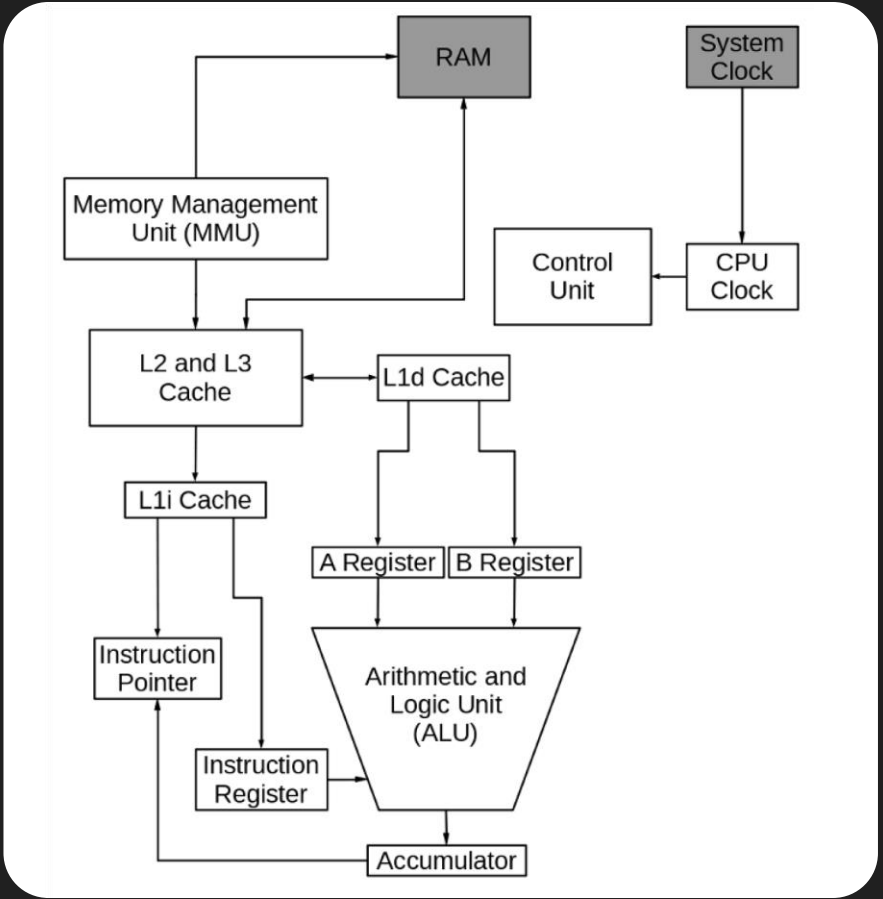
SPECTRAL:
TECHNOLOGIES

WEEK 6

CPU and Memory

CPU Structure

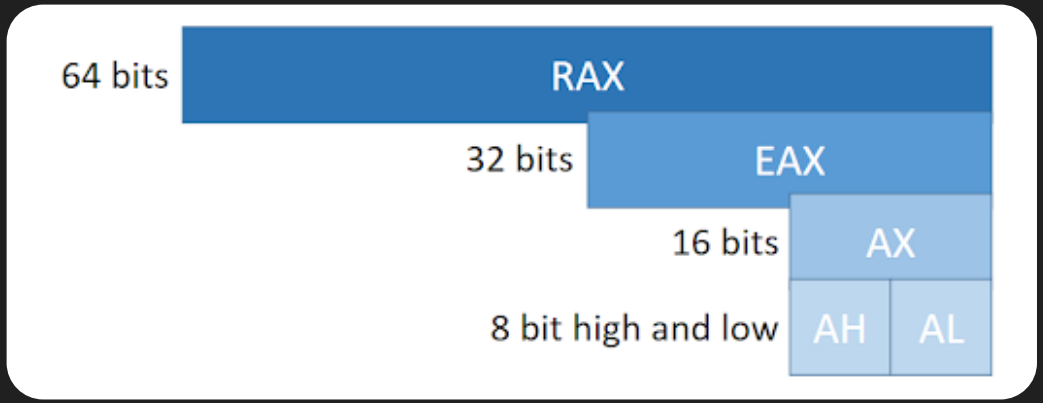


Instructions

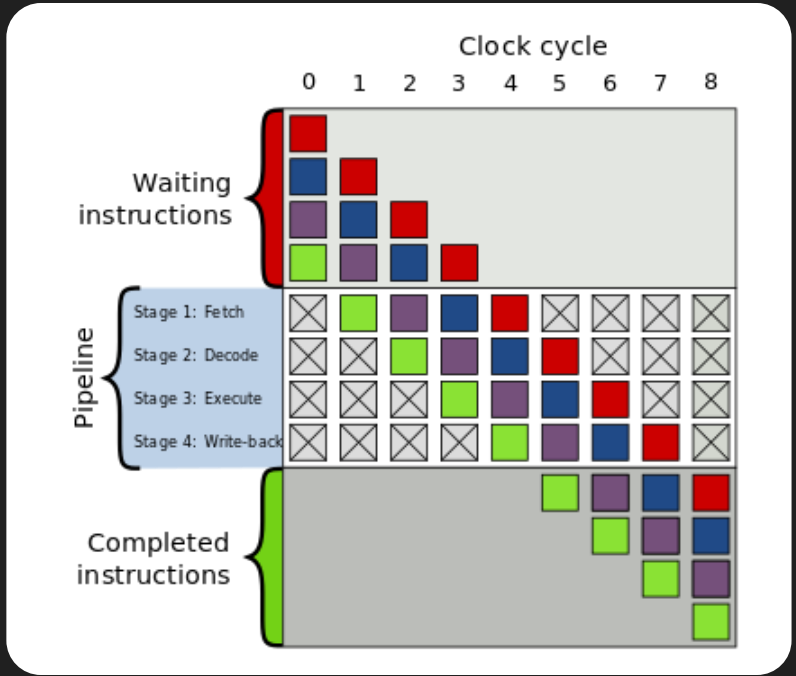
- We've seen some examples of assembly already
- Assembly is a textual representation of binary instructions
- RISC vs CISC, basically a tradeoff between the complexity and richness of instructions and the complexity of decoding and executing these instructions in CPU
- We focus on x86_64, which is a hybrid between CISC and RISC
- Instructions there may have different byte length, which makes decoding harder
