SIMD & Vector Processing
(≈ Data Parallel Programming)

Outline

1. Motivation

2. SIMD

3. Vector Processing (K. Asanovic)

Motivation

Recall:

- Modern CPUs significantly increase performance with deeper pipelines and superscalar execution (as seen so far)
- Tremendous effort required to extract instruction-level parallelism *(in addition to pipelined parallel execution units):* branch predictors, speculation, instruction windows, hit-under-miss, reorder buffers, rename register files, etc.

In fact:

- Let **n** be the number of instructions in flight
- Then all these resources must grow ***proportionally*** to handle **n**
  - *that is,* ***O(n)***

AND

- The logic to track the dependencies grows ***quadratically*** with **n**
  - *that is,* ***$O(n^2)$***

Therefore,

- there is a practical limit on dynamic methods of extracting ILP

Motivation, cont.

However →

- There exist many applications where much ILP *(>> 4)* ***can*** be extracted.

However →

- For those ***parallelizable*** applications, (usually) don't need all the hardware support – the parallelism is directly available
  - *and at multiple levels. Much more about this in a few weeks.*
- Most of chip area is wasted

Therefore →

- Design a processor explicitly for high parallelism applications.
  - Much more compute logic, much less speculation, etc.

However →

- Are there enough of these par apps to be economically viable?

Possible solutions →

- Bundle these "high parallelism" processors together with traditional CPUs
- Add a high parallelism unit onto the CPU

Some Questions

Recall the analysis we did to motivate memory hierarchy. Let's try that now to motivate high-throughput, highly-parallel processors.

1. **Need for new CPU designs**
   - Current processor cores use only a small fraction of their chip area on computing
   - Operating frequency is bounded by power/thermal constraints

2. **Properties of programs**
   - How much parallelism is there?
   - What is the granularity?
   - Can the program be restructured to expose more parallelism?

3. **Availability of Technology**
   - Huge numbers of processor types: multicore, many core, massively parallel processors, networks of PCs, hybrid, GPU, FPGA, vector, SIMD, systolic arrays, etc.

4. **Economics**
   - How big is the application space? Is there a single killer app space? How well do they match various possible processors?
   - Can we leverage existing technology?

Part 1: SIMD, Dataparallel, etc.

SIMD architecture and ***dataparallel programming*** are complementary ideas that were developed independently, but each follows naturally from the other.

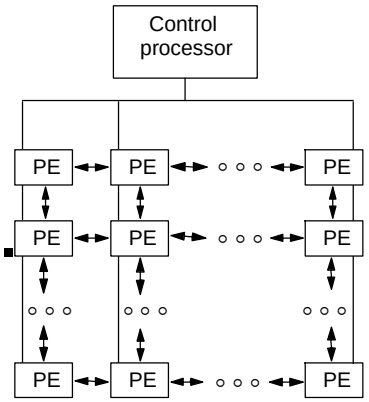
Architecture: Flynn's Taxonomy (late 1960s)

	Single Data Stream	Multiple Data Streams
Single Instruction Stream	SISD	SIMD
Multiple Instruction Streams	MISD	MIMD

Later (1980s), SPMD → Single ***Program*** Multiple Data

- first SIMD computers -- designed 1957 (Holland, Unger)
- built in 1963, the Illiac-III (McCormick)

SIMD Architecture



Single instruction sequencer, multiple datapaths.

Capabilities:

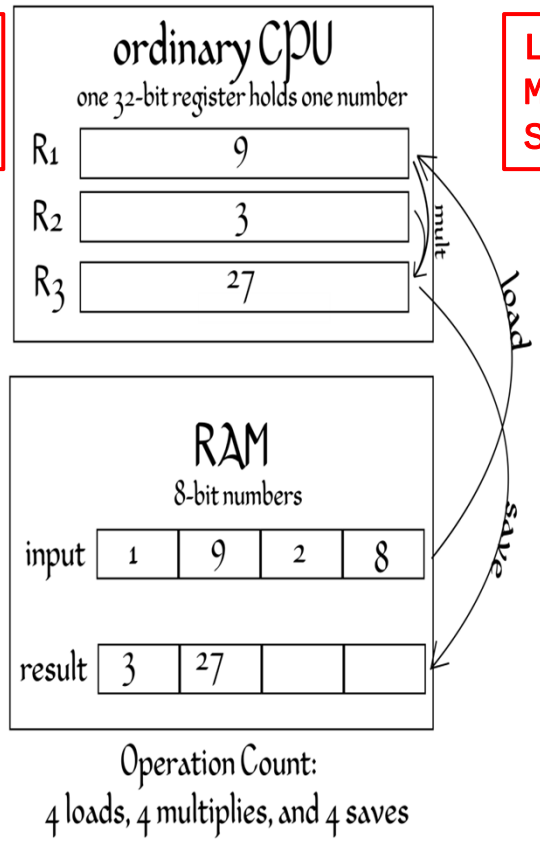
- Element-wise parallel computations in a single instruction
- Communication with neighboring elements for very fast stencil operations, wavefront operations, collective operations, etc.
- Branching through “activity control”
  - condition computed independently in each PE
  - conditional clause executed
  - Only PEs with correct condition store the result
- Often a more powerful inter-PE communication network
- Benefit: cheap PEs, massive parallelism, hugely efficient use of silicon (for the right applications)

Status:

- Large scale SIMD is not techno-politically viable (not based on a commodity)
- Small-scale SIMD is ubiquitous: routers, SSE, GPUs, FPGAs, Cell, etc.

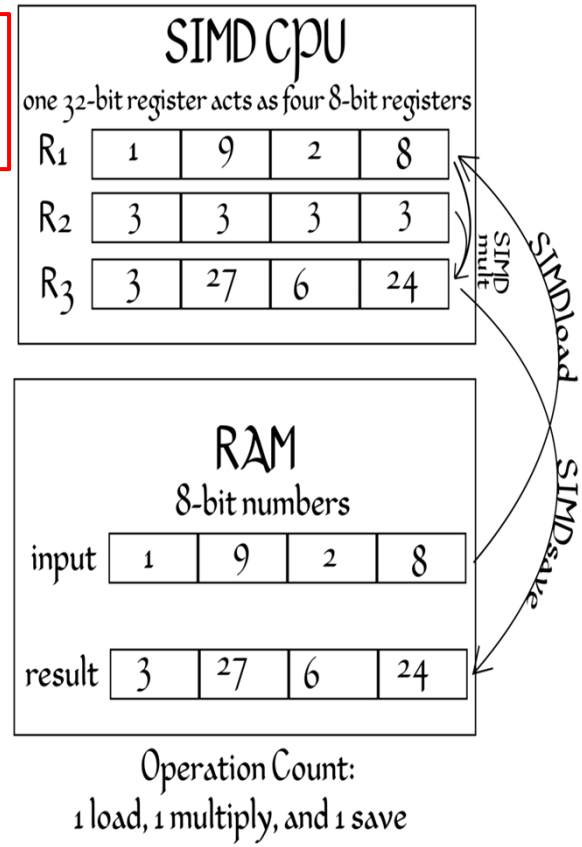
SIMD Operation *(figs from Wikipedia)*

```
LD   R1,input++
MUL  R1,R1,3
ST   R1,output++
```



The ordinary tripling of four 8-bit numbers. The CPU loads one 8-bit number into R1, multiplies it with R2, and then saves the answer from R3 back to RAM. This process is repeated for each number.

```
LD_S   R1,input++
MUL_S  R1,R1,3
ST_S   R1,output++
```

