Code of the Day

- Count the number of set bits (i.e., “1”) in an integer v

```
int c;
int v; // must >= 0
for (c=0; v; v>>=1) {
    c += (v & 1);
}
```

Code of the Day

- Count the number of set bits (i.e., “1”) in an integer v

```
int c;
int v; // must >= 0
for (c=0; v; v>>=1) {
    c += (v & 1);
}
```

```
int c;
int v;
for (c=0; v; v -= (v & -v)) {
    c++;
}
```

An asymptotically optimal solution

- Why? `(v & -v)` isolates the lowest set bit in v
  - `6 = 110`
  - `-6 = 010` (in 2’s complement: complement and plus 1 – ECE/CS 252)

ECE 455

GPU Algorithm and System Design

[Fall 2025]

Why Parallel Computing?

2025/09/08

Before we get started…

- Purpose of today's lecture
  - Why do we need to do parallel computing?
  - Why parallel computing is a "must" for scaling performance with future computer architectures?

- Miscellaneous
  - We will have our first lab starting this Friday
  - Next Monday's lecture (9/15) will be **on zoom 4:30 – 6:00 PM** (delayed by 1 hour) due to TW's travel
  - Next Friday's lab will be managed by TA due to TW's travel

Parallel computing: why, and why now?

Observation: Sequential Computing Makes only Incremental Gains

- Over the last decade, sequential computing execution speed has largely saturated and made only incremental performance gains, primarily due to the following three limitations:
  - Memory wall
  - Instruction level parallelism (ILP) wall
  - Power wall
- For many applications, parallel computing is where speed improvements have come from

Limitation #1: Memory Wall

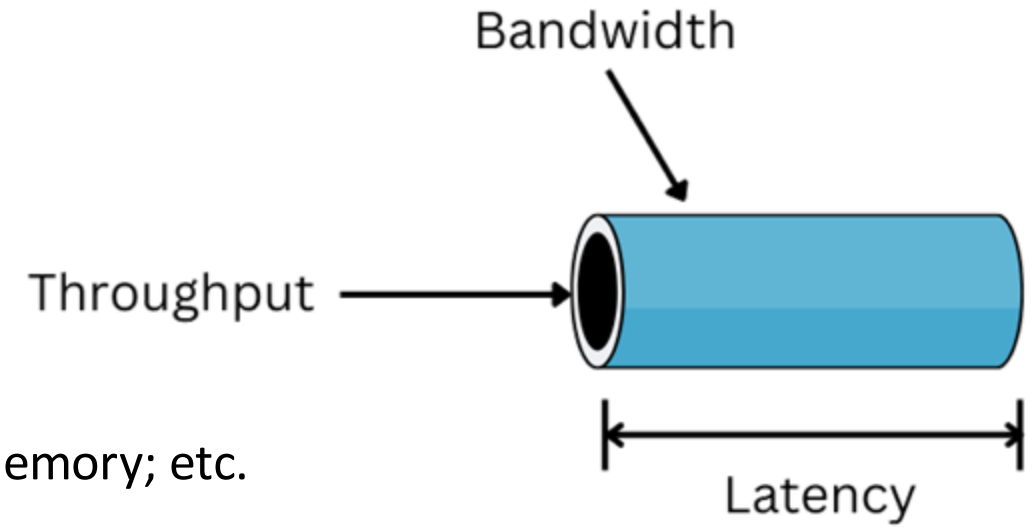
- A fact that the speed gap between CPU and off-chip memory continues to enlarge
  - Processors can perform calculations much quicker than the memory they interact with
    - Ex: CPU can perform billions of operations per second while access data from memory takes hundreds of cycles (10-100x slower)

- Memory accesses have gradually become a barrier to computer performance improvements
  - We can try to grow caches to improve avg memory reference time of data (if programs have good locality)
  - However, cache eats up transistors quickly (i.e., cache is made of SRAM, 6 transistors per bit)
  - Also, as a cache gets gradually larger, it also gets gradually slower (trade-off)

- Ultimately, it is due to latency & limited communication bandwidth beyond chip boundaries
  - The impact of memory hierarchy on performance gets bigger and bigger

Memory Latency vs. Memory Bandwidth

- Memory latency: the time it takes for a piece of data to get from one location to another
  - Metaphor: how long should I wait to get my coffee?
- Memory bandwidth: the amount of data that can be transferred per second
  - Metaphor: how many coffees can be served simultaneously by the front desk of the coffee shop?



- Improving Latency and Bandwidth
  - Promising technology: optic networks; on-package memory; etc.

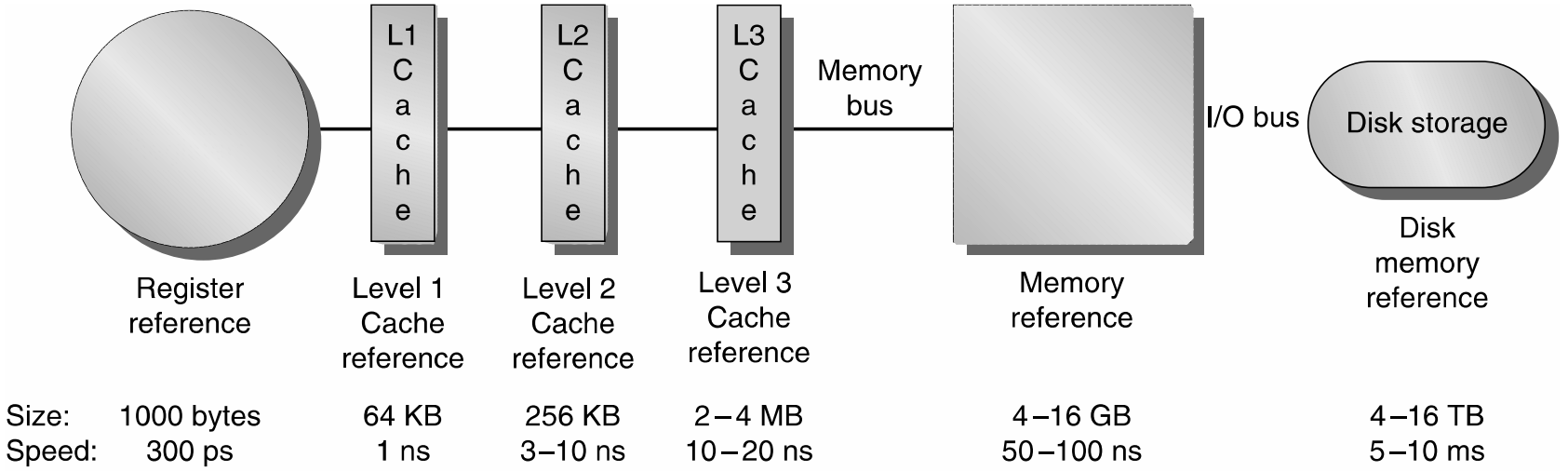
Improving Memory Latency and Bandwidth

- **Memory Latency is more challenging to improve than Memory Bandwidth**
  - Principle of physics – Latency cannot go faster than the speed of light ☺
  - Physical constraints – Latency is constrained by the physical timing of signal delivery (e.g., Clock rate), but bandwidth can be improved by adding more buses to parallelize data transfers and enlarge the bandwidth

- Analogy:
  - If you carry commuters with a train, adding more cars to the train will increase its bandwidth
  - Improving latency requires the construction of high-speed trains, which is more expensive and complex

Takeaway from the Memory Wall

- Try to stay away from long and winding conversations with main memory
  - Cache (L1, L2, L3) is accessed with 1—20 cycles
  - Main memory is accessed with 50—100 cycles



Memory Access Patterns	
To/From Registe	Golden
To/From Cache	Superior
To/From RAM	Getting dicey
To/From Disk	Sad day…

- In practice, you may not always be at the top speed for your memory
  - Your program will bring a block of data, use some of it, and ask for data from a different block as the computation proceeds – *effective bandwidth is typically much smaller than nominal bandwidth*

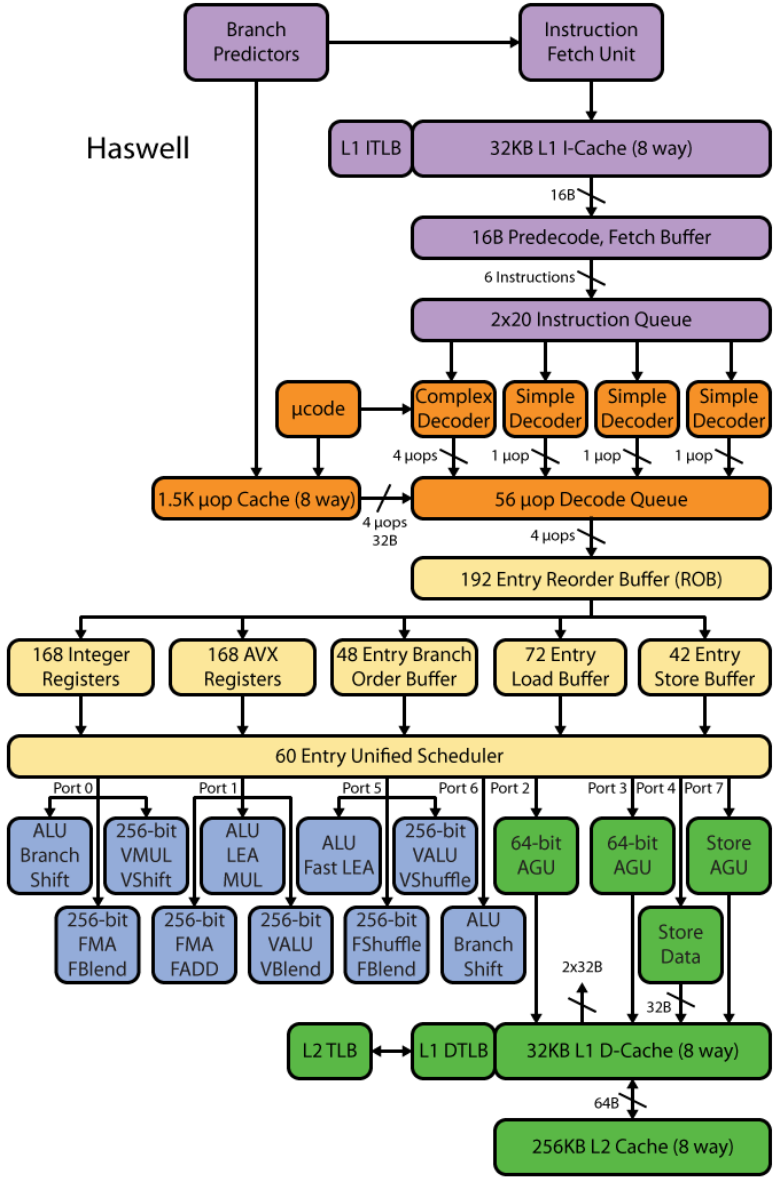
Limitation #2: Instruction Level Parallelism (ILP)

Note: ILP refers to the ability of a processor executing multiple instructions simultaneously within a cycle

- Instruction pipelining: execution of multiple instructions is overlapped across stages
- Superscalar execution: multiple execution units are used to execute multiple instructions
- Out-of-order execution: instructions run in any order w/o breaking data dependencies
- Register renaming: eliminate unnecessary data dependencies between instructions
- Speculative execution & branch prediction: execute instruction ahead of time, helping the CPU guess which direction a branch (e.g., `if` statement) will go to minimize waiting time