- Instructions are further splitted to microcode during the decoding
- ALU in CPU doesn't interact with memory, only with data in registers, where it should be loaded first

Registers

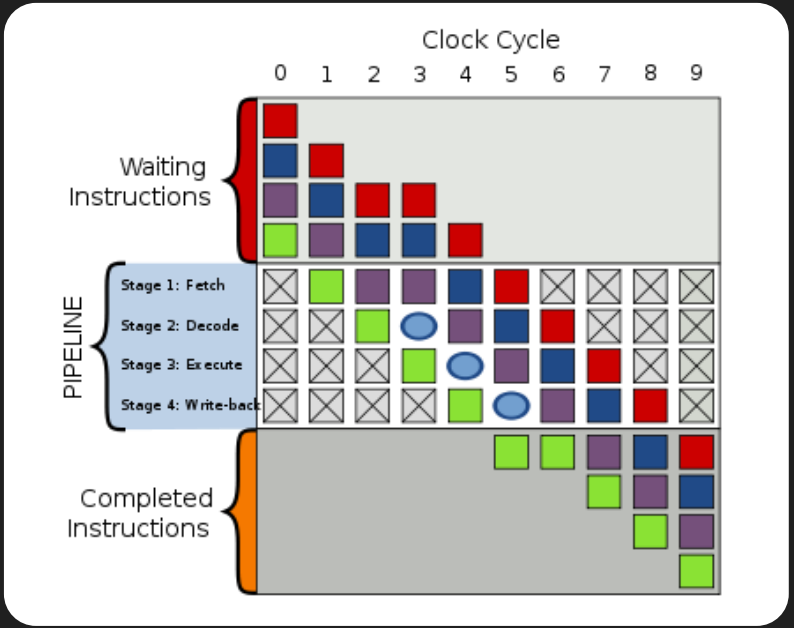
- Ultra-fast memory on CPU
- General-Purpose Registers (**rax**, **rcx**, **r2-r15**, etc.)
  - e.g. rdi, rsi, rdx, rcx, r8, r9 are used for passing parameters to functions
- Stack (**rsp**, **rbp**)
- Instruction pointer **rip**
- Flags
- Floating-point **xmm0-xmm15**



Pipeline


Clock cycle
0 1 2 3 4 5 6 7 8
Waiting instructions
Pipeline
Stage 1: Fetch
Stage 2: Decode
Stage 3: Execute
Stage 4: Write-back
Completed instructions

