

Topic 6

Single Instruction Multiple Data (SIMD) I

Xinfei Guo
xinfei.guo@sjtu.edu.cn

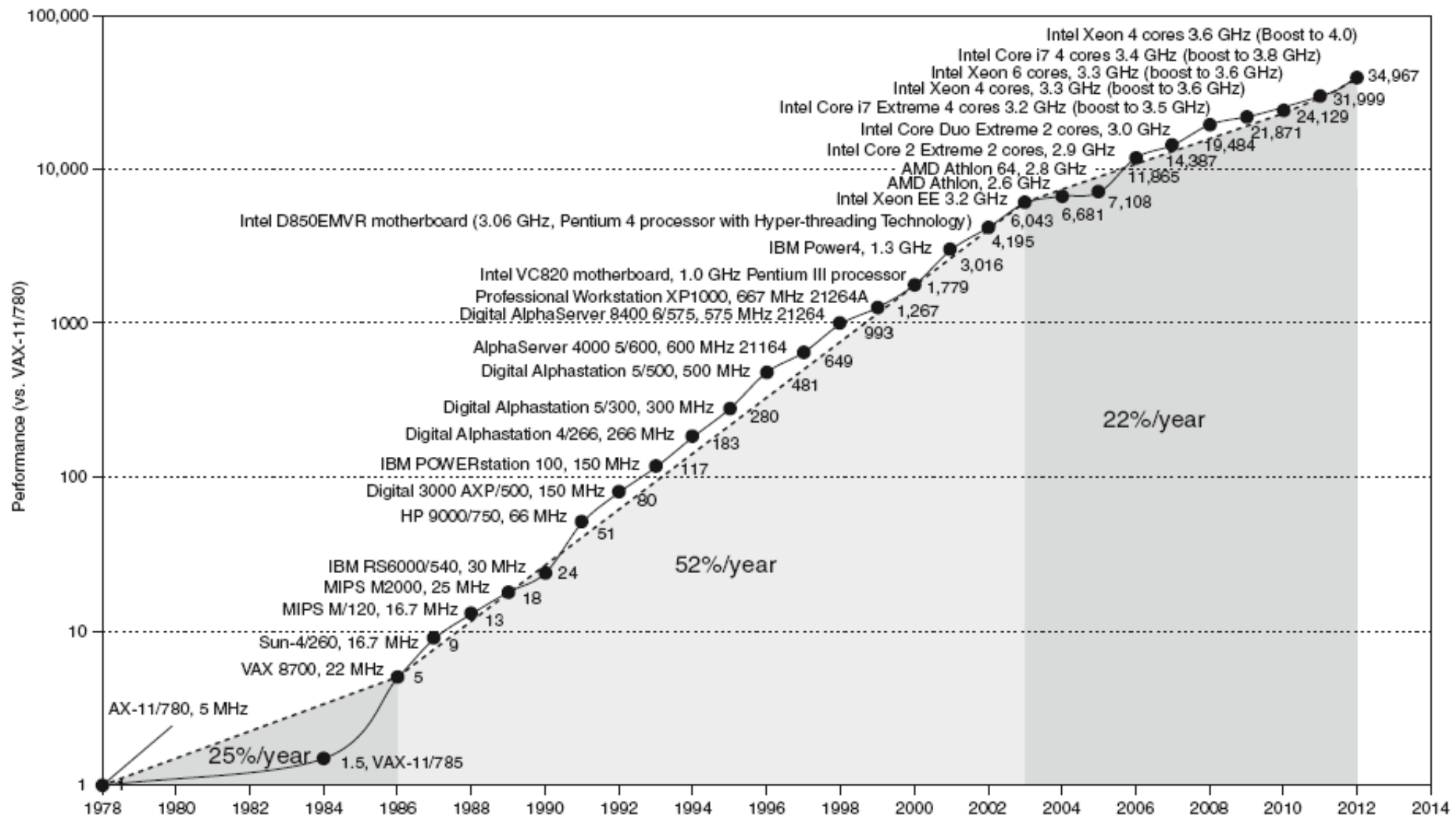
June 27th, 2022



T6 learning goals

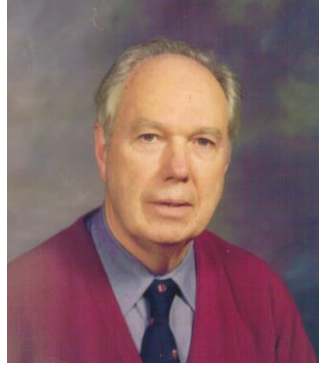
- SIMD
 - Introduction to SIMD
 - Case Study: GPU

Why parallel computing?



Flynn's Classification of Computers

- Michael J Flynn (Stanford) classified computers on the basis of multiplicity of instruction stream and data streams in a computer system.
- Instruction stream: is the sequence of instructions as executed by the machine.
- Data stream: is a sequence of data including input, or partial or temporary result, called by the instruction stream.
- Instructions are decoded by the control unit and then ctrl unit send the instructions to the processing units for execution.
- Data Stream flows between the processors and memory bi directionally

