CS 6023 - GPU Programming

Background: CPU Architecture

18/01/2019

Agenda

How do we contrast CPUs from GPUs?

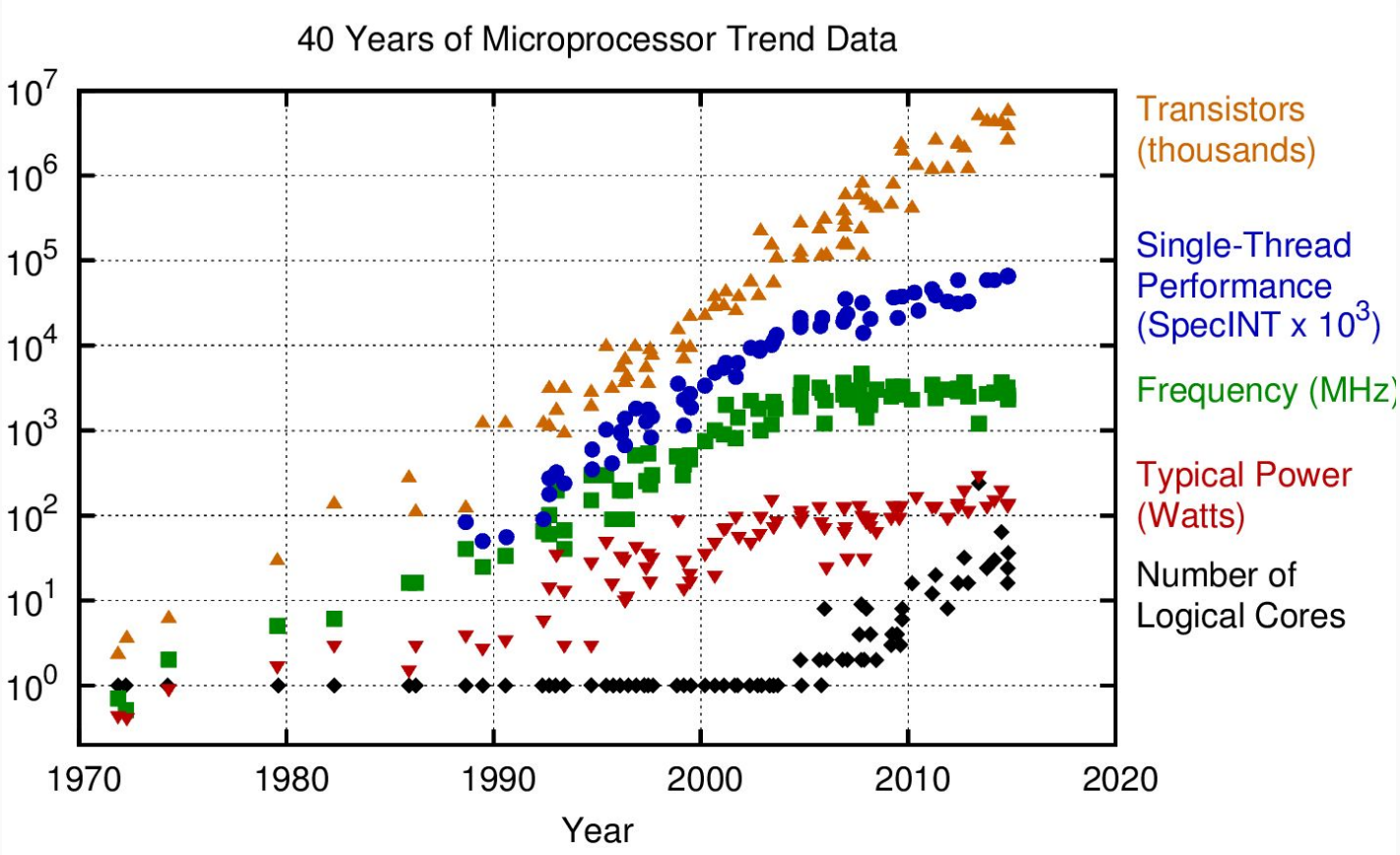
Part 1 of 2:

Today: CPU architecture background

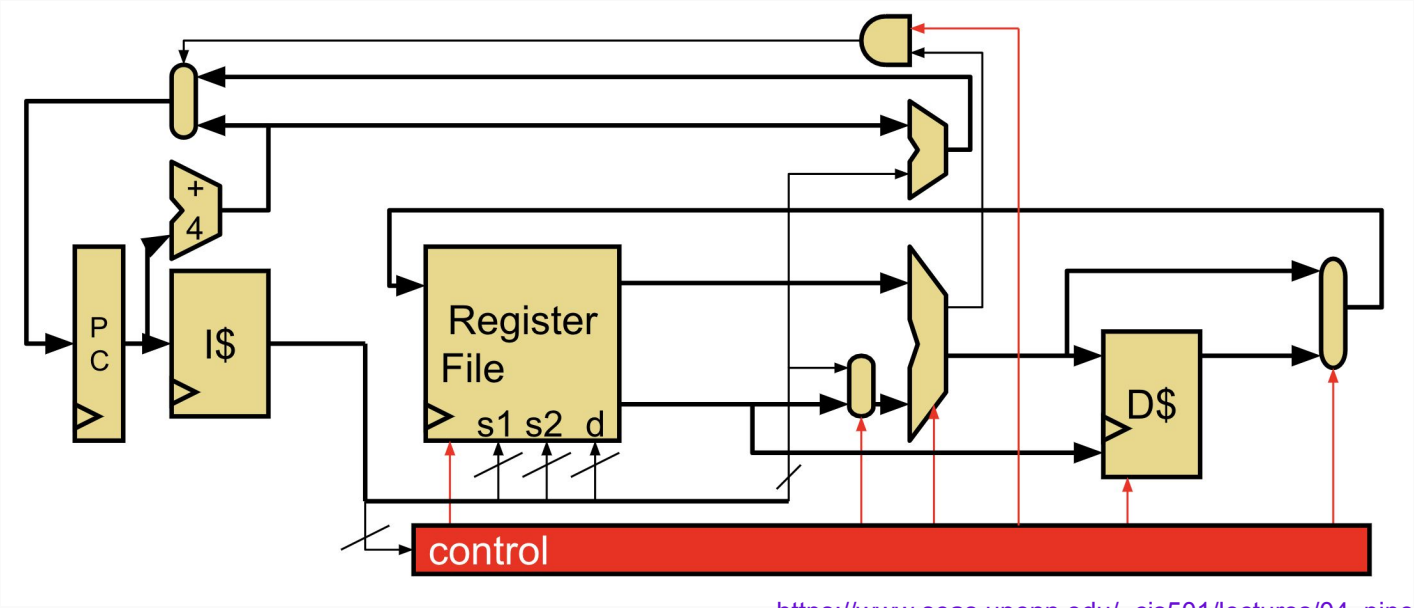
Part 2 of 2:

Next time: Parallel computer architecture background

CPUs have made phenomenal progress



Background - CPU architecture



https://www.seas.upenn.edu/~cis501/lectures/04_pipeline.pdf

- Von Neumann architecture
- Simple CPU stages: Fetch -> Decode -> Execute -> Memory -> Writeback

The performance equation

Latency = seconds / program =
(instructions / program) * (cycles / instruction) * (seconds / cycle)

Instructions / program depends on program, compiler, instruction set architecture (ISA)
Cycles / instruction (CPI) depends on program, compiler, ISA, micro-architecture
Seconds / cycle (clock period) depends on micro-architecture, technology parameters

Fundamental performance quest in single core CPU:
*How to devote transistors on a chip to make a **single stream of instructions** run faster and faster*

Why latency?

- Pizza company needs to decide between delivering pizzas hot or delivering a large number of pizzas per hour
- In the desktop world, **Amdahl's law** restricts potential speed-up
- => Make the usual case faster

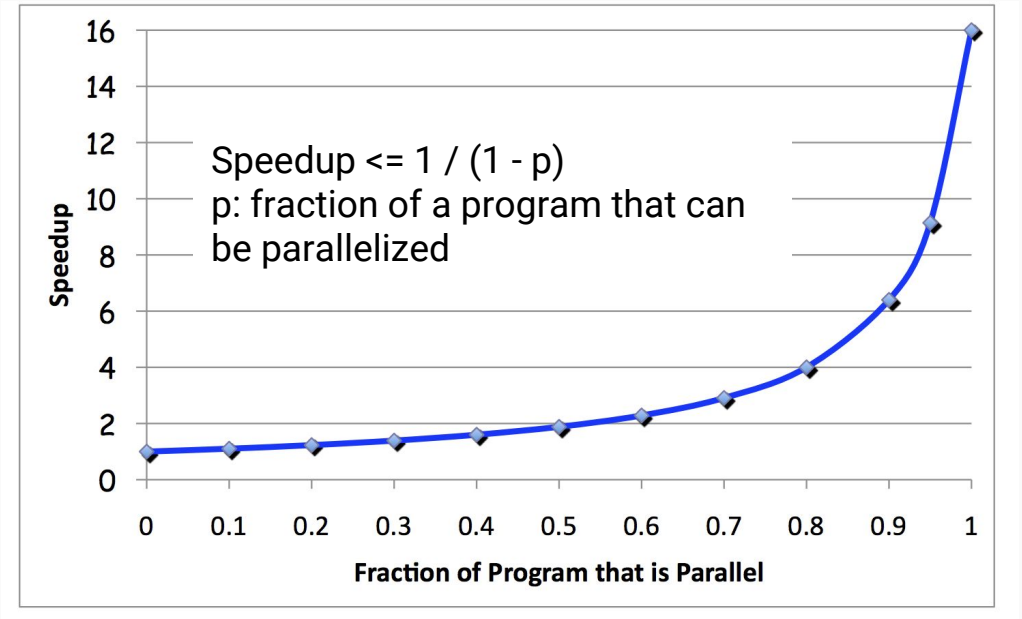
  => Reduce latency for a

  single stream

Why latency?

- Pizza company needs to decide between delivering pizzas hot or delivering a large number of pizzas per hour
- In the desktop world, **Amdahl's law** restricts potential speed-up
- => Make the usual case faster

  => Reduce latency for a single stream

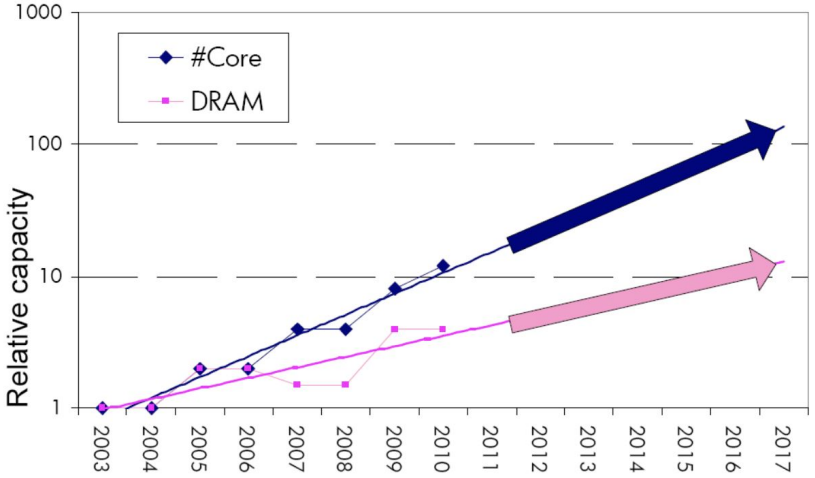


Great performance ideas for CPUs

1. Memory hierarchy (caches)
2. Pipelined execution
3. Branch prediction
4. Out-of-order execution
5. Superscalar
6. Vector processing
7. Multi-threading
8. Multi-core

1. Caching

Core count doubling ~ every 2 years
DRAM DIMM capacity doubling ~ every 3 years



Source: Lim et al., ISCA 2009.

