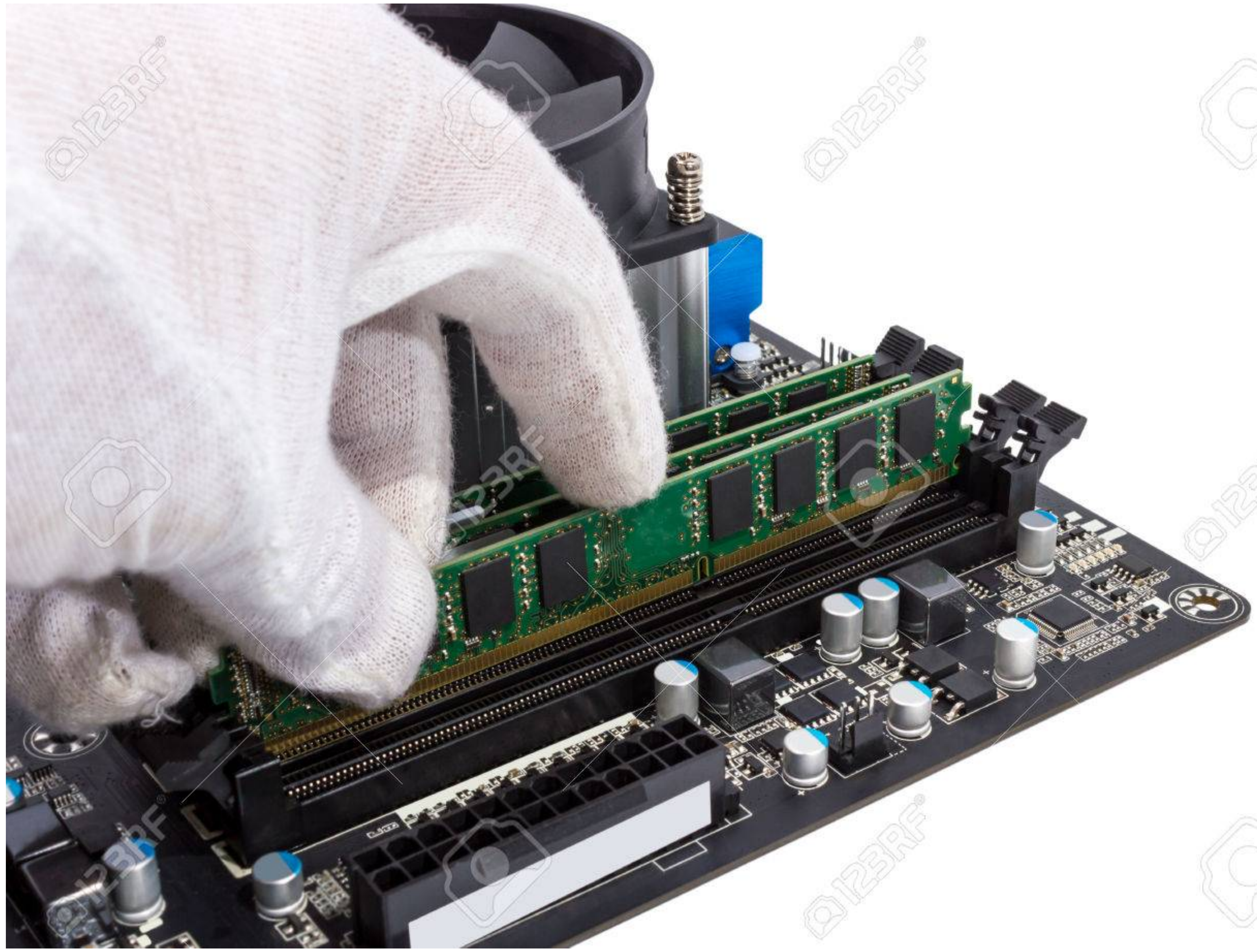
From Memory to Context

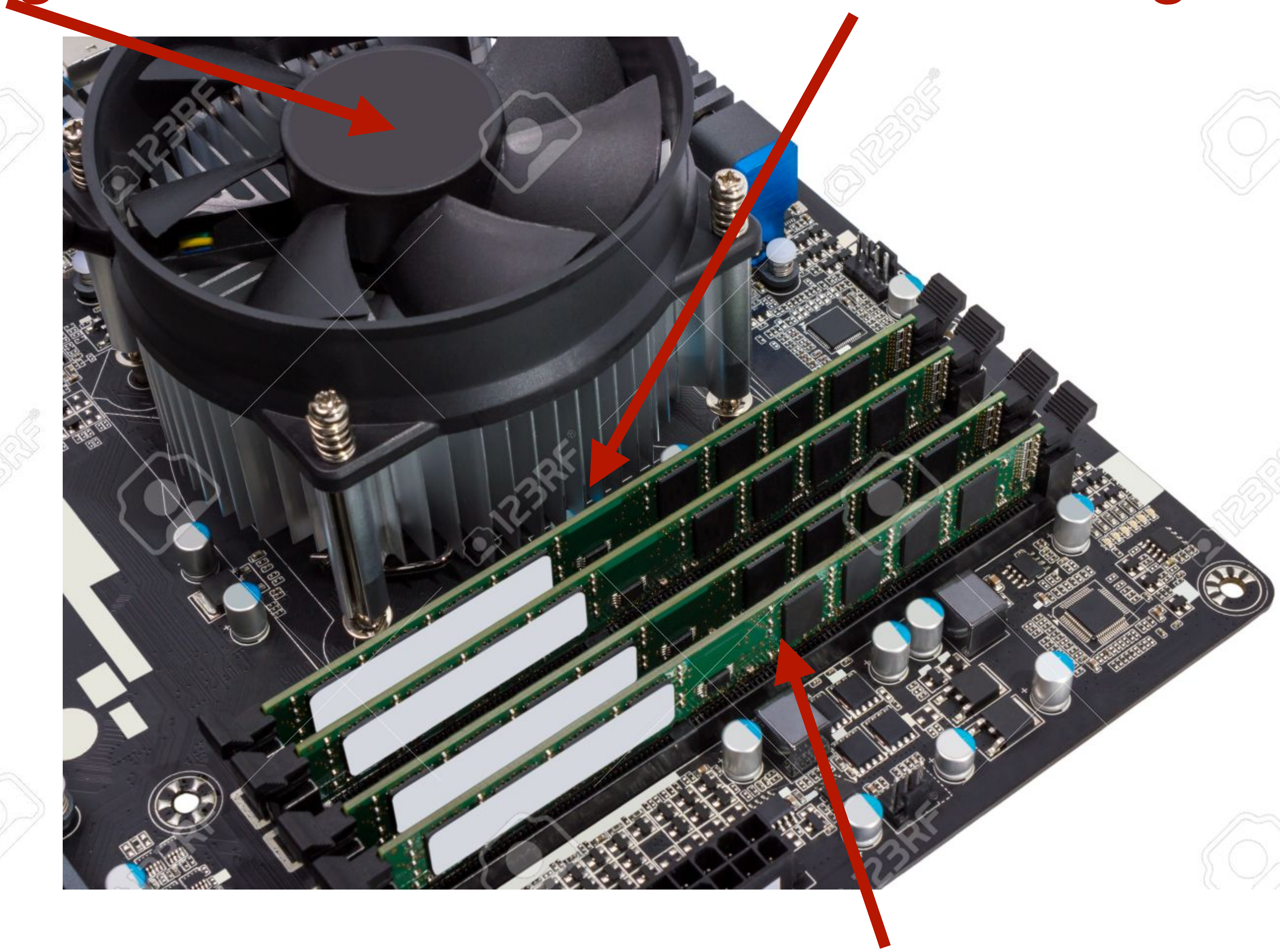
Yunhao Zhang

What is memory? (continue)

What is context?

What is memory?





