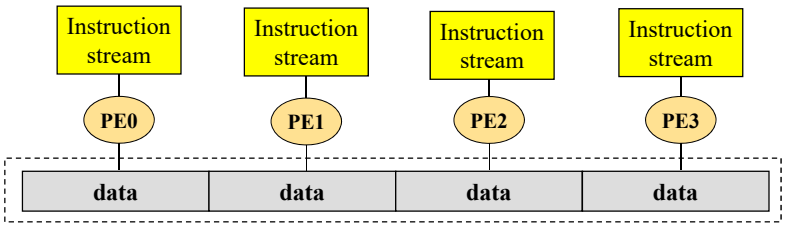
Flynn's hardware taxonomy (Section 6.3)

Looks at instructions and data parallelism. Oldest (1960's) and best known of many taxonomy proposals.

S M	I	S M	D	• S for single • M for multiple
				• I for instruction • D for data.

- SISD is a sequential computer.
- SIMD: one stream of instructions applied to multiple data.
- MIMD: multiple streams of instructions executing on multiple data.
- MISD – need to be innovative to define it.

MIMD



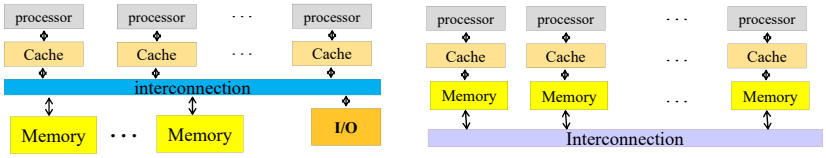
Multiple programs/threads executing on different data – However, if all PEs (processing elements) are to cooperate to solve the problem (as opposed to solving different problems), there should be interaction between the PEs.

Shared address space Vs **separate address spaces (an architecture concept)**

- The address space of an instruction stream executing on a processor consists of the "virtual" memory addresses that can be accessed from lw/sw instructions.
- Two instruction streams that do not share a memory address space can share information through *message passing*.

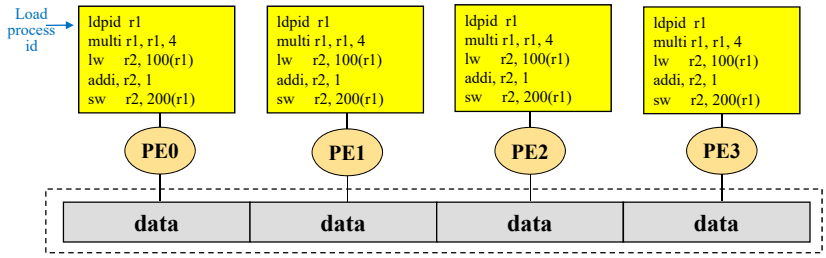
MIMD

- **Virtual addresses are mapped to physical memory locations**
  - The hardware memory system may have shared physical memory modules or distributed physical memory modules
  - Can have shared virtual address spaces on either a shared or distributed physical memory (same applies to separate virtual address spaces)



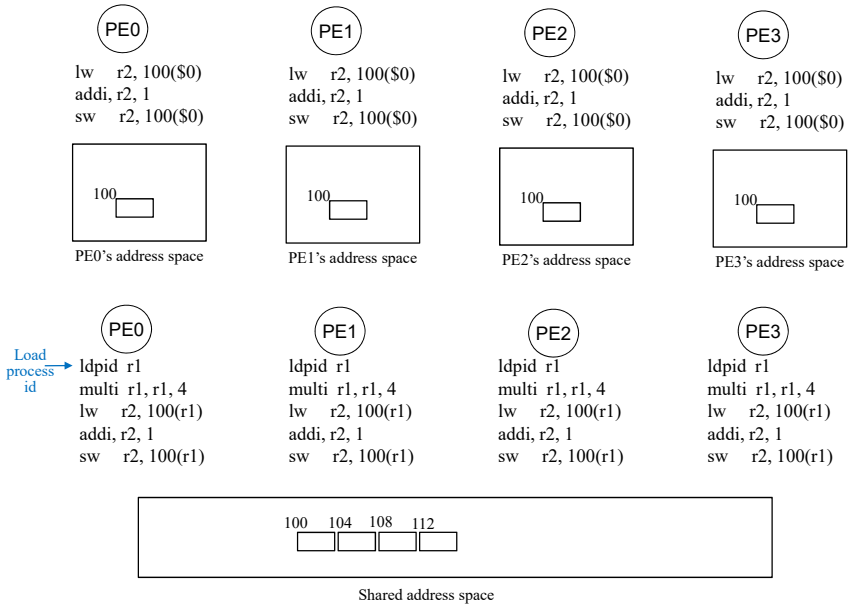
- **Uniform memory access, UMA** Vs **Non-uniform memory access, NUMA**
  - Does the delay for accessing a memory location depend on its address?.
- **Shared memory programming** Vs **distributed memory programming**
  - Variables can be shared (global) → shared memory programming
  - Variables are private (local) → distributed memory programming
  - Shared memory programming allows private as well as shared variables

