# CS 110
# Computer Architecture
# *Thread-Level Parallelism (TLP) and OpenMP Intro*

Instructor:
**Sören Schwertfeger**

**http://shtech.org/courses/ca/**

**School of Information Science and Technology SIST**

**ShanghaiTech University**

**Slides based on UC Berkley's CS61C**

# Review

- Amdahl's Law: Serial sections limit speedup
- Flynn Taxonomy
- Intel SSE SIMD Instructions
  - Exploit data-level parallelism in loops
  - One instruction fetch that operates on multiple operands simultaneously
  - 128-bit XMM registers
- SSE Instructions in C
  - Embed the SSE machine instructions directly into C programs through use of Intrinsics
  - Achieve efficiency beyond that of optimizing compiler

# New-School Machine Structures (It's a bit more complicated!)

*Software*          *Hardware*

- **Parallel Requests**
  Assigned to computer
  e.g., Search "Katz"

- **Parallel Threads**
  Assigned to core
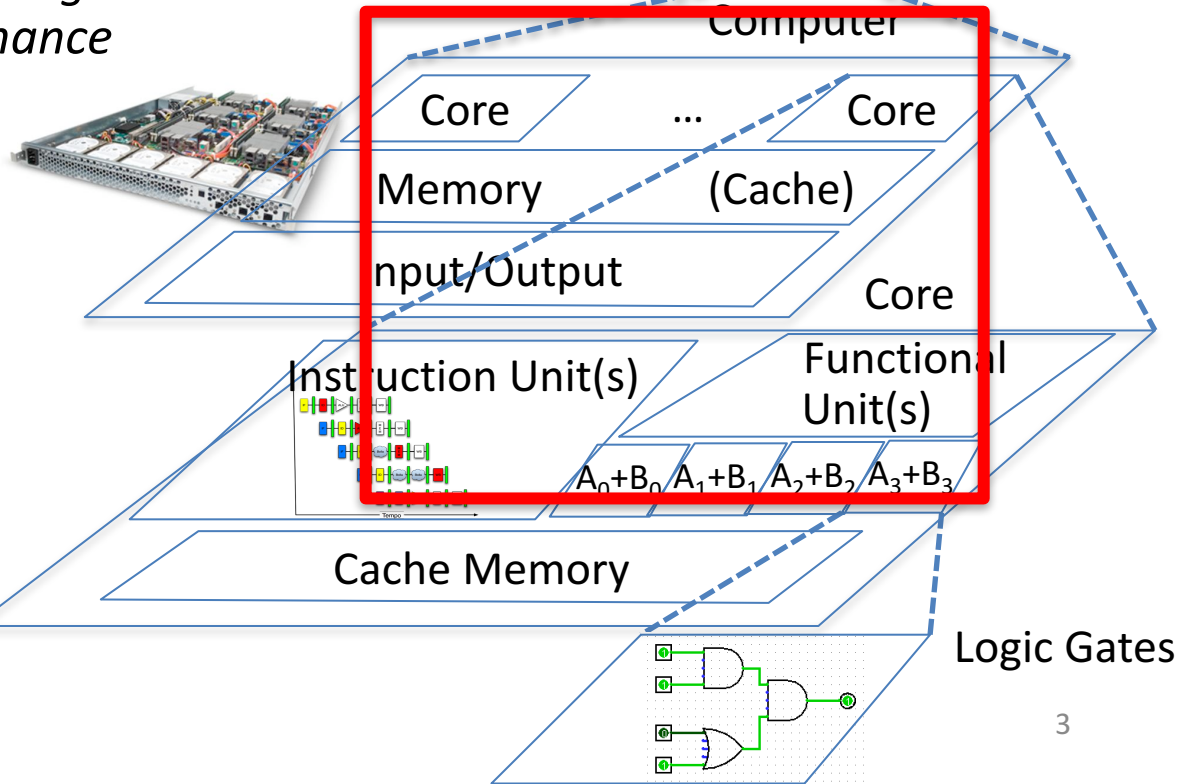  e.g., Lookup, Ads

