Computer Systems

- Everything is represented by a sequence of “0” or “1” (binary) in digital systems. What distinguishes different information is the context in which we view them. The same binary numbers can represent different things such as integer, character string, machine instruction, etc.
- Programs are translated by other programs into different forms
- Processors read and interpret instructions stored in memory
- Storage devices form a hierarchy
- Operating system manages hardware
- Systems communicate with other systems using networks

Assembly Language

- Low-level programming language which is a human readable, textual representation of machine code
- Each statement corresponds to a single machine instruction
- Each assembly language is specific to a particular processor architecture (Instruction Set Architecture, ISA)
- Unlike high level languages, it contains lots of hardware information

```
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
leaq    L_.str(%rip), %rdi
movb    $0, %al
callq   _printf
xorl    %eax, %eax
addq    $16, %rsp
popq    %rbp
retq
```

Observations

- System spends lots of time moving programs and data from one place to another
- System has multiple different memory spaces
  - Disk, Main memory, Cache, Registers
  - Memory spaces hold both program and data
  - All memory spaces except disk are temporary storages
- Computer system consists of hardware and system software that cooperate to run application programs
- Information inside the computer is represented as groups of bits that are interpreted in different ways, depending on the context

Amdahl's Law

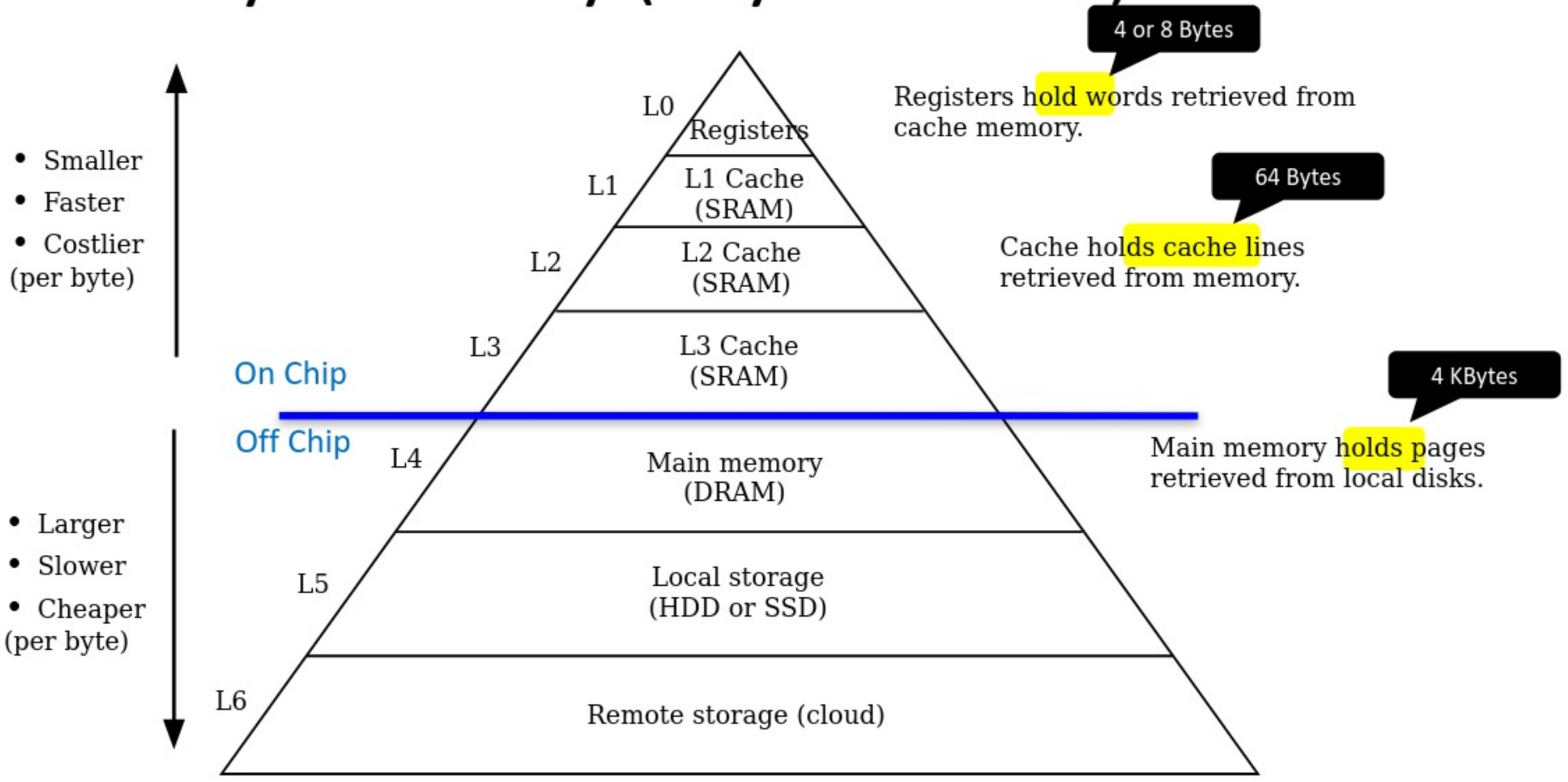
When we speed up one part of a system, the effect on the overall system performance depends on both how significant this part was and how much it sped up.

