Great Idea #4: Parallelism

Software

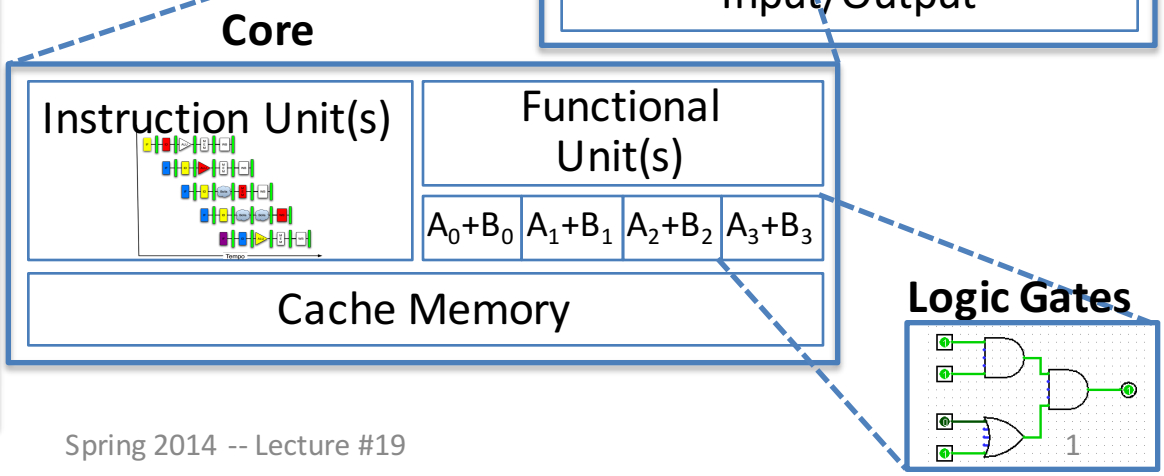
- Parallel Requests
  - Assigned to computer
  - e.g. search "Garcia"
- Parallel Threads
  - Assigned to core
  - e.g. lookup, ads
- Parallel Instructions
  - > 1 instruction @ one time
  - e.g. 5 pipelined instructions
- Parallel Data We are here
  - > 1 data item @ one time
  - e.g. add of 4 pairs of words
- Hardware descriptions
  - All gates functioning in parallel at same time

Hardware

Leverage Parallelism & Achieve High Performance

Warehouse Scale Computer

Smart Phone



Hardware vs. Software Parallelism

		Software	
		Sequential	**Concurrent**
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)

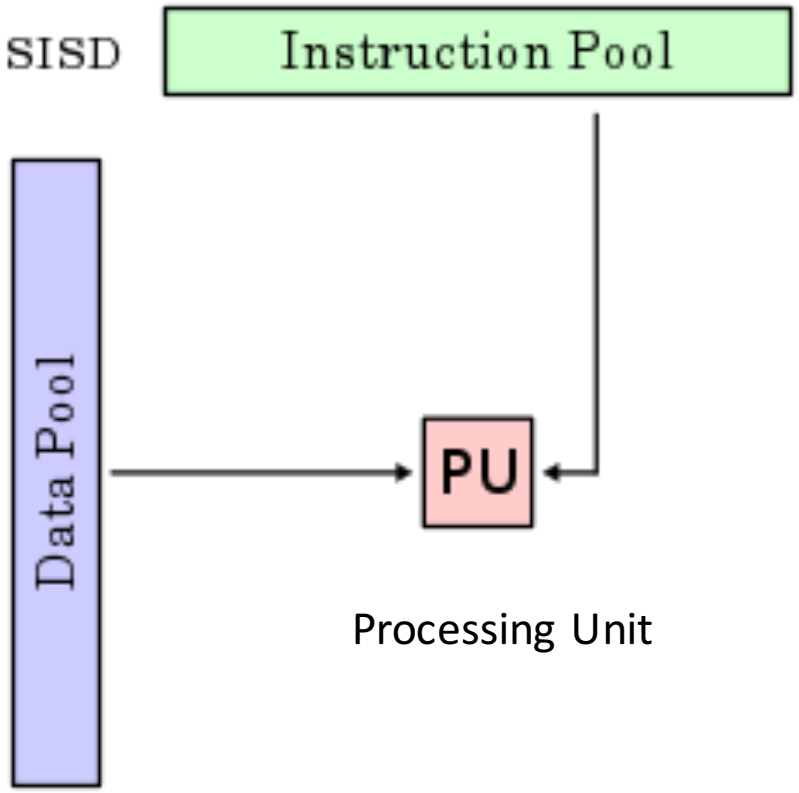
- Choice of hardware and software parallelism are independent
  - Concurrent software can also run on serial hardware
  - Sequential software can also run on parallel hardware
- *Flynn's Taxonomy* is for parallel hardware

