CS3350B Computer Organization Introduction

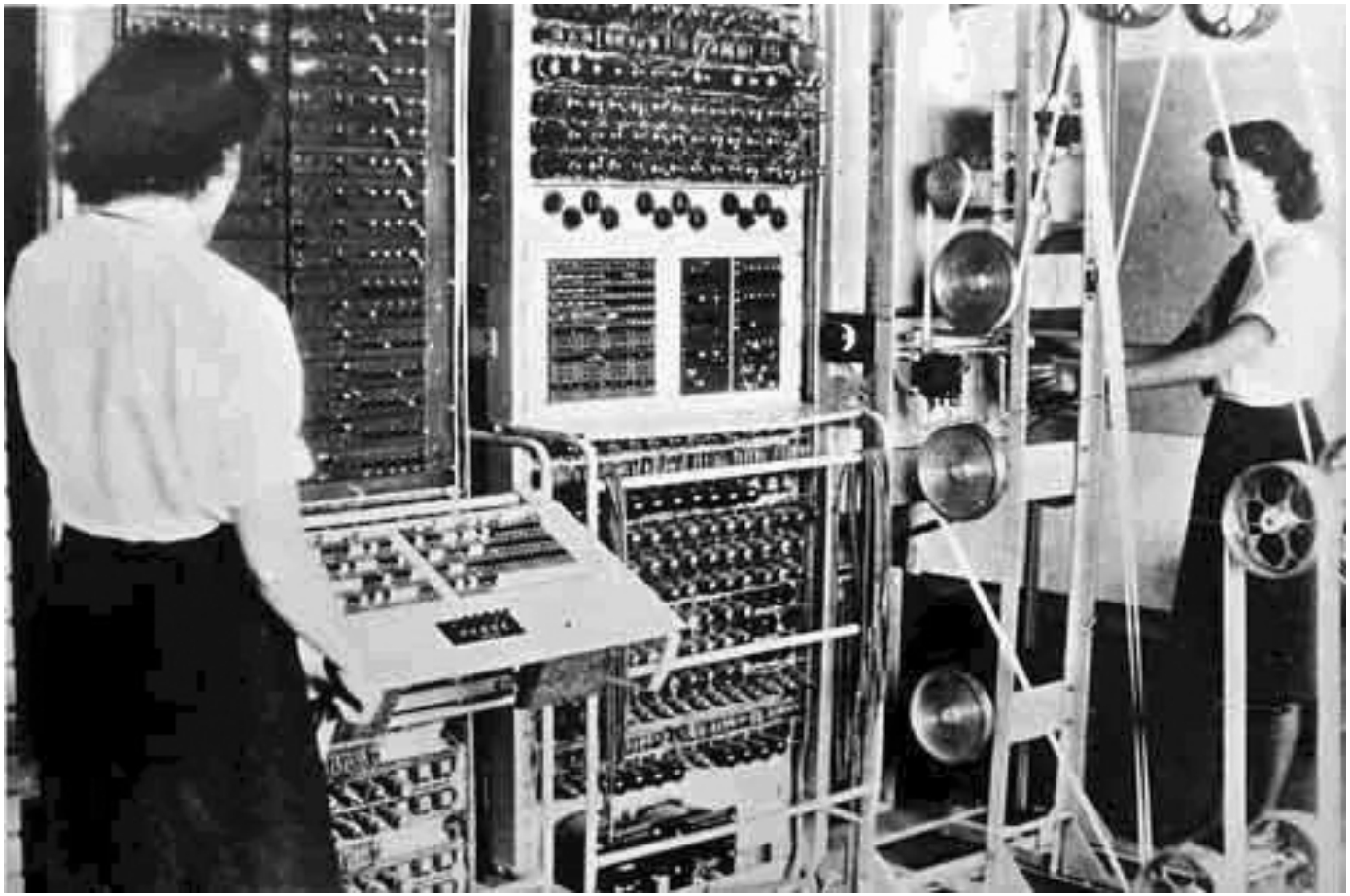
Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

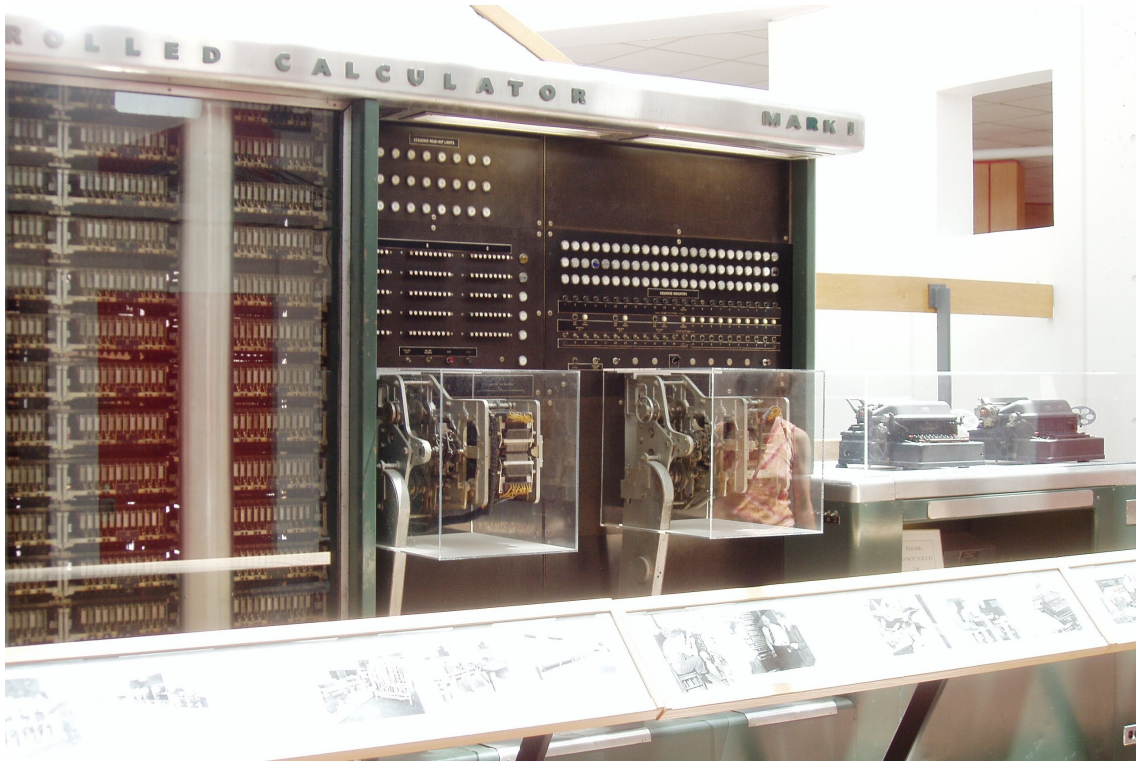
Tuesday January 8, 2019

Outline

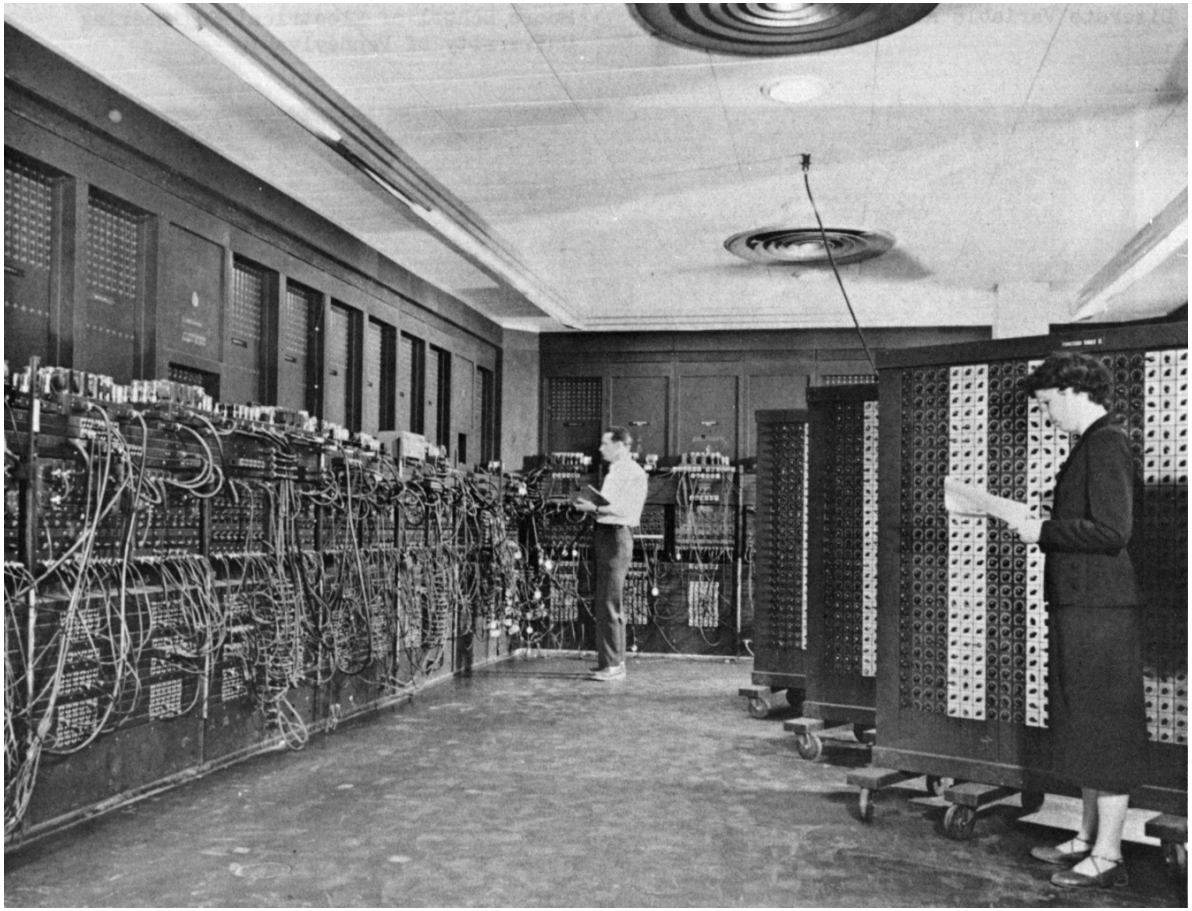
Konrad Zuse's Z3 electro-mechanical computer (1941, Germany). Turing complete, though conditional jumps were missing.



Colossus (UK, 1941) was the world's first totally electronic programmable computing device. But not Turing complete.



Harvard Mark I – IBM ASCC (1944, US). Electro-mechanical computer. No conditional jumps and not Turing complete. It could store 72 numbers, each 23 decimal digits long. It could do three additions or subtractions in a second. A multiplication took six seconds, a division took 15.3 seconds, and a logarithm or a trigonometric function took over one minute.

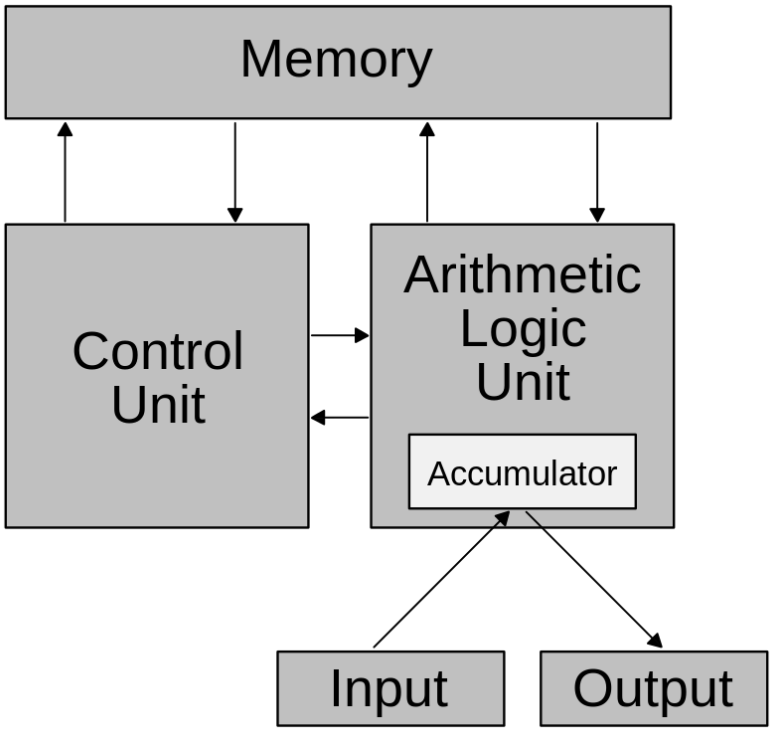


Electronic Numerical Integrator And Computer (ENIAC, 1945). The first general-purpose, electronic computer. It was a Turing-complete, digital computer capable of being reprogrammed and was running at 5,000 cycles per second for operations on the 10-digit numbers.

The IBM Personal Computer (IBM PC) (Introduced on August 12, 1981).

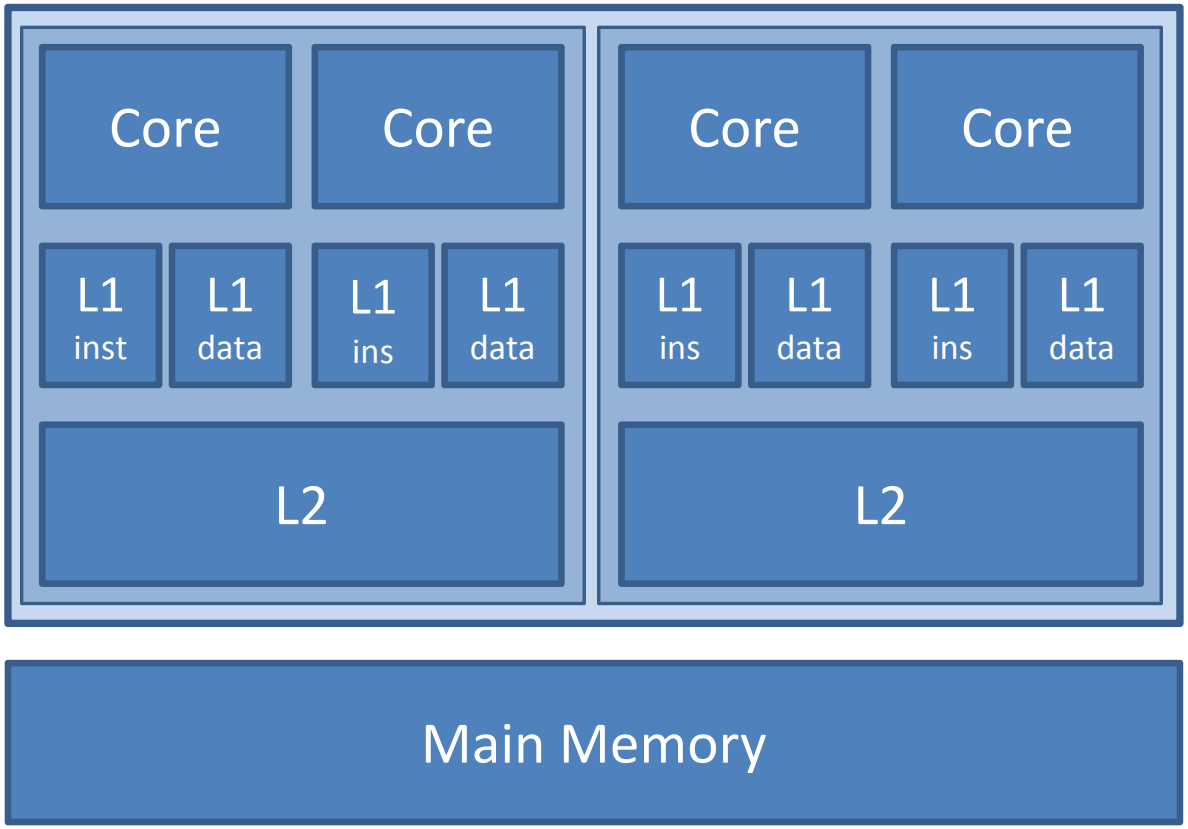
Outline

Von Neumann Architecture



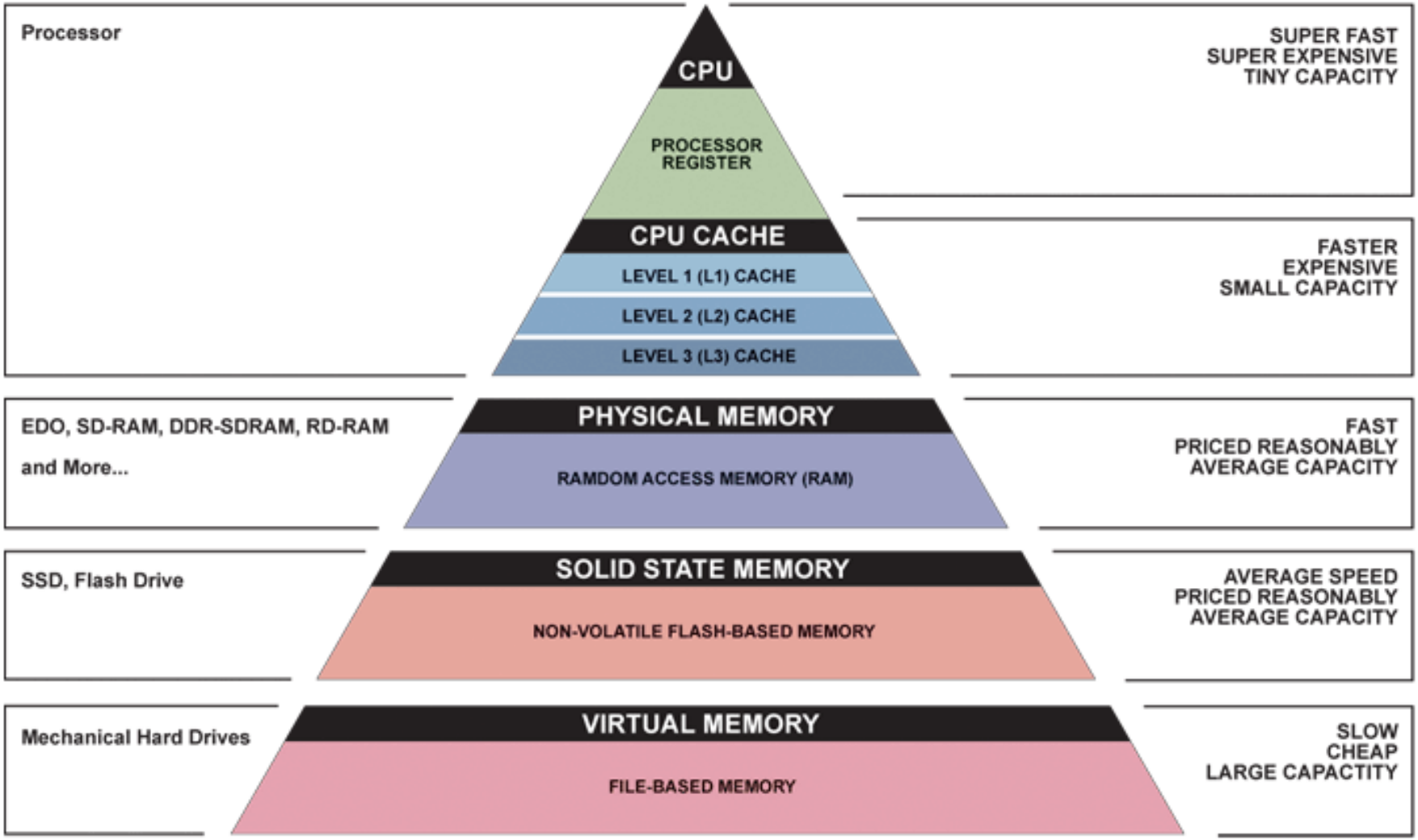
The Von Neumann architecture saw a shared memory for instructions and data. Modern computers, especially w.r.t what software "sees", use his model.

Multi-core Architecture



Modern multi-core processors see a varying hierarchy of independent and shared memories.

Memory Hierarchy



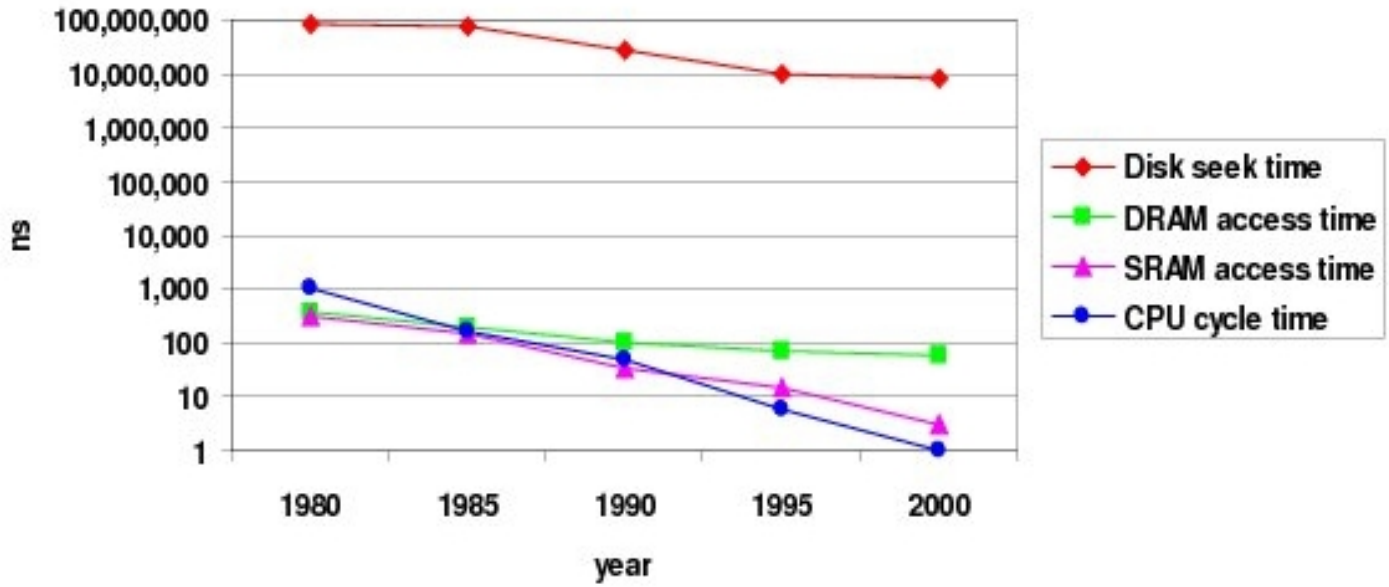
Cache Memory Statistics

L1 Data Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L1 Instruction Cache			
Size	Line Size	Latency	Associativty
32 KB	64 bytes	3 cycles	8-way
L2 Cache			
Size	Line Size	Latency	Associativty
6 MB	64 bytes	14 cycles	24-way

Typical cache specifications of a multicore in 2008.

The CPU-Memory Gap

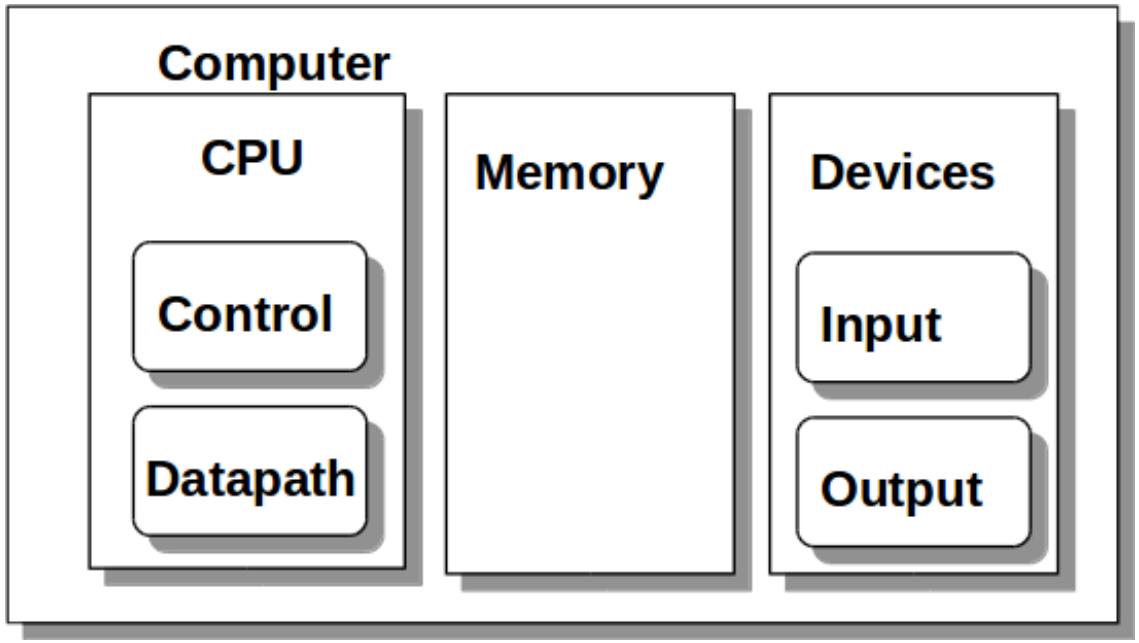
The increasing gap between DRAM, disk, and CPU speeds.



The **Processor-Memory Gap** is a key contributor to the **Memory Wall** – the point where a program's performance is totally determined by memory speed. Faster processors will not make programs run faster!