- **Parallel Instructions**
  >1 instruction @ one time
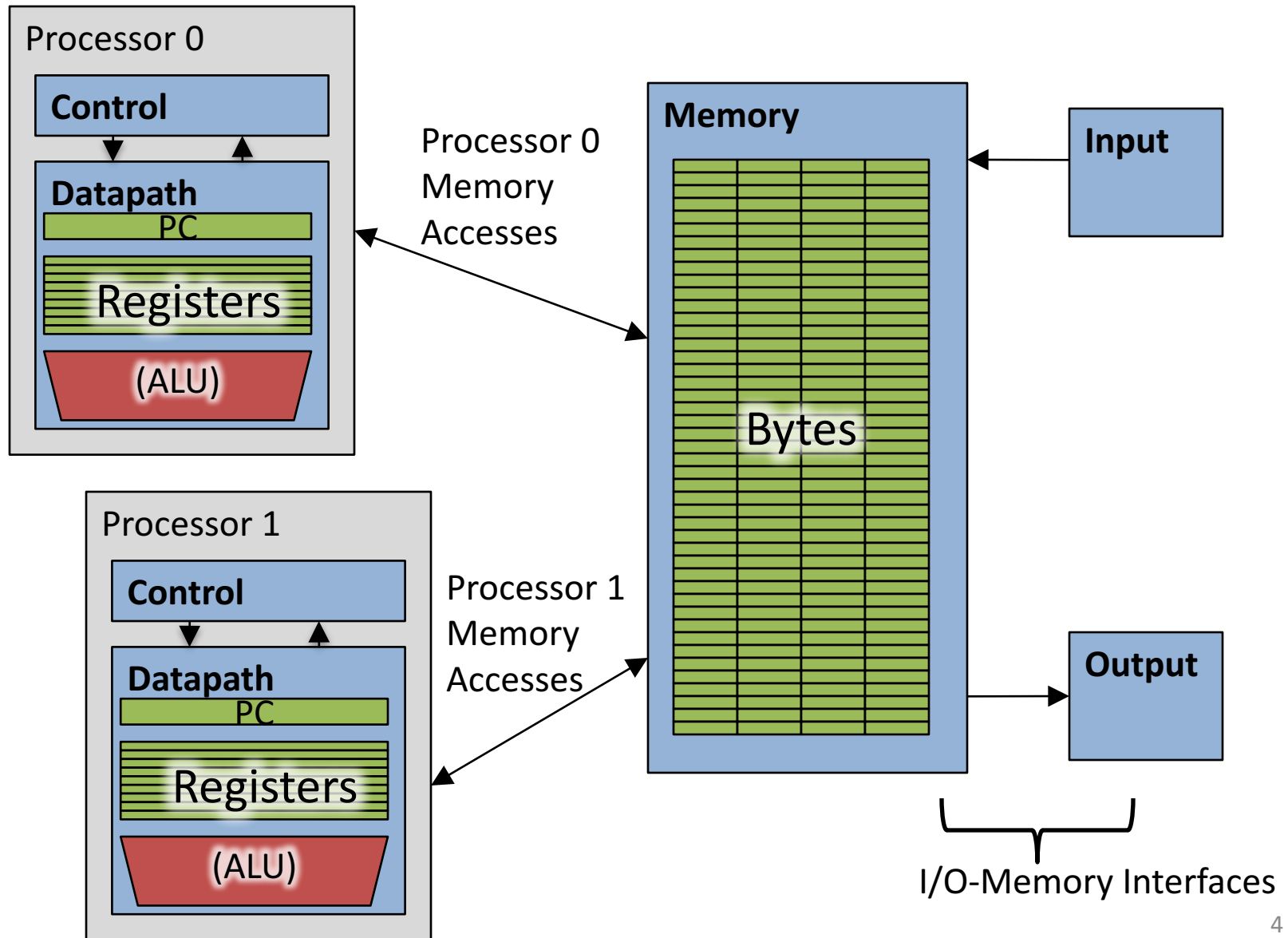  e.g., 5 pipelined instructions

- **Parallel Data**
  >1 data item @ one time
  e.g., Add of 4 pairs of words

- **Hardware descriptions**
  All gates @ one time

- **Programming Languages**

*Harness Parallelism & Achieve High Performance*

Warehouse Scale Computer

Smart Phone

Computer

Core ... Core

Memory (Cache)

Input/Output

Core

Instruction Unit(s)

Functional Unit(s)