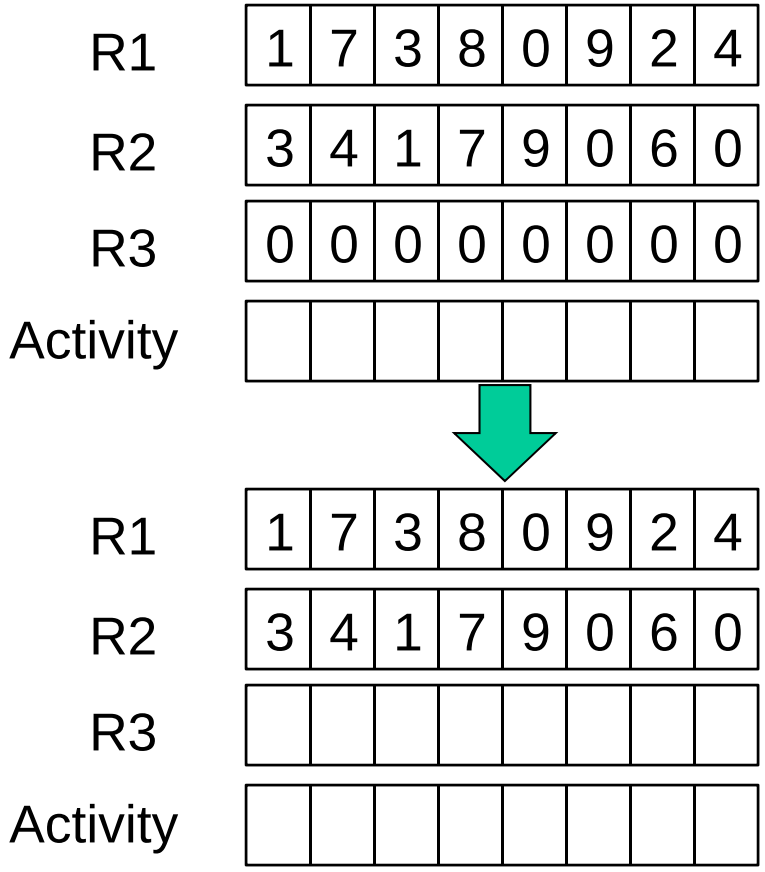


The SIMD tripling of four 8-bit numbers. The CPU loads 4 numbers at once, multiplies them all in one SIMD-multiplication, and saves them all at once back to RAM.

Note: Machine primitives are logical extensions of typical datapath operation – arithmetic & memory operation

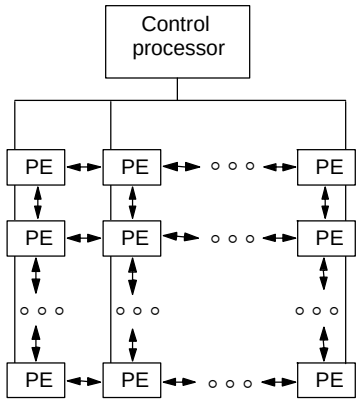
SIMD – Dealing w/ conditional execution

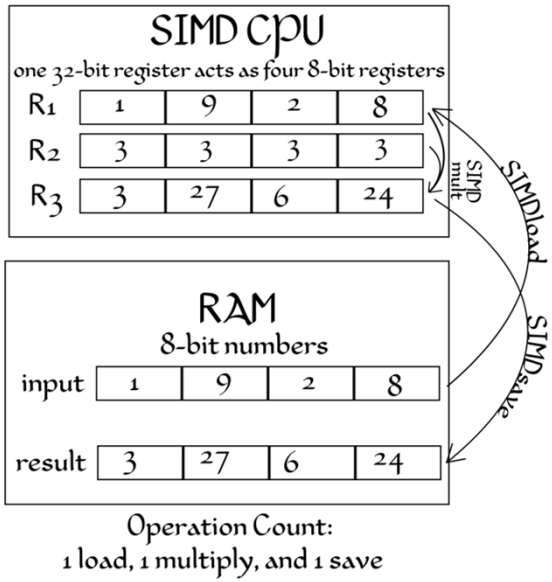
- In this example, each register has 8 slots
- "Activity" register (AR) settings determine whether output to a particular register slot actually gets written.
  → *That is, a 1 in an AR slot write-enables the corresponding slot in the destination register.*



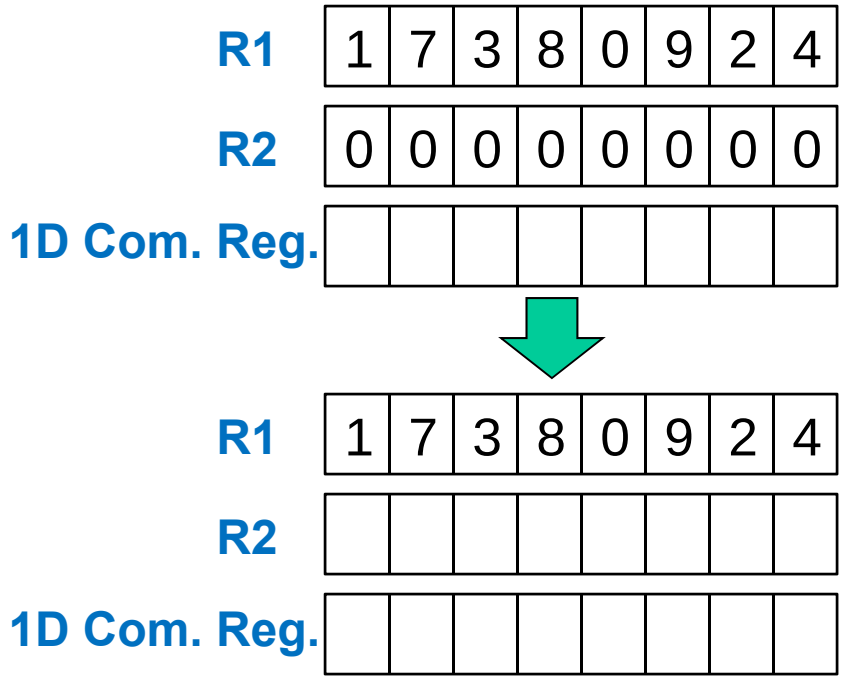

```
// Dataparallel
// (SIMD) instructions
SLT     R2,R1     // set in AR
ADDC    R3,R2,R1  // cond. add
```

SIMD – operations within a register





"Classic" SIMD supports direct inter-PE (intra-register) communication




```
MOV   R1,CR
ROTL  CR
MOV   CR,R2
```

Note two programming models:

1. Multiple PEs
2. Multi-Element registers

SIMD – Possible Communication Nets

SIMD Communication Network ⇔ A mechanism that enables

1. PEs to communicate with each other directly
2. Register elements to be permuted or otherwise reordered or combined.

Typical operations:

- → **combine rows**
- → **Shift/rotate**
- → **transpose**
- → **permute**
- → **broadcast in row by activity ctl**

