The Underlying Architecture

Spring 2019

Prof. Karen Karavanic

Portland State
UNIVERSITY

Acknowledgments

- This presentation includes materials developed by others:
  - 15-213: Introduction to Computer Systems, 2010
    - Randy Bryant and Dave O'Hallaron
  - Kathy Yelick, UC-Berkeley

Outline

- The Single Core Era (-> 2006)
- The Multicore Era (2006 ->)
  - 5. Why Multicore?
  - 6. Manycore

The Single Core Era
Key Hardware Advances

- Instruction Level Parallelism (ILP)
- Pipelining
- Branch Prediction
- Multiple Instruction Issue
- The Memory Hierarchy

Pipelining

The Insight

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions
- Design hardware so that a different instruction can be at each step concurrently
- 1. Instr 1 at stage 1
- 2. Instr 1 at stage 2, Instr 2 at stage 1
- 3. Instr 1 at stage 3, Instr 2 at stage 2, Instr 3 at stage 1
- …

Real-World Pipelines: Car Washes

Sequential

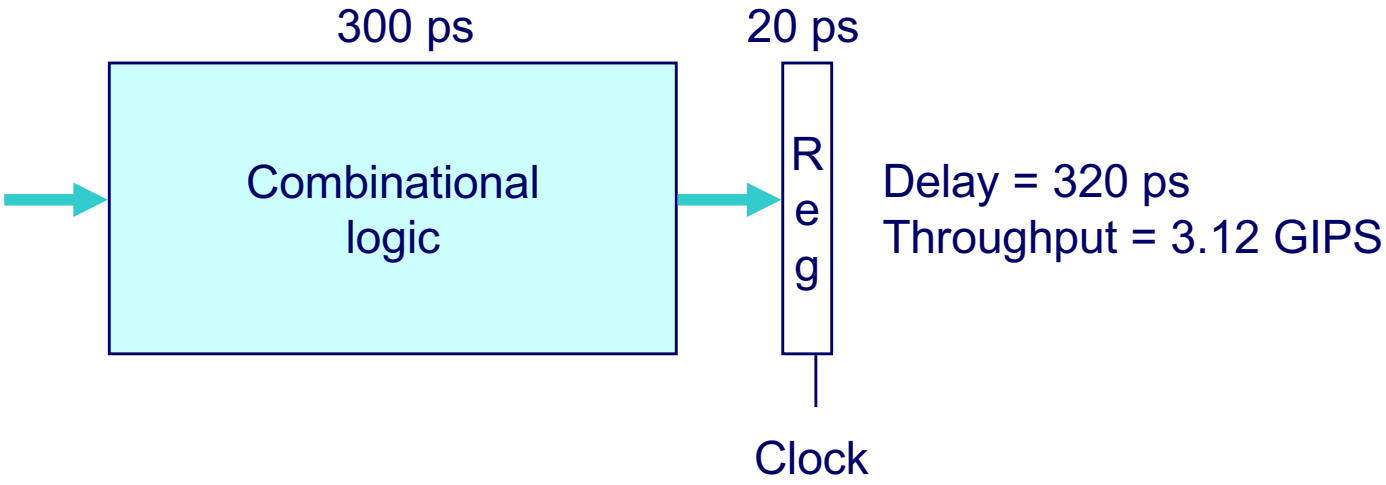
Parallel

Pipelined

Idea

- Divide process into independent stages
- Move each car through stages in sequence
- At any given time, multiple cars being processed

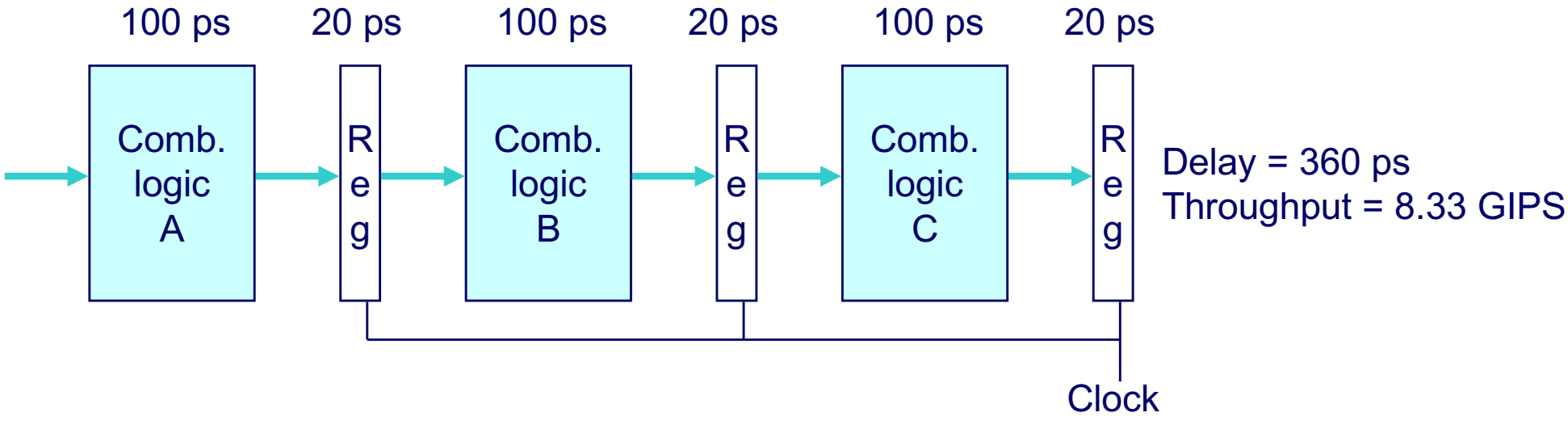
Computational Example



System

- Computation requires total of 300 picoseconds
- Additional 20 picoseconds to save result in register
- Must have clock cycle of at least 320 ps

3-Way Pipelined Version

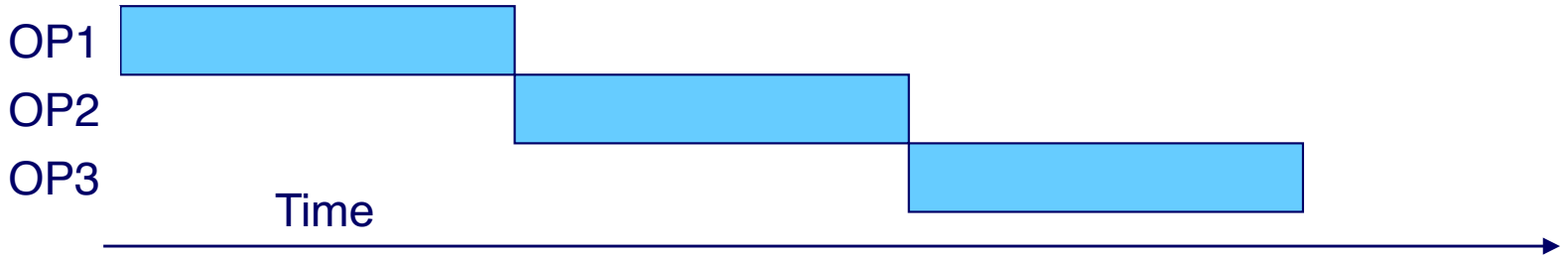


System

- Divide combinational logic into 3 blocks of 100 ps each
- Can begin new operation as soon as previous one passes through stage A.
  - Begin new operation every 120 ps
- Overall latency increases
  - 360 ps from start to finish

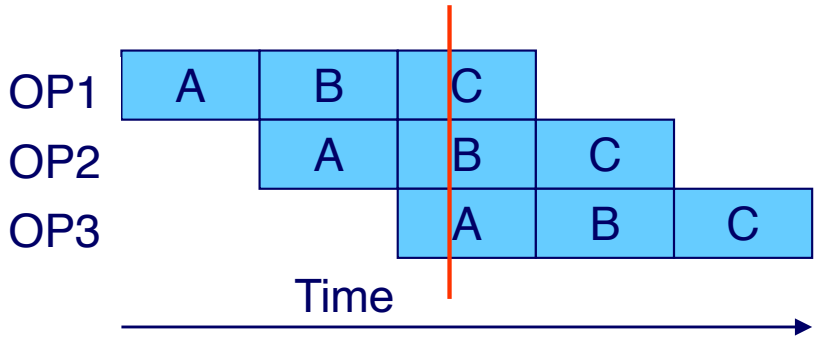
Pipeline Diagrams

Unpipelined



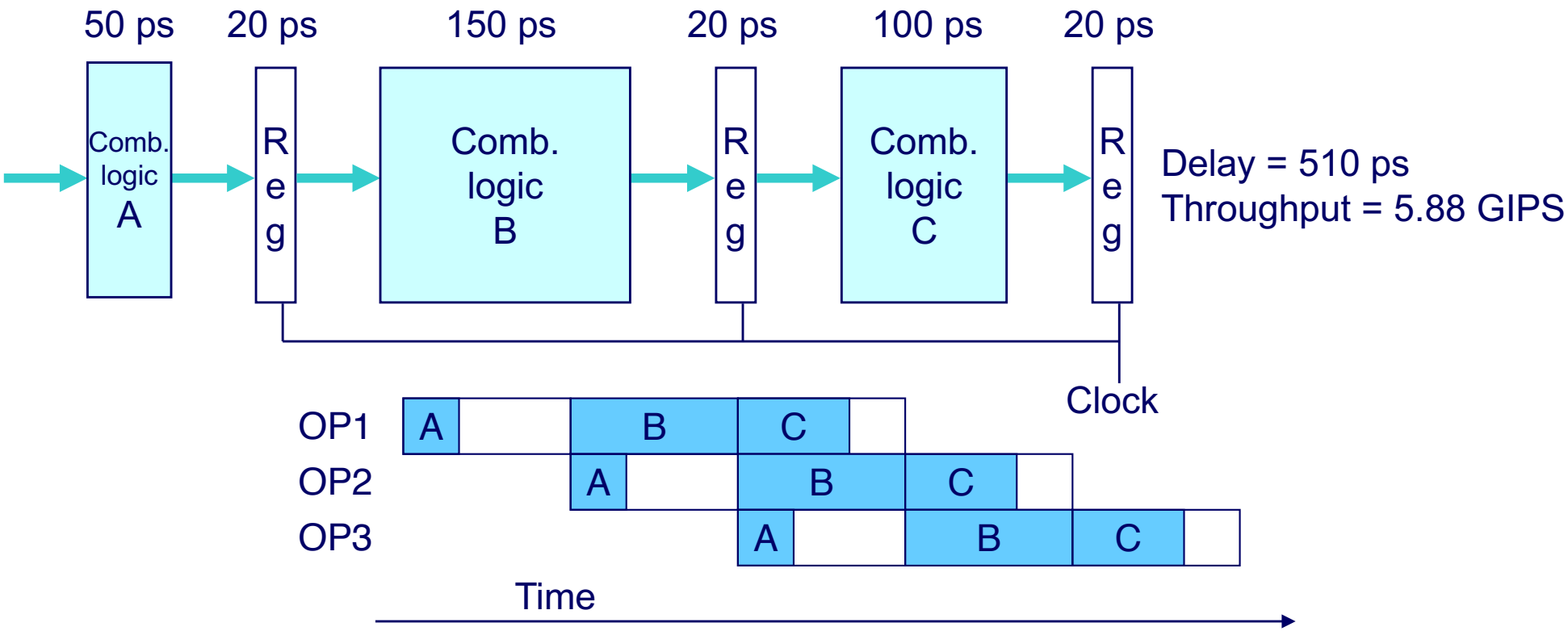
- Cannot start new operation until previous one completes