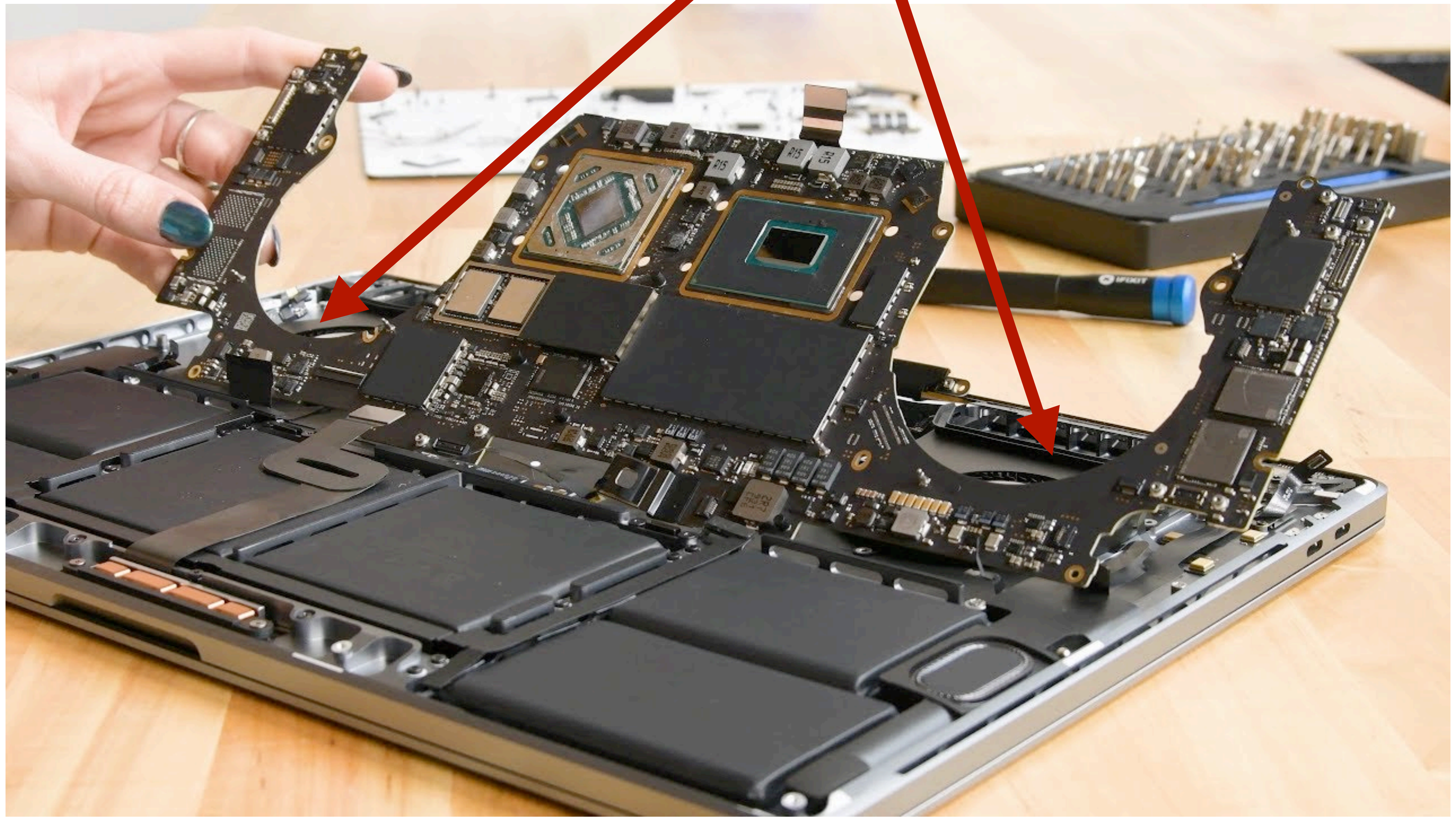
cooling fan

CPU under the cooling fan

