CS61C: Great Ideas in Computer Architecture (aka Machine Structures)

Lecture 16: Thread-Level Parallelism

Instructors: Rosalie Fang, **Charles Hong**, Jero Wang

Announcements

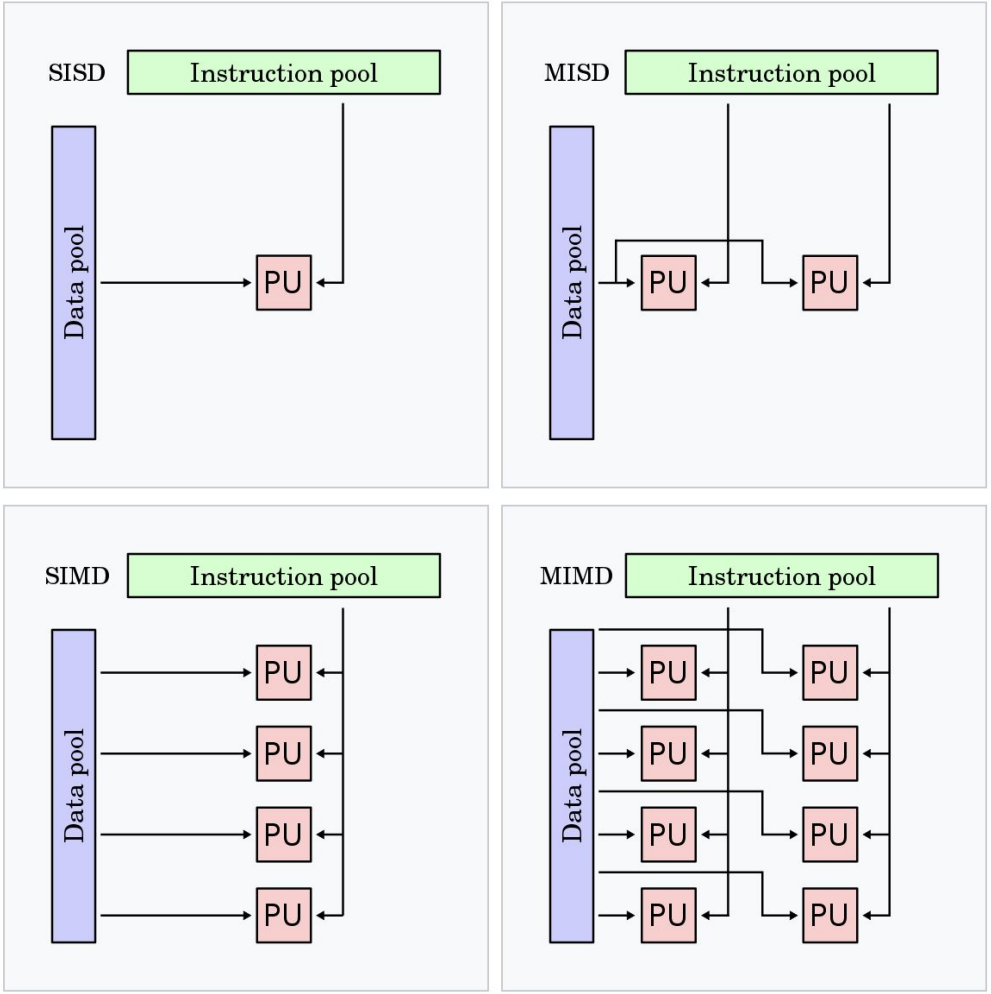
- Exam prep sections continue
    - RISC-V Datapath: Friday 1-3pm, next Tuesday 6-8pm (Cory 540AB)
- Labs 5 and 6 due today 7/20

Agenda

- Flynn’s Taxonomy
- Parallel Computer Architectures
- Threads
- Thread-Level Parallelism
- OpenMP
- Data Races and Critical Sections

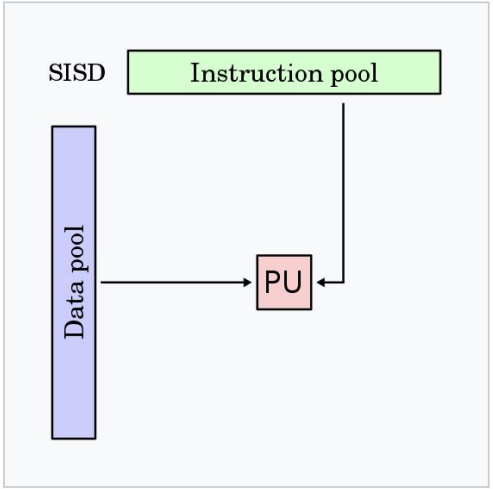
Flynn’s Taxonomy

Flynn’s Taxonomy



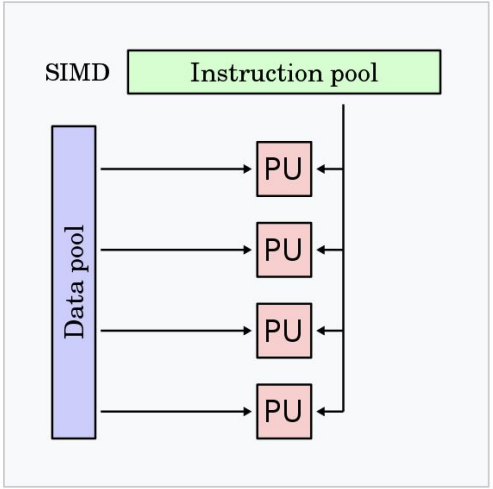
Single Instruction, Single Data (SISD)

- Sequential computer with no parallelism
- Example:
  - The RISC-V datapath we covered in this class.