3-Way Pipelined



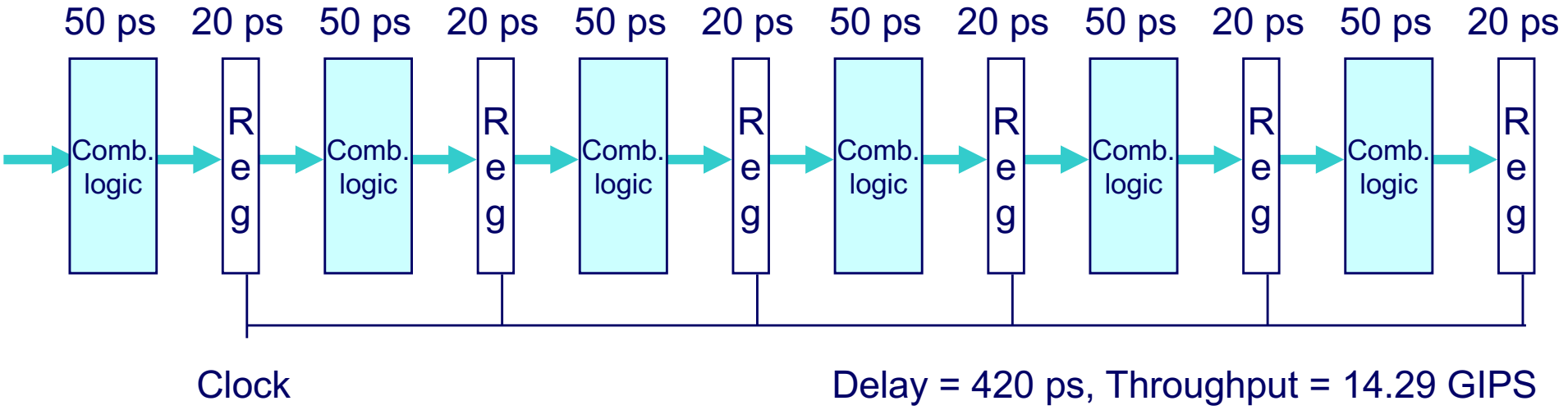
- Up to 3 operations in process simultaneously

Limitations: Nonuniform Delays



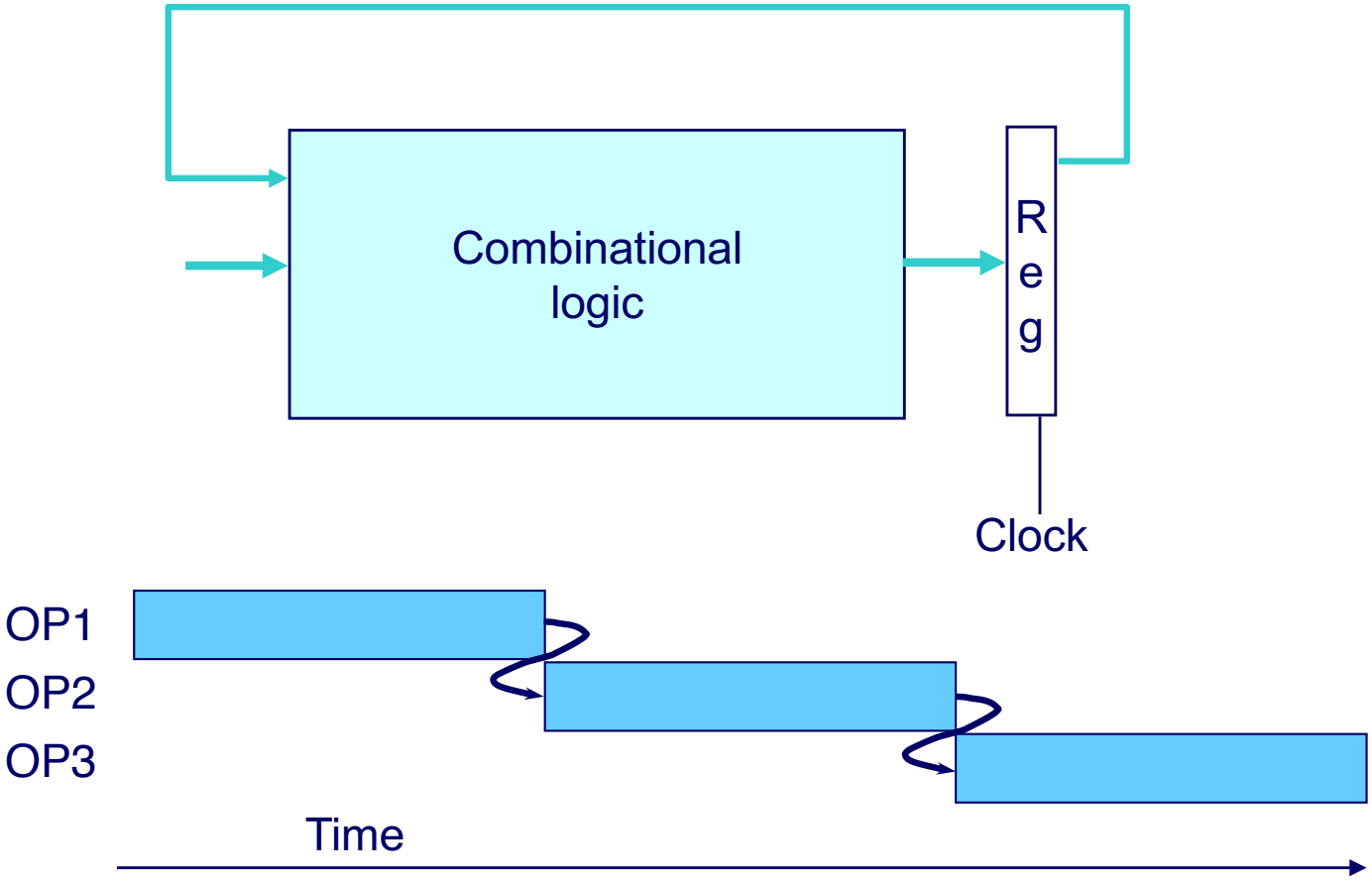
- **Throughput limited by slowest stage**
- **Other stages sit idle for much of the time**
- **Challenging to partition system into balanced stages**

Limitations: Register Overhead



- As try to deepen pipeline, overhead of loading registers becomes more significant
- Percentage of clock cycle spent loading register:
  - 1-stage pipeline: 6.25%
  - 3-stage pipeline: 16.67%
  - 6-stage pipeline: 28.57%
- High speeds of modern processor designs obtained through very deep pipelining

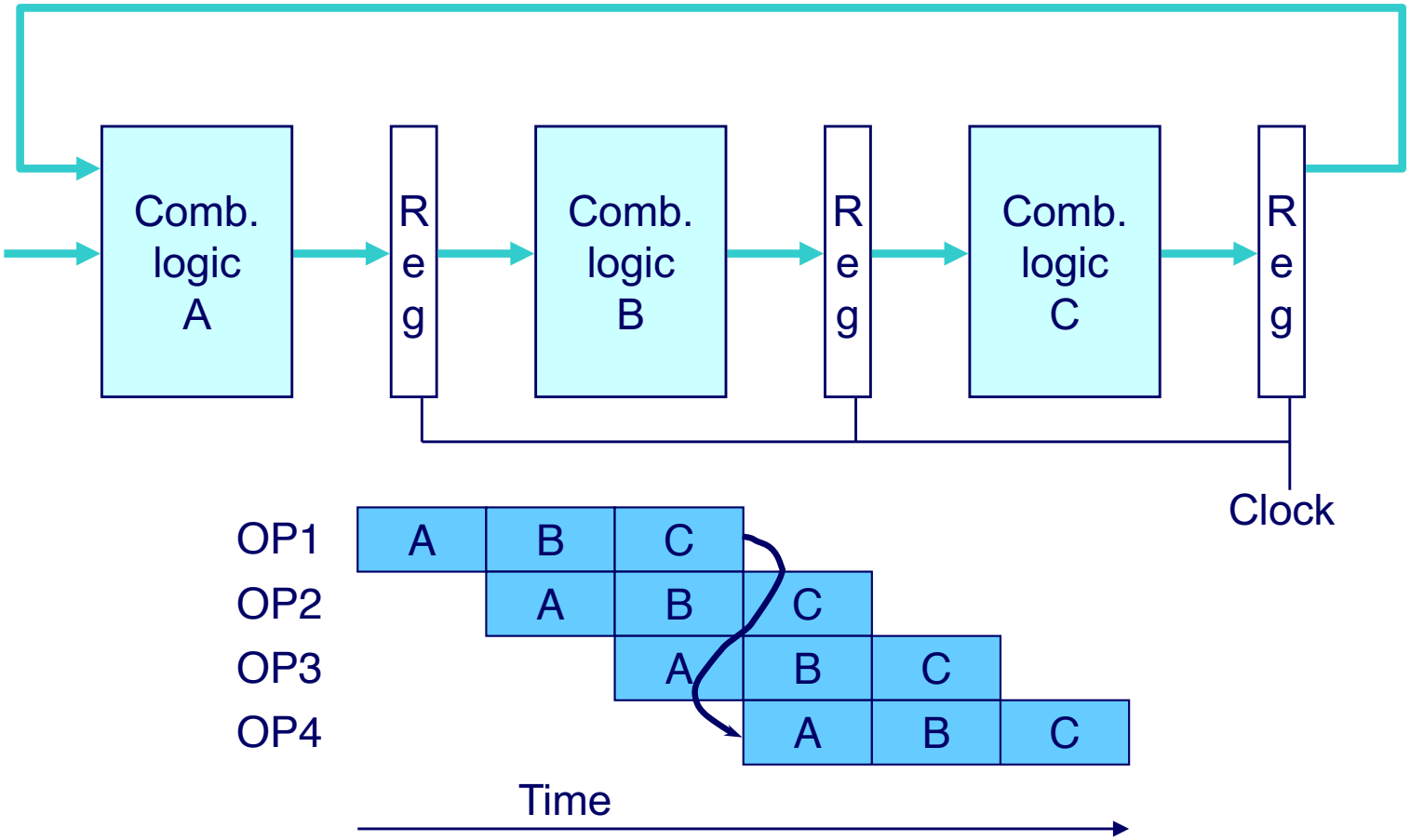
Pipelining Challenges: Data Dependencies