$A_0+B_0$ $A_1+B_1$ $A_2+B_2$ $A_3+B_3$

Cache Memory

Logic Gates

3

# Simple Multiprocessor

**Processor 0**

**Control**

**Datapath**

PC

Registers

(ALU)

Processor 0 Memory Accesses

**Processor 1**

**Control**

**Datapath**

PC

Registers

(ALU)

Processor 1 Memory Accesses

**Memory**

Bytes

**Input**

**Output**

I/O-Memory Interfaces

# Multiprocessor Execution Model

- Each processor has its own PC and executes an independent stream of instructions (MIMD)
- Different processors can access the same memory space
  - Processors can communicate via shared memory by storing/loading to/from common locations
- Two ways to use a multiprocessor:
  1. Deliver high throughput for independent jobs via job-level parallelism
  2. **Improve the run time of a single program that has been specially crafted to run on a multiprocessor - a parallel-processing program**

  **Use term *core* for processor ("Multicore") because "Multiprocessor Microprocessor" too redundant**

# Transition to Multicore

Data partially collected by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond
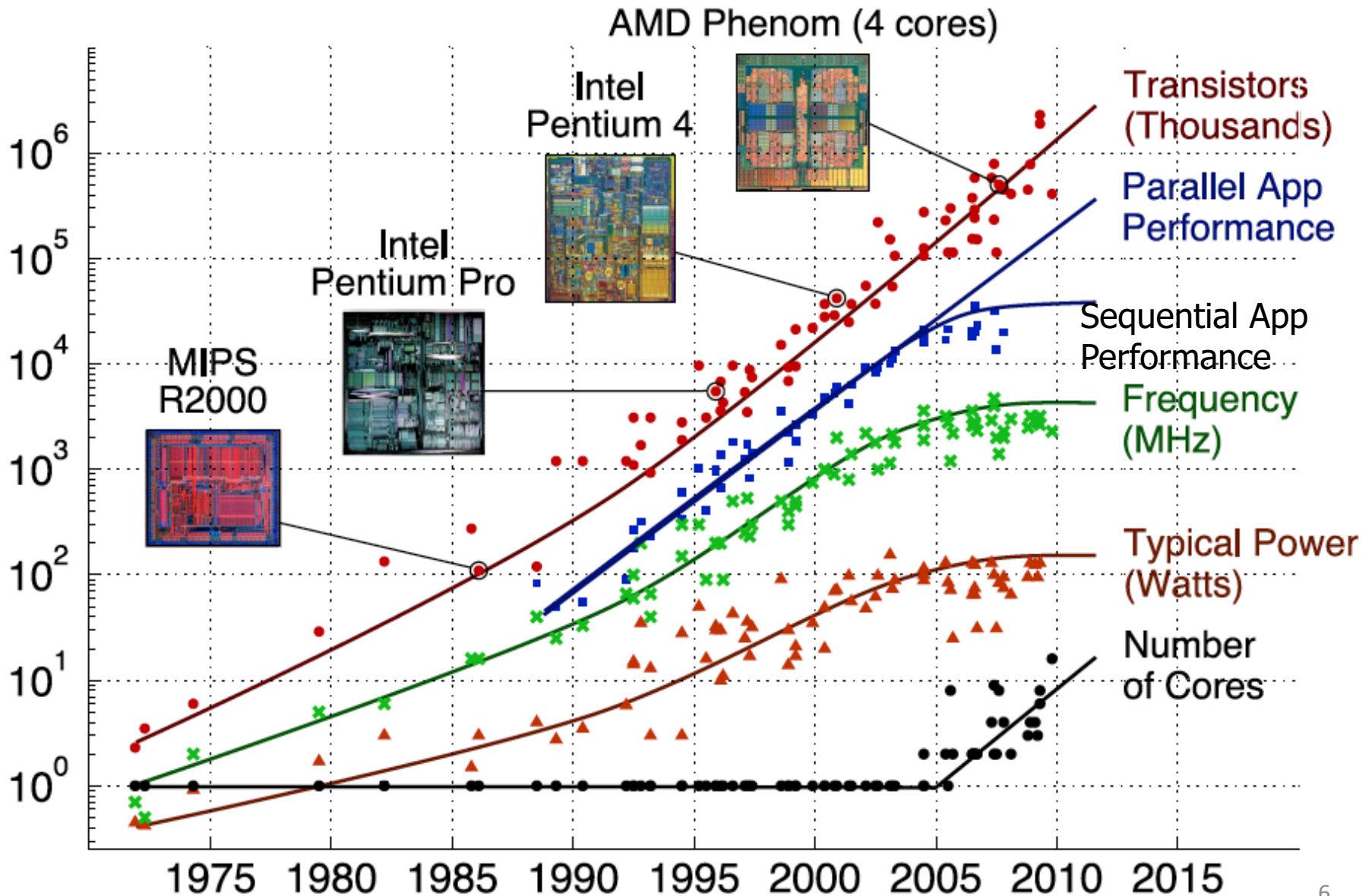
# Current Multi-Core CPUs