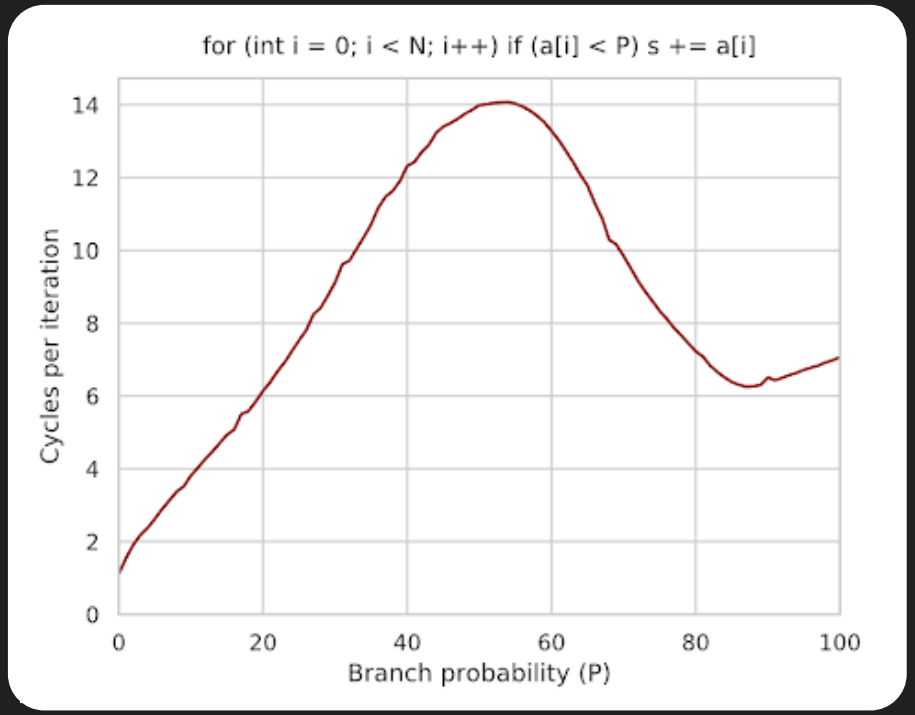
Pipeline Stall



- This shows why branching may greatly affect the performance and especially latency
- A lot of techniques were introduced to counter this issue
- *Branch predictor* is a separate circuit, that helps to optimize the effeciency of speculative execution

Branch prediction

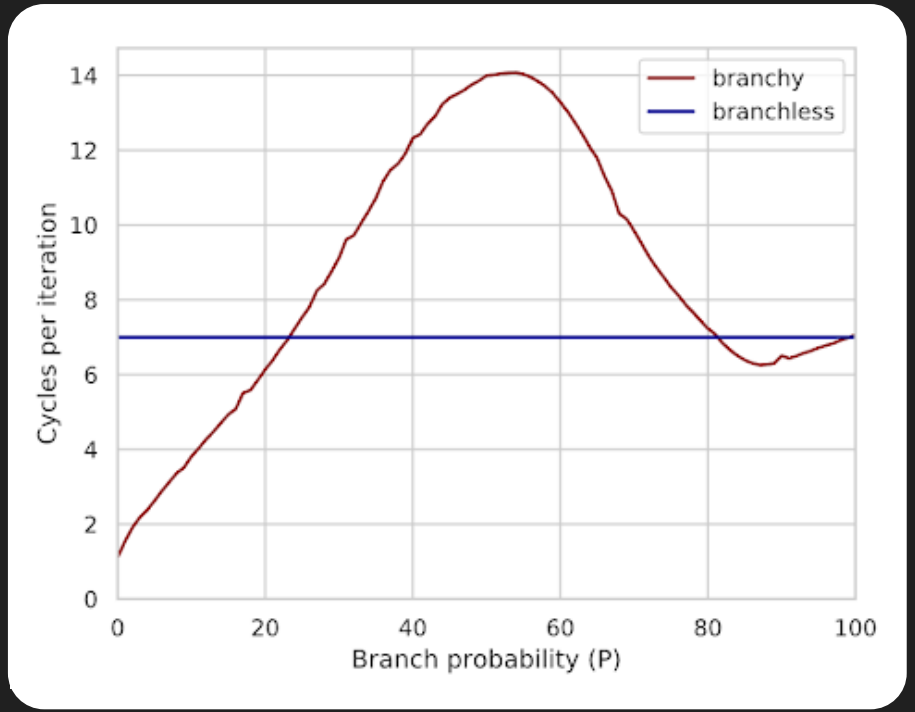
- Numbers a[i] are uniformly distributed on [0, 100]
- Note, that the worst result at P = 50, since its the worst case for branch predictor



Branchless code

– Lets change the code to

```
for (int i = 0; i < N; i++) s += (a[i] < P ? a[i] : 0)
```



Context switch

- For now we implicitly assumed there is only one process running. But in the reality there are a lot of independent processes running
- Each process (or thread) has its own stack and register values, CPU should use the right values for each process
- This process of storing and restoring these values is called **context switch**
- Mainly done by OS with some help from CPU
- Its a heavy operation