ILP Relies on Very Complicated Microarchitecture …

- Microarchitecture components (Intel Haswell):
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Can be regarded as CISC to RISC step
  - **Instruction Scheduling support (yellowish)**
  - Instruction execution
    - Arithmetic (blueish)
    - Memory related (green)



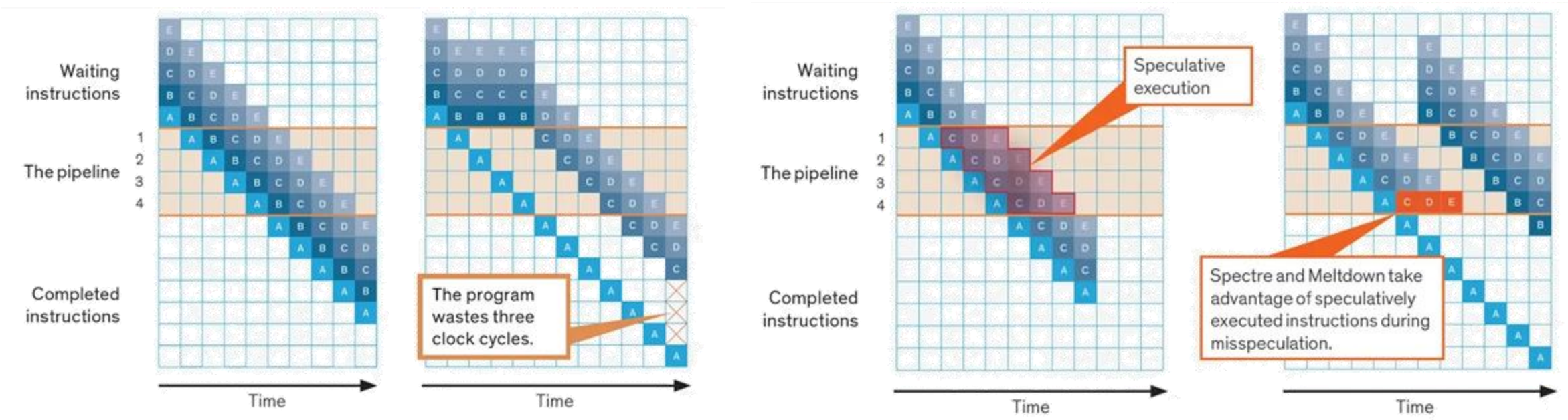
[http://www.realworldtech.com/haswell-cpu/]→

Takeaway from The ILP Wall

- ILP is great but relies on expensive and complicated hardware
  - Ex (speculative execution): Dedicated hardware speculatively executes future instructions before the results of current instructions are known, while providing hardware safeguards to prevent the errors that might be caused by running instructions ahead of time
    - Predicting the future is always very challenging…, and it comes at the cost of microarchitecture complexity and power cost
  - Ex (branch prediction): Branches must be "guessed" to decide what instructions to execute simultaneously
    - If you guessed wrong, you throw away that part of the result – *wasting time*!
- Ultimately, data dependencies may prevent successive instructions from running in parallel
  - Unfortunately, this type of data dependency constraint typically comes from applications
  - Hardware normally has no idea about how your application will manage data

Aggressive Speculative Execution can Cause Security Problems …

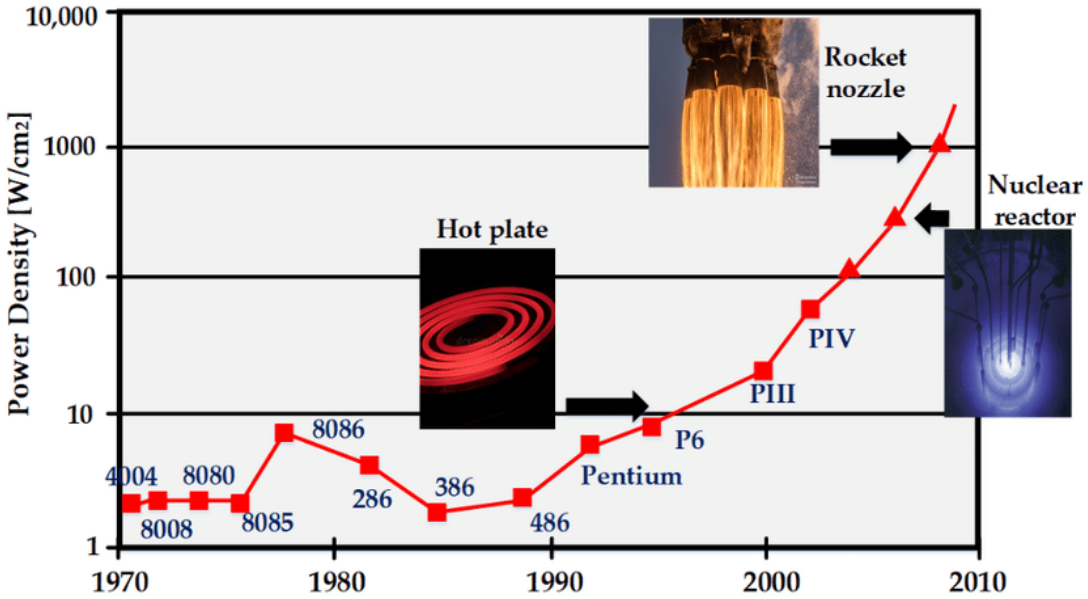
- Allows the CPU to execute potentially sensitive instructions that could leave traces in the cache, enabling side-channel attacks like Spectre and Meltdown to exploit this speculative data for unauthorized access
  - Arbitrary locations in the allocated memory of a program can be read
    - Ex: Spectre (Intel, Apple, ARM, AMD) – everything before 2019 that uses branch prediction



More details: https://spectrum.ieee.org/how-the-spectre-and-meltdown-hacks-really-worked

Limitation #2: The Power Wall

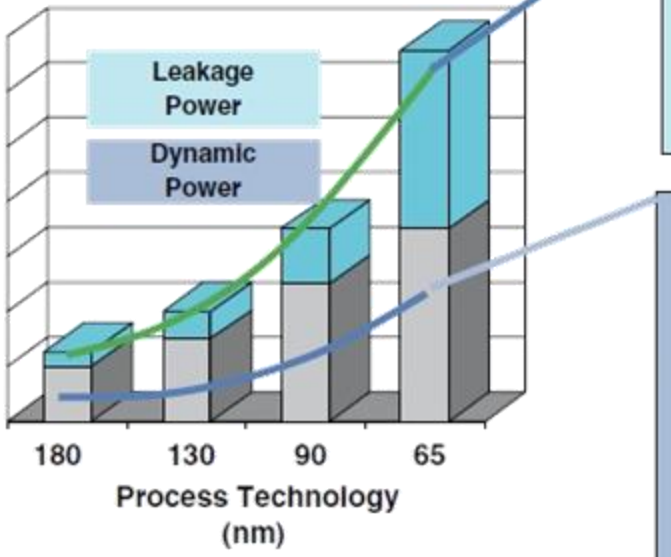
- The faster you want your hardware to go, the more power it will take
  - Speed of your hardware and its power dissipation re both related to the clock frequency -- the faster the clock, the higher the power, due to more frequent switches (in-between 1 and 0)
  - Significant increase in clock frequency is not possible; chips would just simply melt



- Consequence: clock rates have largely saturated at certain values

Power Leakage is Another Major Source to Power Wall

- Power leakage is unwanted and continuous consumption of electrical power in a device
  - Due to leakage currents in transistors, which happen when the transistors are supposed to be in an "off" state but still allow a small amount of current to flow
  - Power leakage becomes more significant as transistor sizes shrink, as the insulating layers become thinner which increases the chances of leakage



$P_{leakage} = \sum$ cell leakage

- Summary of library cell leakage
- Can be state-dependent

$P_{dynamic} = P_{internal} + P_{wires}$

$P_{internal} = \sum$ cell dynamic power

$P_{wires} = ½ \times C_L \times V^2 \times TR$

C_L: Capacitive loading (pin and net)

V: Voltage level

TR: Toggle rate

Dennard's Scaling and its End

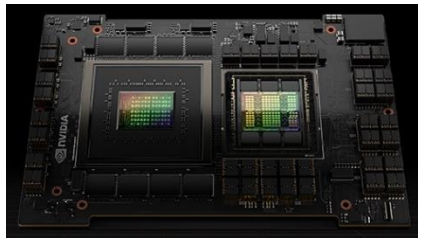
- **Dennard scaling** is a concept in semiconductor electronics that states as transistors get smaller, their power density remains constant
  - Ex: number of transistors increases 4X → each transistor is also improved to use ¼ of power

- **The end of Dennard scaling** refers to the point where the historical trend of decreasing transistor size leading to proportional decreases in power consumption came to a halt
  - Ex: number of transistors increases 4X → each transistor CANNOT be improved to use ¼ of power
    - **Increased leakage current** contributes significantly to power consumption
    - Smaller transistors are more susceptible to leakage, where a small current flows even when the gate voltage is below the threshold voltage

- Takeaway: When transistors became smaller and smaller, it became increasingly difficult to reduce power without compromising performance and reliability

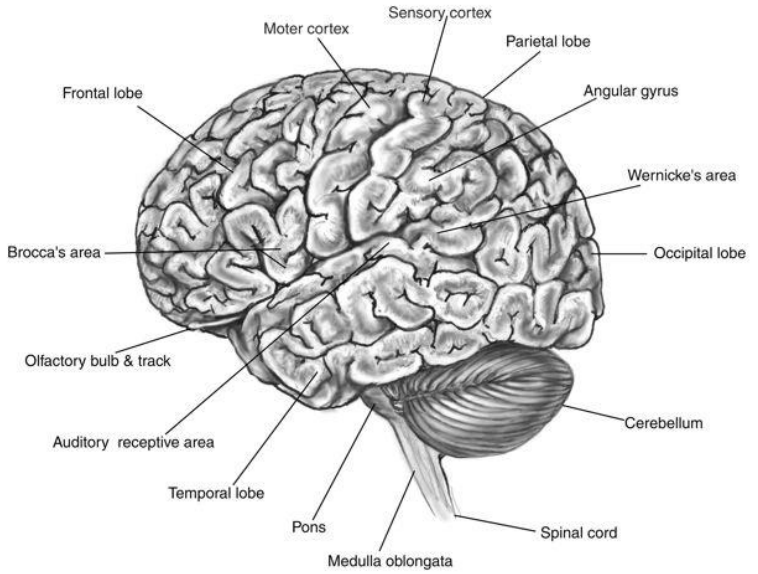
“spending energy” rates vs how much you should eat per day

- NVIDIA Hopper H100
  - Thermal Design Power (TDP): up to 700 Watts



- Intel® Xeon® Processor E7-8890 v3
  - TDP: 165 Watts

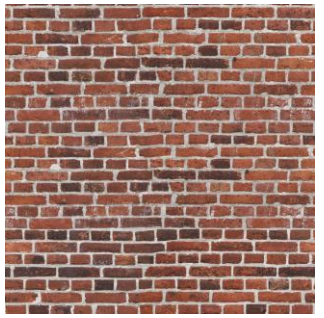
- Human Brain
  - 20 Watts
  - Represents 2% of our mass
  - Burns 20% of all energy in the body at rest
  - Our entire body, ball-park: 100-120 Watts
    - How much should you eat per day?
      - 110 W * 3600 seconds * 24 / (4.184 J per calorie) ≈ 2271.5 Calories/day