- Intel Core i7: 4-10 real cores
- Intel Core i9: 10-18 real cores
- Intel Xeon Platinum: 16, 24, 26, 28 real cores
- AMD Zen: 4-32 real cores
- Apple A11: 2 (high performance) + 4 (low power)
- Samsung S9 (Qualcomm Snapdragon): 4 + 4

# Parallelism the Only Path to Higher Performance

- Sequential processor performance not expected to increase much, and might go down

- If want apps with more capability, have to embrace parallel processing (SIMD and MIMD)

- In mobile systems, use multiple cores and GPUs

- In warehouse-scale computers, use multiple nodes, and all the MIMD/SIMD capability of each node

# Multiprocessors and You

- Only path to performance is parallelism
  - Clock rates flat or declining
  - SIMD: 2X width every 3-4 years
    - 512b Xeon Processors, 1024b in 2019?
  - MIMD: Add 2 cores every 2 years: 2, 4, 6, 8, 10, …
- Key challenge is to craft parallel programs that have high performance on multiprocessors as the number of processors increase – i.e., that scale
  - Scheduling, load balancing, time for synchronization, overhead for communication
- Project 3: fastest code on 10-core computer (SIMD and MIMD!)

# Threads

- *Thread:* a sequential flow of instructions that performs some task

- Each thread has a PC + processor registers and accesses the shared memory

- Each processor provides one (or more) *hardware* threads that actively execute instructions

- Operating system multiplexes multiple *software* threads onto the available hardware threads
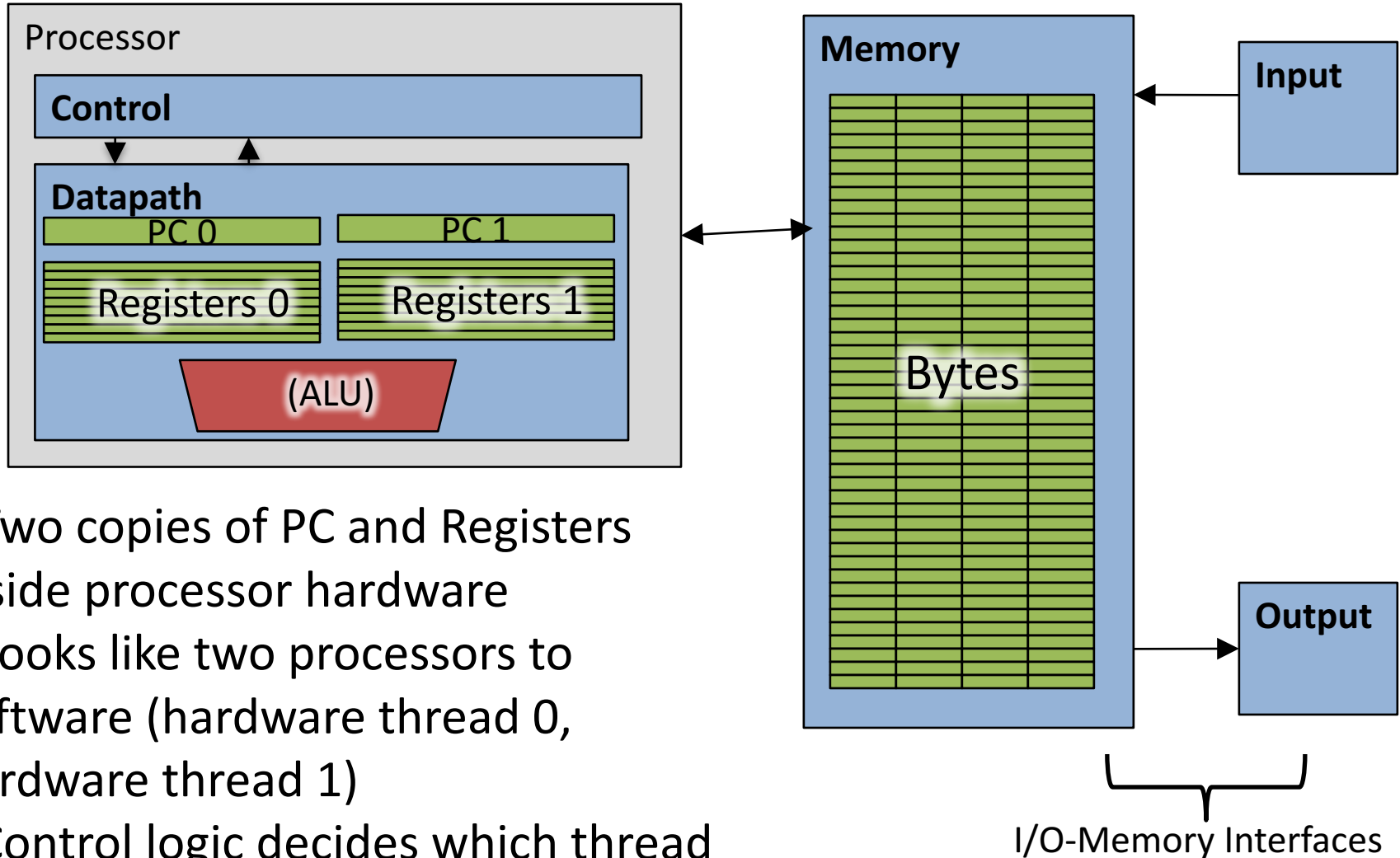
# Operating System Threads