$$S(p) = \frac{1}{(1-f) + \frac{f}{p}}$$

f: Fraction of execution time benefiting from improvement
p: Speedup of the part benefiting from improvement

Example: A part of system that initially consumed 60% of the time (f = 0.6) is sped up by a factor of 3 (p = 3). The speedup is 1/((1-0.6)+0.6/3) which is 1.67x (67%)

Memory Hierarchy (Physical View)



Memory (Logical View)

- Program views memory as a very large array of bytes
  - Referred to as Virtual Memory
  - 4294967296 bytes (4GB) in 32-bit machine (2^{32} = 4294967296)
  - Every byte of memory is identified by a unique number called memory address
  - Physically implemented with a hierarchy of different memory spaces (pyramid shape)

Cache Matters

- Cache memory performance depends on memory access patterns

```
// Version 1
int i, j;
for (i = 0; i < 512; i++)
  for (j = 0; j < 512; j++)
    dst[j][i] = src[j][i];
```

```
// Version 2
int i,j;
for (j = 0; j < 512; j++)
  for (i = 0; i < 512; i++)
    dst[j][i] = src[j][i];
```

Q: Which one do you think is faster? By how much?

Standards

- Once context is known, we need to know what standard to use to interpret binary numbers
- ASCII – a widely used standard for characters
  - 01000001 represents “A” and 01100001 represents “a”
- RGB – a widely used standard for colors
  - 1111….1111 represents “white” and 0000….0000 represents “black”
- Two’s complement – a widely used standard for signed integers
  - 0……00000001 represents “1” and 1……11111111 represents “-1”
- IEEE 754 – a widely used standard for floating point numbers
  - 00111111010000……0 represents a float “0.25”

Data Types and Sizes

You can check the size of data types using `sizeof()` in C/C++

e.g. `printf("%ld", sizeof(char));`

C Data Type	Intel IA32	x86-64
`char`	1 byte	1 byte
`short`	2 bytes	2 bytes
`int`	4 bytes	4 bytes
`long`	4 bytes	8 bytes
`long long`	8 bytes	8 bytes
`float`	4 bytes	4 bytes
`double`	8 bytes	8 bytes
`pointer`	4 bytes	8 bytes