SPECTRAL:
TECHNOLOGIES

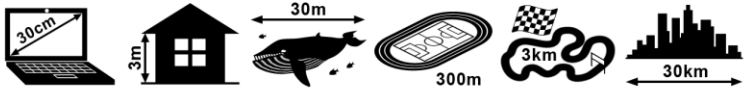
Not all CPU operations are created equal

ithare.com

Operation	Cost in CPU Cycles
“Simple” register-register op (ADD,OR,etc.)	<1
Memory write	~1
Bypass delay: switch between integer and floating-point units	0-3
“Right” branch of “if”	1-2
Floating-point/vector addition	1-3
Multiplication (integer/float/vector)	1-7
Return error and check	1-7
L1 read	3-4
TLB miss	7-21
L2 read	10-12
“Wrong” branch of “if” (branch misprediction)	10-20
Floating-point division	10-40
128-bit vector division	10-70
Atomics/CAS	15-30
C function direct call	15-30
Integer division	15-40
C function indirect call	20-50
C++ virtual function call	30-60
L3 read	30-70
Main RAM read	100-150
NUMA: different-socket atomics/CAS (guesstimate)	100-300
NUMA: different-socket L3 read	100-300
Allocation+deallocation pair (small objects)	200-500
NUMA: different-socket main RAM read	300-500
Kernel call	1000-1500
Thread context switch (direct costs)	2000
C++ Exception thrown+caught	5000-10000
Thread context switch (total costs, including cache invalidation)	10000 - 1 million

10^0 10^1 10^2 10^3 10^4 10^5 10^6

Distance which light travels while the operation is performed

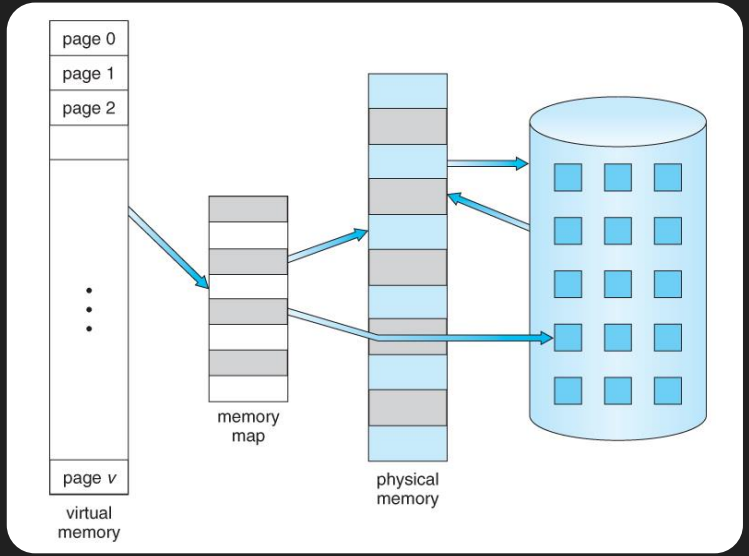


Instructions: things to consider

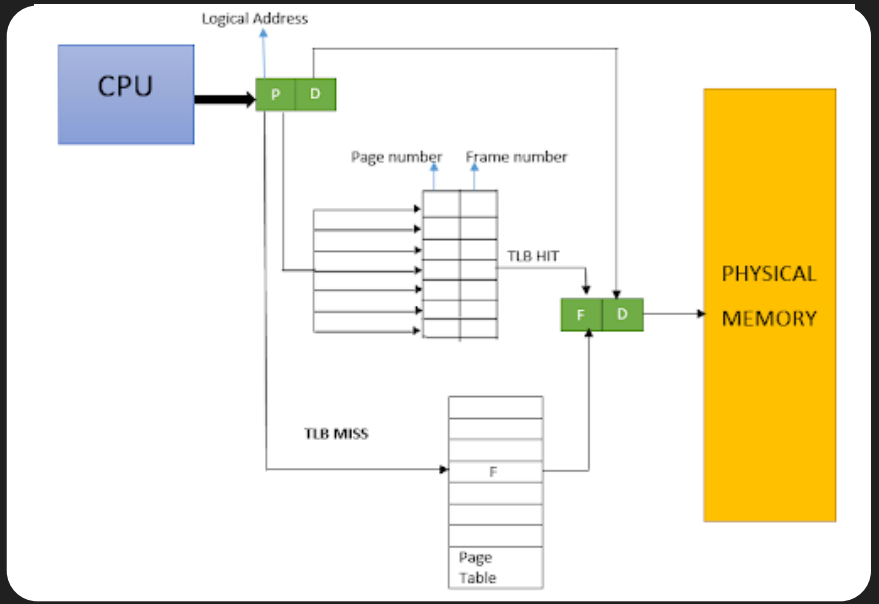
- Different instructions have different cost, https://llvm.org/docs/CommandGuide/llvm-mca.html can be used to analyze the theoretical performance of machine code in a specific CPU
- However, due to how complex the entire pipeline is, its not as simple and may be unintuitive
- Floating-point addition costs more than integer, but the resulting code might execute faster, because FPA is performed by a separate unit
- Branching is one of the most important components to consider
- We'll return to this topic in optimizations lecture