Give the illusion of many active threads by time-multiplexing software threads onto hardware threads

- Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  - Also if one thread is blocked waiting for network access or user input
- Can make a different software thread active by loading its registers into a hardware thread's registers and jumping to its saved PC

# Hardware Multithreading

- Basic idea: Processor resources are expensive and should not be left idle

- Long memory latency to memory on cache miss?

- Hardware switches threads to bring in other useful work while waiting for cache miss

- Cost of thread context switch must be much less than cache miss latency

- Put in redundant hardware so don't have to save context on every thread switch:
  - PC, Registers

- Attractive for apps with abundant TLP
  - Commercial multi-user workloads

# Hardware Multithreading
# (aka Hyperthreading)

**Processor**

**Control**

**Datapath**

PC 0 · PC 1

Registers 0 · Registers 1

(ALU)

**Memory**

Bytes

**Input**

**Output**

I/O-Memory Interfaces

- Two copies of PC and Registers inside processor hardware
- Looks like two processors to software (hardware thread 0, hardware thread 1)
- Control logic decides which thread to execute an instruction from next

13

# Multithreading vs. Multicore

- Multithreading => Better Utilization
  - ≈5% more hardware, 1.10X better performance?
  - Share integer adders, floating-point units, all caches (L1 I$, L1 D$, L2$, L3$), Memory Controller
- Multicore => Duplicate Processors
  - ≈50% more hardware, ≈2X better performance?
  - Share outer caches (L2$, L3$), Memory Controller
- Modern machines do both
  - Multiple cores with multiple threads per core

# Sören's MacBook

- `sysctl -a | grep hw\.`

MacBookPro11,3

…

hw.physicalcpu: 4

hw.logicalcpu: 8

…

hw.cpufrequency = 2,800,000,000

hw.memsize = 17,179,869,184

hw.cachelinesize = 64

hw.l1icachesize: 32,768

hw.l1dcachesize: 32,768

hw.l2cachesize: 262,144

hw.l3cachesize: 6,291,456

on Linux:

`cat /proc/cpuinfo`

15

# Sören's iPad Air 2

Apple A8X processor:

- 3 cores!
- L1 $: 64KB data, 64KB instruction
- L2 $: 2MB
- L3 $: 4MB
- Max 1.5GHz clock
- 64bit ARM ISA
- 2 GB RAM

Comparison:

iPad Pro: A9X processor:

- Back to 2 cores…
- L1 $: 64KB data, 64KB instruction
- L2 $: 3MB
- No L3 $: double memory bandwidth…
- 4 or 2 GB RAM

# 100s of (Mostly Dead) Parallel Programming Languages

| | | | |
|---|---|---|---|
| ActorScript | Concurrent Pascal | JoCaml | Orc |
| Ada | Concurrent ML | Join | Oz |
| Afnix | Concurrent Haskell | Java | Pict |
| Alef | Curry | Joule | Reia |
| Alice | CUDA | Joyce | SALSA |
| APL | E | LabVIEW | Scala |
| Axum | Eiffel | Limbo | SISAL |
| Chapel | Erlang | Linda | SR |
| Cilk | Fortan 90 | MultiLisp | Stackless Python |
| Clean | Go | Modula-3 | SuperPascal |
| Clojure | Io | Occam | VHDL |
| Concurrent C | Janus | occam-π | XC |

# OpenMP

- OpenMP is a language extension used for multi-threaded, shared-memory parallelism
  - Compiler Directives (inserted into source code)
  - Runtime Library Routines (called from your code)
  - Environment Variables (set in your shell)
- Portable
- Standardized
- Easy to compile: `cc –fopenmp name.c`

# Shared Memory Model with Explicit Thread-based Parallelism

- Multiple threads in a shared memory environment, explicit programming model with full programmer control over parallelization
- **Pros:**
  - Takes advantage of shared memory, programmer need not worry (that much) about data placement
  - Compiler directives are simple and easy to use
  - Legacy serial code does not need to be rewritten
- **Cons:**
  - Code can only be run in shared memory environments
  - Compiler must support OpenMP (e.g. gcc 4.2)

# OpenMP in CS110

- OpenMP is built on top of C, so you don't have to learn a whole new programming language
  - Make sure to add `#include <omp.h>`
  - Compile with flag: `gcc -fopenmp`
  - Mostly just a few lines of code to learn
- You will NOT become experts at OpenMP
  - Use slides as reference, will learn to use in lab
- **Key ideas:**
  - Shared vs. Private variables
  - OpenMP directives for parallelization, work sharing, synchronization

# OpenMP Programming Model

- **Fork - Join Model:**
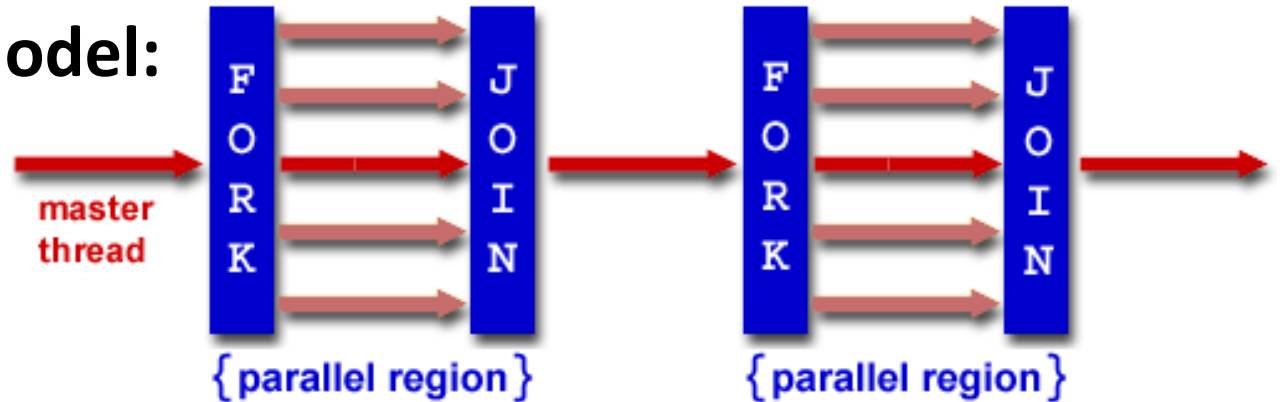


- OpenMP programs begin as single process (*master thread*) and executes sequentially until the first parallel region construct is encountered
  - *FORK:* Master thread then creates a team of parallel threads
  - Statements in program that are enclosed by the parallel region construct are executed in parallel among the various threads
  - *JOIN:* When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