Input

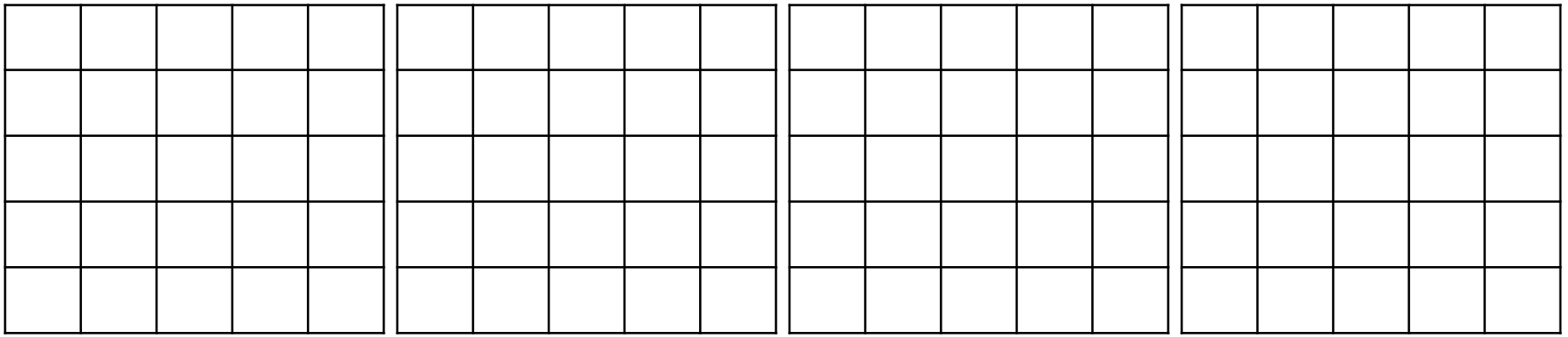
6	23	5	8	9
4	88	56	34	23
5	6	8	2	23
6	23	5	8	9
4	88	56	34	23

Permutation

5	4	3	22	1
6	9	11	12	2
13	15	25	8	7
10	24	21	23	16
20	19	18	14	17

Activity

1	0	0	0	0
1	0	0	0	0
0	0	1	0	0
1	0	0	0	0
0	1	0	1	0



Outputs

SIMD – Application Example

Each element of the output array ← gets the average of the four neighboring elements of the input array.

```
1. temp = input shifted left
   output = temp
2. temp = input shifted right
   output += temp
3. temp = input shifted up
   output += temp
4. temp = input shifted down
   output /= 4;
```

6	23	5	8	9
4	88	56	34	23
5	6	8	2	23
6	23	5	8	9
4	88	56	34	23

	22.5	16.9		

4	88	56	34	23
5	6	8	2	23
6	23	5	8	9
4	88	56	34	23
6	23	5	8	9

1. temp

4	88	56	34	23
6	23	5	8	9
4	88	56	34	23
5	6	8	2	23
6	23	5	8	9

2. temp

9	6	23	5	8
23	4	88	56	34
23	5	6	8	2
9	6	23	5	8
23	4	88	56	34

3. temp

23	5	8	9	6
88	56	34	23	4
6	8	2	23	5
23	5	8	9	6
88	56	34	23	4

4. temp

Example – possible SIMD networks

- PEs have communication interface registers for data and, if needed, communication info

  comm reg, perm matrix in previous examples

- Communication network independent of processors →

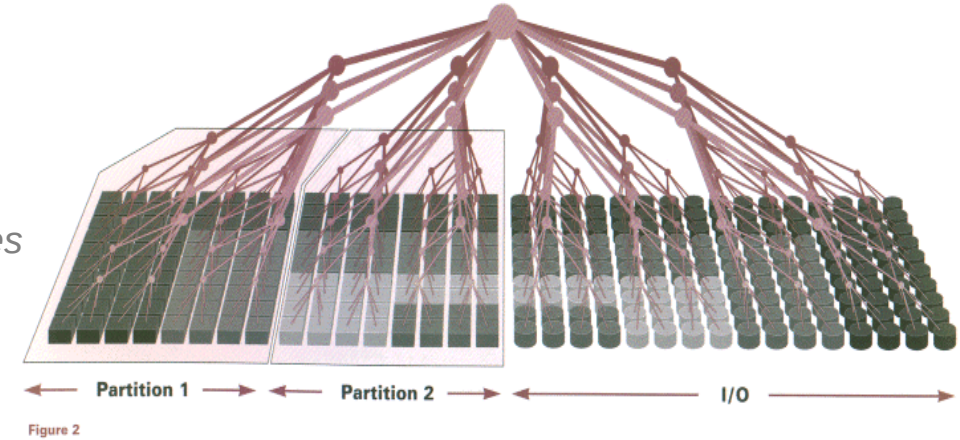


Figure 2

Sample inter-PE networks

Topology	Computer……
None	NVIDIA GPUs
1D	CLIP7, ASP
2D	MPP, CM1
3D	Ultracube
Arb. Permutation	CM2
Broadcast	DAC, CLIP4, CAAPP
Reduction	CM5, CLIP4

Dataparallel Programming

Idea: Operate on ***collections*** of data as units.

- ***Obvious now, obvious then …***
  - (**APL** in the early 1960s had everything!), but continues to get re-re-re-re-re-invented. *>> Anybody hear about **Map-Reduce**???*

Practice:

- Operations performed in parallel on each element of data structure
- Logically a single thread of control which successively performs operations on single elements or collections of elements
- Typical operations on collections:
  - element-wise on pairs of collections: arithmetic, data transfer, conditionals, etc.
  - scalar-element-wise on scalar plus collection
  - within a collection of elements: reductions, parallel prefix, etc.
  - reordering elements in a collection: "sliding planes", permutations, etc.

Status: Now part of almost all programming languages (or their common extensions)

Example of Data Parallel App

- Let **`salary`** be a collection of elements (e.g., *an array*) where each element is the salary of a different employee
  - *In SIMD, each PE contains an employee record with his/her salary*

```
If salary > 100K then      // element-wise compare
   salary = salary * 1.05  // element-wise *
else
   salary = salary * 1.10  // element-wise *
MaxSalary += salary   // MaxSalary is a scalar
                      // += overloaded to mean combine
```

- Logically, each operation is a single step
- Some processors enabled for arithmetic operation, others disabled

- Other examples:
  - Finite differences, linear algebra, ...
  - Document searching, graphics, image processing, ...

Applications of Data Parallelism, cont.

```
int  i,j,k                  // declare three integer scalars
Array(512,512) A,B,C;       // declare three objects of type Array

// basic unary and binary operators (+,=,-, etc.) are overloaded
// for Array types they translate into element-by-element operations
   A = B + C;
   IF (A < 0) A = -A;

// direction methods (east, west, north, south) mean that we get the
// "plane" that has been "slid" from that direction.  "west" is a C++ method.
   B = west.A;

// max and sum are reduction operators.  That is, they combine a set
// of data into a single scalar.  "max" and "sum" are C++ methods.
   i = max.A;
   j = sum.B;

// averaging filter -- each element of C gets the average of the 5 corresponding
// elements of A
   C = .2*A + .2*west.A + .2*east.A + .2*north.A + .2*south.A;
```

Part 2: Vector Architecture

What are the advantages?

1. **Process chunks of data (1D arrays or vectors) as units**
   - SIMD style: element-element, conditionals
   - Vector registers hold fixed-length vectors of data

2. **Trade-off latency for bandwidth: long pipelines**
   - High-bandwidth memory, e.g., massively interleaved banks
   - Vector processing units (VPUs) are deeply pipelined (high throughput)

3. **Reuse data through "chaining"**
   - "pipe" vectors from memory to VPU to VPU to VPU to memory …

4. **All hardware under programmer control**
   - Opposite of ILP/Cache model of current high-end cores

5. **For right applications** *(HPC programmers know them),* **compilers** *(working with programmers)* ***get excellent results***

Problems with conventional CPU approach

Limits to conventional exploitation of ILP:

1) ***pipelined clock rate*****:** at some point, each increase in clock rate has corresponding CPI increase (branches, other hazards)