Virtual memory

- Recall our first lecture: every address exist in a virtual space. So every address should be mapped to the physical one in order to be processed further
- Recall also that memory is divided into pages, and its actually pages, that are mapped. The offset inside a page remains the same
- This is done by a separate hardware unit called MMU (memory management unit)

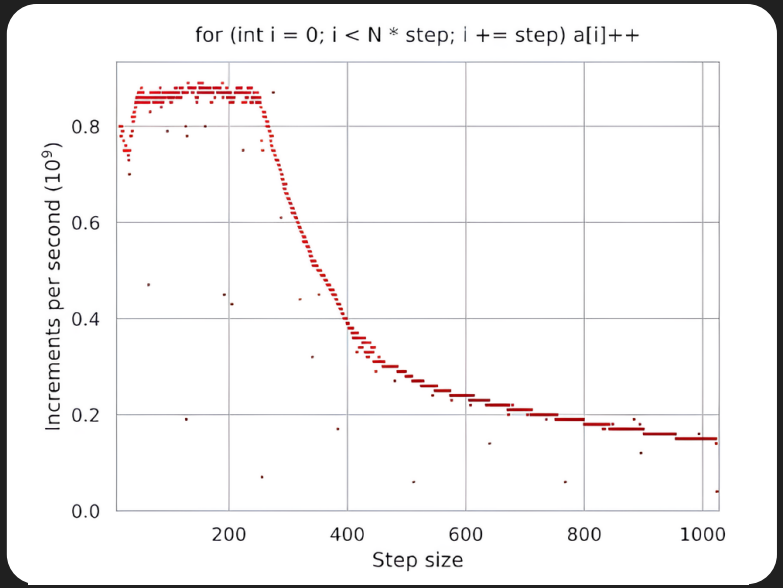


- The address translation is handled by hardware (best case) and OS (worst case)
- TLB is basically a cache of map Page Number -> Physical address
- In the best case the entry is in TLB and job is done. If its not, then *page walk* search through page table (stored in RAM) is performed by MMU
- If the entry still isn't found, MMU signals to OS about it
- This is called **page fault** (it can be minor or major, we'll see the difference on the next lecture)



TLB Miss

- TLB size is a few thousand entries usually (there are a few TLBs just like with other caches)
- TLB Hit takes about 1 cycle
- Page Walk may take hundreds of cycles
- $N = 2^{13}$