Flynn's Taxonomy

		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 (Clovertown)

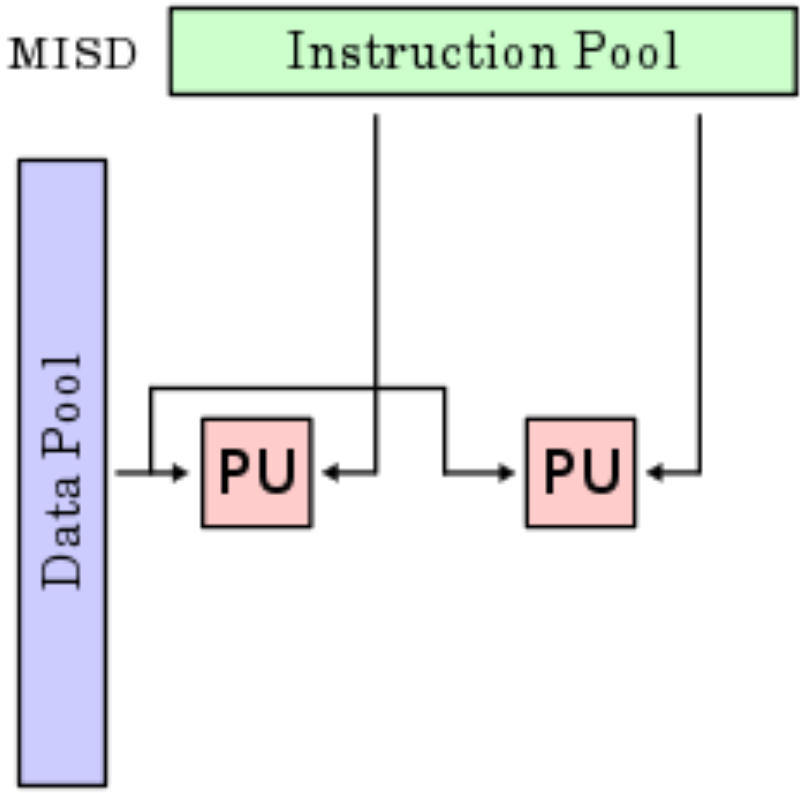
- SIMD and MIMD most commonly encountered today
- Most common parallel processing programming style: Single Program Multiple Data ("SPMD")
  - Single program that runs on all processors of an MIMD
  - Cross-processor execution coordination through conditional expressions (will see later in Thread Level Parallelism)
- SIMD: specialized function units (hardware), for handling lock-step calculations involving arrays
  - Scientific computing, signal processing, multimedia (audio/video processing)