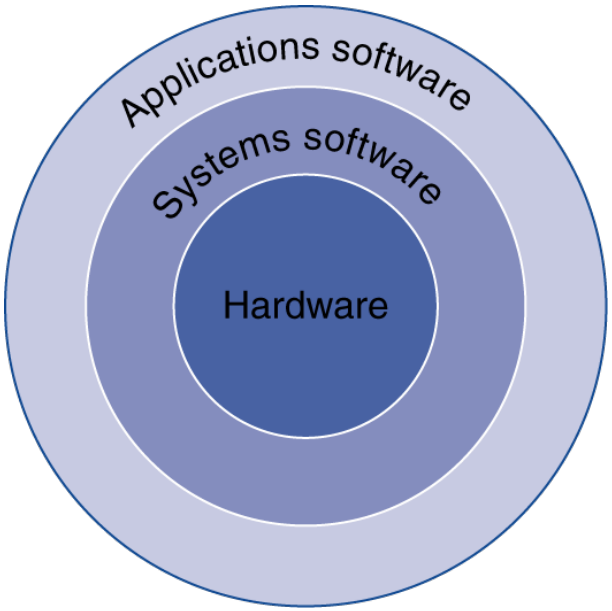
Outline

Components of a computer



- Same components for all kinds of computer
  - ↳ desktop, server, supercomputer, embedded

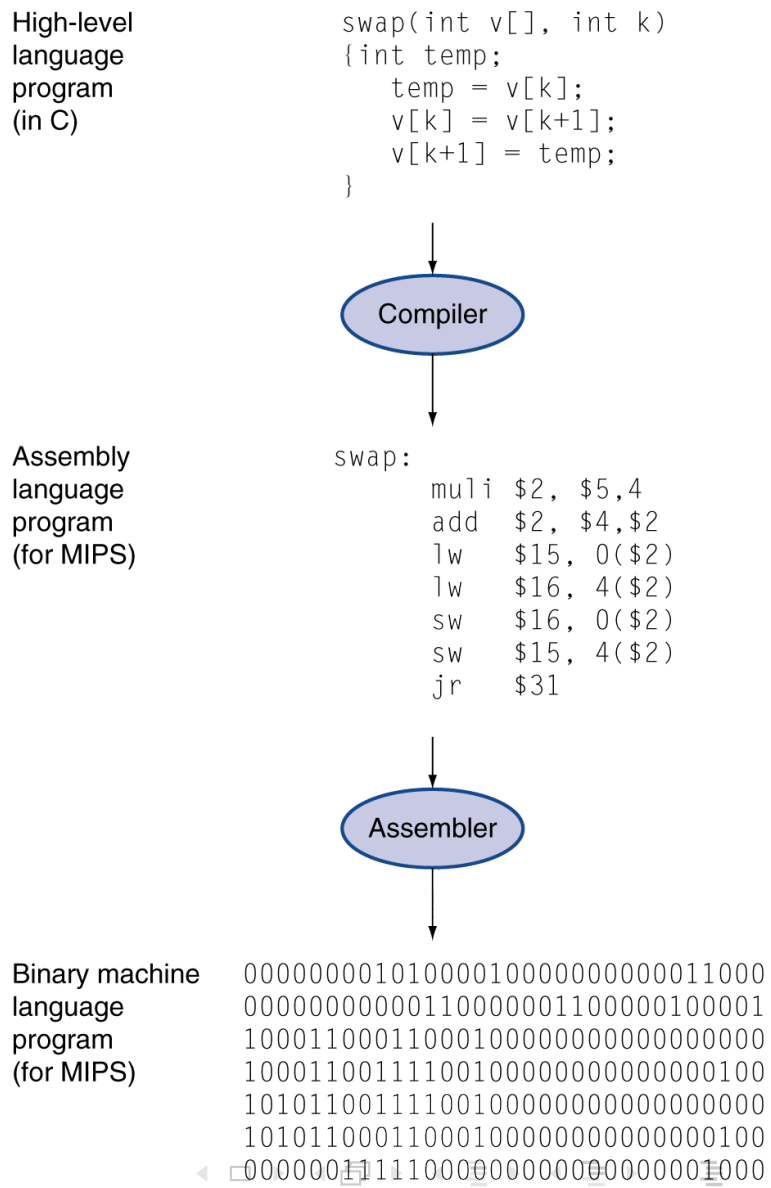
Layers of Software



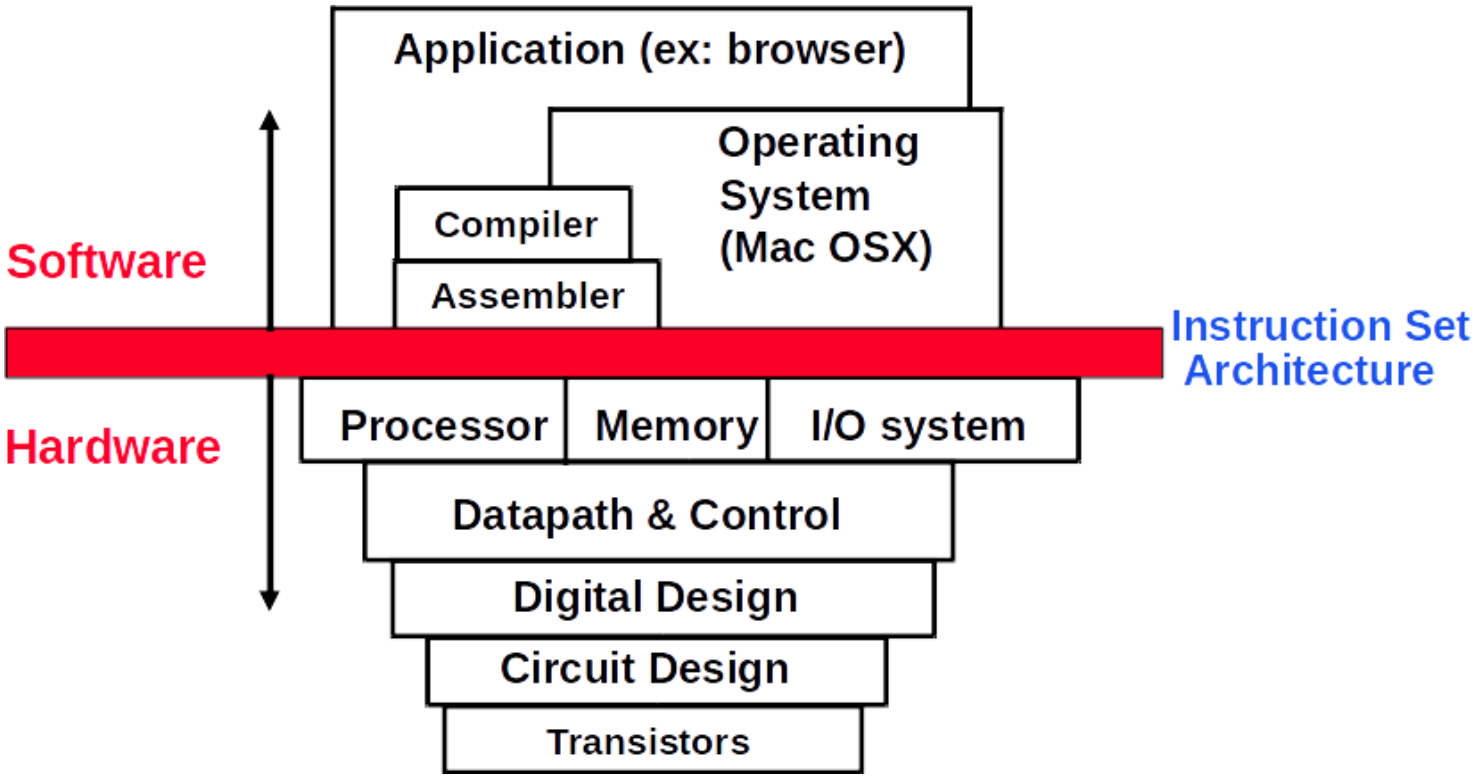
- **Application software**
  - ↳ Written in a high-level language (HLL)
  - ↳ Javascript, Python, Java.
- **System software**
  - ↳ Compiler: translates HLL code to machine code
  - ↳ Operating system: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- **Hardware**
  - ↳ Processor, memory, I/O controllers

Levels of Program Code

- **High-level language**
  - ↳ Level of abstraction closer to problem domain
  - ↳ Provides productivity and portability
- **Assembly language**
  - ↳ Textual representation of machine instructions
- **Hardware representation**
  - ↳ Binary encoding of instructions and data



Layers of Abstraction in Computer Systems



After a brief look at processors and memory, we will start down at circuit design and build up to ISA.

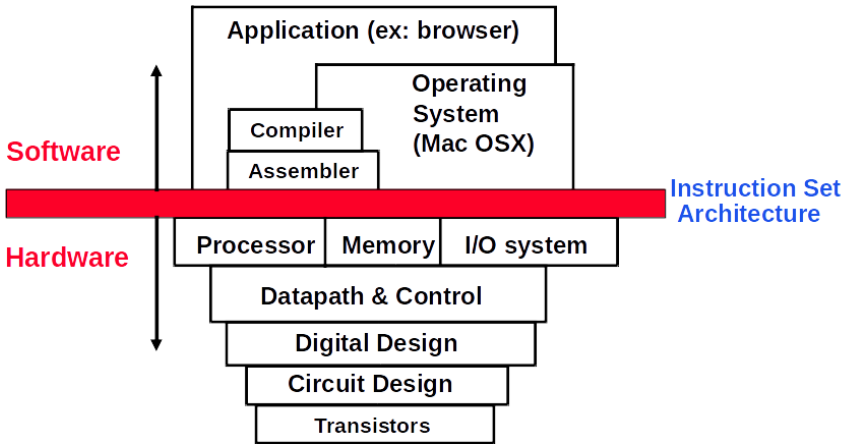
Complicated Systems are Inevitable

Think economic specialization.

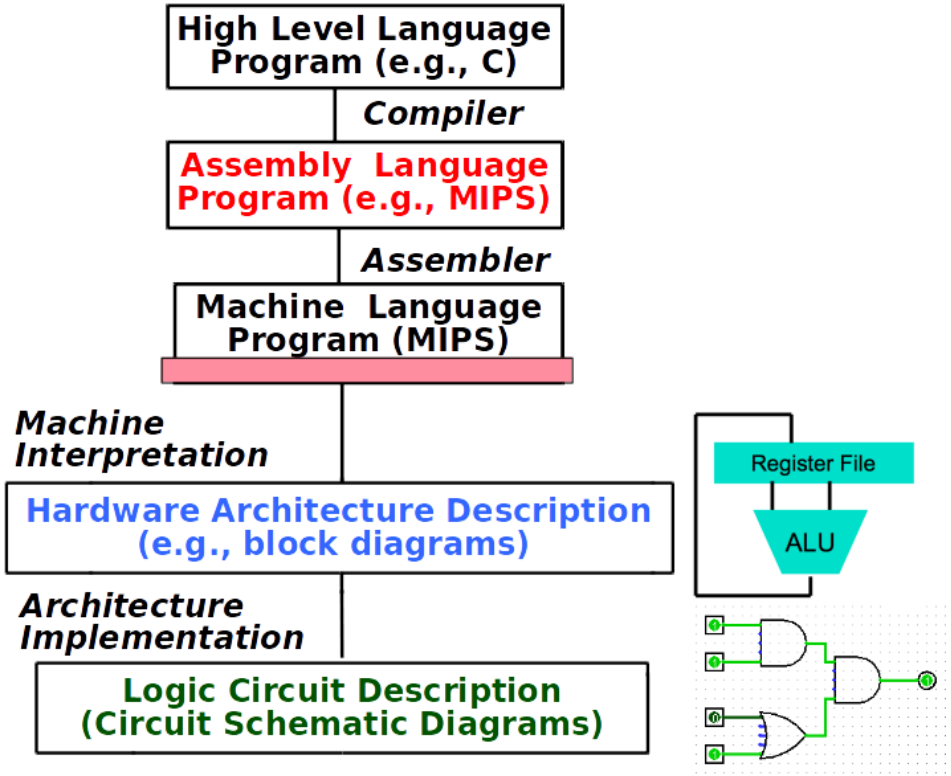
Each layer is separate and can be optimized independently in the pursuit of performance.

Eight great ideas in computer architecture (Patterson)

1. Use abstraction to simplify design
2. Design for Moore's Law
3. Make the common case fast
4. Performance via parallelism
5. Performance via pipelining
6. Performance via prediction
7. Hierarchy of memories
8. Dependability via redundancy



Great Idea #1: Abstraction




```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw  $t1, 4($2)
sw  $t1, 0($2)
sw  $t0, 4($2)
```

```
# Anything can be represented as a
# number, i.e., data or instructions
```

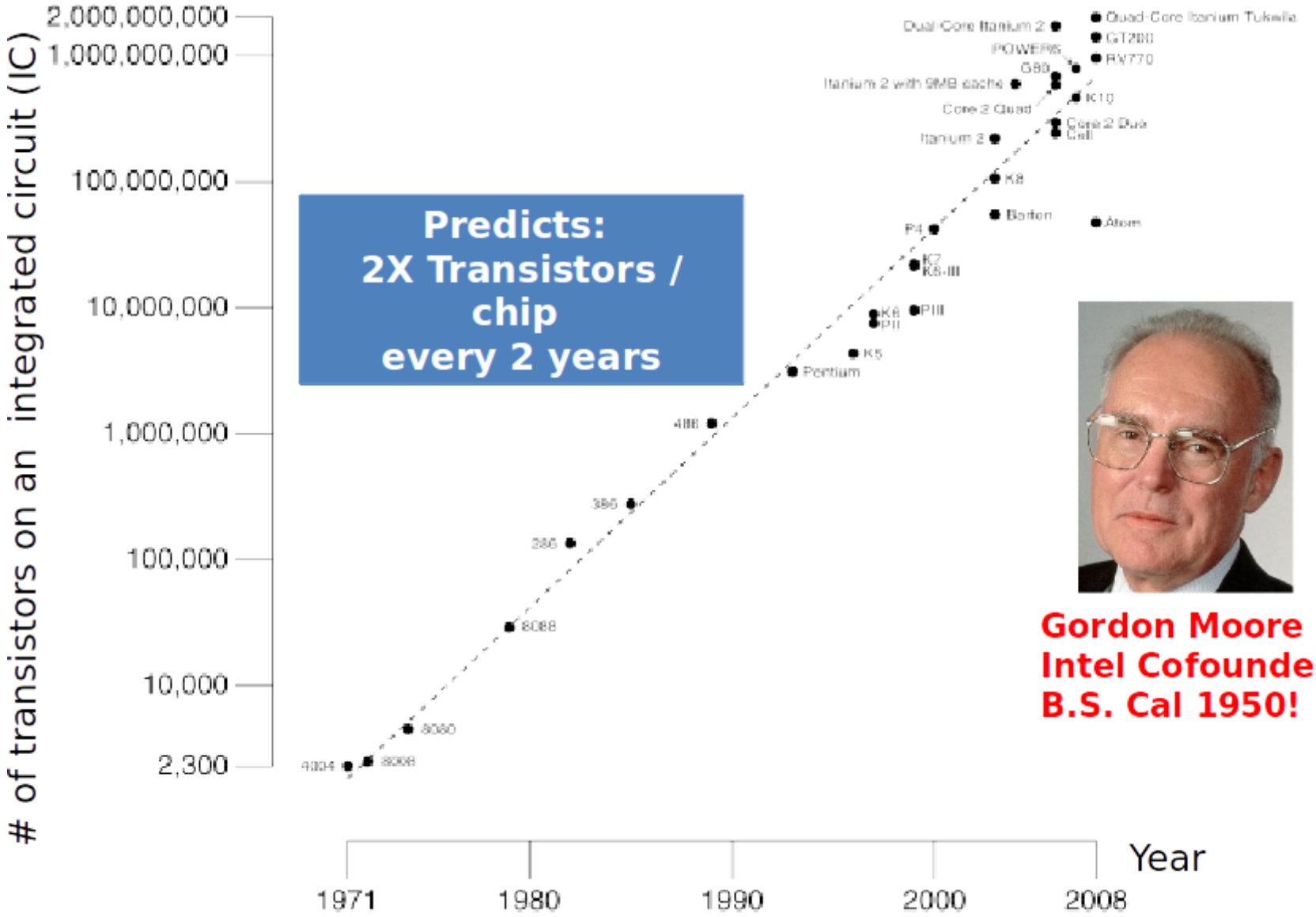
0000 1001 1100 0110 1010 1111 0101 1000

1010 1111 0101 1000 0000 1001 1100 0110

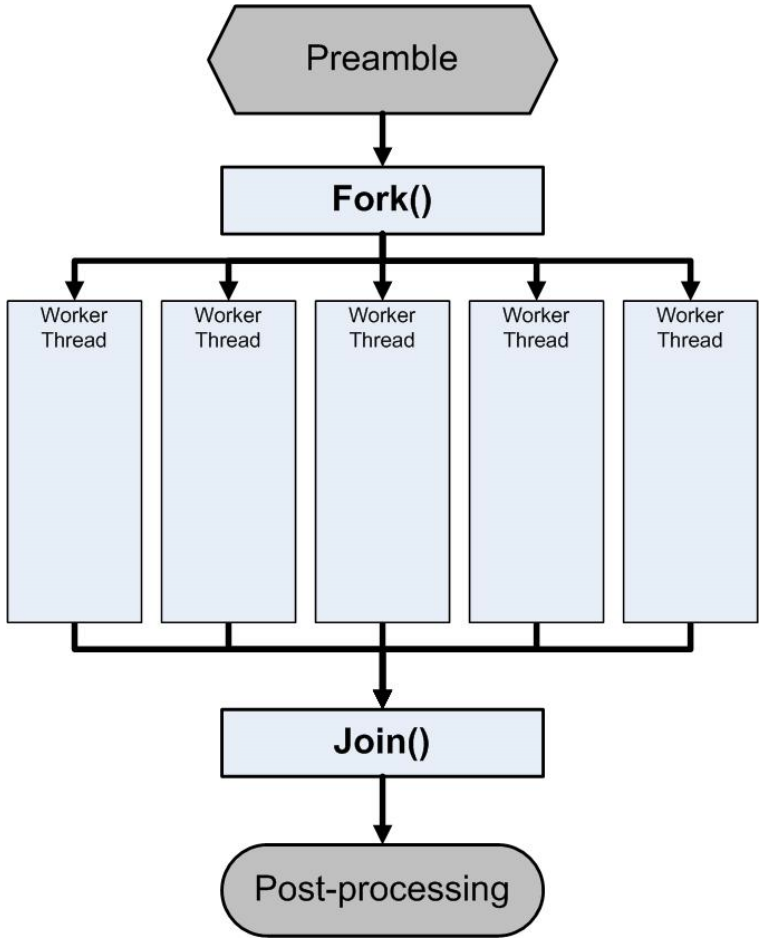
1100 0110 1010 1111 0101 1000 0000 1001

0101 1000 0000 1001 1100 0110 1010 1111

Great idea #2: Design for Moore's Law

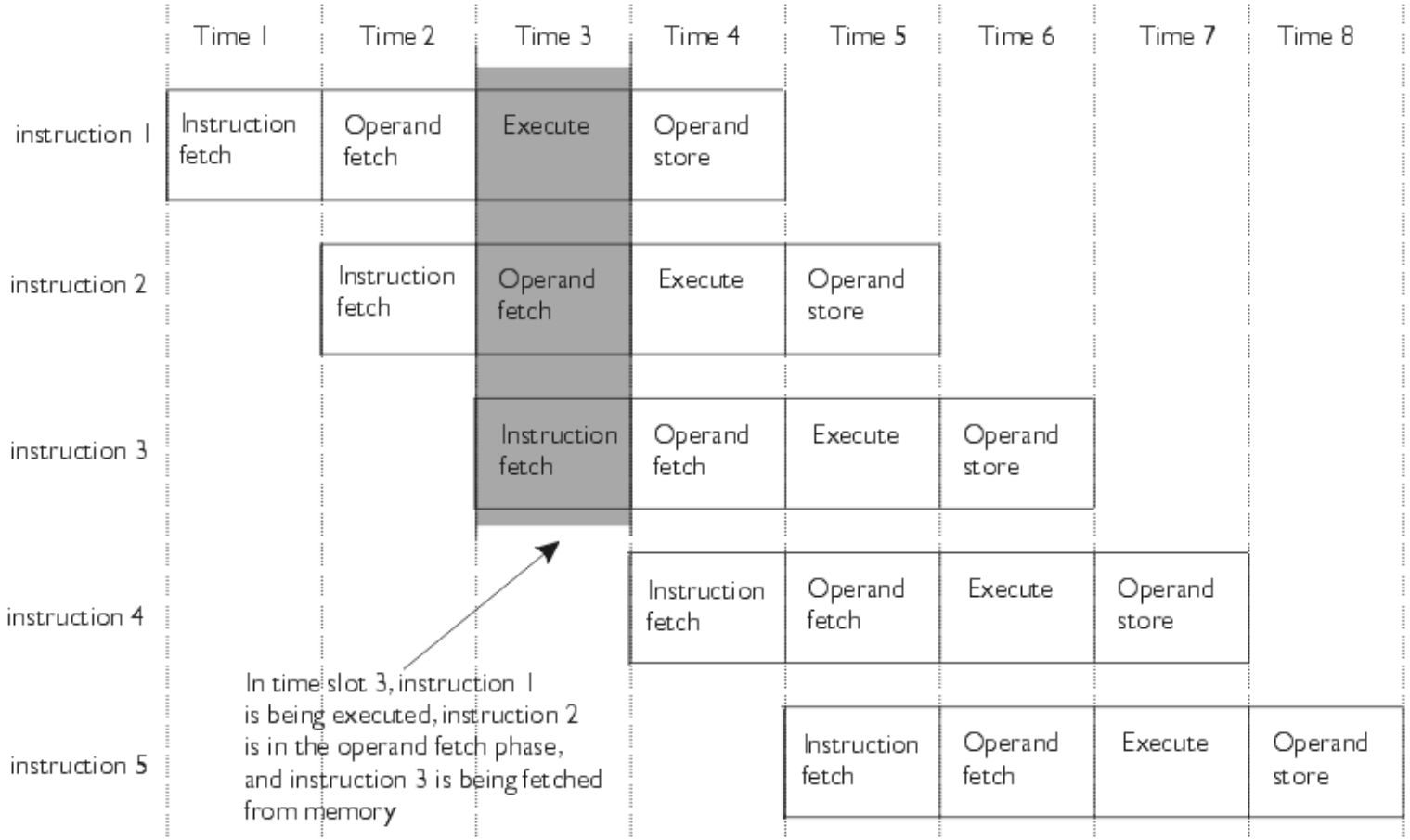


Great idea #4: Performance via Parallelism

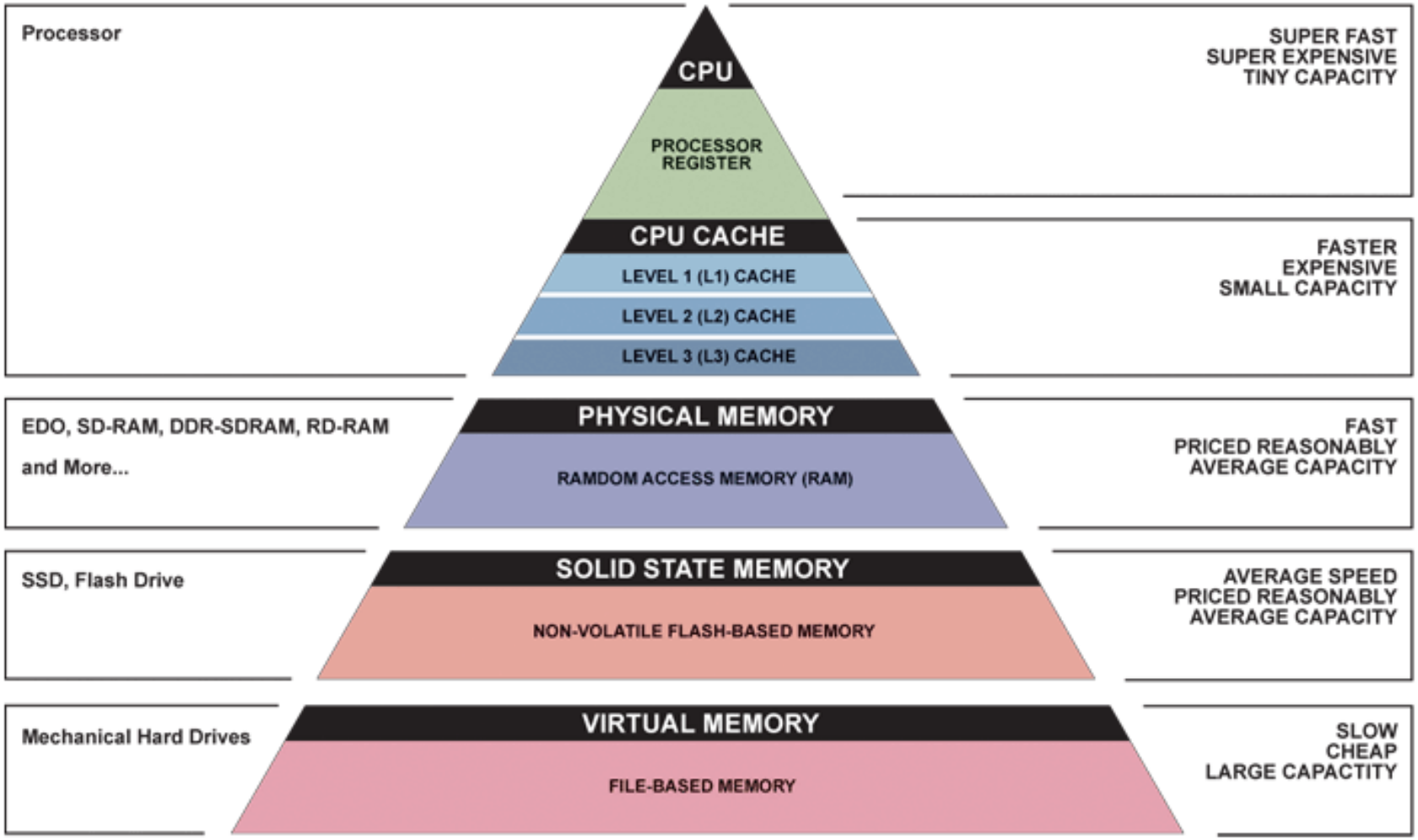


Parallelism can be used to tackle the processor-memory gap.

Great idea #5: Performance via Pipelining

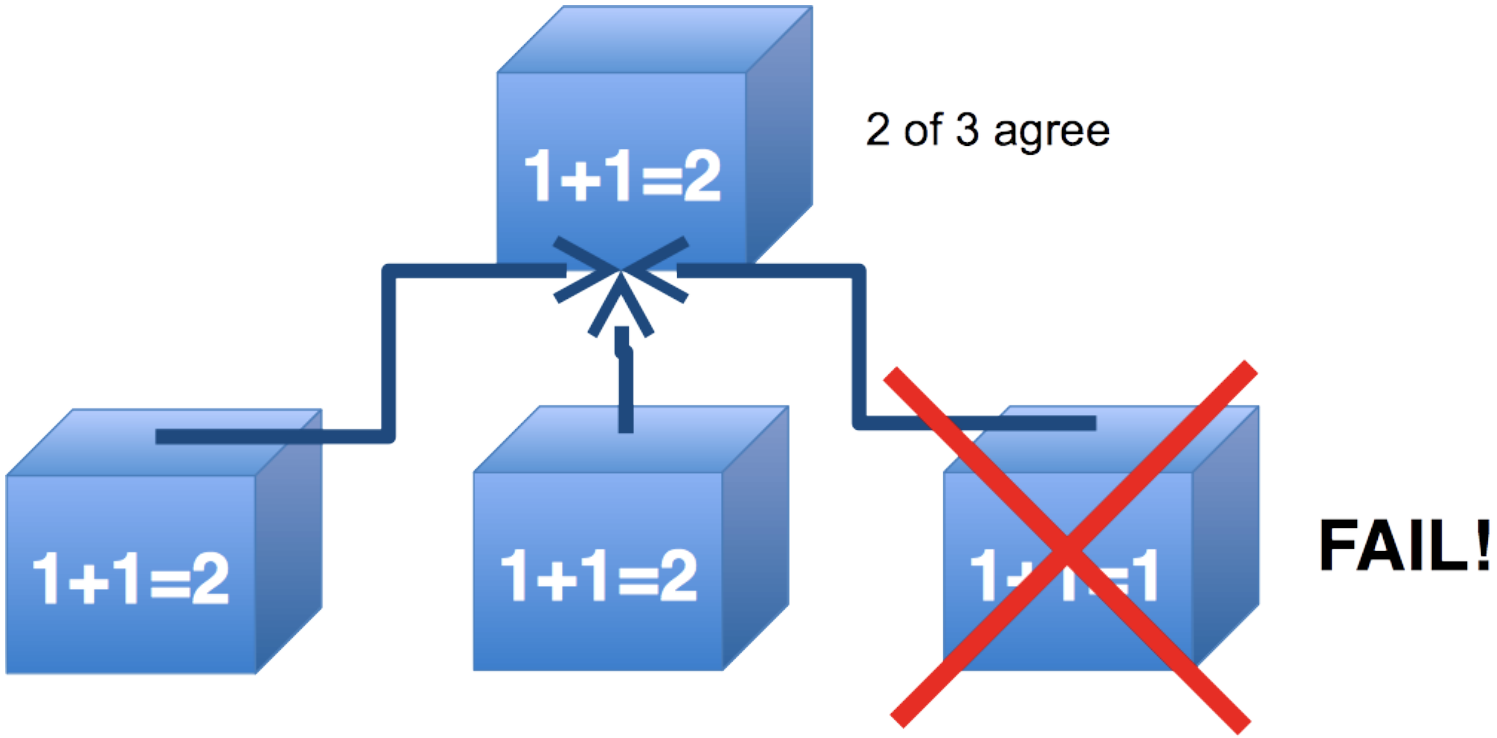


Great idea #7: Memory Hierarchy (Principle of Locality)



Great Idea #8: Dependability via Redundancy

Redundancy so that a failing piece doesn't make the whole system fail



Increasing transistor density reduces the cost of redundancy

Great Idea #8: Dependability via redundancy

Applies to everything from physical hardware components to the data encoded in binary.