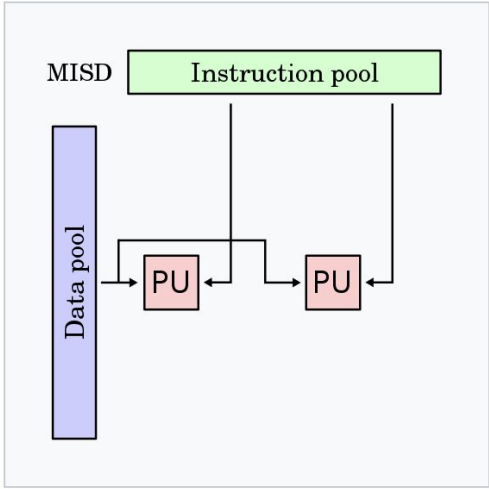
Single Instruction, Multiple Data (SIMD)

- Computer applies a single instruction stream to multiple data streams
- Example:
  - Intel instrinsics, from yesterday
  - GPUs (sort of).



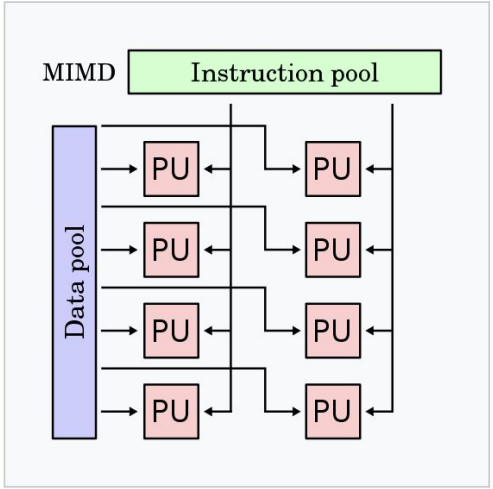
Multiple Instruction, Single Data (MISD)

- Computer applies multiple different operations to the same data
- Example:
  - Pipelining?
  - Deep learning acceleration chips? (repeatedly performs multiplies and adds on the same data)
  - Strictly speaking, no examples today.



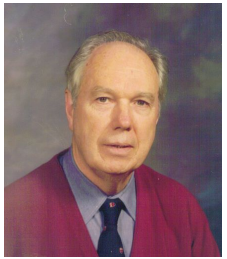
Multiple Instruction, Multiple Data (MIMD)

- Computer applies multiple different operations to multiple different data
- Example:
  - All modern processors.

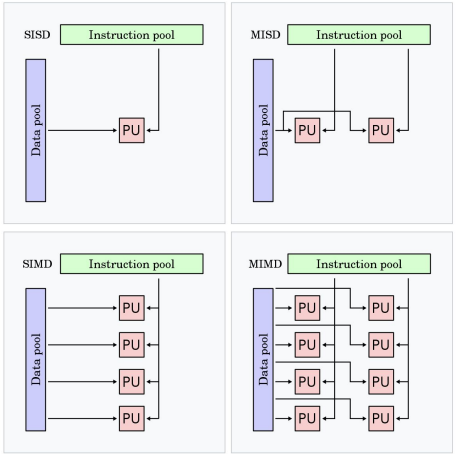


Flynn’s Taxonomy

- A way of classifying parallelism in **hardware**
- Today’s topic: Single Program Multiple Data (“SPMD”)
  - Most common parallel processing programming style
  - Single program that utilizes all processors of a MIMD.