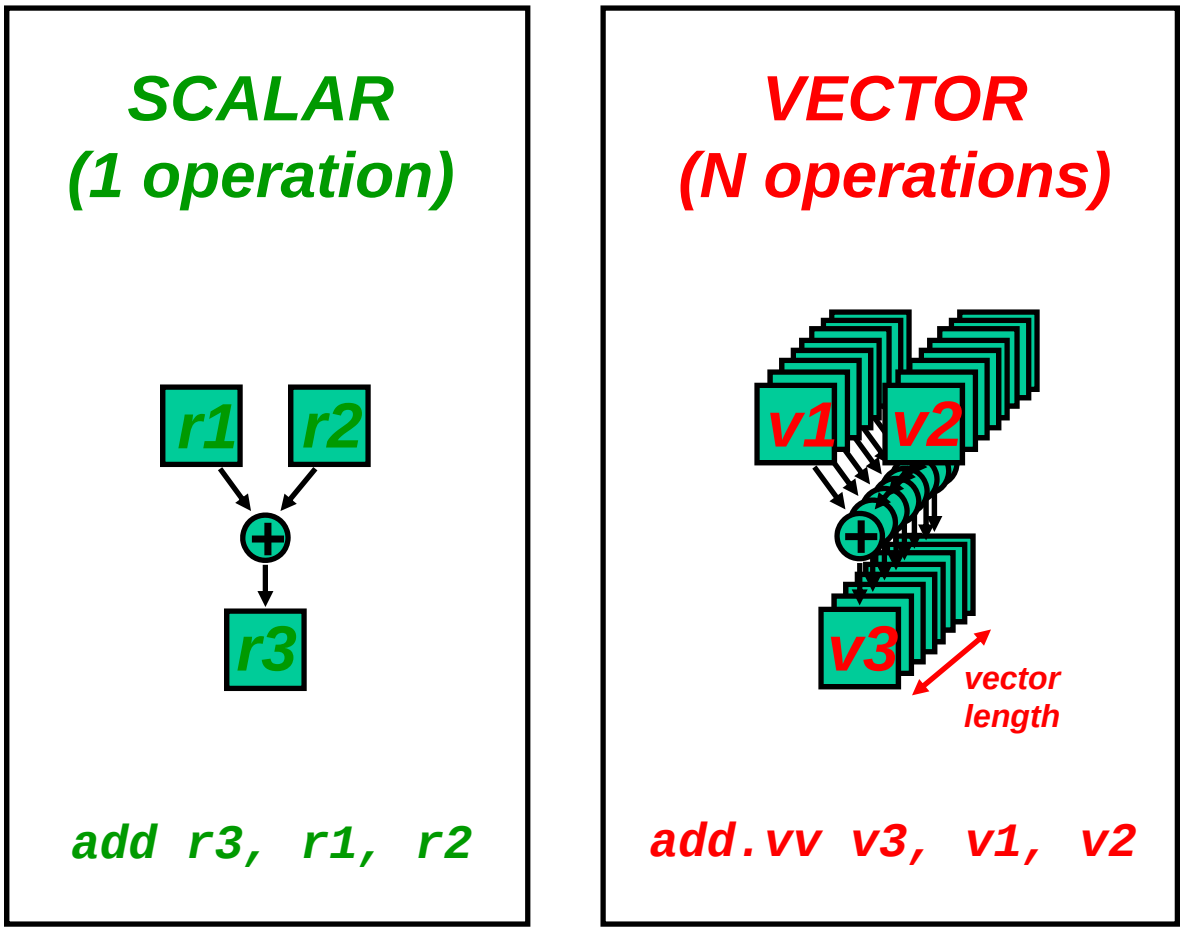
2) ***instruction fetch and decode*****:** at some point, it is hard to fetch and decode additional instructions per clock cycle (dependency checks)

3) ***cache hit rate*****:** some long-running (scientific) programs have very large data sets accessed with poor locality; others have continuous data streams w/ one-time use (multimedia) and hence poor locality

Problem is ***latency versus throughput*** -- High end CPUs are optimized to perform all programs very fast and so have huge resources devoted to branch prediction, prefetch, speculative execution, etc.

Alternative Model: Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"

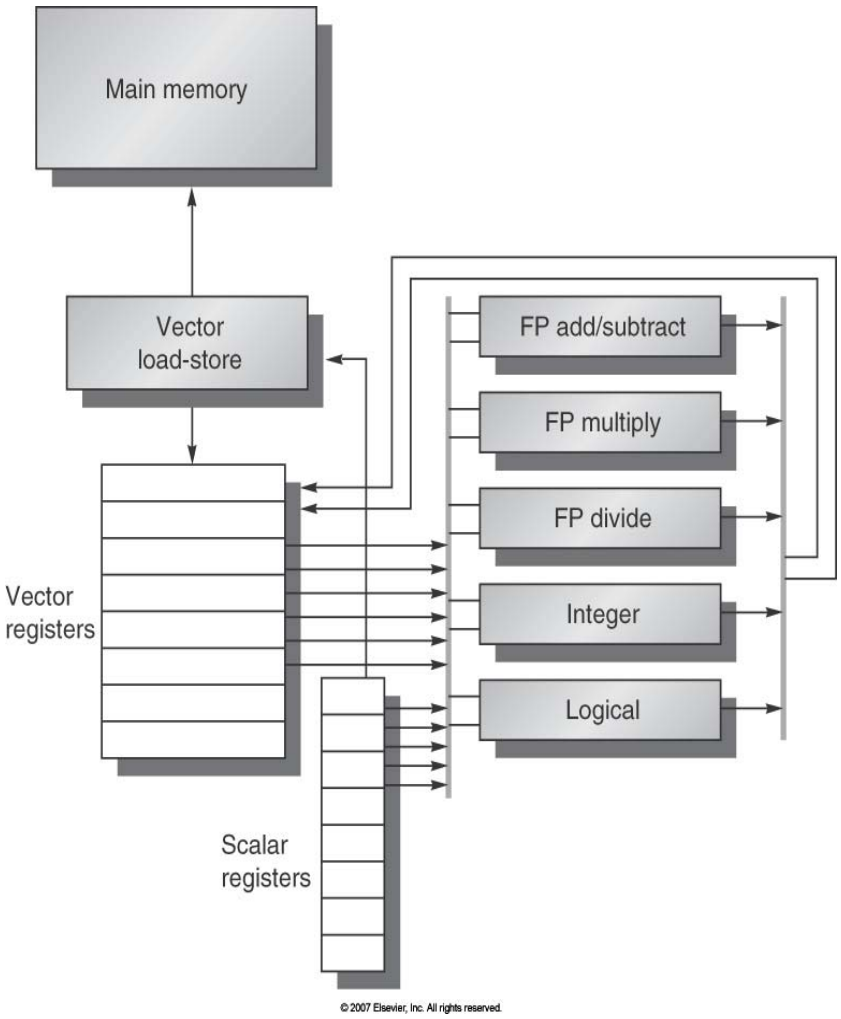


Properties of Vector Processors

- **Each result is independent of previous result**
  - *compiler ensures no dependencies* *>> feature, not a bug!?*
  - high clock rate, long pipeline
- **Vector instructions access memory that has known configuration**
  - highly interleaved memory
  - amortize memory latency of over ~ 64 elements
  - no (data) caches required! (Do use instruction cache)
- **Reduces branches and branch problems in pipelines**
  - *Much simpler branch prediction hardware (if any)*
- **Single vector instruction implies lots of work (loop)**
  - fewer instruction fetches per work done
  - Much simpler fetch and decode per work done
  - *Note – this is an essential aspect of GPU programming*

How is all of this "magic" possible?
Must have data independence available in the application and built into the code.