# OpenMP Extends C with Pragmas

- *Pragmas* are a preprocessor mechanism C provides for language extensions
- Commonly implemented pragmas: structure packing, symbol aliasing, floating point exception modes (not covered)
- Good mechanism for OpenMP because compilers that don't recognize a pragma are supposed to ignore them
  - Runs on sequential computer even with embedded pragmas

# `parallel` Pragma and Scope

- Basic OpenMP construct for parallelization:

```
#pragma omp parallel
{
   /* code goes here */
}
```

This is annoying, but curly brace MUST go on separate line from #pragma

  - *Each* thread runs a copy of code within the block
  - Thread scheduling is *non-deterministic*

- OpenMP default is *shared* variables
  - To make private, need to declare with pragma:

```
#pragma omp parallel private (x)
```

# What Kind of Threads?

- OpenMP threads are operating system (software) threads.

- OS will multiplex requested OpenMP threads onto available hardware threads.

- Hopefully each gets a real hardware thread to run on, so no OS-level time-multiplexing.

- But other tasks on machine can also use hardware threads!

# OMP_NUM_THREADS

- OpenMP intrinsic to set number of threads:

  `omp_set_num_threads(x);`

- OpenMP intrinsic to get number of threads:

  `num_th = omp_get_num_threads();`

- OpenMP intrinsic to get Thread ID number:

  `th_ID = omp_get_thread_num();`

# Parallel Hello World

```c
#include <stdio.h>
#include <omp.h>
int main () {
  int nthreads, tid;

  /* Fork team of threads with private var tid */
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num(); /* get thread id */
    printf("Hello World from thread = %d\n", tid);

    /* Only master thread does this */
    if (tid == 0) {
      nthreads = omp_get_num_threads();
      printf("Number of threads = %d\n", nthreads);
    }
  }  /* All threads join master and terminate */
}
```

# Data Races and Synchronization

- Two memory accesses form a *data race* if from different threads to same location, and at least one is a write, and they occur one after another
- If there is a data race, result of program can vary depending on chance (which thread first?)
- Avoid data races by synchronizing writing and reading to get deterministic behavior
- Synchronization done by user-level routines that rely on hardware synchronization instructions
- (more later)

# Analogy: Buying Milk

- Your fridge has no milk. You and your roommate will return from classes at some point and check the fridge

- Whoever gets home first will check the fridge, go and buy milk, and return

- What if the other person gets back while the first person is buying milk?

  - You've just bought twice as much milk as you need!

- It would've helped to have left a note…

# Lock Synchronization (1/2)

- Use a "Lock" to grant access to a region (*critical section*) so that only one thread can operate at a time
  - Need all processors to be able to access the lock, so use a location in shared memory as *the lock*
- Processors read lock and either wait (if locked) or set lock and go into critical section
  - **0** means lock is free / open / unlocked / lock off
  - **1** means lock is set / closed / locked / lock on

# Lock Synchronization (2/2)

- Pseudocode:

Can loop/idle here
if loked

Check lock
Set the lock
Critical section
(e.g. change shared variables)
Unset the lock

# Possible Lock Implementation

- ## Lock (a.k.a. busy wait)

```
Get_lock:                         # $s0 -> addr of lock
        addiu $t1,$zero,1         # t1 = Locked value
Loop:   lw $t0,0($s0)             # load lock
        bne $t0,$zero,Loop        # loop if locked
Lock:   sw $t1,0($s0)             # Unlocked, so lock
```

- ## Unlock

```
Unlock:
        sw $zero,0($s0)
```

- ## Any problems with this?

# Possible Lock Problem

- Thread 1

```
      addiu $t1,$zero,1
Loop: lw $t0,0($s0)



      bne $t0,$zero,Loop


Lock: sw $t1,0($s0)
```

- Thread 2

```
          addiu $t1,$zero,1
    Loop: lw $t0,0($s0)



          bne $t0,$zero,Loop


    Lock: sw $t1,0($s0)
```

**Time**

*Both threads think they have set the lock!*
*Exclusive access not guaranteed!*

# Hardware Synchronization

- Hardware support required to prevent an interloper (another thread) from changing the value
  - *Atomic* read/write memory operation
  - No other access to the location allowed between the read and write
- How best to implement in software?
  - Single instr?  Atomic swap of register ↔ memory
  - Pair of instr?  One for read, one for write