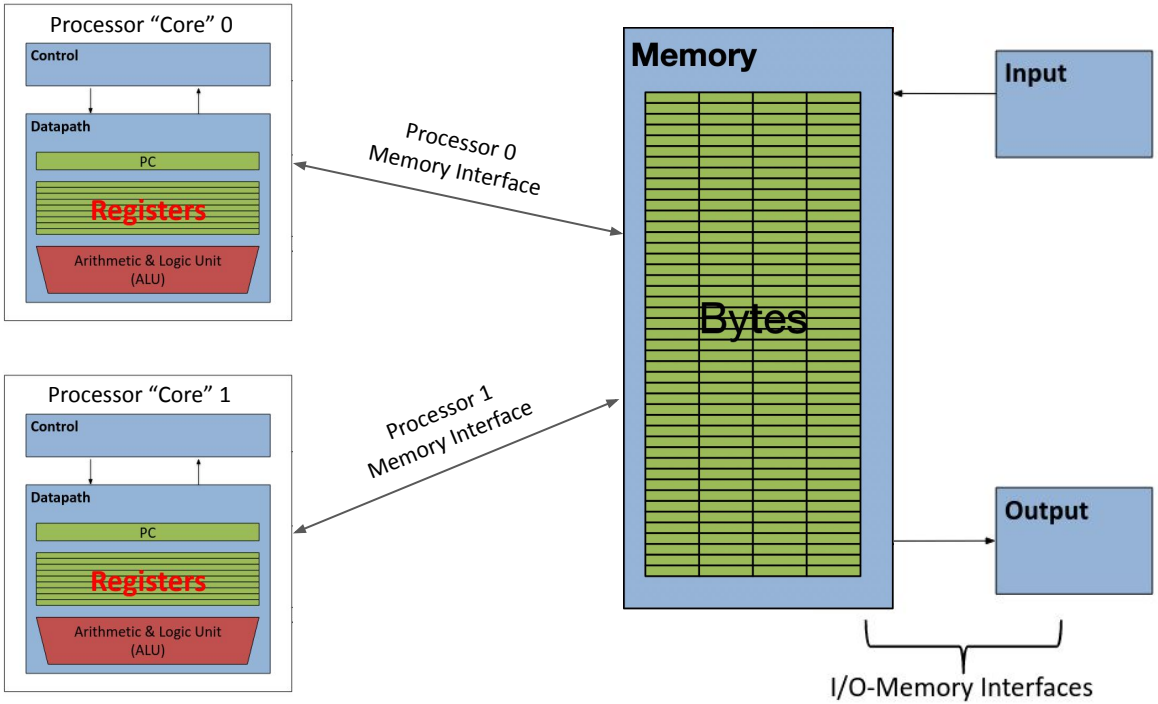


Prof. Michael J. Flynn



Parallel Computer Architectures

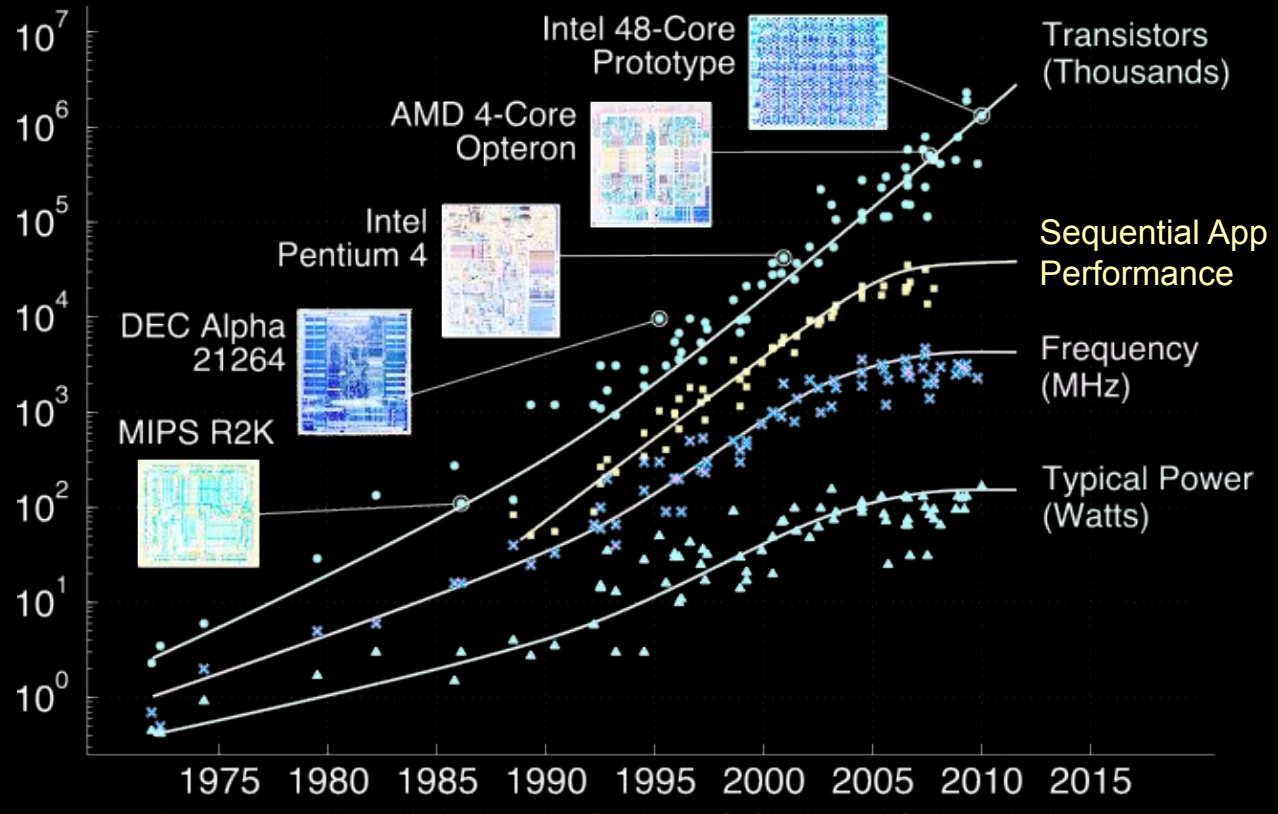
Example: CPU with Two Cores



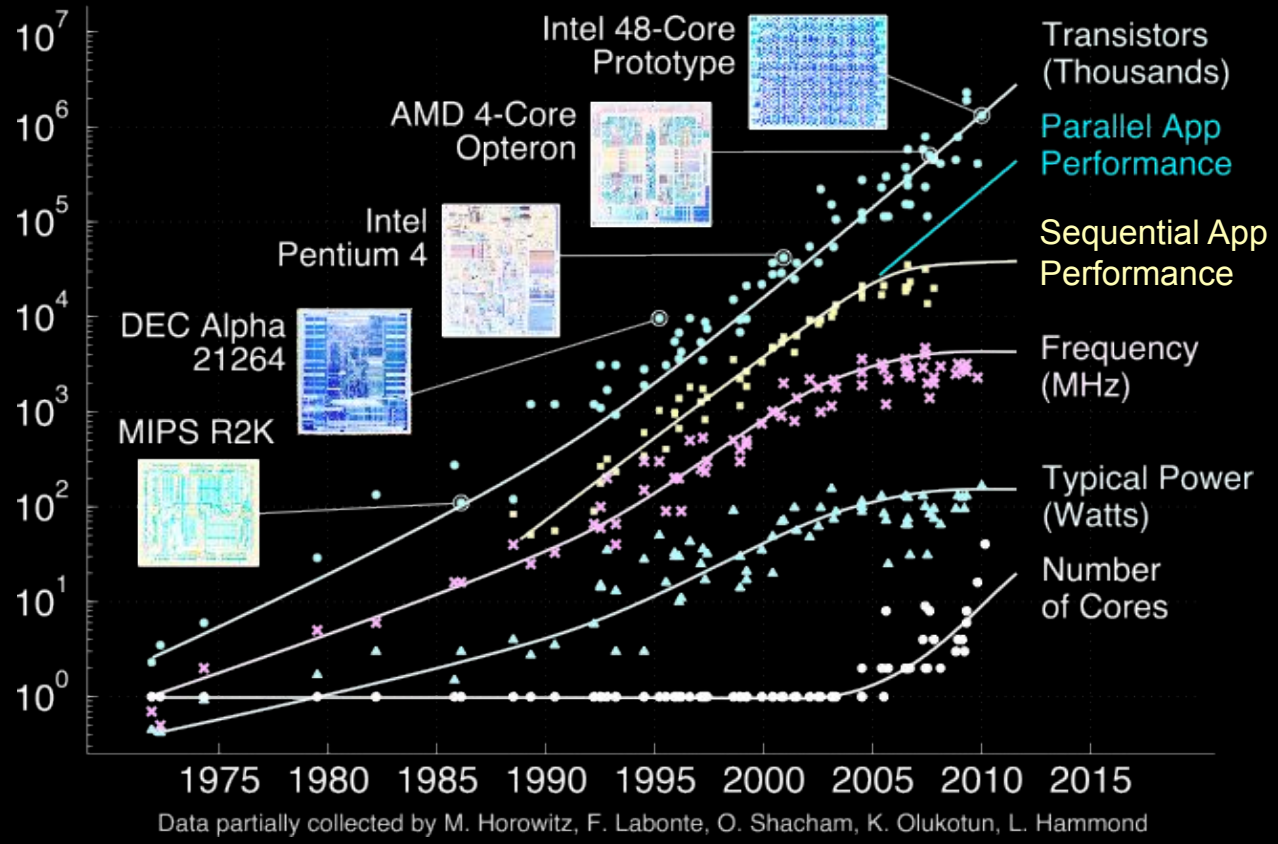
Multiprocessor Execution Model

- Each processor (core) executes its own instructions
- **Separate** resources (not shared)
  - Datapath (PC, registers, ALU)
  - Highest level caches (e.g., 1st and 2nd)
- **Shared** resources
  - Memory (DRAM)
  - Often 3rd level cache
    - Often on same silicon chip