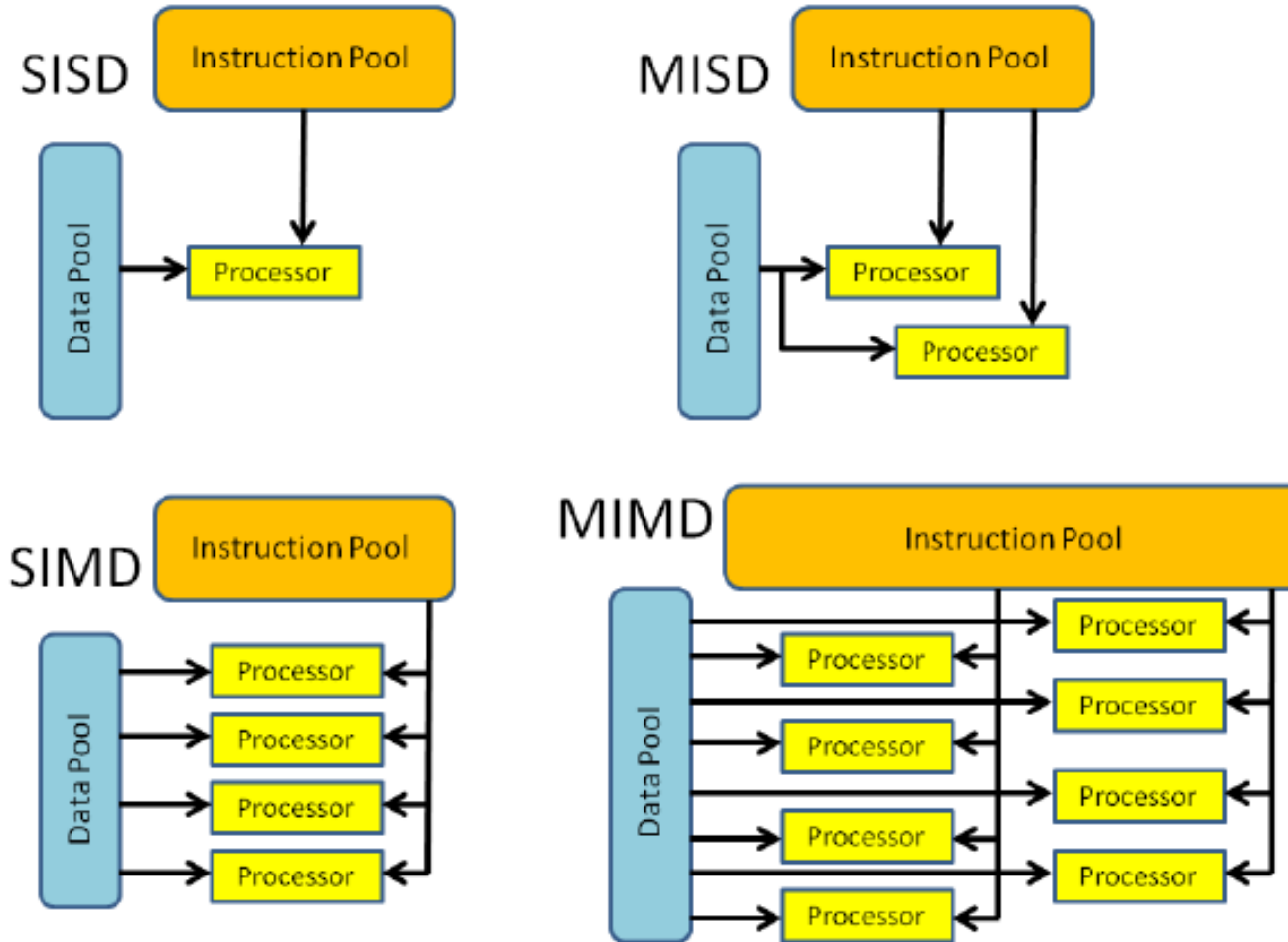


source: Wiki

Parallelism is the key!

- Instruction level parallelism (ILP)
 - Execute independent instructions from one instruction stream in parallel (pipelining, superscalar, VLIW)
- Data-Level Parallelism (DLP)
 - Execute multiple operations of the same type in parallel (vector/SIMD execution)
- Thread-Level Parallelism (TLP)
 - Execute independent instruction streams in parallel (multithreading, multiple cores)

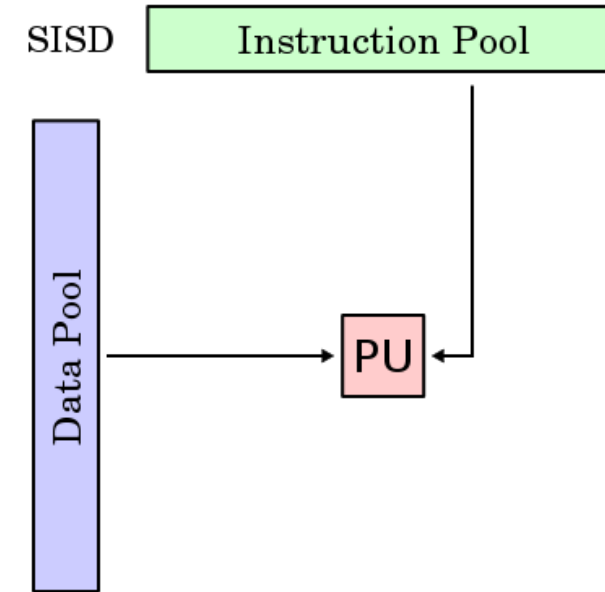
Flynn's Classification



source: <https://quizlet.com/229852945/flynns-taxonomy-diagram/>

SISD (What we learned up to now)

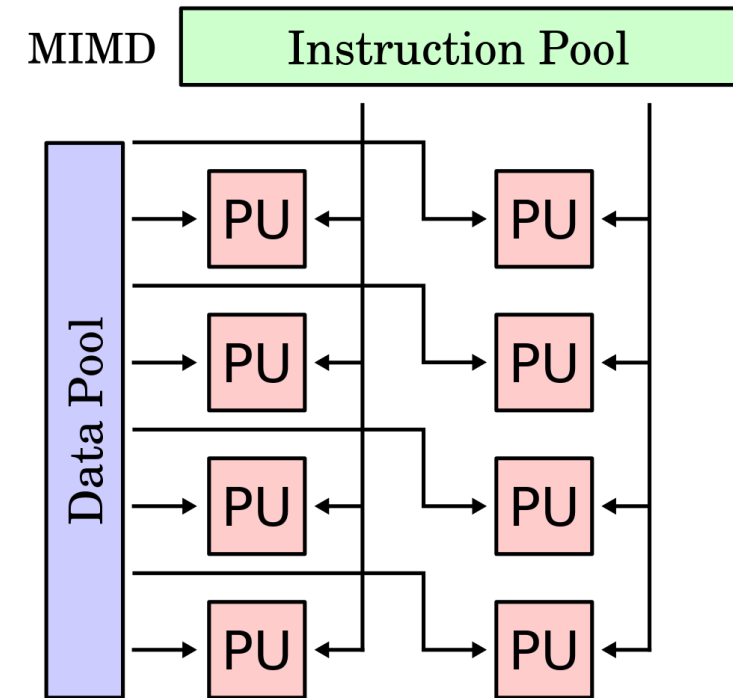
- Single Instruction Single Data
- takes data from a single address in memory and performs a single instruction on the data.
- single processor systems (microcontrollers)
- Low power requirements as only a single core



Source: Wikipedia

MIMD

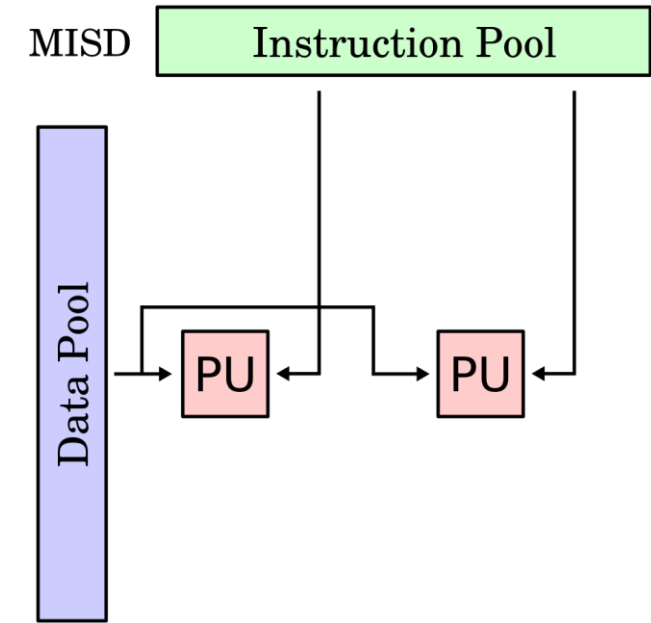
- Multiple Instruction Multiple Data
- Multiple autonomous processors perform operations on different pieces of data, either independently or as part of shared memory space.
- Most modern desktop / laptop / mobile processors are MIMD processors.
- Great to perform a variety of processor and data intensive tasks (such as video editing, game rendering)