System

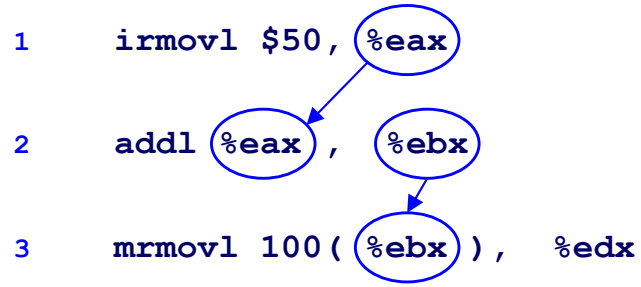
- **Each operation depends on result from preceding one**

Pipelining Challenges: Data Hazards



- **Result does not feed back around in time for next operation**
- **Pipelining has changed behavior of system**

Data Dependencies in Processors



- Result from one instruction used as operand for another
  - Read-after-write (RAW) dependency
- Very common in actual programs
- Must make sure our pipeline handles these properly
  - Get correct results
  - Minimize performance impact

Pipeline Stages

Fetch

- Select current PC
- Read instruction
- Compute incremented PC

Decode

- Read program registers

Execute

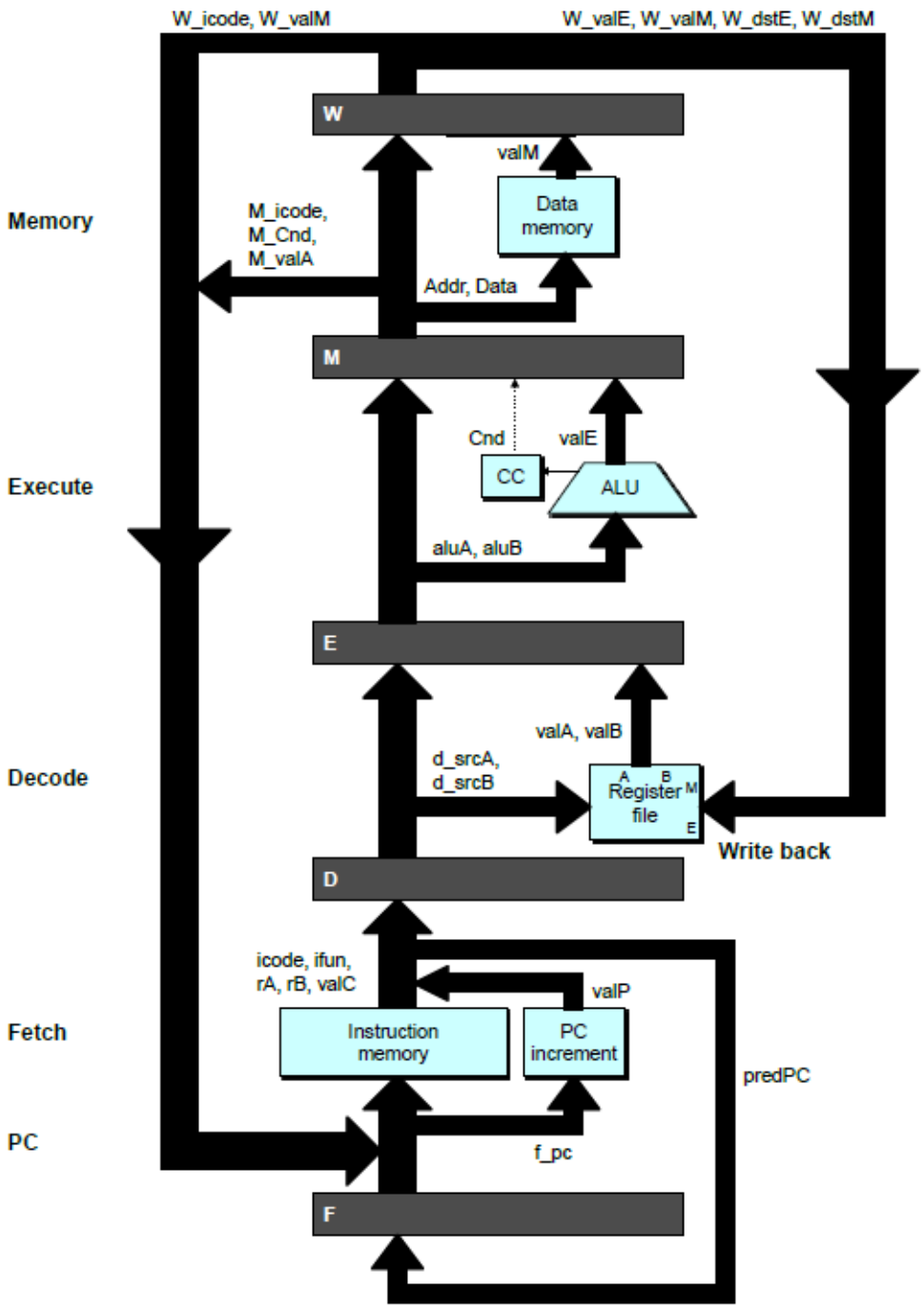
- Operate ALU

Memory

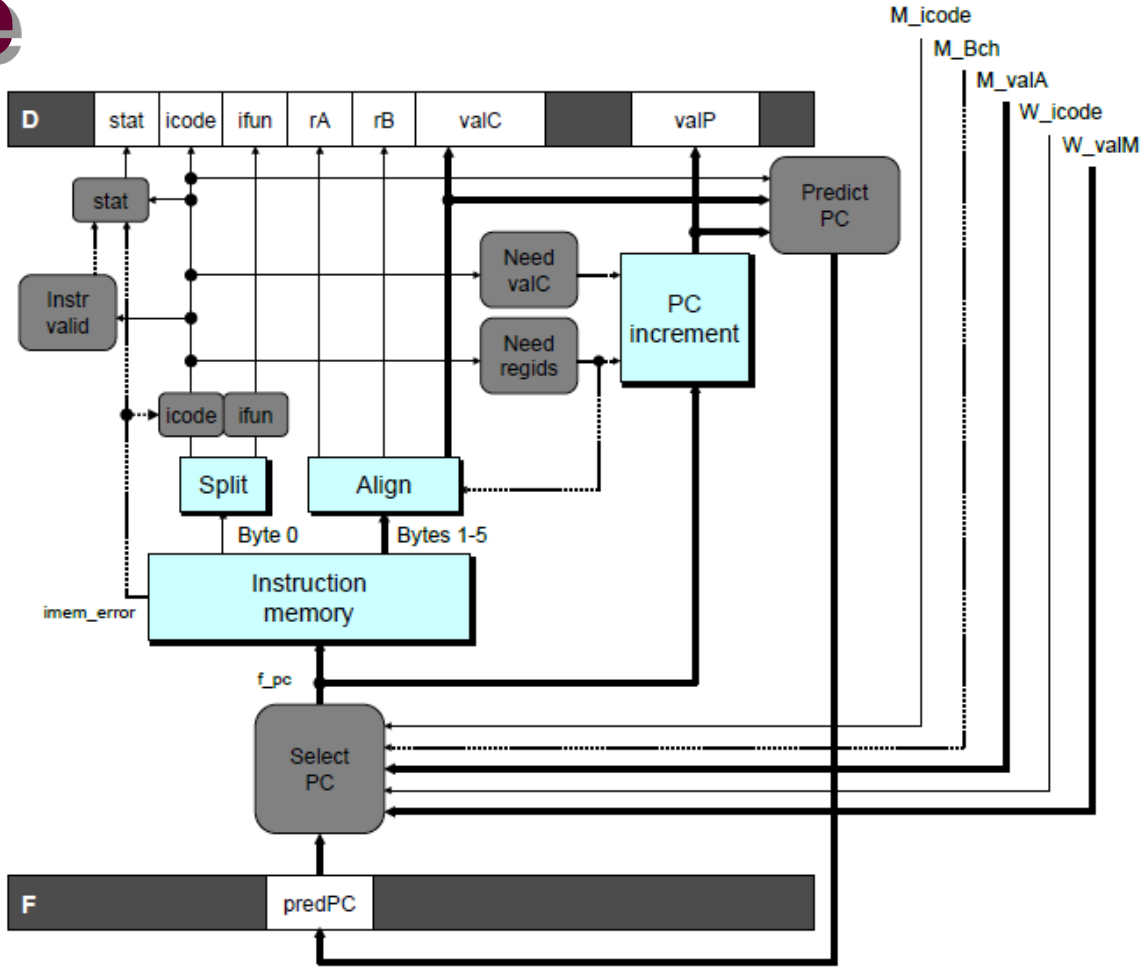
- Read or write data memory

Write Back

- Update register file



Predicting the PC



- Start fetch of new instruction after current one has completed fetch stage
  - Not enough time to reliably determine next instruction
- Guess which instruction will follow
  - Recover if prediction was incorrect

Example: Prediction Strategy

Instructions that Don't Transfer Control

- Predict next PC to be valP
- Always reliable

Call and Unconditional Jumps

- Predict next PC to be valC (destination)
- Always reliable

Conditional Jumps

- Predict next PC to be valC (destination)
- Only correct if branch is taken
  - Typically right 60% of time

Return Instruction

- Don't try to predict

Pipeline Summary

Concept

- Break instruction execution into 5 stages
- Run instructions through in pipelined mode

Limitations

- Can't handle dependencies between instructions when instructions follow too closely
- Data dependencies
  - One instruction writes register, later one reads it
- Control dependency
  - Instruction sets PC in way that pipeline did not predict correctly
  - Mispredicted branch and return

Superscalar Processor

- **Definition:** A superscalar processor can issue and execute *multiple instructions in one cycle*. The instructions are retrieved from a sequential instruction stream and are usually scheduled dynamically.