- Redundant **data centers** so that can lose 1 datacenter but Internet service stays online
- Redundant **disks** so that can lose 1 disk but not lose data (Redundant Arrays of Independent Disks/RAID)
- Redundant **memory bits** of so that can lose 1 bit but no data (Error Correcting Code/ECC Memory)

In This Course...

Check the syllabus!

- Course topics
- Evaluation
- Schedule
- Textbook (recommended, not required)
- Missed quiz policies
- TAs, Office hours

Check OWL for continued updates and announcements. It's your fault if you miss something that was announced.

Acknowledgements

The lecture slides of this course are adapted from previous years' slides by Marc Moreno Maza which in turn have been adapted from text book accompaniments and teaching materials posted on the internet by other Computer Architecture instructors.

CS3350B Computer Organization
Chapter 1: CPU and Memory
Part 1: The CPU

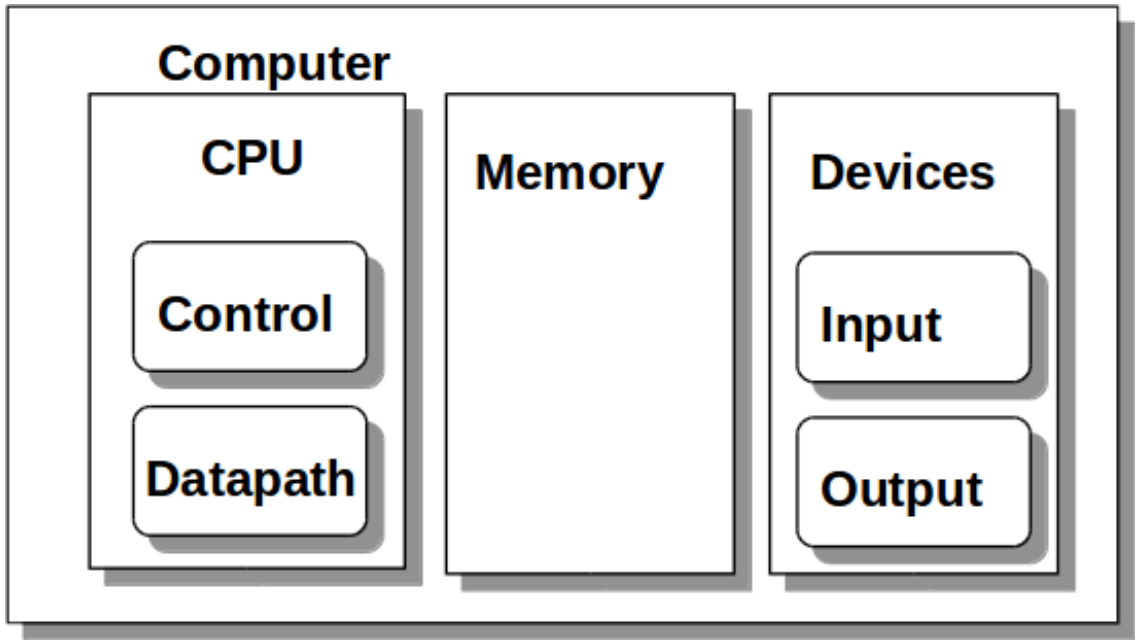
Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday January 10, 2019

Outline

Components of a computer



Micro-architecture: the internals of a CPU (control and datapath) will come later in the course. For now, we look at the CPU as a whole.

CPU and memory are highly coupled, especially when we look at performance. This will be seen by the end of this chapter.

Programmer's View of CPU Performance

At a basic level, the running time of some program on a CPU is determined by:

- The **clock** rate of the CPU (e.g. 3.4 GHz),
- The type of **instructions** being performed,
  - ↳ Addition/Subtraction faster than Multiplication/Division, etc.
  - ↳ Affects the *average* clock cycles per instruction (CPI)
- **Memory** access time.
  - ↳ Recall *processor-memory gap*

Assuming CPU clock rate is fixed, programmers can influence program performance by changing the type of instructions they use as well as how their code accesses memory.

Aside: Changing Instructions for Performance

On 6th-generation Intel CPUs

- 32-bit integer division ~26 clock cycles vs. logical bit shift ~1 clock cycle
  - `int i = 1234567; i /= 2;`
  - `int i = 1234567; i >>= 1;`

- Floating point division ~14 clock cycles vs. multiplication ~5 clock cycles
  - `float x = 1.2f; x /= 2.0f;`
  - `float x = 1.2f; x *= 0.5f;`

Fast Inverse Square Root thanks to *Quake*:
`https://en.wikipedia.org/wiki/Fast_inverse_square_root`

Understanding and Analyzing Performance

- **Algorithmic analysis:** Estimating the complexity of algorithms in some abstract, idealized way. In reality, two different $O(n^2)$ algorithms can have wildly different running times.
- **Programming language, compiler, architecture:**
  - ↳ Programming language and corresponding compiler determine the actual machine instructions the CPU will perform.
  - ↳ Resulting number and type of machine instructions vary by compiler and language and therefore resulting performance varies.
- **Processor and Memory:** Determines how fast instructions are executed and how fast data moves to and from the processor.
- **I/O system (including OS):** Determines how fast I/O operations are executed

Need for Performance Metrics

- Purchasing Perspective. For a collection of machine which one has the
  - ↳ "best" cost?
  - ↳ "best" cost relative to performance?
- Design perspective. Given many design options and directions which one has the
  - ↳ "best" performance improvement?
  - ↳ "best" cost relative to performance improvement?

In either case we need: (i) a basis for comparison, (ii) metrics for evaluation.

Our goal is to understand what factors in the architecture contribute to the overall system performance and the relative importance (and cost) of these factors.

CPU Performance: Latency

CPU performance is largely measured by **latency**, **throughput**, **clock frequency**.

- Want reduced response time (aka execution time, aka latency) – the time between the start and the completion of a task.
  - Important to general PC users.
- To maximize performance of some code segment (program) X, we need to minimize execution time.

$$\text{performance}_X = 1/\text{execution\_time}_X$$

- If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n$$

- *Note:* Can also compare latency of the same single *instruction* on different CPUs.

CPU Performance: Throughput

- Want increased *throughput* - the total amount of work done in a given unit of time.
  - Important to users like data center managers.
- Again, throughput depends on the code segment being executed. Different instructions result in different throughput measures.
- Decreasing response time usually improves throughput, but other factors are important (task scheduling, memory bandwidth, etc.).

CPU Performance: Clock Frequency

Clock Frequency is a code-agnostic measure of CPU performance.

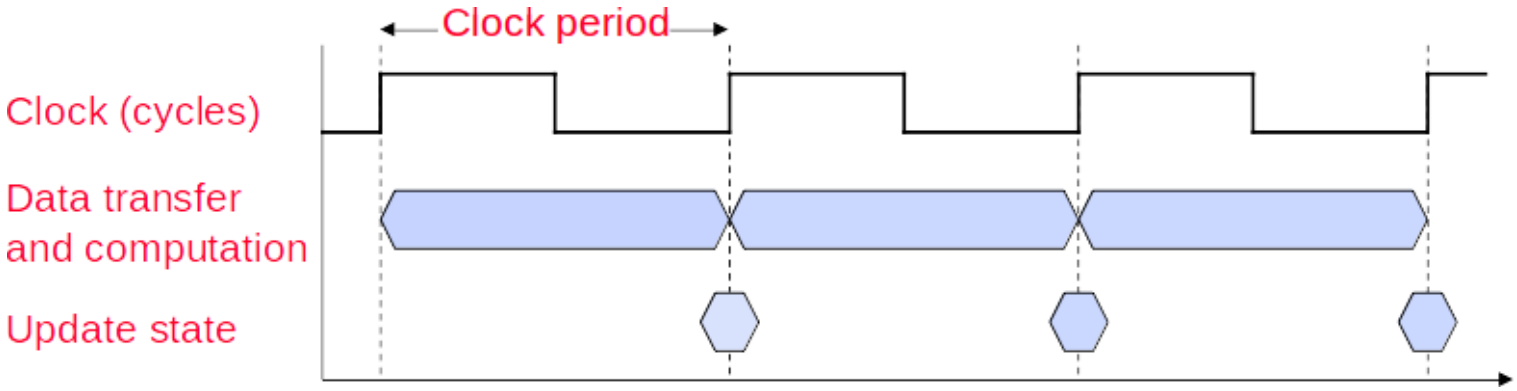
- Typically, a faster clock (higher frequency) yields a higher performing CPU.
- **But** the *micro-architecture* and the *instruction set architecture* play a large role.
  - ↳ They influence how much work the CPU does per cycle (i.e. efficiency)
- **Example:**

  CPU A runs at 3 GHz and a division takes 20 cycles.
  CPU B runs at 2 GHz and a division takes 10 cycles.

$$20 \text{ cycles} \;/\; 3\ GHz = 20 \;/\; 3000000000\ Hz = 6.66ns$$
$$10 \text{ cycles} \;/\; 2\ GHz = 10 \;/\; 2000000000\ Hz = 5.00ns$$

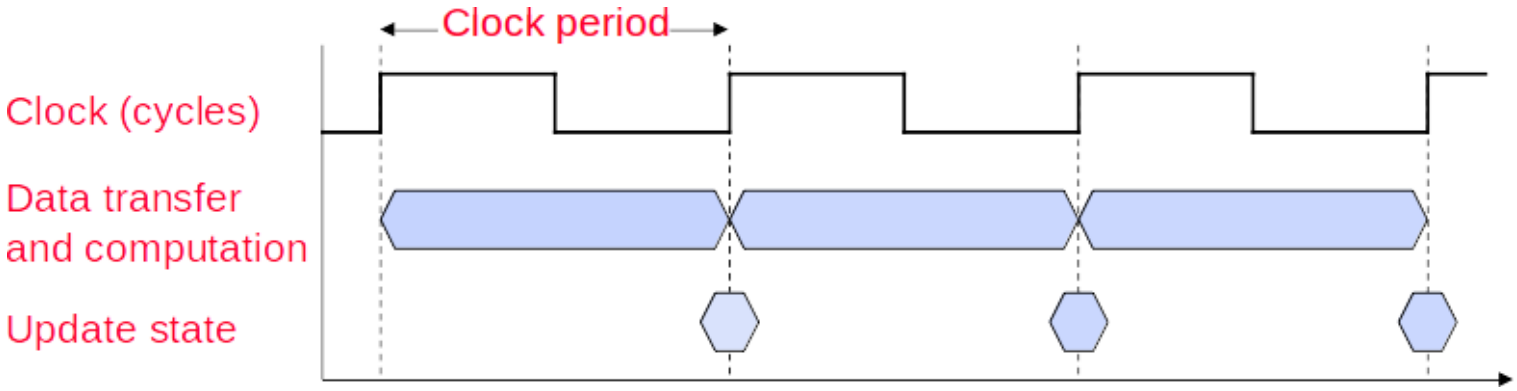
CPU Clocking

Synchronous digital systems (e.g. a CPU) are governed and controlled by a clock. This clock synchronizes internal circuits, memory states, and data movement.



Length of clock period determined by internal circuits and micro-architecture design. We will look at this with CPU datapaths and pipelining.

CPU Clocking



Clock period (cycle): duration of a clock cycle (CC)

- determines the *speed* of a computer processor
- Caveat: again, not necessarily latency or throughput though
- e.g., 250ps = 0.25ns = $250 \times 10^{-12} s$

Clock frequency or rate (CR): cycles per second

- the inverse of the clock period
- e.g., 3.0GHz = 3000MHz = 3.0×10^{9}Hz

CR = 1 / CC.

CPU Time

- It is important to distinguish *elapsed time* and the *time spent on your task*.
  - ↳ **Wall time** vs. **CPU time**
  - ↳ CPU time does not include time waiting for I/O or time spent on other processes

$$\begin{array}{l}\text{CPU execution time} \\ \text{for a program}\end{array} = \begin{array}{l}\#\text{CPU clock cycles} \\ \text{for a program}\end{array} \times \text{clock} - \text{cycle}$$

or

$$\begin{array}{l}\text{CPU execution time} \\ \text{for a program}\end{array} = \begin{array}{l}\#\text{CPU clock cycles} \\ \text{for a program}\end{array} / \text{clock} - \text{rate}$$

- We can improve performance by reducing either the *length of the clock cycle* or the **number of clock cycles required for a program**.

Outline

Instruction Performance

$$\begin{array}{c} \#\text{CPU clock cycles} \\ \text{for a program} \end{array} = \begin{array}{c} \#\text{Instructions} \\ \text{for a program} \end{array} \times \begin{array}{c} \text{Average \# of clock cycles} \\ \text{per instruction} \end{array}$$

- **Clock cycles per instruction (CPI)** - the average number of clock cycles each instruction takes to execute.
  - ↳ Different instructions may take different amounts of time depending on what they do.
  - ↳ A way to compare two different implementations of the same ISA.
  - ↳ Calculated by a simple averaging of each instruction type in a program and their corresponding number of cycles.

The Classic Performance Equation

$$\begin{aligned} \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle} \\ &\text{or} \\ \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} \,/\, \text{clock\_rate} \end{aligned}$$

- Keep in mind that the only complete and reliable measure of computer performance is **time**.
- For example, redesigning the hardware implementation of an instruction set to lower the instruction count may lead to an organization with
  - ↳ a slower clock cycle time or,
  - ↳ higher CPI,

  that offsets the improvement in instruction count.
- *Note:* CPI depends on the type of instruction executed, so the code which executes the fewest instructions may not be the fastest.

A Simple Example (1/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)$$

Op	Freq	CPI_i	Freq $\times$ CPI_i	(1)
ALU	50%	1	.5	.5
Load	20%	5	1.0	
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
			$\sum = 2.2$	

(1) How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

A Simple Example (1/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)$$

Op	Freq	CPI_i	Freq × CPI_i	(1)
ALU	50%	1	.5	.5
Load	20%	5	1.0	.4
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
			$\sum = 2.2$	1.6

(1) How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
CPU time new = 1.6 × IC × CC; so 2.2 versus 1.6 which means 37.5% faster

A Simple Example (2/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)$$

Op	Freq	CPI_i	Freq × CPI_i	(2)	(3)
ALU	50%	1	.5	.5	
Load	20%	5	1.0	1.0	1.0
Store	10%	3	.3	.3	.3
Branch	20%	2	.4		.4
			$\sum = 2.2$		

(2) How does this CPI compare with using branch prediction to save a cycle off the branch time?

(3) What if two ALU instructions could be executed at once?

A Simple Example (2/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)$$

Op	Freq	CPI_i	Freq × CPI_i	(2)	(3)
ALU	50%	1	.5	.5	
Load	20%	5	1.0	1.0	1.0
Store	10%	3	.3	.3	.3
Branch	20%	2	.4	.2	.4
			$\sum = 2.2$	2.0	

(2) How does this CPI compare with using branch prediction to save a cycle off the branch time?
CPU time new = 2.0 × IC × CC so 2.2 versus 2.0 means 10% faster

(3) What if two ALU instructions could be executed at once?

A Simple Example (2/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i)$$

Op	Freq	CPI_i	Freq × CPI_i	(2)	(3)
ALU	50%	1	.5	.5	.25
Load	20%	5	1.0	1.0	1.0
Store	10%	3	.3	.3	.3
Branch	20%	2	.4	.2	.4
			$\sum = 2.2$	2.0	1.95

(2) How does this CPI compare with using branch prediction to save a cycle off the branch time?
CPU time new = 2.0 × IC × CC so 2.2 versus 2.0 means 10% faster

(3) What if two ALU instructions could be executed at once?
CPU time new = 1.95 × IC × CC so 2.2 versus 1.95 means 12.8% faster

Understanding Program Performance

$$\text{CPU Time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware.

	Instruction_count	CPI	clock_cycle
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Processor organization		X	X

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\frac{15 \times 0.6}{1.1} = 8.2$ sec
b. $15 \times 0.6 \times 1.1 = 9.9$ sec
c. $\frac{15 \times 1.1}{0.6} = 27.5$ sec

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\frac{15 \times 0.6}{1.1} = 8.2$ sec
b. $15 \times 0.6 \times 1.1 = 9.9$ sec
c. $\frac{15 \times 1.1}{0.6} = 27.5$ sec

b!

$$
\begin{aligned}
Time_1 &= IC_1 \times CPI_1 \times CC_1 \\
Time_2 &= IC_2 \times CPI_2 \times CC_2 \\
&= (IC_1 \times 0.6) \times (CPI_1 \times 1.1) \times CC_1 \\
&= Time_1 \times 0.6 \times 1.1 \\
&= 15 \times 0.6 \times 1.1 = 9.9
\end{aligned}
$$

Outline

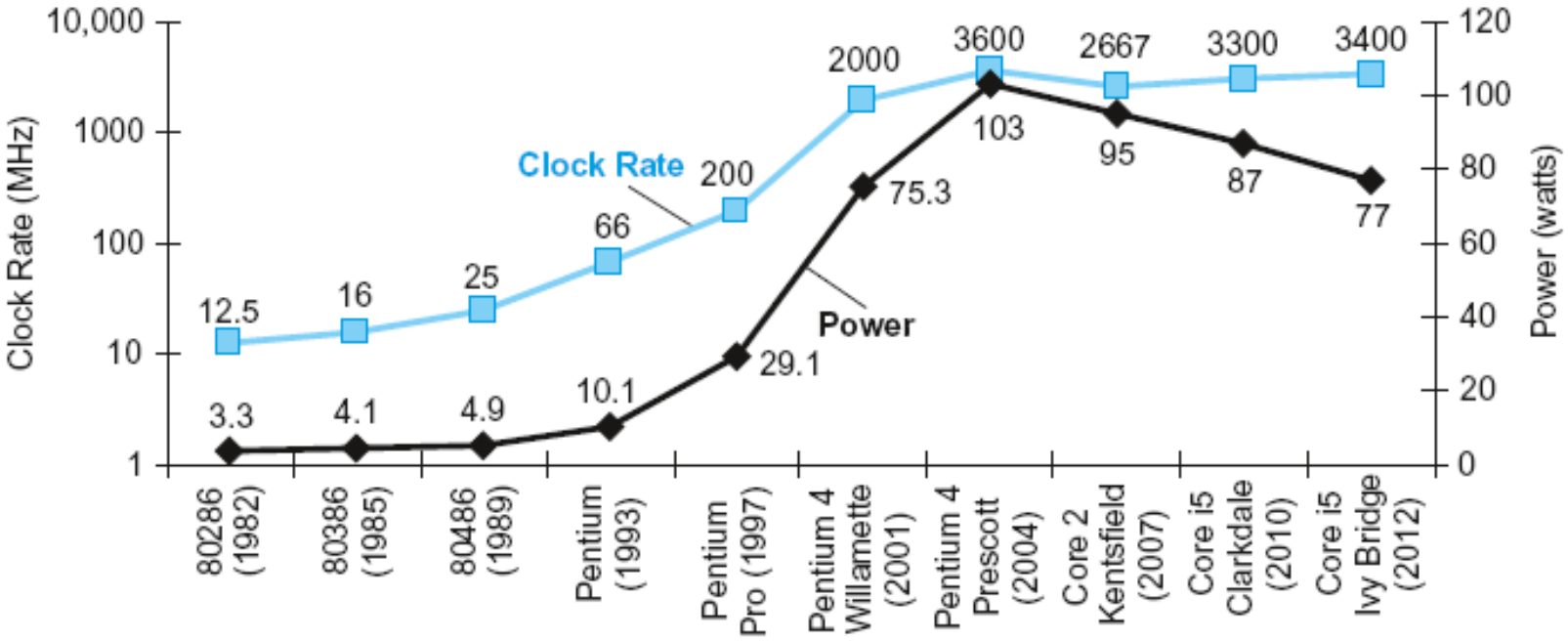
CPU Power Usage

Depending on an architect's design goals they may want to look at metrics different from latency, throughput, time, or clock frequency.

Power Usage $\Longrightarrow$ Temperature
Power Usage $\Longrightarrow$ Battery Life

Ultrabooks are so (not) hot right now

Power Trends



Complementary metal oxide semiconductor (CMOS) integrated circuits, the technology which implements the physical circuity inside modern CPUs, has a (simplified) power equation:

$$\underset{(\times 30)}{\text{Power}} = \text{Capacitive load} \times \underset{(5V \rightarrow 1V)}{\text{Voltage}^2} \times \underset{(\times 1000)}{\text{Frequency switched}}$$

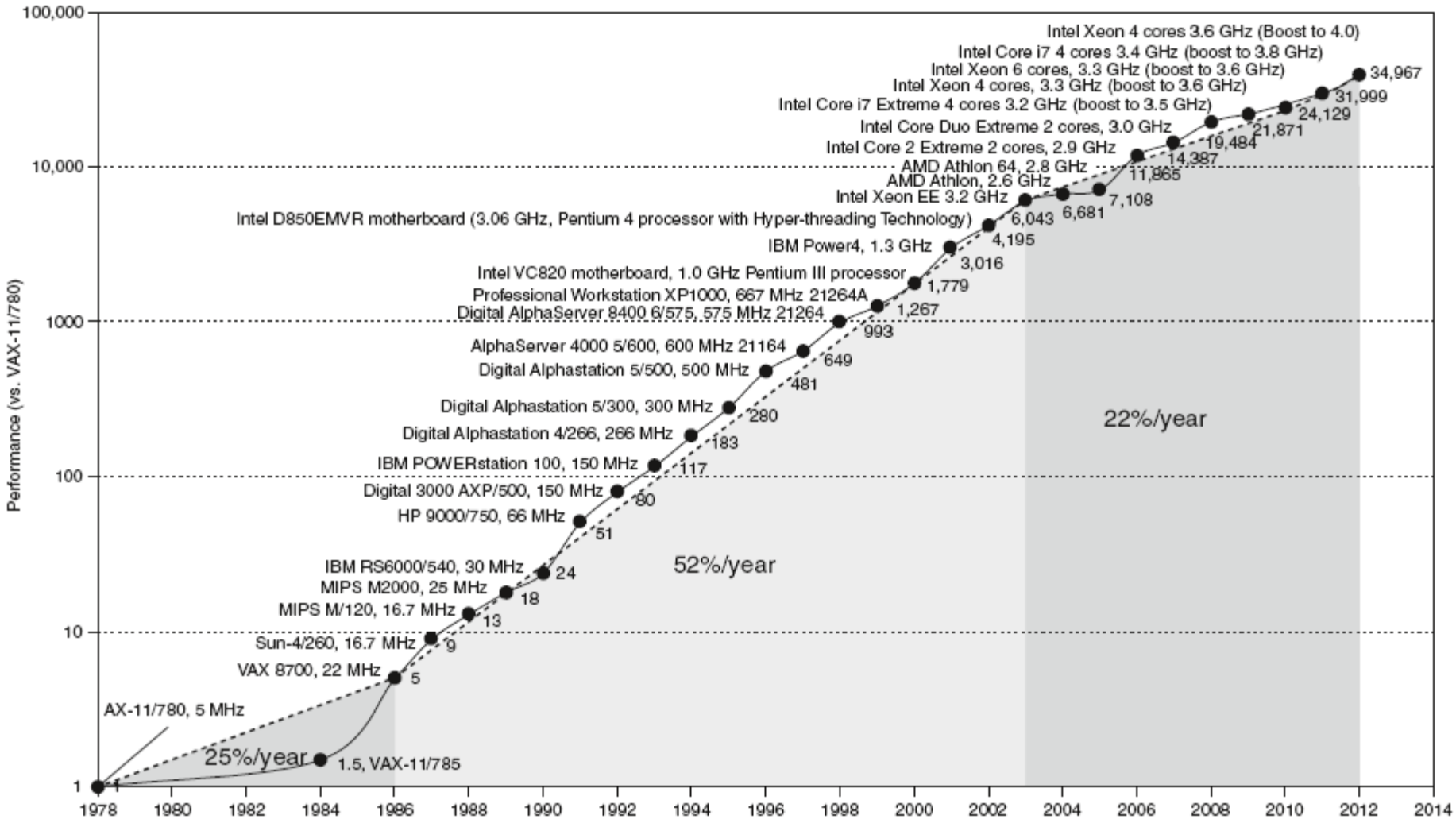
Reducing Power

- Suppose a new CPU has
  - ↳ 85% of capacitive load of old CPU
  - ↳ 15% voltage and 15% frequency reduction

$$\frac{\mathrm{P_{new}}}{\mathrm{P_{old}}} = \frac{\mathrm{C_{old}} \times 0.85 \times (\mathrm{V_{old}} \times 0.85)^2 \times \mathrm{F_{old}} \times 0.85}{\mathrm{C_{old}} \times \mathrm{V_{old}^2} \times \mathrm{F_{old}}} = 0.85^4 = 0.52$$

- **The Power Wall**
  - ↳ Decreasing transistor size $\Longrightarrow$ more transistors per chip $\Longrightarrow$ greater power density
  - ↳ Required input voltage may decrease but more transistors use more power.
  - ↳ Reducing voltage further very difficulty.
  - ↳ Removing extra heat very difficult.

CPU Performance: Relative Performance vs VAX-11



Multi-Processors to the Rescue

Moore's Law failing? Great idea #4: Performance via Parallelism.