    - But not a requirement
- Nomenclature
  - Could be called “Multiprocessor Microprocessor”
  - More commonly: Multicore processor
    - E.g., four core CPU (central processing unit)
    - Executes four different instruction streams simultaneously.

Transition to Multiprocessing



Transition to Multiprocessing



Threads

Recall: New School Machine Architecture

Software

Parallel Requests
Assigned to computer
e.g., Search “Cats”

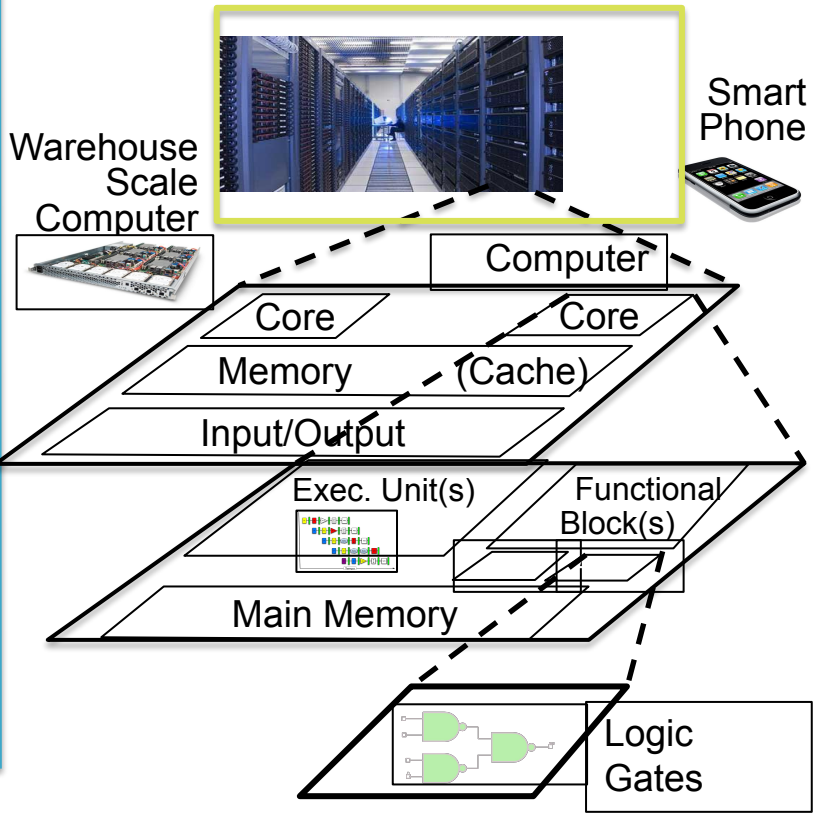
Parallel Threads
Assigned to core e.g., Lookup, Ads