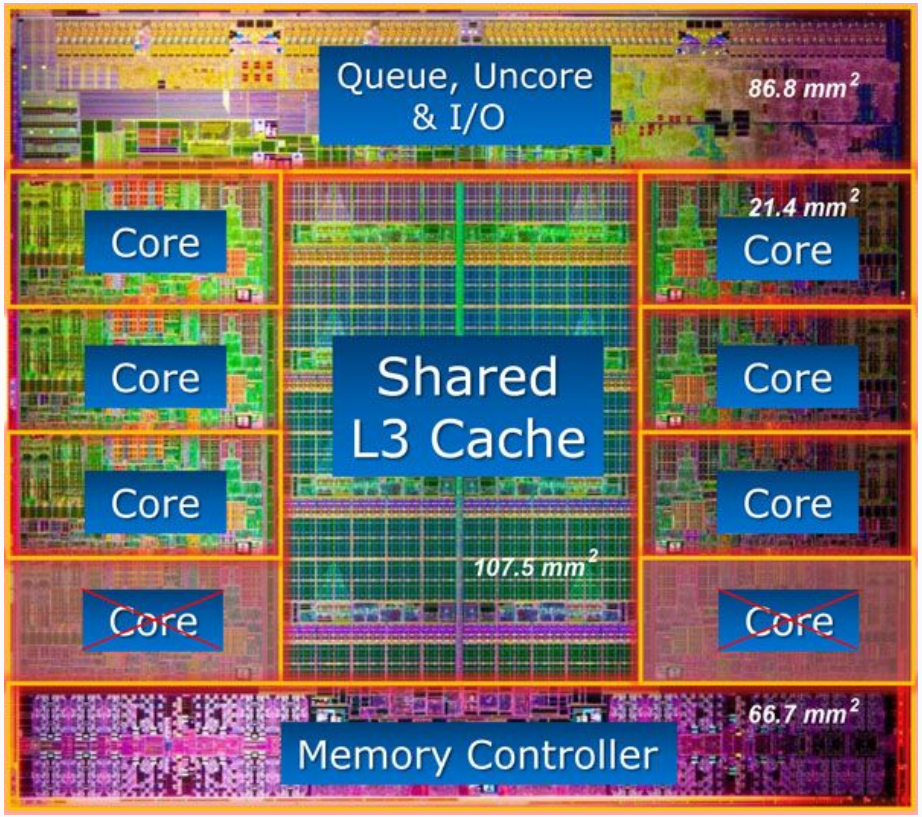
Keep more frequent data closer

Take advantage of locality: Spatial, temporal

Costs: Large on-chip area

	Latency	Bandwidth	Size
SRAM (L1, L2, L3)	1-2ns	200GBps	1-20MB
DRAM (memory)	70ns	20GBps	1-20GB
Flash/SSD (disk)	70-90µs	200-500MBps	100-1000GB
HDD (disk)	10ms	1-150MBps	500-3000GB