- Multi-core processors!
  - ↳ More than one processor per chip
  - ↳ Huge benefits to OS and multiple processes.
- Within a single process requires explicit parallel programming
  - ↳ Compared with instruction level parallelism:
    - Hardware executes multiple instructions at once (pipelining/multi-issue)
    - Hidden from the programmer
  - ↳ Hard to do:
    - Programming for performance
    - Thread management, Load Balancing
    - Optimizing communication and synchronization

Outline

SPEC CPU Benchmark

- Programs used to measure performance
  - ↳ Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - ↳ Develops benchmarks for CPU, I/O, Web, ...
- SPEC CPU2006
  - ↳ Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - ↳ Normalize relative to reference machine
  - ↳ Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

CINT2006 for Intel Core i7 920

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7

Profiling Tools

- Many profiling tools
  - `gprof` (static instrumentation)
  - `cachegrind`, `Dtrace` (dynamic instrumentation)
  - `perf` (performance counters)

- `perf` in `linux-tools`, based on event sampling
  - Keep a list of where "interesting events" (cycle, cache miss, etc) happen
  - CPU Feature: Counters for hundreds of events
    - Performance: Cache misses, branch misses, instructions per cycle, ...
  - Intel®64 and IA-32 Architectures Software Developer's Manual: Appendix A lists all counters `http://www.intel.com/products/processor/manuals/index.html`
  - `perf` user guide: `https://perf.wiki.kernel.org/index.php/Tutorial`

Exercise 1

```
void copymatrix1(int** src,
        int** dst, int n) {
  int i,j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      dst[i][j] = src[i][j];
}
```

```
void copymatrix2(int** src,
        int** dst, int n) {
  int i,j;
  for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
      dst[i][j] = src[i][j];
}
```

- `copymatrix1` vs `copymatrix2`
  - What do they do?
  - What is the difference?
  - Which one performs better? Why?
- `perf stat -e cycles -e cache-misses ./copymatrix1`
  `perf stat -e cycles -e cache-misses ./copymatrix2`
  - What does the output like?
  - How to interpret it?
  - Which program performs better?

Exercise 1: Results

```
abrandt5@node04:~/test-code$ gcc -o copy.bin copymatrix.c
abrandt5@node04:~/test-code$ perf stat -e cycles -e cache-misses ./copy.bin

 Performance counter stats for './copy.bin':

       557,458,379      cycles
         2,071,981      cache-misses

       0.210788972 seconds time elapsed

abrandt5@node04:~/test-code$ vi copymatrix.c
abrandt5@node04:~/test-code$ gcc -o copy.bin copymatrix.c
abrandt5@node04:~/test-code$ perf stat -e cycles -e cache-misses ./copy.bin

 Performance counter stats for './copy.bin':

     1,418,374,698      cycles
         3,980,687      cache-misses

       0.534273226 seconds time elapsed
```

Exercise 2

```
void lower1 (char* s) {
  int i;
  for (i = 0; i < strlen(s); i++)
    if (s[i]>='A' && s[i]<='Z')
      s[i] -= 'A'-'a';
}
```

```
void lower2 (char* s) {
  int i;
  int n = strlen(s);
  for (i = 0; i < n; i++)
    if (s[i]>='A' && s[i]<='Z')
      s[i] -= 'A'-'a';
}
```

- `lower1` vs `lower2`
  - ↳ What do they do?
  - ↳ What is the difference?
  - ↳ Which one performs better? Why?
- `perf stat -e cycles -e cache-misses ./lower1`
  `perf stat -e cycles -e cache-misses ./lower2`
  - ↳ What does the output like?
  - ↳ How to interpret it?
  - ↳ Which program performs better?

Exercise 2: Results

```
abrandt5@node04:~/test-code$ gcc -o lower.bin lower.c
abrandt5@node04:~/test-code$ perf stat -e cycles -e cache-misses ./lower.bin

 Performance counter stats for './lower.bin':

   131,658,606,345      cycles
            19,799      cache-misses

      43.195108384 seconds time elapsed

abrandt5@node04:~/test-code$ vi lower.c
abrandt5@node04:~/test-code$ gcc -o lower.bin lower.c
abrandt5@node04:~/test-code$ perf stat -e cycles -e cache-misses ./lower.bin

 Performance counter stats for './lower.bin':

        24,846,861      cycles
            12,013      cache-misses

       0.010991759 seconds time elapsed
```

CS3350B Computer Organization
Chapter 1: CPU and Memory
Part 2: The Memory Hierarchy

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Tuesday January 15, 2019

Recap: CPU Time

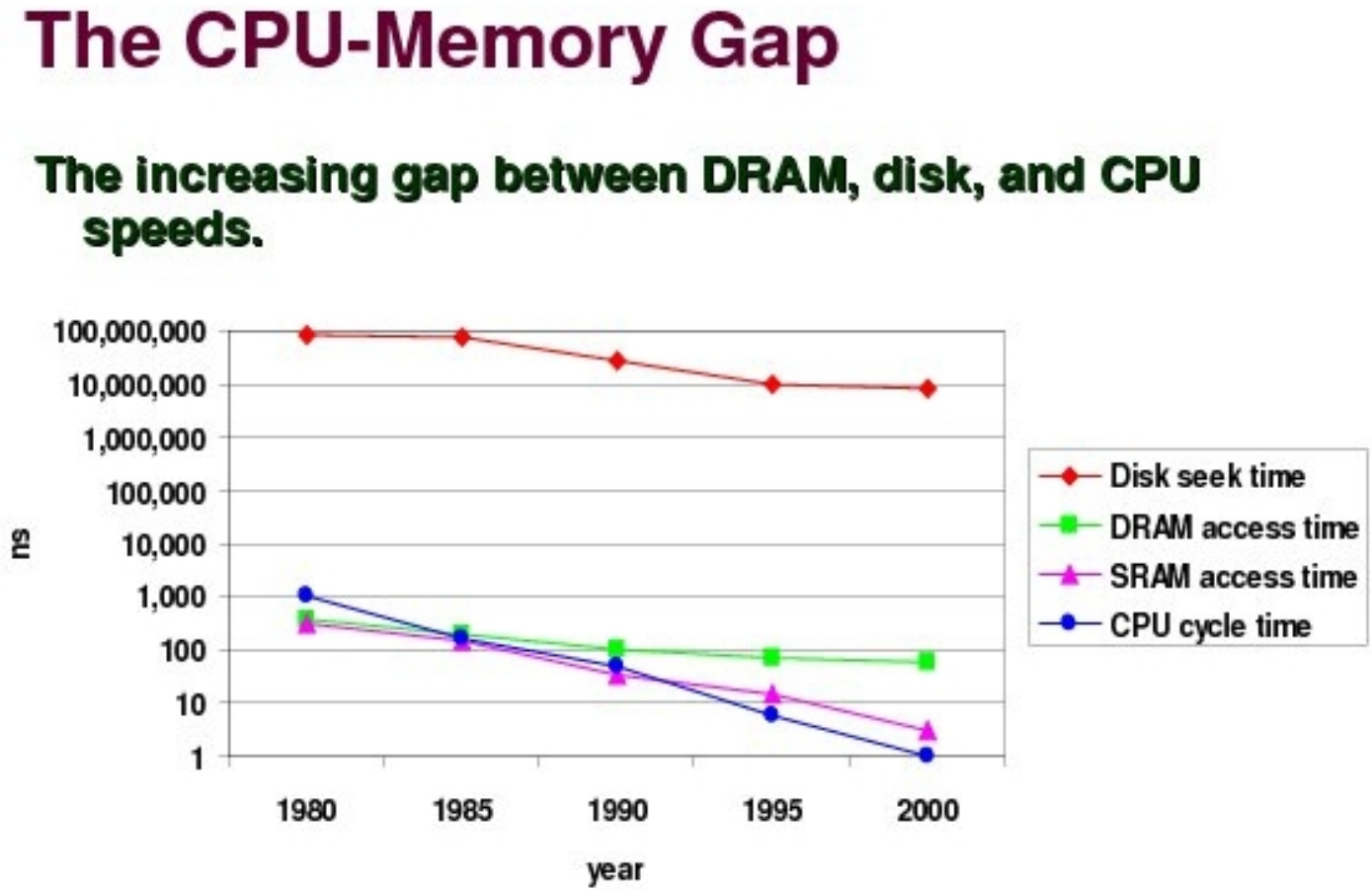
$$\text{CPU Time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

	Instruction_count	CPI	clock_cycle
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Processor organization		X	X

From a programmer's point of view CPU performance depends on:

- CPU Frequency
- The type of instructions performed
- **Memory access time**

Recap: Processor-Memory Gap



The **Processor-Memory Gap** is a key contributor to the **Memory Wall** – the point where a program's performance is totally determined by memory speed. Faster processors will not make programs run faster!

Recap: Memory Wall Example

```
void copymatrix1(int** src,
int** dst, int n) {
  int i,j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      dst[i][j] = src[i][j];
}
```

```
Performance counter stats for './copy.bin':

     557,458,379      cycles
       2,071,981      cache-misses

     0.210788972 seconds time elapsed
```

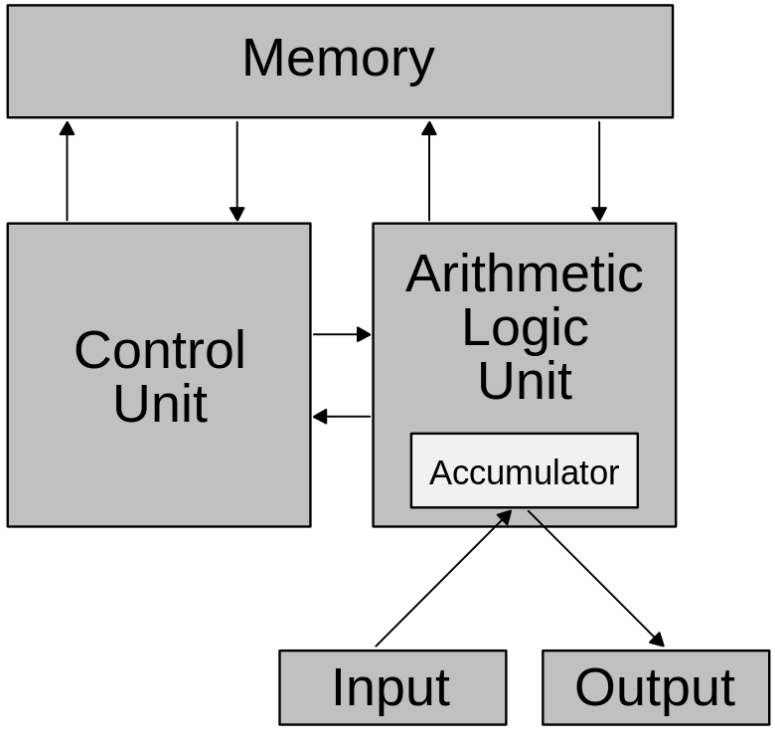
```
void copymatrix2(int** src,
int** dst, int n) {
  int i,j;
  for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
      dst[i][j] = src[i][j];
}
```

```
Performance counter stats for './copy.bin':

   1,418,374,698      cycles
       3,980,687      cache-misses

     0.534273226 seconds time elapsed
```

Cache misses are the reason for the performance drop.

Outline

Von Neumann Architecture



A simple shared memory for both instructions and data.

- From a software's point of view this model is still very much true.
- Hardware takes care of all of the sophistication.

Before the Hierarchy

In the 1950s, 60s, and 70s, software development was mostly mathematical and scientific.

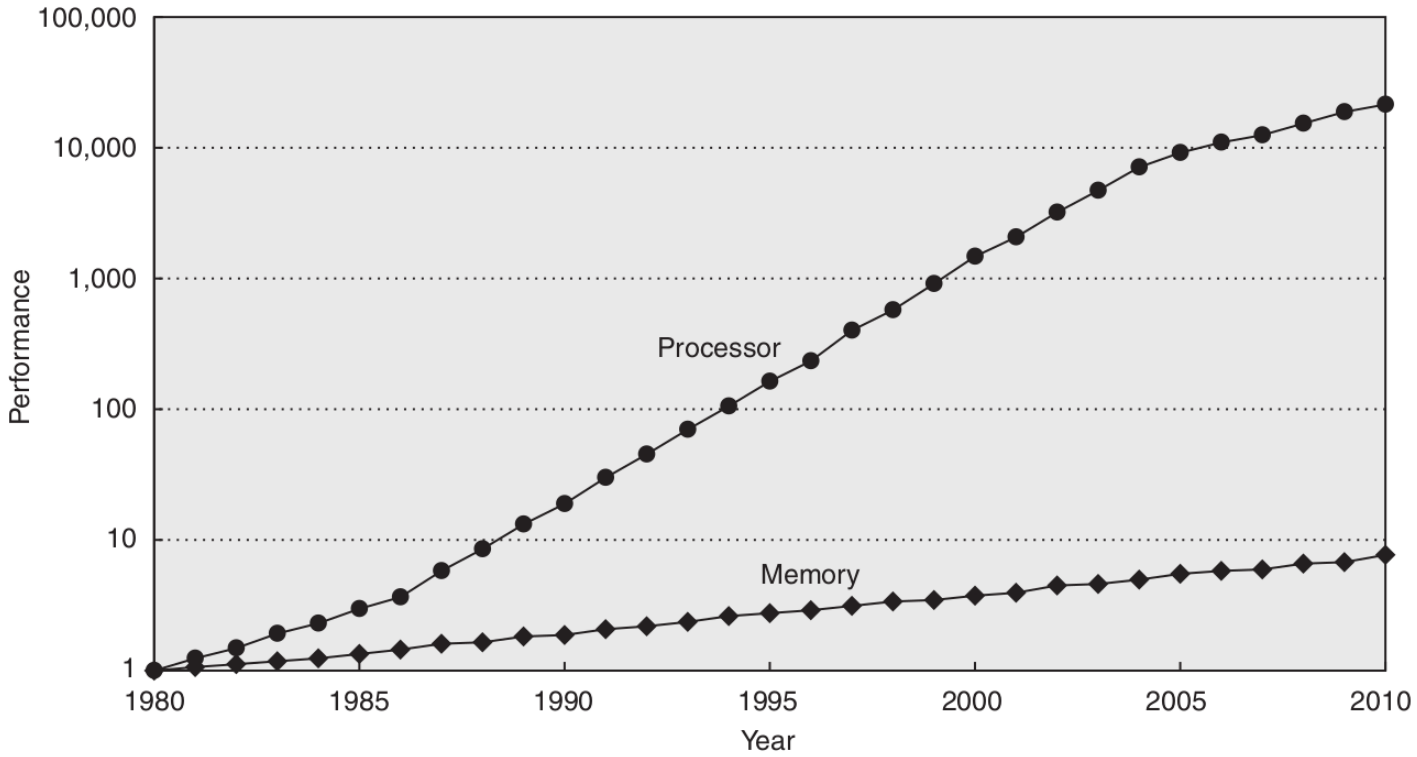
- The heyday of FORTRAN.
- Computing resources were very limited.
  - ↳ As late as the 1980s memory was only 64 **mega**bytes.
- Algorithm development focused on minimizing amount of memory a program used in order to solve larger problems.
  - ↳ **Sparse** mathematics, linear algebra, symbolic computations.
  - ↳ Minimizing storage of numerical types: char/INT1 (1 byte), short/INT2 (2 bytes), int/INT4 (4 bytes), ...

Current Memory Resources

- Historically, programmers where concerned with *how much* memory was used.
  - ↳ Speeds of processor and memory were about the same.
  - ↳ Memory access time was roughly same as arithmetic time.
- Today, we are for more concerned with *how* memory is used.
- Memory *size* is no longer limiting factor:
  - ↳ Simple laptops $\Longrightarrow$ up to 64 **giga**bytes.
  - ↳ Compute clusters $\Longrightarrow$ hundreds of gigabytes *per processor*.
- **Memory Wall** (memory speed) is now the limiting factor.
- **Memory Hierarchy** introduced to tackle Memory Wall and limit the effects of the **Processor-Memory Gap**.

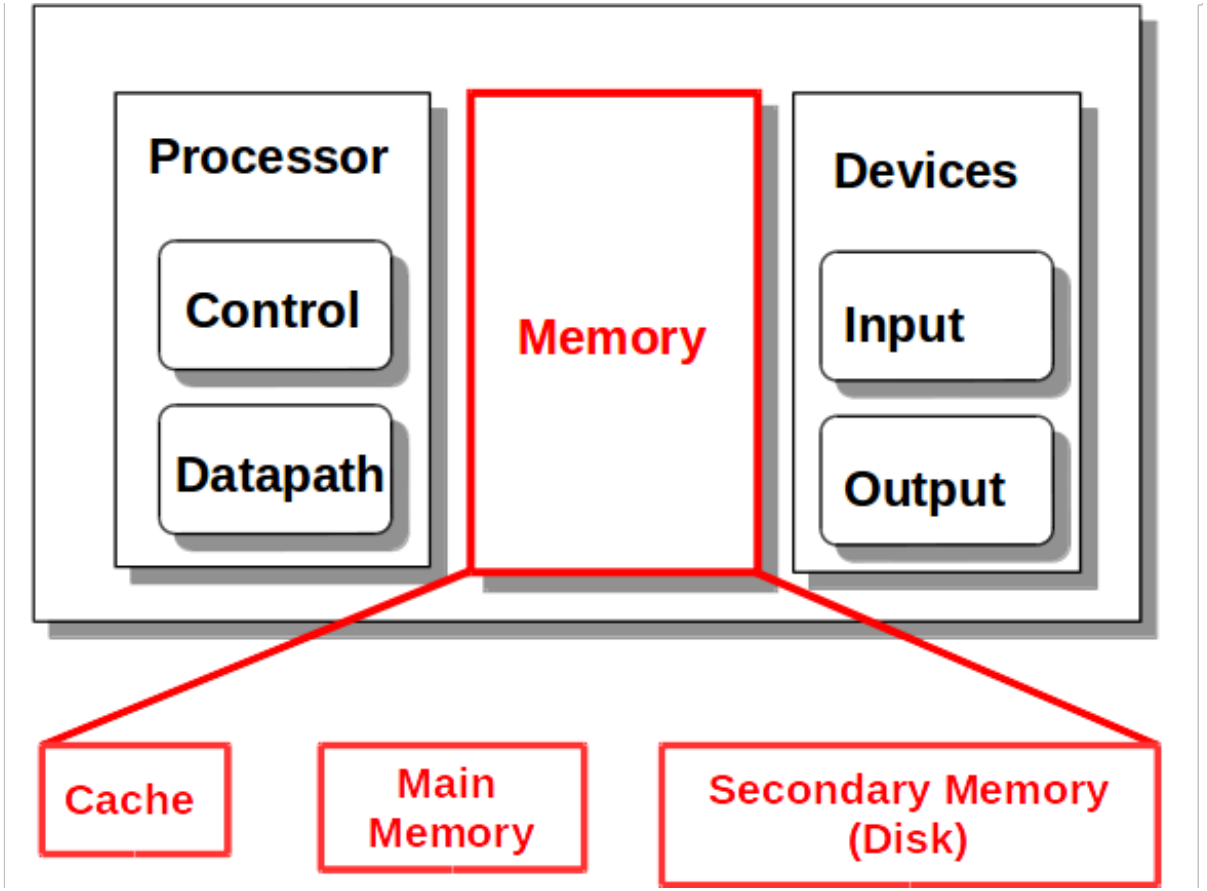
The Memory Wall

- Processor vs DRAM speed disparity continues to grow
- Notice log scale!



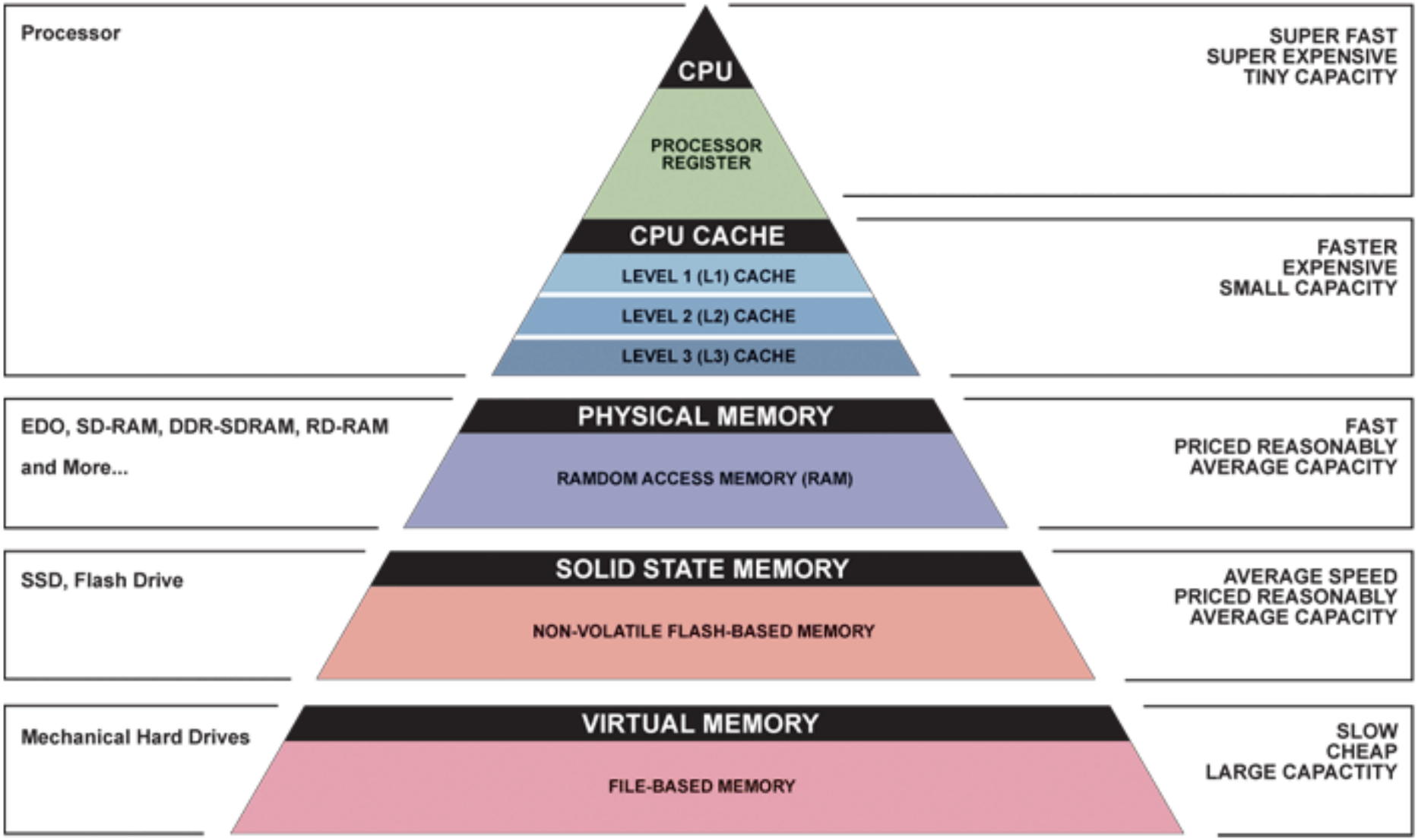
Hennessy & Patterson, *Computer Architecture: A Quantitative Approach*

Components of a Computer: Memory



At first, cache was a single, simple entity. Around the 1990s **multi-level caches** began.

Memory Hierarchy



But Why a Hierarchy?

Enduring properties of hardware and software:

- Fast storage technologies (SRAM) cost more per byte.
- Fast storage technologies take up more physical space per byte.
- Gap between CPU and main memory (DRAM) speed is widening.
- **Principle of Locality**.

Whole sections of memory are copied from lower levels of hierarchy to higher levels.

- Assuming locality, this transfer is done infrequently and computations can occur continuously by only referencing data high in the hierarchy.

Outline

The Principle of Locality

"Programs tend to reuse data and instructions they have recently used."

↳ Hennessy and Patterson, *Computer Architecture*.

During a short time range, a program is likely to access only a relatively small portion of:

- available address space,
- available program code.

This principle is a driving factor in computer architecture and design.

- Always in an effort to improve performance.
- Locality directs hardware design. If your program breaks this principle then it **will not** use the hardware properly.

Locality in Time and Space

Temporal Locality

- Recently accessed items are likely to be accessed again in the near future.

Spatial Locality

- Recently accessed items are likely to have their adjacent (or near-by) items accessed in the near future.

Note: items can mean data memory addresses or instructions.

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?

- Spatial?

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?
  - ↳ Loops
- Spatial?

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?
  - ↳ Loops
  - ↳ Repeated calls to the same method
- Spatial?

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?
  - ↳ Loops
  - ↳ Repeated calls to the same method
- Spatial?
  - ↳ Loops

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?
  - ↳ Loops
  - ↳ Repeated calls to the same method
- Spatial?
  - ↳ Loops
  - ↳ Repeated calls to the same method

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?
  - ↳ Loops
  - ↳ Repeated calls to the same method
- Spatial?
  - ↳ Loops
  - ↳ Repeated calls to the same method
  - ↳ **Sequential** operation, no unconditional jumps!

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Instructions

Appreciating locality for instructions is more difficult than for memory addresses. What programming constructs lead to locality for instructions?

- Temporal?
  - ↳ Loops
  - ↳ Repeated calls to the same method
- Spatial?
  - ↳ Loops
  - ↳ Repeated calls to the same method
  - ↳ **Sequential** operation, no unconditional jumps!
    - Unless *inlined*, method calls can act like unconditional jumps.

Note: Continually using the same *type* of instruction is not locality. It must be the exact same instruction *after compilation*. (Same instruction type and operands).

Locality in Data

Data locality easier to understand:

- Repeated access to the same variable.
- Sequential access to an array.
- Initializing a variable just before it is used.
- Re-use previous variables that are finished being useful instead of initializing new ones.
  - ↳ When a new variable is created, it must be loaded into cache. Why not re-use existing variables?

Prime Example: *Stride-1 access pattern*

```
int arr[n];
for (int i = 0; i < n; ++i) {
    printf("%d", arr[i]);
}
```

Aside: Layout of C Arrays in Memory

- C arrays allocated in row-major order.
  - Each row in contiguous memory locations.
  - `a[i][j]` $\Longleftrightarrow$ `a[i * ncols + j]`

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix} \Longrightarrow \begin{bmatrix} 0, 1, 2, 3, 4, 5, 6, 7, 8 \end{bmatrix}$$

- FORTRAN, Matlab arrays allocated in column-major order.
  - Each column in contiguous memory locations.

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \Longrightarrow \begin{bmatrix} 1, 4, 7, 2, 5, 8, 3, 6, 9 \end{bmatrix}$$

Locality Exercises: Warm-up

Does this function in C have good locality? If yes, which type?

```
int sumArray(int* a, int n) {
    int sum = 0;
    for (i=0; i<n; i++)
        sum += a[i];
    return sum;
}
```

Locality Exercises: Warm-up

Does this function in C have good locality? If yes, which type?

```
int sumArray(int* a, int n) {
    int sum = 0;
    for (i=0; i<n; i++)
        sum += a[i];
    return sum;
}
```

- stride-1 access to a $\Longrightarrow$ spatial locality

Locality Exercises: Warm-up

Does this function in C have good locality? If yes, which type?

```
int sumArray(int* a, int n) {
    int sum = 0;
    for (i=0; i<n; i++)
        sum += a[i];
    return sum;
}
```

- stride-1 access to `a` $\Longrightarrow$ spatial locality
- temporal locality in access to `sum`, `i`, `n`

Locality Exercises: Warm-up

Does this function in C have good locality? If yes, which type?

```
int sumArray(int* a, int n) {
    int sum = 0;
    for (i=0; i<n; i++)
        sum += a[i];
    return sum;
}
```

- stride-1 access to `a` $\Longrightarrow$ spatial locality
- temporal locality in access to `sum`, `i`, `n`

Note: Temporal locality in reference to the *pointer* `a` but *not* the array elements. `a[i]` is just syntactic sugar.