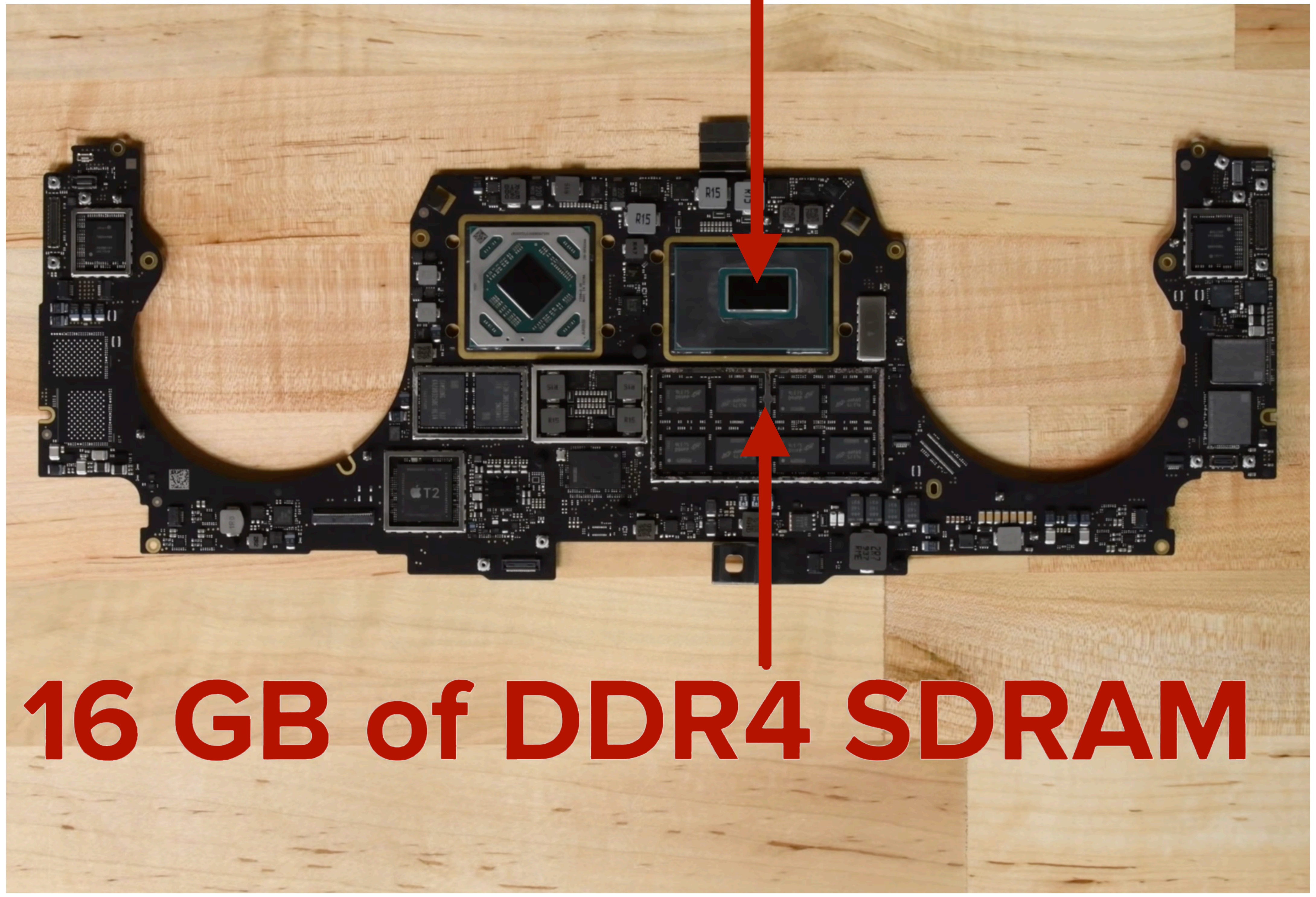
Memory

* Images from https://www.123rf.com/

What is memory?