Single Instruction/Single Data Stream



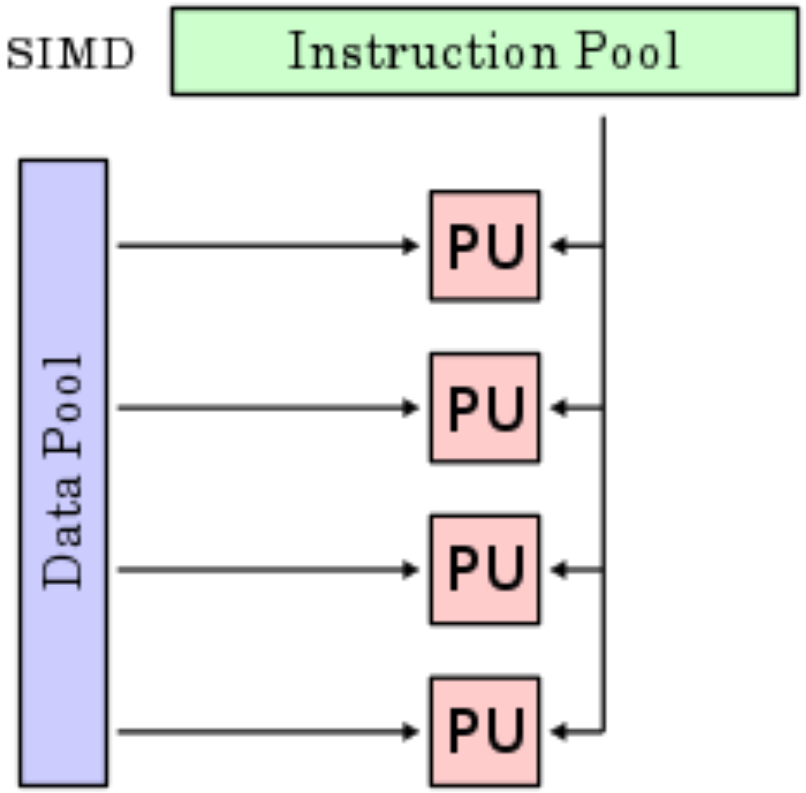
- Sequential computer that exploits no parallelism in either the instruction or data streams
- Examples of SISD architecture are traditional uniprocessor machines

Multiple Instruction/Single Data Stream



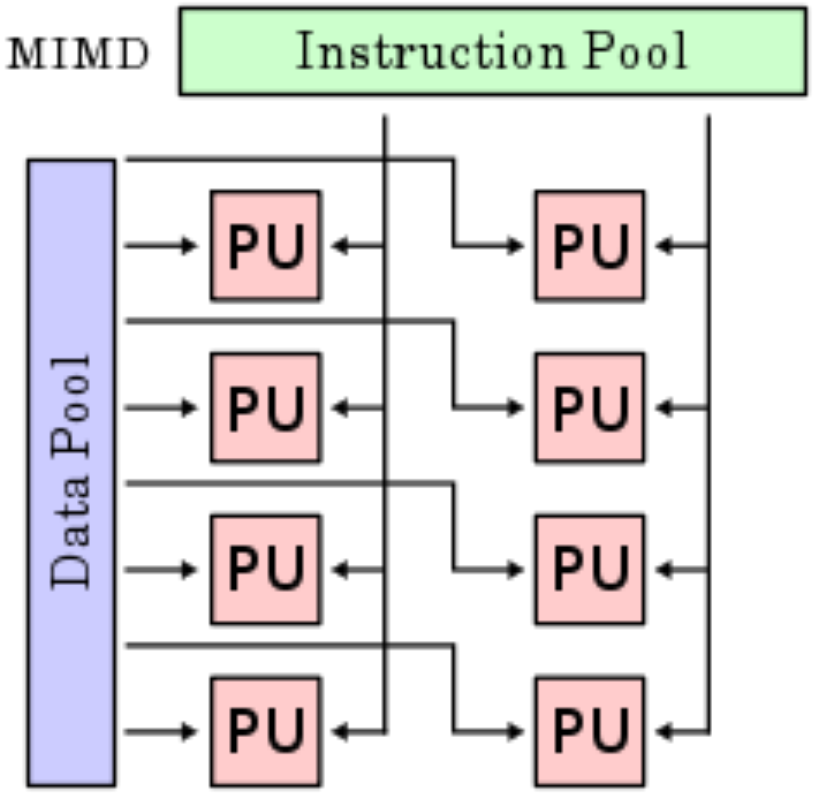
- Exploits multiple instruction streams against a single data stream for data operations that can be naturally parallelized (e.g. certain kinds of array processors)
- MISD no longer commonly encountered, mainly of historical interest only

Single Instruction/Multiple Data Stream



- Computer that applies a single instruction stream to multiple data streams for operations that may be naturally parallelized
(e.g. SIMD instruction extensions or Graphics Processing Unit)

Multiple Instruction/Multiple Data Stream



- Multiple autonomous processors simultaneously executing different instructions on different data
- MIMD architectures include multicore and Warehouse Scale Computers