← Machine dependent

Bit-Level Operations

- & (AND), | (OR), ~ (NOT), ^ (Exclusive OR)
  - Apply to any "integral" data type such as long, int, short, char, etc.
  - View arguments as bit vectors
  - Operations applied bit-wise
- Examples (char data type)
  - ~0x41 & 0xBE
    - $\sim 01000001_2 \;\&\; 10111110_2 = 10111110_2 \;\&\; 10111110_2 = 10111110_2$
  - ~0xF0 | 0xFF
    - $\sim 11110000_2 \mid 11111111_2 = 00001111_2 \mid 11111111_2 = 11111111_2$
- Self XOR (e.g., x ^ x)
  - Resetting (always zero)

Logical Operations

- && (AND), || (OR), ! (NOT)
  - View 0 as “False” and anything nonzero as “True”
  - Always return 0 or 1
  - Early termination (short-circuit evaluation)
- Examples (`char` data type)
  - !0x41 && 0x00 = 0x00 && 0x00 = 0x00 (False)
  - !0x00 || 0x01 = 0x01 || 0x01 = 0x01 (True)
  - !!0x41 && 0x01 = 0x01 && 0x01 = 0x01 (True)

Shift Operations

- Left Shift: x << y
  - Shift bit-vector x left y positions
    - Throw away y MSB
    - Fill the right end with y zeros
  - Same impact as multiply by 2^Y
- Right Shift: x >> y
  - Shift bit-vector x right y positions
    - Throw away y LSB
  - Logical shift
    - Fill the left end with y zeros
  - Arithmetic shift
    - Replicate MSB on the left end
  - Same impact as divide by 2^Y

x = 01100010	x << 3	00010000
	x >> 2 (logical)	00011000
	x >> 2 (arithmetic)	00011000
y = 10100010	y << 3	00010000
	y >> 2 (logical)	00101000
	y >> 2 (arithmetic)	11101000

ISA (Instruction Set Architecture)

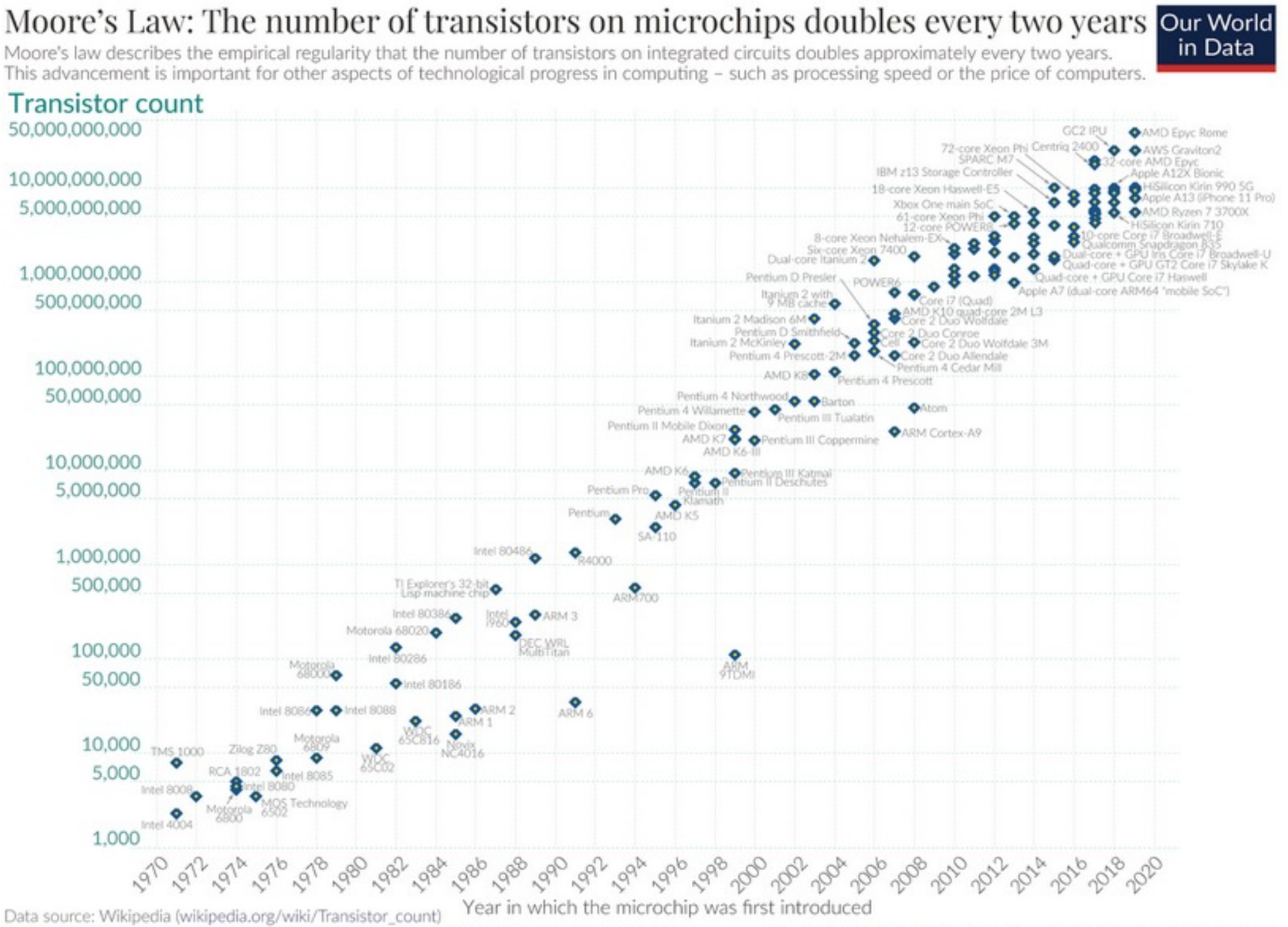
- Interface between software and hardware
  - SW/compiler assumes and HW promises
- Abstraction of the format and behavior of processor
- Examples: x86, ARM, MIPS, PowerPC, Alpha, and many more
- Portion of computer visible to ASM programmer or compiler writer
- Standard for hardware design