- Benefit: without programming effort, superscalar processor can take advantage of the *instruction level parallelism* that most programs have

- Most CPUs since about 1998 are superscalar.
- Intel: since Pentium Pro

Superscaler example: Nehalem CPU

- **Multiple instructions can execute in parallel**
  - 1 load, with address computation
  - 1 store, with address computation
  - 2 simple integer (one may be branch)
  - 1 complex integer (multiply/divide)
  - 1 FP Multiply
  - 1 FP Add

- **Some instructions take > 1 cycle, but can be pipelined**

Instruction	*Latency*	*Cycles/Issue*
Load / Store	4	1
Integer Multiply	3	1
Integer/Long Divide	**11--21**	**11--21**
Single/Double FP Multiply	4/5	1
Single/Double FP Add	3	1
Single/Double FP Divide	**10--23**	**10--23**

Loop Unrolling

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    int length = vec_length(v);
    int limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    int i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- Perform 2x more useful work per iteration

Effect of Loop Unrolling

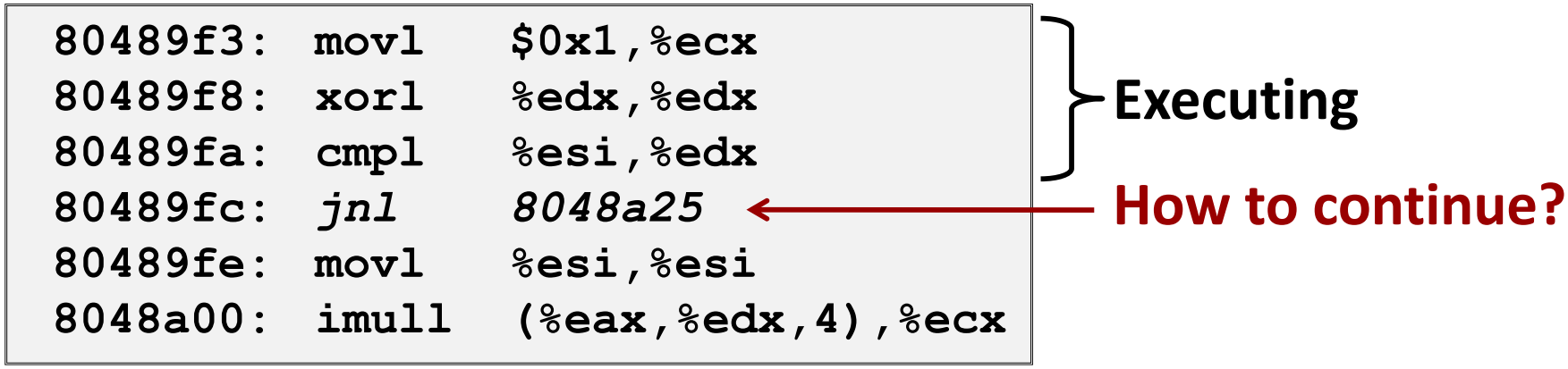
Method	Integer		Double FP	
Operation	Add	Mult	Add	Mult
Combine4	2.0	3.0	3.0	5.0
Unroll 2x	2.0	1.5	3.0	5.0
Latency Bound	1.0	3.0	3.0	5.0

- **Helps integer multiply**
  - below latency bound
  - Compiler does clever optimization
- **Others don't improve.** ***Why?***
  - Still sequential dependency

```
x = (x OP d[i]) OP d[i+1];
```

What About Branches?

- **Challenge**
  - Instruction Control Unit must work well ahead of Execution Unit to generate enough operations to keep EU busy



  - When encounters conditional branch, cannot reliably determine where to continue fetching

Branch Outcomes

- **When encounter conditional branch, cannot determine where to continue fetching**
  - Branch Taken: Transfer control to branch target
  - Branch Not-Taken: Continue with next instruction in sequence
- **Cannot resolve until outcome determined by branch/integer unit**

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
80489fe:  movl   %esi,%esi
8048a00:  imull  (%eax,%edx,4),%ecx
```

Branch Not-Taken

Branch Taken

```
8048a25:  cmpl   %edi,%edx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%ebp),%eax
8048a2c:  leal   0xffffffe8(%ebp),%esp
8048a2f:  movl   %ecx,(%eax)
```

Branch Prediction

- **Idea**
  - Guess which way branch will go
  - Begin executing instructions at predicted position
    - But don't actually modify register or memory data

```
80489f3:  movl   $0x1,%ecx
80489f8:  xorl   %edx,%edx
80489fa:  cmpl   %esi,%edx
80489fc:  jnl    8048a25
. . .
```

Predict Taken

```
8048a25:  cmpl   %edi,%edx
8048a27:  jl     8048a20
8048a29:  movl   0xc(%ebp),%eax
8048a2c:  leal   0xfffffff8(%ebp),%esp
8048a2f:  movl   %ecx,(%eax)
```

Begin Execution

Branch Prediction Through Loop

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 98

Predict Taken (OK)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 99

Predict Taken (Oops)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

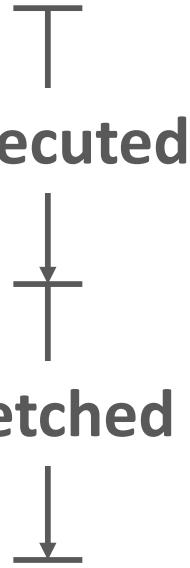
i = 100

Read invalid location

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 101

Assume vector length = 100



Branch Misprediction Invalidation

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 98

Predict Taken (OK)

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
```

i = 99

Predict Taken (Oops)

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~
~~80488b7: cmpl %esi,%edx~~
~~80488b9: jl 80488b1~~

i = 100

~~80488b1: movl (%ecx,%edx,4),%eax~~
~~80488b4: addl %eax,(%edi)~~
~~80488b6: incl %edx~~

i = 101

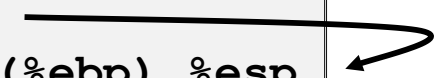
Invalidate

Assume
vector length = 100

Branch Misprediction Recovery

```
80488b1:   movl   (%ecx,%edx,4),%eax
80488b4:   addl   %eax,(%edi)
80488b6:   incl   %edx
80488b7:   cmpl   %esi,%edx
80488b9:   jl     80488b1
80488bb:   leal   0xfffffff8(%ebp),%esp
80488be:   popl   %ebx
80488bf:   popl   %esi
80488c0:   popl   %edi
```

i = 99



Definitely not taken

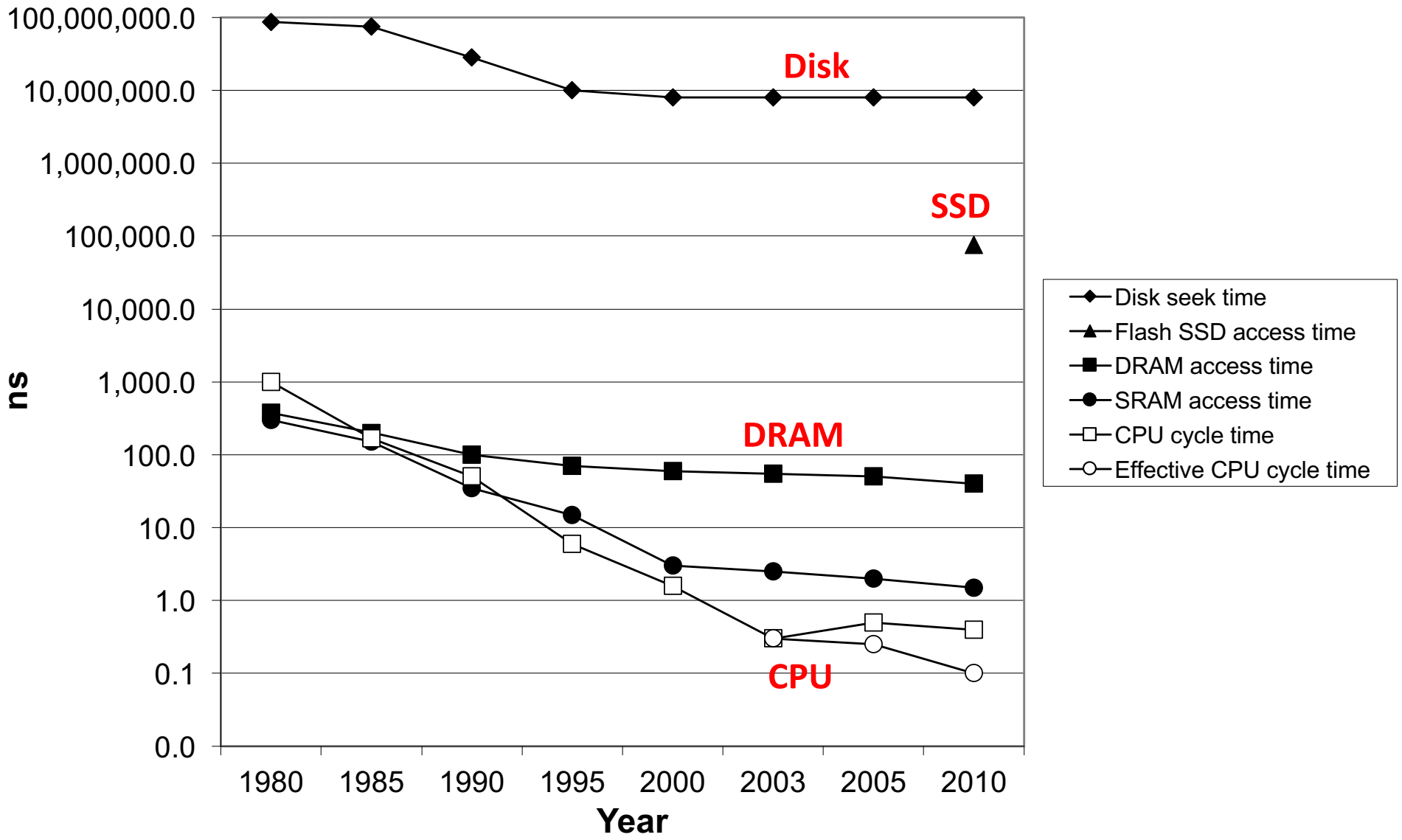
- **Performance Cost**
  - Multiple clock cycles on modern processor
  - Can be a major performance limiter

The Single Core Era
The Memory Hierarchy

- Why have a hierarchy of memory?
- How does it work?

The CPU-Memory Gap

The gap widens between DRAM, disk, and CPU speeds.