SIMD Architectures

- *Data-Level Parallelism (DLP):* Executing one operation on multiple data streams
- **Example:** Multiplying a coefficient vector by a data vector (e.g. in filtering)

$$y[i] := c[i] \times x[i], \quad 0 \leq i < n$$

- Sources of performance improvement:
  - One instruction is fetched & decoded for entire operation
  - Multiplications are known to be independent
  - Pipelining/concurrency in memory access as well

Example: SIMD Array Processing

```
for each f in array:
    f = sqrt(f)
```

pseudocode

```
for each f in array {
    load f to the floating-point register
    calculate the square root
    write the result from the register to memory
}
```

SISD

```
for every 4 members in array {
    load 4 members to the SSE register
    calculate 4 square roots in one operation
    write the result from the register to memory
}
```

SIMD

Summary

- Flynn Taxonomy of Parallel Architectures
  - SIMD: Single Instruction Multiple Data
  - MIMD: Multiple Instruction Multiple Data
  - SISD: Single Instruction Single Data
  - MISD: Multiple Instruction Single Data (unused)

Data Level Parallelism and SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
- Usually specified in programs as loops

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```

- How can we reveal more data level parallelism than is available in a single iteration of a loop?
  - *Unroll the loop* and adjust iteration rate

Looping in MIPS

Assumptions:

$s0 → initial address (beginning of array)

$s1 → scalar value s

$s2 → termination address (end of array)

```
Loop:
        lw      $t0,0($s0)
        addu    $t0,$t0,$s1     #  add s to array element
        sw      $t0,0($s0)      #  store result
        addiu   $s0,$s0,4       #  move to next element
        bne     $s0,$s2,Loop    #  repeat Loop if not done
```

Loop Unrolled

```
Loop: lw   $t0,0($s0)
      addu $t0,$t0,$s1
      sw   $t0,0($s0)
      lw   $t1,4($s0)
      addu $t1,$t1,$s1
      sw   $t1,4($s0)
      lw   $t2,8($s0)
      addu $t2,$t2,$s1
      sw   $t2,8($s0)
      lw   $t3,12($s0)
      addu $t3,$t3,$s1
      sw   $t3,12($s0)
      addiu $s0,$s0,16
      bne  $s0,$s2,Loop
```

NOTE:

1. Using different registers eliminate stalls

2. Loop overhead encountered only once every 4 data iterations

3. This unrolling works if `loop_limit mod 4 = 0`

Loop Unrolled Scheduled

Note: We just switched from integer instructions to single-precision FP instructions!

```
Loop: lwc1   $t0,0($s0)
      lwc1   $t1,4($s0)
      lwc1   $t2,8($s0)
      lwc1   $t3,12($s0)
      add.s  $t0,$t0,$s1
      add.s  $t1,$t1,$s1
      add.s  $t2,$t2,$s1
      add.s  $t3,$t3,$s1
      swc1   $t0,0($s0)
      swc1   $t1,4($s0)
      swc1   $t2,8($s0)
      swc1   $t3,12($s0)
      addiu  $s0,$s0,16
      bne    $s0,$s2,Loop
```

4 Loads side-by-side:
Could replace with 4 wide SIMD Load

4 Adds side-by-side:
Could replace with 4 wide SIMD Add

4 Stores side-by-side:
Could replace with 4 wide SIMD Store

Loop Unrolling in C

- Instead of compiler doing loop unrolling, could do it yourself in C:

```
for(i=0; i<1000; i++)
  x[i] = x[i] + s;
```

Loop Unroll

```
for(i=0; i<1000; i=i+4) {
  x[i]   = x[i]   + s;
  x[i+1] = x[i+1] + s;
  x[i+2] = x[i+2] + s;
  x[i+3] = x[i+3] + s;
}
```

What is downside of doing this in C?

Generalizing Loop Unrolling

- Take a loop of **n iterations** and perform a **k-fold** unrolling of the body of the loop:
  - First run the loop with k copies of the body **floor(n/k)** times
  - To finish leftovers, then run the loop with 1 copy of the body **n mod k** times
- (Will revisit loop unrolling again when get to pipelining later in semester)