Assembly Language

- Why different ISAs? Different target workloads
  - Desktop: Performance of various programs
  - Database, File, and Web servers: Floating-point is less important
  - Mobile and embedded: energy consumption, code size, memory footprint

Moore’s Law

- Transistor counts on a single chip doubles every 18~24 months
- It implies that the computing power of a single chip also doubles every 18~24 months



Three Types of Instructions

1. Perform ALU operation on register or memory data
   - e.g., addl (%edx), %eax
2. Move (transfer) data between memory and register
   - Load data from memory into register
   - Store register data into memory
   - e.g., movl 8(%edx), %eax
3. Control execution flow
   - Unconditional or conditional branches
   - e.g., jmp .L1

Random-Access Memory (RAM)

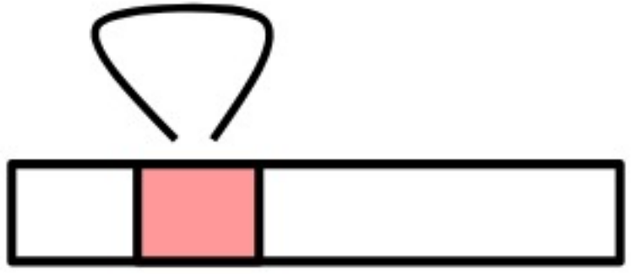
- Broadly two types: SRAM vs DRAM, both are volatile

	Static RAM (SRAM)	Dynamic RAM (DRAM)
Transistors per bit	4~6	1
Cost	High	Low
Access time	Fast	Slow
Need refresh?	No	Yes (every 10-100 ms)
Sensitivity to electrical noise (EMI), radiation	Low	High
Need Error Detection Code (EDC)	Maybe	Yes
Applications	Cache memories	Main memories, Frame buffer