Source: Wikipedia

MISD

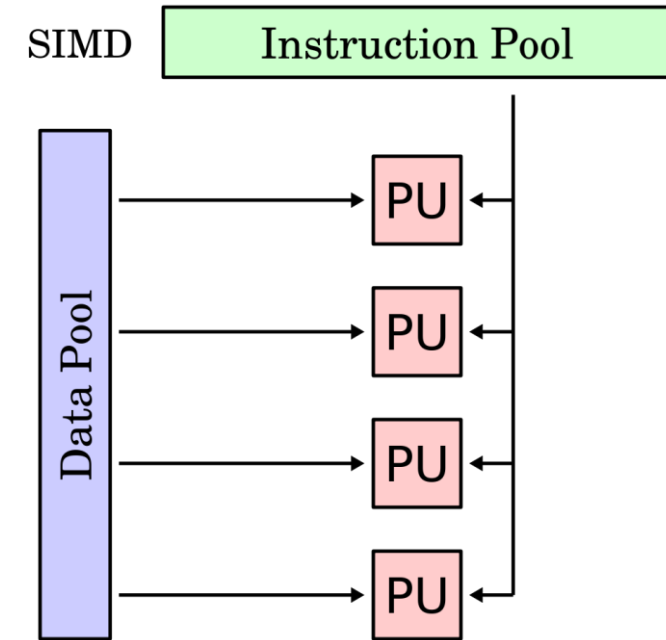
- Multiple Instruction Single Data
- Different systems perform operations on the data and all the results must agree.
- Used on flight control systems where fault detection is critical.
- Not used commercially.



Source: Wikipedia

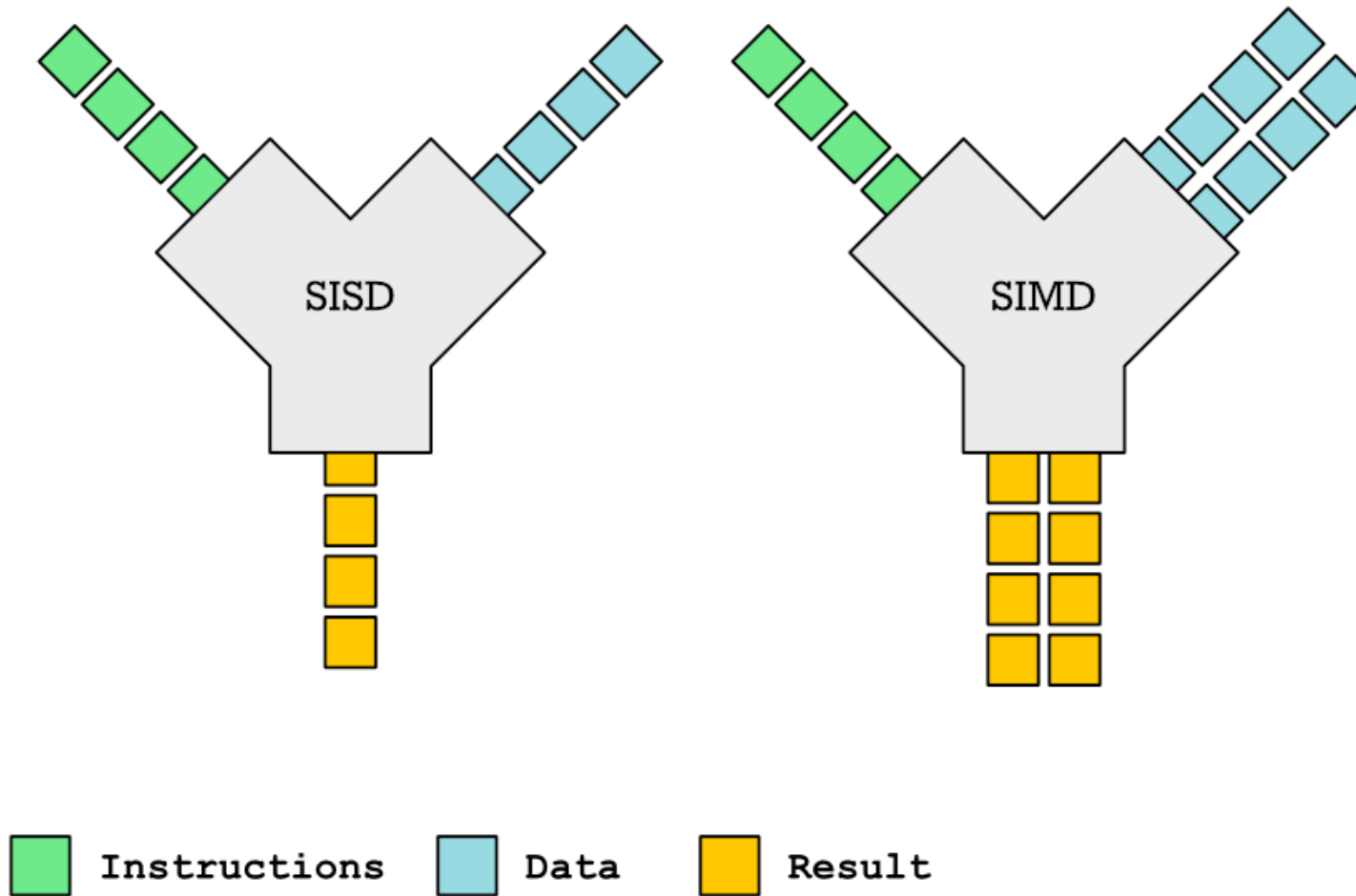
SIMD

- Single Instruction Multiple Data
- These instructions can be performed sequentially, taking advantage of pipelining, or in parallel using multiple processors.
- Example: GPUs
- Very efficient where you need to perform instruction on large amounts of data



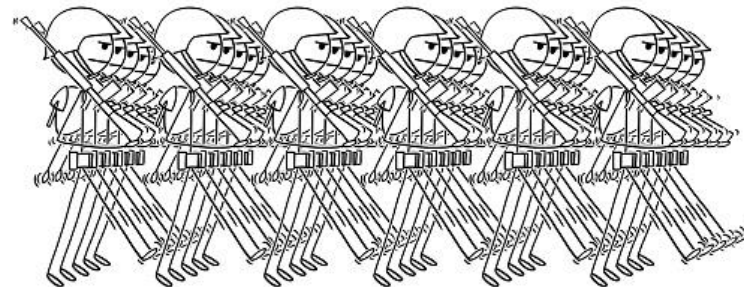
Source: Wikipedia

SISD vs. SIMD



How to View a SIMD Machine

- Think of soldiers all in a unit.
- The commander selects certain soldiers as active – for example, the first row.
- The commander barks out an order to all the active soldiers, who execute the order synchronously.
 - The remaining soldiers do not execute orders until they are re-activated.



