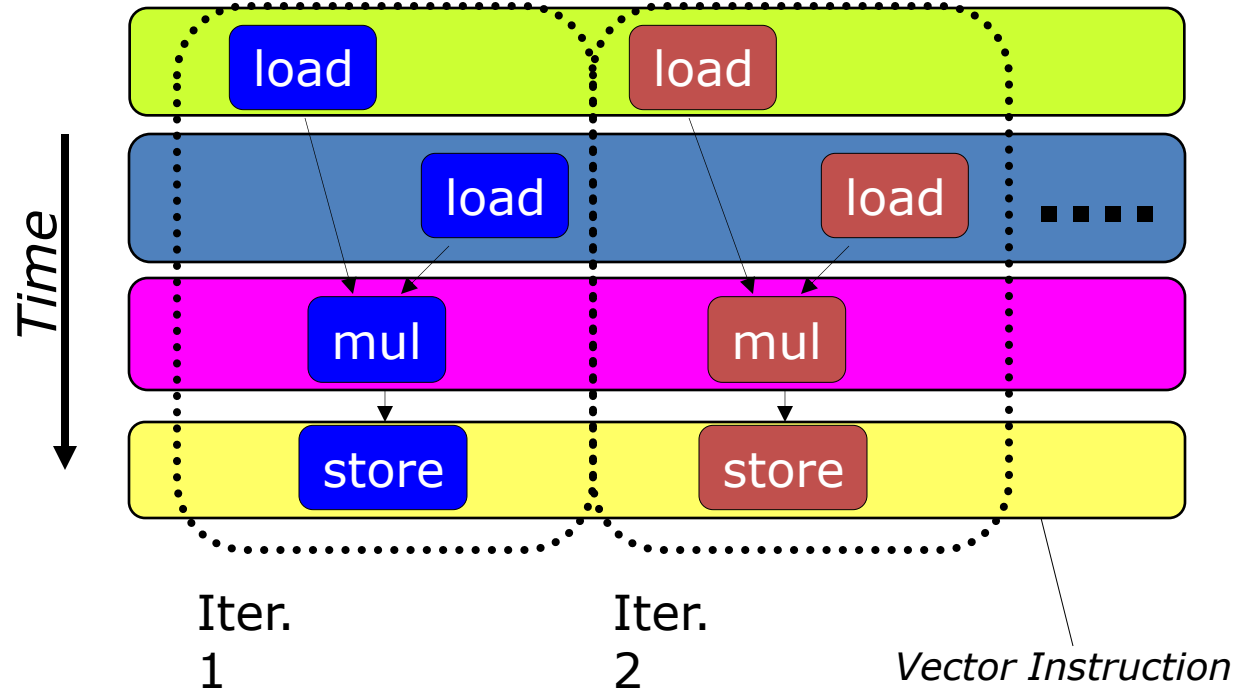
```
for (i=0; i < N; i++)
```

```
C[i] = A[i] * B[i];
```