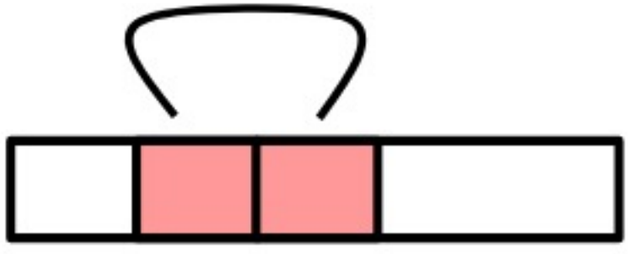
Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently

- Temporal locality:
  - Recently referenced items are likely to be referenced again in near future



- Spatial locality:
  - Items with nearby addresses tend to be referenced close together in time



Types of Cache Misses

- Cold (compulsory) miss
  - Cold misses occur because the cache is empty at the beginning of program execution
- Conflict miss
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k.
    - E.g. Block i at level k+1 must be placed in block (i mod 4) at level k.
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block.
    - E.g. Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time.
- Capacity miss
  - Occurs when the set of active cache blocks (working set) is larger than the cache

Writing (Updating) Memory

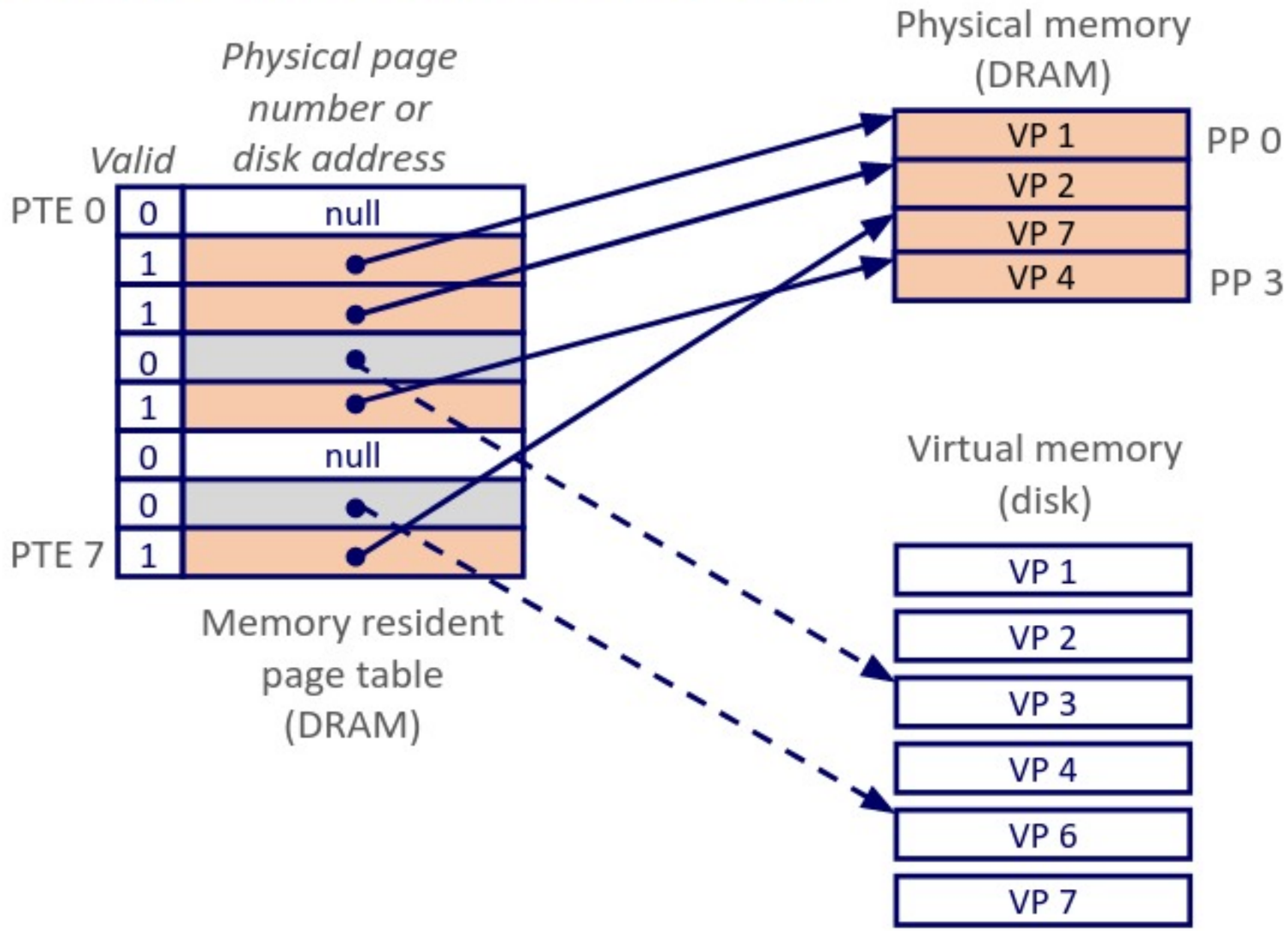
- Multiple copies of data exist in memory hierarchy
  - $L1, $L2, $L3, Main Memory, Disk
- What to do on a write-hit (cache has it)?
  - Write-through (write immediately to lower level)
  - Write-back (defer write to lower level until replacement of line)
    - Need a dirty bit (indicating the line is updated)
- What to do on a write-miss (cache does not have it)?
  - Write-allocate (load cache line into cache, then update line)
    - Great if more writes to the cache line follow (locality)
  - No-write-allocate (write immediately to lower level)
- Typical selection
  - Write-through + No-write-allocate
  - **Write-back + Write-allocate** (choice of almost all modern processors)

