/INFOMOV/

Optimization & Vectorization

J. Bikker - Sep-Nov 2015 - Lecture 3: “Caching (1)”

Welcome!

Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Practical Assignment 1

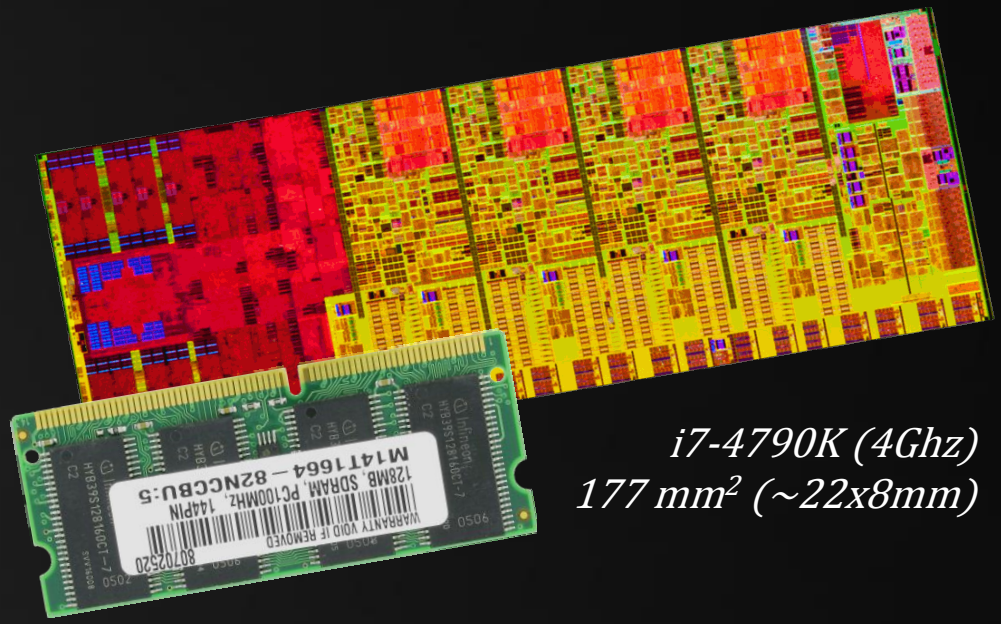
Introduction

Feeding the Beast

Let’s assume our CPU runs at 4Ghz.
What is the maximum physical distance between memory and CPU if we want to retrieve data every cycle?

Speed of light (vacuum): 299,792,458 m/s
Per cycle: ~0.075 m
➔ ~**3.75cm** back and forth.

In other words: we cannot physically query RAM fast enough to keep a CPU running at full speed.



i7-4790K (4Ghz)
177 mm² (~22x8mm)

Introduction

Feeding the Beast

Sadly, we can’t just divide the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR3-2133/PC3-17000):

- RAM runs at a much lower clock speed than the CPU
  - 17000 here means: theoretical bandwidth in MB/s
  - Bandwidth is 64-byte transfers per second, times 8
  - So, we get 2133 million transfers per second
  - DDR is ‘double data rate’, so actual I/O clock speed is 1067Mhz
- Latency between query and response: 11-14 cycles.