Why SIMD?

- SIMD architectures can exploit significant data-level parallelism for:
 - Matrix-oriented scientific computing
 - Media-oriented image and sound processors
- SIMD is more energy efficient than MIMD
 - Only needs to fetch one instruction per data operation
 - Makes SIMD attractive for personal mobile devices
- SIMD allows programmer to continue to think sequentially

Multimedia / Scientific Applications

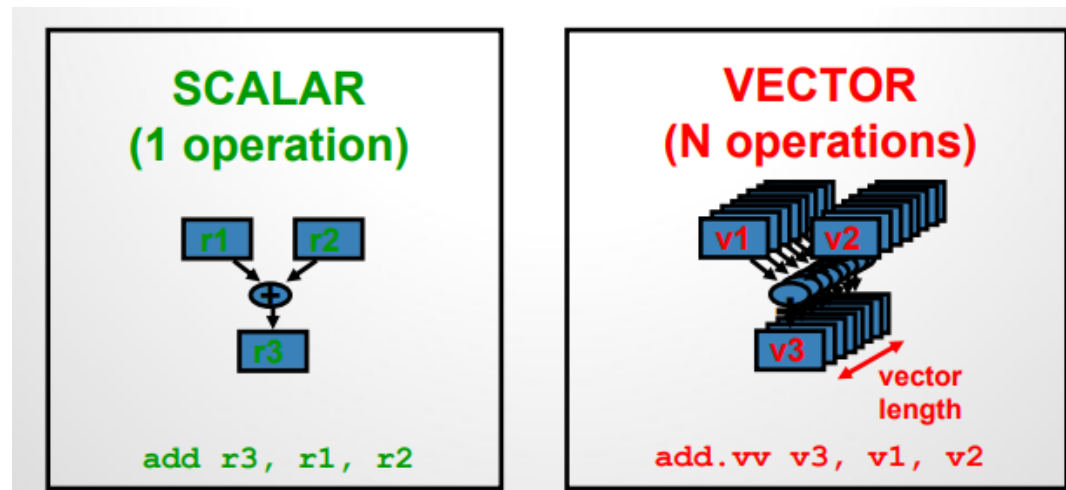
- Image
 - Graphics : 3D games, movies
 - Image recognition
 - Video encoding/decoding : JPEG, MPEG4
- Sound
 - Encoding/decoding: IP phone, MP3
 - Speech recognition
 - Digital signal processing: Cell phones
- Scientific applications
 - Double precision Matrix-Matrix multiplication (DGEMM)
 - $Y[] = a * X[] + Y[]$ (SAXPY)

SIMD Parallelism

- Vector architectures
- SIMD extensions
- Graphics Processor Units (GPUs)
- For x86 processors:
 - Expect two additional cores per chip per year
 - SIMD width to double every four years
 - Potential speedup from SIMD to be twice that from MIMD!

Description of Vector Processors

- CPU that implements an instruction set that operates on 1-D arrays, called **vectors**
- Vectors contain multiple data elements
- Many scientific/commercial programs use vectors
- Number of data elements per vector is typically referred to as the vector length
- Both instructions and data are pipelined to reduce decoding time



Advantages of Vector Processors

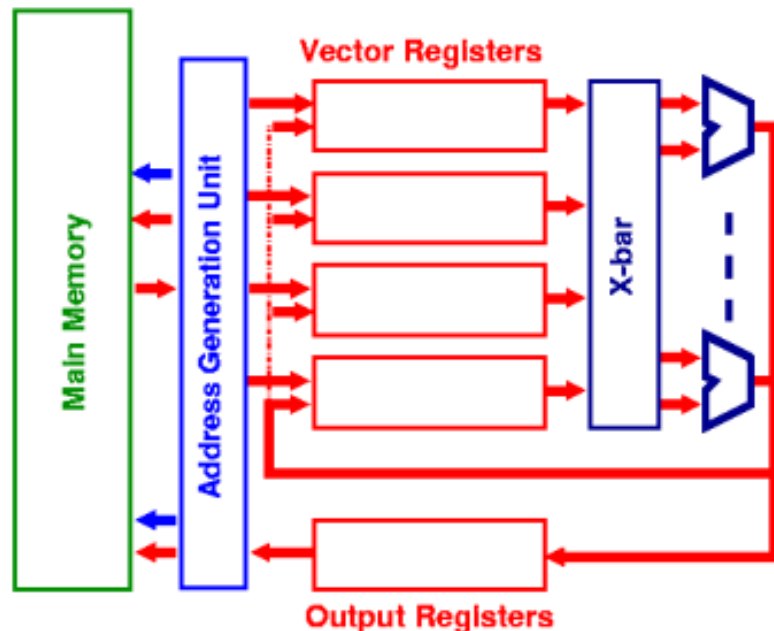
- Require Lower Instruction Bandwidth
 - Reduced by fewer fetches and decodes
- Easier Addressing of Main Memory
 - Load/Store units access memory with known patterns
- Elimination of Memory Wastage
 - Unlike cache access, every data element that is requested by the processor is actually used – no cache misses
 - Latency only occurs once per vector during pipelined loading
- Simplification of Control Hazards
 - Loop-related control hazards from the loop are eliminated
- Scalable Platform
 - Increase performance by using more hardware resources
- Reduced Code Size
 - Short, single instruction can describe N operations

Vector Architectures

- Basic idea:
 - Read sets of data elements into “vector registers”
 - Operate on those registers
 - Disperse the results back into memory
- Registers are controlled by compiler
 - Used to hide memory latency
 - Leverage memory bandwidth