Components of Vector Processors



- *Vector Registers*: fixed length banks holding single vectors
  - has at least 2 read and 1 write ports
  - typically 8 - 32 vector registers, each holding 64 - 128 64-bit elements
- *Vector Functional Units (FUs)*: fully pipelined, start new operation every clock
  - typically 4 to 8 FUs: FP add, FP mult, FP reciprocal (1/X), integer add, logical, shift; may have multiple of same unit
- *Vector Load-Store Units (LSUs)*: fully pipelined unit to load or store a vector; may have multiple LSUs
- *Scalar registers*: single element for FP scalar or address
- Cross-bar to connect FUs, LSUs, registers

“DLXV” Vector Instructions

Instr.	Operands	Operation	Comment
ADDV	**V1,V2,V3**	V1=V2+V3	vector + vector
MULTV	**V1,V2,V3**	V1=V2xV3	vector x vector
ADDSV	**V1,F0,V2**	V1=F0+V2	scalar + vector
MULSV	**V1,F0,V2**	V1=F0xV2	scalar x vector
LV	**V1,R1**	V1=M[R1..R1+63]	load, stride=1
LVWS	**V1,R1,R2**	V1=M[R1..R1+63*R2]	load, stride=R2
LVI	**V1,R1,V2**	V1=M[R1+V2i,i=0..63]	indir.("gather")
CeqV	**VM,V1,V2**	VMASKi = (V1i=V2i)?	comp. set vec mask
MOV	**VLR,R1**	Vec. Len. Reg. = R1	set vector length
MOV	**VM,R1**	Vec. Mask = R1	set vector mask

Memory operations

- Load/store operations move data between registers and memory
- Three types of addressing
  - Unit stride *>> fastest*

    EX: **LV V1,Rx ; load vector register V1 with a vector of**
    ; contiguous data whose base address is in Rx

  - Non-unit (but constant) stride

    EX: **LVWS V1,Rx,Rs ; load vector register V1 with a vector of data**
    ; whose base address is in Rx, but with stride Rs

  - Indexed (gather-scatter)

    EX: **LVI V1,Rx,V2 ; load vector register V1 with a vector of data**
    ; whose base address is in Rx, but with element-
    ; wise offsets in V2

    - Vector equivalent of register indirect
    - Good for sparse arrays of data
    - Increases number of programs that vectorize

Vector Load/Store Units & Memories

- Start-up overheads usually longer for Load/Store Units (LSUs)
- Memory system must sustain (# lanes x word) /clock cycle

- Many Vector Processors use banks (versus simple interleaving):

  1) support multiple loads/stores per cycle
  => multiple banks & address banks independently

  2) support non-sequential accesses (see soon)

- Note: # memory banks **>** memory latency to avoid stalls
  - *m* banks ➔ *m* words per memory w/ latency = *l* clocks
  - if *m* **<** *l*, then gap in memory pipeline:

clock:	0	…	*l*	*l*+1	*l*+2	…	*l*+*m*- 1	l+m	…	2 *l*
word:	--	…	0	1	2	…	*m*-1	--	…	*m*

  - may have 1024 banks in SRAM

Memory Banks

Memory Banks versus Interleaved Memory

→ Memory Banks have fully independent control and access

Why Memory Banks?

→ Many processors can share the same memory system (and get good performance)

→ Need to load/store data that are not sequential

Why multiple memory banks?

→ Memory cycle time is generally longer than CPU cycle time

Memory Bank Example 1

Example: Cray T90

- The CPU clock period = 2.167 ns
- In its largest configuration (Cray T932) has 32 processors
- Each processor is capable of generating 4 loads and 2 stores per cycle.
- The cycle time of the SRAMs used in the memory system is 15 ns.

Question

What is the minimum number of memory banks required to allow all CPUs to run at full memory bandwidth?

Answer

- The maximum number of memory references each cycle is 192
  - (32 CPUs times 6 references per CPU).
- Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles (7 clock cycles)

Therefore, we require a minimum of 192 × 7 = 1344 memory banks!

The Cray T932 actually has 1024 memory banks, and so the early models could not sustain full bandwidth to all CPUs simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

Memory Bank Example 2

Example

- we want to fetch a vector of 64 elements
- starting at byte address 136
- a memory access takes 6 clocks

Q: How many memory banks must we have to support one fetch per clock cycle?
Q: With what addresses are the banks accessed?
Q: When will the various elements arrive at the CPU?

Answer

Six clocks per access require at least six banks, but because we want the number of banks to be a power of two, we choose to have eight banks.

Note

Early research proposed having a prime number of memory banks, e.g., 17, to reduce to probability of systematic conflict.

	Bank							
Cycle no.	0	1	2	3	4	5	6	7
0		136						
1		busy	144					
2		busy	busy	152				
3		busy	busy	busy	160			
4		busy	busy	busy	busy	168		
5		busy	busy	busy	busy	busy	176	
6			busy	busy	busy	busy	busy	184
7	192			busy	busy	busy	busy	busy
8	busy	200			busy	busy	busy	busy
9	busy	busy	208			busy	busy	busy
10	busy	busy	busy	216			busy	busy
11	busy	busy	busy	busy	224			busy
12	busy	busy	busy	busy	busy	232		
13		busy	busy	busy	busy	busy	240	
14			busy	busy	busy	busy	busy	248
15	256			busy	busy	busy	busy	busy
16	busy	264			busy	busy	busy	busy

Figure F.7 Memory addresses (in bytes) by bank number and time slot at which access begins. Each memory bank latches the element address at the start of an access and is then busy for 6 clock cycles before returning a value to the CPU. Note that the CPU cannot keep all eight banks busy all the time because it is limited to supplying one new address and receiving one data item each cycle.

DAXPY (Y = a * X + Y)

Assuming vectors X, Y are length 64

Scalar vs. Vector

```
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(i)
      LD      F4, 0(Ry)   ;load Y(i)
      ADDD    F4,F2,F4    ;a*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
```

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

578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)

578 (2+9*64) vs.
6 instructions (96X)

64 operation vectors + no loop overhead

also 64X fewer pipeline hazards

Vector Length

What to do when vector length n is not exactly 64?

```
      do 10 i = 1, n
10       Y(i) = a * X(i) + Y(i)
```

- What if → you don't know n until runtime?
  → n > Max. Vector Length (MVL)?

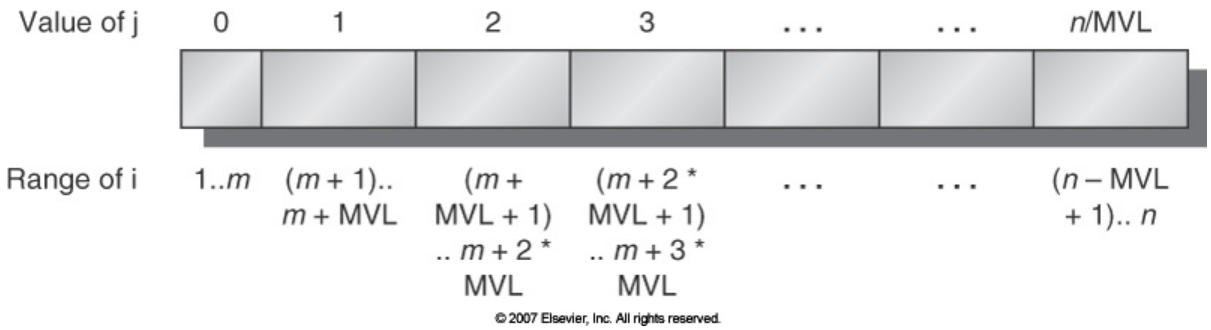
- *vector-length register* (VLR) controls the length of any vector operation, including a vector load or store.
  - must be **<** the length of vector registers

Vector Length -- Strip Mining

What to do if Vector Length > Max. Vector Length (MVL)?