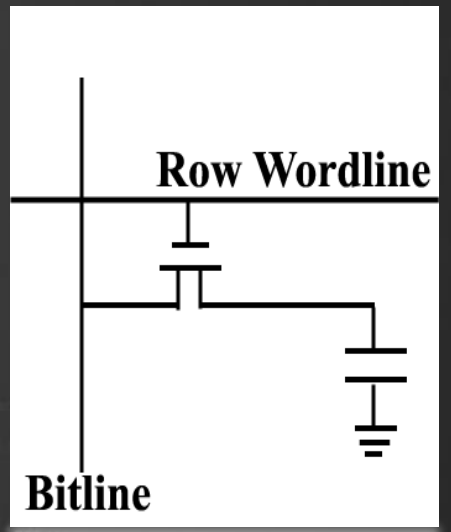
Introduction

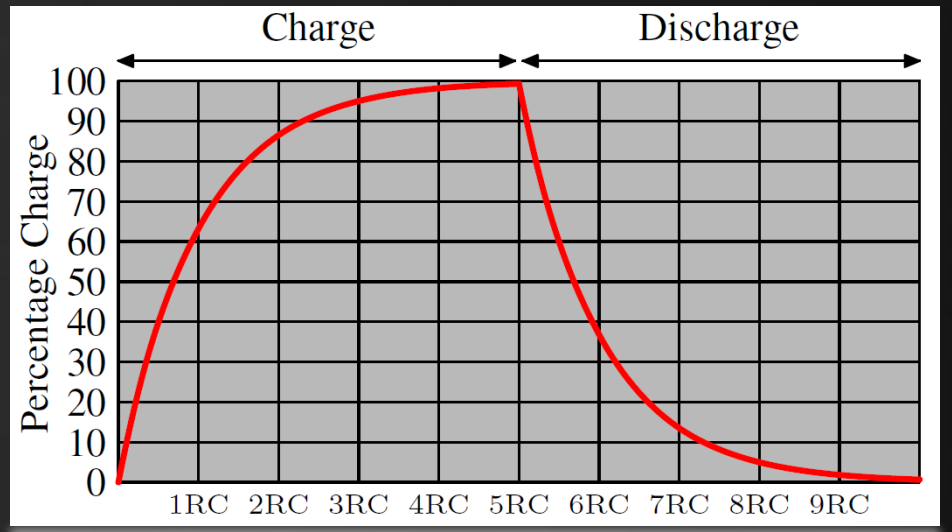
Feeding the Beast

Sadly, we can’t just divide the physical distance between CPU and RAM to get the cycles required to query memory.

Factors include (stats for DDR3-2133/PC3-17000):

- Latency between query and response: 11-14 cycles.





Introduction

Feeding the Beast

Sadly, we can’t just divide the physical distance between CPU and RAM to get the cycles required to query memory.

Additional delays may occur when:

- Other devices than the CPU access RAM;
- DRAM must be refreshed every 64ms due to leakage.

For a processor running at 2.66GHz, latency is roughly 110-140 CPU cycles.

Details in: “What Every Programmer Should Know About Memory”, chapter 2.

Introduction

Feeding the Beast

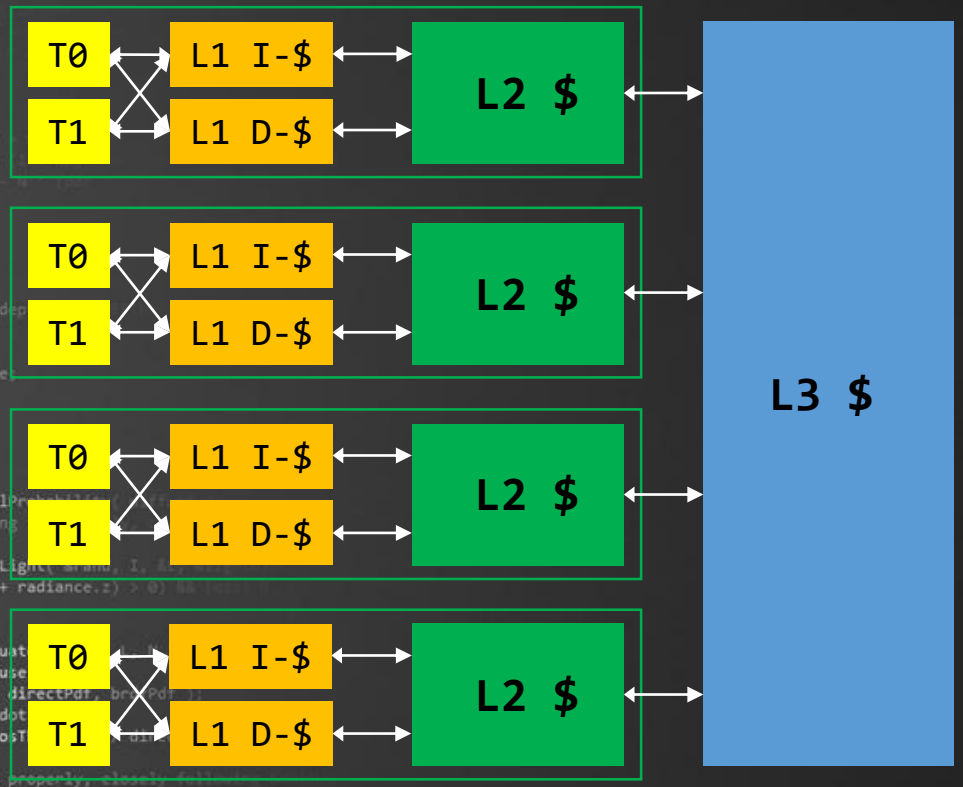
“We cannot physically query RAM fast enough to keep a CPU running at full speed.”

How do we overcome this?

We keep a copy of frequently used data in fast memory, close to the CPU: the ***cache***.

