Master Informatics Eng.

2017/18

A.J.Proença

Data Parallelism 1 (*vector & SIMD extensions*)

(most slides are borrowed)

Instruction and Data Streams

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

Instruction and Data Streams

		Data Streams	
		Single	Multiple
Instruction Streams	Single	**SISD**: Intel Pentium 4	**SIMD**: SSE instructions of x86
	Multiple	**MISD**: No examples today	**MIMD**: Intel Xeon e5345

- SPMD: Single Program Multiple Data
  - A parallel program on a MIMD computer
  - Conditional code for different processors

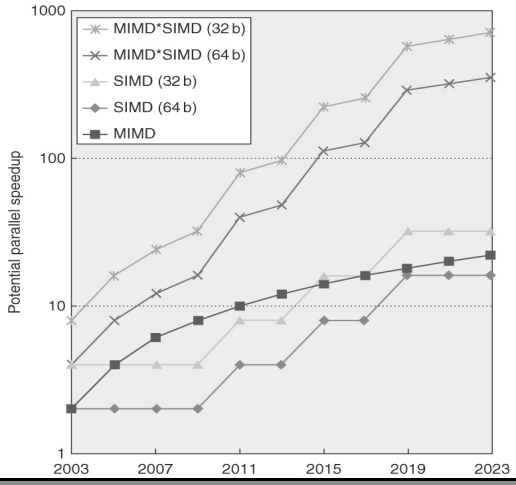
Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processing
- 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 programmers to continue to think sequentially

SIMD Parallelism

- Vector architectures *(slides 5 to 19)*
- SIMD & extensions *(slides 20 to 30)*
- Graphics Processor Units (GPUs) *(next set)*

- For x86 processors:
  - Expected grow:
    2 more cores/chip/year
  - SIMD width:
    2x every 4 years
  - Potential speedup:
    SIMD 2x that from MIMD!



Vector Architectures

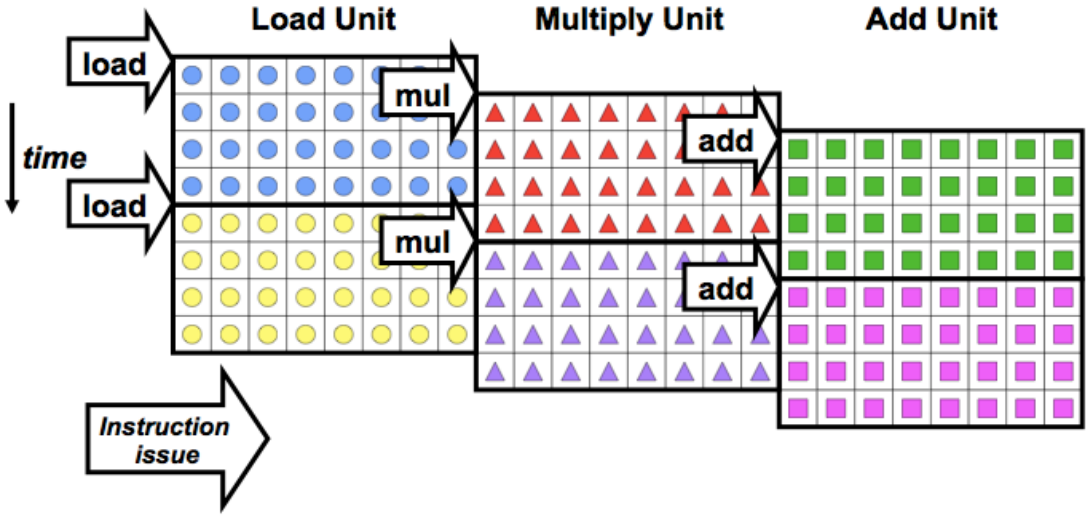
- Basic idea:
  - Read sets of data elements *(gather from memory)* into "vector registers"
  - Operate on those registers
  - Store/scatter the results back into memory

- Registers are controlled by the compiler
  - Used to hide memory latency
  - Leverage memory bandwidth

Vector Instruction Parallelism

Can overlap execution of multiple vector instructions

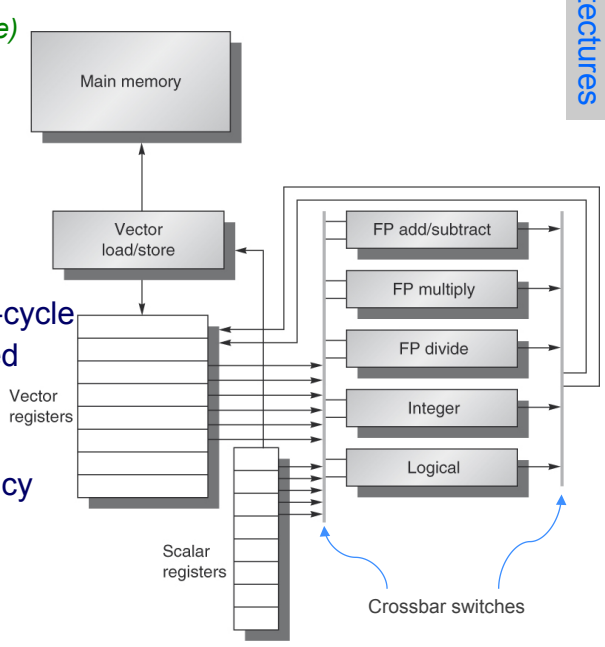
- Consider machine with 32 elements per vector register and 8 lanes:

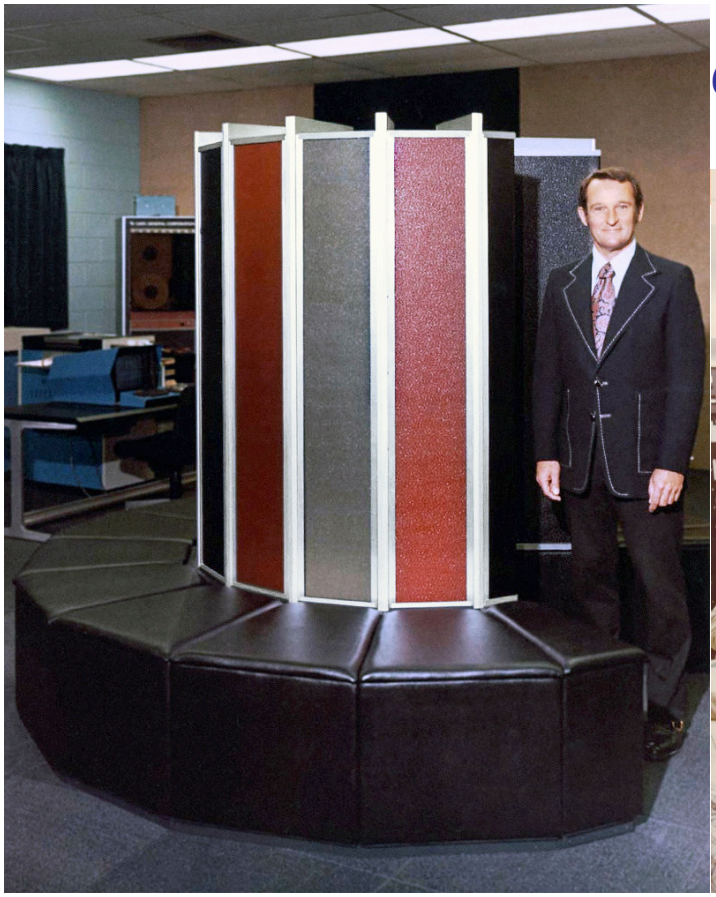


Complete 24 operations/cycle while issuing 1 short instruction/cycle

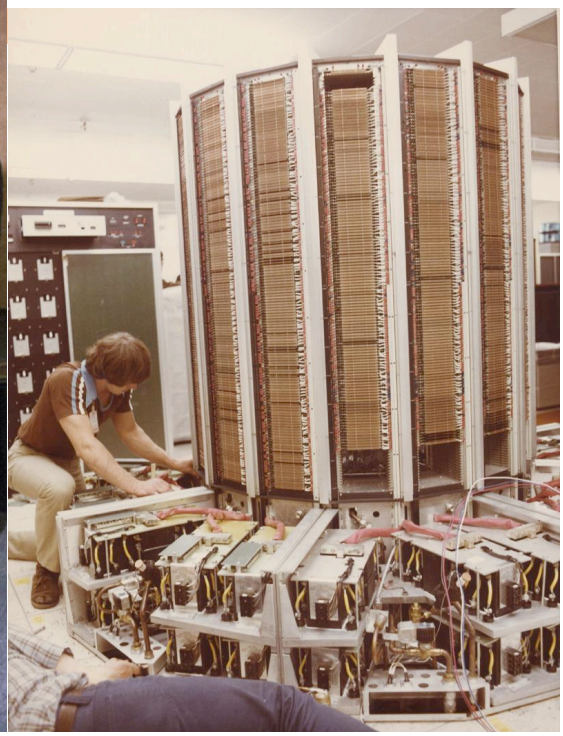
VMIPS

- Example architecture: VMIPS
  - Loosely based on Cray-1 *(next slide)*
  - Vector registers
    - Each register holds a 64-element, 64 bits/element vector
    - Register file has 16 read ports and 8 write ports
  - Vector functional units
    - Fully pipelined, new op each clock-cycle
    - Data & control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - 1 word/clock-cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers





Cray-1 Supercomputer (1976)



VMIPS Instructions

- ADDVV.D: add two vectors
- ADDVS.D: add vector to a scalar
- LV/SV: vector load and vector store from address

- Example: DAXPY *(Double-precision A x X Plus Y)*

```
L.D      F0,a        ; load scalar a
LV       V1,Rx       ; load vector X
MULVS.D  V2,V1,F0    ; vector-scalar multiply
LV       V3,Ry       ; load vector Y
ADDVV    V4,V2,V3    ; add
SV       Ry,V4       ; store the result
```

- Requires the execution of 6 instructions *versus* almost 600 for MIPS *(assuming DAXPY is operating on a vector with 64 elements)*

Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS 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 in one unit of time, *chime*

Challenges

- Start up time
  - Latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Improvements:
  - > 1 element per clock cycle *(1)*
  - Non-64 wide vectors *(2)*
  - IF statements in vector code *(3)*
  - Memory system optimizations to support vector processors *(4)*
  - Multiple dimensional matrices *(5)*
  - Sparse matrices *(6)*
  - Programming a vector computer *(7)*