# Synchronization in MIPS

- *Load linked:*      `ll rt,off(rs)`
- *Store conditional:*    `sc rt,off(rs)`
  - Returns **1** (success) if location has not changed since the `ll`
  - Returns **0** (failure) if location has changed

- Note that `sc` *clobbers* the register value being stored (`rt`)!
  - Need to have a copy elsewhere if you plan on repeating on failure or using value later

# Synchronization in MIPS Example

- Atomic swap (to test/set lock variable)

  Exchange contents of register and memory:
  **$s4 ↔ Mem($s1)**

```
try: add $t0,$zero,$s4 #copy value
     ll  $t1,0($s1)    #load linked
     sc  $t0,0($s1)    #store conditional
     beq $t0,$zero,try #loop if sc fails
     add $s4,$zero,$t1 #load value in $s4
```

`sc` would fail if another threads executes `sc` here

# Test-and-Set

- In a single atomic operation:
  - *Test* to see if a memory location is set (contains a 1)
  - *Set* it (to 1) if it isn't (it contained a zero when tested)
  - Otherwise indicate that the Set failed, so the program can try again
  - While accessing, no other instruction can modify the memory location, including other Test-and-Set instructions
- Useful for implementing lock operations
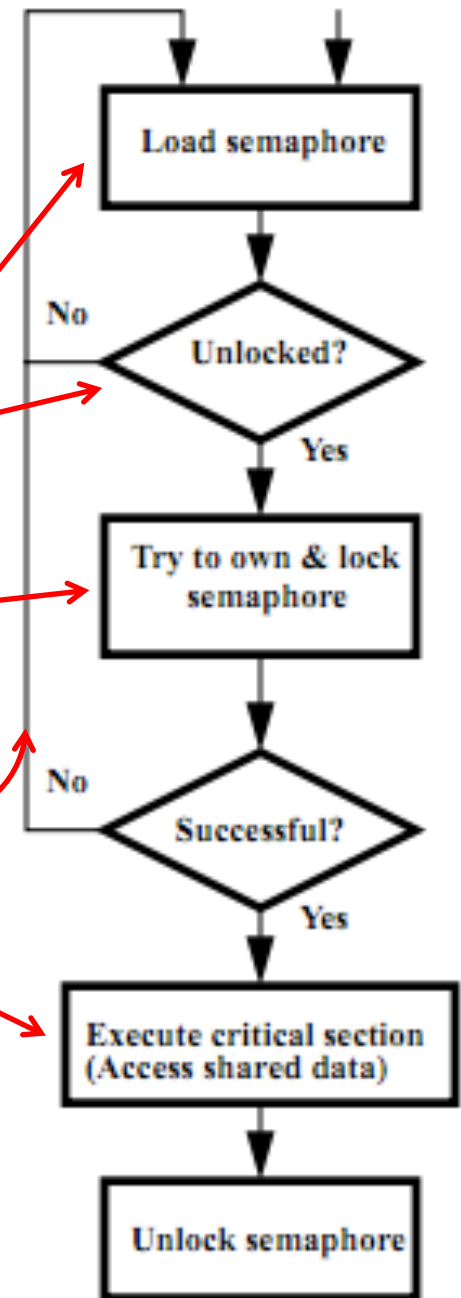
# Test-and-Set in MIPS

- Example: MIPS sequence for implementing a T&S at ($s1)

**Load semaphore**

**Unlocked?** No / Yes
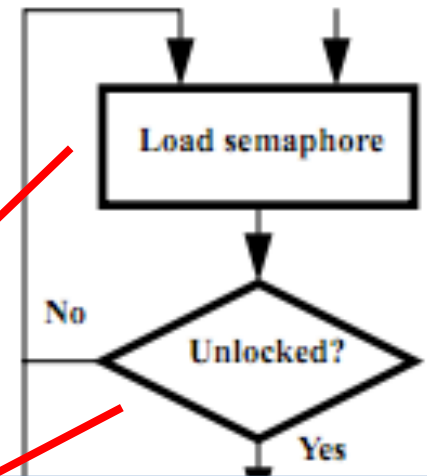
Idea is that not for programmers to use this directly, but as a tool for enabling implementation of parallel libraries

```
        # critical section

Unlock:
        sw $zero,0($s1)
```

**Execute critical section (Access shared data)**

**Unlock semaphore**