Introduction

The Memory Hierarchy – Core i7-9xx (4 cores)

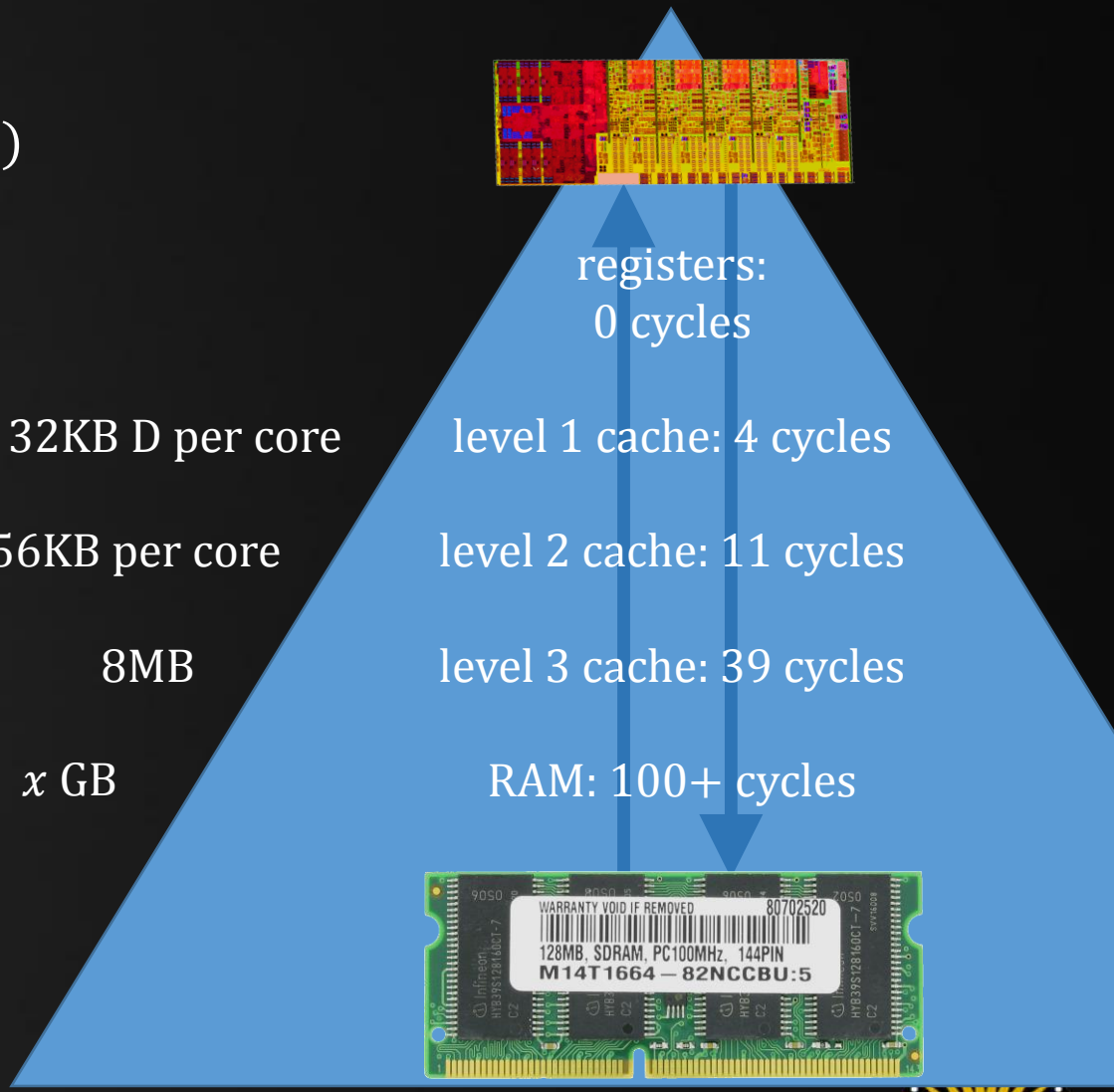


32KB I / 32KB D per core

256KB per core

8MB

x GB



Introduction

Caches and Optimization

Considering the cost of RAM vs L1 cache access, it is clear that the cache is an important factor in code optimization:

- Fast code communicates mostly with the caches
- We still need to get data into the caches
- But ideally, only once.

Therefore:

- The working set must be small;
- Or we must maximize *data locality*.

Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Practical Assignment 1

Architectures

Cache Architecture

The simplest caching scheme is the *fully associative cache*.

```
struct CacheLine
{
    uint address;
    unsigned char data;
    bool valid;
};
CacheLine cache[256];
```

This cache holds 256 bytes.

address	data	valid
0x00000000	0xFF	0
0x00000000	0xFF	0
0x00000000	0xFF	0
0x00000000	0xFF	0
0x00000000	0xFF	0
…	…	…
0x00000000	0xFF	0

Notes on this layout:

- We will rarely read 1 byte at a time
- So, we switch to 32bit values
- We will rarely read those at odd addresses
- So, we drop 2 bits from the address field.