↳ `a[i]` $\Longleftrightarrow$ `*(a + i)`

Locality Exercise 1

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Locality Exercise 1

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

- Spatial locality in access to a, *row-major order in C*.

Locality Exercise 1

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

- Spatial locality in access to a, *row-major order in C*.
- Temporal locality in access to sum, i, j, N.

Locality Exercise 2

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

Locality Exercise 2

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

- Temporal locality in access to `sum`, `i`, `j`, `M`.

Locality Exercise 2

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

- Temporal locality in access to `sum`, `i`, `j`, `M`.
- No spacial locality in access to `a`.

Locality Exercise 2

Where is locality present in this C function? For each specify the type of locality.

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (j = 0; j < N; j++)
        for (i = 0; i < M; i++)
            sum += a[i][j];
    return sum;
}
```

- Temporal locality in access to `sum`, `i`, `j`, `M`.
- No spacial locality in access to `a`.
- If `M` *small* possibility for temporal locality in `N`.

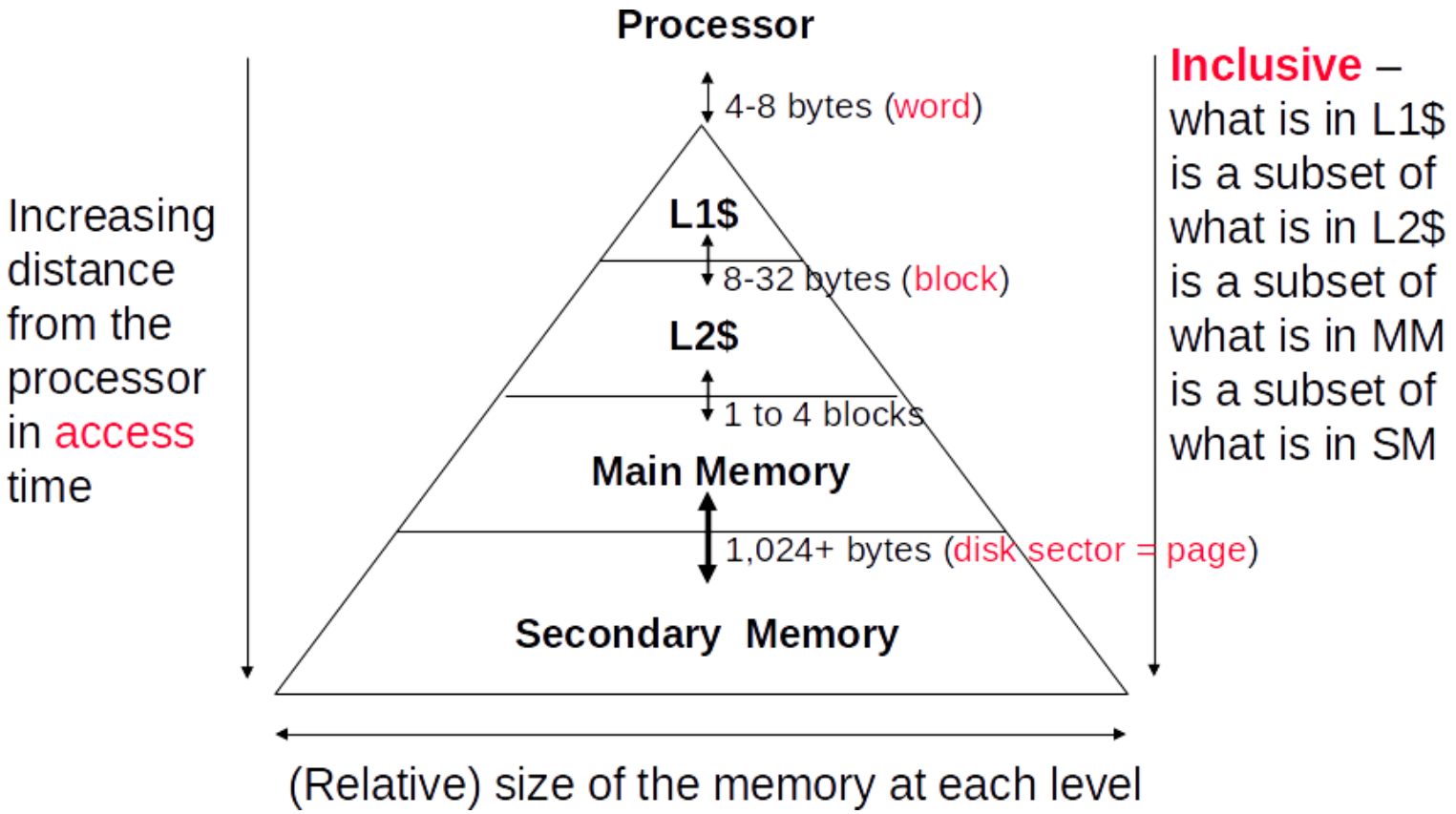
Locality Exercise 3

Does this C function have good spatial locality? If not, permute the loops so that spatial locality is achieved by stride-1 access.

```
int sumarray3d(int[][][] a, int N) {
    int i, j, k, sum = 0;

    for (j = 0; j < N; i++)
        for (i = 0; i < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

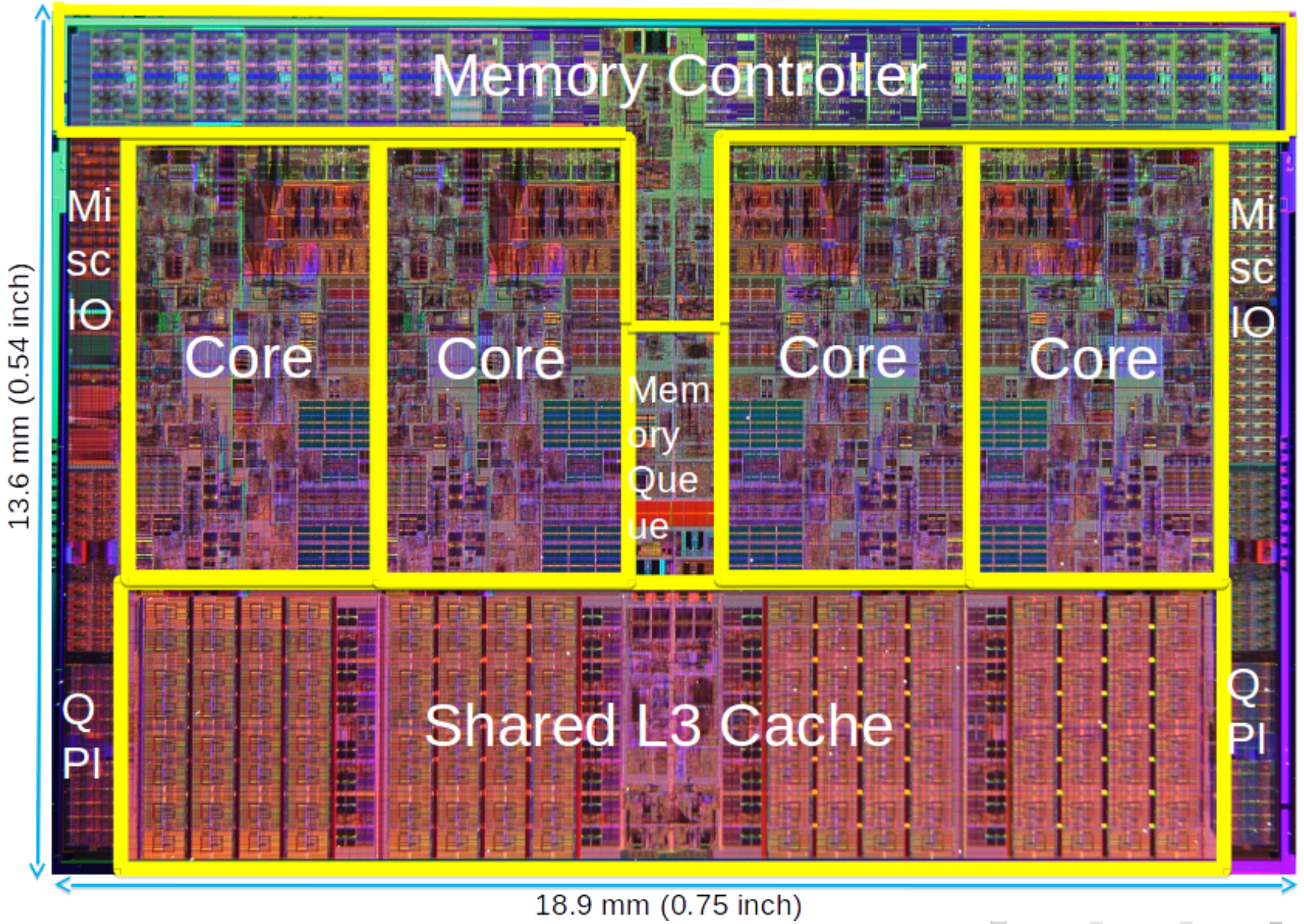
Outline

Characteristics of the Memory Hierarchy



- CPU looks first for data in L1, then in L2, ..., then in main memory.
- Modern computers usually have three levels of cache before main memory.

Photo of a CPU: Nehalem (First-Generation i7) Die



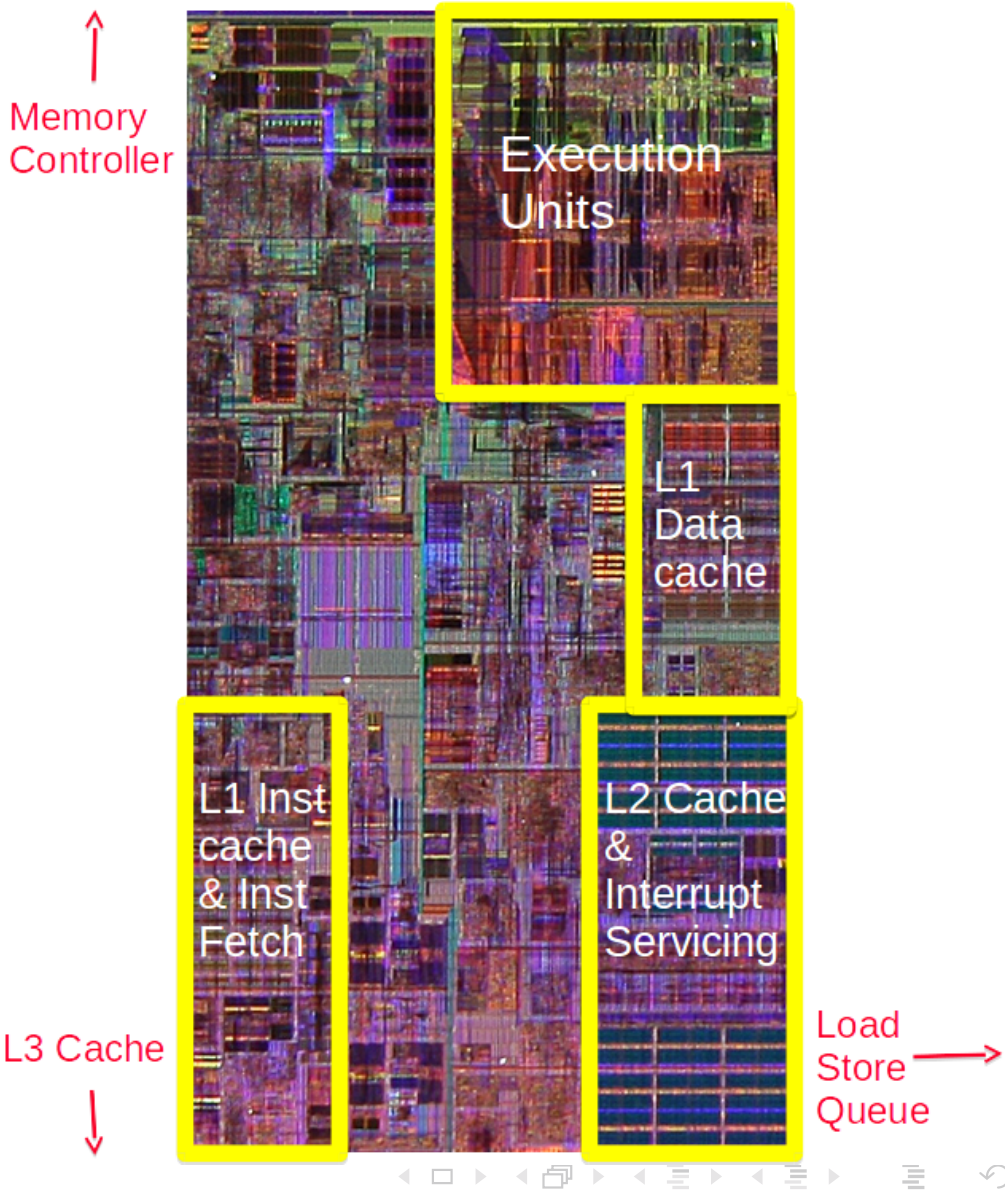
Core Area Breakdown

32KB I$ per core

32KB D$ per core

512KB L2$ per core

Share one 2-24MB L3$



What is a Cache?

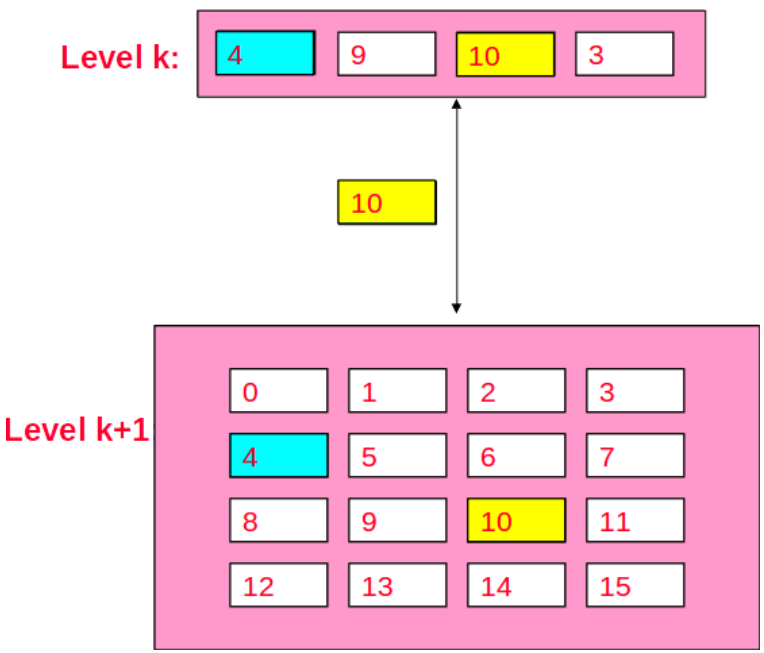
Cache: A small and fast (usually SRAM) storage device that acts as a staging area between the CPU and the larger and slower main memory.

- Fundamental idea of a memory hierarchy:
  - ↳ For each k, the fast and small device at level k serves as cache for the larger and slower device at level $k+1$.
  - ↳ The data in cache at level k is always a subset of the data available at level $k+1$.

Why do memory hierarchies work?

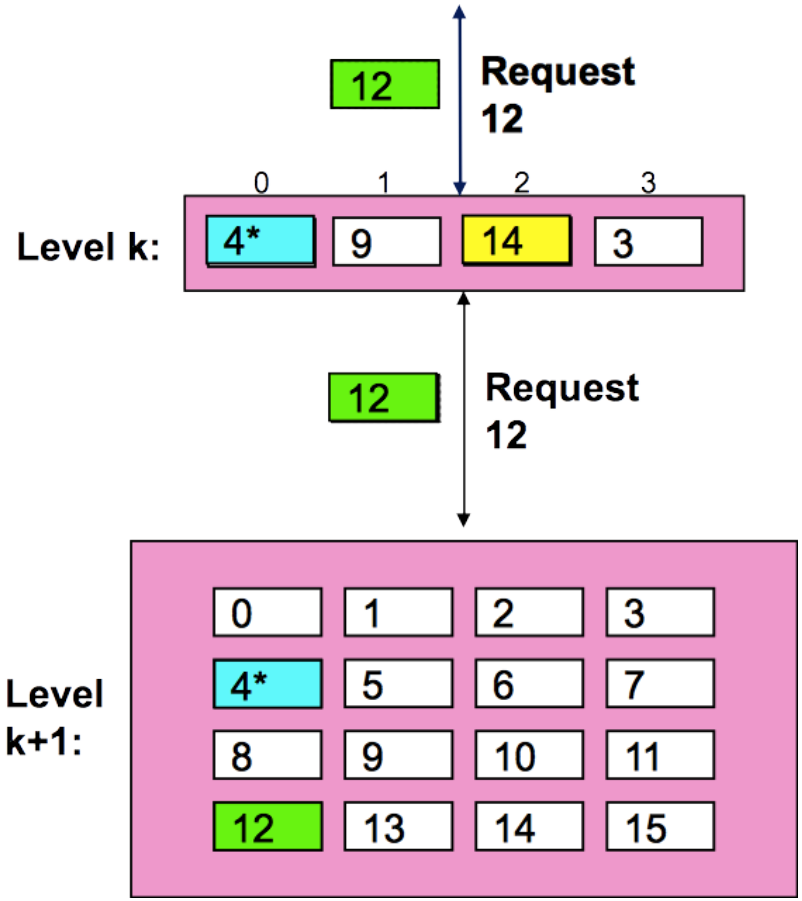
- Programs tend to access data at level k more often than they access data at level $k+1$ (locality!)
- Storage at level $k+1$ can be slower, larger, and cheaper per byte.
- Net effect: Large pool of memory that costs as little as the cheap storage near the bottom, but that serves data to programs at $\approx$ rate of the fast storage near the top.

Caching in a Memory Hierarchy



- Each level of memory is partitioned into blocks of consecutive bytes.
- Each block within a level of cache is the same size.
- The smaller, faster, more expensive storage-device at level k *caches* a subset of the data from level $k+1$
- If block-size is same between two levels of cache, the subset of data is actually a subset of blocks.
- Data is copied between levels in block-sized transfer units.

Cache Hits and Cache Misses



The processor requests a memory address contained in a block b.

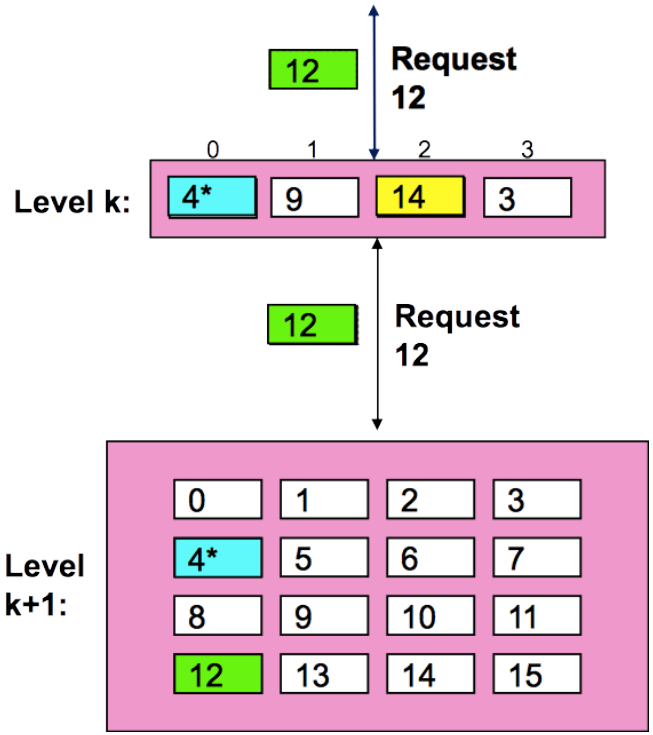
Cache hit (at level k)

- The Program finds b in the cache at level k. e.g., block 14

Cache miss (at level k)

- b is not at level k, so the level k cache must fetch it from level $k+1$.
  e.g., block 12
- If level k cache is full, then some block must be replaced (evicted) by the newly requested block. Which one is the "victim"?

Cache Policies



Which one is the "victim"?

- **Placement (mapping) policy:** where can the new block go?
  - ↳ Each block has its (one or more) slot that it fits into.
  - ↳ If all slots it fits into are full, overwrite whatever is in that slot.
  - ↳ e.g., $b \mod 4$, a "mod-4 mapping"

- **Replacement policy:** which block should be evicted? Replace a block based on some policy:
  - ↳ LRU (least recently used).
  - ↳ FIFO (first in, first out).

Cache Misses Explained (1/2)

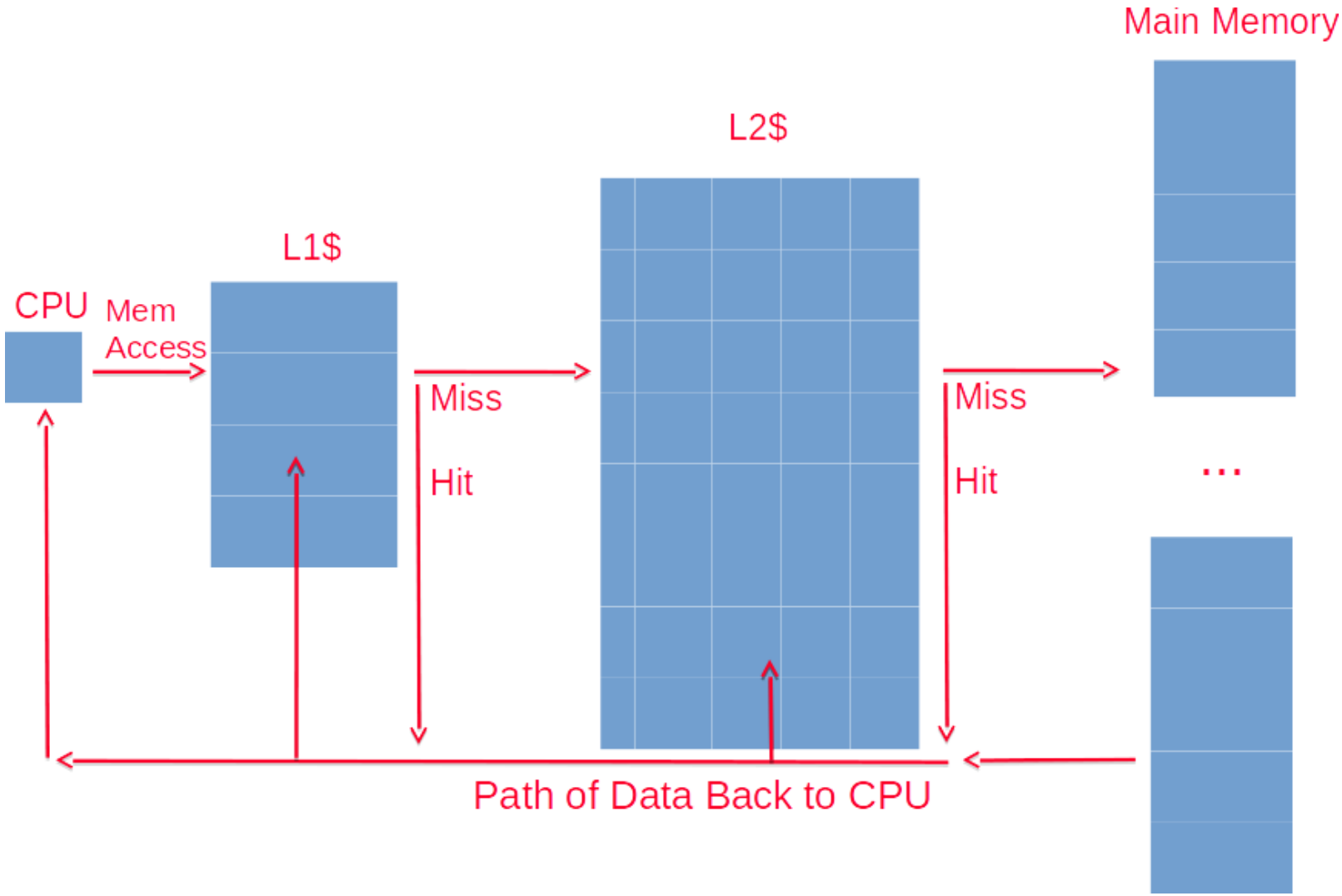
- **Cold (compulsory) Miss** (at level k)
  - A cold miss occurs at level k for a block b when this block is missing *for the first time* at level k cache (i.e. it is the first request for block b).
  - Cold misses always occur.

- **Capacity Miss** (at level k)
  - Occurs when the set of active blocks (that is, the data set with which the program is actively working on) is larger than the size of the cache at level k.
  - Can occur under a mapping policy or a replacement policy.
  - Essentially a special kind of Conflict Miss when the cache happens to be full.

Cache Misses Explained (2/2)

- **Conflict Miss** (at level k)
  - Under a mapping policy caches limit the positions a block can be placed to a small subset (sometimes a singleton) of the total available positions.
  - Conflict misses occur at level k when multiple data items from level $k+1$ all map to the same position at level k
  - e.g. Using a mod-4 mapping policy, the blocks 13 and 25 both map to index 1.
  - *Note:* **thrashing** can occur if two blocks are fighting for the same slot.
  - e.g. Requesting blocks 0, 8, 0, 8, 0, 8, ... would cause a miss every time under a mod-4 mapping.

Multiple Cache Levels



Caveat: While cache levels are numbered 1, 2, etc. it is common to call a cache at level $k+1$ a *lower level cache* with respect to a cache at level k.

Outline

CPU Time with Caching

Simplified CPU Time:

$$\text{CPU Time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

↳ Here, CPI is $\text{CPI}_{\text{ideal}}$.

CPU Time with Memory:

$$\begin{aligned} \text{CPU time} &= \text{IC} \times \text{CPI} \times \text{CC} \\ &= \text{IC} \times (\underbrace{\text{CPI}_{\text{ideal}} + \text{Average memory stall cycles}}_{\text{CPI}_{\text{stall}}}) \times \text{CC} \end{aligned}$$

Average memory stall cycles = access_count × miss_rate × miss_penalty

Note: It is generally assumed that cache hit time is included as part of CPI for load/store instructions.

Quantifying Cache Hits and Misses

- **Hit Rate**: The percentage of memory accesses which are found in a given level of the memory hierarchy.
- **Hit Time**: The time to retrieve data during memory access at a given level of the memory hierarchy, consisting of:

  Time to determine hit/miss + Time to access/transmit the block.

- **Miss Rate**: The percentage of memory accesses which are not found in a given level of the memory hierarchy, that is, 1 - (Hit Rate).
- **Miss Penalty**: The time required to search and retrieve the requested block from a lower (and slower) level of cache.