The concept of SPMD

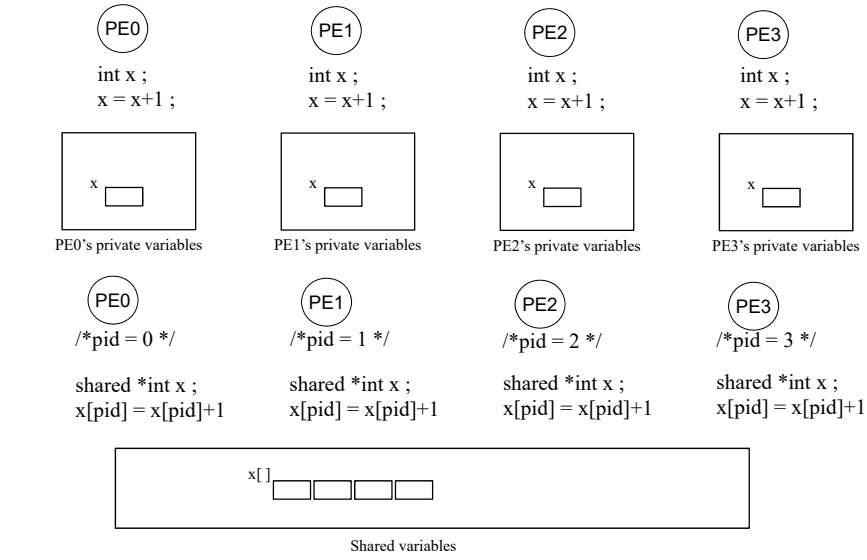


- **The concept of single Program Multiple Data (SPMD):**
  (applies to both distributed memory and shared memory MIMD programming)
  - User writes one program to be executed by all processors (threads).
  - How do you make the program do different things?

Shared Vs distributed address space

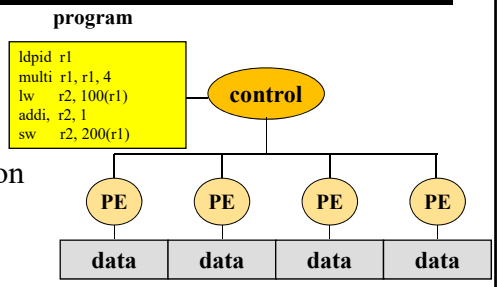


Programming with private and shared (global) variables



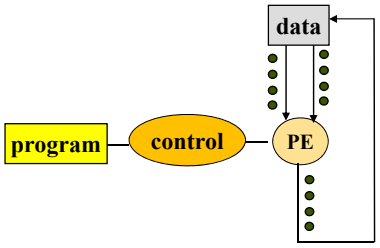
Note: in languages that allow shared variables, a variable not declared "shared" is private

SIMD (two flavors)



1) Synchronous, lockstep execution

All PEs execute the same instructions on different data

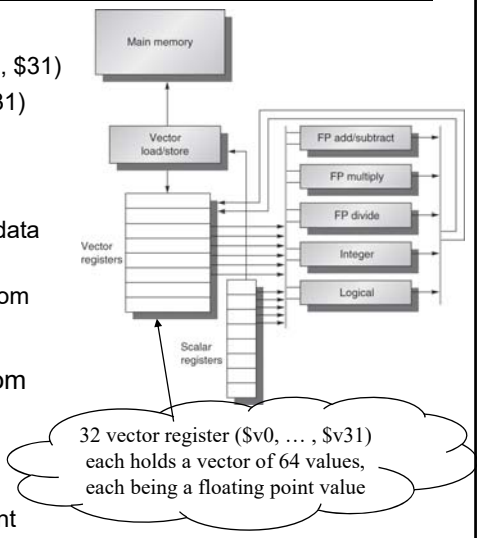


2) Vector processing

The same instruction is repeatedly executed on different data

Vector Processors

- **Example**: Vector extension to MIPS
  - The usual 32 integer registers ($0, … , $31)
  - 32 floating point registers ($f0, … , $f31)
  - 32 vector registers ($v0, … , $v31)
- Can move vectors from/to memory
  - lv → an instruction to load a vector of data from memory into a vector registers
  - sv → an instruction to store a vector from a vector register to memory
- Vector instructions to stream data from vector registers to highly pipelined functional units
  - addv.d → add two vectors
  - addvs.d → add a scalar to each element of a vector