Computer Architecture Trend in Face of the Three Walls

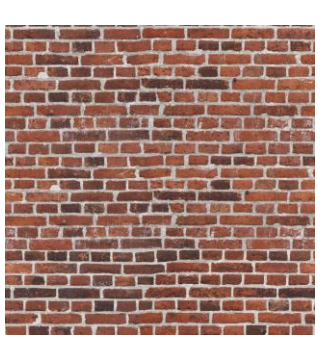
- Old: Transistors are big and not so many; Power is free
- New: Transistors are tiny and massive; Power management very challenging
  (consequences, “lots of power”: heat to be dissipated & battery management)

Power Wall



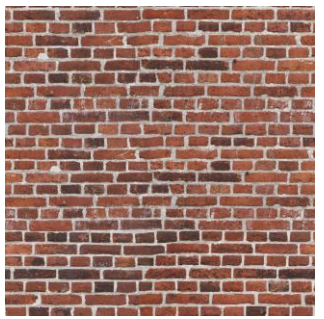
- Old: Processor is slow; Memory access is fast (comparably)
- New: Processor is fast; Memory slow (120 cycles for DRAM memory access, 1 cycle for pipelined FMA)

Memory Wall



- Old: Increasing Instruction Level Parallelism
  hardware and compilers (Out-of-order execution, speculation, …)
- New: ILP sees diminishing returns; pursuit of ILP is not effective

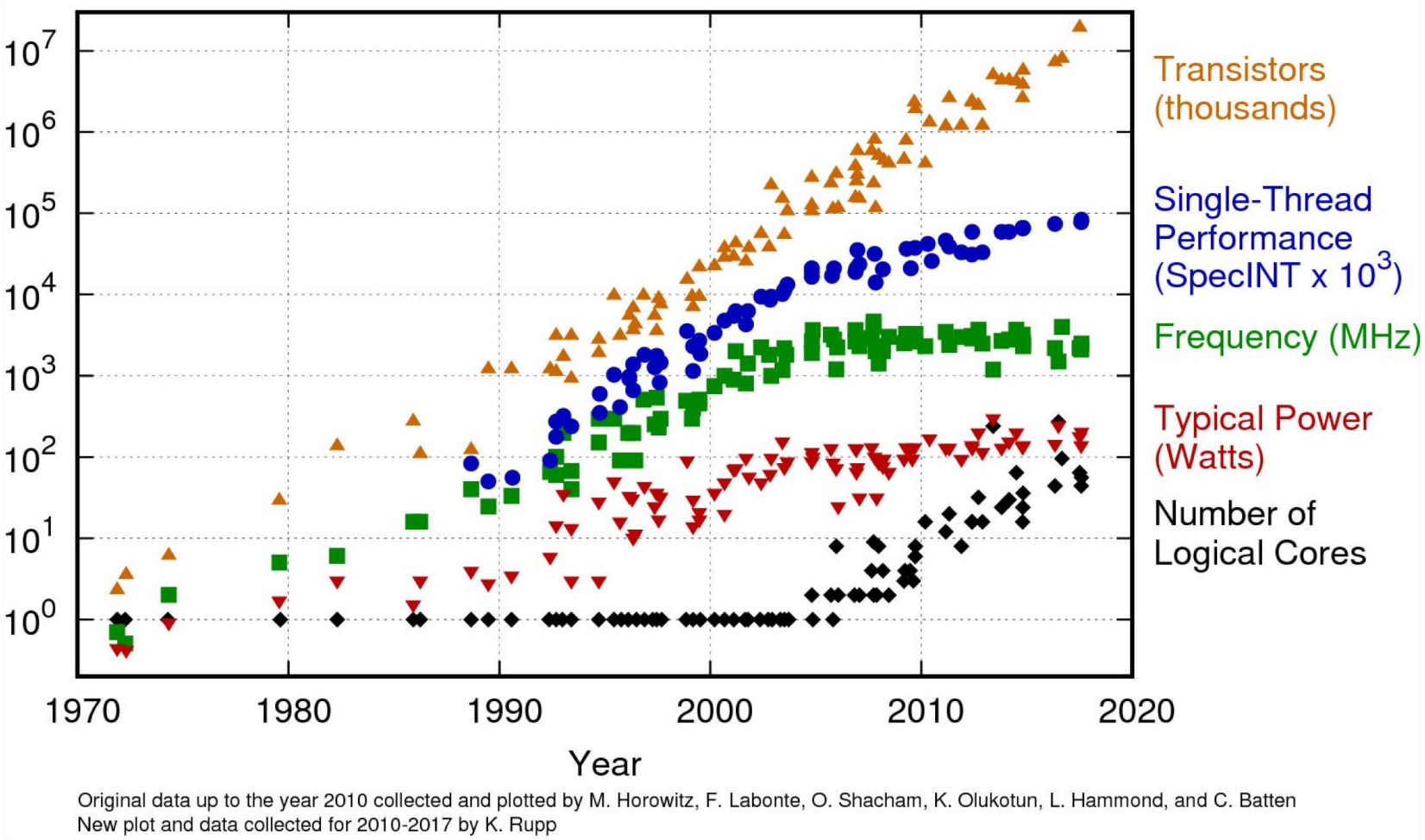
ILP Wall



Good News: Number of Transistors Keeps Going Up!

More transistors are still good:

- Microprocessors
- Memory capacity & speed
- Storage controllers

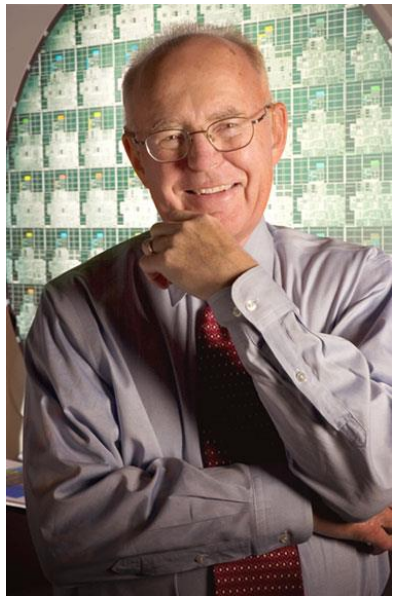


Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

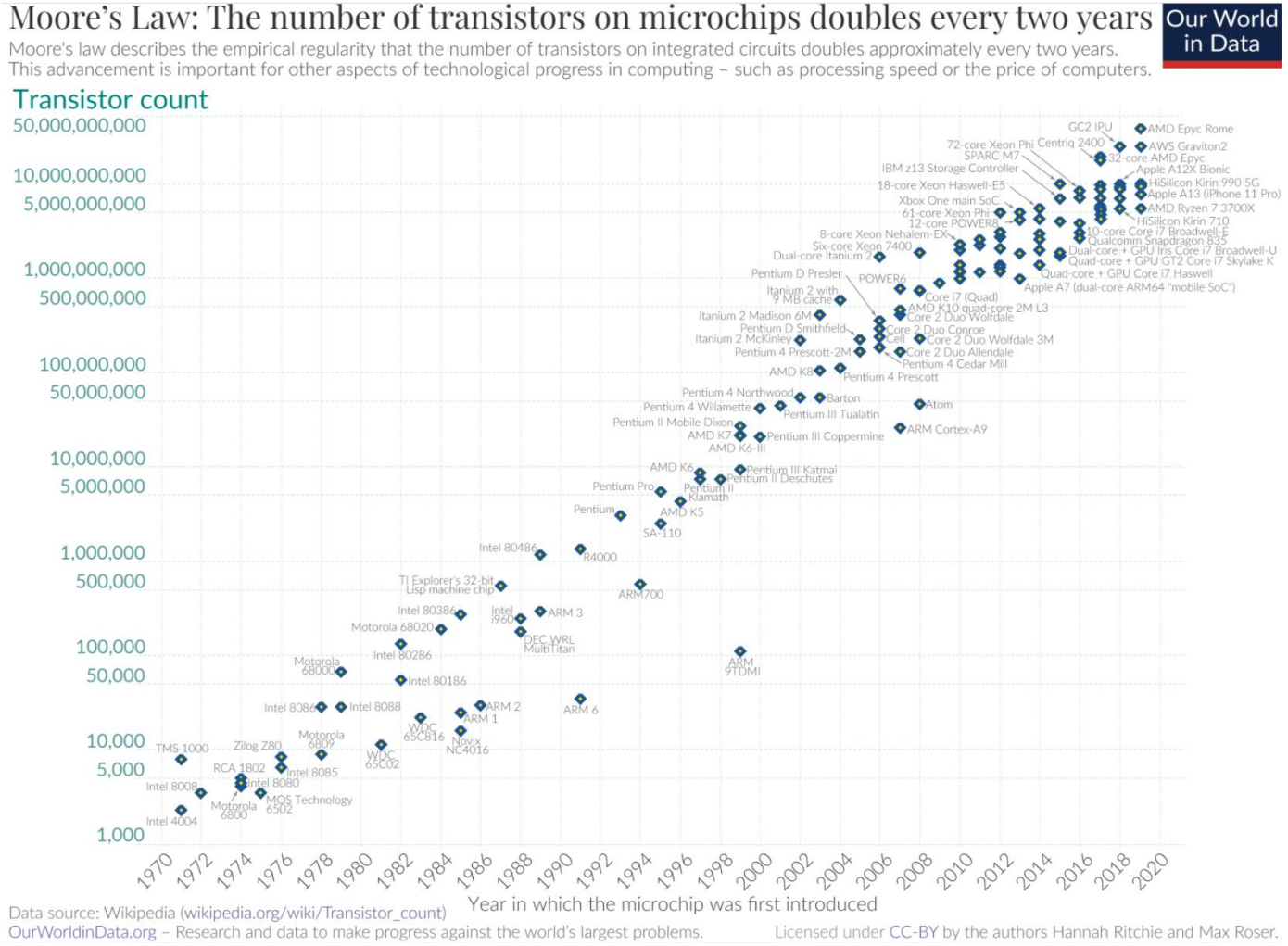
[Kunle Olukotun]→

Moore’s Law

- 1965 paper: Doubling of the number of transistors on IC every two years
- Note: Moore himself wrote only about the density of transistors at minimum cost