Architectures

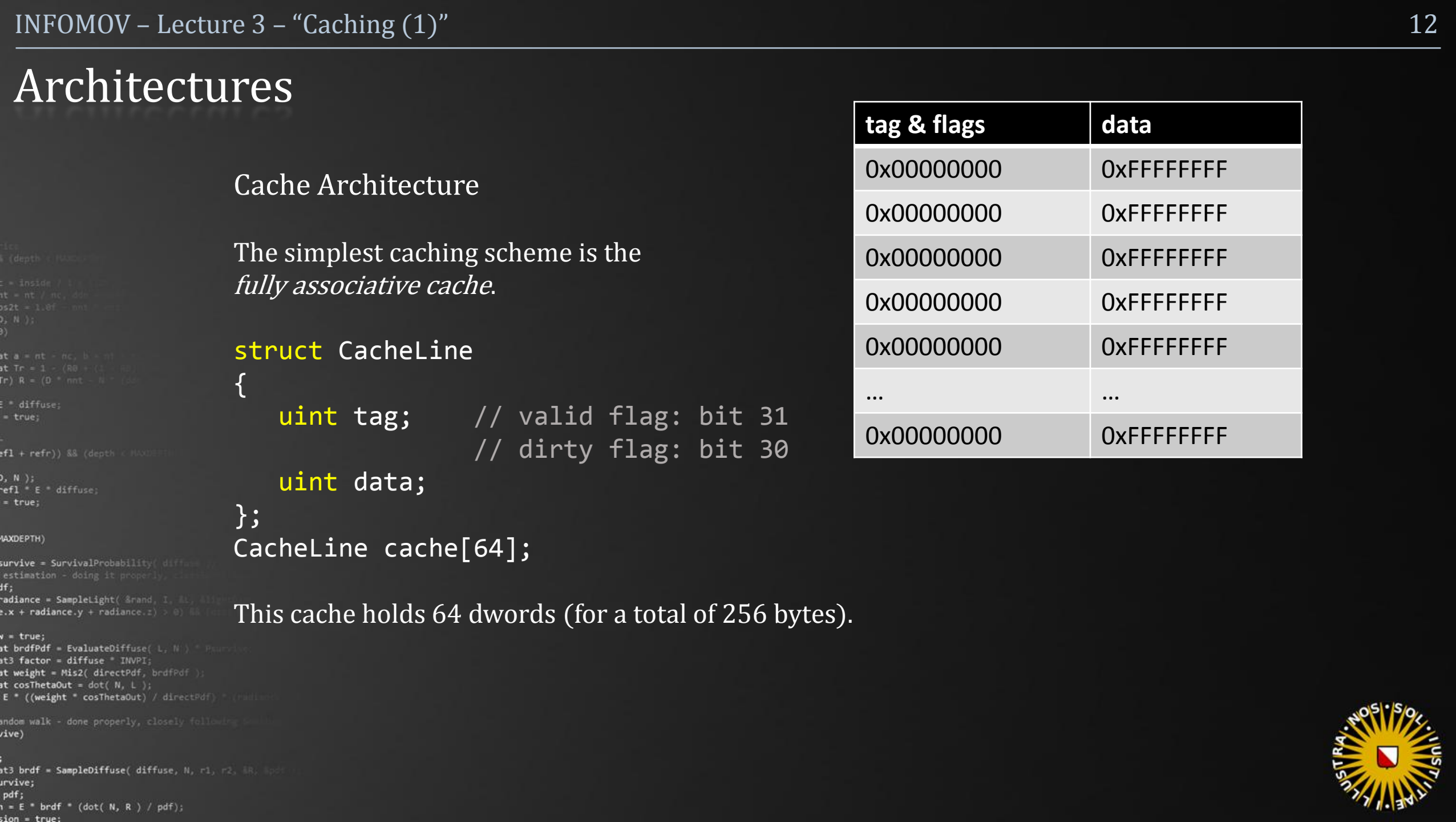
Cache Architecture

The simplest caching scheme is the *fully associative cache*.

```
struct CacheLine
{
   uint tag;     // valid flag: bit 31
                 // dirty flag: bit 30
   uint data;
};
CacheLine cache[64];
```

This cache holds 64 dwords (for a total of 256 bytes).

tag & flags	data
0x00000000	0xFFFFFFFF
0x00000000	0xFFFFFFFF
0x00000000	0xFFFFFFFF
0x00000000	0xFFFFFFFF
0x00000000	0xFFFFFFFF
…	…
0x00000000	0xFFFFFFFF



Architectures

Cache Architecture

The simplest caching scheme is the *fully associative cache*.

```
struct CacheLine
{
   uint tag;     // valid flag: bit 31
                 // dirty flag: bit 30
   uint data;
};
CacheLine cache[64];
```

Single-byte read operation:


```
for ( int i = 0; i < 64; i++ )
   if (cache[i].valid)
      if (cache[i].tag == tag)
         return cache[i].data[offs];

uint d = RAM[tag].data; // cache miss

WriteToCache( tag, d );

return d[offs];
```

Architectures

Cache Architecture

The simplest caching scheme is the *fully associative cache*.

```
struct CacheLine
{
   uint tag;     // valid flag: bit 31
                 // dirty flag: bit 30
   uint data;
};
CacheLine cache[64];
```

This cache holds 64 dwords (256 bytes).