cooling fan



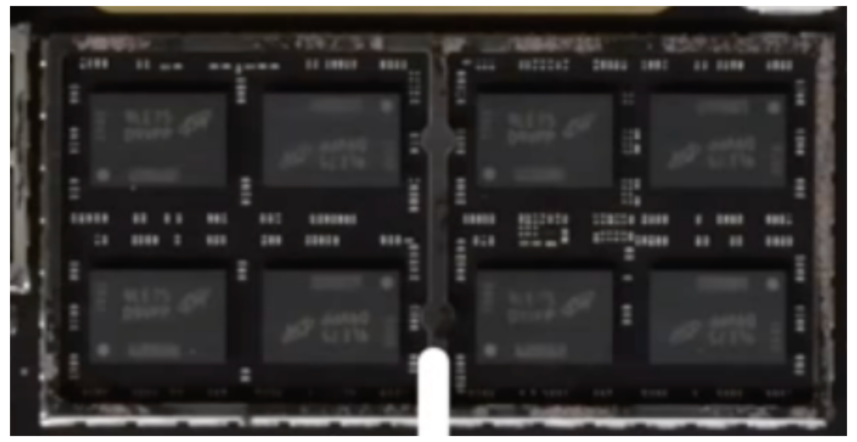
Intel i7 CPU

16 GB of DDR4 SDRAM

* https://www.youtube.com/watch?v=Mjb12GCKycw

What is memory?

- ECE students study
  - pins, voltage, clock frequency, … of



- CS students study
  - a simple abstraction: memory address space

Memory address space

- A simple table with two rows: content and address
  - e.g., 32bit address space can represent up to 2^32 bytes

Content	1st byte	2nd byte	3rd byte	……	2^32th byte
Address	0x0000 0000	0x0000 0001	0x0000 0002	……	0xFFFF FFFF

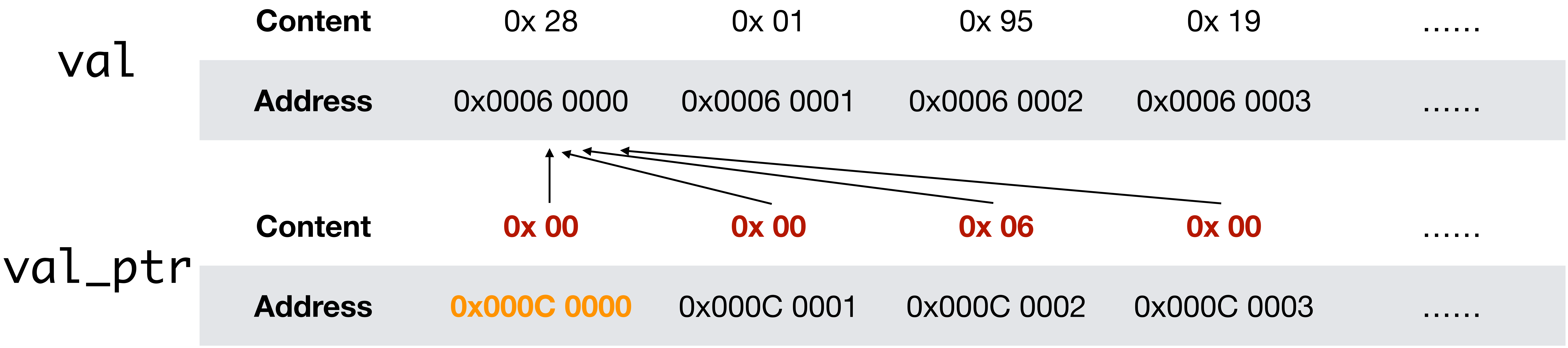
A variable is just some bytes

```
int val = 0x19950128;

// compiler decides where to put val
// say the compiler puts val at 0x60000
```

Content	**0x 28**	**0x 01**	**0x 95**	**0x 19**	……
Address	**0x0006 0000**	0x0006 0001	0x0006 0002	0x0006 0003	……

A pointer variable stores an address

```
int val = 0x19950128;

int* val_ptr = &val;
// say the compiler puts val_ptr at 0xC0000
```