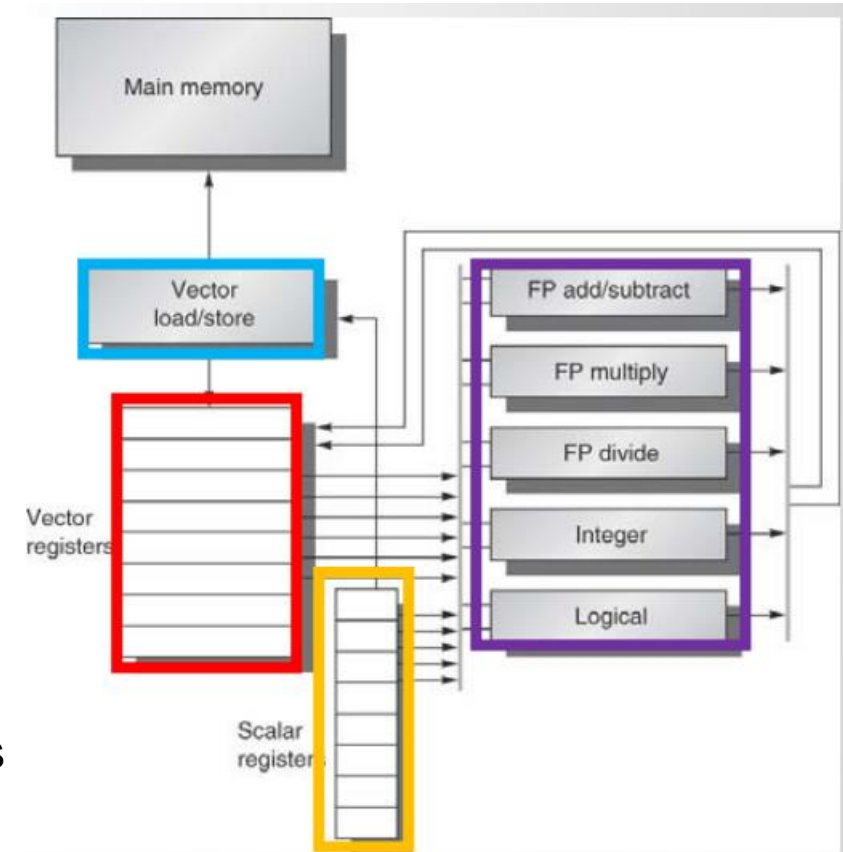
Vector Architectures

- All vector operations occur between vector registers
- If necessary, operands are fetched from main memory into a set of vector registers (load-store unit)
- Includes all vector machines since the late 1980s:
 - Convex, Cray, Fujitsu, Hitachi, NEC



Components of Vector Processors

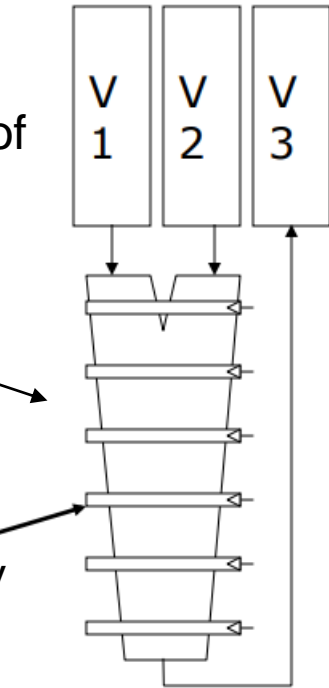
- Vector Registers
 - Typically 8-32 vector registers with 64 - 128 64-bit elements
 - Each contains a vector of double-precision numbers
 - Register size determines the maximum vector length
 - Each includes at least 2 read and 1 write ports
- Vector Functional Units (FUs)
 - Fully pipelined, new operation every cycle
 - Performs arithmetic and logic operations
 - Typically 4-8 different units
- Vector Load-Store Units (LSUs)
 - Moves vectors between memory and registers
- Scalar Registers
 - Single elements for interconnecting FUs, LSUs, and registers



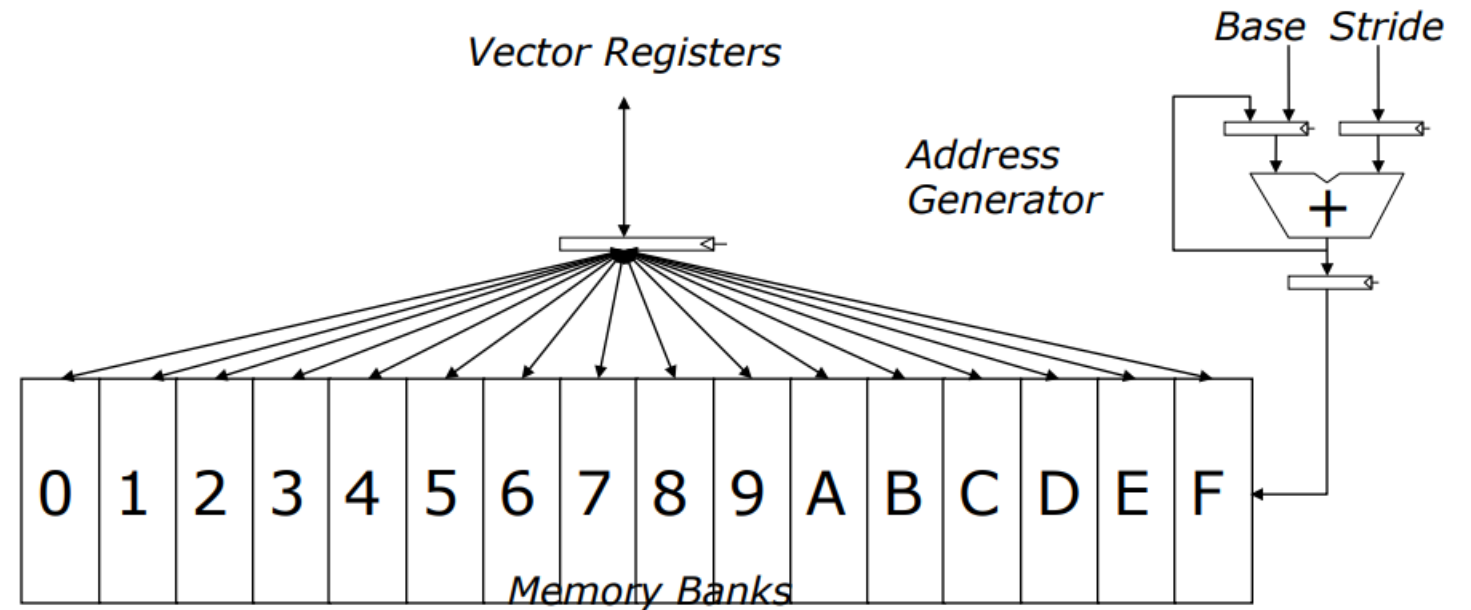
Vector FUs and Memory Systems

Simplifies control of deep pipeline because elements in vector are independent

Six stage multiply pipeline



$$V3 \leftarrow v1 * v2$$



Can sustain parallel accesses if they go to different banks

VMIPS (2001)

- Example architecture: RV64V
 - Loosely based on Cray-1
 - 32 62-bit vector registers
 - Register file has 16 read ports and 8 write ports
 - Vector functional units
 - Fully pipelined
 - Data and control hazards are detected
 - Vector load-store unit
 - Fully pipelined
 - One word per clock cycle after initial latency
 - Scalar registers
 - 31 general-purpose registers
 - 32 floating-point registers