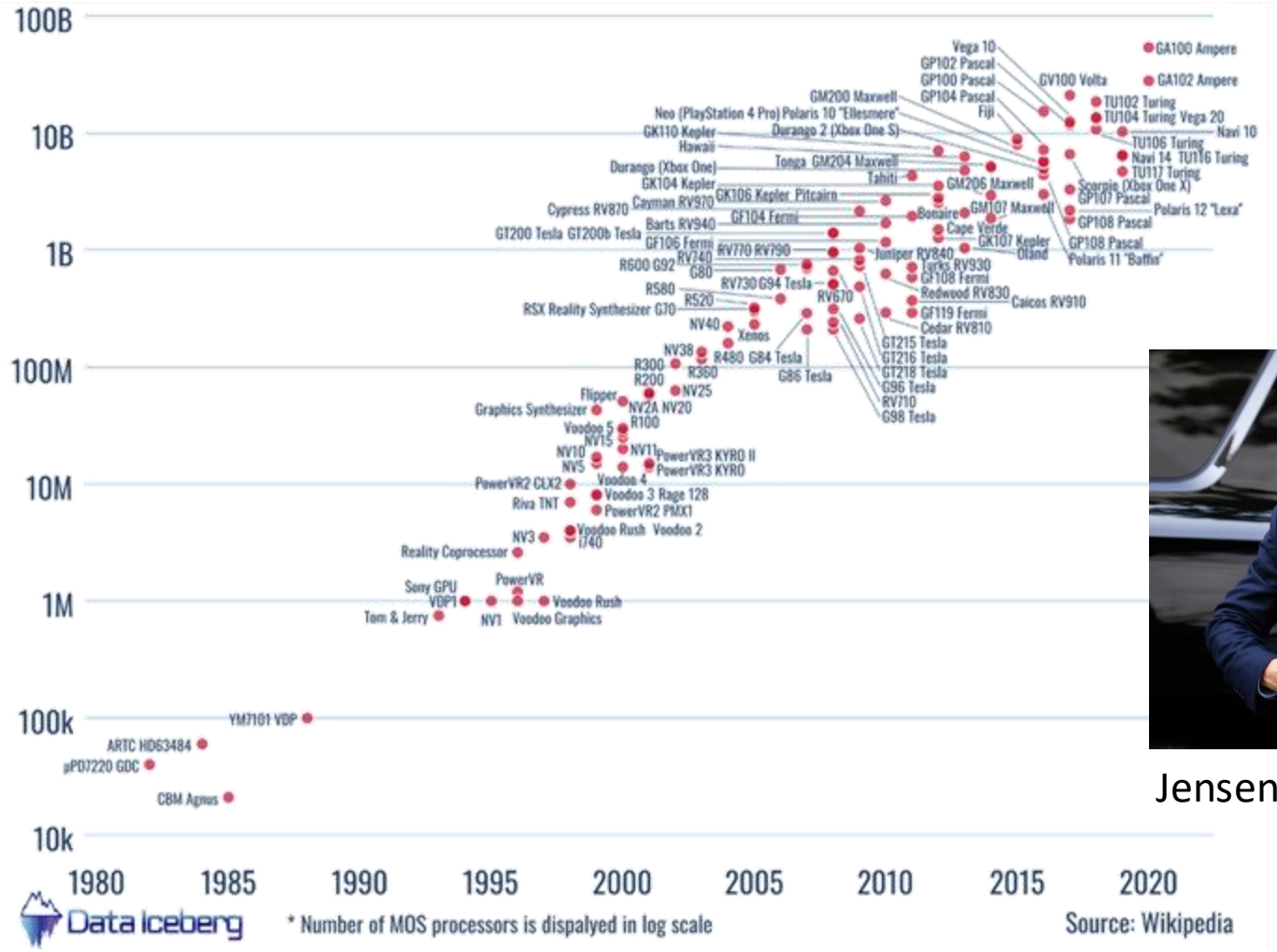
Born: 1929, Pescadero, CA
Died: March 24, 2023, Waimea, HI



http://news.cnet.com/Images-Moores-Law-turns-40/2009-1041_3-5649019.html

In Contrast to Moore's Law: Huang's Law (?)

- The performance of GPUs will double (or even more) every two years

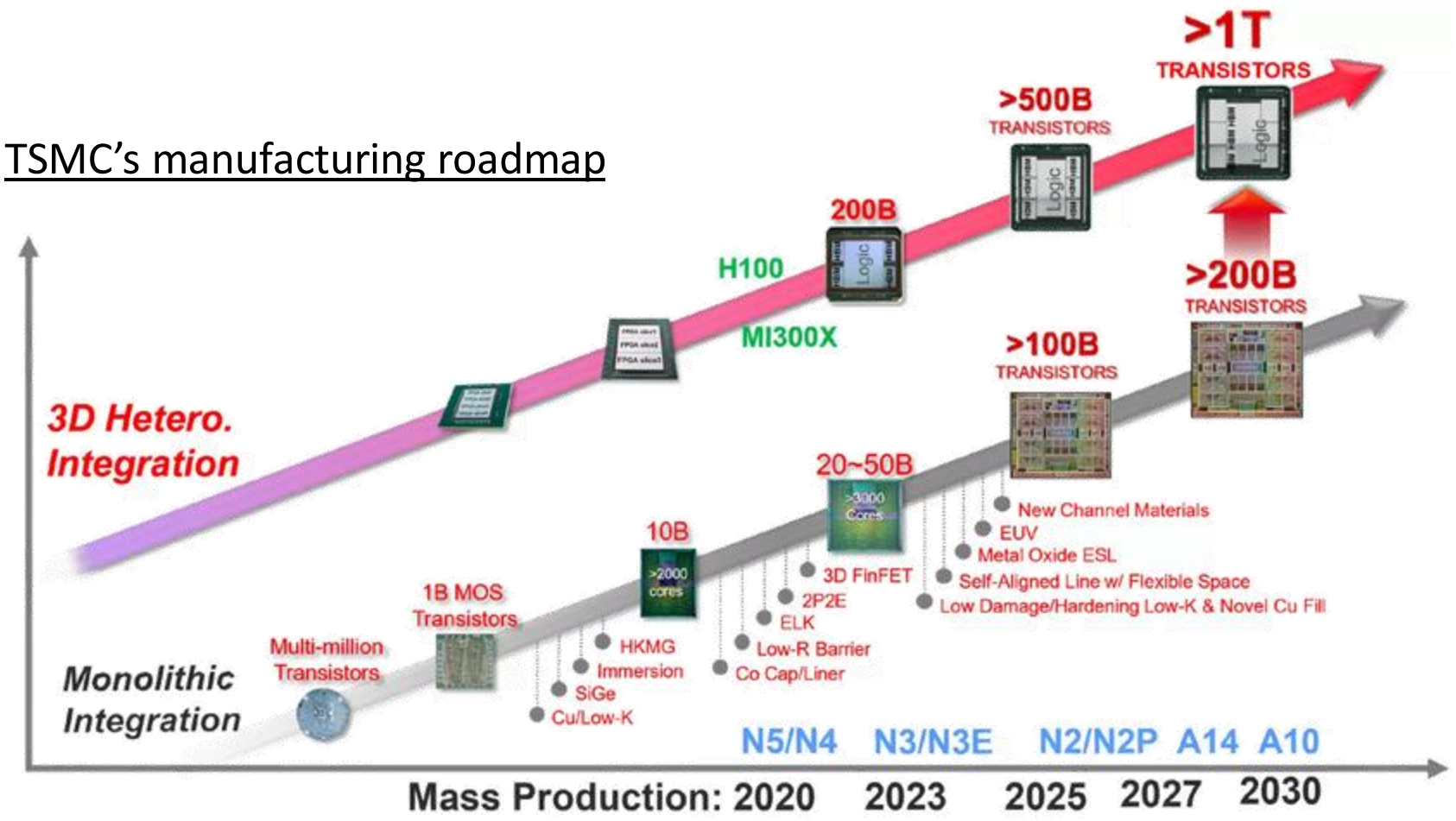


Jensen Huang, CEO of Nvidia

More Transistors – Thanks to Manufacturing Technology

- Advanced manufacturing technology keeps the transistor going smaller and smaller so we can accommodate more transistors in the chip

TSMC's manufacturing roadmap



Shift in Perception to Performance Improvement

- The memory, ILP, and power walls have resulted in a shift in performance improvement

Back in the day…

Increasing clock frequency is primary method of performance improvement

→

Today's take

Processors parallelism is primary method of performance improvement

Yesterday's school of thought

Less than linear scaling for a multiprocessor is failure

→

Today's take

Even sub-linear speedups are beneficial as long as you beat the sequential on a watt-to-watt basis

Parallel Computing Paradigm: Oxen vs. Chickens

If you were plowing a field, would you use two strong oxen or 1024 chickens? – Seymour Cray

- Ox: one powerful CPU core, running at high frequency, large cache to hide memory latency
- Chicken: many small processors, running at lower frequency, not much cache or control unit brain
  - Requires other ways to hide memory latency

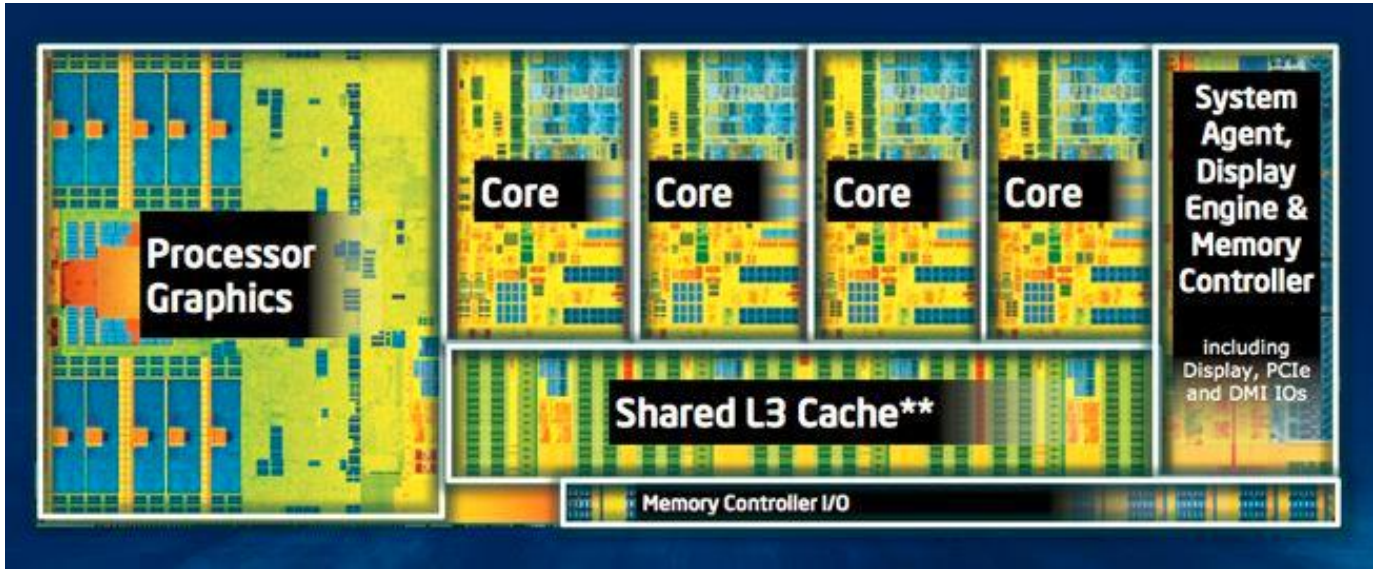
Two Examples of Parallel Hardware

- Intel Haswell (CPU)
  - Multicore architecture – each core is strong at high frequency
  - In our previous figure: two strong oxen

- NVIDIA Fermi (GPU)
  - Large number of scalar processors – each core not so strong
  - In our previous figure: 1024 chickens