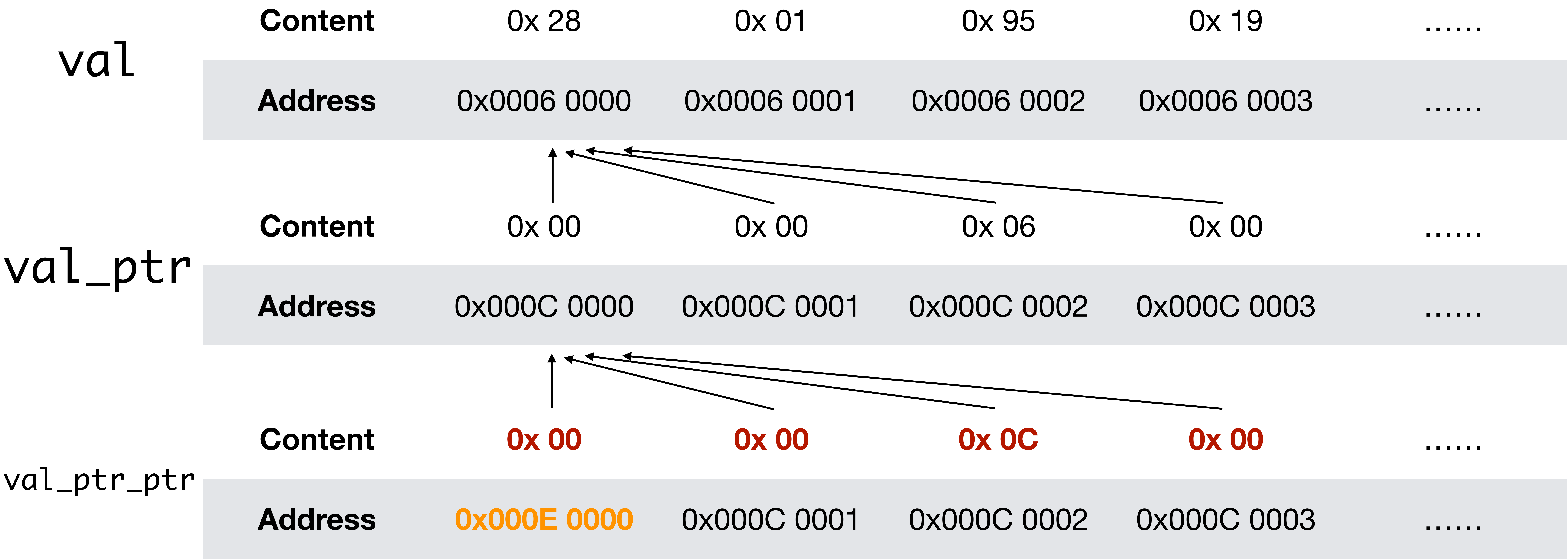
val

Content	0x 28	0x 01	0x 95	0x 19	......
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	......

val_ptr

Content	**0x 00**	**0x 00**	**0x 06**	**0x 00**	......
Address	**0x000C 0000**	0x000C 0001	0x000C 0002	0x000C 0003	......

```
int val = 0x19950128;
int* val_ptr = &val;

int** val_ptr_ptr = &val_ptr;
// say the compiler puts val_ptr_ptr at 0xE0000
```



val

Content	0x 28	0x 01	0x 95	0x 19	……
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	……

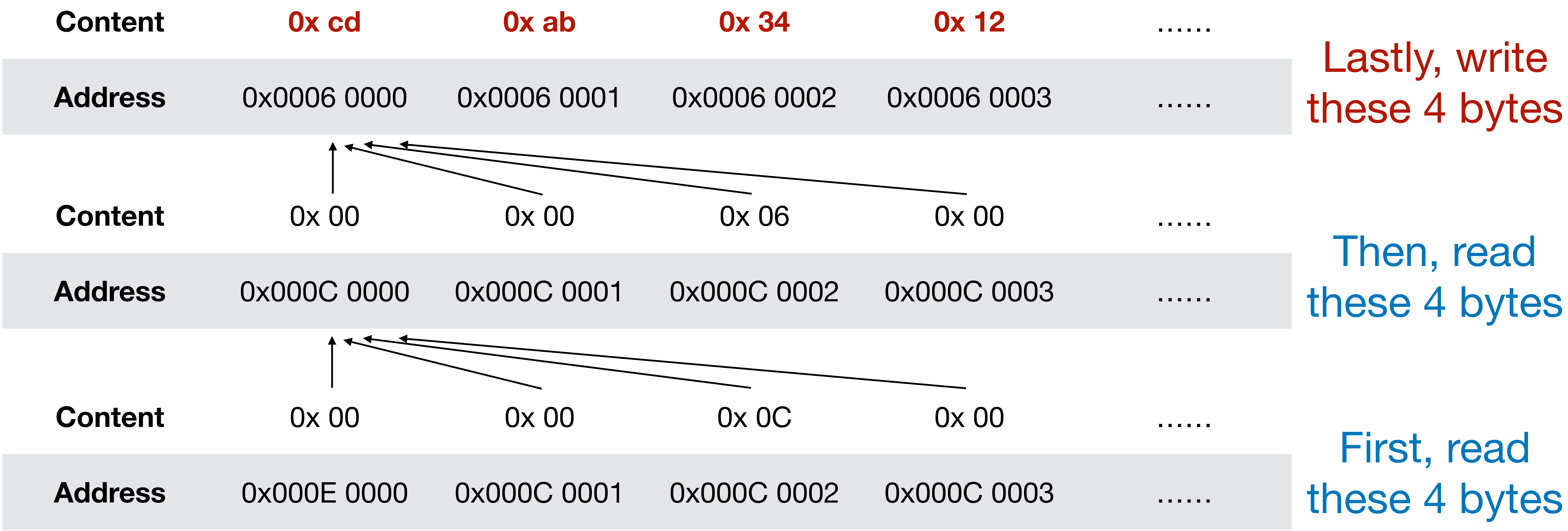
val_ptr

Content	0x 00	0x 00	0x 06	0x 00	……
Address	0x000C 0000	0x000C 0001	0x000C 0002	0x000C 0003	……

val_ptr_ptr

Content	**0x 00**	**0x 00**	**0x 0C**	**0x 00**	……
Address	**0x000E 0000**	0x000C 0001	0x000C 0002	0x000C 0003	……

```
int val = 0x19950128;
int* val_ptr = &val;
int** val_ptr_ptr = &val_ptr;

void update() { **val_ptr_ptr = 0x1234abcd; }
```



Content	**0x cd**	**0x ab**	**0x 34**	**0x 12**	……
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	……