  Time to determine hit/miss
  + Time to access the block in the lower level
  + Time to transmit that block back to the current level
  + Time to insert the block in that level
  + Time to pass the block to the requester

Hit Time $\ll$ Miss Penalty

Impacts of Cache Performance

$$\text{CPI}_{\text{stall}} = \text{CPI}_{\text{ideal}} + \text{Average memory stall cycles}$$

$$\text{Avg. mem. stall cycles} = \text{access\_count} \times \text{miss\_rate} \times \text{miss\_penalty}$$

- The relative cost for a cache miss increases as processor performance increases (↑ clock rate, ↓ CPI).
  - ↳ Cache miss penalty, $\text{CPI}_{\text{stall}}$ *measured in clock cycles*.
  - ↳ The lower the $\text{CPI}_{\text{ideal}}$ the more pronounced the effect of cache miss.
- For a fixed CPU (and main memory), cannot change miss_penalty.
- Stall cycles mainly affected by program and compiler:
  - ↳ number of memory accesses
  - ↳ miss_rate: locality!
- This calculation assumes an **idealized cache**: one level of cache between CPU and main memory.

CPI Example (1/2)

A program running on a particular processor has a $\mathrm{CPI}_{\mathrm{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and a 4% miss rate. What is the average memory stall cycles? What is $\mathrm{CPI}_{\mathrm{stall}}$?

CPI Example (1/2)

A program running on a particular processor has a $\mathrm{CPI}_{\mathrm{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and a 4% miss rate. What is the average memory stall cycles? What is $\mathrm{CPI}_{\mathrm{stall}}$?

- Memory-stall cycles $= 36\% \times 4\% \times 100 = 1.44$
- So $\mathrm{CPI}_{\mathrm{stall}} = 2 + 1.44 = 3.44$

CPI Example (1/2)

A program running on a particular processor has a $\mathrm{CPI}_{\mathrm{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and a 4% miss rate. What is the average memory stall cycles? What is $\mathrm{CPI}_{\mathrm{stall}}$?

- Memory-stall cycles $= 36\% \times 4\% \times 100 = 1.44$
- So $\mathrm{CPI}_{\mathrm{stall}} = 2 + 1.44 = 3.44$

What if the previous miss rate is broken down as a 2% instruction-cache miss rate and 4% data-cache miss rate?

CPI Example (1/2)

A program running on a particular processor has a $\mathrm{CPI}_{\mathrm{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and a 4% miss rate. What is the average memory stall cycles? What is $\mathrm{CPI}_{\mathrm{stall}}$?

- Memory-stall cycles $= 36\% \times 4\% \times 100 = 1.44$
- So $\mathrm{CPI}_{\mathrm{stall}} = 2 + 1.44 = 3.44$

What if the previous miss rate is broken down as a 2% instruction-cache miss rate and 4% data-cache miss rate?

- Memory-stall cycles $= 1 \times 2\% \times 100 + 36\% \times 4\% \times 100 = 3.44$
- So $\mathrm{CPI}_{\mathrm{stall}} = 2 + 3.44 = 5.44$

CPI Example (2/2)

A program running on a particular processor has a $\mathrm{CPI}_{\mathrm{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and a 4% miss rate. What is the average memory stall cycles? What is $\mathrm{CPI}_{\mathrm{stall}}$?

What if the $\mathrm{CPI}_{\mathrm{ideal}}$ is reduced to 1?

What if the data-cache miss rate went up by 1%? (Instruction-cache miss rate still 2%)

CPI Example (2/2)

A program running on a particular processor has a $\mathrm{CPI}_{\mathrm{ideal}}$ of 2, a 100 cycle miss penalty, 36% load/store instr's, and a 4% miss rate. What is the average memory stall cycles? What is $\mathrm{CPI}_{\mathrm{stall}}$?

What if the $\mathrm{CPI}_{\mathrm{ideal}}$ is reduced to 1?

What if the data-cache miss rate went up by 1%? (Instruction-cache miss rate still 2%)

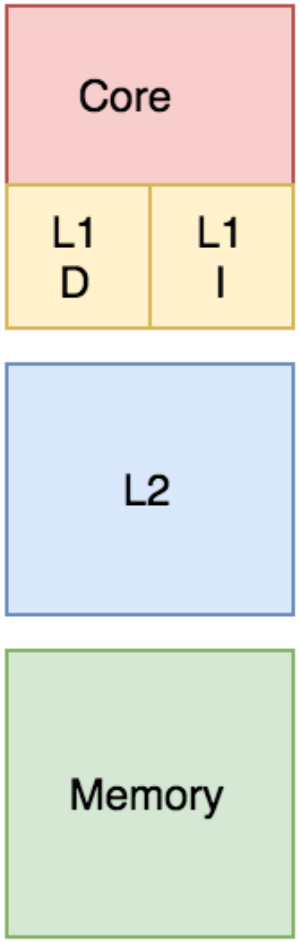
- Memory-stall cycles $= 1 \times 2\% \times 100 + 36\% \times 5\% \times 100 = 3.80$
- So $\mathrm{CPI}_{\mathrm{stall}} = 2 + 3.80 = 5.80$

Aside: Banked and Unified Caches

Banked Cache: A cache that is divided into two sections: one for instructions and one for data.

- ↳ L1 cache usually banked.
- ↳ Allows CPU to easily (and simultaneously) access both instructions and data.
- ↳ Remember this when we talk about pipelining later.

Unified Cache: A cache where instructions and data are stored together and likely intermixed.



Another Memory Performance Metric: **AMAT**

Cache Miss Rate: $\frac{\text{number of cache misses}}{\text{total number of cache references (accesses)}}$

↳ Miss rate + Hit rate = 1.0 (100%)

Miss Penalty: the access time of a memory level one below the given memory level, where a requested block is missing.

Average Memory Access Time (AMAT) is the average time to access memory considering both hits and misses.

$$\text{AMAT} = \text{Time for a Hit} + \text{Miss Rate} \times \text{Miss Penalty}$$

Note: In contrast to $\text{CPI}_{\text{stall}}$ this is measured in *time*.

Simple AMAT Example

What is the AMAT for a processor with a 200 ps clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

Simple AMAT Example

What is the AMAT for a processor with a 200 ps clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

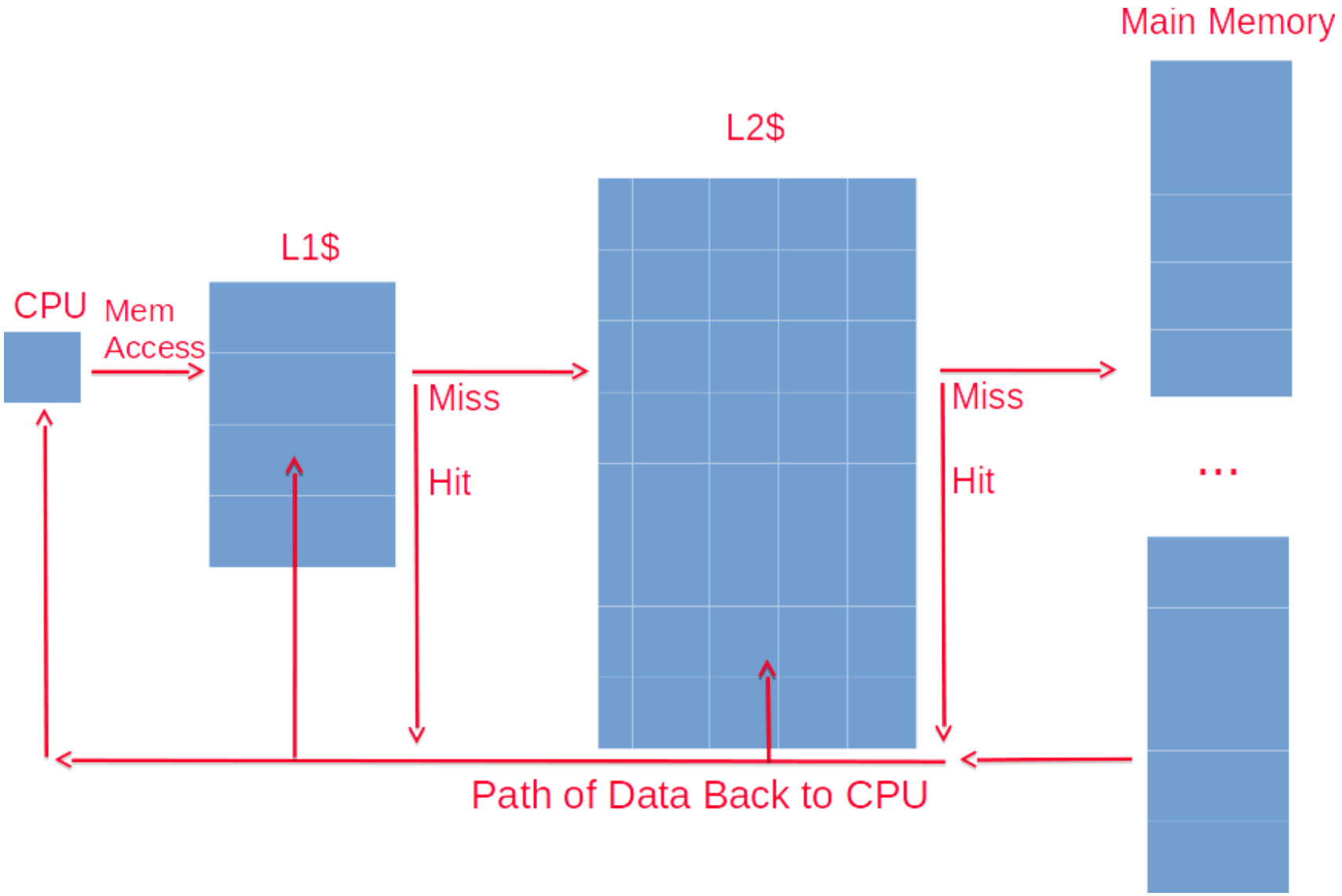
$$\text{AMAT} = \text{Time for a Hit} + \text{Miss Rate} \times \text{Miss Penalty}$$

Simple AMAT Example

What is the AMAT for a processor with a 200 ps clock, a miss penalty of 50 clock cycles, a miss rate of 0.02 misses per instruction and a cache access time of 1 clock cycle?

$$\begin{aligned} \text{AMAT} &= \text{Time for a Hit} + \text{Miss Rate} \times \text{Miss Penalty} \\ &= 1 + 0.02 * 50 = 2 \text{ clock cycles, or } 2 * 200 = 400 \text{ ps} \end{aligned}$$

Multiple Cache Levels (again)



AMAT & Multiple Cache Levels

Given the memory hierarchy, speed at each level differs. Must calculate miss penalty on a *per level* basis.

Miss penalties defined per cache level:

$$\begin{aligned}
\text{L1 Miss Penalty} &= \text{L2 Hit Time} + \text{L2 Miss Rate} \times \text{L2 Miss Penalty} \\
\text{L2 Miss Penalty} &= \text{L3 Hit Time} + \text{L3 Miss Rate} \times \text{L3 Miss Penalty} \\
\text{L3 Miss Penalty} &= \text{Main Memory Hit Time (usually)}
\end{aligned}$$

New AMAT:

$$\begin{aligned}
\mathbf{AMAT} &= \text{L1 Hit Time} + \text{L1 Miss Rate} * \text{L1 Miss Penalty}, \\
\text{L1 Miss Penalty} &= \text{L2 Hit Time} + \text{L2 Miss Rate} * \text{L2 Miss Penalty}, \\
&\vdots
\end{aligned}$$

New AMAT Example

Calculate AMAT given:

- 200 ps clock cycle,
- 1 cycle L1 hit time, 2% L1 miss rate,
- 5 cycle L2 hit time, 5% L2 miss rate,
- 100 cycle main memory access time.

Without L2 cache:

With L2 cache:

New AMAT Example

Calculate AMAT given:

- 200 ps clock cycle,
- 1 cycle L1 hit time, 2% L1 miss rate,
- 5 cycle L2 hit time, 5% L2 miss rate,
- 100 cycle main memory access time.

Without L2 cache:

$$\text{AMAT} = 1 + .02 \times 100 = 3 \text{ cycles, } 3 \times 200 = 600 \text{ ps}$$

With L2 cache:

New AMAT Example

Calculate AMAT given:

- 200 ps clock cycle,
- 1 cycle L1 hit time, 2% L1 miss rate,
- 5 cycle L2 hit time, 5% L2 miss rate,
- 100 cycle main memory access time.

Without L2 cache:

$$\text{AMAT} = 1 + .02 \times 100 = 3 \text{ cycles}, \; 3 \times 200 = 600 \text{ ps}$$

With L2 cache:

$$\text{AMAT} = 1 + .02 \times (5 + .05 \times 100) = 1.2 \text{ cycles}, \; 1.2 \times 200 = 240 \text{ ps}$$

Cache Memories & Relative Speeds

"Cache" Type	**What Cached**	**Where Cached**	**Latency** (cycles)	**Managed By**
Registers	4/8-byte word	CPU registers	0.5	Compiler
TLB	Address translations	On-Chip TLB	0.5	Hardware
L1 cache	32-byte block	On-Chip L1	1	Hardware
L2 cache	32-byte block	On/Off-Chip L2	10	Hardware
Buffer cache	Parts of files	Main memory	100	OS
Virtual Memory (Paging)	4-KB page	NVMe HDD	100,000 1,000,000	Hardware+ OS
Network buffer cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

- The TLB (Translation lookaside buffer) "address-translation cache" caches virtual memory/physical memory mappings.
- The Andrew File System (AFS) and Network File System (NFS) are distributed file system protocols

Outline

Advanced Temporal Locality

```
int sumarray(int[][] a, int M, int N) {
    int i, j, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            sum += a[i][j];
    return sum;
}
```

Recall: Temporal locality of `M` depends on size of `N`?

Depending on size of cache and on size of `N`, locality is possible.

- ↳ If `sizeof(int)` $\times$ (`N` + 6) $\leq$ cache size then `M` still in cache!
- ↳ Assumes LRU cache policy and data-specific cache.

Advanced Spatial Locality

In a *row-major* language:

- Stepping through columns in one row:

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

  - Accesses successive elements of size k bytes
  - If block size (B) $> k$ bytes, exploit spatial locality
    Cold miss rate $= k$ bytes / B.
  - Typically $k = 4$ or 8 (32/64 bits) and $B = 8\,k$ or $B = 16\,k$.

- Stepping through rows in one column:

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

  - Accesses distant elements (assuming number of columns in a is large).
  - No spatial locality!
  - Cold miss rate $= 1$ (i.e. 100%)

Advanced Spatial Locality Example

Calculate the number of cache hits and cache misses given:

- A row-major programming language with `a` having `INT4` elements,
- Cache capacity of 4 Kilobytes, cache block is 32 bytes,
- All local variables are stored in registers, not cache.

```
int i, sum = 0, N = 16384;
for (i = 0; i < N; i++)
    sum += a[i];
```

Advanced Spatial Locality Example

Calculate the number of cache hits and cache misses given:

- A row-major programming language with `a` having `INT4` elements,
- Cache capacity of 4 Kilobytes, cache block is 32 bytes,
- All local variables are stored in registers, not cache.

```
int i, sum = 0, N = 16384;
for (i = 0; i < N; i++)
    sum += a[i];
```

- **Stride-1 access to `a`.**
- Since cache block is 32 bytes, each block holds $32/4 = 8$ ints.
- One cold miss for each block.
- $(1/8) \times 16384 =$ **2048 cache misses**.
- 16384 - 2048 cache misses = **14336 cache hits**.

Summary

- We want a **large, cheap, fast memory**
  - ↳ Unreasonable in size/cost for everything to be SRAM (L1).
- Solution: **Memory Hierarchy**
  - ↳ Successively lower levels contain "most used" data from next higher level.
  - ↳ Exploits **temporal & spatial locality** of programs.
  - ↳ Great Idea #3: Do the common case fast, worry less about the exceptions (RISC design principle).
- **Challenges to programmer**:
  - ↳ Develop cache friendly (efficient) programs.

Extra Reading

Being able to look at code and have a *qualitative sense of its locality* is a key skill for programmers.

Some projects driven by data locality (and other features):

- BLAS `http://www.netlib.org/blas/`
- FFTW, by Matteo Frigo and Steven G. Johnson `http://www.fftw.org/`
- M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran, *Cache-Oblivious Algorithms*, 1999.

Wikipedia is surprisingly well written on some of these topics.

- `https://en.wikipedia.org/wiki/CPU_cache`
- `https://en.wikipedia.org/wiki/Cache_hierarchy`
  `https://en.wikipedia.org/wiki/Cache-oblivious_algorithm`

CS3350B Computer Organization
Chapter 1: CPU and Memory
Part 3: Cache Implementations

Alex Brandt

Department of Computer Science
University of Western Ontario, Canada

Thursday January 17, 2019

Outline

How is the Hierarchy Managed?

Recall: CPU ← Registers ← L1 ← . . . ← Main Memory ← Disk

- Registers ↔ Cache Memory:
  - ↳ The compiler (sometimes with the programmer's help) decides which values are stored in which registers.
- Caches ↔ Main Memory:
  - ↳ The **cache controller** (thus the hardware) handles cache memory movement.
- Main Memory ↔ Disk:
  - ↳ The operating system (which controls the virtual memory).
  - ↳ the TLB (thus the hardware) which assists the virtual-to-physical address mapping.
  - ↳ The programmer (who organizes the data into files), if data comes from file.

Cache Design Questions

Q1 How best to organize the memory blocks (a.k.a lines) inside the cache?

Q2 To which block (line) of the cache does a given (main) memory address map?

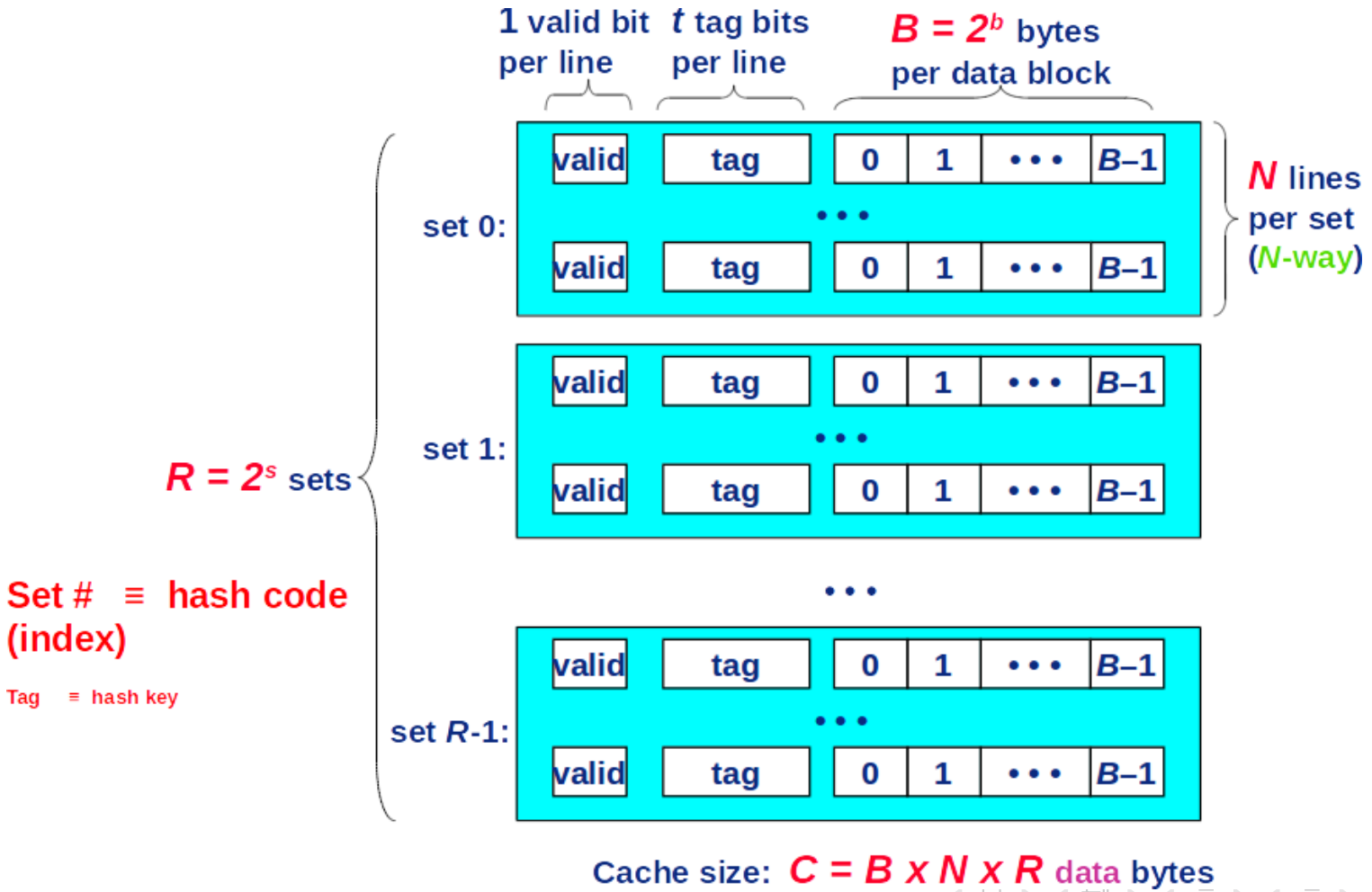
↳ *Note:* since the cache is a subset of the main memory, multiple memory addresses can map to the same cache location.

Q3 How do we know if a block of the main memory currently has a copy in cache?

Q4 How do we quickly find a particular copy of main memory (memory address contents) in the cache?

General Organization of a Cache Memory (1/2)

- Cache is an array of $R = 2^s$ sets
- Each set contains $N \geq 1$ lines
- Each **cache line** (block) holds $B = 2^b$ bytes of data

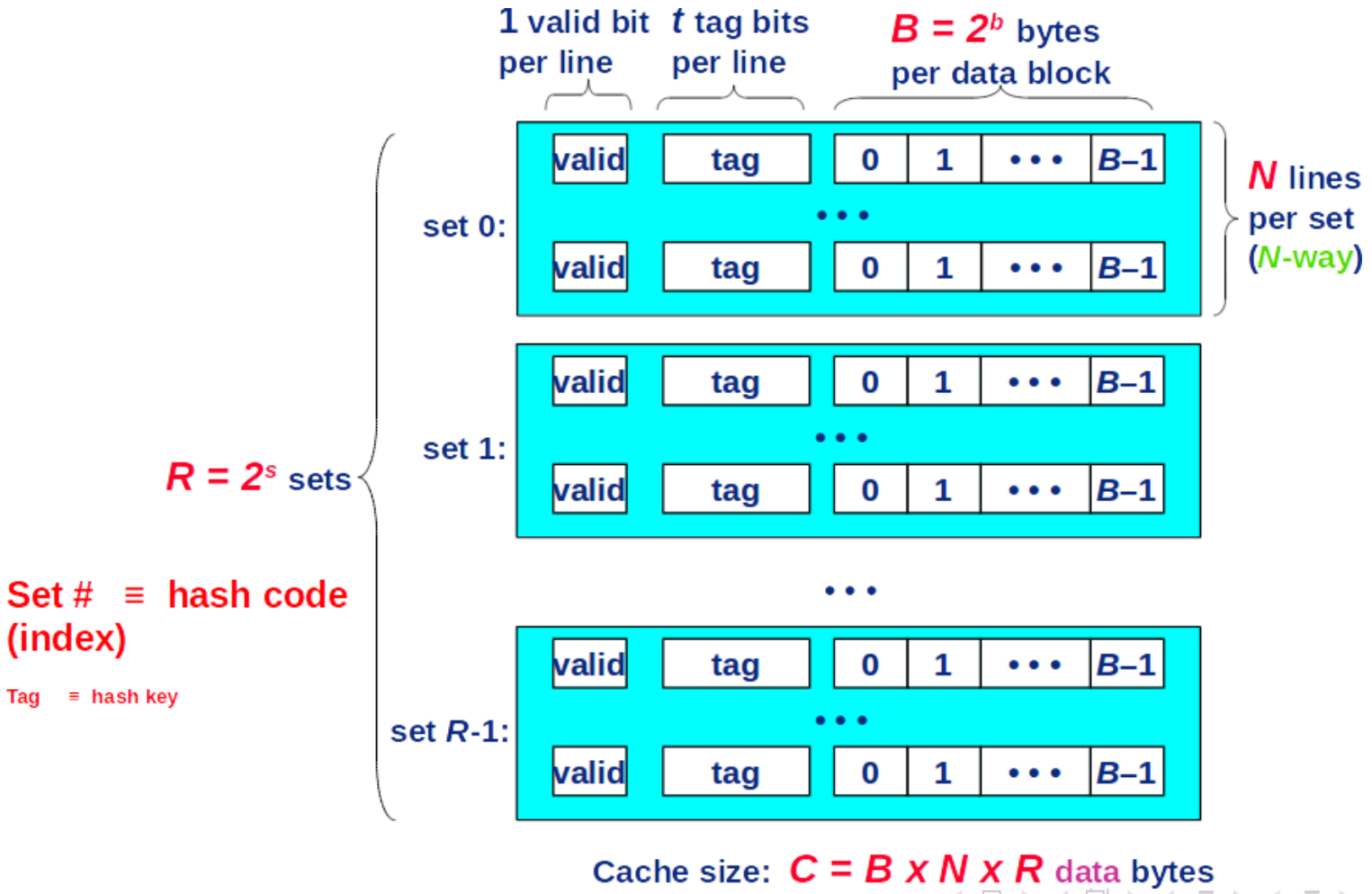


General Organization of a Cache Memory (2/2)

- **Cache size** = bytes per line × lines per set × number of sets

$$C = B \times N \times R$$

- Everything is a power of 2.



Memory-Cache Mapping (Addressing Cache Memories)

- The data word at the m-bit **address A** is in the cache if the tag bits in one of the <valid> lines in set <set index> match <tag>
- The word contents begin at offset <block offset> bytes from the beginning of the block

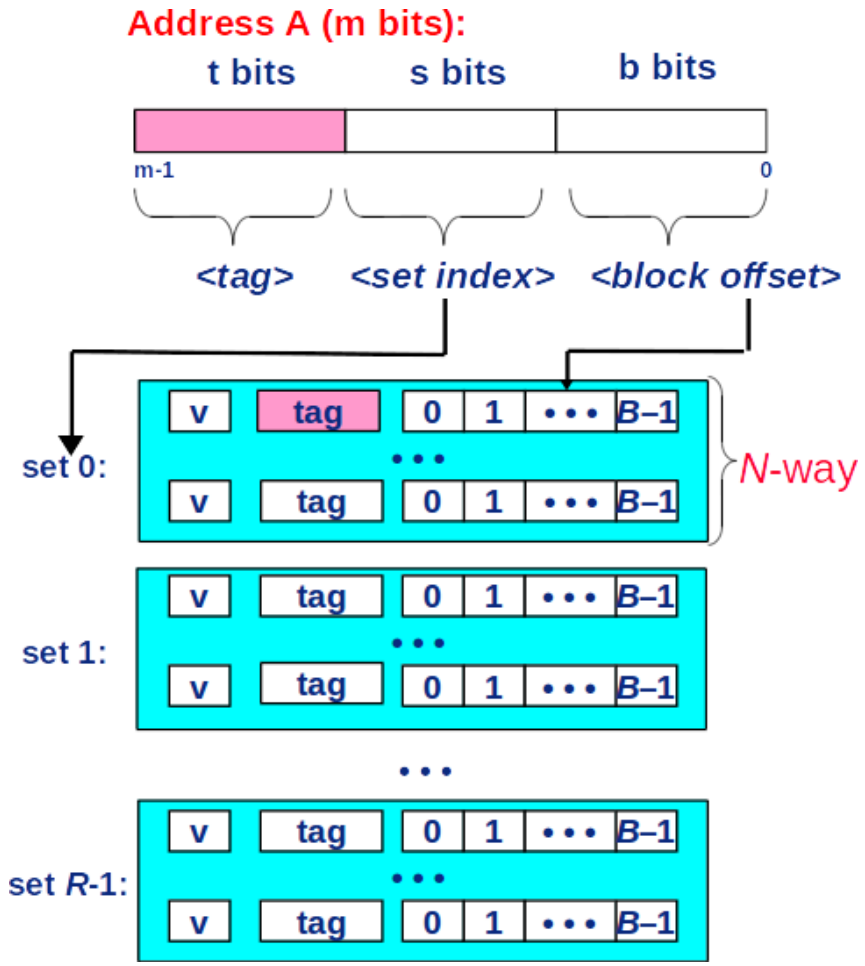
Address Mapping:

block address = <tag> || <set index>

set# = (block address) $\mod R$

↳ just take the "s bits" as set index.