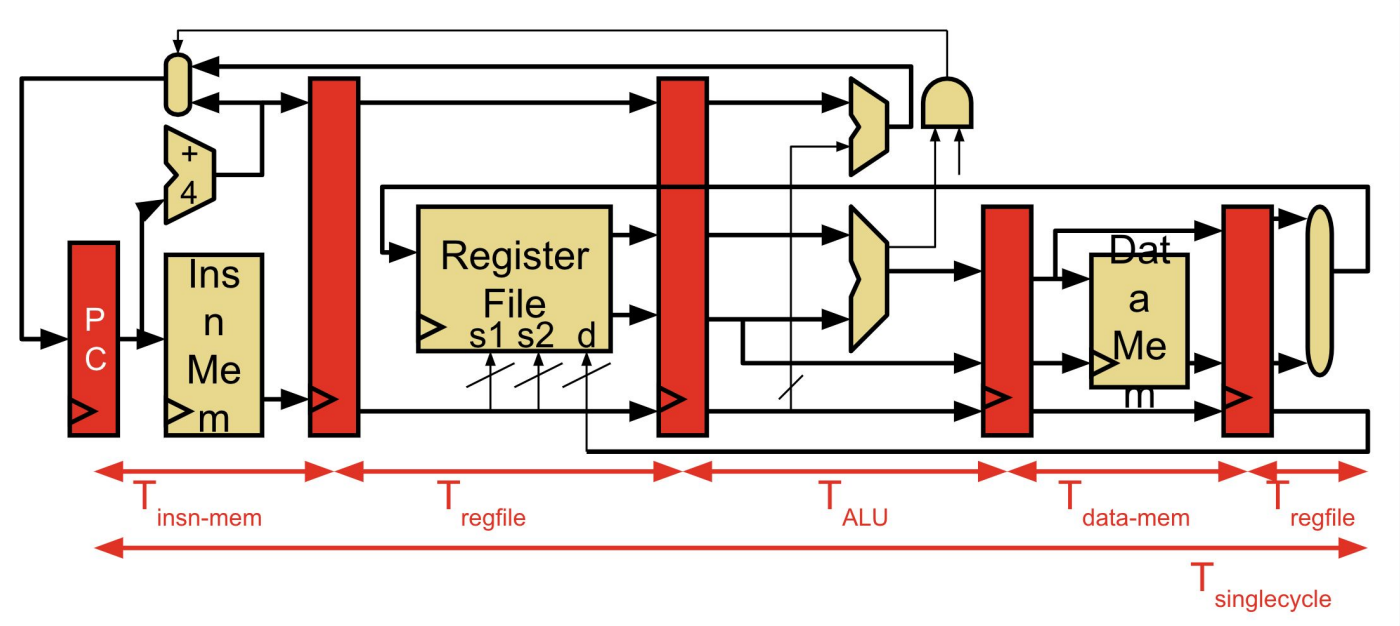
1. Caching



Intel Core i7

www.lostcircuits.com

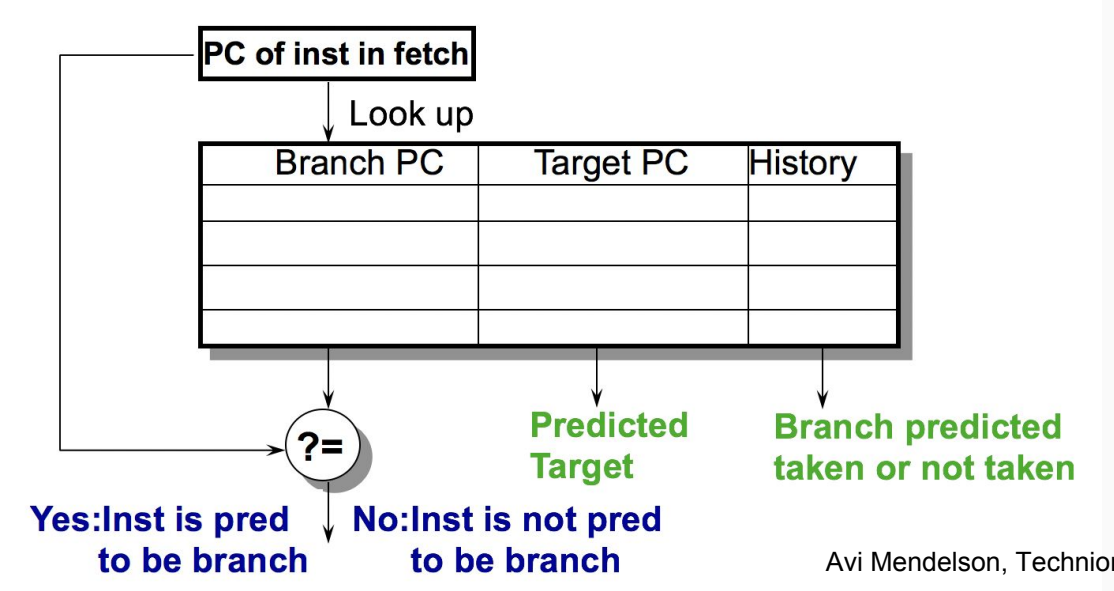
2. Pipelining



https://www.seas.upenn.edu/~cis501/lectures/04_pipeline.pdf

- Increases clocks / sec
- Costs: Larger chip area / transistor count

3. Branch prediction



- Pipelines can stall if branch instruction arises. Branch prediction to ‘guess’ the branch and preemptively execute. In case of error, roll-back
- Costs: Large chip area and transistors for prediction logic, roll-back mechanism

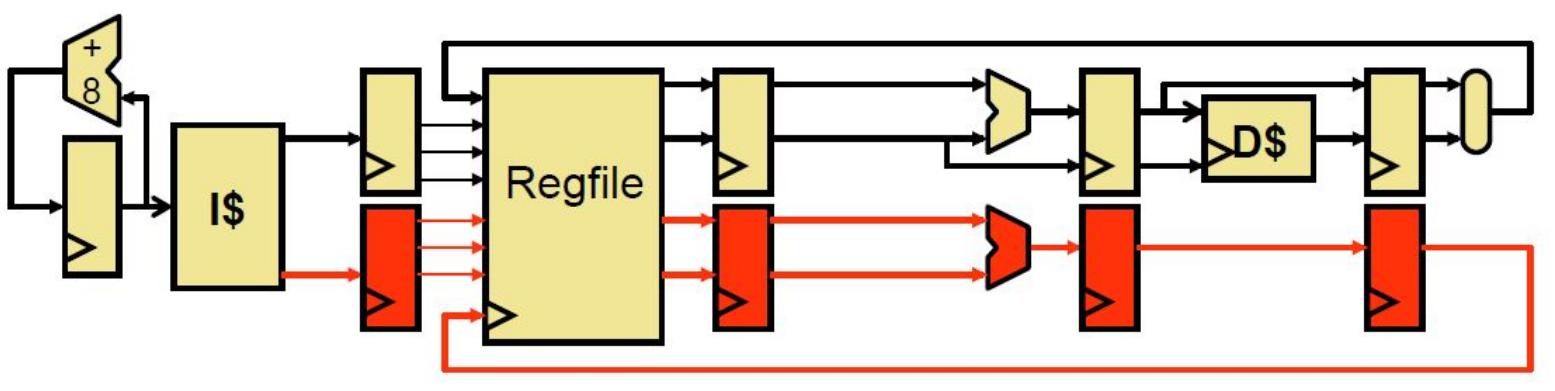
4. Out of order execution

```
IMUL R3 ← R1, R2
ADD  R3 ← R3, R1
ADD  R4 ← R7, R6
```

```
LD   R3 ← R1(0)
ADD  R3 ← R3, R1
ADD  R4 ← R7, R6
```

- Move dependent instructions out of the way of dependent instructions
- In the above examples, instruction 2 stalls in-order pipeline
- Execute instructions **dynamically** based on when required sources are available
- Think of what can be done in compile-time and what is run-time on muArch
- Costs: Significant hardware costs of reservation stations, reorder buffer, scheduler

5. Superscalar



- Increase instructions per cycle by executing concurrently non-dependent instructions
- Requires hardware units and hardware control logic. Limited dependence on compiler
- Cost: Transistor count, hardware control-path complexity

6. Vector processing

```
for (int i = 0; i < N; i+= 4) {
    // in parallel
    A[i]   = B[i]   + C[i];
    A[i+1] = B[i+1] + C[i+1];
    A[i+2] = B[i+2] + C[i+2];
    A[i+3] = B[i+3] + C[i+3];
}
```

- Make ALUs, registers really wide enough to support operations on vectors
- Requires hardware units and compiler support. Vector extensions to x86:
  - **SSE2**: 4-wide vector operations on Intel Pentium 4, AMD Athlon 64
  - **AVX**: 8-wide vector operations on Intel Sandy Bridge, AMD Bulldozer
- Cost: Compiler support. Hardware cost