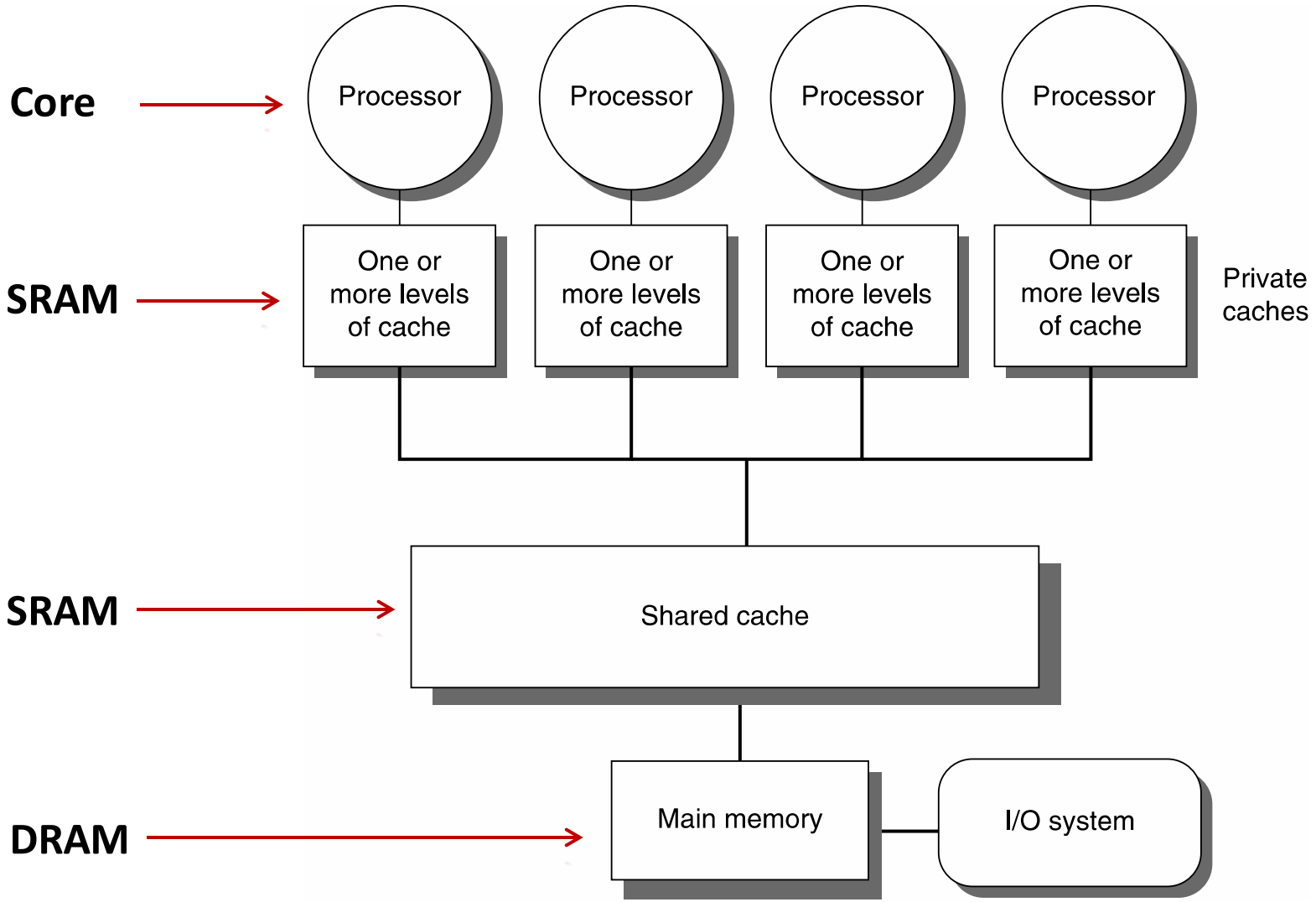
A detailed Look into Intel Haswell

- June 2013
- 22 nm technology
- 1.4 billion transistors
- 4 cores, HTT
- Integrated on-chip GPU
- System-on-a-chip (SoC) design
- Operate at high clock rate (3.5GHz)



128 MB of eDRAM available on Intel Core i7 4950HQ [off-chip but on-package]

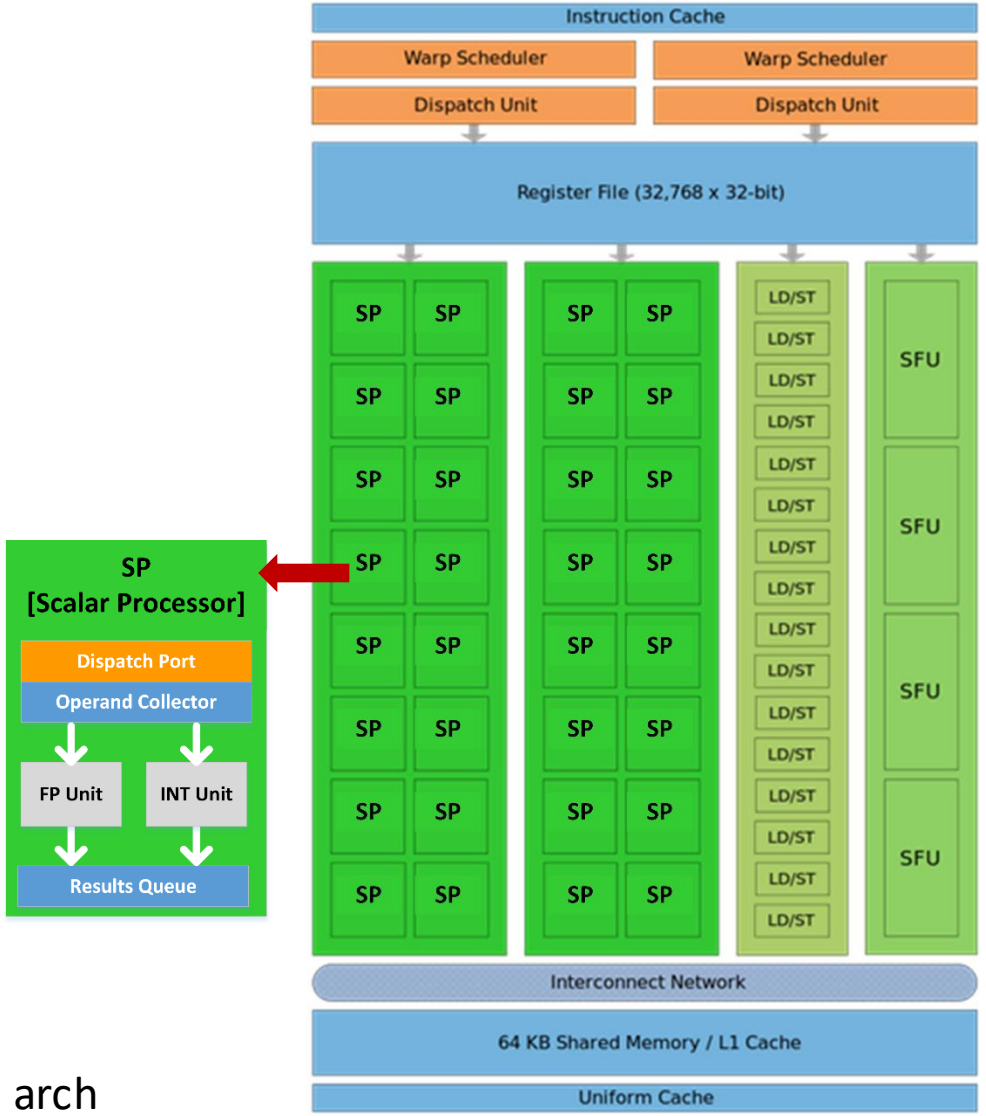
Haswell is Based on Shared-memory Multiprocessor Architecture



Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fifth edition

The Nvidia Fermi GPU Architecture [Stream Multiprocessor (SM) – Core]

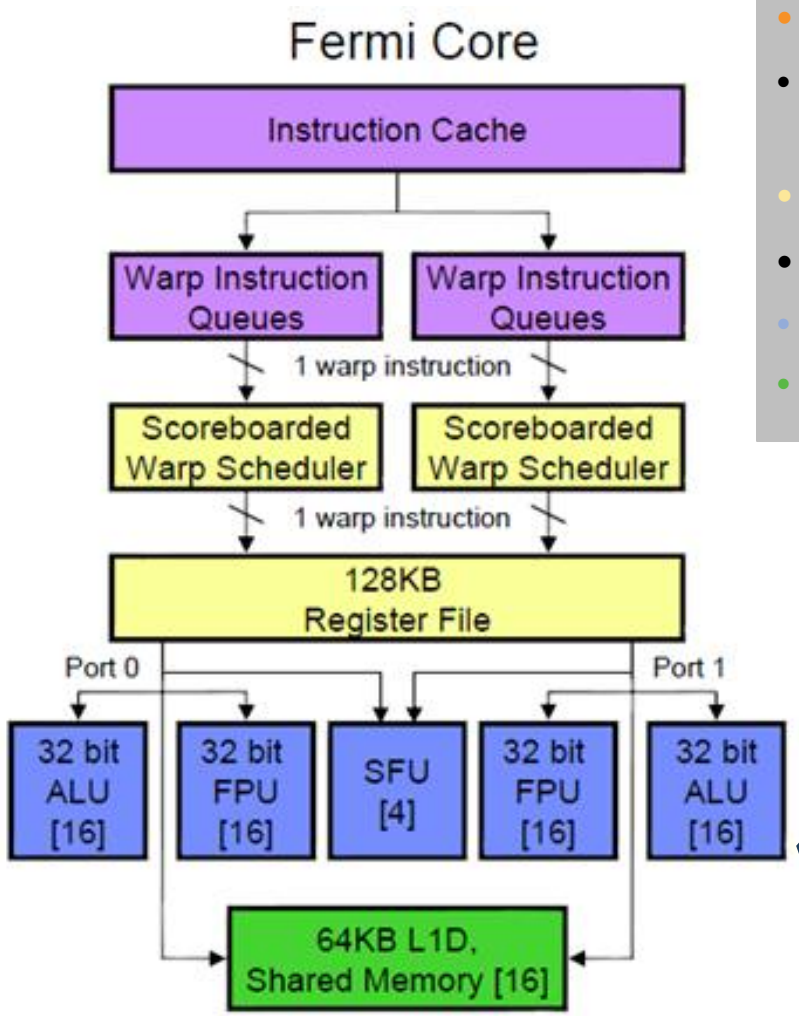
- Late 2009
- 40 nm technology
- Three billion transistors
- 512 Scalar Processors (SPs), or “shaders”
  - These are the chickens
- L1 cache
- L2 cache
- 6 GB of global memory
- Operates at low clock rate (1.4GHz)
- High bandwidth (for 2010, ≈200 GB/s)