Lastly, write these 4 bytes

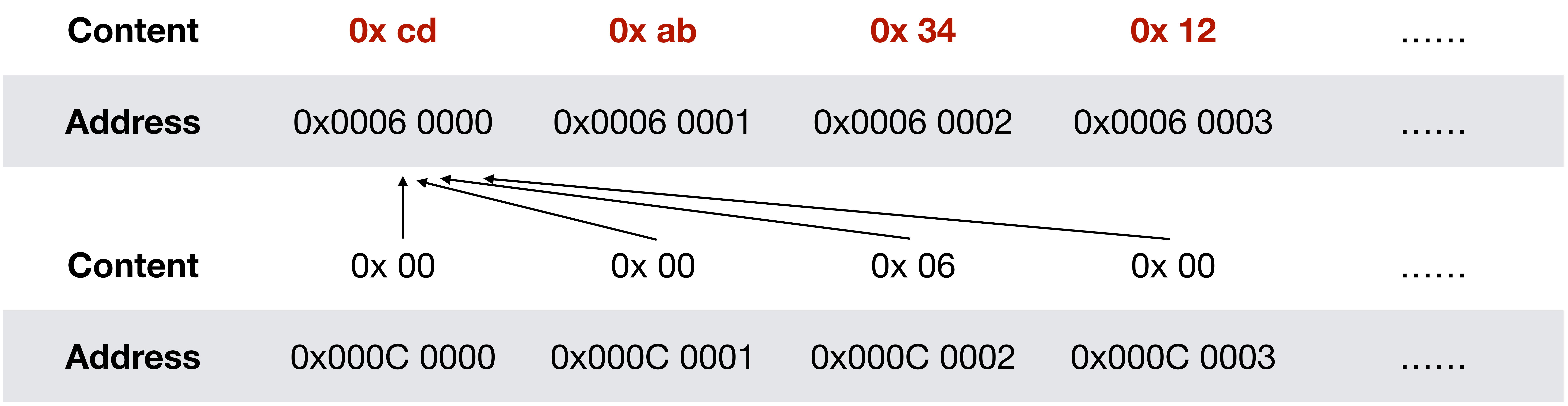
Content	0x 00	0x 00	0x 06	0x 00	……
Address	0x000C 0000	0x000C 0001	0x000C 0002	0x000C 0003	……

Then, read these 4 bytes

Content	0x 00	0x 00	0x 0C	0x 00	……
Address	0x000E 0000	0x000C 0001	0x000C 0002	0x000C 0003	……

First, read these 4 bytes

```
int val = 0x19950128;
int* val_ptr = &val;
int** val_ptr_ptr = &val_ptr;

void update() { *val_ptr = 0x1234abcd; }
```



Content	**0x cd**	**0x ab**	**0x 34**	**0x 12**	……
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	……

Then, write these 4 bytes

Content	0x 00	0x 00	0x 06	0x 00	……
Address	0x000C 0000	0x000C 0001	0x000C 0002	0x000C 0003	……

First, read these 4 bytes

```
int val = 0x19950128;
int* val_ptr = &val;
int** val_ptr_ptr = &val_ptr;

void update() { val = 0x1234abcd; }
```

Content	**0x cd**	**0x ab**	**0x 34**	**0x 12**	……
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	……

Write these 4 bytes

```
int val = 0x19950128;         // integer
int* val_ptr = &val;          // pointer
int** val_ptr_ptr = &val_ptr; // pointer of pointer


void update() {
    val           = 0x1234abcd; // write directly
    *val_ptr      = 0x1234abcd; // read then write
    **val_ptr_ptr = 0x1234abcd; // read*2 then write
}
```

Types tell the size of variables

```
int val = 0x19950128;
int* val_ptr = &val;
int** val_ptr_ptr = &val_ptr;
```

```
sizeof(val)         == sizeof(int)    == 4
sizeof(val_ptr)     == sizeof(int*)   == 4  // 32bit CPU
sizeof(val_ptr_ptr) == sizeof(int**)  == 4  // 32bit CPU
```

Different types have different sizes

Type	**sizeof(Type)**	**Type**	**sizeof(Type)** (32bit CPU)	**sizeof(Type)** (64bit CPU)
char	1	**char***	**4**	8
int	4	**int***	**4**	8
long long	8	**long long***	**4**	8
float	4	**float***	**4**	8
void	cannot define variable of void	**void***	**4**	8

Type conversion

```
int val = 0x19950128;
int* val_ptr = &val;

char c = *((char*) val_ptr);
// convert int* to char*
// the same 4 bytes as an address
// but compiler read 1 byte from that address
// after the type conversion
```

Type conversion

```
int val = 0x19950128;
int* val_ptr = &val;
char c = *((char*) val_ptr);
```

Content	0x 28	0x 01	0x 95	0x 19	……
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	……

Then, read 1 byte here

Content	0x 00	0x 00	0x 06	0x 00	……
Address	0x000C 0000	0x000C 0001	0x000C 0002	0x000C 0003	……

First, read these 4 bytes

Content	**0x 28**	……	……	……	……
Address	0x000F 0000	……	……	……	……

Lastly, write 1 byte here

Why void pointer ?

```
char  item0 = ‘a’;
int   item1 = 0x19950128;
float item2 = 3.14;
                            // It’s up to the user of the
queue_t q = queue_new();    // queue to decide whether
                            // the item is char, int or
// queue is generic         // other types
queue_enqueue(q, &item0);
queue_enqueue(q, &item1);
queue_enqueue(q, &item2);
```

```
char  item0 = ‘a’;
int   item1 = 0x19950128;
float item2 = 3.14;

queue_t q = queue_new();

// queue is generic
queue_enqueue(q, &item0);
queue_enqueue(q, &item1);
queue_enqueue(q, &item2);
```

```
char*  item3;
int*   item4;
float* item5;

queue_dequeue(q, &item3);
queue_dequeue(q, &item4);
queue_dequeue(q, &item5);

// now
// item3 == &item0
// item4 == &item1
// item5 == &item2
```