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

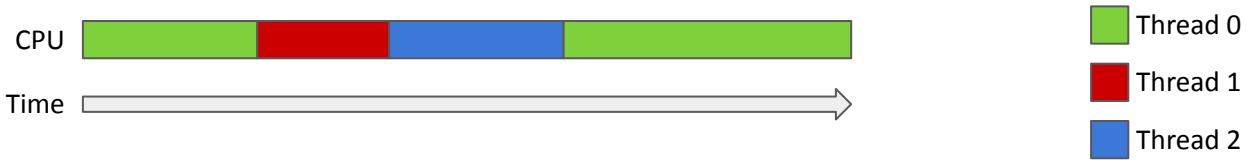
Hardware descriptions
All gates work in parallel at same time

Hardware



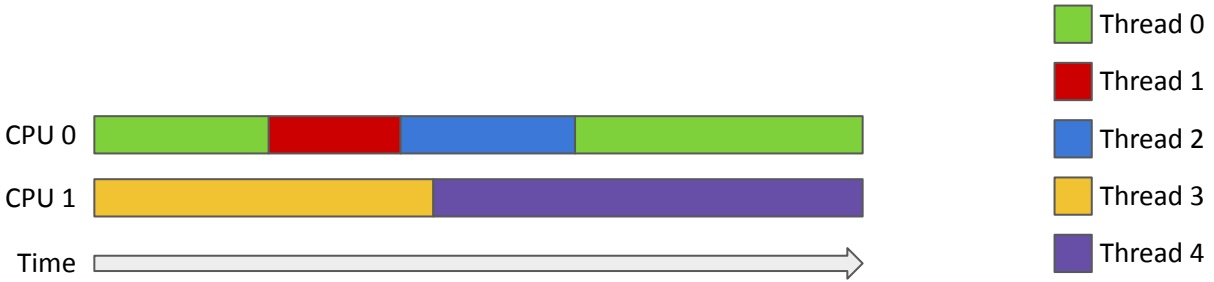
Threads

- A Thread stands for “thread of execution”, is a single stream of instructions
  - A program / process can split, or fork itself into separate threads, which can (in theory) execute simultaneously.
  - An easy way to describe/think about parallelism
- With a single core, a single CPU can execute many threads by **time sharing**



Software vs Hardware Threads (1/2)

- Each core provides one or more hardware threads that actively execute instructions
  - If more than 1 thread per core, called simultaneous multithreading, or hyper-threading
- Operating system multiplexes multiple software threads onto the available hardware threads
  - Remove a software thread from a hardware thread by interrupting its execution and saving its registers and PC into memory
  - How to load a new software thread?

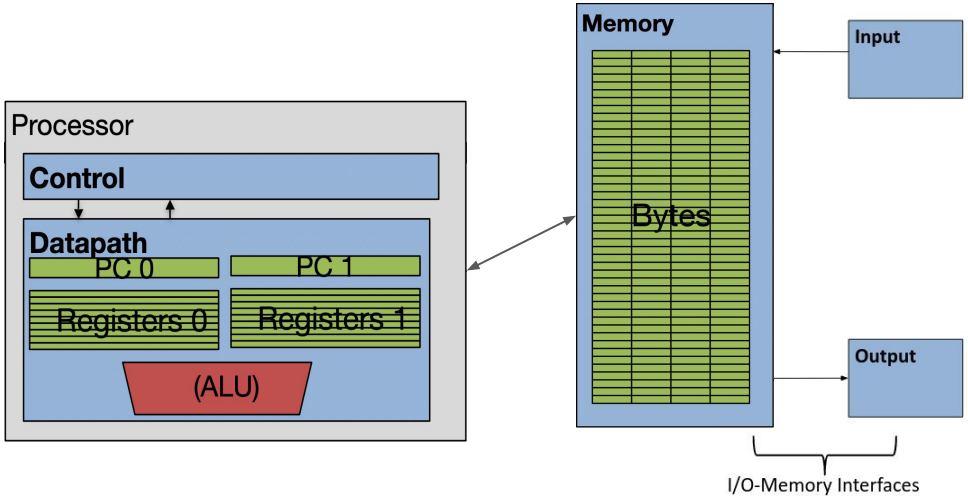


Software vs Hardware Threads (2/2)

- On most modern computers, number of active software threads >> number of available hardware threads, so most threads are idle at any given time
  - One reason why runtime can vary, even when running the same program
  - For Project 4, we'll run programs on dedicated systems, so you have the full resource allocation.