7. Multi-threading

Plot from 2002 paper from Intel that proposed hyperthreading

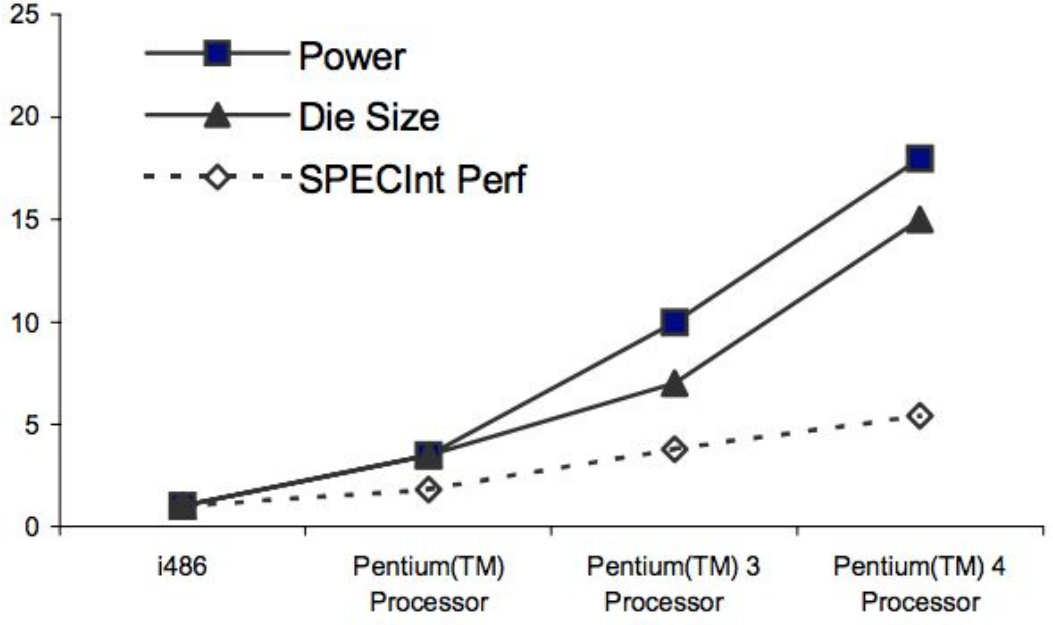


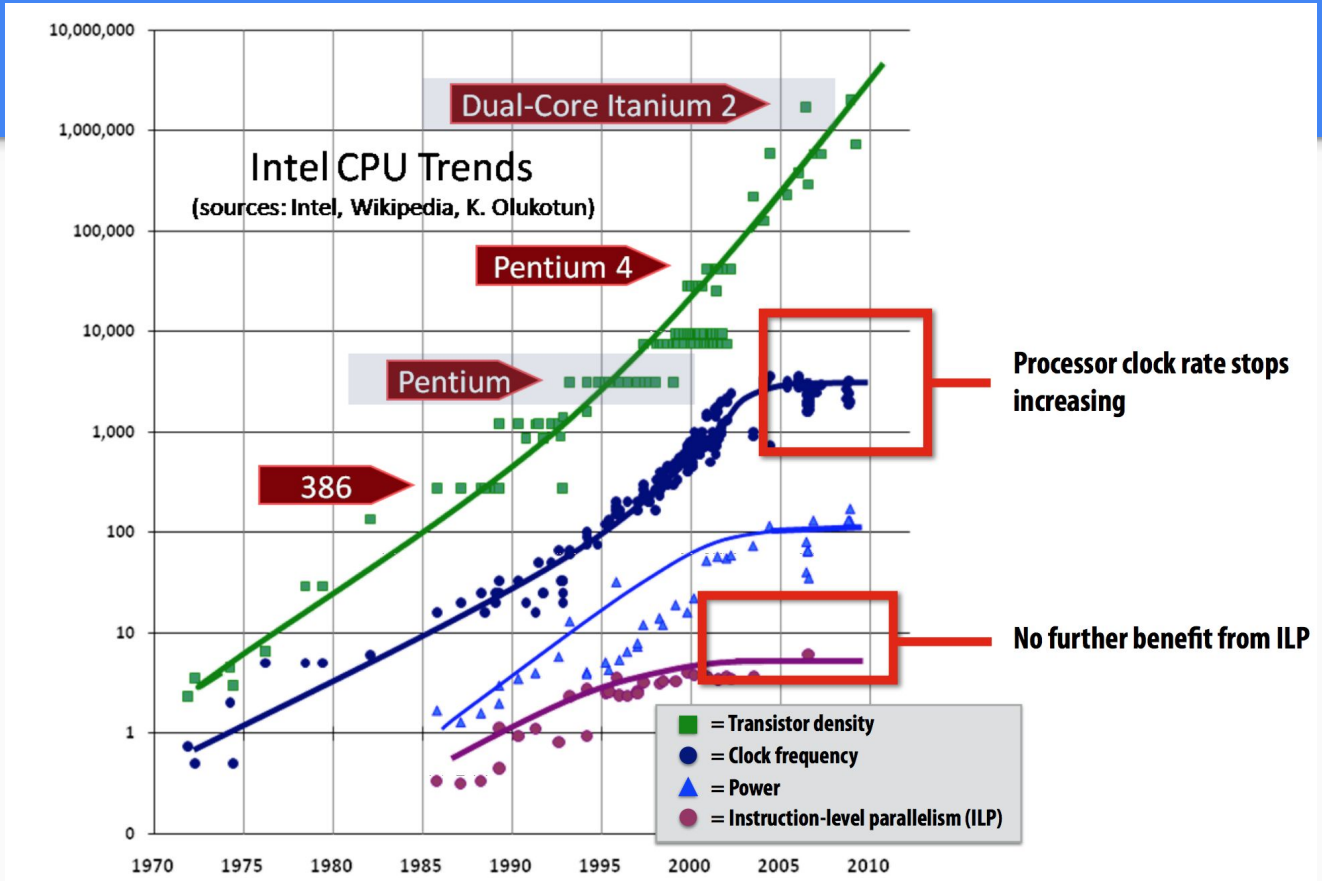
Figure 1: Single-stream performance vs. cost

- Thread composition requires maintaining architectural state: PC, registers, stack, shared globals, partitioning of ROB, other buffers
- Cost: Hardware for maintaining and switching contexts, OS management, cache conflicts, programming complexity