VMIPS Instructions

- .vv: two vector operands
- .vs and .sv: vector and scalar operands
- LV/SV: vector load and vector store from address
- Example: DAXPY

```
vsetdcfg  4*FP64      # Enable 4 DP FP vregs
fld       f0,a        # Load scalar a
vld       v0,x5        # Load vector X
vmul      v1,v0,f0     # Vector-scalar mult
vld       v2,x6        # Load vector Y
vadd      v3,v1,v2     # Vector-vector add
vst       v3,x6        # Store the sum
vdisable                      # Disable vector regs
```
- 8 instructions, 258 for RV64V (scalar code)

Vector Execution Time

- Execution time depends on three factors:
 - Length of operand vectors
 - Structural hazards
 - Data dependencies
- RV64V functional units consume one element per clock cycle
 - Execution time is approximately the vector length
- *Convoy*
 - Set of vector instructions that could potentially execute together

Chimes

- Sequences with read-after-write dependency hazards placed in same convoy via *chaining*
- *Chaining*
 - Allows a vector operation to start as soon as the individual elements of its vector source operand become available
- *Chime*
 - Unit of time to execute one convoy
 - m convoys executes in m chimes for vector length n
 - For vector length of n , requires $m \times n$ clock cycles

Example

```
vld    v0,x5      # Load vector X
vmul   v1,v0,f0    # Vector-scalar multiply
vld    v2,x6      # Load vector Y
vadd   v3,v1,v2    # Vector-vector add
vst    v3,x6      # Store the sum
```

Convoys:

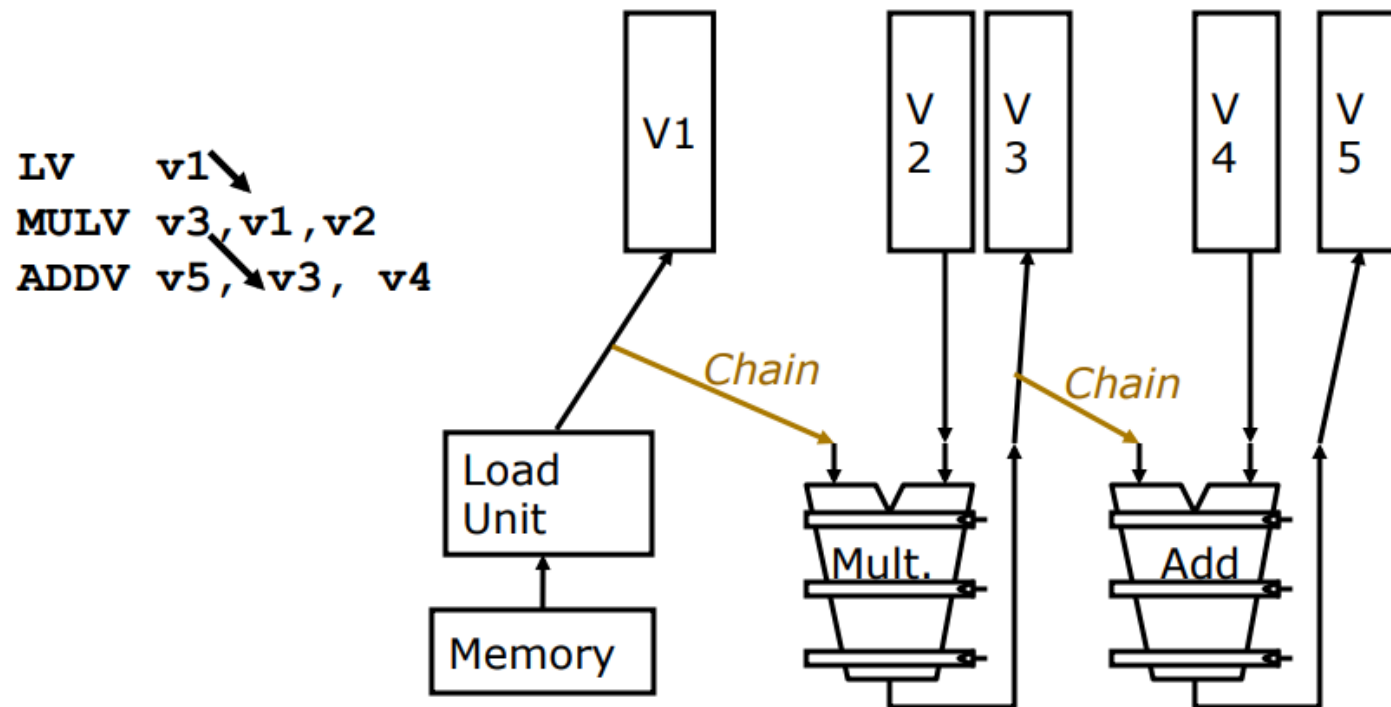
```
1      vld          vmul
2      vld          vadd
3      vst
```

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

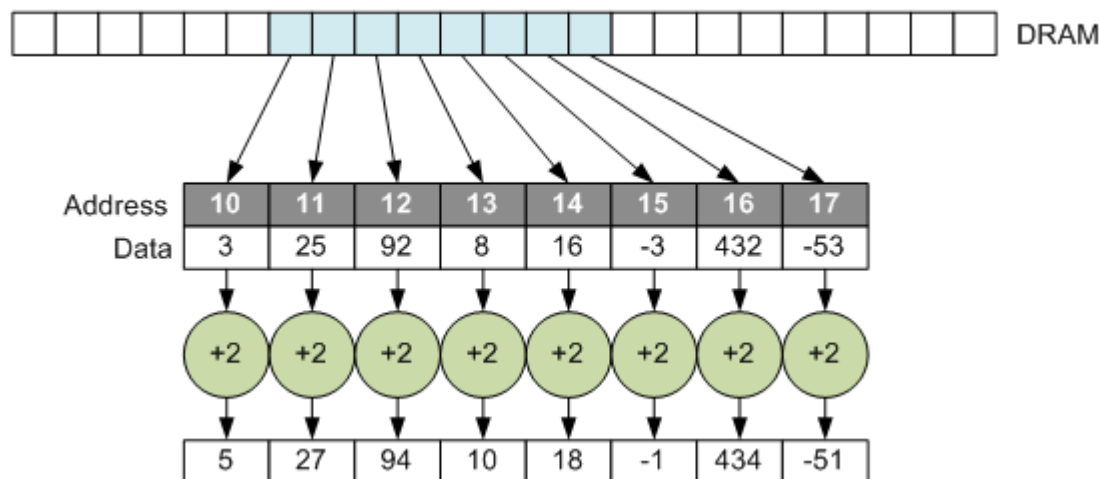
For 64 element vectors, requires $32 \times 3 = 96$ clock cycles