Single-byte write operation:

```
for ( int i = 0; i < 64; i++ )
   if (cache[i].valid)
      if (cache[i].tag == a)
         cache[i].data[offs] = d;
         cache[i].dirty = true;
         return;

for ( int i = 0; i < 64; i++ )
   if (!cache[i].valid)
      cache[i].tag = a;
      cache[i].data[offs] = d;
      cache[i].valid|dirty = true;
      return;

i = BestSlotToOverwrite();

if (cache[i].dirty) SaveToRam(i);
cache[i].tag = a;
cache[i].data[offs] = d;
cache[i].valid|dirty = true;
```

One problem remains… We store one byte, but the slot stores 4. What should we do with the other 3?

Architectures

BestSlotToOverwrite() ?

The best slot to overwrite is the one that will not be needed for the longest amount of time. This is known as Bélády’s algorithm, or the *clairvoyant* algorithm.

Alternatively, we can use:

- LRU: least recently used
- MRU: most recently used
- Random Replacement
- LFU: Least frequently used
- …

AMD and Intel use ‘pseudo-LRU’ (until Ivy Bridge; after that, things got complex*).

*: http://blog.stuffedcow.net/2013/01/ivb-cache-replacement

Architectures

The Problem with Being Fully Associative

Read / Write using a fully associative cache is O(N): we need to scan each entry. This is not practical for anything beyond 16~32 entries.

An alternative scheme is the *direct mapped cache*.

Architectures

Direct Mapped Cache

```
struct CacheLine
{
   uint tag;
   uint data;
};
CacheLine cache[64];
```

This cache again holds 256 bytes.

In a direct mapped cache, each address can only be stored in a single cache line.

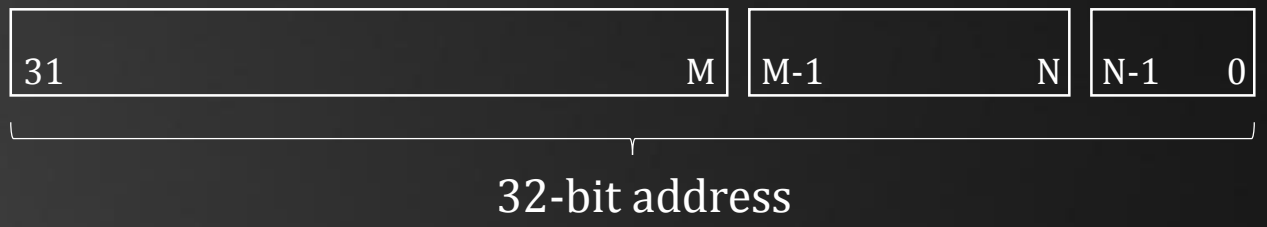
Read/write access is therefore O(1).

For a cache consisting of 64 cache lines:

- Bit 0 and 1 still determine the offset within a slot;
- 6 bits are used to determine which slot to use;
- The remaining bits are the tag.

Architectures

Direct Mapped Cache



In general:

$N = \log_2(cache\ line\ width)$
$M = \log_2(number\ of\ slots\ in\ the\ cache)$

- Bits 0..N-1 are used as offset in a cache line;
- Bits N..M-1 are used as slot index;
- Bits M..31 are used as tag.

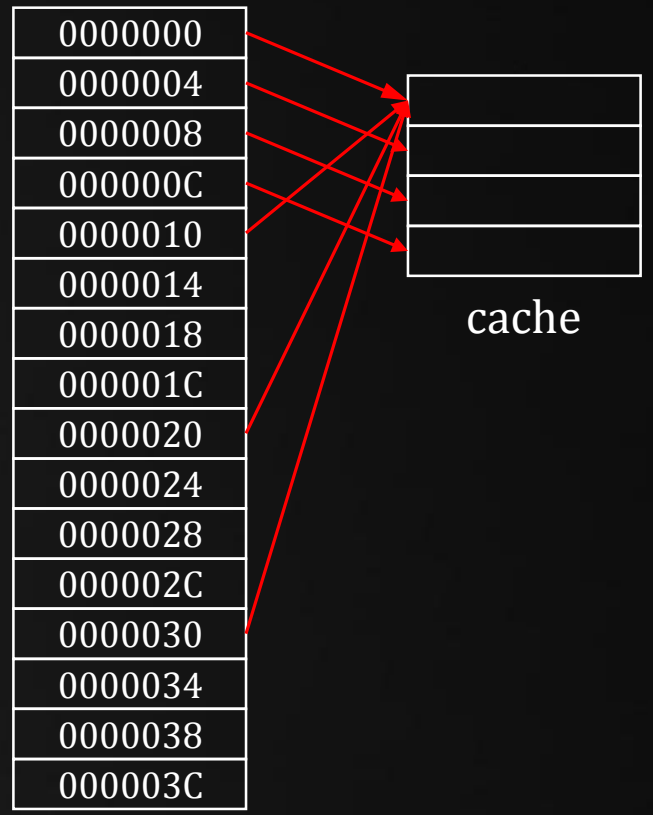
Architectures

The Problem with Direct Mapping

In this type of cache, each address maps to a single cache line, leading to O(1) access time. On the other hand, a single cache line ‘represents’ multiple memory addresses.