- *Strip mining*: generation of code such that each vector operation is done for a size $\leq$ MVL
- 1st loop do short piece (n mod MVL), for rest VL = MVL

```
   low = 1
   VL = (n mod MVL)          /*find the odd size piece*/
   do 1 j = 0,(n/MVL)                    /*outer loop*/
       do 10 i = low,low+VL-1   /*runs for length VL*/
           Y(i) = a*X(i) + Y(i)     /*main operation*/
10     continue
       low = low+VL             /*start of next vector*/
       VL = MV                  /*reset the length to max*/
1  continue
```



Vector Stride

What to do if adjacent elements not sequential in memory?

```
   do 10 i = 1,100
       do 10 j = 1,100
              A(i,j) = 0.0
              do 10 k = 1,100
10            A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

- Either B or C accesses not adjacent (800 bytes between)

- *stride*: distance separating elements that are to be merged into a single vector (caches do unit stride)
  => LVWS (load vector with stride) instruction

- Strides ➔ can cause bank conflicts (e.g., stride = 32 and 16 banks)
- Think of address per vector element

Vector Stride, cont.

```
   do 10 i = 1,100
       do 10 j = 1,100
              A(i,j) = 0.0
              do 10 k = 1,100
10            A(i,j) = A(i,j)+B(i,k)*C(k,j)
```

```
  ; inner loop
      LV      V1, R_B        ; load vector B
      LVWS    V2, R_C, Rs    ; load vector C w/ stride
      MULTV   V3, V2, V1     ; vector-vector mult.
      ADDV    V4, V0, V3     ; vector-vector add
```

Vector Stride Example

Example

Suppose we have

- 8 memory banks
- with a bank busy time of 6 clocks
- a total memory latency of 12 cycles.

Q: How long will it take to complete a 64-element vector load with a stride of 1?

Q: With a stride of 32?

Answer

Since the number of banks is larger than the bank busy time, for a stride of 1
→ The load will take 12 + 64 = 76 clock cycles, or 1.2 clocks per element.

The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks.

→ Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time.

→ The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clocks per element.

Conditional Execution

What to do if only some elements in a vector are to be operated upon?

```
      do 100 i = 1, 64
            if (A(i) .ne. 0) then
                  A(i) = A(i) – B(i)
            endif
100   continue
```

- *vector-mask control* takes a Boolean vector
- when *vector-mask register* is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.

- Still requires clock even if result not stored; if still performs operation, what about divide by 0?

Conditional Execution, cont.

```
      do 100 i = 1, 64
            if (A(i) .ne. 0) then
                  A(i) = A(i) – B(i)
            endif
  100 continue
```

```
; inner loop
    LV        V1,Ra       ; load vector A into V1
    LV        V2,Rb       ; load vector B
    L.D       F0,#0       ; load FP zero into F0
    SNEVS.D   V1,F0       ; sets VM(i) to 1 if V1(i)!=F0
    SUBV.D    V1,V1,V2    ; subtract under vector mask
    CVM                   ; set the vector mask to all 1s
    SV        Ra,V1       ; store the result in A
```

Sparse Matrices

What to do if only a small number of randomly distributed elements in an array are to be operated upon?

```
      do    100 i = 1,n
100           A(K(i)) = A(K(i)) + C(M(i))
```

- *gather* (LVI) operation takes an *index vector* and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector ➔ a nonsparse vector in a vector register

- After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a *scatter* store (SVI), using the same index vector

- Can't be done by compiler since can't know by compiler directive whether the Ki elements are distinct, with no dependencies
- Use CVI to create index 0, 1xm, 2xm, ..., 63xm

Sparse Matrices, cont.

```
      do    100 i = 1,n
100           A(K(i)) = A(K(i)) + C(M(i))
```

```
; inner loop
    LV        Vk,Rk             ; load K
    LVI       Va,(Ra+Vk)        ; load A(K(I))
    LV        Vm,Rm             ; load M
    LVI       Vc,(Rc+Vm)        ; load C(M(I))
    ADDV.D    Va,Va,Vc          ; add them
    SVI       (Ra+Vk),Va        ; store A(K(I))
```

Challenges: Vector Example with dependency

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++)
{
    for (j=1; j<n; j++)
    {
        sum = 0;
        for (t=1; t<k; t++)
        {
            sum += a[i][t] * b[t][j];
        }
        c[i][j] = sum;
    }
}
```

Problem: creating sum of elements in a vector = slow and requires use of scalar unit

Optimized Vector Example

Consider vector processor as a collection of 32 virtual processors! Does not need reduce!

```
/* Multiply a[m][k] * b[k][n] to get c[m][n] */
for (i=1; i<m; i++) {
  for (j=1; j<n; j+=32) {                    /* Step j 32 at a time. */
    sum[0:31] = 0;          /* Initialize a vector register to zeros. */
    for (t=1; t<k; t++) {
      a_scalar = a[i][t];                  /* Get scalar from a matrix. */
      b_vector[0:31] = b[t][j:j+31];   /* Get vector from b matrix. */
      prod[0:31] = b_vector[0:31]*a_scalar;   /* Do a vector-scalar */
                                                      multiply. */
      sum[0:31] += prod[0:31];    /* Vector-vector add into results.*/
      }
    c[i][j:j+31] = sum[0:31];                /* Unit-stride store of */
                                                /* vector of results */
   }
}
```

Vector Surprise – 2 programmer's models

- Use vectors for inner loop parallelism (no surprise)
  - One dimension of array: A[0, 0], A[0, 1], A[0, 2], ...
  - think of machine as, say, 32 vector regs each with 64 elements
  - 1 instruction updates 64 elements of 1 vector register

- and for outer loop parallelism!
  - 1 element from each column: A[0,0], A[1,0], A[2,0], ...
  - think of machine as 64 "virtual processors" (VPs) each with 32 scalar registers! (multithreaded processor)
  - 1 instruction updates 1 scalar register in 64 VPs