## Scalar Sequential Code



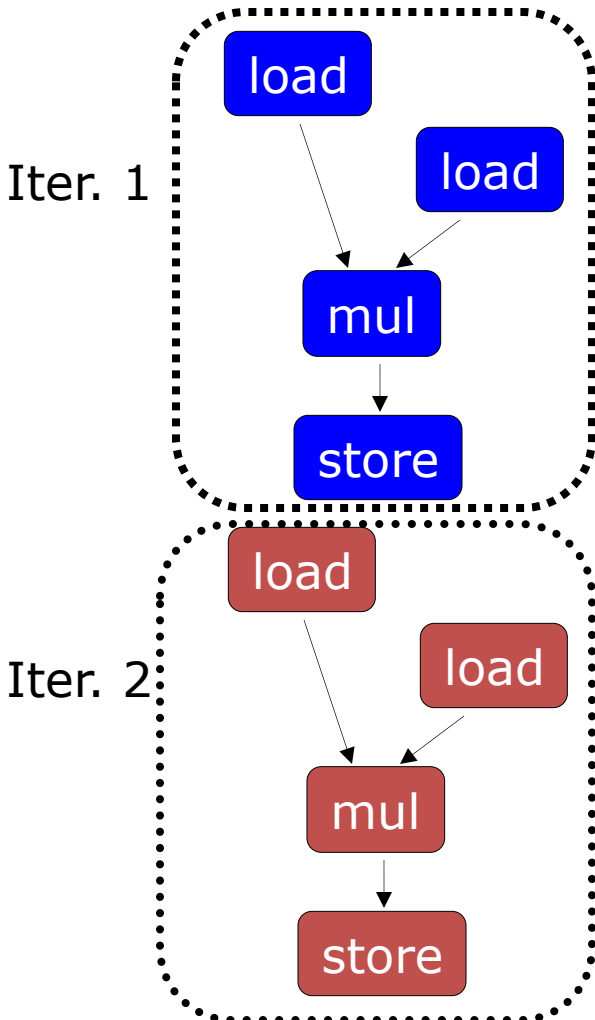
## Vectorized Code



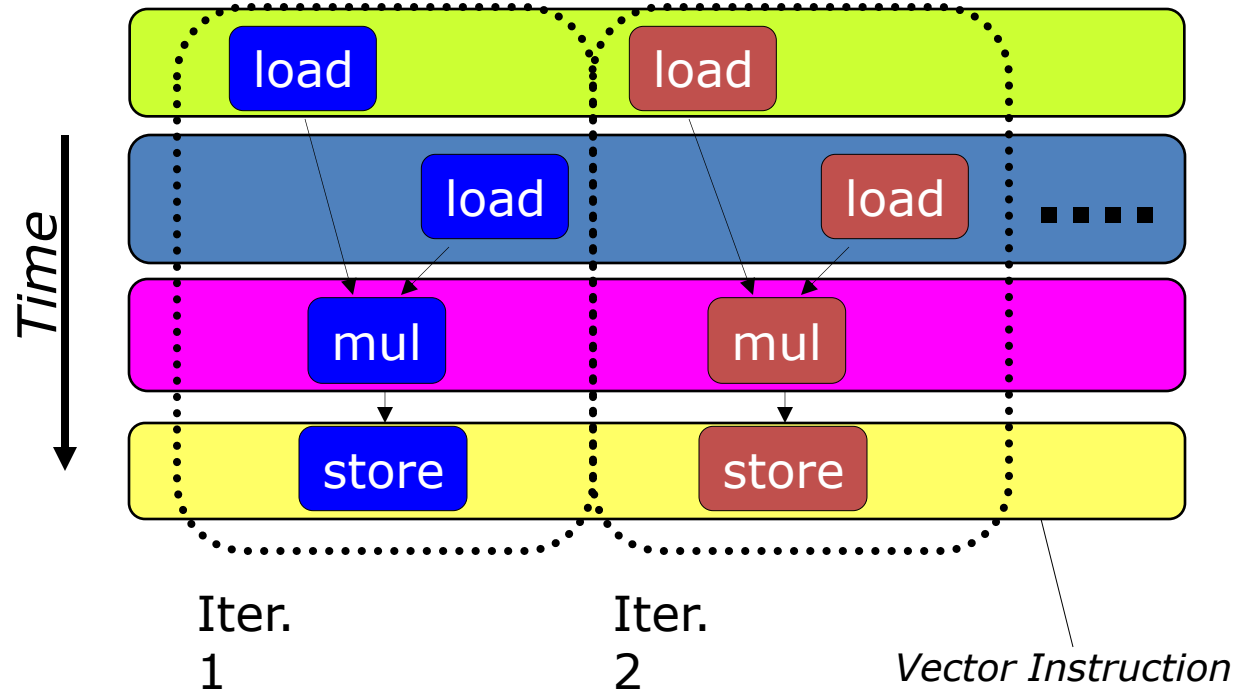
# Automatic Code Vectorization

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

*Scalar Sequential Code*



*Vectorized Code*



Vectorization is a massive compile-time reordering of operation sequencing  
⇒ requires extensive loop dependence analysis

# 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

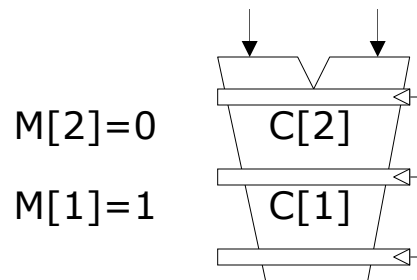
M[7]=1 A[7] B[7]

M[6]=0 A[6] B[6]

M[5]=1 A[5] B[5]

M[4]=1 A[4] B[4]

M[3]=0 A[3] B[3]



M[0]=0

*Write Enable*

C[0]

*Write data port*

# Masked Vector Instructions

## Simple Implementation

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

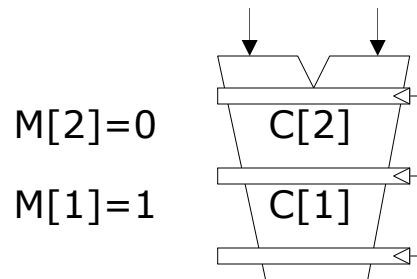
M[7]=1 A[7] B[7]

M[6]=0 A[6] B[6]

M[5]=1 A[5] B[5]

M[4]=1 A[4] B[4]

M[3]=0 A[3] B[3]