This leads to a number of issues:

- A program may use two variables that occupy the same cache line, resulting in frequent cache misses (collisions);
- A data layout may heavily use one part of the cache, and underutilize another.



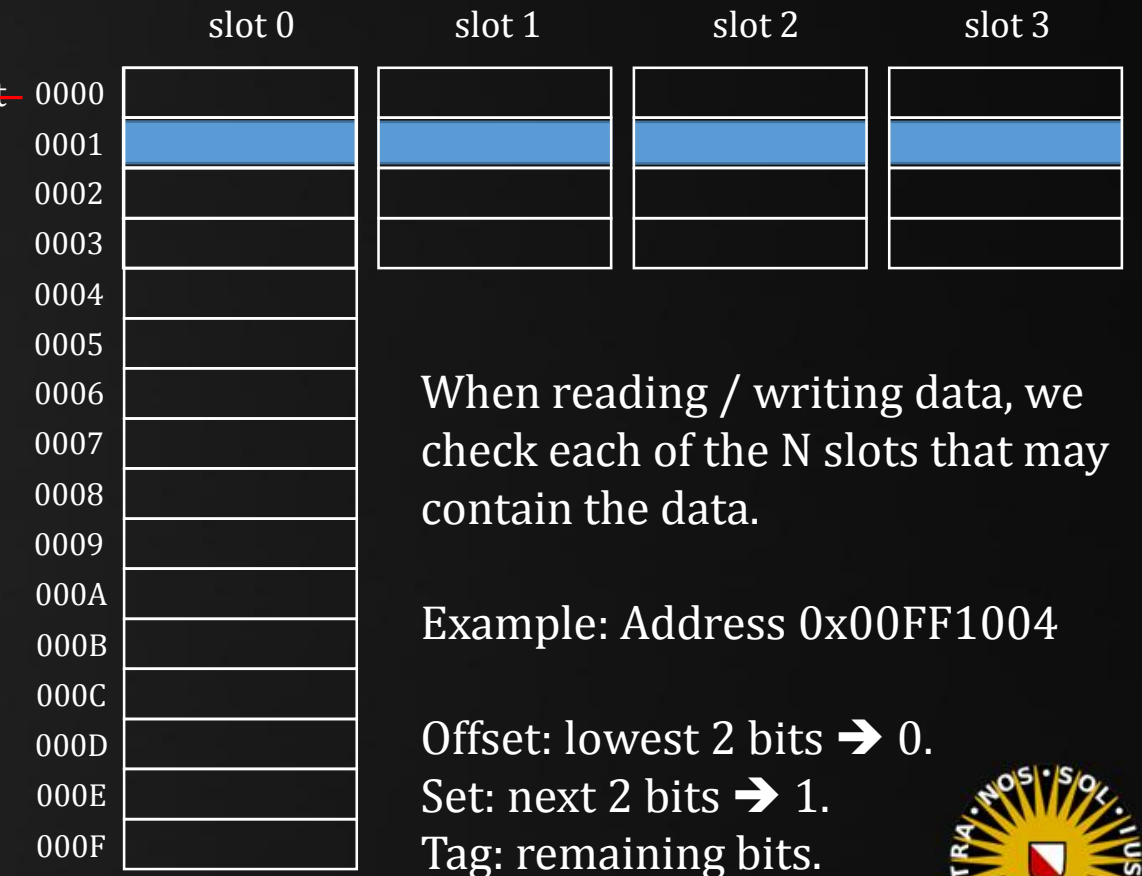
Architectures

N-Way Set Associative Cache

```
struct CacheLine
{
   uint tag;
   uint data;
};
CacheLine cache[16][4];
```

This cache again holds 256 bytes.

In an N-way set associative cache, we use sets of N slots per cache line.



When reading / writing data, we check each of the N slots that may contain the data.

Example: Address 0x00FF1004

Offset: lowest 2 bits ➔ 0.
Set: next 2 bits ➔ 1.
Tag: remaining bits.

Architectures

Caching Architectures

The Intel i7 processors use three on-die caches:

L1: 32KB 4-way set associative instruction cache + 32KB 8-way data cache per core
L2: 256KB 8-way set associative cache per core
L3: 2MB x cores global 16-way set associative cache.