Hardware Multithreading

- Called simultaneous multithreading (SMT), or hyperthreading (HT)
- Basic idea: Hardware switches threads to bring in other useful work while waiting for memory
- Two copies of PC and Registers inside processor hardware
  - Execution resources (ALU, etc.) shared
- Particularly effective when paired with superscalar processors (multiple execution units)
- Used by Intel and AMD, but not by ARM processors



Multiprocessing vs Hardware Multithreading

- Hardware multithreading → Better Utilization
  - ≈1% more hardware, ≈10% better performance
  - Share integer adders, floating-point units, caches, memory controller
- Multiprocessor/multicore → Duplicate Processors
  - ≈50% more hardware, ≈2x better performance
  - Share memory controller, caches not integrated in core.

Big/Little Processors

- You need "big" processors for high performance
  - “Big”: Higher frequency, more superscalar pipelines, larger caches
- But such processors are inefficient
  - Lots of power, lots of silicon
  - And a lot of time you don't need a big processor, because you don't need the performance
- Modern big/little design
  - Intel Raptor Lake i9: 8 performance cores (with 2 threads/core) + 16 efficiency cores
    - Efficiency cores only support one thread/core, and are designed in a block of 4 with a shared cache
  - Qualcomm Snapdragon 8 Gen 2: 1 big core, 4 medium cores, 3 little cores
  - Apple M1 Pro: 8 performance cores, 2 efficiency cores
    - All cores are single thread/core.

Thread-Level Parallelism (TLP)

Terminology

- A **program** is a sequence of instructions to run (such as an executable)
- A **process** is the actual execution of a program. Each process is a largely separate entity, with its own memory space
- Each process is composed of **threads**, which are independently running instruction sequences that share most memory
- The datapath we've discussed so far is a CPU **core**. A single core can run one thread at any given time
- The CPU is composed of multiple cores, which thus allows multiple threads to be run simultaneously.

Terminology

- A **single-threaded program** is a program that only runs one thread
  - All the programs we've discussed so far are single-threaded
- **Multi-threaded** and **multi-process** programs are programs that use multiple threads or processes
- The **Operating System**, or OS, is responsible for managing which threads get run on which cores (among other tasks).

Recall: Iron Law and Parallelism

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

- Instructions/Program
  - At some point, we will have reduced our program down to the basic operations that must be carried out.
  - Work per instruction can be increased with SIMD, but that's at most an 8x speedup assuming 256-bit vectors
- Cycles/Instruction
  - Clever ordering of instructions can help, but pipelining already optimizes this fairly well.
  - Memory instructions have high cycles/instruction, so we can improve this via caches.
- Time/Cycle
  - Up until recently, done by improving manufacturing to increase frequency (doubled every ~2 years according to Moore's Law).
  - Moore's Law has slowed down!
- Need to parallelize
  - Increase the work that gets done per instruction, or
  - Increase the number of processors that get used by a single program.

Parallel Computing

- Instead of just one thread, let's write a program that runs with multiple instruction sequences simultaneously
- Why?
  - Our computer has N cores, so we can use them all
  - If a thread is waiting on memory accesses, might as well set up another thread to keep working
  - Computers can have up to hundreds or thousands (or millions!) of cores, so if we want to write code that runs as fast as possible, we need to parallelize
- Two main choices:
  - Increase the number of threads per process: Today
  - Increase the number of processes for a program: Monday.

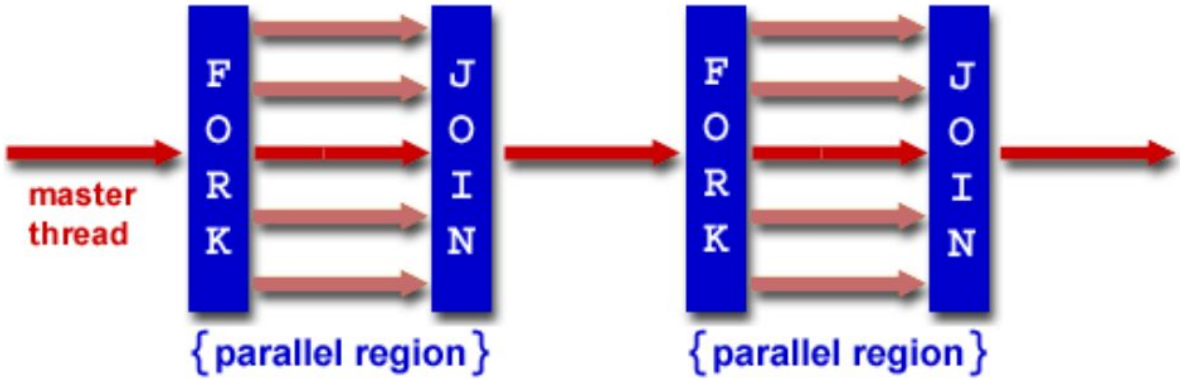
Threads vs Processes

- Threads: Different instruction sequences on the same process
  - Threads on the same process share memory
  - "Easy" to communicate
  - Limited to a set of cores wired to the same memory block (1 node)
- Processes: Largely independent from each other
  - Different processes can't share memory
  - "Difficult" and time consuming to communicate
  - Can expand to as many cores as you have available, over as many nodes as you want
- Example: Fugaku has 158,976 nodes, each with 48 cores
  - With threads: use up to 48 cores
  - With processes: use up to 158,976 x 48 = 7,630,848 cores
- More info: CS 162, CS 267.