Locality Example

```
sum = 0;
for (i = 0; i < n; i++)
    sum += a[i];
return sum;
```

- **Data references**
  - Reference array elements in succession (stride-1 reference pattern). — **Spatial locality**
  - Reference variable `sum` each iteration. — **Temporal locality**
- **Instruction references**
  - Reference instructions in sequence. — **Spatial locality**
  - Cycle through loop repeatedly. — **Temporal locality**

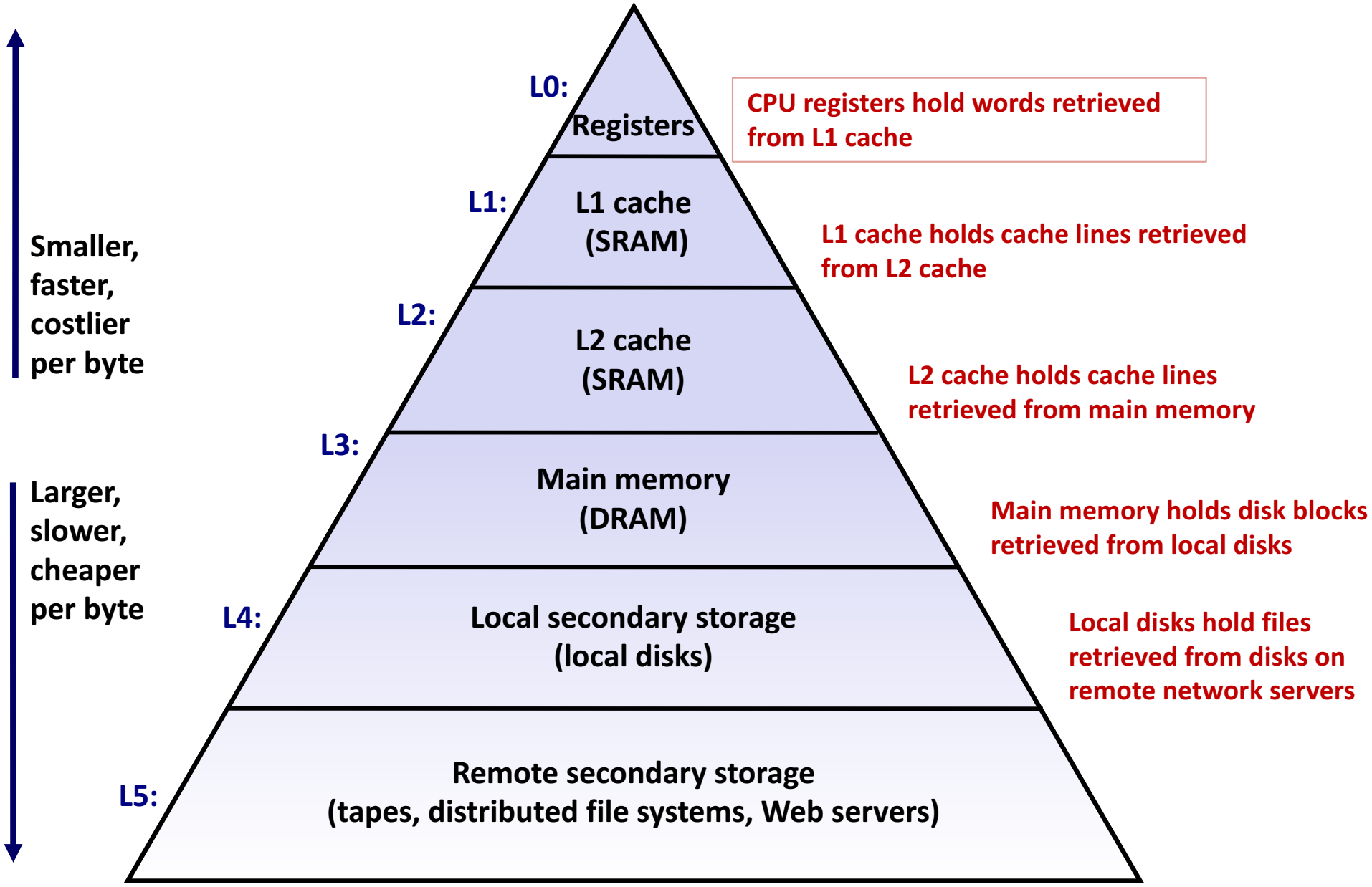
Memory Hierarchies

- **Some fundamental and enduring properties of hardware and software:**
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!).
  - The gap between CPU and main memory speed is widening.
  - Well-written programs tend to exhibit good locality.

- **These fundamental properties complement each other beautifully.**

- **They suggest an approach for organizing memory and storage systems known as a memory hierarchy.**

An Example Memory Hierarchy



Caches

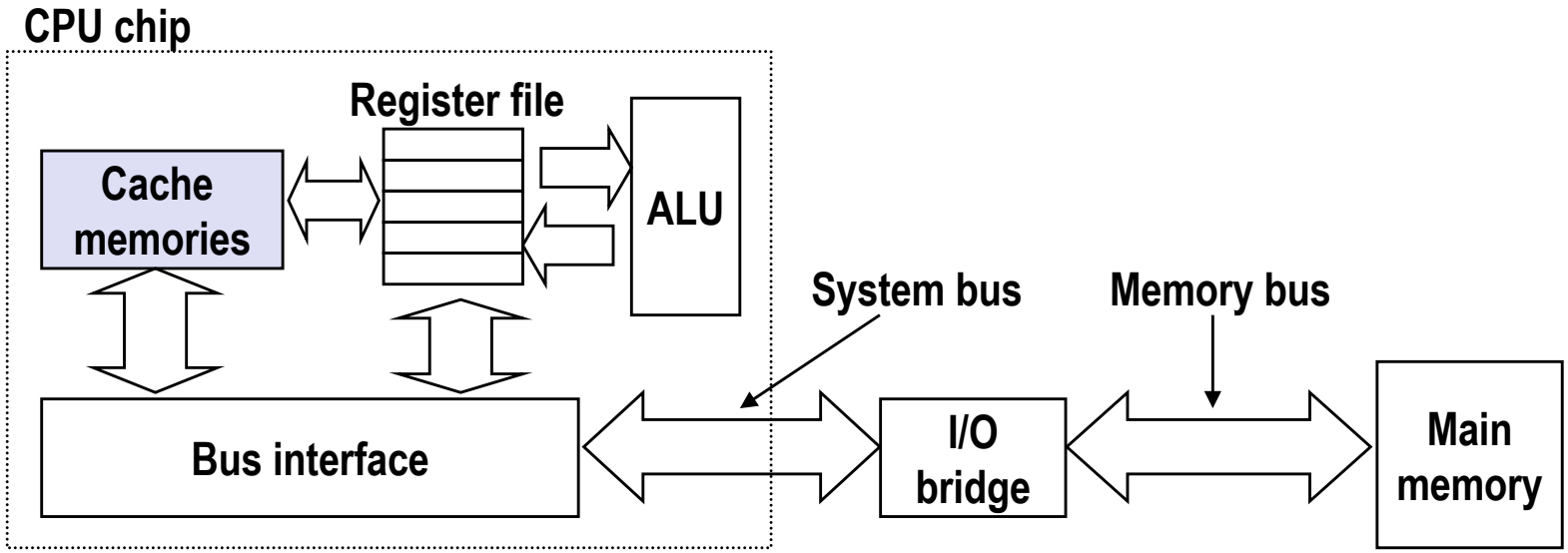
- ***Cache:*** **A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device.**
- **Fundamental idea of a memory hierarchy:**
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k+1.
- **Why do memory hierarchies work?**
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k+1.
  - Thus, the storage at level k+1 can be slower, and thus larger and cheaper per bit.
- ***Big Idea:*** **The memory hierarchy creates a large pool of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top.**

The Memory Hierarchy: Summary

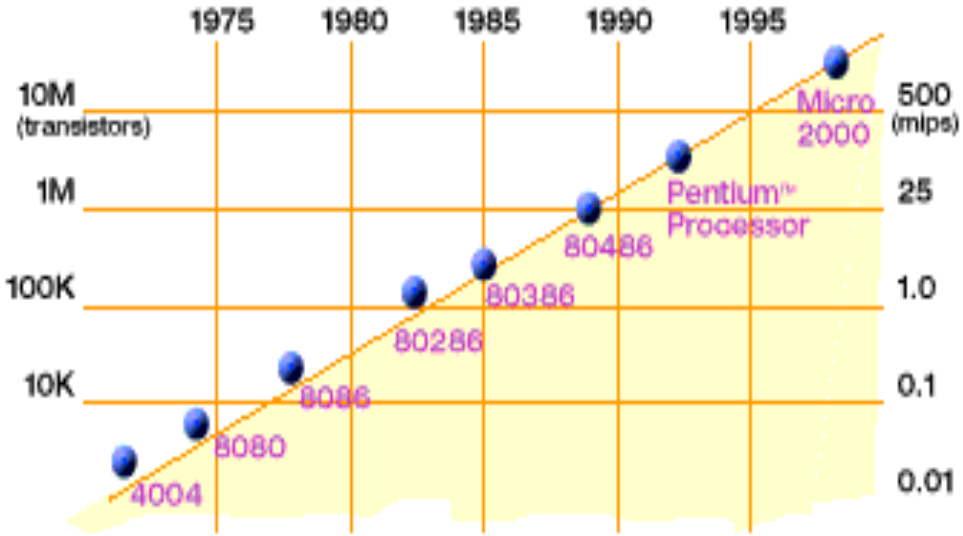
- The speed gap between CPU, memory and mass storage continues to widen.
- Well-written programs exhibit a property called locality.
- Memory hierarchies based on caching close the gap by exploiting locality.

Cache Memories

- **Cache memories are small, fast SRAM-based memories managed automatically in hardware.**
  - Hold frequently accessed blocks of main memory
- **CPU looks first for data in caches (e.g., L1, L2, and L3), then in main memory.**
- **Typical single core system structure:**

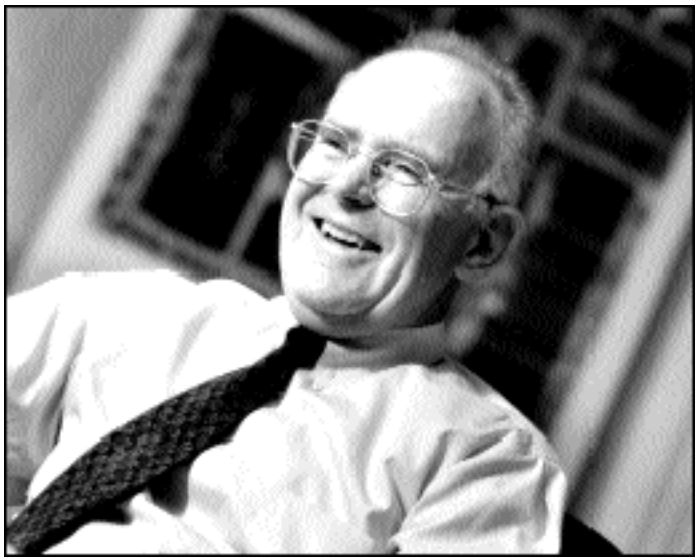


Technology Trends: Microprocessor Capacity



2X transistors/Chip Every 1.5 years

Called "Moore's Law"