The AMD Phenom also uses three on-die caches:

L1: 64KB 2-way set associative (32+32) per core
L2: 512KB 16-way set associative per core
L3: 1MB x cores global 48-way set associative cache.

Both AMD and Intel currently use 64 byte cache lines.

Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Practical Assignment 1

Assignment 1

First Practical Assignment: Create a Cache Simulator

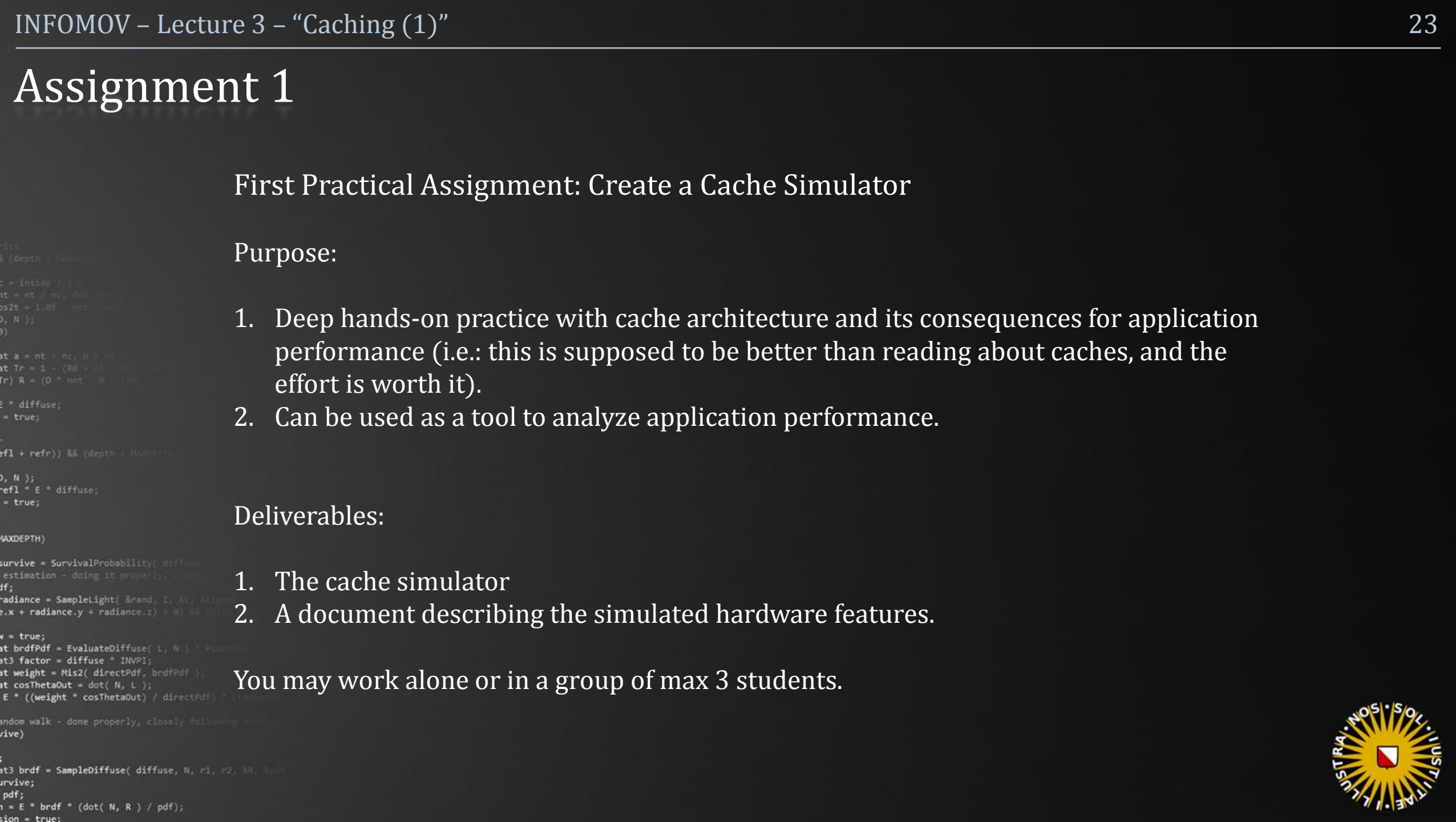
Purpose:

1. Deep hands-on practice with cache architecture and its consequences for application performance (i.e.: this is supposed to be better than reading about caches, and the effort is worth it).
2. Can be used as a tool to analyze application performance.

Deliverables:

1. The cache simulator
2. A document describing the simulated hardware features.

You may work alone or in a group of max 3 students.



Assignment 1

First Practical Assignment: Create a Cache Simulator

Details, minimum requirements (for a 6):

1. implement a correct set associative cache within the supplied demo application;
2. implement a reasonable eviction policy. This requires some research.