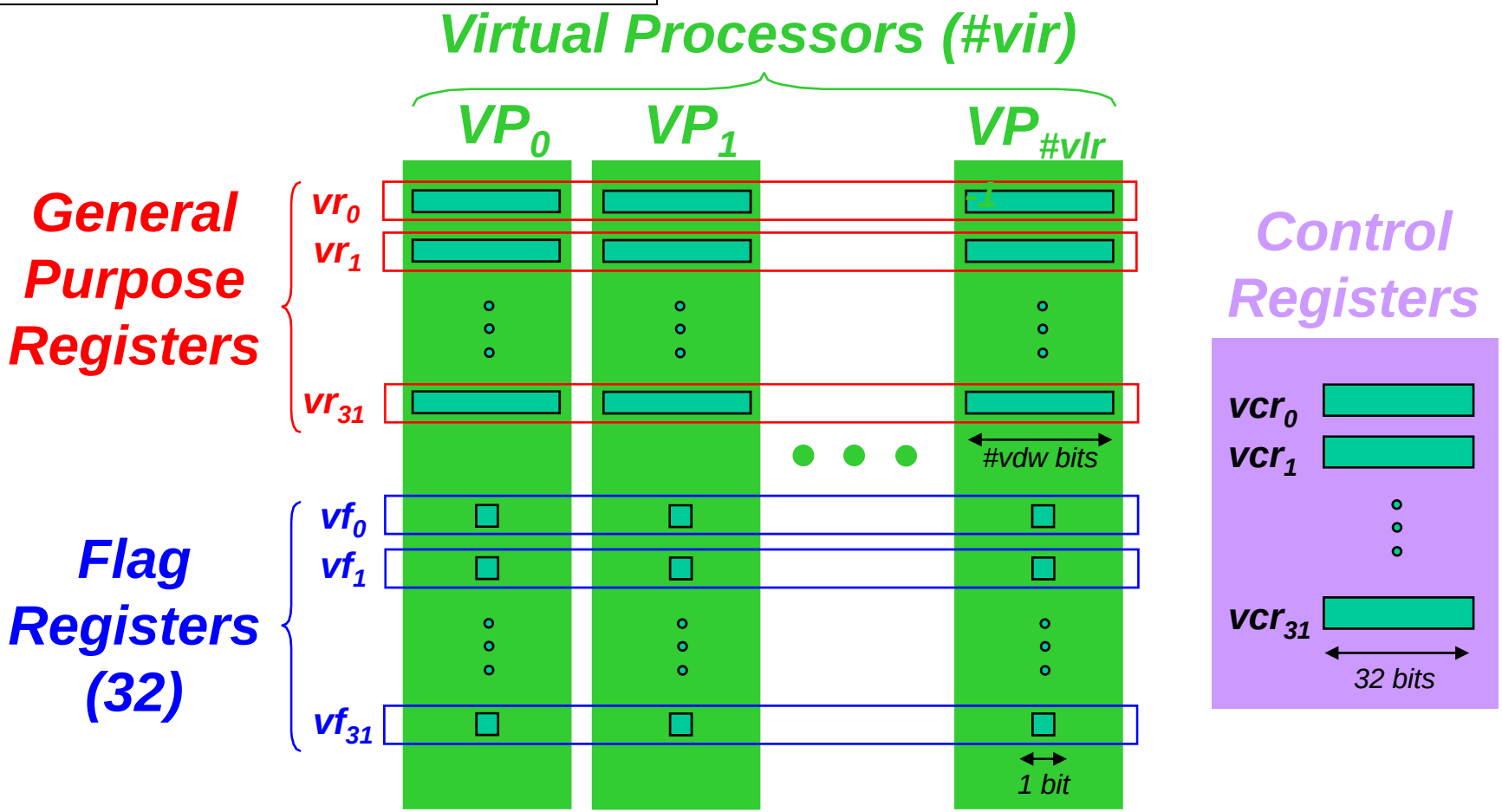
- Hardware identical, just 2 compiler perspectives

SIMD Model

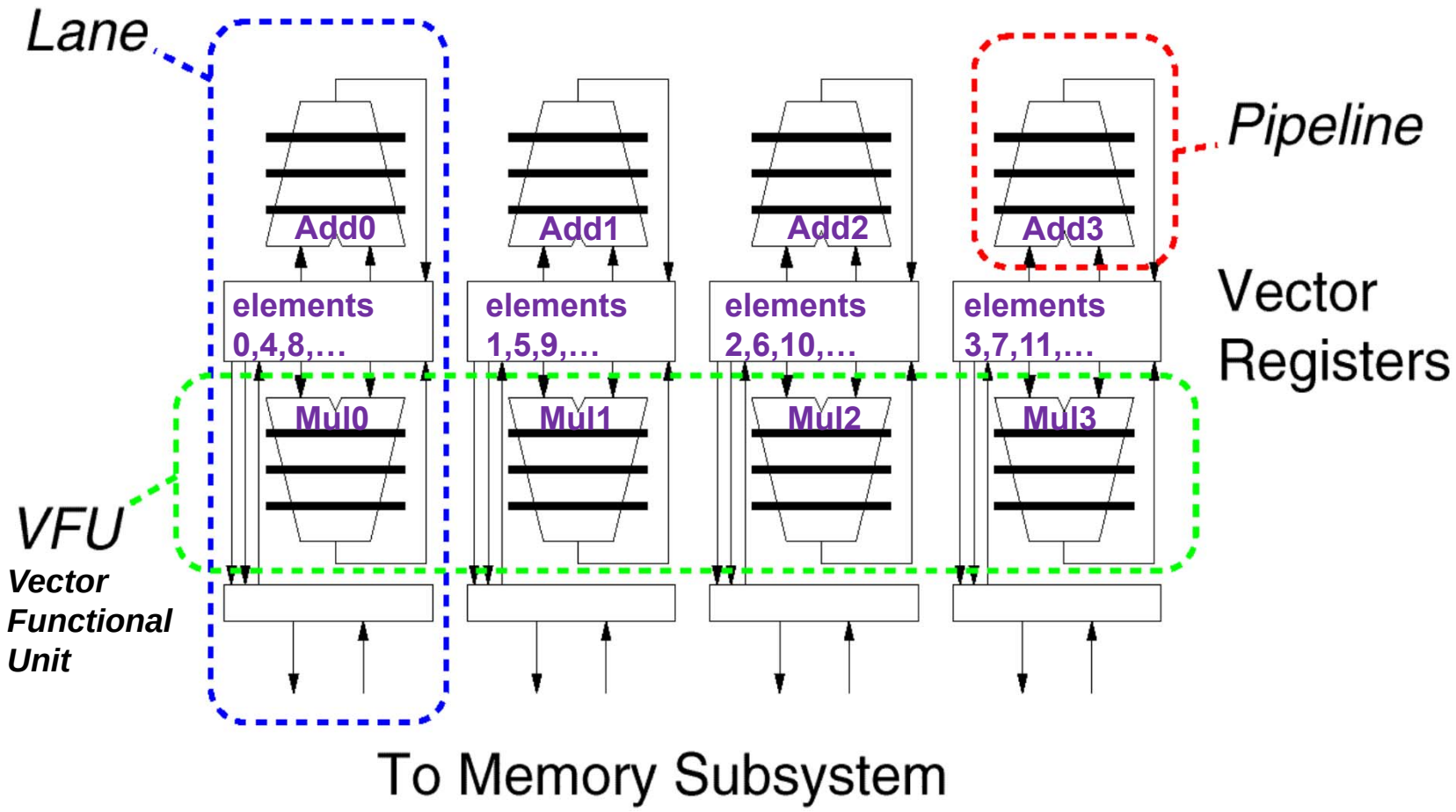
- Vector operations are SIMD (single instruction multiple data) operations
- Each element is computed by a virtual processor (VP)
- Number of VPs given by vector length
  - vector control register

Vector Implementation

- Vector register file
  - Each register is an array of elements
  - Size of each register determines maximum vector length
  - Vector length register determines vector length for a particular operation
- Multiple parallel execution units = "lanes" (sometimes called "pipelines" or "pipes")



Add Vector Units: Example has 4 lanes, 2 VFUs



The vector-register storage is divided across lanes, with each lane holding every fourth element of each vector register. There are three VFUs shown, FPADD and FPMUL. Each of the vector arithmetic units contains four execution pipelines, one per land, 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, but the scalar value must be broadcast to all lanes.

Vector Execution Time

- **Time = f(vector length, data dependencies, struct.hazards)**
- *Initiation rate*: rate that FU consumes vector elements (= number of lanes; usually 1 or 2 on Cray T-90)
- *Convoy*: set of vector instructions that can begin execution in same clock (no structural or data hazards)
- *Chime*: approx. time for a vector operation
- *m* convoys take *m* chimes; if each vector length is n, then they take approx. *m* x *n* clock cycles (ignores overhead; good approximation for long vectors)

```
1: LV     V1,Rx     ;load vector X
2: MULSV  V2,F0,V1  ;vector-scalar mult.
   LV     V3,Ry     ;load vector Y
3: ADDV   V4,V2,V3  ;add
4: SV     Ry,V4     ;store the result
```

4 convoys, 1 lane, VL=64
➔ 4 x 64 = 256 clocks
(or 4 clocks per result)

DLXV Start-up Time

***Start-up time*:** pipeline latency time (depth of FU pipeline); + other sources of overhead

Operation	Start-up penalty (from CRAY-1)
Vector load/store	12
Vector multiply	7
Vector add	6

Assume convoys don't overlap; vector length = n

Convoy	*Start*	*1st result*	*last result*
1. LV	***0***	***12***	***11+n (12+n-1)***
2. MULV, LV	***12+n***	***12+n+7***	***18+2n Multiply startup***
	12+n+1	***12+n+13***	***24+2n Load start-up***
3. ADDV	***25+2n***	***25+2n+6***	***30+3n Wait convoy 2***
4. SV	***31+3n***	***31+3n+12***	***42+4n Wait convoy 3***

Why startup time for each vector instruction?