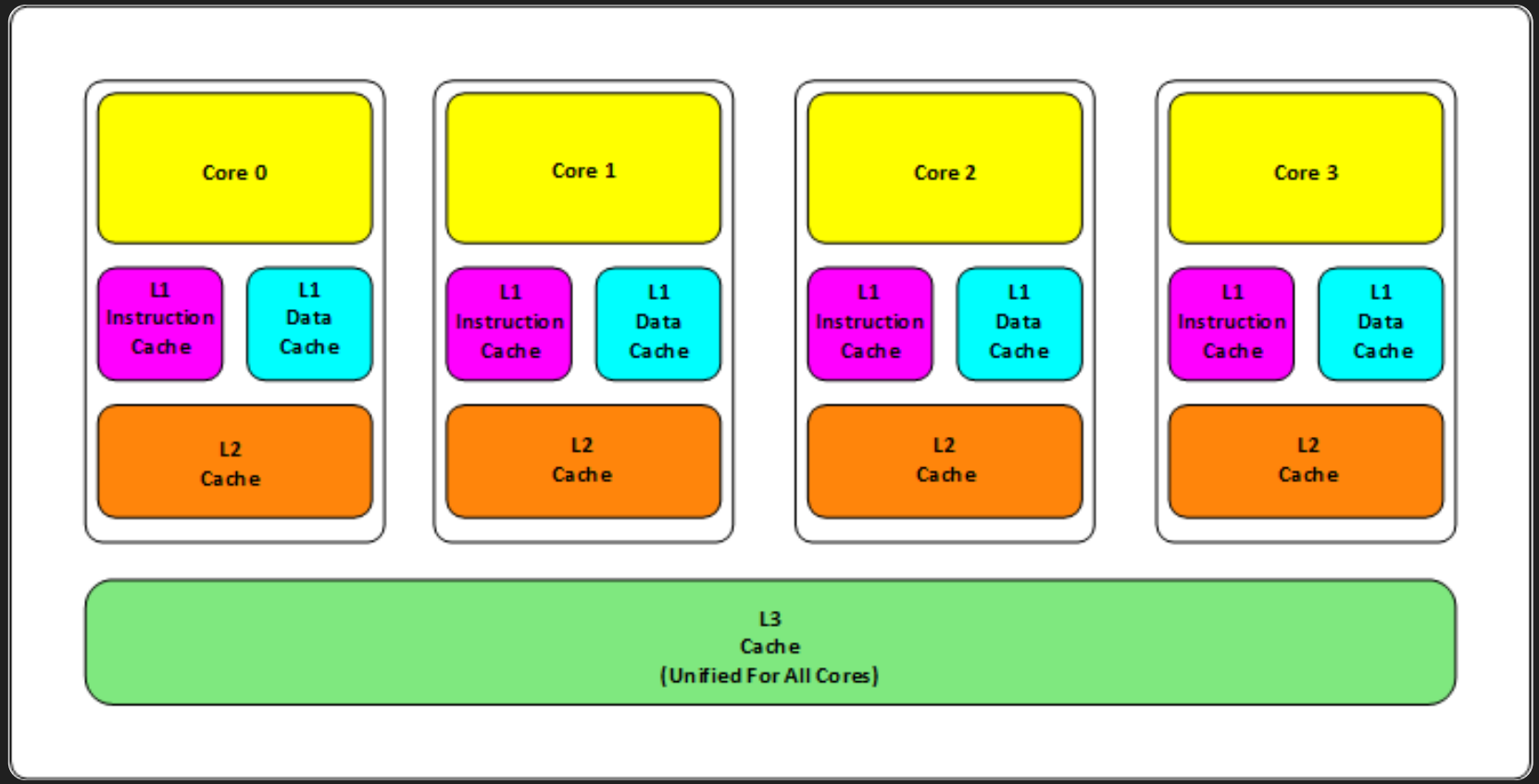


SPECTRAL:
TECHNOLOGIES

Latency numbers every programmer should know

```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns              14x L1 cache
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns              20x L2 cache, 200x
Compress 1K bytes with Zippy             3,000   ns
Send 1K bytes over 1 Gbps network       10,000   ns    0.01 ms
Read 4K randomly from SSD*             150,000   ns    0.15 ms
Read 1 MB sequentially from memory     250,000   ns    0.25 ms
Round trip within same datacenter      500,000   ns    0.5  ms
Read 1 MB sequentially from SSD*     1,000,000   ns    1    ms  4X memory
Disk seek                           10,000,000   ns   10    ms  20x datacenter rour
Read 1 MB sequentially from disk    20,000,000   ns   20    ms  80x memory, 20X SSI
Send packet CA->Netherlands->CA    150,000,000   ns  150    ms
```

Caches


Core 0
L1 Instruction Cache
L1 Data Cache
L2 Cache
Core 1
L1 Instruction Cache
L1 Data Cache
L2 Cache
Core 2
L1 Instruction Cache
L1 Data Cache
L2 Cache
Core 3
L1 Instruction Cache
L1 Data Cache
L2 Cache
L3 Cache (Unified For All Cores)