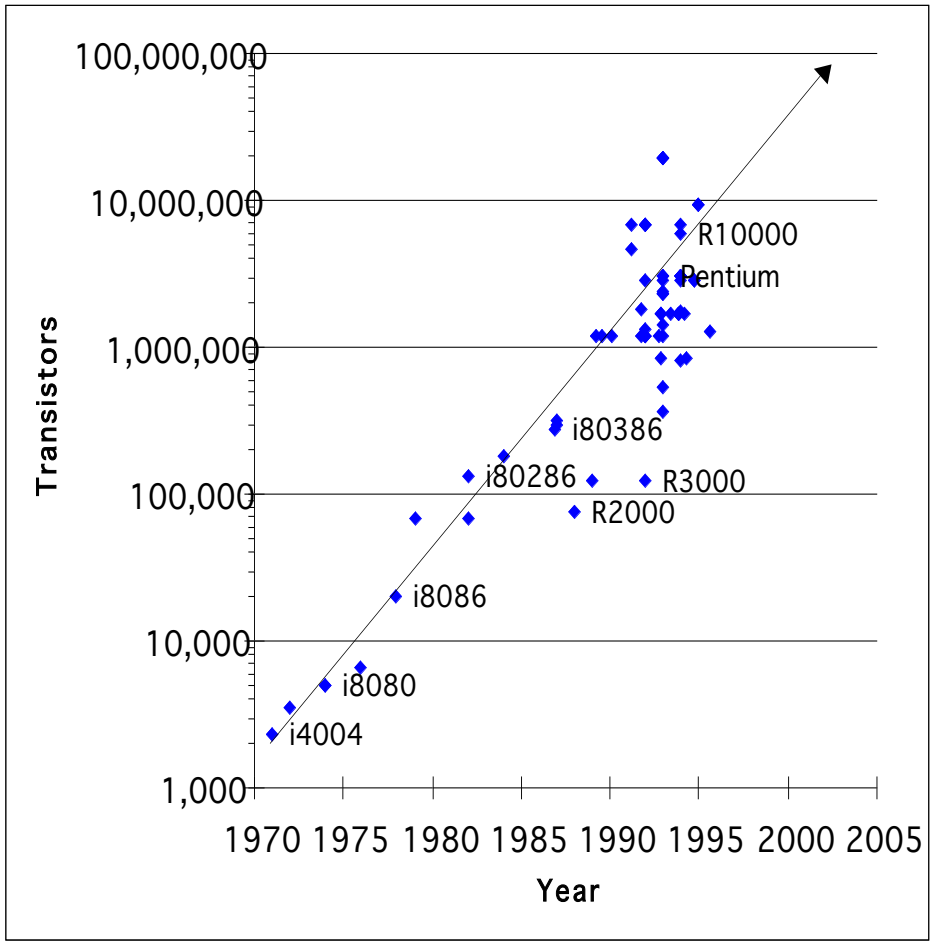
Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

Microprocessors have become smaller, denser, and more powerful.

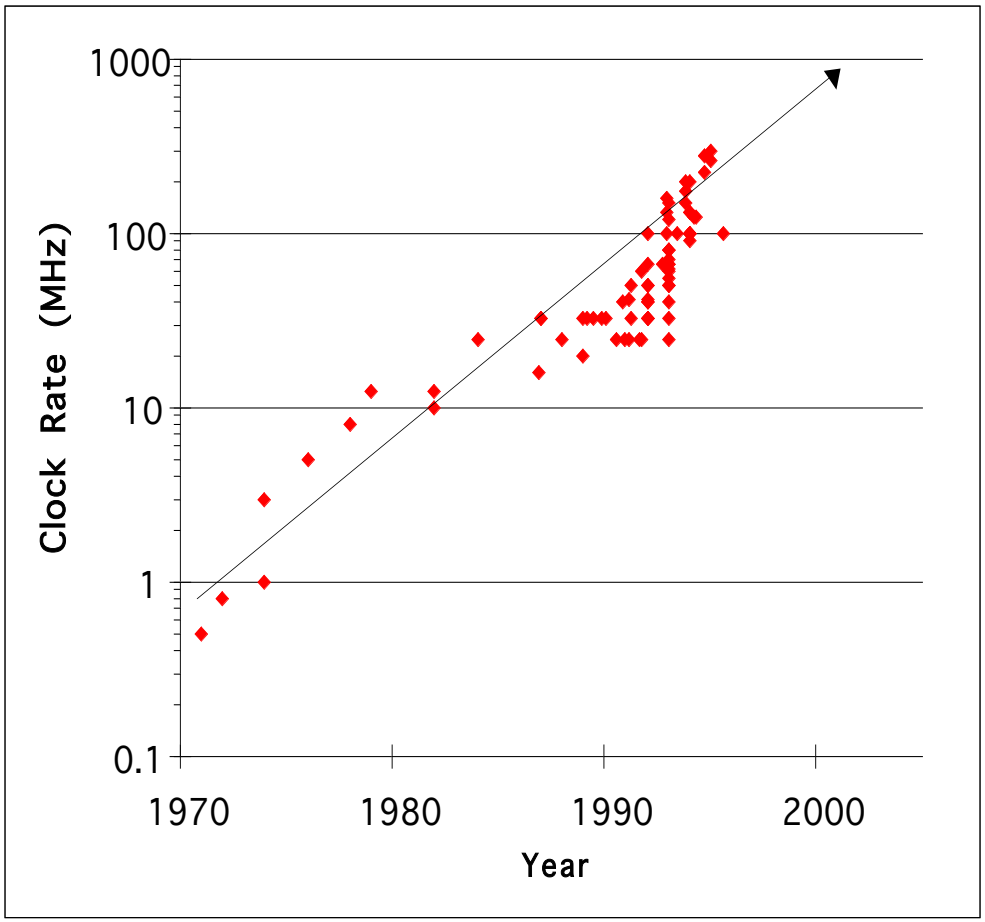
Slide source: Jack Dongarra

Microprocessor Transistors and Clock Rate

Growth in transistors per chip



Increase in clock rate



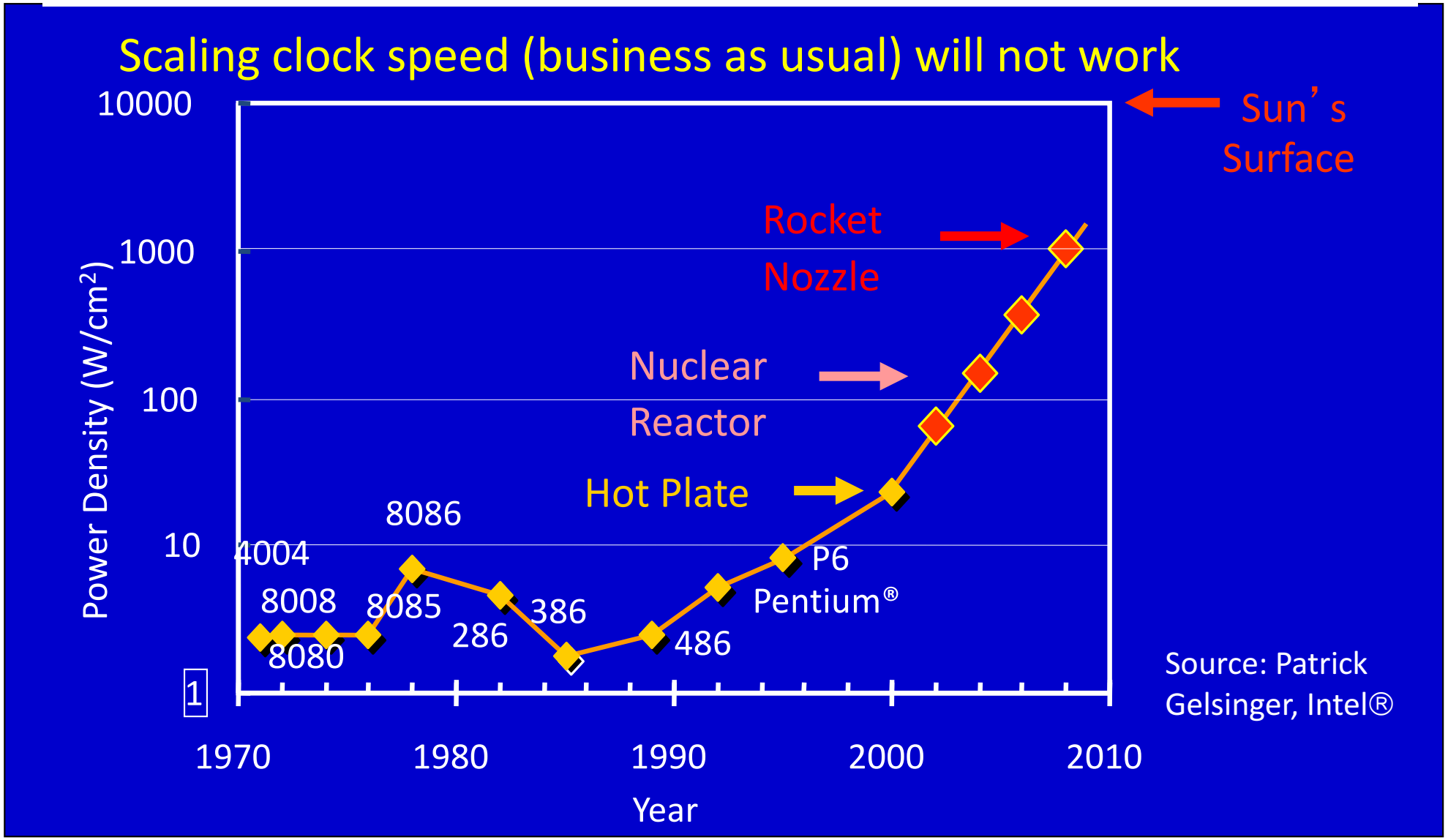
Why bother with parallel programming? Just wait a year or two…

Portland State UNIVERSITY

Limit #1: Power density

Can soon put more transistors on a chip than can afford to turn on.

-- Patterson '07



Parallelism Saves Power

- Exploit explicit parallelism for reducing power

$$\text{Power} = (C * V^2 * F)/4 \qquad \text{Performance} = (\text{Cores} * F)*1$$