- *Why not overlap startup time of back-to-back vector instructions?*
- *Cray machines built from many ECL chips operating at high clock rates; hard to do?*
- *Berkeley vector design ("T0") didn't know it wasn't supposed to do overlap, so no startup times for functional units (except load)*

Chaining

- Suppose:

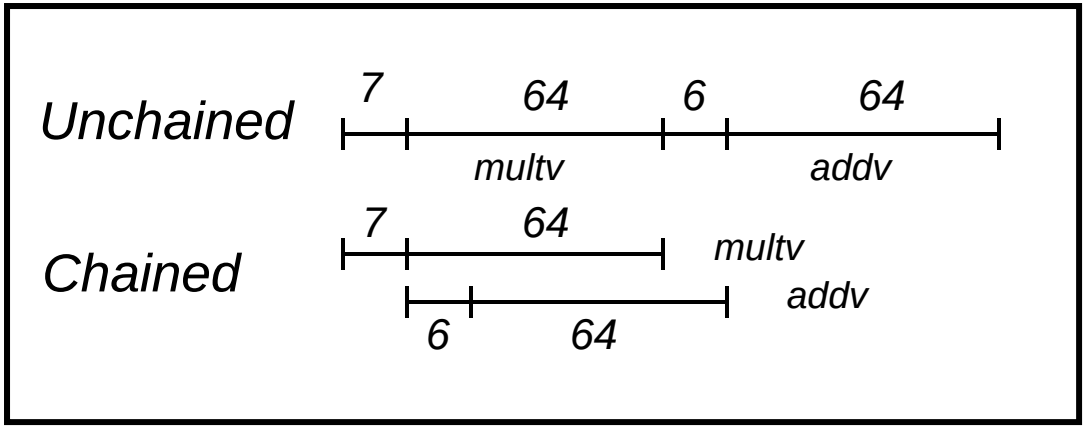
  MULV V1,V2,V3

  ADDV V4,*V1*,V5 ; separate convoy?

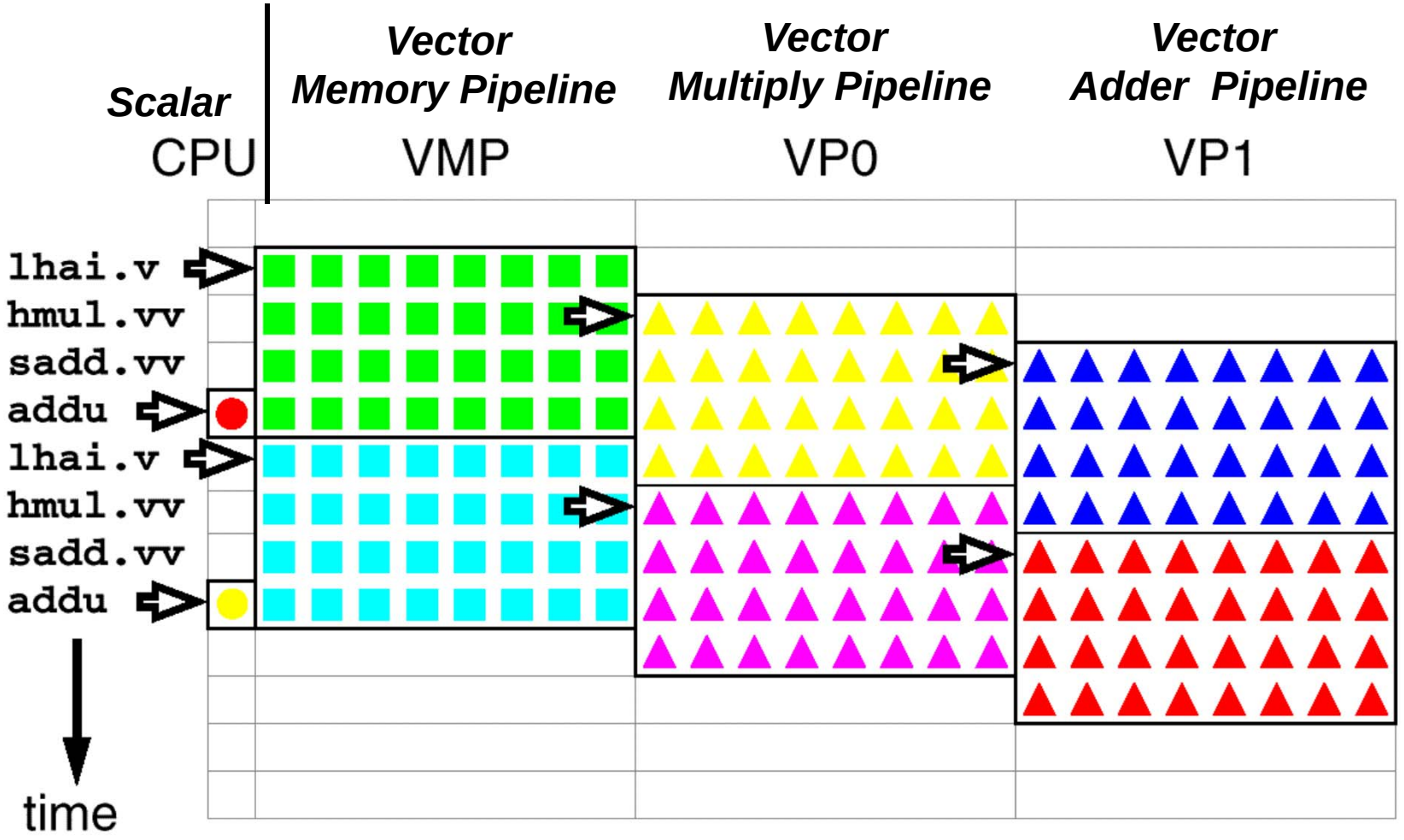
- *chaining*: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector

- *Flexible chaining*: allow vector to chain to any other active vector operation => more read/write ports

- As long as enough HW, increases convoy size



Example Execution of Vector Code



8 lanes, vector length 32, chaining

Vector Pitfalls

- Pitfall: Concentrating on peak performance and ignoring start-up overhead:
  - e.g. N_V (length faster than scalar) > 100 (CDC-star)
- Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)
  - failure of Cray competitor from his former company
- Pitfall: Good processor vector performance without providing good memory bandwidth
  - MMX?

Vector Advantages

- Easy to get high performance; N operations:
  - are independent
  - use same functional unit
  - access disjoint registers
  - access registers in same order as previous instructions
  - access contiguous memory words or known pattern
  - can exploit large memory bandwidth
  - hide memory latency (and any other latency)
- Scalable (get higher performance as more HW resources available)
- Compact: Describe N operations with 1 short instruction (v. VLIW)
- Predictable (real-time) performance vs. statistical performance (cache)
- Multimedia ready: choose N * 64b, 2N * 32b, 4N * 16b, 8N * 8b
- Mature, developed compiler technology
- Vector Disadvantage: Out of Fashion

Vector Summary

- Alternate model accommodates long memory latency, doesn't rely on caches as does Out-Of-Order, superscalar/VLIW designs
- Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer hazards, fewer branches, fewer mispredicted branches, ...
- What % of computation is vectorizable?
- Is vector a good match to apps such as multimedia, DSP?