Multithreading Framework Overview

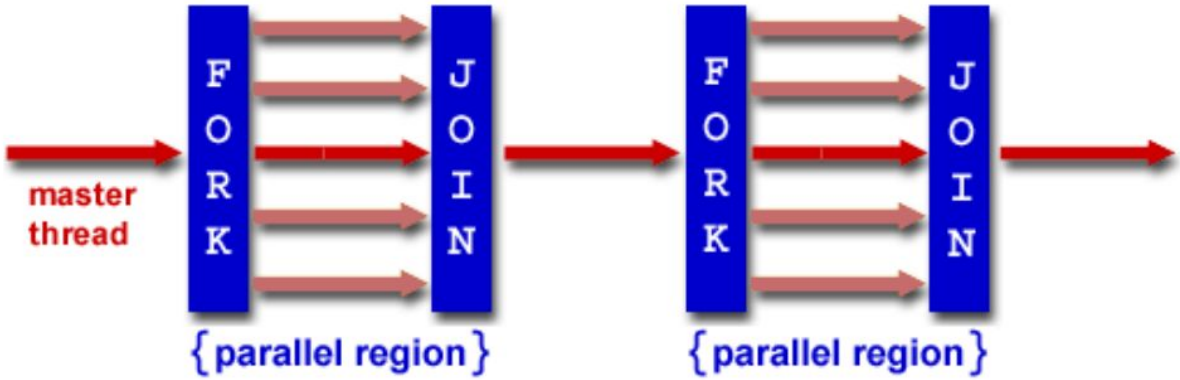
- Each thread has its own:
  - Registers
  - PC
  - Stack
- Threads of same process share:
  - Heap
- Communication is done through shared memory
- Threads run simultaneously, so we have no control over the order in which threads do their work.

Multithreading: Fork-Join Model



- Many different multithreading models, but for this class, we'll consider the **fork-join model**
- Program starts serial
- **Fork:** Master thread creates multiple threads for a segment. Divide the work to all threads
- **Join:** Wait until all threads finish their work, synchronize, and terminate all but the master thread
- Fork and Join take a while (on the order of a few thousand memory operations), so goal is to minimize the number of forks/joins, and minimize the serial parts.

Multithreading: Fork-Join Model in Human Terms



- Let's say I want to make a big mural
- Step 1: I plan out things on paper, set up the wall for painting, etc.
- Step 2: Find a bunch of other people to help me paint (takes some time)
- Step 3: Assign each person some chunk of the wall to paint
- Step 4: Wait until the last person finishes painting their section
- Step 5: I do some cleanup, last minute checks, etc.

Load Balancing

- How do we utilize threads to get the fastest runtime?
  - Main problems:
    - Minimize the nonparallelizable portion (i.e. Minimize the time spent doing solo work, and overhead in finding people)
    - **Balance the load**
- We're limited by the person who takes the most time to finish
- Not everyone paints at the same speed, some parts of the mural might have more detail than others and therefore take longer to paint
- Often impossible to perfectly load balance, so we have to make do with "close enough" and "statistically, everyone should have about the same amount of work".

Strong and Weak Scaling

- Two different ways to measure scaling:
- **Strong scaling**: If I double the number of people working, how much faster does the problem get (ideally close to 2x faster)?
    - Measured w/ Amdahl's Law
- **Weak scaling**: If I double the number of people working and work for the same amount of time, how much more work can I do (ideally close to 2x)?
    - A bit easier to get, but doing more work has to be useful!

OpenMP

OpenMP

- OpenMP is an extension of C used for multi-threaded code (shared memory, so no multi-node computation)
- Compiled with the additional flag "gcc -fopenmp foo.c"
  - "#include <omp.h>"
- Generally follows the fork-join framework
- Each thread has its own stack for private variables, but otherwise shares memory with other threads
- OpenMP code generally is written using lines like "#pragma omp <command>".

#pragma omp parallel

- In order to create a parallel section:

```
#pragma omp parallel
{
    //parallel code
}
```

- (brackets needed if more than 1 line of code)
- The code in the parallel section gets run on all threads, so we need some way to distinguish threads
- In a parallel segment:
    - `omp_get_num_threads()` returns the number of threads running
    - `omp_get_thread_num()` returns a unique number from 0-num_threads per thread

Parallel Hello World (Demo)

```
#include <stdio.h>
#include <omp.h>
int main () {
    int x = 0; //Shared variable
     #pragma omp parallel
     {
          int tid = omp_get_thread_num(); //Private variable
          x++;
          printf("Hello World from thread %d, x = %d\n", tid, x);
          if(tid==0) {
               printf("Number of threads = %d\n", omp_get_num_threads());
          }
     }
     printf("Done with parallel segment\n");
}
```

Shared vs Private variables

- By default, any variable declared outside the parallel segment is shared: all threads write/read from the same variable
- Any variable declared inside the parallel segment is private: Each thread has its own version of the variable
- Can overrule this with the private and shared keywords:

```
#pragma omp parallel private(var) shared(var2)
```

- Note that heap memory is always shared, though we can have private pointers and private mallocs (if we do the malloc in the parallel segment, and free them within the parallel segment).

For loops

- Problem: You have to do some work over an array of 1 million numbers, with 4 threads. How do you split the work?
- Assumptions:
  - We need to decide this before we run the code
  - Each element of the array is independent, so we can do this in any order
  - The threads are about equally fast at work, so we want to assign each of them 250k numbers.