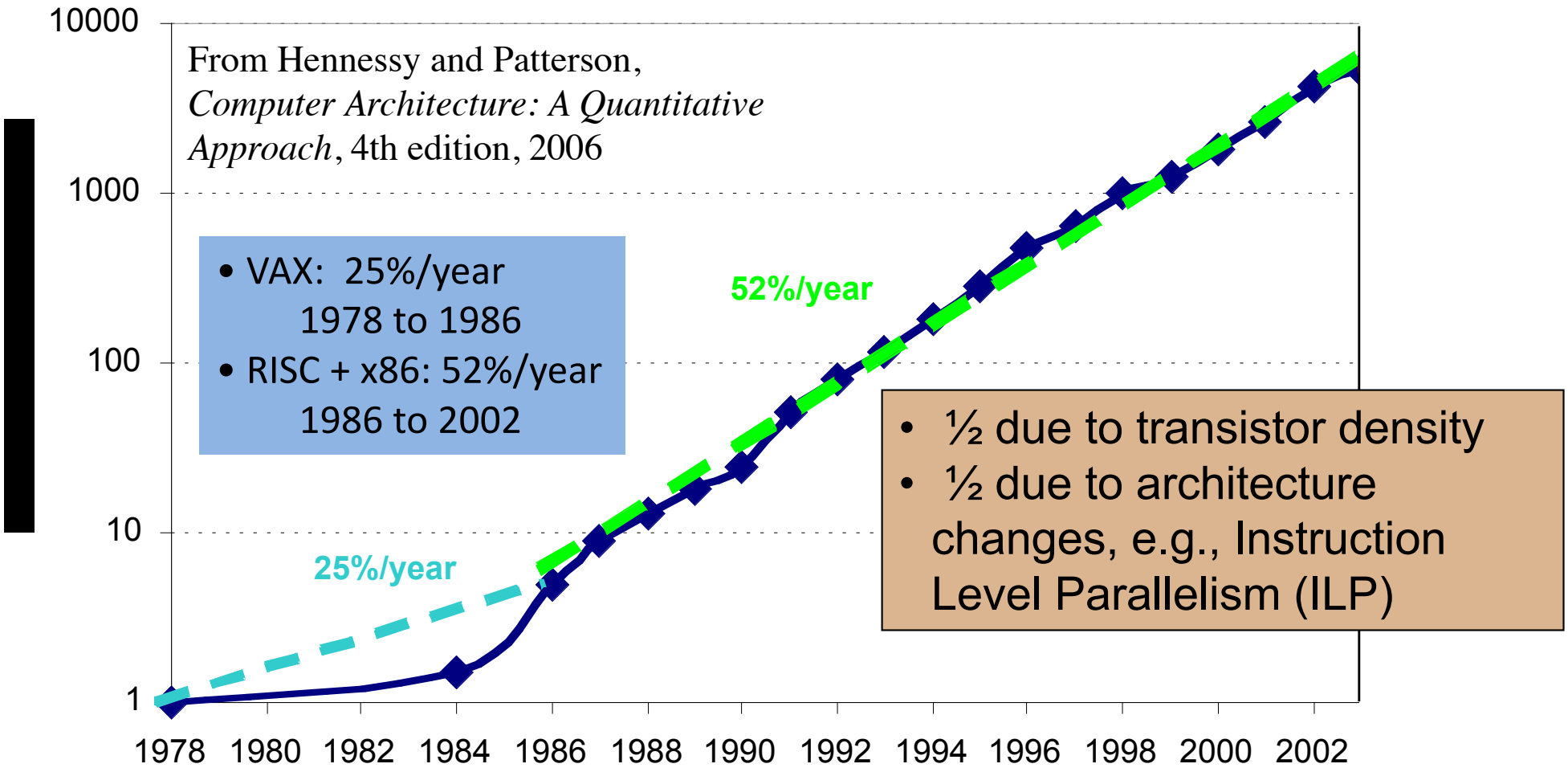
Capacitance Voltage Frequency

- Using additional cores
  - Increase density (= more transistors = more capacitance)
  - Can increase cores (2x) and performance (2x)
  - Or increase cores (2x), but decrease frequency (1/2): same performance at ¼ the power

- Additional benefits
  - Small/simple cores → more predictable performance

Limit #2: Hidden Parallelism Tapped Out

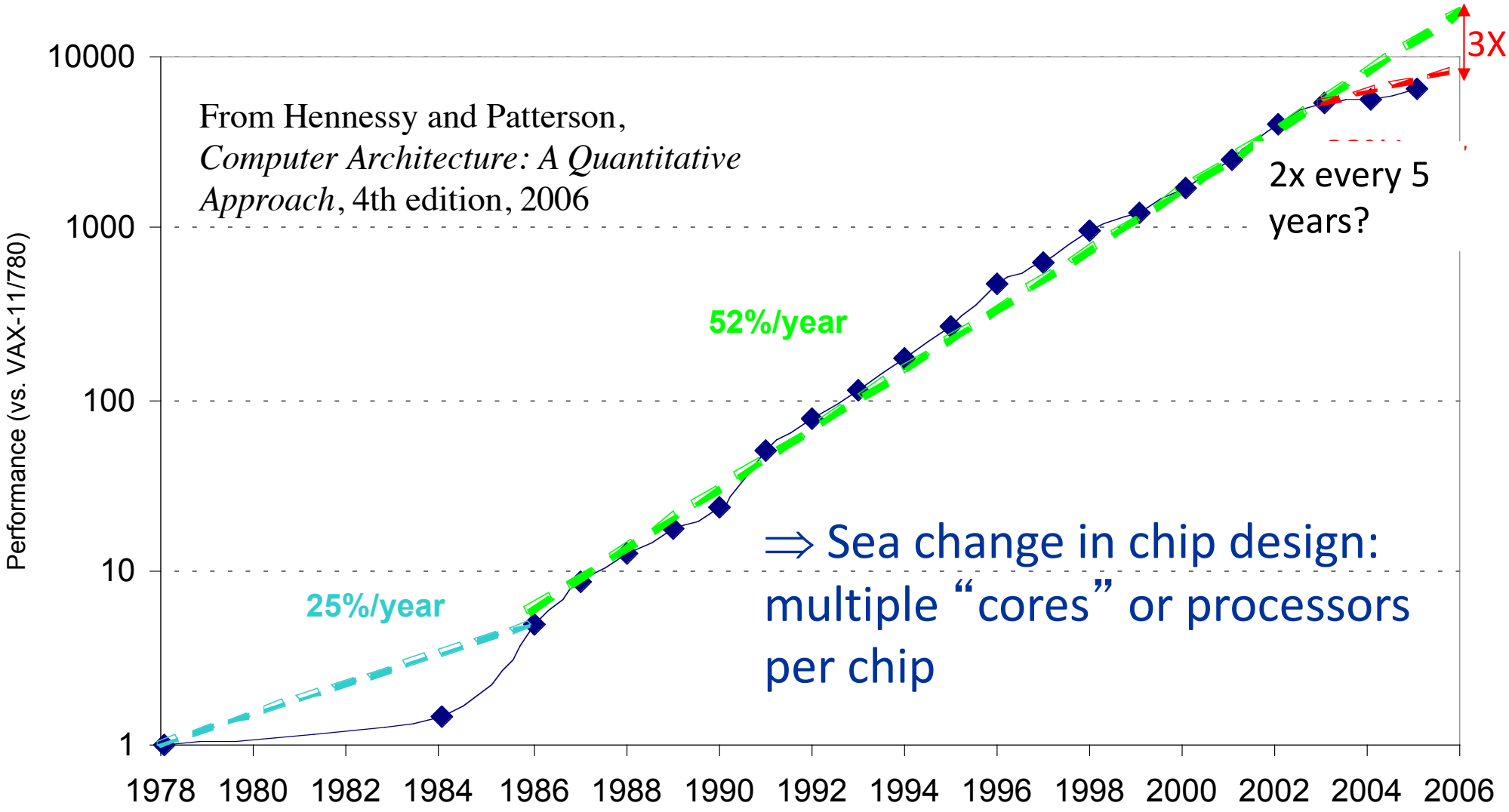
Application performance was increasing by 52% per year as measured by the SpecInt benchmarks here



Limit #2: Hidden Parallelism Tapped Out

- **Superscalar (SS) designs were the state of the art; many forms of parallelism not visible to programmer**
  - **multiple instruction issue**
  - **dynamic scheduling: hardware discovers parallelism between instructions**
  - **speculative execution: look past predicted branches**
  - **non-blocking caches: multiple outstanding memory ops**
- **Unfortunately, these sources have been used up**

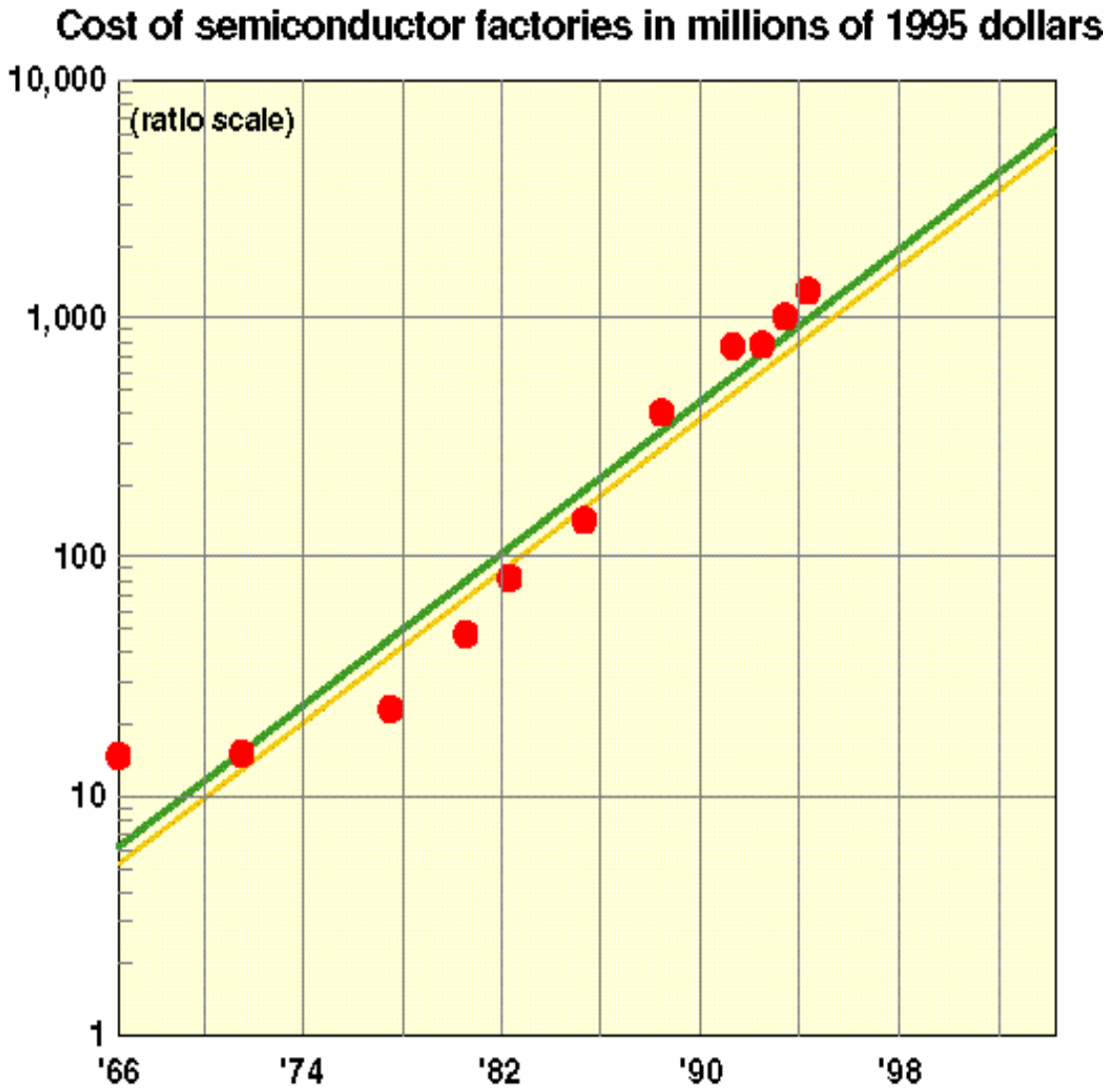
Uniprocessor Performance (SPECint) Today