Note: An SM is often considered as a GPU core in the context of GPU arch

One SM of Fermi's 16 SMs (SM: Streaming Multiprocessor) vs Haswell

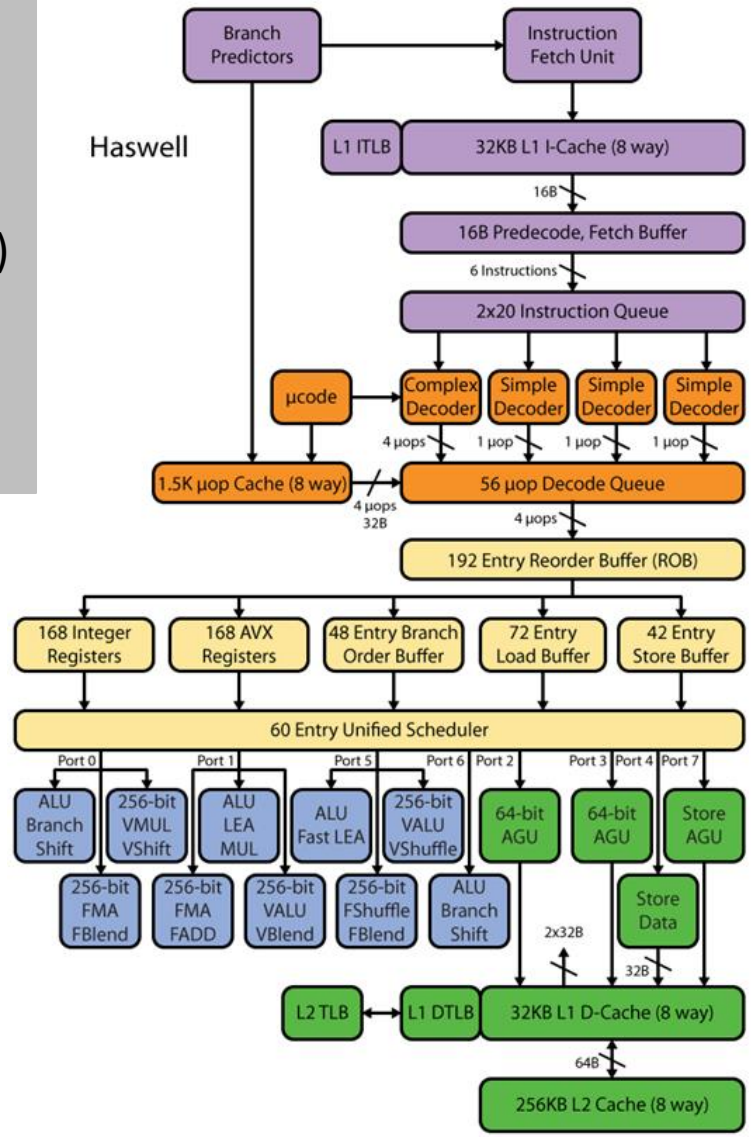
Fermi Stream Multiprocessor



- Instruction pre-fetch support (purple)
- Instruction decoding support (orange)
- CISC into uops
  - Can be regarded as CISC to RISC step
- **Instruction Scheduling support (yellowish)**
- Instruction execution
- Arithmetic (blueish)
- Memory related (green)

Not a lot of brain here in a GPU core (SM), compared to CPU. Specifically, there are no orange boxes, which have a lot to do with ILP.

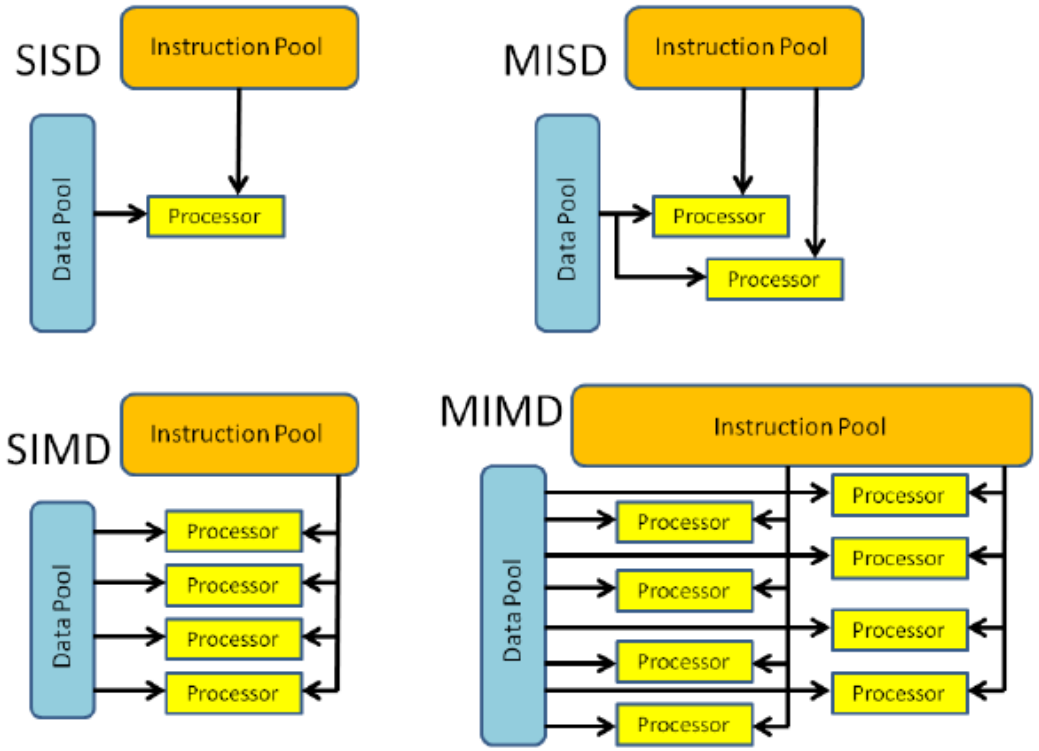
[NOTE: color code is the same for SM & CPU]



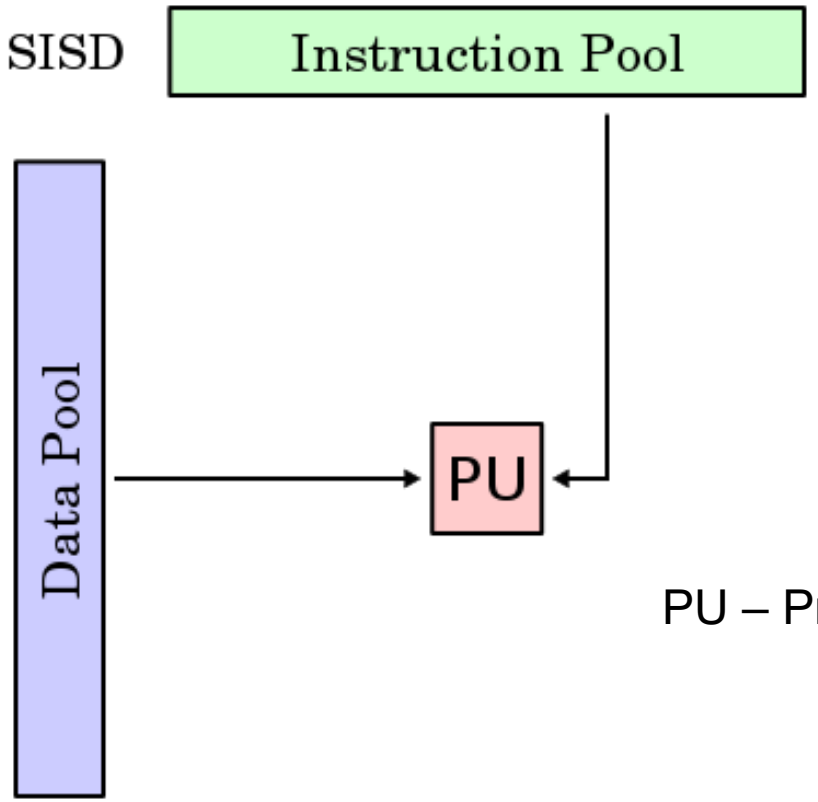
Parallelism of Multi-core CPUs

Flynn's Taxonomy of Parallel Architectures

- A software-centric aspect to classify modern computer architectures
- Flynn's classification is based on how instructions are executed in relation to data
  - SISD - Single Instruction runs Single Data
  - SIMD - Single Instruction runs Multiple Data
  - MISD - Multiple Instructions run Single Data
  - MIMD - Multiple Instructions run Multiple Data