Note: Main memory address space and cache size/implementation highly coupled.



$$b = \log_2(B),\ R = C/(B * N)$$
$$s = \log_2(R),\ t = m - s - b$$

Outline

Types of Cache Organization

Direct-Mapped

- **N = 1**
  - ↳ One line per set.
  - ↳ Each memory address is mapped to exactly one line in the cache.
- $b = \log_2(B)$, $R = C/B$, $s = \log_2(R)$, t (tag size) $= m - s - b$.

Fully Associative

- R = 1 (allow a memory address to be mapped to any cache block).
- Tag is whole address except block offset.
- $b = \log_2(B)$, $N = C/B$, $s = 0$, $t = m - b$.

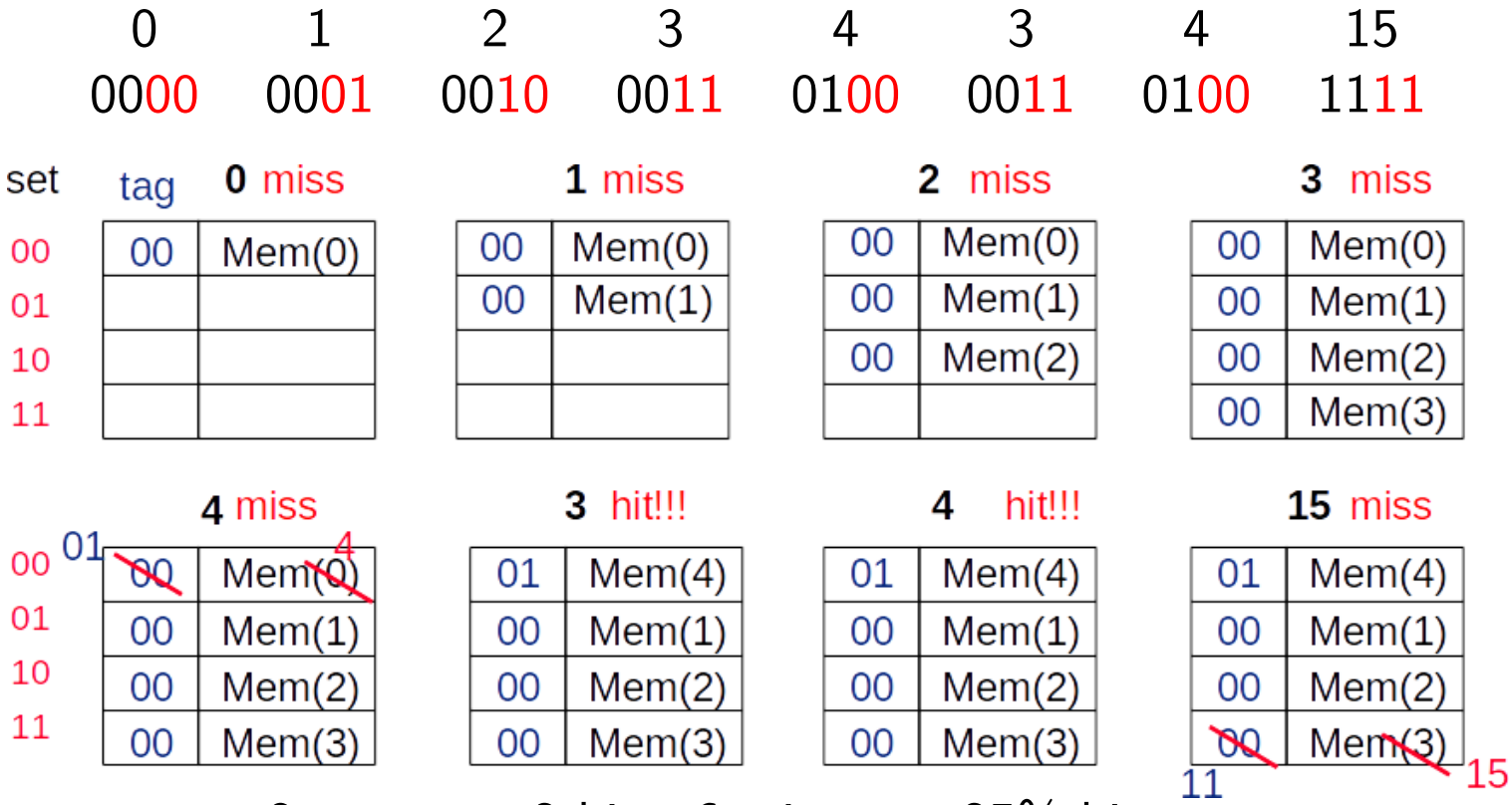
N-way set associative

- N is typically 2, 4, 8, or 16.
- A memory block maps to a specific set but can and can be placed in any **way** of that set (so there are N choices of mapping).
- $b = \log_2(B)$, $R = C/(B \times N)$, $s = \log_2(R)$, $t = m - s - b$.

Direct-Mapped Cache Example

- Direct-Mapped $\implies$ $N = 1$. Let $B = 1$, $R = 4$.
- Therefore we have a 2-bit set index. Assume a 2-bit tag $\implies$ $m = 4$.
- Start with an empty cache – blanks are considered invalid.

Consider the sequence of memory address accesses:

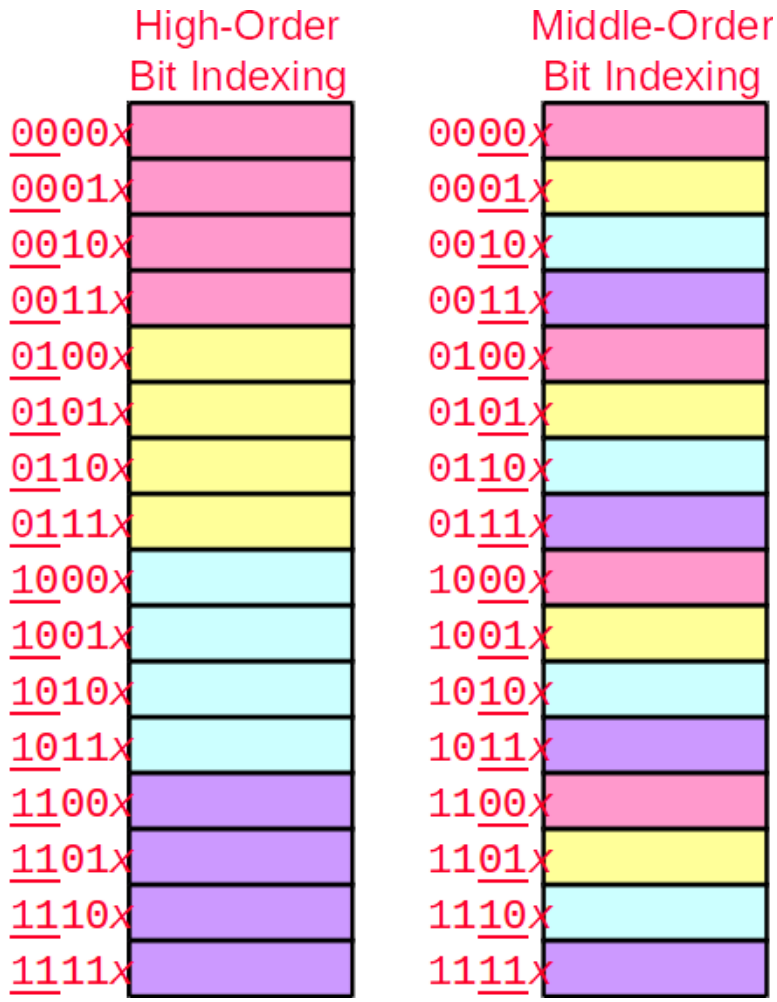


8 requests, 2 hits, 6 misses $=$ 25% hit rate

Why Middle Bits For Set Index?

Consider a 4-line direct-mapped cache; sets are indexed as 00, 01, 10, 11:

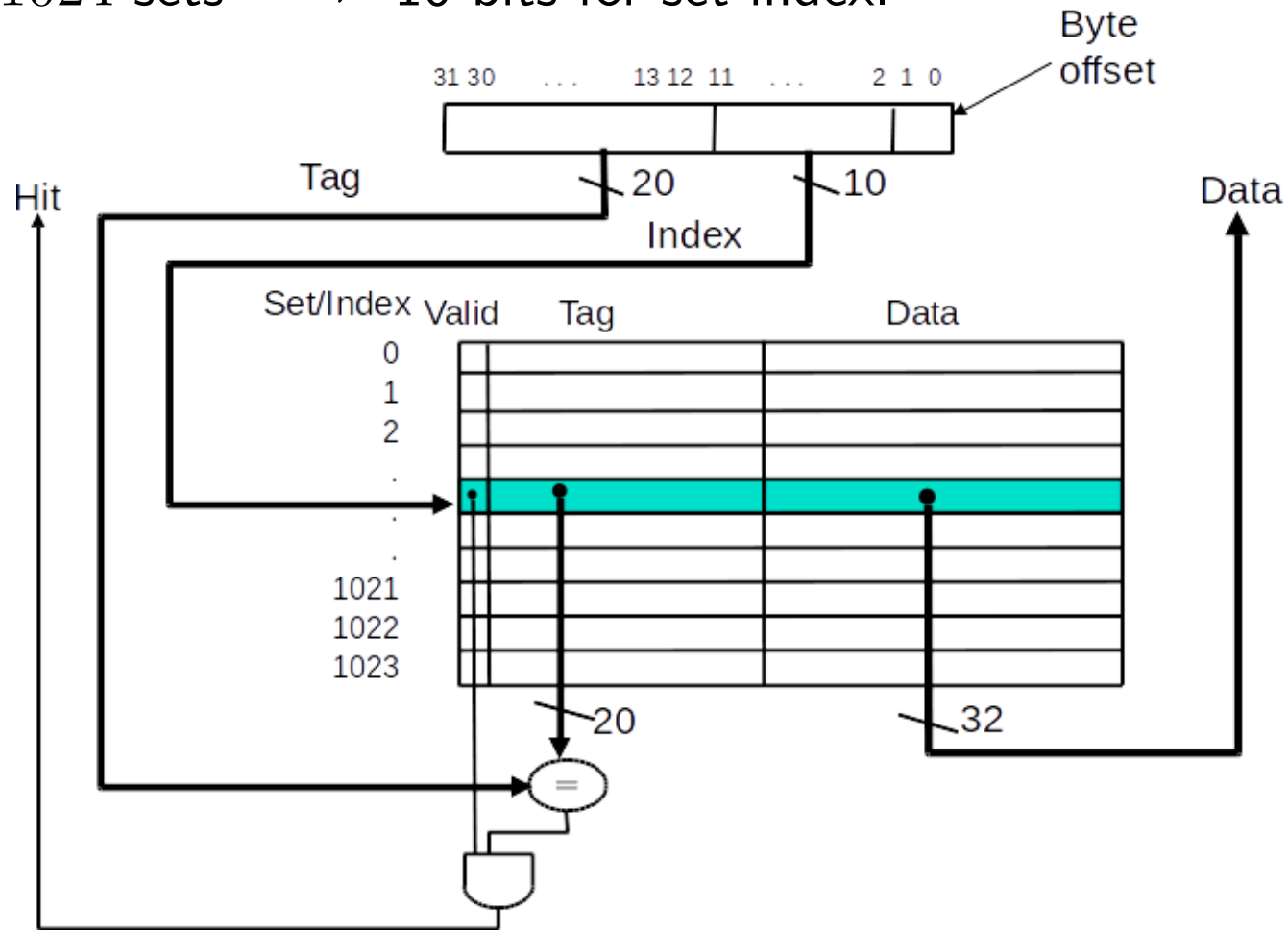
- High-Order Bit Indexing
  - ↳ Adjacent memory lines would map to same cache entry
  - ↳ Poor use of locality
- Middle-Order Bit Indexing
  - ↳ Consecutive memory lines map to different cache lines
  - ↳ Can hold C-bytes of contiguous memory in cache at one time



In this example (0000x, 0001x, . . .) with an m-bit address, x must be $m-4$ bits.

Direct-Mapped Cache Circuit Logic

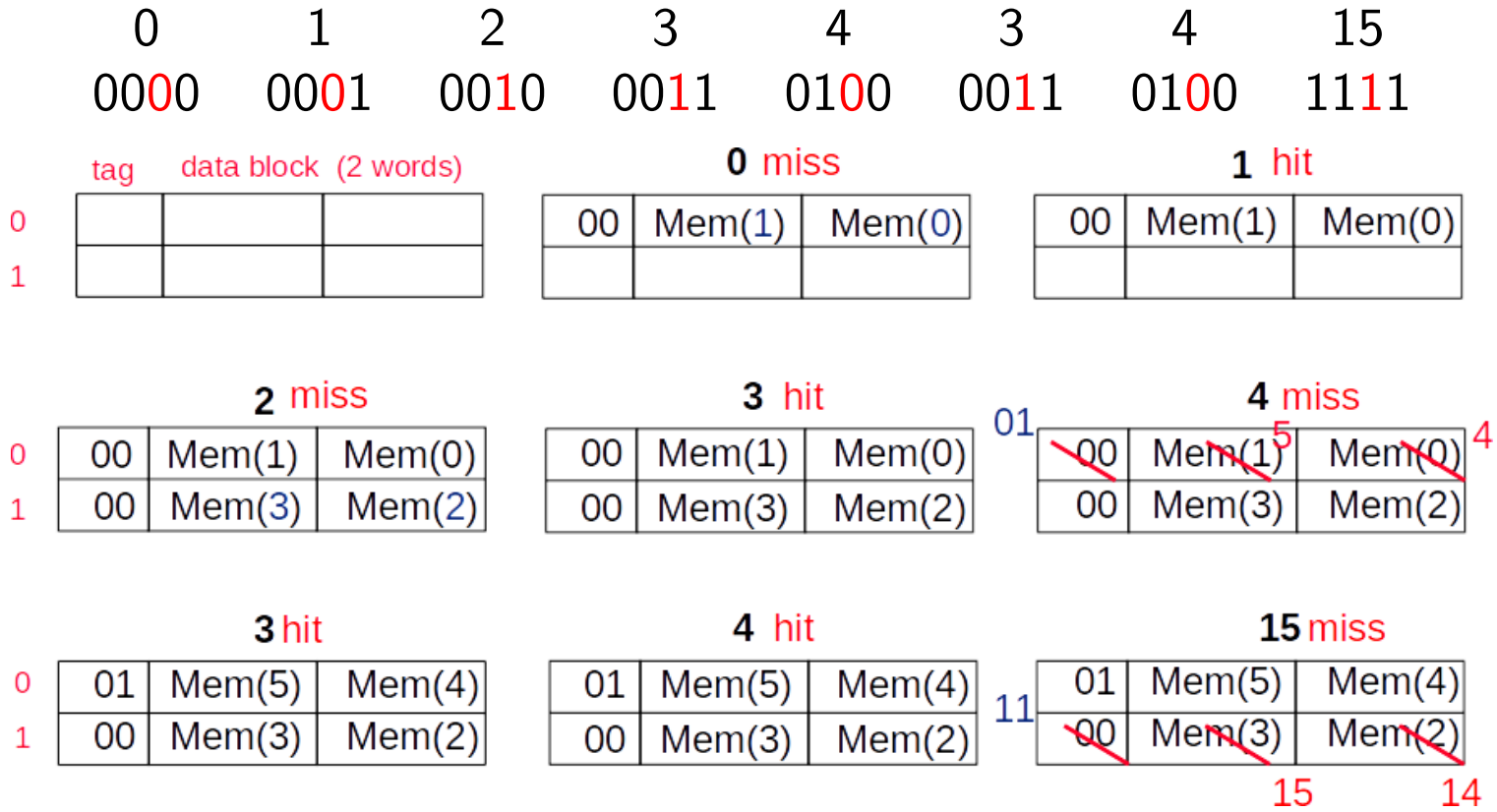
- One word (4 byte) cache lines ($B = 4$).
- Notice byte offset related to, but not exactly, block offset.
- $2^{10} = 1024$ sets $\implies$ 10 bits for set index.



Direct-Mapped Cache Example 2: $(B > 1)$

- Let the cache line hold two words $=$ bytes $(B = 2)$.
- Direct-Mapped $\implies$ $N = 1$. $R = 2$.
- Therefore 1-bit set index. Assume a 2-bit tag again.
- Start with an empty cache – blanks are considered invalid.

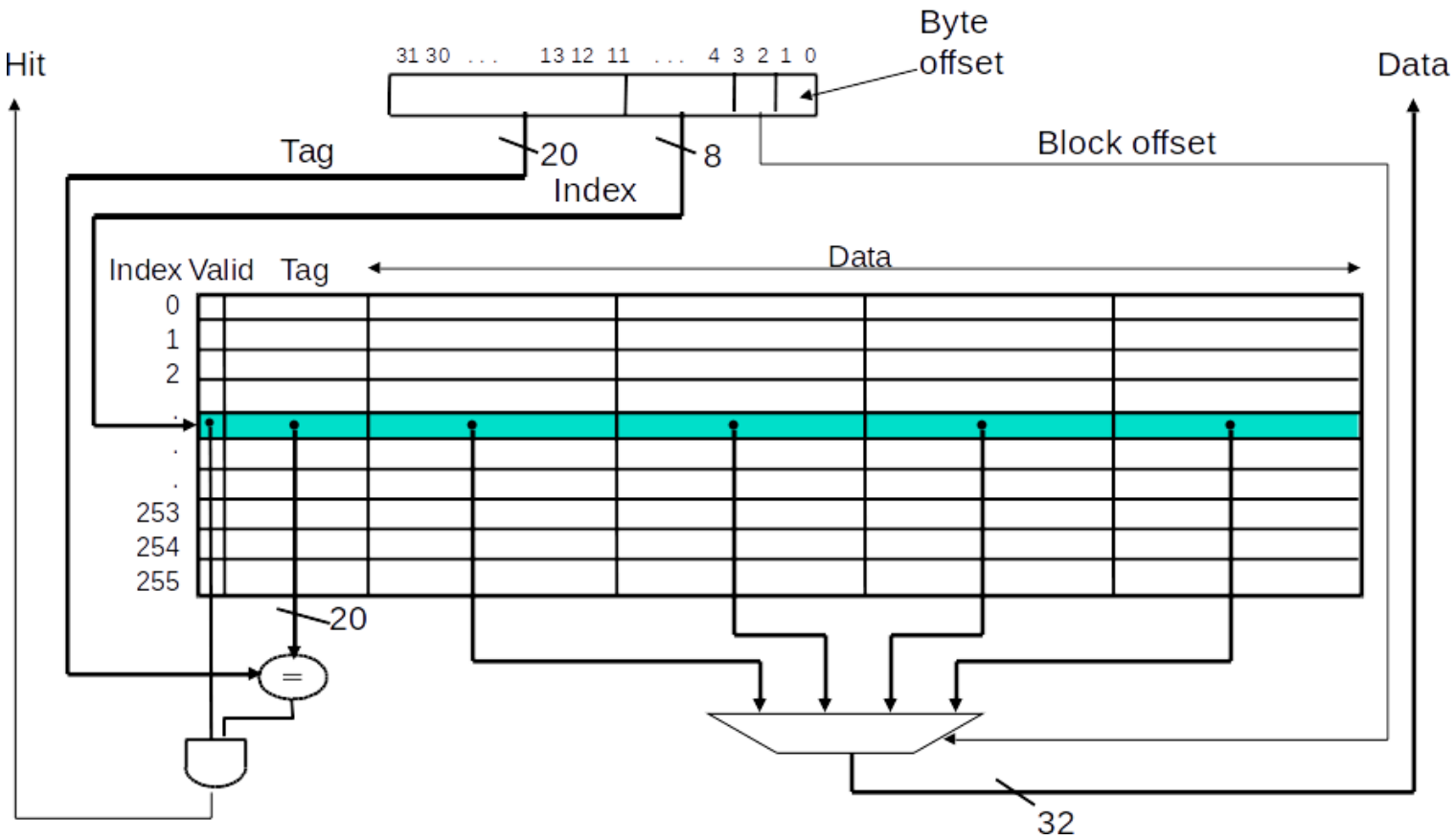
Consider the sequence of memory address accesses:



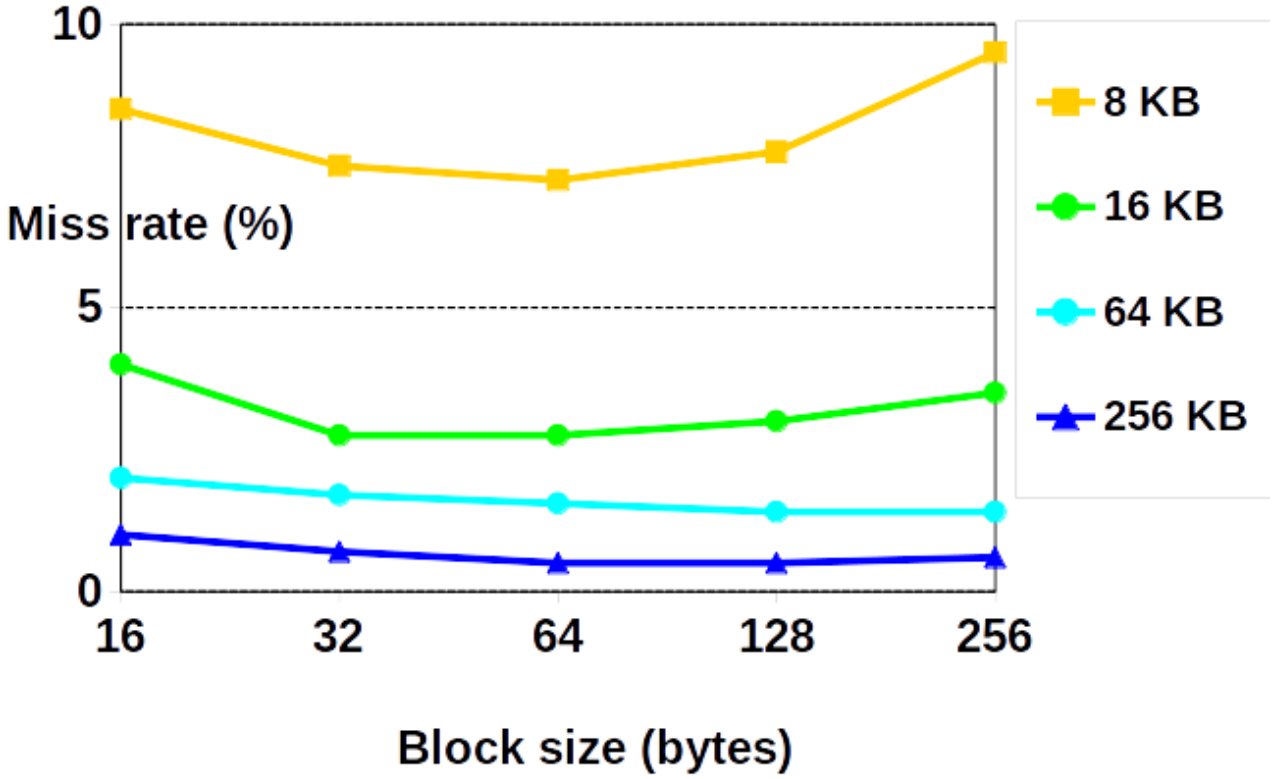
8 requests, 4 hits, 4 misses $=$ 50% hit rate!

Direct-Mapped Cache Circuit Logic 2

- Four data words/block ($B = 16$).
- $2^8 = 256$ sets $\implies$ 8 bits for set index.



Block Size & Cache Performance



Miss rate goes up if the block size becomes a significant fraction of the cache size.

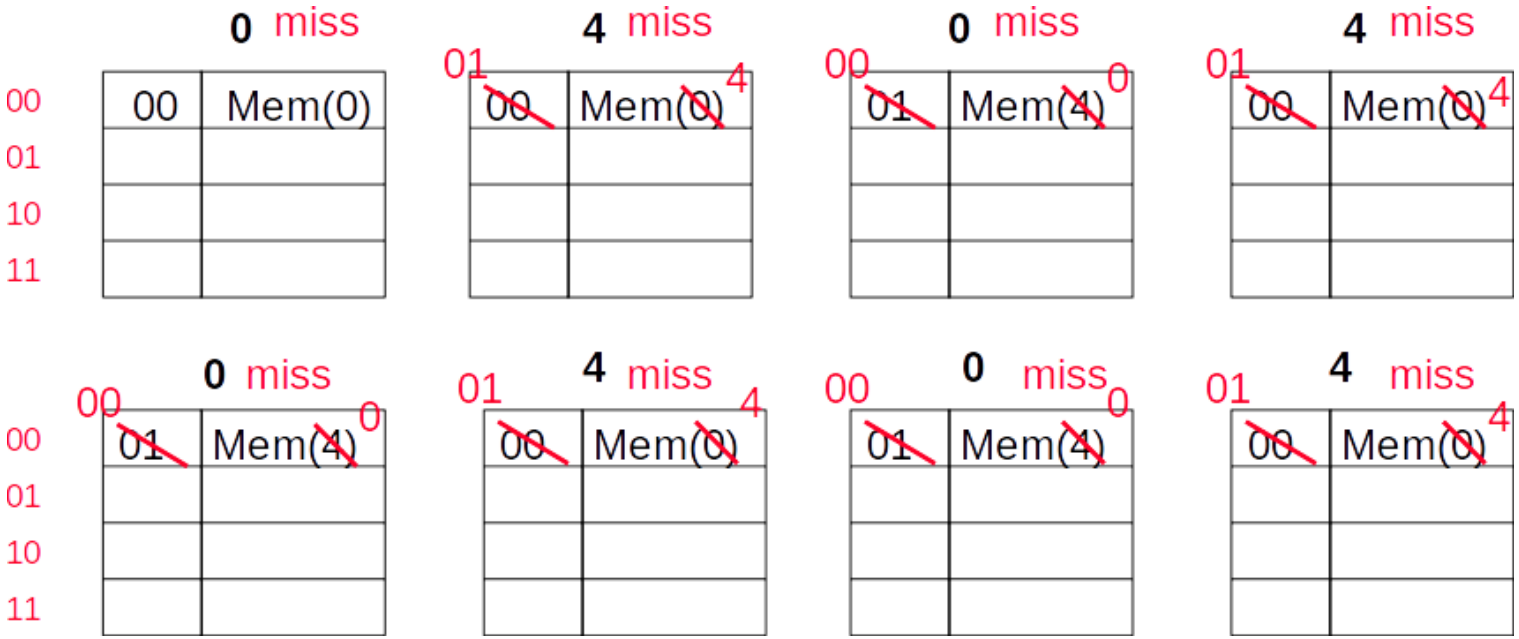
- ↳ For a fixed cache size, ↑ block size $\Longrightarrow$ ↓ number of blocks.
- ↳ Increases number of **capacity misses**.