- VAX : 25%/year 1978 to 1986
- RISC + x86: 52%/year 1986 to 2002
- RISC + x86: ??%/year 2002 to present

Limit #3: Chip Yield

Manufacturing costs and yield problems limit use of density

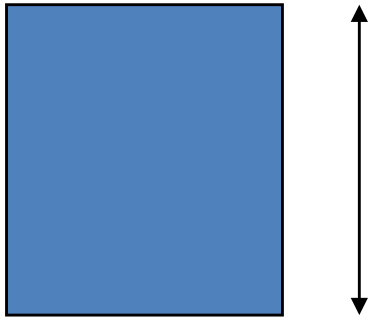


- Moore's (Rock's) 2nd law: fabrication costs go up
- Yield (% usable chips) drops

- Parallelism can help
  - More smaller, simpler processors are easier to design and validate
  - Can use partially working chips:
  - E.g., Cell processor (PS3) is sold with 7 out of 8 "on" to improve yield

Limit #4: Speed of Light (Fundamental)

1 Tflop/s, 1 Tbyte sequential machine

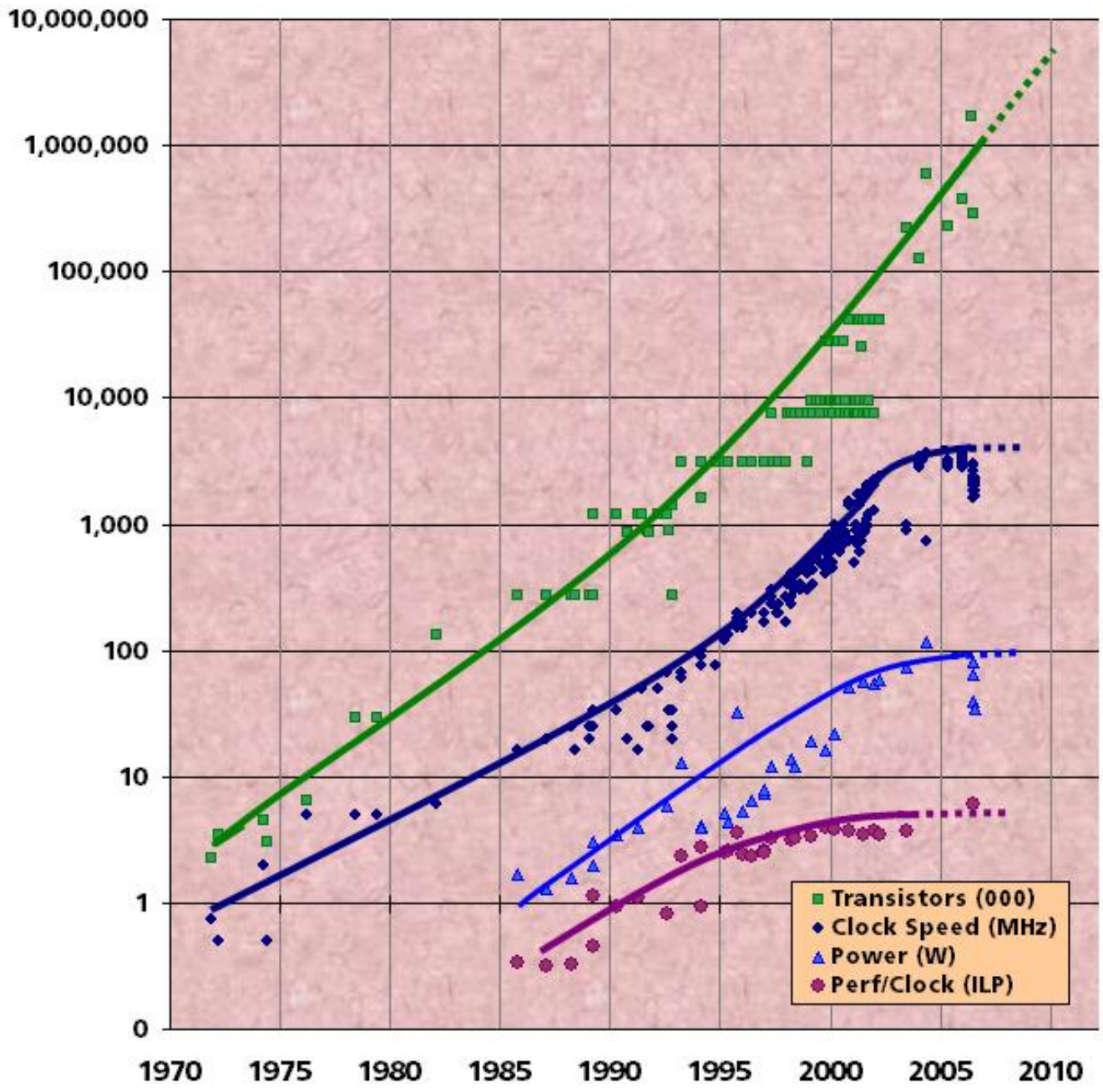


r = 0.3 mm

- Consider the 1 Tflop/s sequential machine:
  - Data must travel some distance, r, to get from memory to CPU.
  - To get 1 data element per cycle, this means 10^{12} times per second at the speed of light, c = $3x10^8$ m/s. Thus r < $c/10^{12}$ = 0.3 mm.
- Now put 1 Tbyte of storage in a 0.3 mm x 0.3 mm area:
  - Each bit occupies about 1 square Angstrom, or the size of a small atom.
- No choice but parallelism

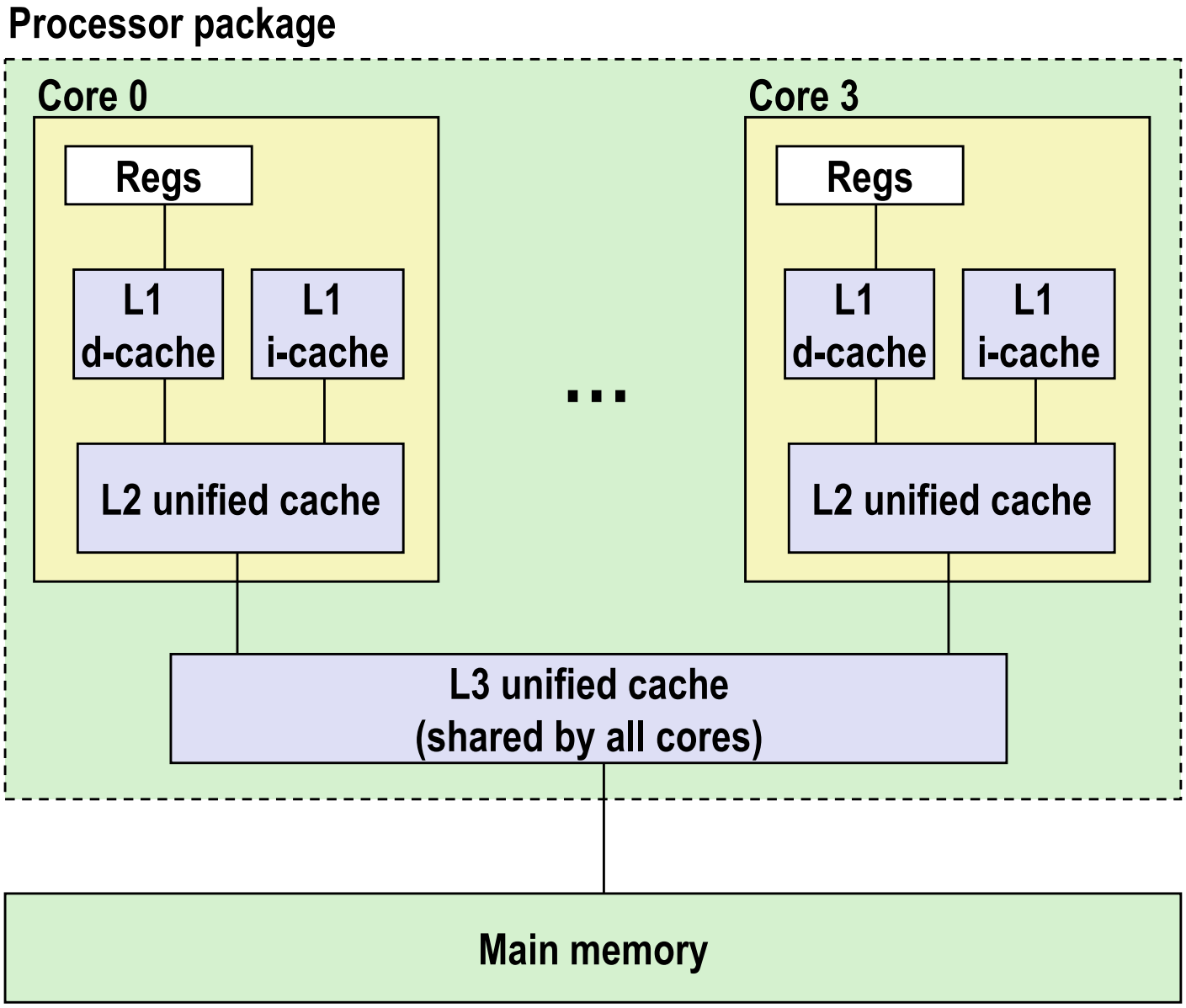
Thus, the Multicore Era

- Chip density is continuing increase ~2x every 2 years*
  - Clock speed is not
  - Number of processor cores may double instead
- There is little or no hidden parallelism (ILP) to be found
- Parallelism must be exposed to and managed by software



Source: Intel, Microsoft (Sutter) and Stanford (Olukotun, Hammond)

Intel Core i7 Cache Hierarchy



L1 i-cache and d-cache:
32 KB, 8-way,
Access: 4 cycles

L2 unified cache:
256 KB, 8-way,
Access: 11 cycles

L3 unified cache:
8 MB, 16-way,
Access: 30-40 cycles

Block size: 64 bytes for all caches.

What is Manycore ?

- What if we use all of the transisters on a chip for as many cores as we can fit??
- Beyond the edge of number of cores in common "multicore" architectures
- Dividing line is not clearly defined
- Active research, now in embedded & clusters
- Examples:
  - NVIDIA Fermi Graphics Processing Unit (GPU)
    - First model: 32 "CUDA cores" per SM, 16 SMs
      - (SM = "streaming multiprocessor")
    - K20 model: 2496 CUDA cores, peak 3.52 Tflops

Ex: NVIDIA Fermi

Portland State UNIVERSITY

What is Manycore ?

- Examples (cont'd)
  - Intel Xeon Phi coprocessor and Knights Landing
    - Up to 61 cores
    - Example: Tianhe-2 Supercomputer has 32,000 multicore central processing units (CPUs) and 48,000 coprocessors ("accelerators"); peak 33.86 Petaflops
  - Tilera Tile-Gx
    - 100 cores, 64-bit