Why Virtual Memory (VM)?

1. Makes programming so much easier
2. Uses DRAM as a cache for the parts of a large virtual address space
3. Simplifies memory management
   - Each process gets the same uniform linear address space
4. Isolates address spaces (easier memory protection)
   - One process can't interfere with another's memory
   - User program cannot access privileged kernel information

Page Tables

- Page table is an array of page table entries (PTEs) that maps virtual pages to physical pages.
  - Per-process kernel data structure in DRAM



Instruction Pipelining

- Divide the execution of an instruction into a series of sequential steps (stages) performed by different hardware units
- Execute multiple instructions with different parts of instructions processed in parallel
- A hardware technique used in the design of modern processors to increase their instruction throughput
  - Throughput - the number of instructions executed in a unit of time
- Improve instruction level parallelism

Parallelism

Parallelism is a driving force of modern computer design

1. ILP (Instruction Level Parallelism)
   - Instruction pipelining, multiple issue, and more techniques
2. TLP (Task or Thread Level Parallelism)
   - Multi-core processors
3. DLP (Data Level Parallelism)
   - Vector processors, GPUs, etc.

Loop Unrolling

- Replicate the loop body multiple times

```
for(int i = 0; i<9; i++) {
    C[i] = A[i] + B[i];
}
```

- Partial loop unrolling (unroll factor 3)

```
for(int i = 0; i<9; i+=3) {
    C[i] = A[i] + B[i];
    C[i+1] = A[i+1] + B[i+1];
    C[i+2] = A[i+2] + B[i+2];
}
```

- Loop Unrolling
  - Advantages
    - # branch related insts ↓
    - Less control hazard
    - Better instruction scheduling
  - Disadvantage
    - Code size ↑
    - Register needed (register pressure) ↑

```
LD
LD
ADD
SD
ADD
CMP
BL
```

```
LD
LD
ADD
SD
~~ADD~~
~~CMP~~
~~BL~~
LD
LD
ADD
SD
~~ADD~~
~~CMP~~
~~BL~~
LD
LD
ADD
SD
```

```
LD
LD
LD
LD
LD
LD
ADD
ADD
ADD
SD
~~ADD~~
~~CMP~~
~~BL~~
ADD
CMP
BL
```