Direct-Mapped Cache Worst-Case

Direct-Mapped $\implies$ $N = 1$. Let $B = 1$, $R = 4$.

Consider the sequence of memory addresses accesses:

$$0 = (0000),\ 4 = (0100),\ 0,\ 4,\ 0,\ 4,\ 0,\ 4,\ \ldots$$



8 requests, 8 misses = 100% miss rate!

Thrasing!

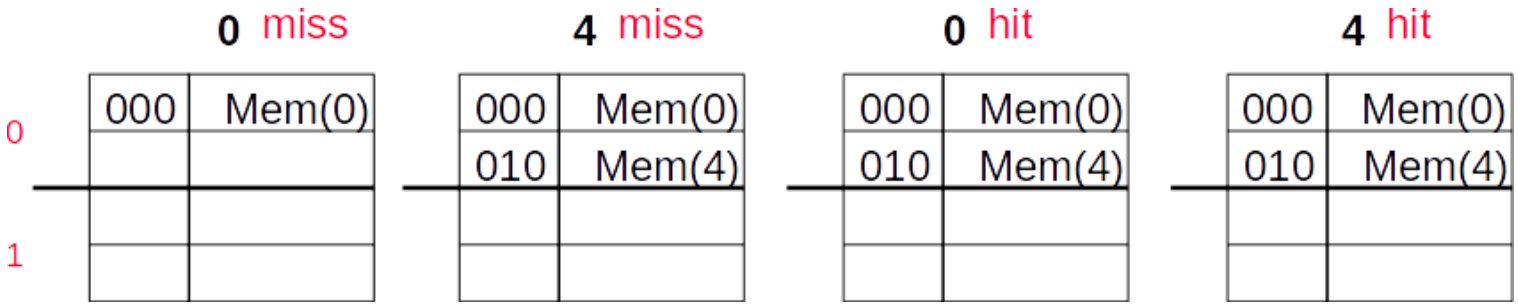
↳ **conflict misses** – two addresses map into the same cache block

Thrashing Fix with Set Associativity

2-way associative cache $\implies$ $N = 2$. Let $B = 1$, $R = 2$ now.

Consider the sequence of memory addresses accesses:

$$0 = (00\textcolor{red}{00}),\ 4 = (01\textcolor{red}{00}),\ 0,\ 4,\ 0,\ 4,\ 0,\ 4,\ \ldots$$



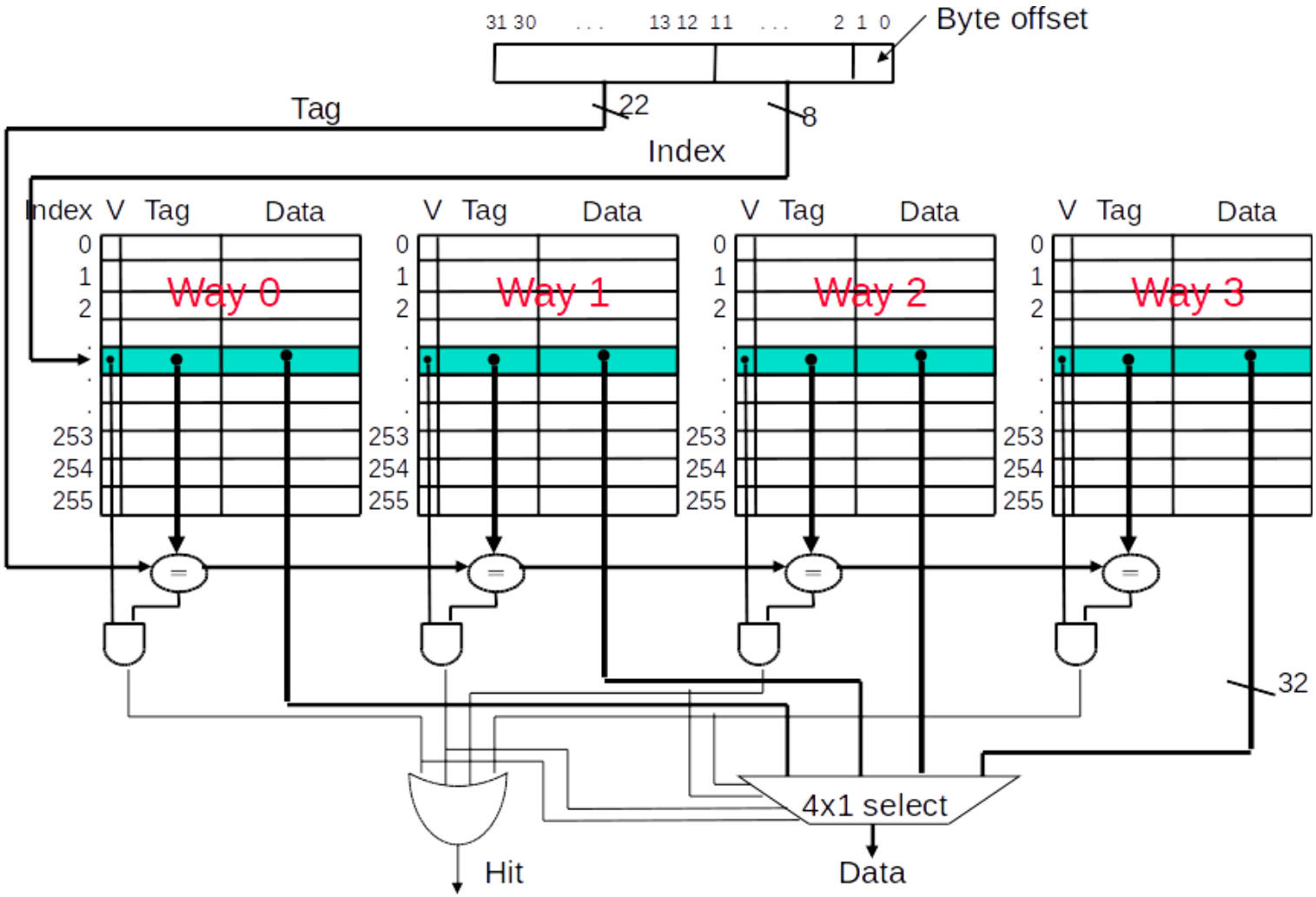
8 requests, 2 misses $=$ 75% hit rate

Solves thrashing of direct-mapped cache caused by *conflict* misses

↳ Now two memory locations that map to the same block can co-exist.

Four-way Set Associative Cache Circuit Logic

- $2^8 = 256$ sets each with four ways (each with one block of one word)



Set Associativity Costs Extra

When a miss occurs, which way do we pick for replacement?

- **Least Recently Used (LRU)**: the block replaced is the one that has been unused for the longest time.
  - ↳ Hardware must keep track of when each way was last used relative to the other blocks in the set.
  - ↳ For 2-way set associative, takes **one bit per set**.
  - ↳ Set the bit when a block is referenced (and reset the other way's bit).
- **First In First Out (FIFO)**
  - ↳ Can be implemented as a circular buffer or counter.

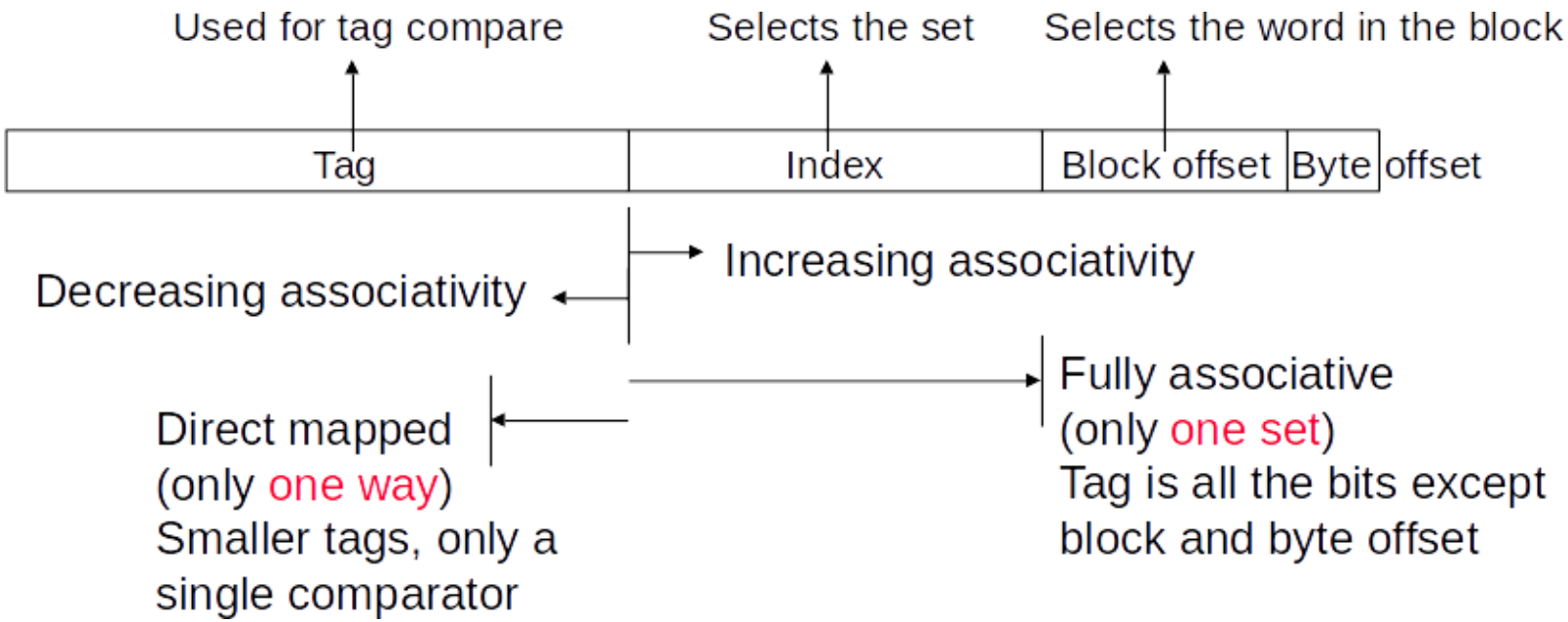
N-way set associative cache costs:

- N comparators for LRU policy (delay and physical area on chip).
- MUX delay (selecting the proper way) before data is available.
- In a direct mapped cache, the cache block is available *before* the hit/miss decision/
  - ↳ Impossible to assume a hit and recover later in set associative caches.

Range of Set Associative Caches

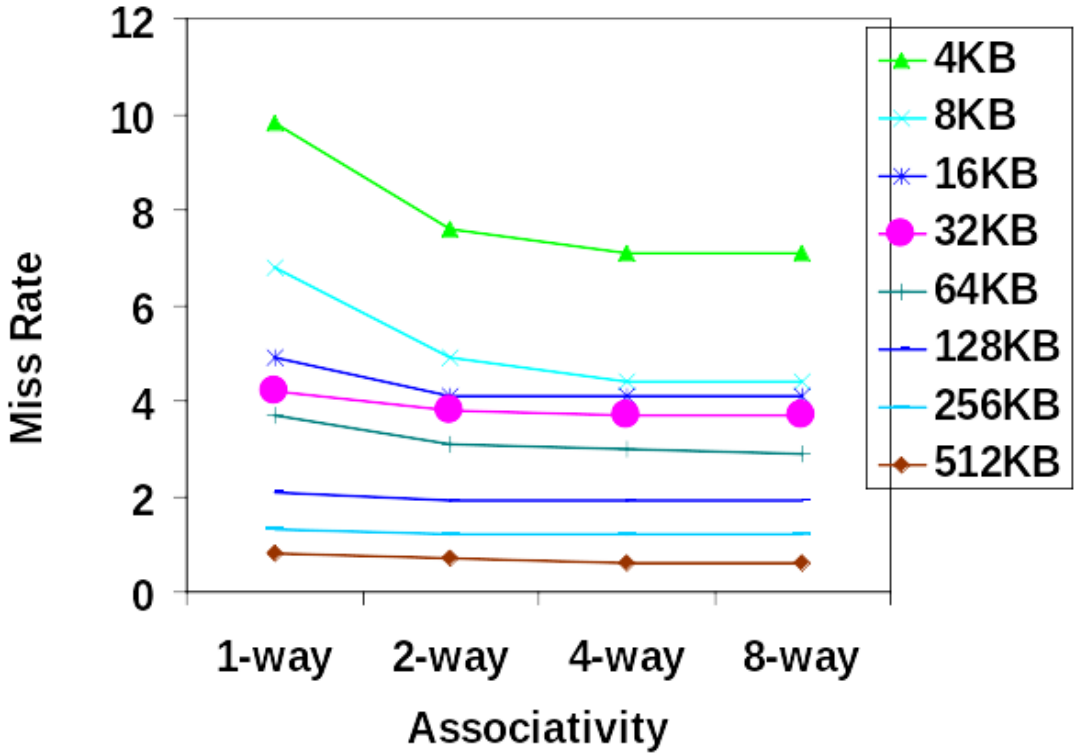
For a **fixed size cache** (and fixed size of cache lines) each doubling in associativity (ways):

- Doubles number of blocks per set,
- Halves number of sets,
- Decreases size of set-index by 1 bit,
- Increases size of tag by 1 bit.



Benefits of Set Associative Caches

The choice between *direct mapped* (1-way) and *set associative* depends on the cost of a miss versus the cost of implementation.



↳ Largest gains are in going from direct mapped to 2-way (>20% reduction in miss rate).

Cache Structure in Different Processors

What do we know so far?

	Intel Nehalem	**AMD Barcelona**
L1 cache size & organization	32KB for each per core; 64B blocks; Split I$ and D$	64KB for each per core; 64B blocks; Split I$ and D$
L1 associativity	4-way (I), 8-way (D) set assoc.; ~LRU replacement	2-way set assoc.; LRU replacement
L1 write policy	write-back, write-allocate	write-back, write-allocate
L2 cache size & organization	256KB per core; 64B blocks; Unified	512KB per core; 64B blocks; Unified
L2 associativity	8-way set assoc.; ~LRU	16-way set assoc.; ~LRU
L2 write policy	write-back, write-allocate	write-back, write-allocate
L3 cache size & organization	8192KB (8MB) shared by cores; 64B blocks; Unified	2048KB (2MB) shared by by cores; 64B blocks; Unified
L3 associativity	16-way set assoc.	32-way set assoc.; evict block shared by fewest cores
L3 write policy	write-back, write-allocate	write-back, write-allocate

Outline

Handling Cache Hits

Read Hits (Instructions and Data)

- This is what we want! $\Longrightarrow$ Do nothing special.

Write Hits (Data only) has two policies:

- **Write-through:** Require the cache and backing memory to always be **consistent**.
  - ↳ When writing to a cache, also pass the value to the next lower level cache (or main memory) to update its copy.
  - ↳ In naïve implementation this is very slow. Speed of cache writes limited by the speed of the next lower level.
  - ↳ Use a **write buffer** between levels and stall only if buffer is full.
- **Write-back:** Allow cache and memory to be **inconsistent**
  - ↳ Write data into the cache block, this becomes a **dirty block**.
  - ↳ Only update the next lower level when a dirty block is evicted.
  - ↳ Requires **two cycles** – one to check for evict/dirty and another to actually do write – or again use a **write buffer** and use only one cycle.

Handling Cache Misses

Read Misses (Instruction and Data)

- **Stall** the execution, fetch block from the next lower level of memory, write ("install") it into cache, pass it to next higher level of memory (or the processor), and let processor resume.

Write Misses (Data only) **stall** execution and perform one of two policies:

- **Write allocate:**
  - ↳ Fetch the block from the next level in the memory hierarchy,
  - ↳ Install it in the cache and write the updated word into the new block
  - ↳ Installing block and writing the updated word can be done simultaneously.
- **No-Write Allocate:** Skip fetching/installing block into cache, write directly into lower level of memory (or a write buffer).

then let the processor resume.

Dealing With Cache Misses with Hardware Design

Compulsory Misses:

- Caused by **cold** starts, process migrations or very first references.
- Reduce impact by

Capacity Misses:

- Caused by the cache becoming full; it cannot hold all blocks referenced by the program.
- Reduce impact by

Conflict Misses:

- Caused by multiple addresses mapping to the same cache block.
- Reduce impact by

Dealing With Cache Misses with Hardware Design

Compulsory Misses:

- Caused by **cold** starts, process migrations or very first references.
- Reduce impact by increasing block size. But this causes increased miss penalty and could increase miss rate.

Capacity Misses:

- Caused by the cache becoming full; it cannot hold all blocks referenced by the program.
- Reduce impact by increasing cache size. But this may increase access time.

Conflict Misses:

- Caused by multiple addresses mapping to the same cache block.
- Reduce impact by increase associativity or increasing cache/block size and/or increase associativity (may increase access time)
  - ↳ Larger cache $\Longrightarrow$ more sets $\Longrightarrow$ fewer addresses map to same loc.

Outline

Reducing effects of Cache Miss

Reducing Cache Miss Rate $\Longrightarrow$ increase cache size

- With increasing technology (especially transistor size and density) there is more room for larger caches.

Reducing Cache Miss Penalty $\Longrightarrow$ use more levels of cache

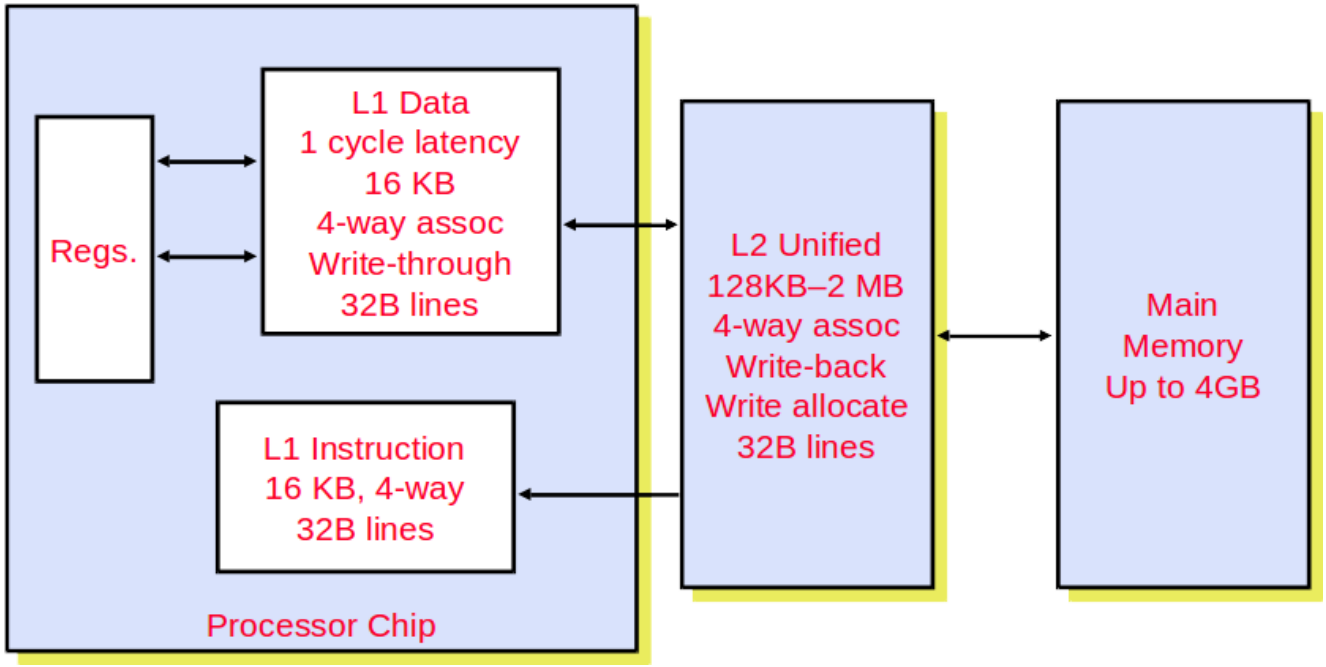
- L1 cache around for a *long* time.
- L2 cache first appeared with Intel's Celeron processors with only 128KB (1999).
- L3 was first seen in 2003 but prohibitively expensive. Became standard with Intel's Nehalem in 2008.

Recall: New AMAT Example

1 cycle L1 hit time, 2% L1 miss rate, 5 cycle L2 hit time, 5% L2 miss rate,100 cycle main memory access time

- Without L2 cache: AMAT = 1 + .02*100 = 3
- With L2 cache: AMAT = 1 + .02*(5 + .05*100) = 1.2

Intel Pentium Memory Hierarchy



Multilevel Cache Design

Design considerations for L1 and L2 caches are very different:

- Primary cache should focus on **minimizing hit time** in support of faster processor clock.
  - ↳ Smaller capacity with smaller block sizes.
- Secondary cache(s) should focus on **reducing miss penalty** by reducing miss rate to main memory.
  - ↳ Larger capacity with larger block sizes.
  - ↳ Higher levels of associativity.
- The miss penalty of the L1 cache is significantly reduced by the presence of an L2 cache.
  - ↳ L1 cache can be smaller and faster (but results in higher miss rate).
- Hit time is less important for L2 cache than miss rate.
  - ↳ But, L2 hit time determines L1's miss penalty.
  - ↳ Presence of L3 cache greatly improves the situation, allowing L2 miss rate to be *slightly* less important.

It's all a balancing act.

Improving Cache Performance (1/2)

AMAT = Time for a Hit + Miss Rate * Miss Penalty

(1) Reduce the **time to hit** in the cache.

- ↳ Smaller cache size, smaller block size.
- ↳ Direct-Mapped
- ↳ For writes, two possible strategies:
  - No-Write Allocate: no "hit" on cache, just write to write buffer. Makes subsequent reads tricky.
  - Write Allocate: avoid 2 cycles using write buffer to write to lower cache.

(2) Reduce the **miss rate**.

- ↳ Larger cache, Larger block size (16 - 64 bytes typical).
- ↳ More flexible placement (increase associativity).
- ↳ Use a **victim-cache** – a small buffer holding most recently discarded blocks.

Improving Cache Performance (2/2)

AMAT = Time for a Hit + Miss Rate * Miss Penalty

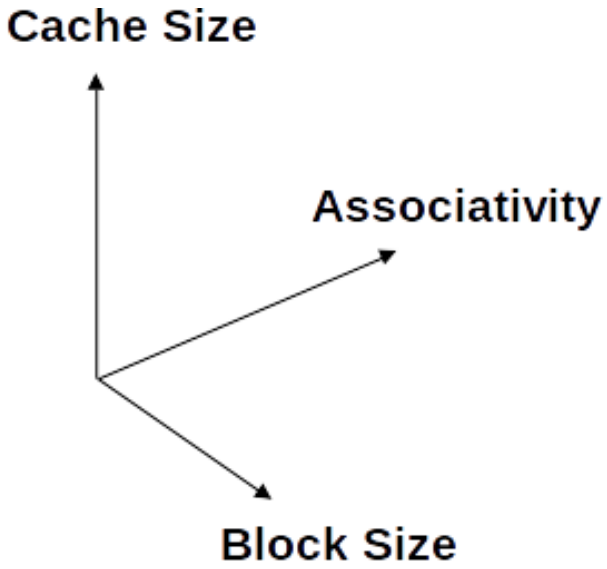
(3) Reduce the **miss penalty**

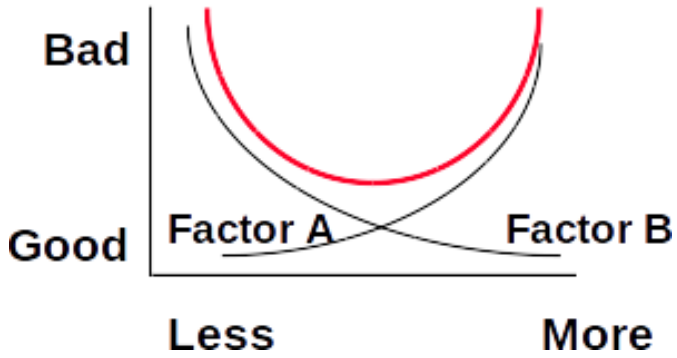
- ↳ Smaller blocks.
- ↳ Use a write-buffer.
- ↳ Check the write-buffer (or the victim-cache) on a read miss: *luck!*
- ↳ Pre-fetch critical word first, then rest of cache block.
- ↳ Use multiple cache levels.
- ↳ Faster backing store (= main memory).
- ↳ Improved **memory bandwidth** – amount and speed of memory transfer between levels (e.g. wider buses).

The Cache Design Space

It's all a balancing act.

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a **compromise**
  - depends on access characteristics (what the program is doing).
  - depends on technology / cost.
- *Simplicity* often wins.





Memory Hierarchy: Critical Aspects

The Principle of Locality:

- Program likely to access a relatively small portion of the address space at any instant of time.
  - ↳ **Temporal locality:** Locality in Time.
  - ↳ **Spatial locality:** Locality in Space.

Three major types of **cache misses**:

- **Compulsory/Cold Misses**: Sad facts of life. The *first* reference to an address/block.
- **Conflict misses**: Increase cache size and/or associativity. **Thrashing!**
- **Capacity misses**: It turns out size does matter :(

Cache Design Space

- Total size, Block size, Associativity (& Replacement policy).
- Write-hit policy: **write-through**, **write-back**
- Write-miss policy: **(no)-write-allocate**. Use write buffers.

Questions to Consider

Q1 Where can an entry be placed or found in cache?

- ↳ Cache Organization, Direct-Mapping, Associativity.

Q2 Which entry should be replaced on a miss?

- ↳ Replacement Policies: LRU, FIFO.

Q3 What happens on a write?

- ↳ Write hit/miss strategies: write-through, write-back, write-allocate.

Typical Example: Given list of memory references describe the cache's state after each reference.

Q1: Where can an entry be placed?

	# of sets	Entries per set
Direct mapped	# of cache lines	1
Set associative	(# of cache lines) / associativity	Associativity (typically 2 to 16)
Fully associative	1	# of entries

	Location method	# of comparisons
Direct mapped	Index	1
Set associative	Index the set; compare set's tags	Degree of associativity
Fully associative	Compare all entries' tags	# of entries

Q2: Which entry should be replaced on a miss?

- Easy for direct mapped - only one choice.
- Set associative or fully associative:
  - ↳ Random
  - ↳ LRU (Least Recently Used)
  - ↳ FIFO (First In First Out)
- For 2-way set associative LRU is easy and ideal.
- Comparisons required for LRU can be too costly for high levels of associativity ($>$ 4-way).