Classifications (flavors) of MIMD

• Shared address space Vs separate address spaces (an architecture concept)
  – Which memory locations can a processor access from a lw/sw instruction?
  – Two processors that do not share a memory address space can share
    information through message passing.

• Uniform memory access, UMA Vs Non-uniform memory access, NUMA
  (applies in the case of shared address space)
  – Does the delay for accessing a memory location depend on the address of that
    location?

• Shared memory programming Vs distributed memory programming
  – Variables can be shared (global) → programming with shared memory
  – Variables are private (local) → programming with distributed memory
  – Shared memory programming allows private variables (but not vise-versa)

• The concept of single Program Multiple Data (SPMD):
  (applies to both distributed memory and shared memory programming)
  – User writes one program to be executed by all processors (threads).
  – How do you make the program do different things?
Programming with private and shared (global) variables

```
int x;
x = x+1;
```

PE0's private variables

```
int x;
x = x+1;
```

PE1's private variables

```
int x;
x = x+1;
```

PE2's private variables

```
int x;
x = x+1;
```

PE3's private variables

```
/*pid = 0*/
shared *int x;
x[pid] = x[pid]+1
```

PE0

```
/*pid = 1*/
shared *int x;
x[pid] = x[pid]+1
```

PE1

```
/*pid = 2*/
shared *int x;
x[pid] = x[pid]+1
```

PE2

```
/*pid = 3*/
shared *int x;
x[pid] = x[pid]+1
```

PE3

```
x[]
```

Shared variables

SIMD (two flavors)

1) Synchronous, lockstep execution

All PEs execute the same instructions on different data

2) Vector processing

The same instruction is repeatedly executed on different data
Vector Processors

- Highly pipelined function units
- Stream data from/to vector registers to functional units
  - Vectors of data loaded from memory into vector registers (with one load instruction)
  - Vectors stored from registers to memory (with one store instruction)
- Example: Vector extension to MIPS
  - The usual 32 integer registers ($0, \ldots, $31)
  - 32 floating point registers ($f0, \ldots, f31$
  - 32 vector registers ($v0, \ldots, v31$
  - Vector instructions
    - $lv, sv$: load/store vector
    - $addv.d$: add vectors of double
    - $addvs.d$: add scalar to each element of vector
  - Significantly reduces instruction fetches

Example: DAXPY ($Y = a \times X + Y$)

- Conventional MIPS-64 code to execute $y(i) = a \times x(i) + y(i)$ for $i=0,\ldots,63$.

  \[
  \begin{align*}
  &l.d \ f0, a(\$sp) ; \text{load scalar } a \\
  &addi \ \$s2, \$s0, 512 ; \text{64 elements (}64 \times 8\text{=512 bytes)} \\
  \text{loop: l.d } \ &\ f2, 0(\$s0) ; \text{load } x(i) \\
  &mul.d \ f2, f2, f0 ; \text{multiply } a \text{ and } x(i) \\
  &l.d \ f4, 0(\$s1) ; \text{load } y(i) \\
  &add.d \ f4, f4, f2 ; \text{add } y(i) \text{ to } a \times x(i) \\
  &s.d \ f4, 0(\$s1) ; \text{store into } y(i) \\
  &addi \ \$s0, \$s0, 8 ; \text{increment index to } x \\
  &addi \ \$s1, \$s1, 8 ; \text{increment index to } y \\
  &subu \ \$t0, \$s2, \$s0 ; \text{compute bound} \\
  &bne \ \$t0, \$zero, loop ; \text{check if done}
  \end{align*}
  \]

- Vector MIPS code

  \[
  \begin{align*}
  &l.d \ f0, a(\$sp) ; \text{load scalar } a \\
  &lv \ \$v1, 0(\$s0) ; \text{load vector } x \\
  &mulvs.d \ \$v2, \$v1, f0 ; \text{vector-scalar multiply} \\
  &lv \ \$v3, 0(\$s1) ; \text{load vector } y \\
  &addv.d \ \$v4, \$v2, \$v3 ; \text{add } y \text{ to product} \\
  &sv \ \$v4, 0(\$s1) ; \text{store the result}
  \end{align*}
  \]
Using multiple Lanes

Instead of using one pipelined functional unit for all the vector elements, multiple units can be used, in parallel.

EXAMPLE: 4 pipeline units can be used, each operating on 1/4th of the vector.

Hardware Multi-threading (Sec. 6.4)

- Software-based thread context switching (Posix Threads)
  - Hardware traps on a long-latency operation
  - Software saves the context of the current thread, puts it on hold and starts the execution of another ready thread
  - Relatively large overhead (saving old context and loading new context)
  - Context = registers, PC, stack pointer, pointer to root page table, ....

- Hardware-based multithreading
  - Threads = user defined threads or compiler generated threads
  - Replicate registers (including PC and stack pointer)
  - Hardware-based thread-context switching (fast)

- Example: IBM Power5 and Pentium-4 supports hardware-based multithreading
Scheduling multiple threads

- Fine-grain multithreading
  - Switch threads after each cycle
  - Interleave instruction execution
  - If one thread stalls, others are executed

- Coarse-grain multithreading
  - Only switch on long stall (e.g., L2-cache miss)
  - Simplifies hardware, but doesn’t hide short stalls (eg, data hazards)

- SMT – Simultaneous Multi Threading
  - Schedule instructions from multiple threads
  - Instructions from independent threads execute when ready
  - Dependencies within each thread are handled separately

SMT Examples

This example assumes a 4-issue pipeline (can issue as many as 4 instructions every cycle)
Shared memory systems (CMP, multicores, manycores) (sec. 6.5)

On one chip

Cache

interconnection

Memory

I/O

Chip Multiprocessors

Shared L2 systems

Private L2 systems

• Examples: Intel Pentium

• Examples: AMD Opteron
**Example: The Sun Fire E25 K**

http://www.sun.com/servers/highend/sunfire_e25k/specs.xml

- Board = 4 SPARCS IV + 64 GB memory
- Up to 18 boards connected by crossbars
- 1.15 TB of Distributed shared memory