8. Multi-core

Inflection point in 2004, when Intel hits the power wall

Source: “No free lunch” by Dr. Dobbs in 2005



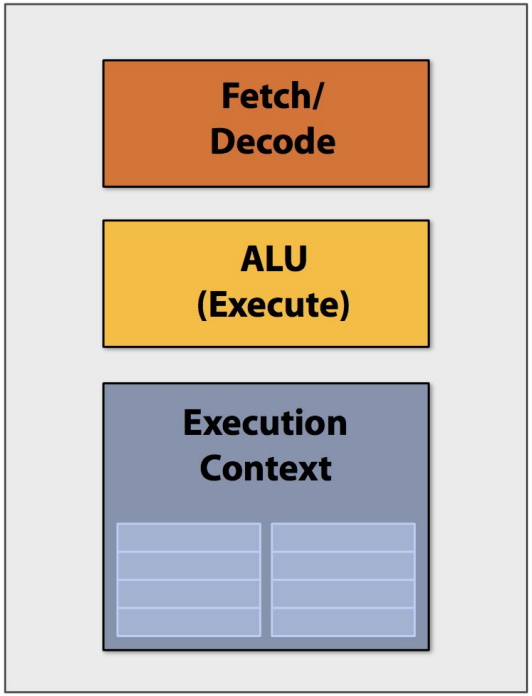
- Finally, just replicate the entire pipeline multiple times on the same chip
- No resource sharing, except for last-level (usually L3) cache
- Cost: Programmer difficulty, resource contentions

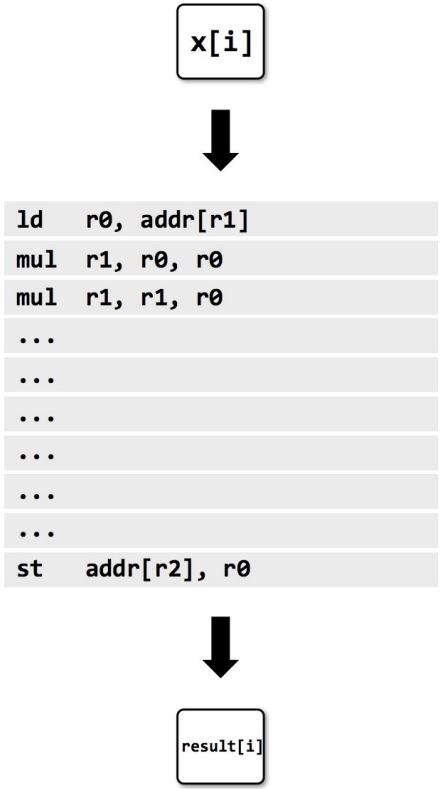
Piecing it together

- 1970s: Single stage combinational processing
- 1980s: Pipelining to exploit temporal parallelism
- 1990s: Exploit instruction-level and data-level parallelism

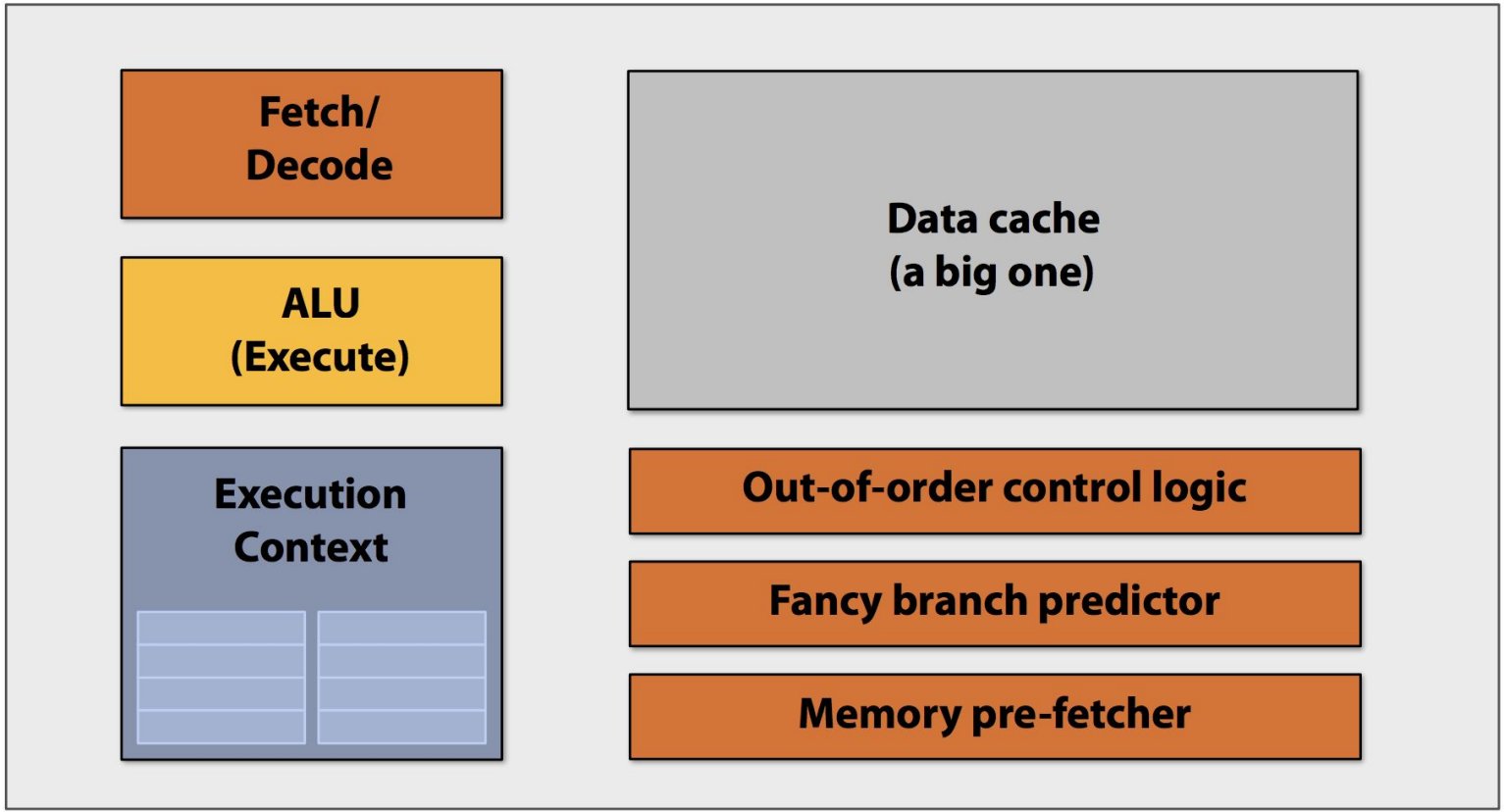
  Out-of-order execution, superscalar, speculative execution, VLIW
- 2000s: Thread level parallelism with simultaneous multithreading (SMT)
- Mid to Late 2000s: Multi-core processing with duplicated pipelines
- 2010s: Many core processing