Optional (towards a perfect 10):

3. cover the full L1-L2-L3 cache hierarchy;
4. experiment with various eviction policies;
5. do a real-time visualization of cache efficiency;
6. implement read/write of 16 and 32-bit data types.

Limitations:

- you may assume a single core (i.e. cache coherency and shared L3 doesn’t have to be simulated);
- the simulator doesn’t have to be efficient (memory / speed wise).

Assignment 1

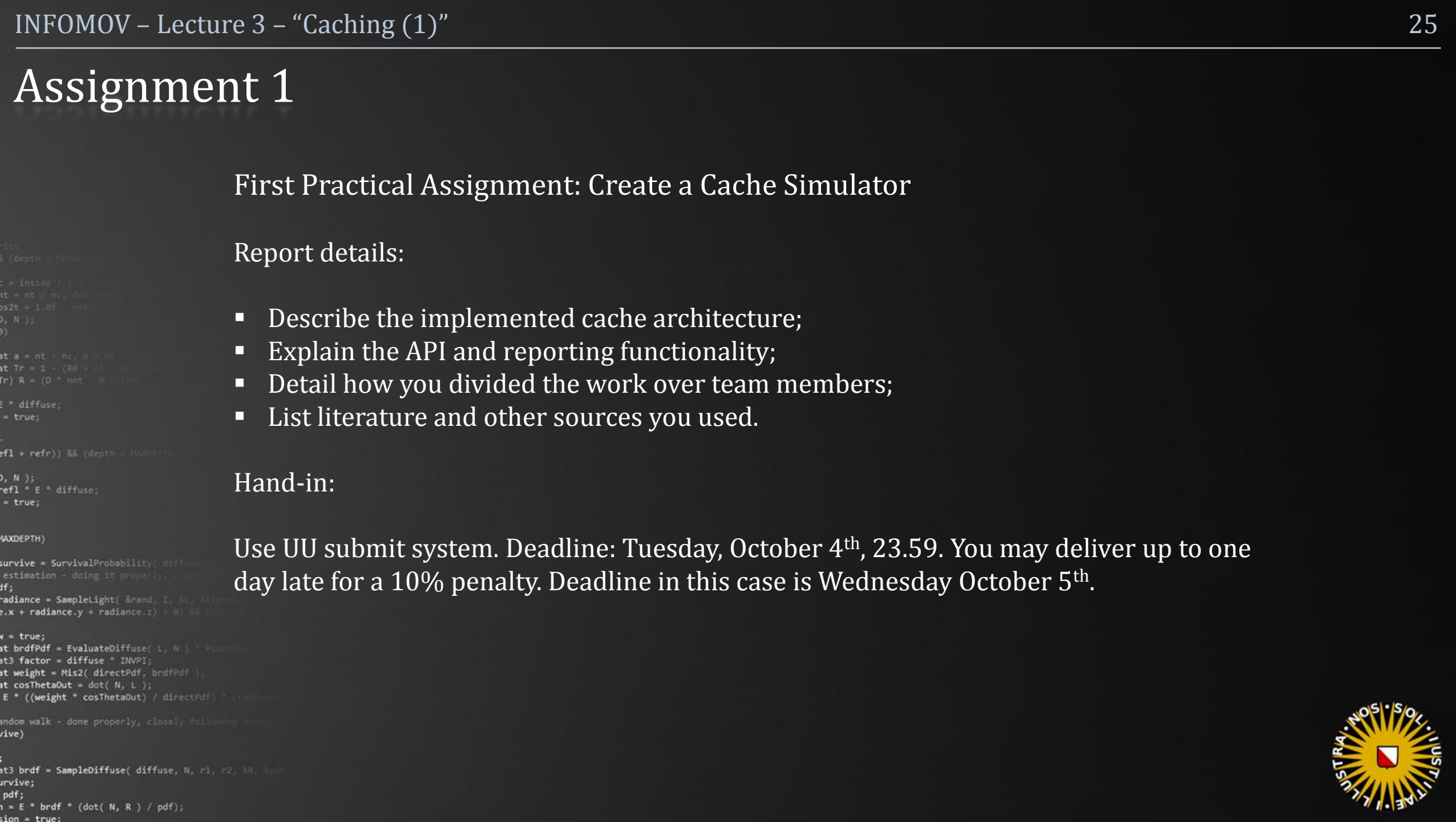
First Practical Assignment: Create a Cache Simulator

Report details:

- Describe the implemented cache architecture;
- Explain the API and reporting functionality;
- Detail how you divided the work over team members;
- List literature and other sources you used.

Hand-in:

Use UU submit system. Deadline: Tuesday, October 4th, 23.59. You may deliver up to one day late for a 10% penalty. Deadline in this case is Wednesday October 5th.



Today’s Agenda:

- The Problem with Memory
- Cache Architectures
- Practical Assignment 1

/INFOMOV/

END of “Caching (1)”

next lecture: “low level (2)”