PARALLEL COMPUTING PLATFORMS, PIPELINING, VECTOR COMPUTING

- **Terminology and Flynn's taxonomy**
- **The PRAM model**
- **Pipelining**
- **Vector computers, vector processing (AVX, etc)**
- **Superscalar processing; VLIW**

Flynn's Taxonomy of parallel computers

➤ Distinguishes architectures by the way processors execute their instructions on the data.

Instructions streams	Data streams: single	Data streams: multiple
single	SISD	SIMD
multiple	MISD	MIMD

Four categories :

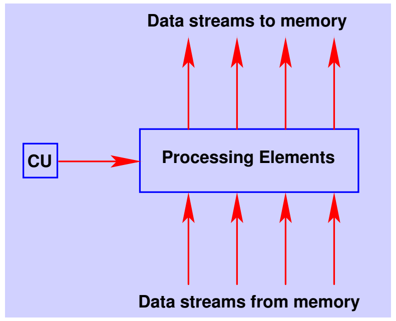
1. Single Instruction stream Single Data stream (SISD)
2. Single Instruction stream Multiple Data stream (SIMD)
3. Multiple Instruction stream Single Data stream (MISD)
4. Multiple Instruction stream Multiple Data stream (MIMD)

SISD Architecture This is the standard von Neuman organization

MISD Architecture Same data simultaneously exploited by several processors or computing units that operate on it. Pipelined processors, systolic arrays, ...,

SIMD Architecture Same instruction is executed simultaneously on different "data streams". Single Control Unit (CU) dispatches a single stream of instructions.

➤ Includes pipelined vector computers + many classical machines (e.g., ILLIAC IV in the 60s, the CM2 in early 90s)



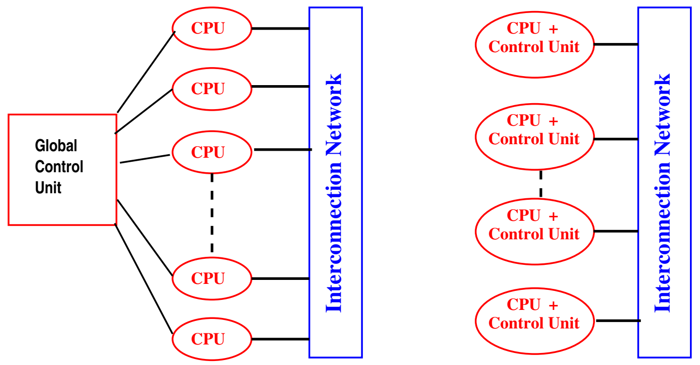
MIMD Architecture MIMD machines simultaneously execute different instructions on different data streams.

➤ Processing elements have their own control units - but coordinate computational effort with the other processors.

➤ MIMD computers further classified in two important subcategories

- *Shared memory models* : processors have very little local or 'private' memory; they exchange data and co-operate by accessing a global shared memory.
- *Distributed memory models* : No shared global address space. Each processor has its own local memory. Interconnections between processors allow exchange of data and control information.

Typical SIMD (left) and MIMD (right) organizations.



The PRAM model

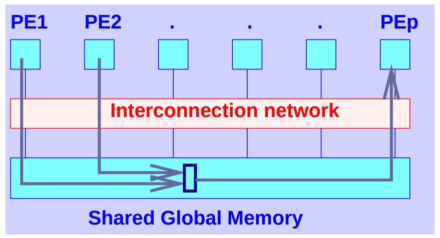
➤ PRAM = Parallel Random Access Memory. Model used mostly for theory.

➤ Extends the Von-Neumann model.

➤ P processors having access to a large memory

➤ Question: How to manage conflicts? [2 processors wanting to read & write to same location]

➤ Four different models used



EREW - Exclusive Read, Exclusive Write

CREW - Concurrent Read, Exclusive Write

ERCW - Exclusive Read, Concurrent Write

CRCW - Concurrent Read, Concurrent Write

Concurrent reads are harmless. Concurrent writes require arbitration. Four methods:

Common	Writes successful if data is same
Arbitrary	One PE only succeeds in the write
Priority	PEs prioritized. Highest priority PE writes
Combine results	e.g., add results then write

Architecture options for exploiting parallelism

1. Multiple functional units $*, +, ..$
2. Pipelining, Superscalar pipelines
3. Vector processing
4. Multiple Vector pipelines
5. Multiprocessing
6. Distributed computing

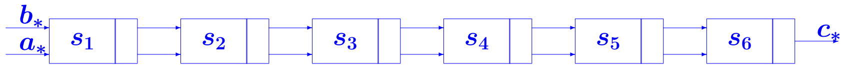
Pipelining

Idea of the assembly line. Assume an operation (e.g., FP adder) takes s stages to complete. Then we pass the operands through the s stages instead of waiting for all stages to be complete for first two operands.