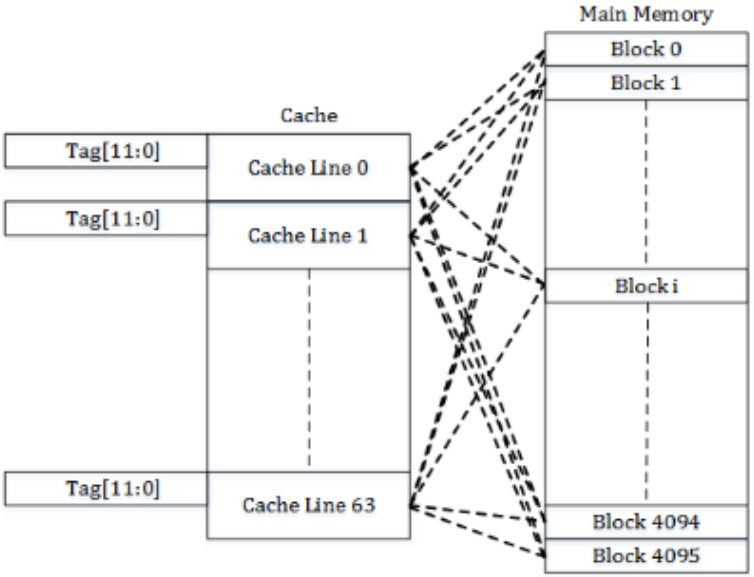
Cache

- L1 ~ 512Kb, L2 ~ 6Mb, L3 ~ 32Mb on modern CPUs
- The entry is a *cache line* (usually 64 bytes)
- Maybe inclusive or exclusive (or none)
- May have different replacement policy (LRU e.g.)
- There is an important trait called **associativity**

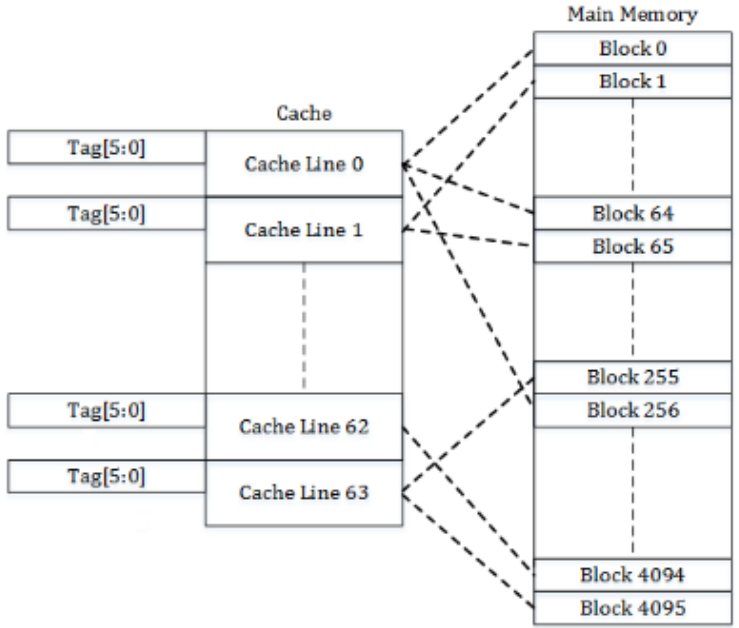
SPECTRAL:
TECHNOLOGIES

Cache Associativity

Fully Associative

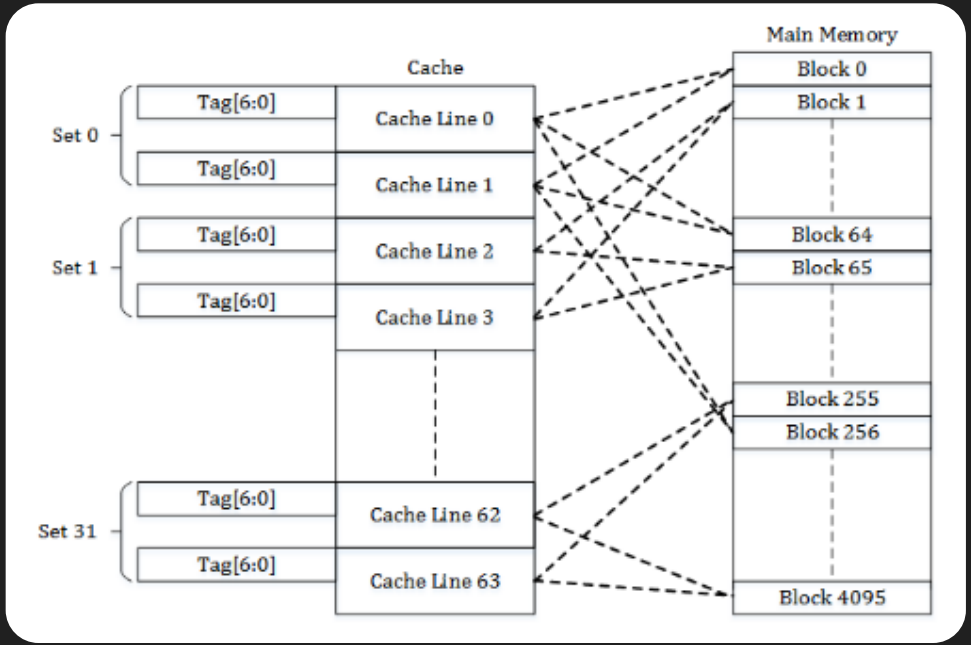


Direct



Cache Associativity

- Instead in-between version is used: *set-associative cache*



- Associativity specifies the number of cache entries a block can be mapped to
- Bigger associativity leads to more cache hits, but makes cache slower