M[0]=0

Write Enable

Write data port

## Density-Time Implementation

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

M[7]=1

M[6]=0

M[5]=1

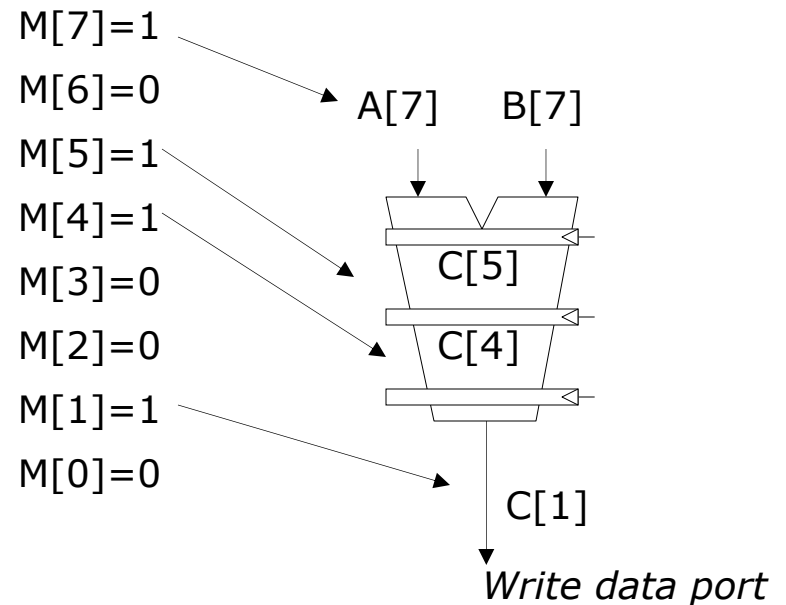
M[4]=1

M[3]=0

M[2]=0

M[1]=1

M[0]=0





# Vector Reductions

Problem: Loop-carried dependence on reduction variables

```
sum = 0;
for (i=0; i<N; i++)
    sum += A[i];  # Loop-carried dependence on sum
```

Solution: Re-associate operations if possible, use binary tree to perform reduction

```
# Rearrange as:
sum[0:VL-1] = 0                    # Vector of VL partial sums
for(i=0; i<N; i+=VL)              # Stripmine VL-sized chunks
    sum[0:VL-1] += A[i:i+VL-1];  # Vector sum
# Now have VL partial sums in one vector register
do {
    VL = VL/2;                    # Halve vector length
    sum[0:VL-1] += sum[VL:2*VL-1] # Halve no. of partials
} while (VL>1)
```