For loops

- Option 1: Do every 4
  - `for(int i = tid; i<1000000;i+=4)`
  - "Interweaving"
- Option 2: Thread 0 does 0-249999, 1 does 250000-499999, etc.
  - `for(int i = tid*250000;i<(tid+1)*250000;i++)`
  - "Blocking"
- Which one's better?
  - With standard multithreading, Option 1 actually is as slow as the serial version of this code due to cache coherency issues… More on this when we cover caching.
  - Option 2 speeds things up correctly.

For loops

- Option 3: Let the compiler do it for you with the `#pragma omp for` keyword.
- `#pragma omp for` must be written inside an already existing parallel segment
- If a parallel segment consists only of one for loop, we can combine the two declarations with `#pragma omp parallel for`

```
#pragma omp parallel for
for(int i = 0; i<1000000;i++) { … }
```

Data Races and Critical Sections

Data Races

- Note that when we ran Hello World parallel, we ended up with the threads running in random order
  - In fact, every time we run Hello World, we get a different order!
  - The x values stayed largely in-order, but didn't always strictly increase
- Recall: OS choose whichever threads it wants to run, and can change threads at any time
- One big downside to multithreading: program no longer deterministic, has random behavior
- Formally, a multithreaded program is only considered correct if ANY interlacing of threads yield the same result.

Data Race: Example

- If we run this code on 4 threads, what possible values could x have at the end?

```
int x = 0;
#pragma omp parallel {
   x = x + 1;
}
```

Data Race: Example

- To analyze this, we need to see the equivalent assembly code (let's use RISC-V)
- Only the loads and stores affect shared memory, so only need to consider different ways we can order loads and stores
- How many possible orders of loads and stores?
  - $8!/(2!)^4 = 2520$ different possible orders
  - Can use the fact that all the threads are identical to reduce this to 105 orders, but still too many to check manually.

```
                    sw x0 0(sp)
lw t0 0(sp)     lw t0 0(sp)     lw t0 0(sp)     lw t0 0(sp)
addi t0 t0 1    addi t0 t0 1    addi t0 t0 1    addi t0 t0 1
sw t0 0(sp)     sw t0 0(sp)     sw t0 0(sp)     sw t0 0(sp)
```

Data Race: Example

- Case 1: All the threads run one at a time
  - Purple thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread reads x = 1
  - Brown thread stores x = 2
  - Red thread reads x = 2
  - Red thread stores x = 3
  - Blue thread reads x = 3
  - Blue thread stores x = 4
- Final value: 4

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
```

Data Race: Example

- Case 2: The threads are perfectly interleaved
  - Purple thread reads x = 0
  - Red thread reads x = 0
  - Brown thread reads x = 0
  - Blue thread reads x = 0
  - Purple thread stores x = 1
  - Brown thread stores x = 1
  - Red thread stores x = 1
  - Blue thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Case 3: Same as case 1, except purple's store happens last
  - Purple thread reads x = 0
  - Brown thread reads x = 0
  - Brown thread stores x = 1
  - Red thread reads x = 1
  - Red thread stores x = 2
  - Blue thread reads x = 2
  - Blue thread stores x = 3
  - Purple thread stores x = 1
- Final value: 1

```
sw x0 0(sp)
lw t0 0(sp)
addi t0 t0 1
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Data Race: Example

- Some other ordering?
  - We can find orderings that give x=2,3
- Can we do any more/less?
- Can't go above 4
  - Only 4 "+1s" overall, so can't increase to 5 or more
- Can't go below 1
  - The smallest value that can be loaded by a thread is 0, so the smallest value that can be stored is 1. Therefore, the last store must be at least 1.
- Therefore, we can get any value between 1 and 4

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
sw t0 0(sp)
```

```
sw x0 0(sp)
lw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
lw t0 0(sp)
addi t0 t0 1
sw t0 0(sp)
sw t0 0(sp)
```

Critical sections

- Fortunately, OpenMP gives you some commands that let you use critical segments completely safely
- `#pragma omp barrier`
  - Forces all threads to wait until all threads have hit the barrier
- `#pragma omp critical`
  - Creates a critical segment in parallel code; only one thread can run a critical segment at a time.

Parallel Hello World with Critical Segments

```
#include <stdio.h>
#include <omp.h>
int main () {
    int x = 0; //Shared variable
     #pragma omp parallel
     {
          int tid = omp_get_thread_num(); //Private variable
          #pragma omp critical
          {
               x++;
               printf("Hello World from thread %d, x = %d\n", tid, x);
          }
          #pragma omp barrier
          if(tid==0) {
               printf("Number of threads = %d\n", omp_get_num_threads());
          }
     }
     printf("Done with parallel segment\n");
}
```

Avoiding Data Races

- Formally, a multithreaded program is only considered correct if ANY interleaving of threads yield the same result.
- Ideally, try to have each thread works on independent data (no two threads write to the same value, or read a value that another thread wrote to)
- If coordination is necessary, critical sections are one important tool
  - CS 162: Locks, atomic operations
- The hardest part of multithreading: maintaining correctness while also speeding up the code.

Summary

- **Flynn's Taxonomy** classifies the different ways parallel hardware can be designed
- **Software threads** get scheduled on **hardware threads** by the OS and can be spawned by a process to utilize **thread-level parallelism (TLP)**
- **OpenMP** is a library that allows you to write TLP code in C and other languages
- Spawning threads can make your program non-deterministic; need to handle issues like **data races**
- Monday: **Process-level parallelism (PLP)**