Example: an arithmetic adder:

1	Unpack operands	*4*	Add fractions;
2	Compare exponents	*5*	Normalize fraction
3	Align significands	*6*	Pack operands

➤ To operate on a stream of data (a pair of vectors), no need to wait for all the stages to complete on each pair. Instead, "pipeline" the stream through the 6 stages.



➤ Assume that each stage takes one clock period. After s clocks, the pipe is filled, i.e., all stages are active. Then an operation is produced at each clock cycle.

➤ If each stage takes time τ, then, operation with n numbers will take

$$s\tau + (n-1)\tau = (n+s-1)\tau \text{ seconds}$$

Instead of

$$ns\tau \text{ seconds}$$

➤ Gain a factor of

$$\frac{ns}{(n+s-1)}$$

Vector pipelines

➤ Simplest example: $s1 = s2 + s3$

Machine operations (informal assembler notation)

```
                         cycles
  load   s2 --> R1          1   % R1-R3 = registers
  load   s3 --> R2          1
  add    R1 + R2 --> R3     6   % (6 stages)
  store  R3 --> s1          1
```

➤ 9 cycles to complete one operation.

➤ On a sequential computer (scalar mode) the loop

```
      do i=1,64
         s1(i)=s2(i)+s3(i)
      enddo
```

would take 64*9=576 cycles + loop overhead.

➤ If operations are pipelined: one operation delivered at each cycle once pipe is filled. Total time: 63 + 9 cycles. (!)

➤ Vector computers: (1) exploit ideas of pipelining at all levels (loads, store, vector adds, ...) to reduce cost and (2) provide "vector" instructions

✍1 Assume that r is the ratio of the peak vector speed to peak scalar speed (the 'speed-up') and that the fraction of vector to scalar operations in your calculation is f (if 5% of operations are purely scalar then $f = 0.95$). Show that the speed-up for your calculation will be

$$S(f) = \frac{r}{(1-f)*r+f}$$

What speed up do you get when vector processing is very fast ($r = \infty$) and $f = 0.9$

Vector computers - continued

- Hardware component: vector registers
- Vector registers are used in vector instructions.

Example: The loop `a(1:64)=b(1:64)+c(1:64)`

May be translated as:

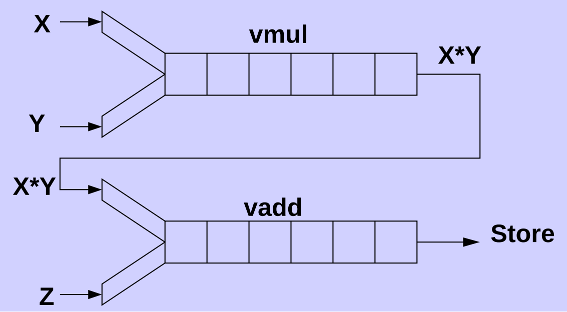
```
vload  b(1), 64, 1 --> V1   % Vector load to V1
vload  c(1), 64, 1 --> V2
vadd   V1 + V2 --> V3       % Vector addition
vstore a(1), 64, 1 <-- V3
```

- b(1) = start address, "64"=vector length, "1" = stride

Chaining

Take the result of one pipe (e.g. a multiplier) as an input to another pipe (e.g. adder).

```
t(1:64)=z(1:64)+x(1:64)*y(1:64)
```



- Main assembly instructions: vmul V1*V2 → V3
  vadd V3+V4 → V5

Vector Extensions: SSE, SSE2, ..., AVX

- SSE = Streaming SIMD Extensions
- Modern versions of vector instructions added to common chips
- History: MMX → SSE → SSE2 → AVX→ AVX2→ AVX512
- Additions to the x86 instruction set architectures for Intel and AMD processors.
- Look at the X86 instruction set
- SSE registers: 4 FP32 packed into a single 128b register
- 8 registers xmm0, xmm1, ..., xmm7
- SSE2: xmm registers can be 4 × FP32D or 2 × FP64, or 4 × INT32 , or 8 × INT16 , or 16 × 1 Char(1B)
- Similarly for 256 long registers