# 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 Supercomputers

*Epitomized by Cray-1, 1976:*

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

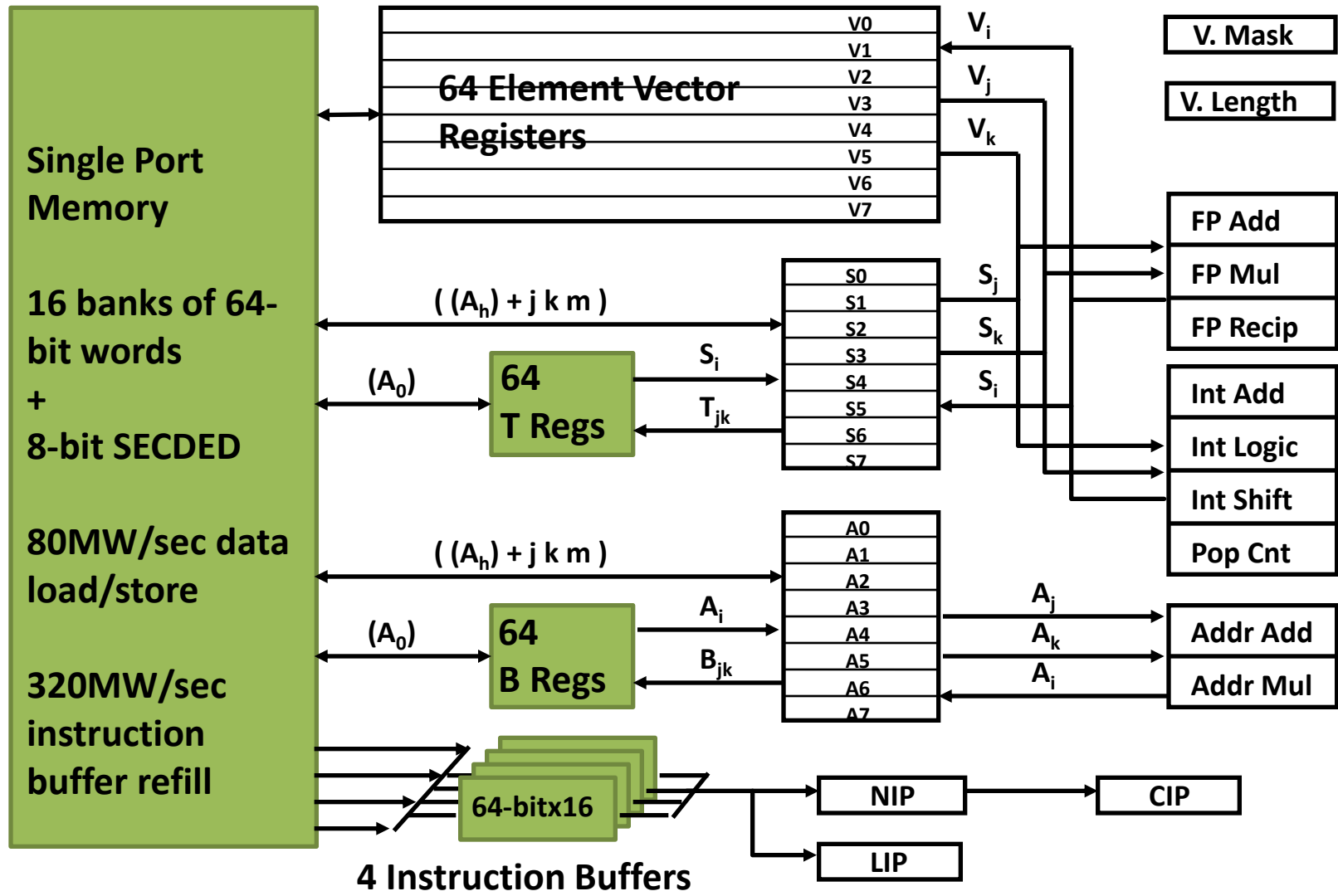


Cray 1 at The Deutsches Museum

Image Credit: Clemens Pfeiffer

<http://en.wikipedia.org/wiki/File:Cray-1-deutsches-museum.jpg>

# Cray-1 (1976)

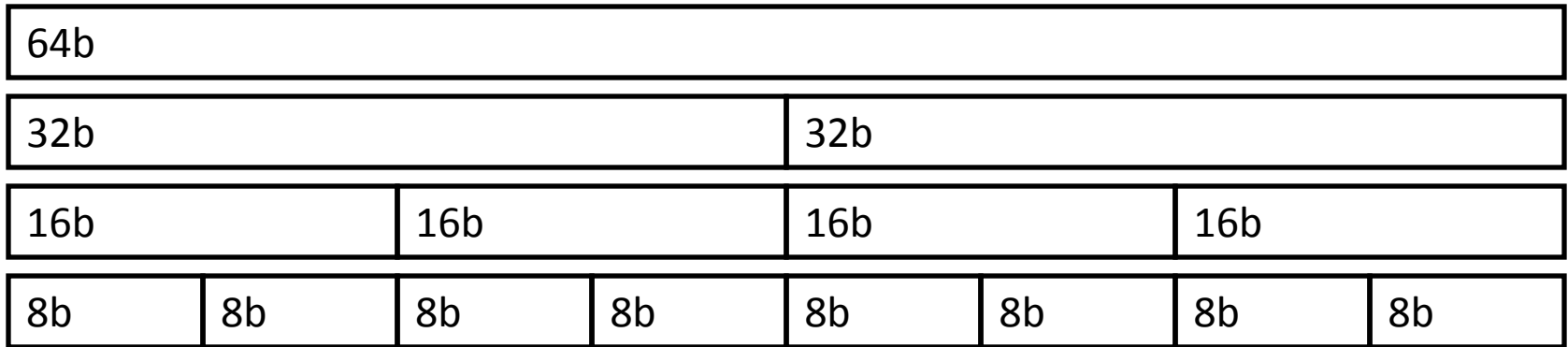


*memory bank cycle 50 ns    processor cycle 12.5 ns (80MHz)*

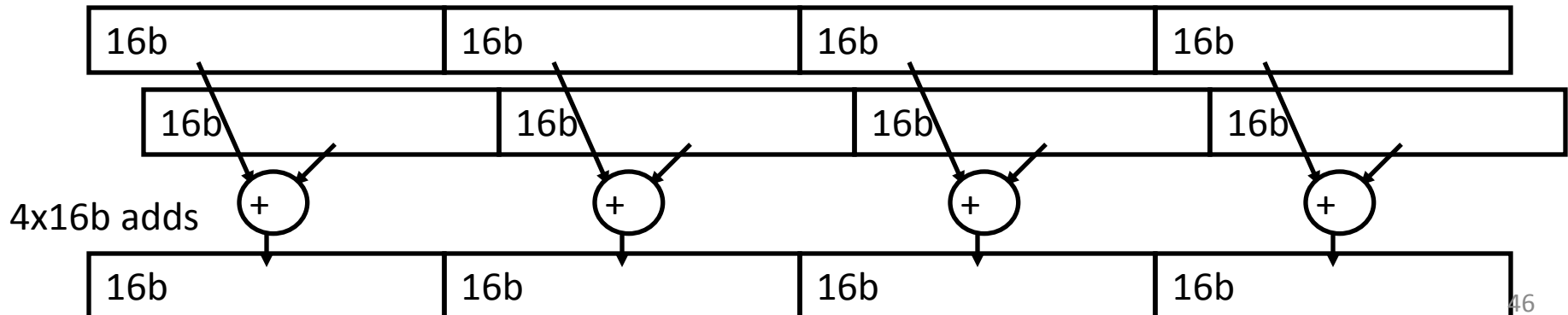
# Agenda

- Vector Processors
- Single Instruction Multiple Data (SIMD)  
Instruction Set Extensions
- Graphics Processing Units (GPU)