Performance Optimization - Vector Chaining

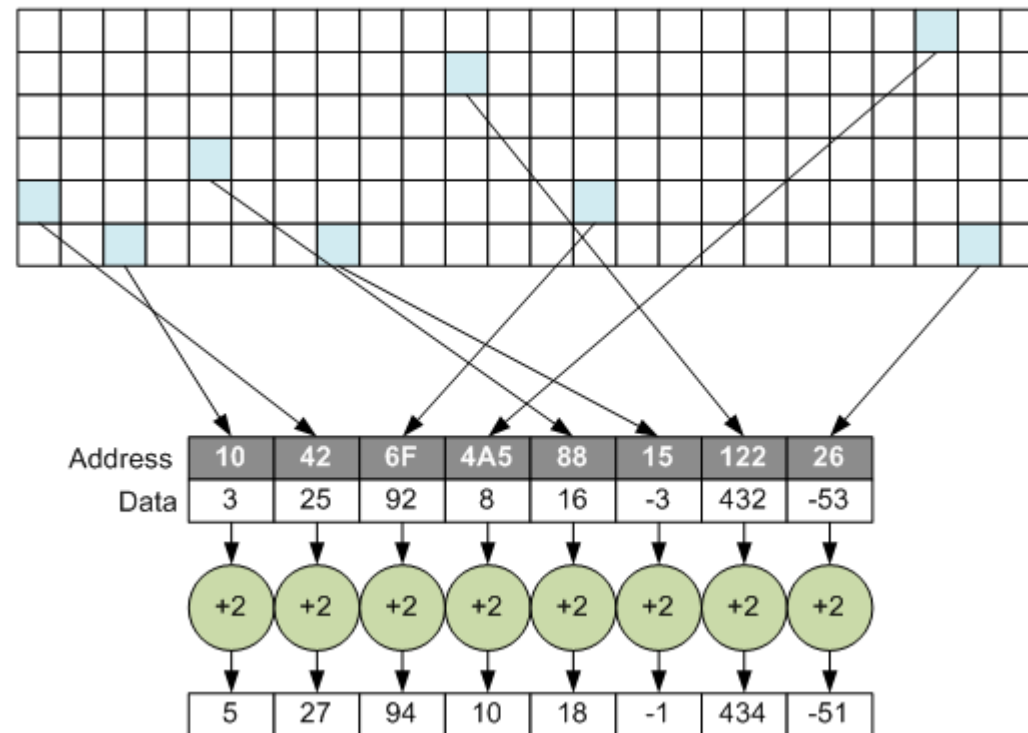
- Vector chaining: Data forwarding from one vector functional unit to another (recall data forwarding)
- Results of one pipeline are fed into operand registers of another pipeline



Performance Optimization - Scatter and Gather



SIMD operate on multiple pieces of data at the same time, in parallel



What if you want to be able to do is exactly that same thing, but without the requirement that addresses be contiguous?

The idea is to find a way to “gather” data from far-flung locations, work with them as a **single vector**, and then, if desired, take the results and “scatter” them back out into their original far-flung locations.

Performance Optimization - Scatter and Gather

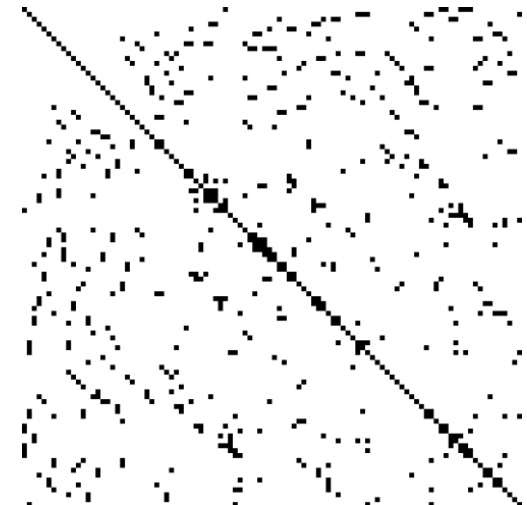
- Retrieves data elements scattered throughout memory and packs them into sequential vectors in vector registers
 - Promotes data locality and reduces data pollution
 - Use indirection (index vectors) to combine elements into vector registers
 - A gather operation takes an index vector and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector.
 - After these elements are operated on in a dense form, the sparse vector can be stored in an expanded form by a scatter store, using the same index vector.

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
```



Performance Optimization – Strip Mining

- What if # data elements > # elements in a vector register?
 - Need to break loops so that each iteration operates on # elements in a vector register
 - E.g., 527 data elements, 64-element VREGs
 - 8 iterations where Vector length = 64
 - 1 iteration where length = 15 (need to change value of vector length)
 - Strip Mining (used in compilers)
 - Generates code to allow vector operands whose size is less than or greater than size of vector registers

```
i = 0;
VL = n mod MVL;
for (j=0; j<n/MVL; j++){
    for(i<VL; i++)
        {Y(i)=a*X(i)+Y(i)}
    VL = MVL;
}
```

Performance Optimization – Masked Operations

- What if some operations should not be executed on a vector?

```
loop: if (a[i] != 0) then b[i]=a[i]*b[i]
goto loop
```

This loop cannot normally be vectorized because of the conditional execution of the body; however, if the inner loop could be run for the iterations for which $a[i] \neq 0$

- Idea: Masked operations

- VMASK register is a bit mask determining which data element should not be acted upon

```
VLD V0 = A
VLD V1 = B
VMASK = (V0 != 0)
VMUL V1 = V0 * V1
VST B = V1
```

These registers use a Boolean vector to control the execution of a vector instruction, just as conditionally executed instructions use a Boolean condition to determine whether to execute a scalar instruction.

Another Example with Masking