Significantly reduces instruction fetch and execution time

EX: compute $y(i) = a * x(i) + y(i), i = 0, \ldots, 63$

- Conventional MIPS code (assuming 64-bit architecture, i.e. a word = 8 bytes).

```
      l.d   $f0,0($sp)      ;load scalar a to $f0
      addi  $s2,$s0,512     ;64 elements (64*8=512 bytes)
loop: l.d   $f2,0($s0)      ;load x(i) into $f2
      mul.d $f2,$f2,$f0     ;multiply a and x(i)
      l.d   $f4,0($s1)      ;load y(i) into $f4
      add.d $f4,$f4,$f2     ;add y(i) to a x(i)
      s.d   $f4,0($s1)      ;store back into y(i)
      addi  $s0,$s0,8       ;increment index to x
      addi  $s1,$s1,8       ;increment index to y
      subu  $t0,$s2,$s0     ;# of elements left to process
      bne   $t0,$zero,loop  ;loop if not done
```

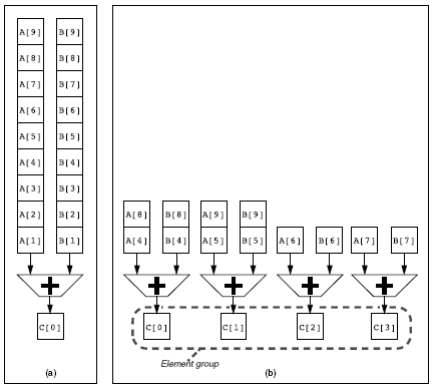
- Vector MIPS code

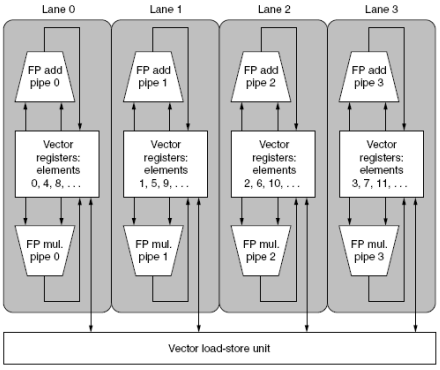
```
      l.d     $f0,0($sp)    ;load scalar a to $f0
      lv      $v1,0($s0)    ;load vector x (64 values) to $v1
      mulvs.d $v2,$v1,$f0   ;multiply vector x by scalar a
      lv      $v3,0($s1)    ;load vector y (64 values) to $v3
      addv.d  $v4,$v2,$v3   ;add two vectors
      sv      $v4,0($s1)    ;store back the result vector
```


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



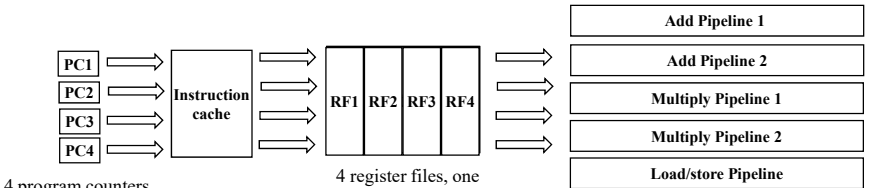


Hardware Multi-threading (Sec. 6.4)

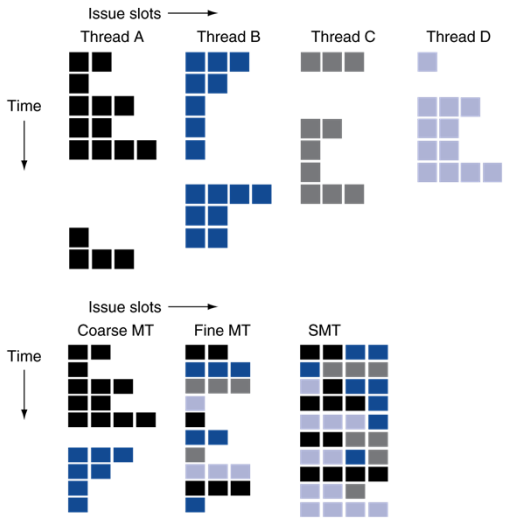
- Software-based thread context switching (Posix Threads)
  - Hardware traps on a long-latency operation
  - Software saves the context of the current thread, puts it on hold and starts the execution of another ready thread
  - Relatively large overhead (saving old context and loading new context)
  - 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 multi-threading

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



Single thread execution

This example assumes a 4-issue pipeline (can issue as many as 4 instructions every cycle)

Multithread execution