# SIMD / Multimedia Extensions



- Very short vectors added to existing ISAs for microprocessors
- Use existing 64-bit registers split into 2x32b or 4x16b or 8x8b
  - This concept first used on Lincoln Labs TX-2 computer in 1957, with 36b datapath split into 2x18b or 4x9b
  - Newer designs have 128-bit registers (PowerPC AltiVec, Intel SSE2/3/4) or 256-bit registers (Intel AVX)
- Single instruction operates on all elements within register



# Multimedia Extensions versus Vectors

- Limited instruction set:
  - no vector length control
  - no strided load/store or scatter/gather
  - unit-stride loads must be aligned to 64/128-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
  - Better support for misaligned memory accesses
  - Support of double-precision (64-bit floating-point)
  - New Intel AVX spec (announced April 2008), 256b vector registers (expandable up to 1024b)

# Agenda

- Vector Processors
- Single Instruction Multiple Data (SIMD)  
Instruction Set Extensions
- Graphics Processing Units (GPU)



# Graphics Processing Units (GPUs)

- Original GPUs were dedicated fixed-function devices for generating 3D graphics (mid-late 1990s) including high-performance floating-point units
  - Provide workstation-like graphics for PCs
  - User could configure graphics pipeline, but not really program it
- Over time, more programmability added (2001-2005)
  - E.g., New language Cg for writing small programs run on each vertex or each pixel, also Windows DirectX variants
  - Massively parallel (millions of vertices or pixels per frame) but very constrained programming model
- Some users noticed they could do general-purpose computation by mapping input and output data to images, and computation to vertex and pixel shading computations
  - Incredibly difficult programming model as had to use graphics pipeline model for general computation

# General Purpose GPUs (GPGPUs)

- In 2006, Nvidia introduced GeForce 8800 GPU supporting a new programming language: CUDA
  - “Compute Unified Device Architecture”
  - Subsequently, broader industry pushing for OpenCL, a vendor-neutral version of same ideas.
- Idea: Take advantage of GPU computational performance and memory bandwidth to accelerate some kernels for general-purpose computing
- Attached processor model: Host CPU issues data-parallel kernels to GP-GPU for execution
- This lecture has a simplified version of Nvidia CUDA-style model and only considers GPU execution for computational kernels, not graphics

# Simplified CUDA Programming Model

- Computation performed by a very large number of independent small scalar threads (*CUDA threads* or *microthreads*) grouped into *thread blocks*.

```
// C version of DAXPY loop.
```

```
void daxpy(int n, double a, double*x, double*y)
{   for (int i=0; i<n; i++)
        y[i] = a*x[i] + y[i]; }
```

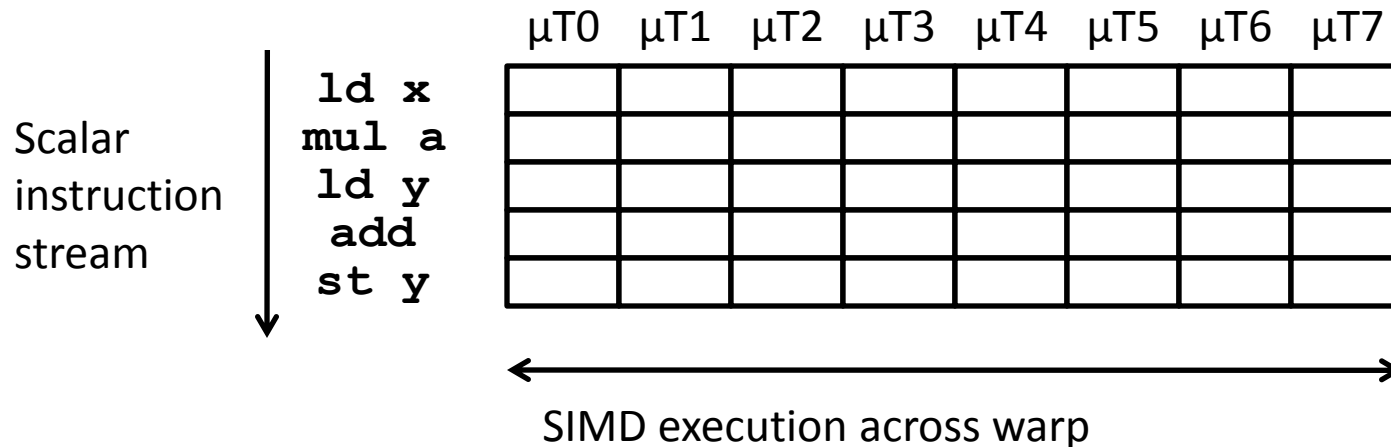
```
// CUDA version.
```

```
__host__ // Piece run on host processor.
int nblocks = (n+255)/256; // 256 CUDA threads/block
daxpy<<<nblocks,256>>>(n,2.0,x,y);
```