Example: Suppose we want to add FP32 vectors of 4 components $\boldsymbol{v1.a, v1.b, v1.c, v1.d}$ and $\boldsymbol{v2.a, v2.b, v2.c, v2.d}$ Normally we would have 3 FP32 adds to get the result

$$\begin{aligned} \boldsymbol{w.a} &= \boldsymbol{v1.a + v2.a} \\ \boldsymbol{w.b} &= \boldsymbol{v1.b + v2.b} \\ \boldsymbol{w.c} &= \boldsymbol{v1.c + v2.c} \\ \boldsymbol{w.d} &= \boldsymbol{v1.d + v2.d} \end{aligned}$$

With SSE:

```
movaps xmm0 [v1] ;        % move vector v1 to xmm0
addps xmm0, [v2] ;        % add into xmm0
movaps [w], xmm0 ;        % move to [w]
```

AVX

➤ AVX augments the "Streaming SIMD extensions" (SSE) introduced in '99 with Pentium III.

➤ Started with the 'Sandy-bridge' Family from Intel in 2011 followed by AMD.

➤ Borrowed from vector processing by adding vector instruction sets

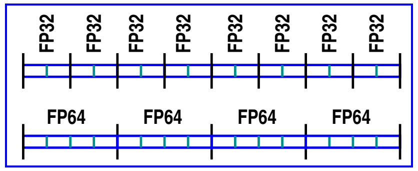
AVX2 AVX introduced in 2008 (Intel) - modified from legacy SSE and SSE2 (circa 2000)

➤ All Intel Processors in the category *core i3/i5/i7* support SSE2/AVX. Also AMD Zen/Zen2-based processors

➤ AVX Uses 16 registers (of 2 Bytes each) to perform SIMD computations on data

➤ The 16 registers can hold: 8 Sing. Prec. (32b each)

➤ or 4 Doub. Prec. (64b each)



➤ Instructions can perform FP operation (+,*,/,...) on data in registers in parallel.

Most modern processors combine some form of vector processing (multiple data) with superscalar processing (multiple instructions)

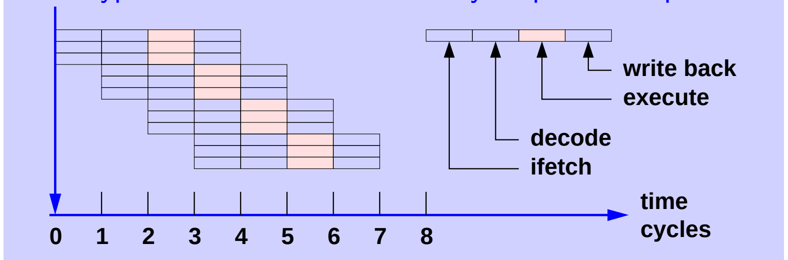
➤ Discussed next

Superscalar processors

➤ An $\boldsymbol{m}$-way superscalar processor can issue $\boldsymbol{m}$ instructions per cycle. Parallel execution of instructions is scheduled at run time (hardware). Note: $\boldsymbol{m}$ is typically small (2 or 3).

➤ "Instruction level parallelism" – different from pipelining – executes $\boldsymbol{m}$ instructions in parallel.

➤ Typical execution in a 3-way superscalar processor

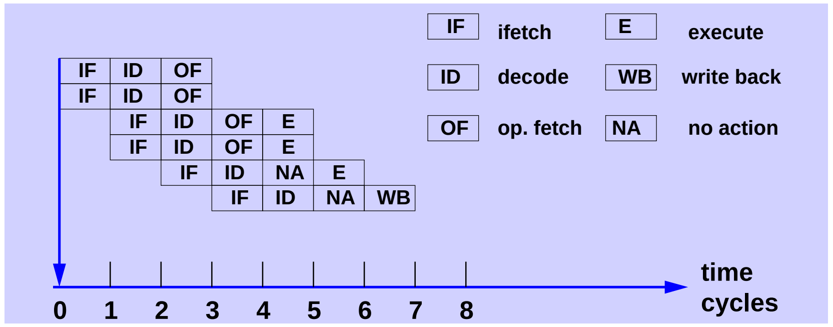


Superscalar processors: An example

The assembly code:

```
1. load R1, @1000      4. add R2, @100C
2. load R2, @1008      5. add R1, R2
3. add R1, @1004       6. store R1, @2000
```

will be executed as :



➤ In-order execution: Order of instructions is not changed - This leads to limited parallelism

➤ Out-of-order execution: instructions are rearranged (i.e., executed in an order different from initial order) to increase parallelism.