Why pointer of pointer ?

```
char*  item3;
int*   item4;
float* item5;                // queue_dequeue needs to
                             // modify item3, so it takes
queue_dequeue(q, &item3);    // the address of item3, namely
queue_dequeue(q, &item4);    // &item3, as parameter
queue_dequeue(q, &item5);
                             // and &item3 is a pointer to
// item3 == &item0           // a pointer with type void**
// item4 == &item1
// item5 == &item2
// Global variable
char item0 = ‘a’;

void test() {
  char* item3;
  queue_enqueue(q, &item0);
  queue_dequeue(q, &item3);
  // item3 == &item0
}
```

```
int queue_dequeue(queue_t q,
void** pitem){

pitem = … // modifies the local
          // variable pitem

*pitem = … // modifies the item3
           // variable in test()

**pitem = … // modifies ???

// only 1 of these 3 is needed

}
```

Some more C features

other than pointers

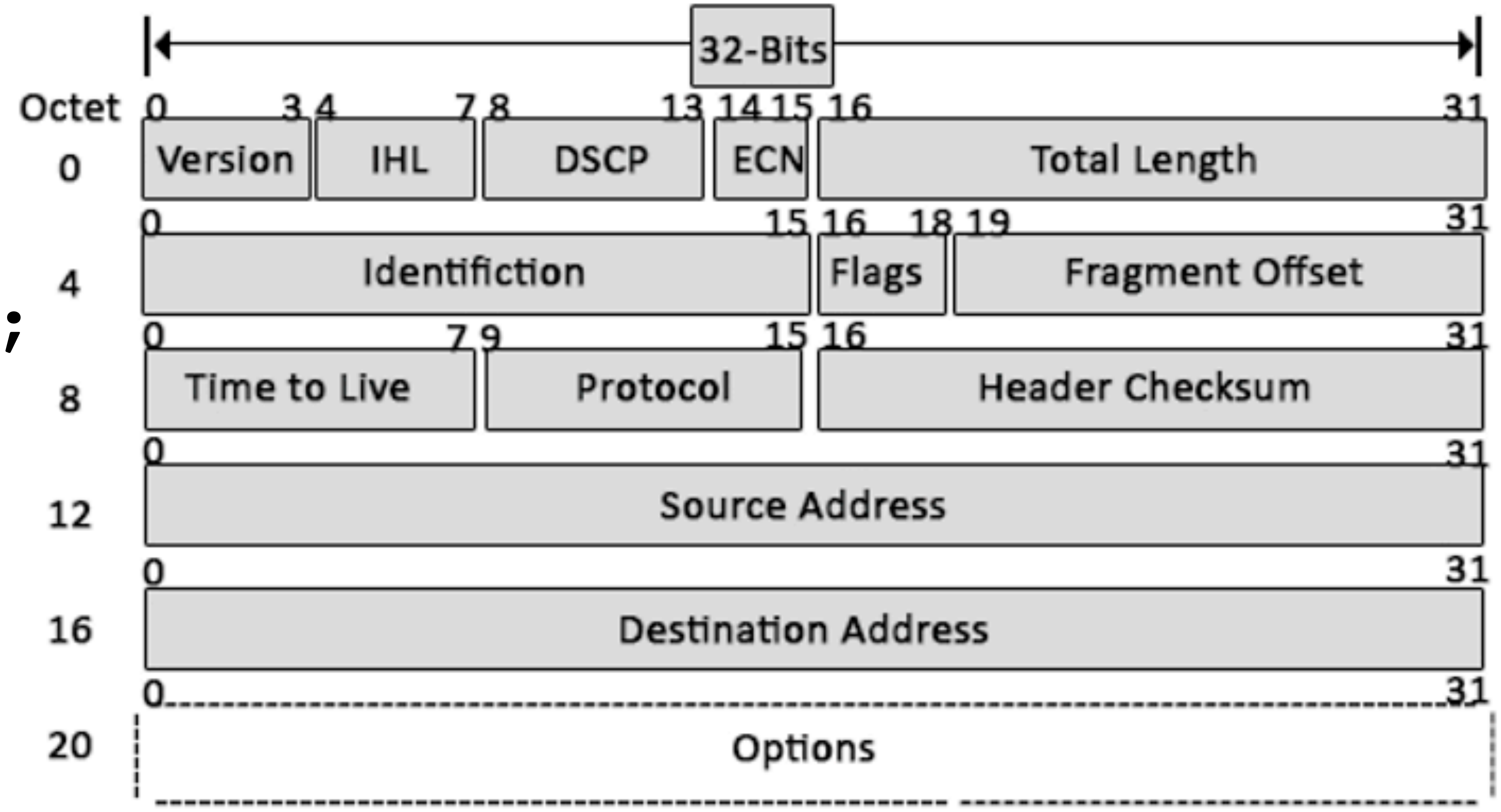
Why using typedef ?

```
typedef unsigned int   uint32_t;

// Now, the following are equivalent
unsigned int val;
uint32_t     val;
// But uint32_t makes it clear that
// the size of variable is 32 bits and it is unsigned

// Similarly
typedef unsigned char  uint8_t;
typedef unsigned short uint16_t;
```