```
for (i = 0; i < 64; ++i)
  if (a[i] >= b[i]) then c[i] = a[i]
  else c[i] = b[i]
```

A	B	VMASK
1	2	0
2	2	1
3	2	1
4	10	0
-5	-4	0
0	-3	1
6	5	1
-7	-8	1

Steps to execute loop

1. Compare A, B to get VMASK
2. Masked store of A into C
3. Complement VMASK
4. Masked store of B into C

Masked Vector Instructions

Simple Implementation

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

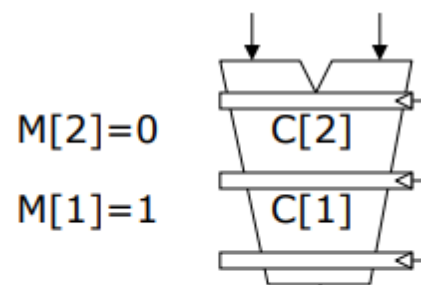
$M[7]=1$ $A[7]$ $B[7]$

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

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

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

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



$M[0]=0$

Write Enable Write data port

Density-Time Implementation

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

$M[7]=1$

$M[6]=0$

$M[5]=1$

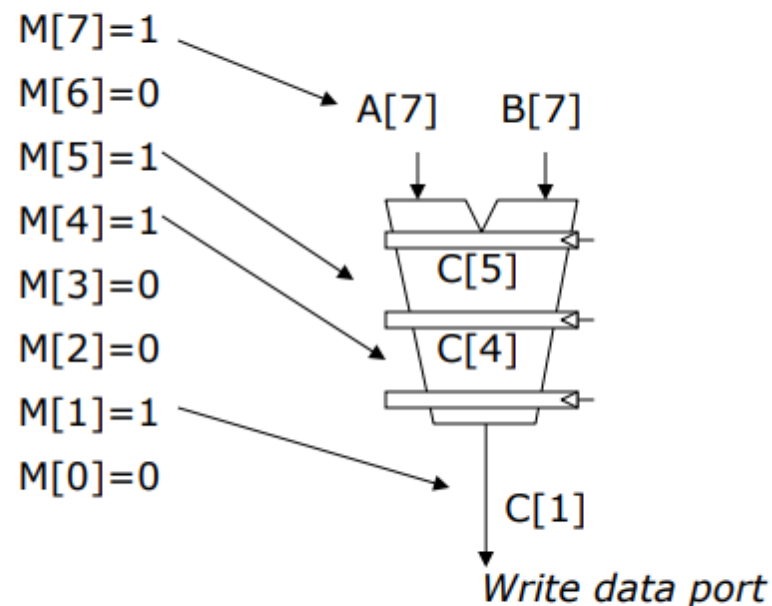
$M[4]=1$

$M[3]=0$

$M[2]=0$

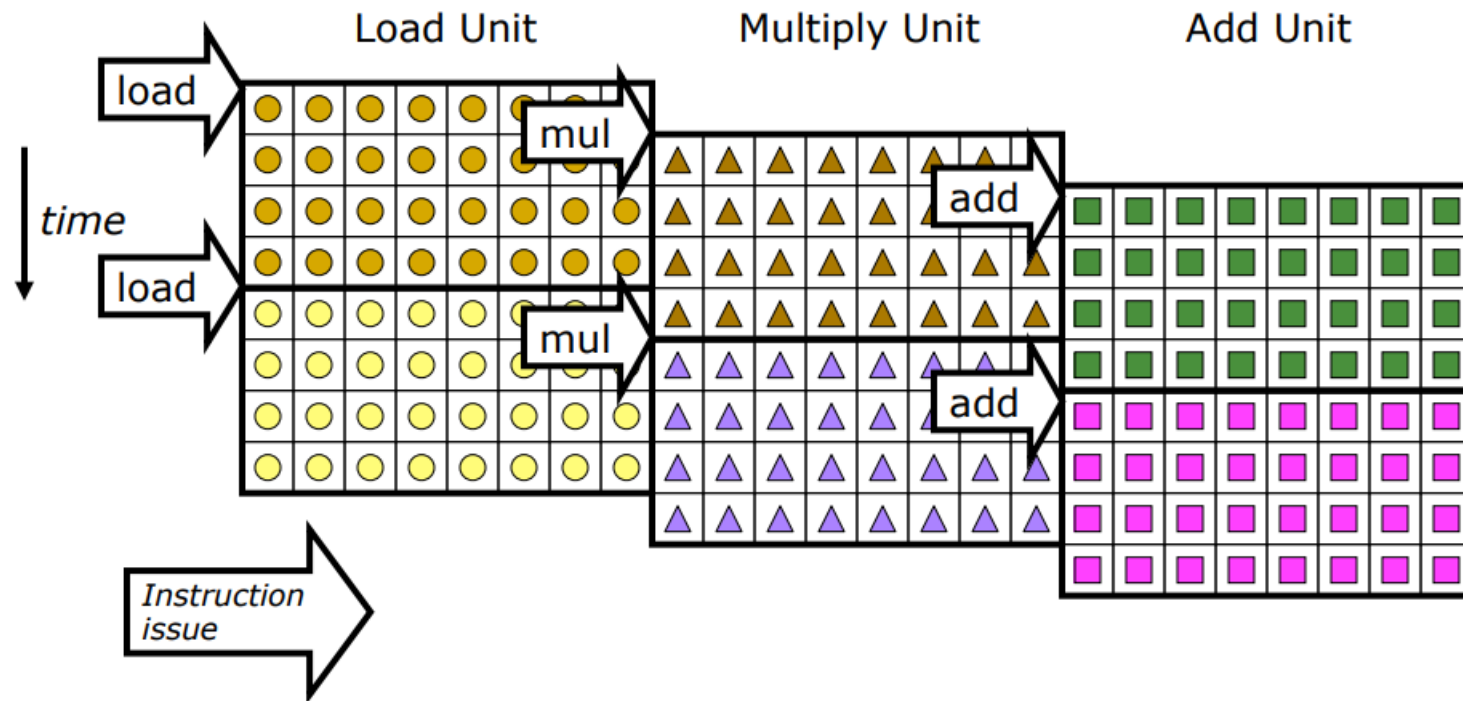
$M[1]=1$

$M[0]=0$



Vector Instruction Level Parallelism

- Can overlap execution of multiple vector instructions
 - example machine has 32 elements per vector register and 8 lanes
 - Complete 24 operations/cycle while issuing 1 short instruction/cycle



Other Performance Optimizations

- Increase Memory Bandwidth
 - Memory banks are used to reduce load/store latency
 - Allow multiple simultaneous outstanding memory requests
- Multiple Parallel Lanes, or Pipes
 - Allows vector operation to be performed in parallel on multiple elements of the vector

SIMD Challenges

- Larger registers and functional units use more chip area and power
- Difficult to parallelize some algorithms (Amdahl's Law)
- Parallelization requires explicit instructions from the programmer or compiler

Today's SIMD

- SIMD functionality is sometimes embedded in sequential machines.
- Others are being build as part of hybrid architectures.
- Some SIMD and SIMD-like features are included in some multi/many core processing units
- Some SIMD-like architectures have been build as special purpose machines, although some of these could classify as general purpose.
 - Some of this work has been proprietary.
 - The fact that a parallel computer is SIMD or SIMD-like is often not advertised by the company building them.

Where are we Heading?

- T6: GPUs

Acknowledgement

Slides in this topic are inspired in part by material developed and copyright by:

- ARM Courseware
- Prof. Eric Welch & James Evans @ RIT
- Prof. Mikko H. Lipasti @ U of Wisconsin Madison
- Prof. Joe Devietti @ Upenn, CIS 571
- Prof. Ron Dreslinski @ UMich, EECS 470
- Prof. Hakim Weatherspoon @ Cornell, CS 3410
- Prof. Krste Asanovic @ UCB, CS252
- Xinfei Guo @ JI, VE370 2021 SU

Action Items

- HW #4
- Lab #4
- Final project
- Reading Materials
 - Ch. 4.1 – 4.3