Single Instruction/Single Data (SISD) Architectures

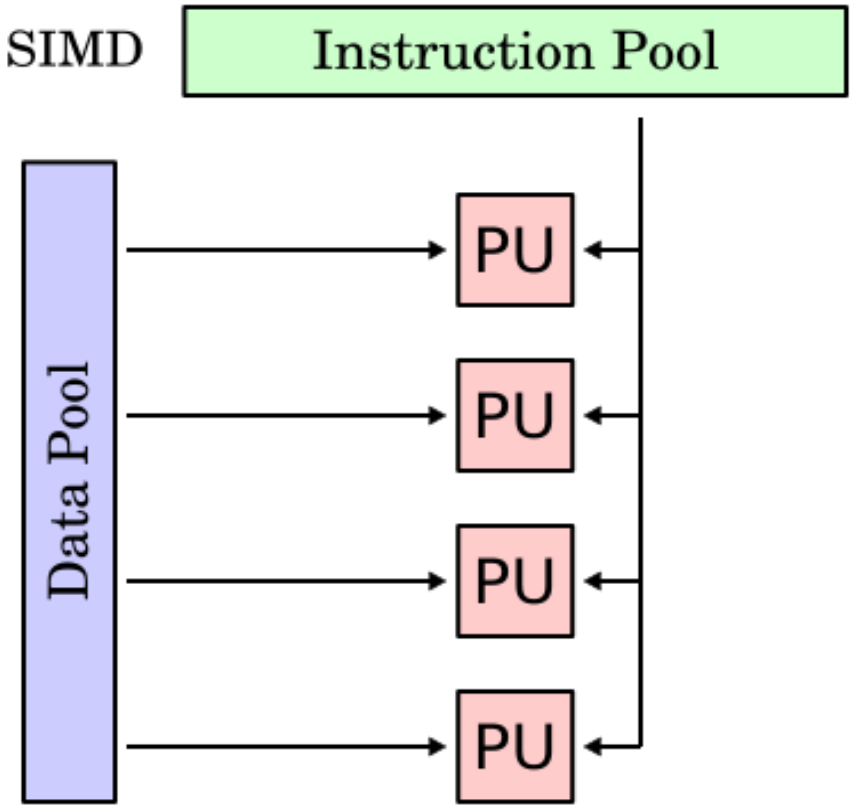


PU – Processing Unit

SISD is the most basic computer architecture in the single-core domain

[Wikipedia, Flynn's Taxonomy]→

Single Instruction/Multiple Data (SIMD) Architectures

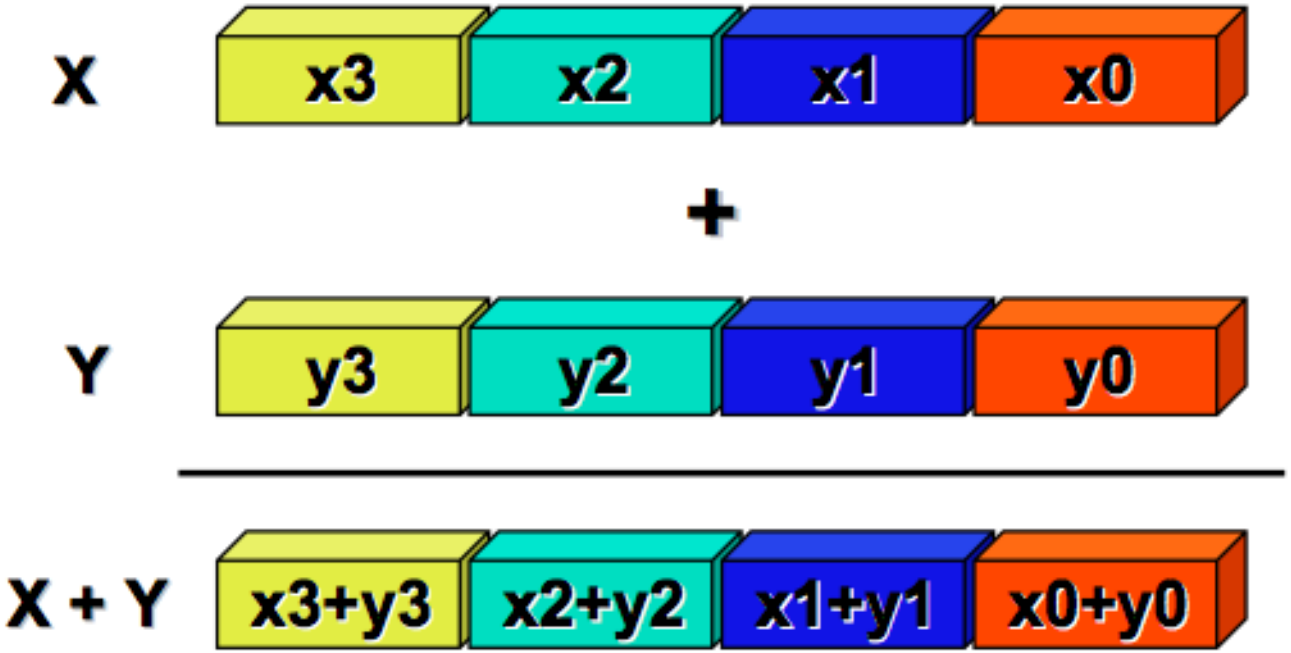


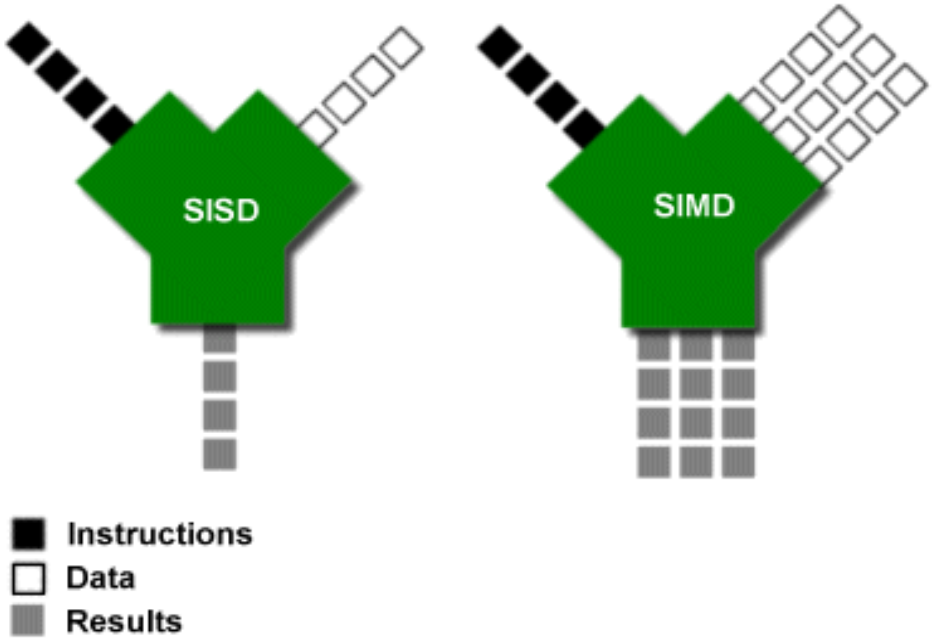
SIMD is a parallelized architecture in the single-core domain

[Wikipedia, Flynn's Taxonomy]→

Single Instruction/Multiple Data (SIMD) Architectures

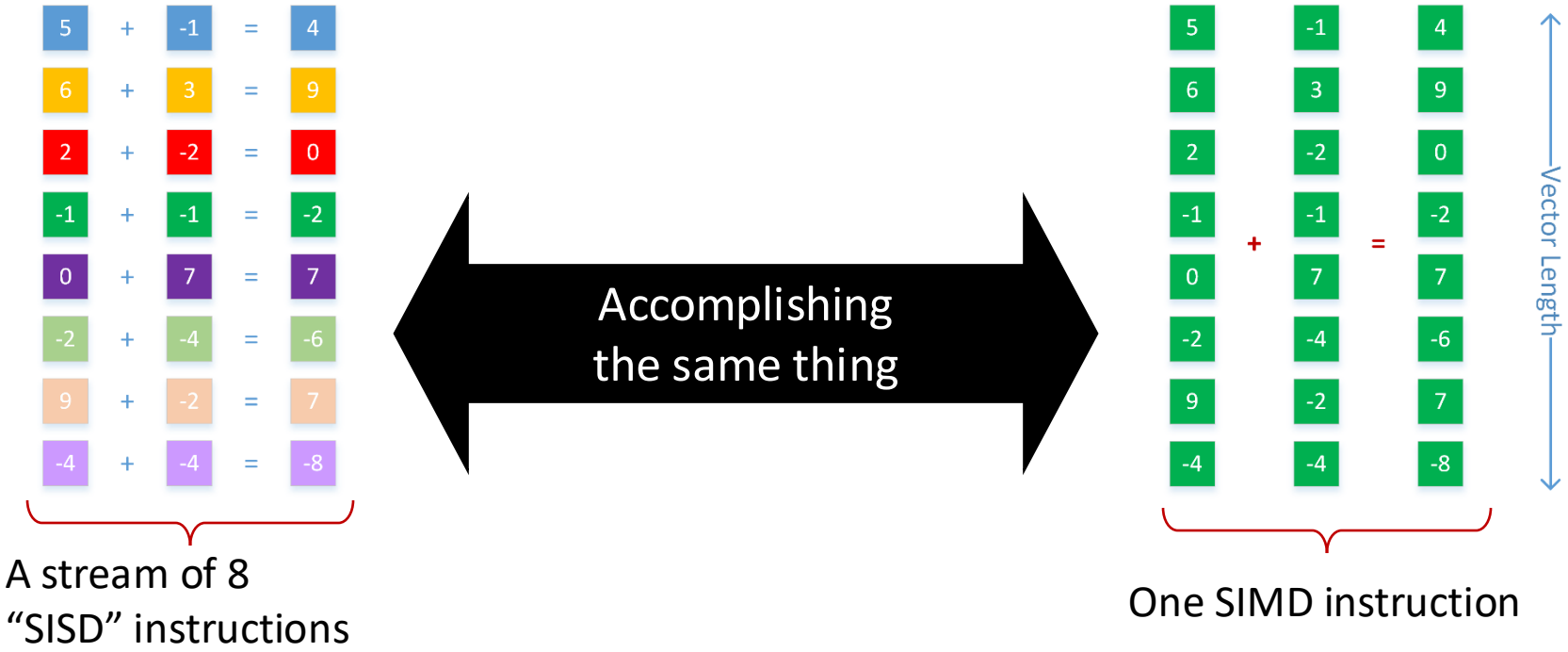
- SIMD architecture is useful for many math applications
  - Ex: adding two vectors (X and Y) of numbers





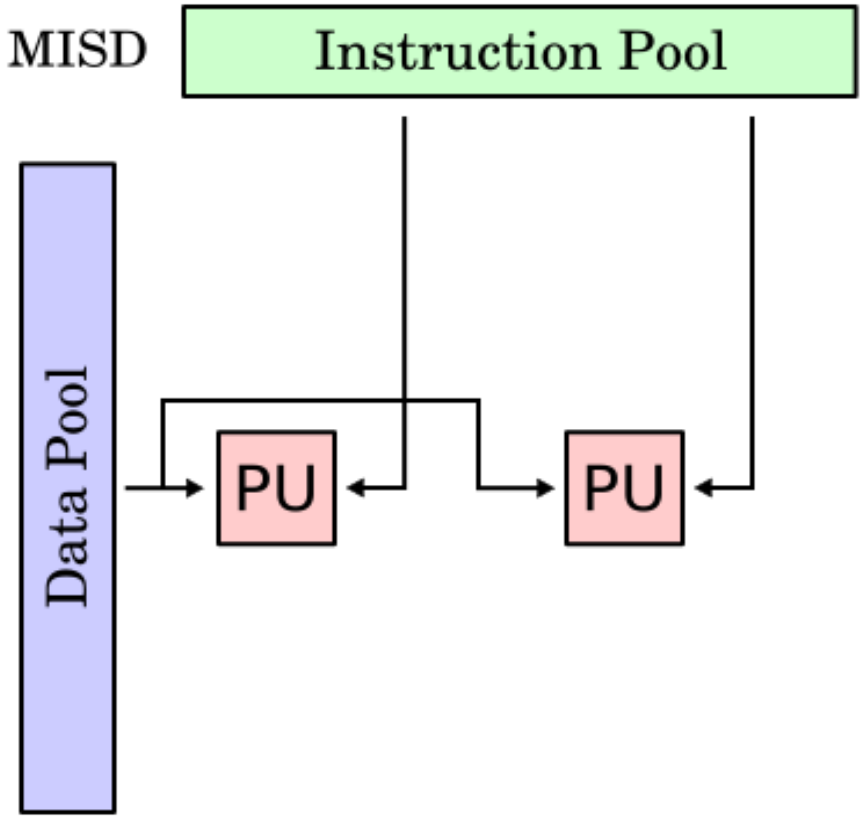
[Klimovitski & Macri, Intel]→

SISD vs. SIMD; Vector Length



- The biggest difference between SISD and SIMD is the data vector length
  - Assume numbers shown above are integers
  - 8 ints × 4 Bytes each × 8 Bits/Byte = 256 Bit Vector Length (AVX2)
  - Intel AVX512: 512-bit vector length – can simultaneously process 16 integer adds