CPU - single core cartoon

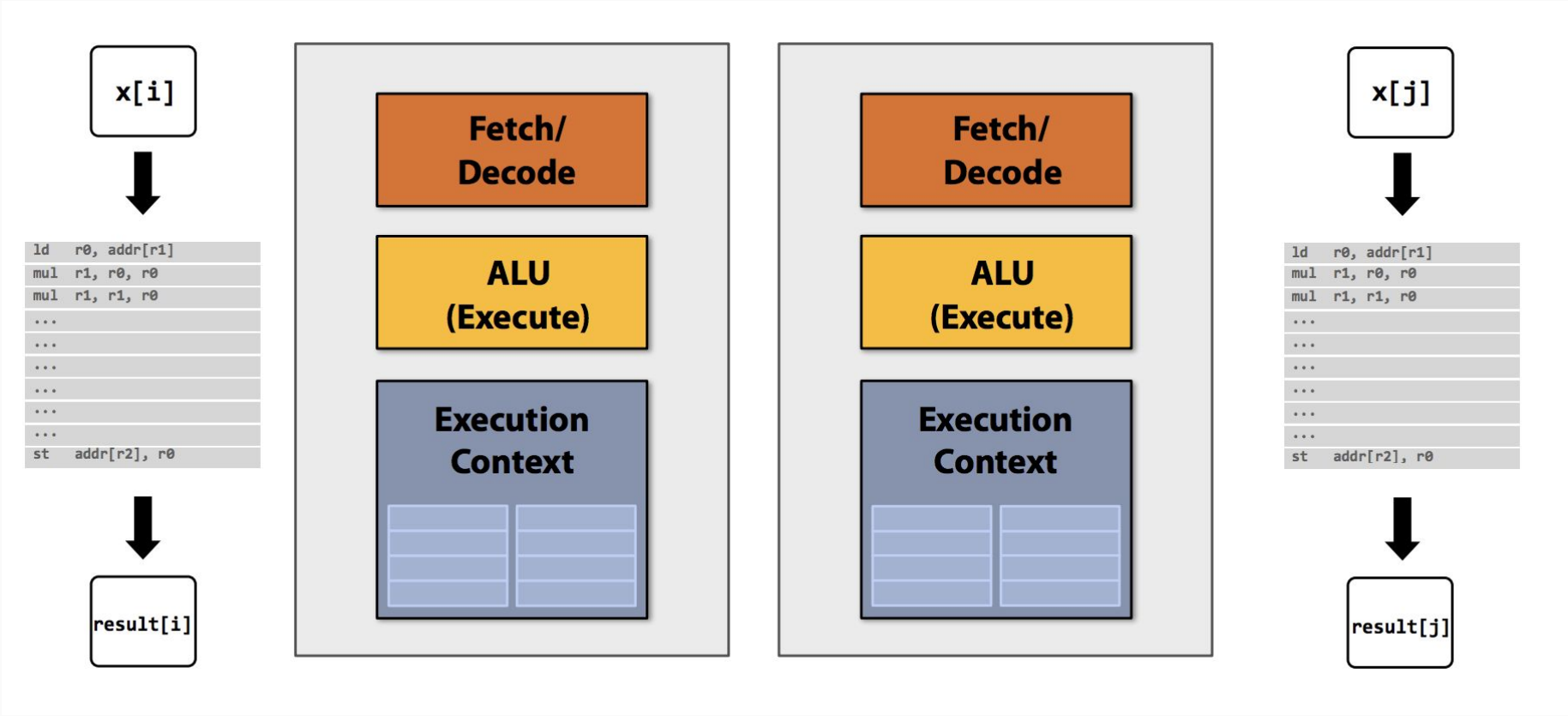




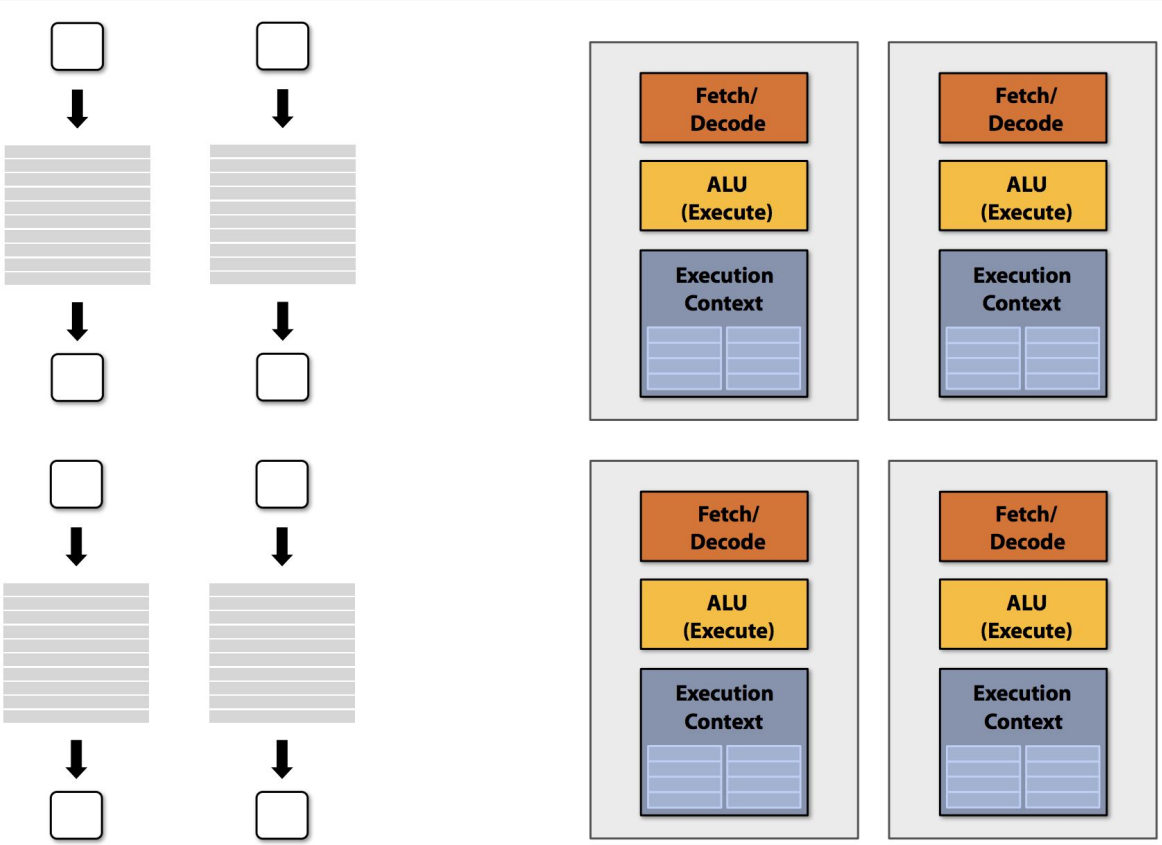
CPU cartoon - single core era



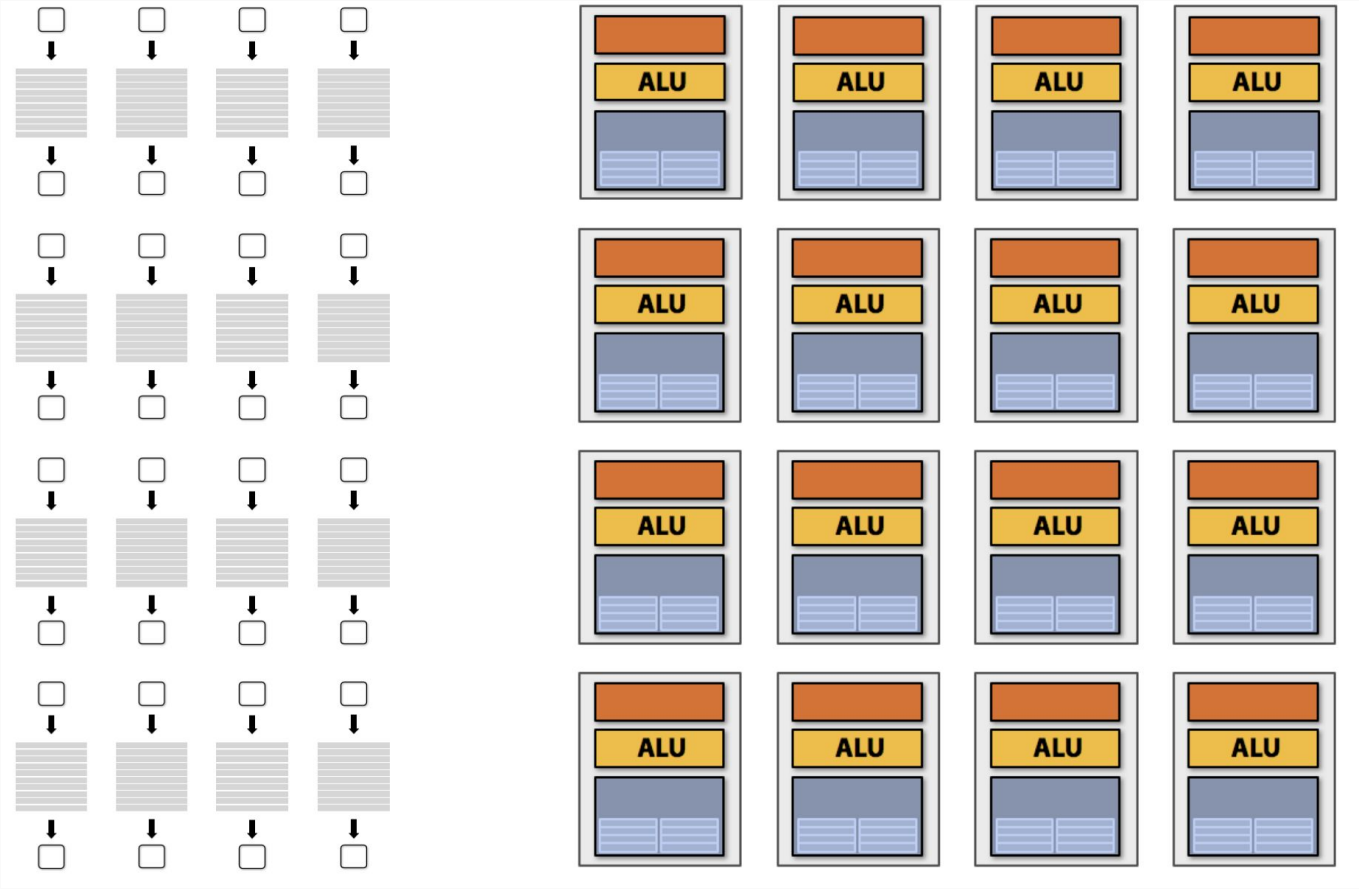
CPU cartoon - multi-core



CPU cartoon - many core



CPU cartoon - many core



Why don’t we keep doing this for better and better CPUs?

Typical desktop workload

	`vim`	`ls`
Conditional branches		
Memory accesses		
Vector instructions		

Typical desktop workload

- Desktop Applications
  - Lightly threaded
  - Lots of branches
  - Lots of memory accesses

	`vim`	`ls`
Conditional branches	13.6%	12.5%
Memory accesses	45.7%	45.7%
Vector instructions	1.1%	0.2%

Next time

- Parallel computer architecture - background