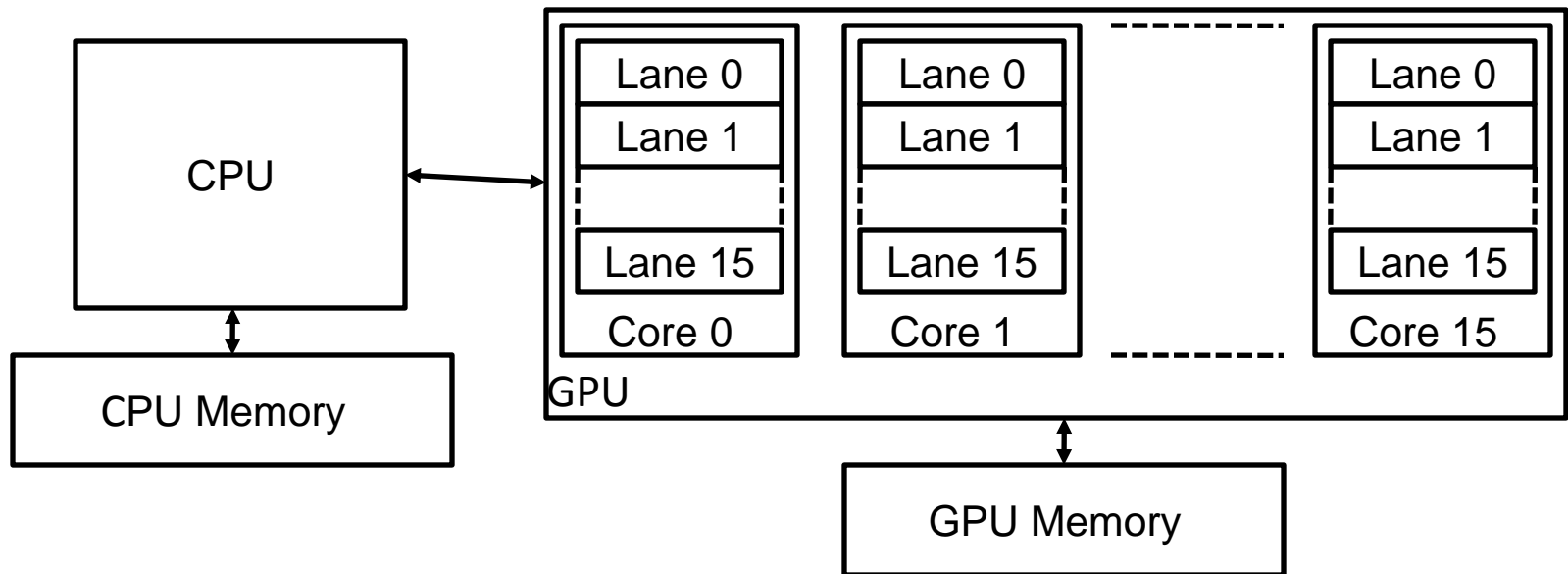
```
__device__ // Piece run on GPGPU.
void daxpy(int n, double a, double*x, double*y)
{   int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i<n) y[i]=a*x[i]+y[i]; }
```

# “Single Instruction, Multiple Thread”

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*)