Cache Prefetch

- In a lot of cases memory access is local and/or sequential, e.g. a lot of code boils down to iterating over some arrays
- Whenever a cache miss happens at memory location X, its rational to assume, that the following access will be at X + small offset
- This is especially accurate for instructions
- So usually some prefetch logic is implemented in CPU, which preloads data to cache in advance
- Its not possible to implement a complex pattern access prediction right on hardware, so only simple patterns may be recognized
- Sequential access is still the best choice (especially due to the fact, that data loaded by cache lines)

Cache coherence

- Each core has it's own L1 and L2 cache, however these caches should be coherent with each other
- If two cores store the same cache line in their caches, and then one of the cores updates it, the other core should read the updated version as well
- Various cache coherency protocols are used for that
- L3 cache is shared, so there is no such issue, however another issue of contention may arise

False sharing

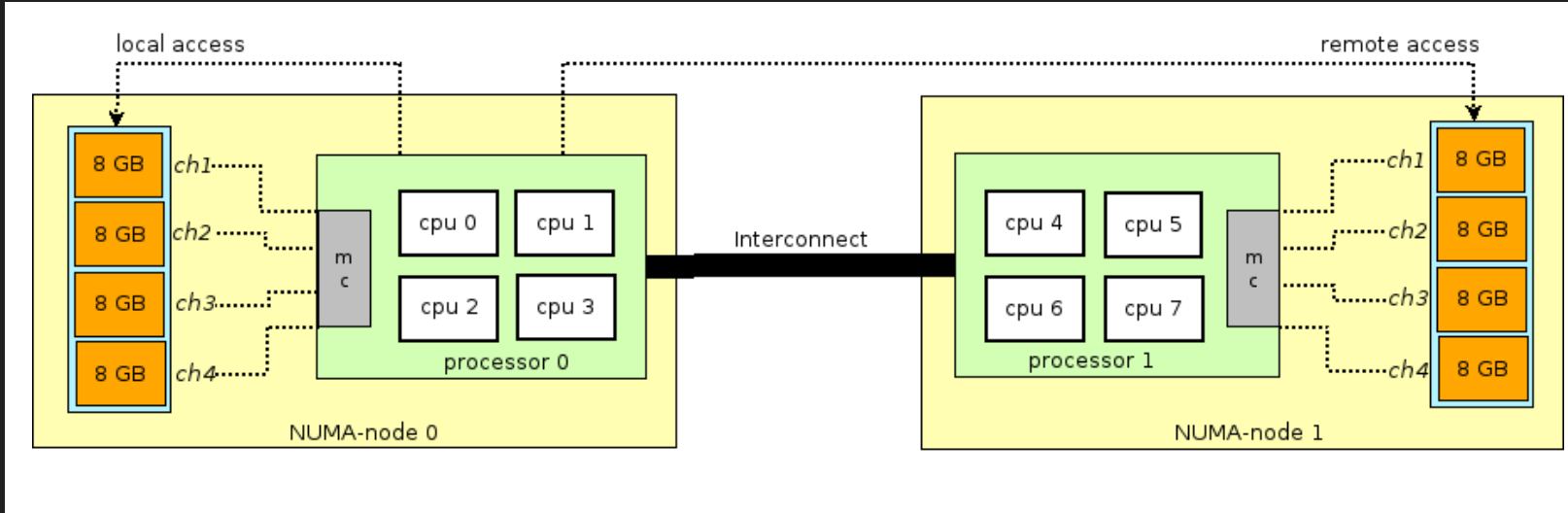
```
int sum1;
int sum2;

void thread1(int v[ ], int v_count) {
    sum1 = 0;
    for (int i = 0; i < v_count; i++)
        sum1 += v[i];
}

void thread2(int v[ ], int v_count) {
    sum2 = 0;
    for (int i = 0; i < v_count; i++)
        sum2 += v[i];
}
```

NUMA (Non-uniform memory access)

- On systems with multiple CPUs the latency between a core and RAM can vary a lot depending whether its a local or remote access
- This setup is pretty common in cloud instances like AWS EC3 and etc



Memory: things to consider

- Always prefer sequential access due to cache lines and prefetching
- Avoid using powers of 2 as an array size or offset values in some cases
  Don’t forget about cache associativity
- Once you made an application multithreaded, there is a possibility for false sharing
- If you see a linux system with a lot of cores, that usually means its a NUMA system
- Usually you want to run connected applications on the same numa node
- We'll return to these questions later in optimizations lecture

Literature

- https://en.algorithmica.org/hpc/