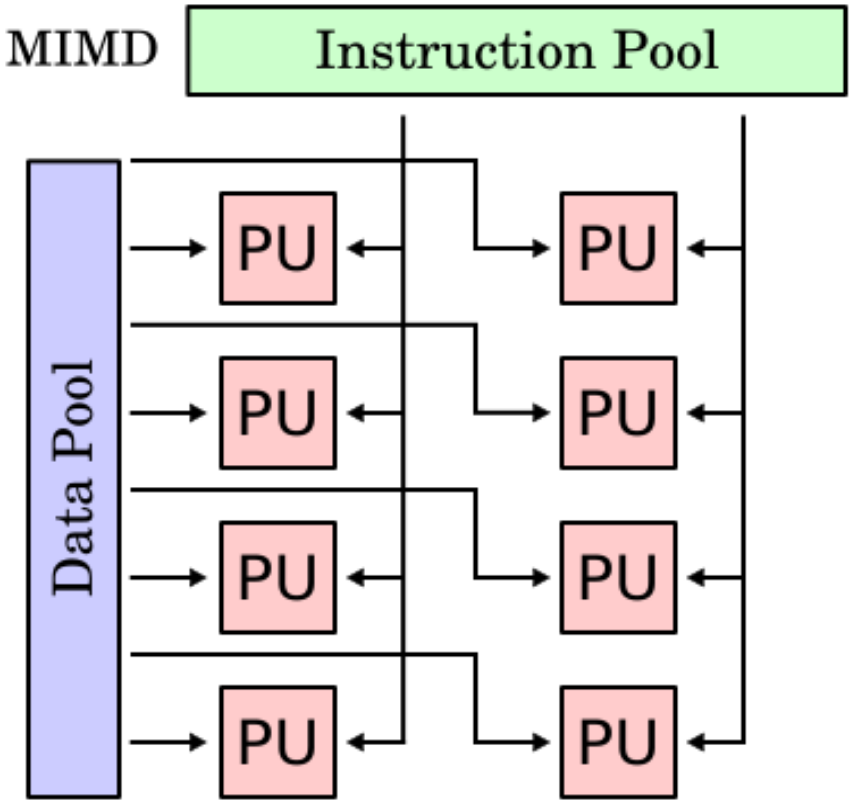
Multiple Instruction/Single Data



Not useful, not aware of any commercial implementation …

[Wikipedia, Flynn's Taxonomy]→

Multiple Instruction/Multiple Data



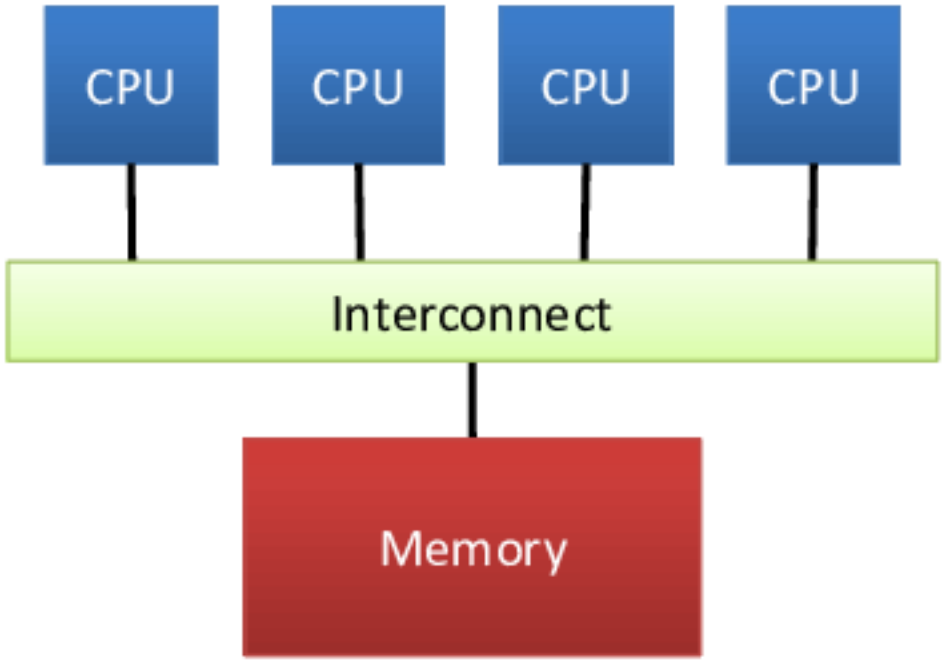
Almost all our desktop/laptop chips are MIMD systems.

Many cores process different data at one time.

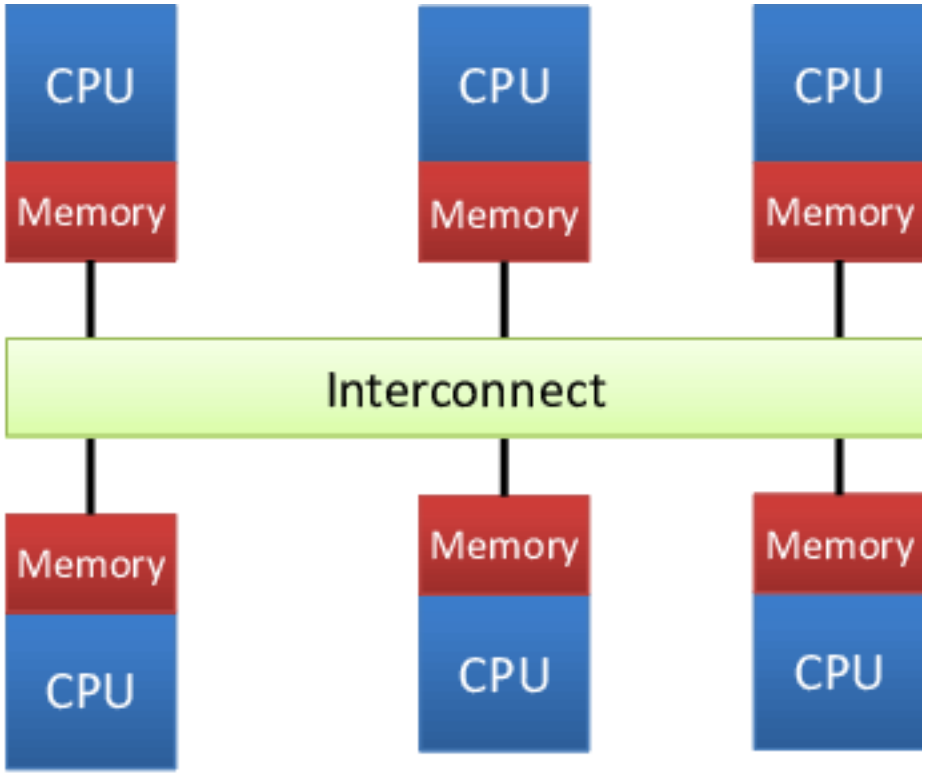
[Wikipedia, Flynn's Taxonomy]→

Multiple Instruction/Multiple Data

- The sky is the limit: each PU is free to do as it pleases
- Can be of either shared memory or distributed memory categories



Shared memory



Distributed memory

How do You Know Parallel Programming Makes Sense?

- Amdahl's Law: speed-up of parallelizing a program is limited by its sequential portion
  - The maximum speed-up gain is limited by the portion of the program that can be parallelized

- Sometimes called the law of diminishing returns -- can't benefit too much from adding more processing power if a significant part of the task can't be parallelized …

- Let r_s capture the amount of time that a program spends in components that can only be run sequentially
- Let r_p capture the amount of time spent in those parts of the code that can be parallelized.
- Assume that r_s and r_p are normalized, so that $r_s + r_p = 1$
- Let n be the number of threads used to parallelize the part of the program that can be executed in parallel
- The "best case scenario" speedup S is

$$S = \frac{T_{old}}{T_{new}} = \frac{r_s + r_p}{r_s + \frac{r_p}{n}} = \frac{1}{r_s + \frac{r_p}{n}}$$

- Algorithms for which $r_p = 1$ are called "embarrassingly parallel"

Quiz: Amdahl's Law

- Suppose a program has a sequential portion of 60%, and the rest 40% can be parallelized

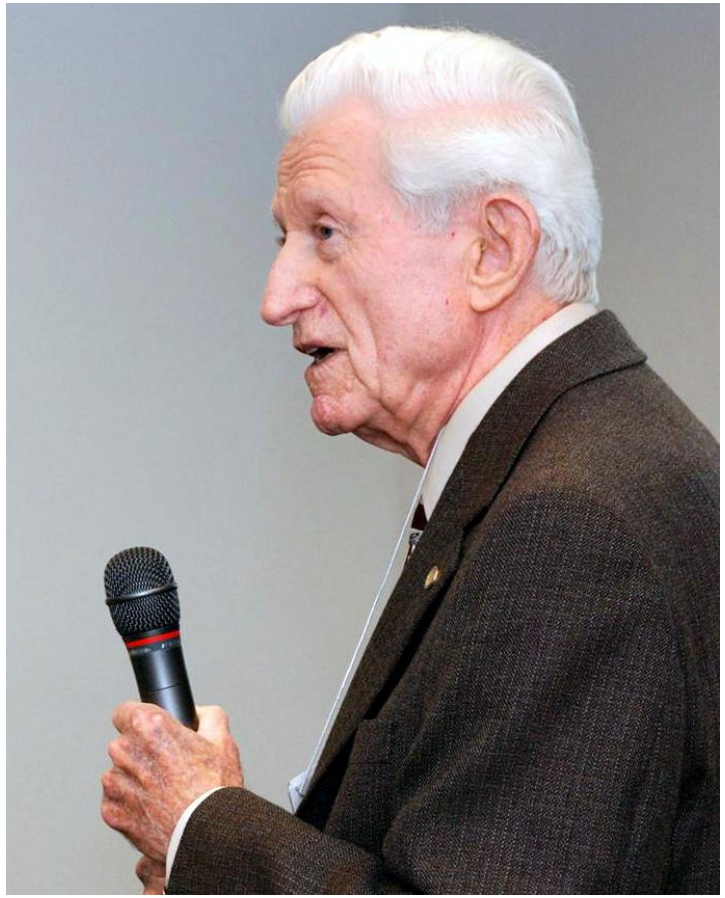
- Assume that you buy a multicore chip and can throw 6 parallel threads at this problem
  - What is the maximum amount of speedup that you can expect given this investment?
  - Asymptotically, what is the maximum speedup that you can ever hope for?

Maximum speed-up we can expect: (60 + 40)/(60 + 40/6) = 1.5x

Asymptotically maximum speed-up we can hope for: (60 + 40)/(60 + 40/∞) = 100/60 / 1.666x

Trivia: Do You Know Amdahl is an Alumni of UW-Madison?

- Dr. Gene Amdahl (1922-2015)
- Graduated from UW-Madison
  - 1952 PhD, Mathematics and Physics
- Amdahl's law (1967)



A Simple C++ Thread Program using `std::thread`

```
#include <iostream>
#include <thread>

// function for the first thread
void printNumbers() {
   for (int i = 1; i <= 5; ++i) {
      printf("Number: %d\n”, i);
   }
}

// function for the second thread
void printLetters() {
   for (char letter = 'A'; letter <= 'E'; ++letter) {
      printf(“Letter: %c\n”, letter);
   }
}
```

```
int main() {
   // create two threads, each running a function
   std::thread t1(printNumbers);
   std::thread t2(printLetters);

   // join each thread
   t1.join();
   t2.join();

   std::cout << "Both threads finished!" << std::endl;

   return 0;
}
```

Output of the Program

- Output can interleave in-between the two threads – *execution order is not deterministic*
  - That is, which thread runs which instruction is decided by operating system

```
Number: 1
Letter: A
Number: 2
Letter: B
Number: 3
Letter: C
Number: 4
Letter: D
Number: 5
Letter: E
Both threads finished!
```

```
Number: 1
Number: 2
Letter: A
Letter: B
Letter: C
Number: 3
Number: 4
Number: 5
Letter: D
Letter: E
Both threads finished!
```

Parallel Programming is NOT as Easy as You Thought …

```
std::string* ptr = 0;
int data = 0;
void producer() {
   std::string* p = new std::string("Hello");
   data = 42;
   ptr = p;
}
void consumer() {
   std::string* p2;
   while (!(p2 = ptr)) ;
   assert(*p2 == "Hello");          // can this fail?
   assert(data == 42);              // can this fail?
}
int main() {          // sequentially run prod/cons
   producer();
   consumer();
}
```

```
std::string* ptr = 0;
int data = 0;
void producer() {
   std::string* p = new std::string("Hello");
   data = 42;
   ptr = p;
}
void consumer() {
   std::string* p2;
   while (!(p2 = ptr)) ;
   assert(*p2 == "Hello");          // can this fail?
   assert(data == 42);              // can this fail?
}
int main() {          // parallelly run prod/cons
   std::thread t1(producer);
   std::thread t2(consumer);
   t1.join(); t2.join();
}
```

Opportunities for Efficiency Gains through Parallelism

Cluster (distributed mem.)	Group of nodes communicate through fast interconnect	MPI, Charm++, Chapel
Multi-Socket Node	Group of CPUs on the same node, talk through main mem.	OpenACC, OpenMP, MPI
Acceleration (GPU)	Compute devices accelerating parallel computation on one node	CUDA, OpenCL, OpenMP, OpenACC
Multi-Core Processor	Communication through shared caches and main mem.	OpenMP, TBB, pthreads, OpenACC
Vectorization	Higher operation throughput via special/fat registers	AVX, SSE, OpenMP
Pipelining	Sequence of instruction sharing functional units	Instruction-level
Superscalar	Non-sequence instructions sharing functional units	Instruction-level

We have full control → We have little to no control →