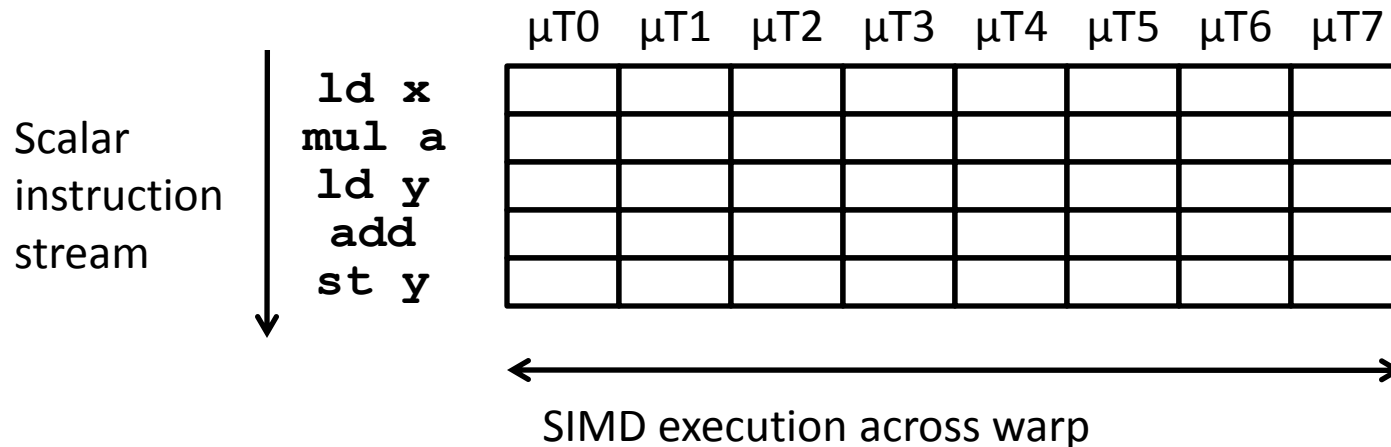
# Hardware Execution Model



- GPU is built from multiple parallel cores, each core contains a multithreaded SIMD processor with multiple lanes but with no scalar processor
- CPU sends whole “grid” over to GPU, which distributes thread blocks among cores (each thread block executes on one core)
  - Programmer unaware of number of cores

# “Single Instruction, Multiple Thread”

- GPUs use a SIMT model, where individual scalar instruction streams for each CUDA thread are grouped together for SIMD execution on hardware (Nvidia groups 32 CUDA threads into a *warp*)



# Implications of SIMT Model

- All “vector” loads and stores are scatter-gather, as individual  $\mu$ threads perform scalar loads and stores
  - GPU adds hardware to dynamically coalesce individual  $\mu$ thread loads and stores to mimic vector loads and stores
- Every  $\mu$ thread has to perform stripmining calculations redundantly (“am I active?”) as there is no scalar processor equivalent
- If divergent control flow, need predicates