Example: The code:

```
1. load R1, @1000      4. add R2, @100C
2. add R1, @1004       5. add R1, R2
3. load R2, @1008      6. store R1, @2000
```

Has a problem: (2) depends on (1)

It can be rearranged by swapping (2) and (3) [Gives earlier code fragment]

VLIW and the Multiflow experience

➤ Above idea is a reduced version of 'trace scheduling' –

➤ A compiler technique based on unraveling parallelism 'automatically'

➤ Main idea: used 'Very Long Instruction Words' [VLIW]. Idea of 'speculation' is exploited.

➤ 'Multiflow, Inc' commercialized such computers in mid-1980s.

➤ Had a great impact on design –

Read more about the Multiflow :
http://en.wikipedia.org/wiki/Multiflow

History: A few milestones

Name (year)	PEs, topology	Prog. model
ILLIAC IV ('72)	64, mesh	SIMD
DAP ('74)	4096, mesh	SIMD
MPP ('80)	16384, h-cube	SIMD
CM-1 ('83)	4096, h-cube	SIMD
CM-2 ('85)	16384, h-cube	SIMD
BBN Butterfly ('81)	256, Butterfly	MIMD
Ncube ('85)	1024, h-cube	MIMD
IPSC-1 ('85)	128, hypercube	MIMD
Meiko ('85)	40, reconf.	MIMD
Sequent Symmetry ('86)	30, bus	Shared Mem
Encore multimax ('87)	11, bus	Shared Mem

Name (year)	PEs, topology	Prog. Model
iPSC/2 ('87)	256, h-cube	MIMD
CRAY Y-MP ('88)	16, multiport	sh mem
Ultracomputer ('88)	4096, Omega	MIMD
Cedar ('89)	32, Multistage	MIMD
iPSC/860 ('90)	512 h-cube	MIMD
CM-5 ('91)	1024 Fat-Tree	MIMD
Paragon XP ('91)	4096 2-D mesh	MIMD
Cray T3D-T3E ('96)	1024, 3-D torus	MIMD
IBM SP series ('97 –)	8192, Switch	MIMD
SGI Origin/2000 ser ('98)	h-cube + crossbar	MIMD
SGI Origin/Altix ser ('03)	h-cube+crossbar	MIMD
IBM Blue Gene ('04)	100,000+, 3D mesh+tree	MIMD
CRAY XT3 ('04)	10,000+, 3D torus	MIMD

Recent top 500s

Year	Name	PFlop/s	cores	interconne.
2008n	IBM Roadrunner	1.105	129,600	Voltaire Infiniband
2009	Cray Jaguar XT5	1.759	298,592	Cray Gemini
2010	Tianhe-IA	2.566	186,368	Proprietary
2011	Fujitsu K computer	10.51	705,024	Custom
2012j	IBM Sequoia	16.32	1,572,864	Custom
2012n	Cray Titan	17.59	560,640	Cray Gemini
2013	NUDT Tianhe-2	33.86	4,981,760	TH Express-2
2016	Sunway TaihuLight	93.01	10,649,600	Sunway
2018j	IBM Summit	122.3	2,282,544	Mellanox Infiniband
2020n	Figaka (Riken)	442.0	7,630,848	Tofu interconnect D

How about the Minnesota Supercomputer Institute?

➤ MSI was once in the top 10 of the top 500 (1996?)

➤ No longer in top 500 .

Two main machines –

Mesabi Cluster https://www.msi.umn.edu/content/mesabi

➤ HP Linux cluster with 741 nodes – Different types of nodes.

- Total of 17,784 compute cores Intel Haswell E5-2680v3 processors
- 616 Nodes have 64GB of RAM, 24 nodes have 256 GB, and 16 nodes have 1 TB of RAM each
- 40 nodes equipped with NVIDIA K20X GPUs [peak of 2.63 TFlop/s each] 128GB RAM each
- 32 Nodes have 480 GB SSD memory (each) for High Perf. I/O

Mangi https://www.msi.umn.edu/mangi

Recently added as a replacement to Ithasca [Jan 2020]

* HPC cluster with 128 AMD 'Rome' compute nodes (EPYC family, Zen-2 arch.)

* Total of 20,992 compute cores

* 12 of the nodes include 2-way Nvidia V100 GPUs + 2 nodes with 4 way GPUs, and one with 8-way GPU.

* Memory: 144 nodes × 256 GB; 10 nodes × 512 GB; 10 nodes × 2TB; or RAM.

* Interconnect: HDR100 (100 Gb/second) infiniband network

Note on Terminology: Infiniband is a standard [like 'ethernet'] for HPC with high bandwidth and low latency. The term is also used to designate a type of network. Competing networks: Mellanox, Fibre Chanel. HDR stands for High-DataRate. There is also: Enhance Data Rate (EDR), FDR, all different speeds in IB technology.