# GPGPUs are Multithreaded SIMD

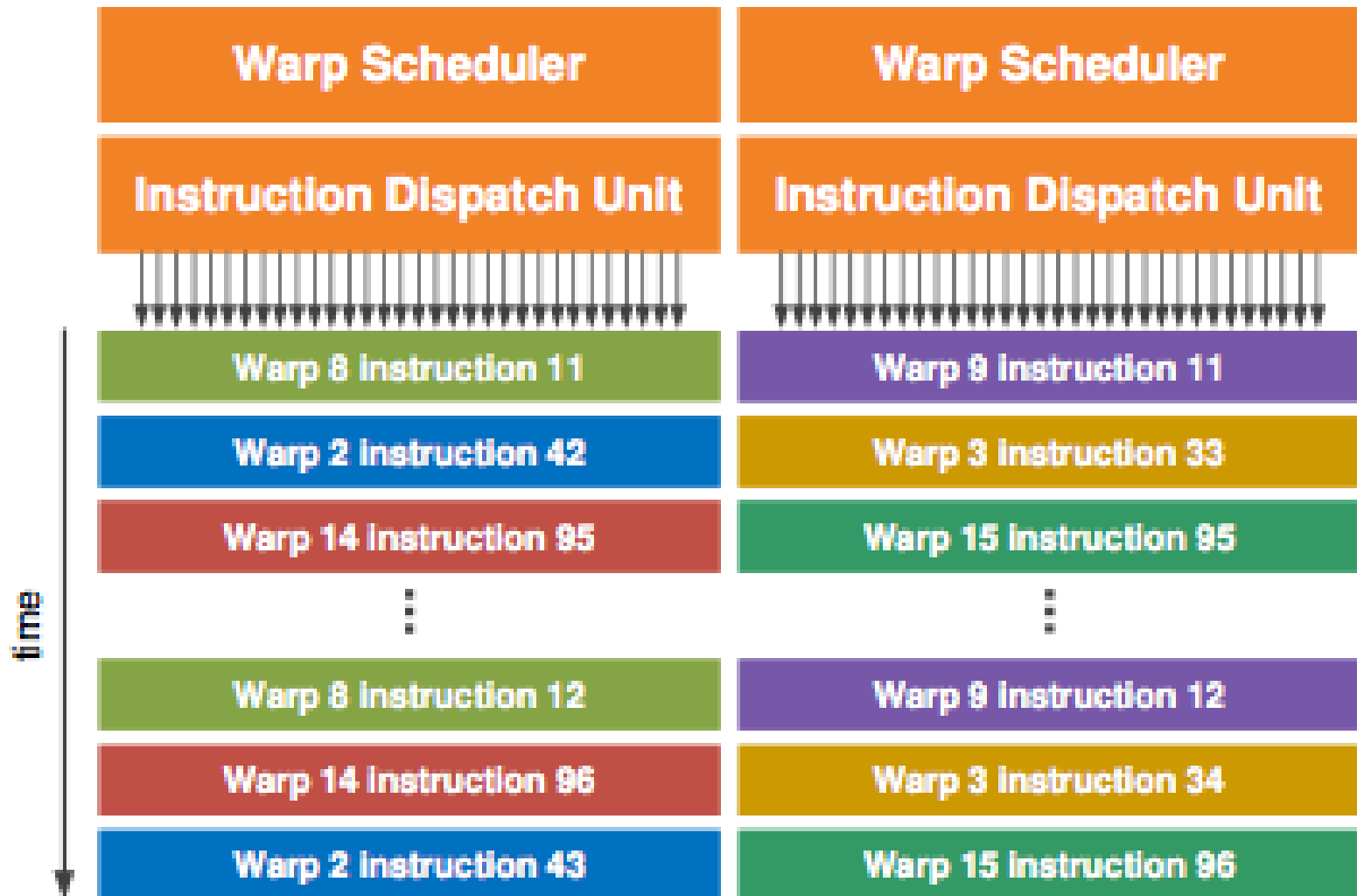


Image Credit: NVIDIA



# Nvidia Fermi GF100 GPU



Image Credit: NVIDIA [Wittenbrink, Kilgariff, and Prabhu, Hot Chips 2010]

# Fermi “Streaming Multiprocessor” Core

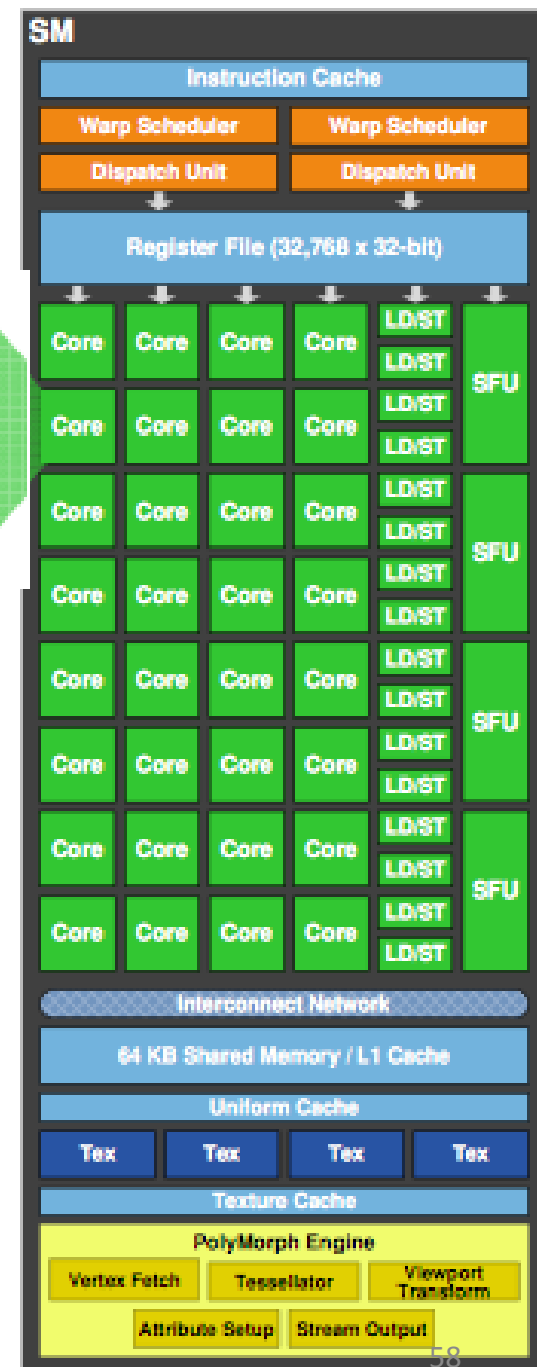
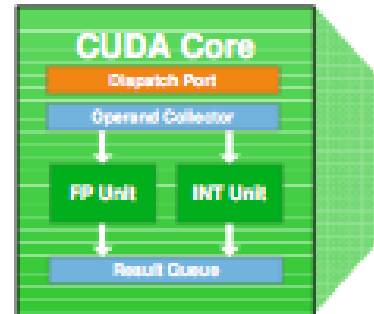


Image Credit: NVIDIA

[Wittenbrink, Kilgariff, and Prabhu, Hot Chips 2010]

# Acknowledgements

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)
  - Christopher Batten (Cornell)
- MIT material derived from course 6.823
- UCB material derived from course CS252 & CS152
- Cornell material derived from course ECE 4750

Copyright © 2013 David Wentzlaff