Why using struct ?

```
struct header {
    uint8_t version:4;
    uint8_t ihl:4;
    uint8_t tos;
    uint16_t len;
    uint16_t id;
    uint16_t flags:3;
    uint16_t frag_offset:13;
    uint8_t ttl;
    uint8_t proto;
    uint16_t csum;
    uint32_t saddr;
    uint32_t daddr;
};
```

```
// Data have structures
// For example,
// an IPv4 network packet header:
```



Why using malloc() ?

```
// Consider a network packet queue; Every network packet
// has a header and a payload; The payload size is unknown

void recv_packet() {
  struct header* header = malloc(sizeof(struct header));
  net_recv_header(header);
  char* payload = malloc(header->payload_size);
  net_recv_payload(payload);

  queue_enqueue(q, header);
  queue_enqueue(q, payload);
}
```

Why using free() ?

```
void recv_packet() {
  struct header* header = malloc(sizeof(struct header));
  char* payload = malloc(header->payload_size);
  ……
}

void process_packet {
  struct header* header;        char* payload;
  queue_dequeue(q, header);     queue_dequeue(q, payload);
  ……
  // when header and payload are no longer useful
  free(header);                 free(payload);
}
```

What is memory?

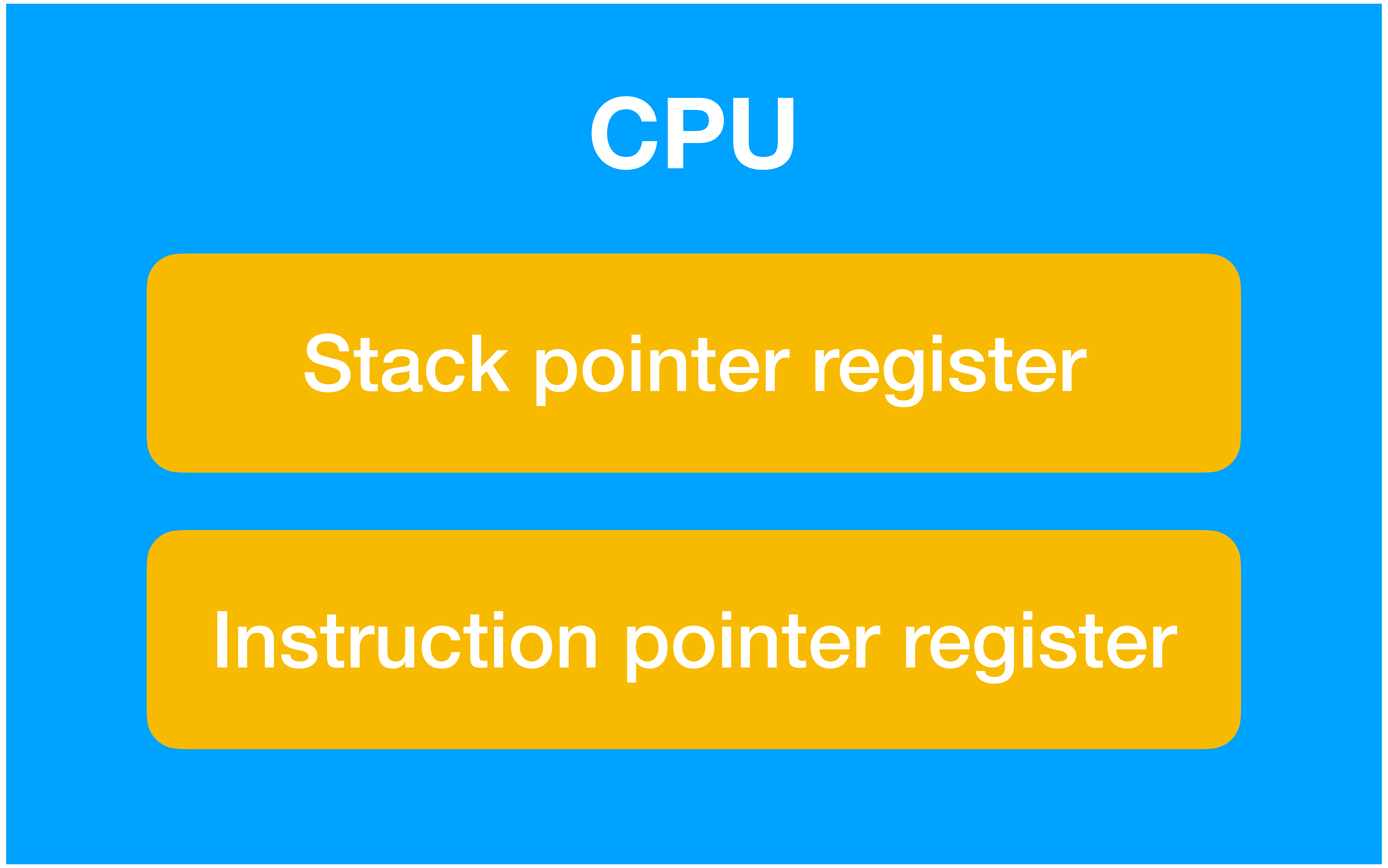
What is context?

Lesson1: the minimal memory requirement of program execution is code and stack.

For example, MacOS puts the code & stack of both my zoom and Keynote in memory.

zoom stack   **zoom** code   **Keynote** code   **Keynote** stack

Content	1st byte	......	......	......	2^32th byte
Address	0x0000 0000	......	......	......	0xFFFF FFFF



CPU

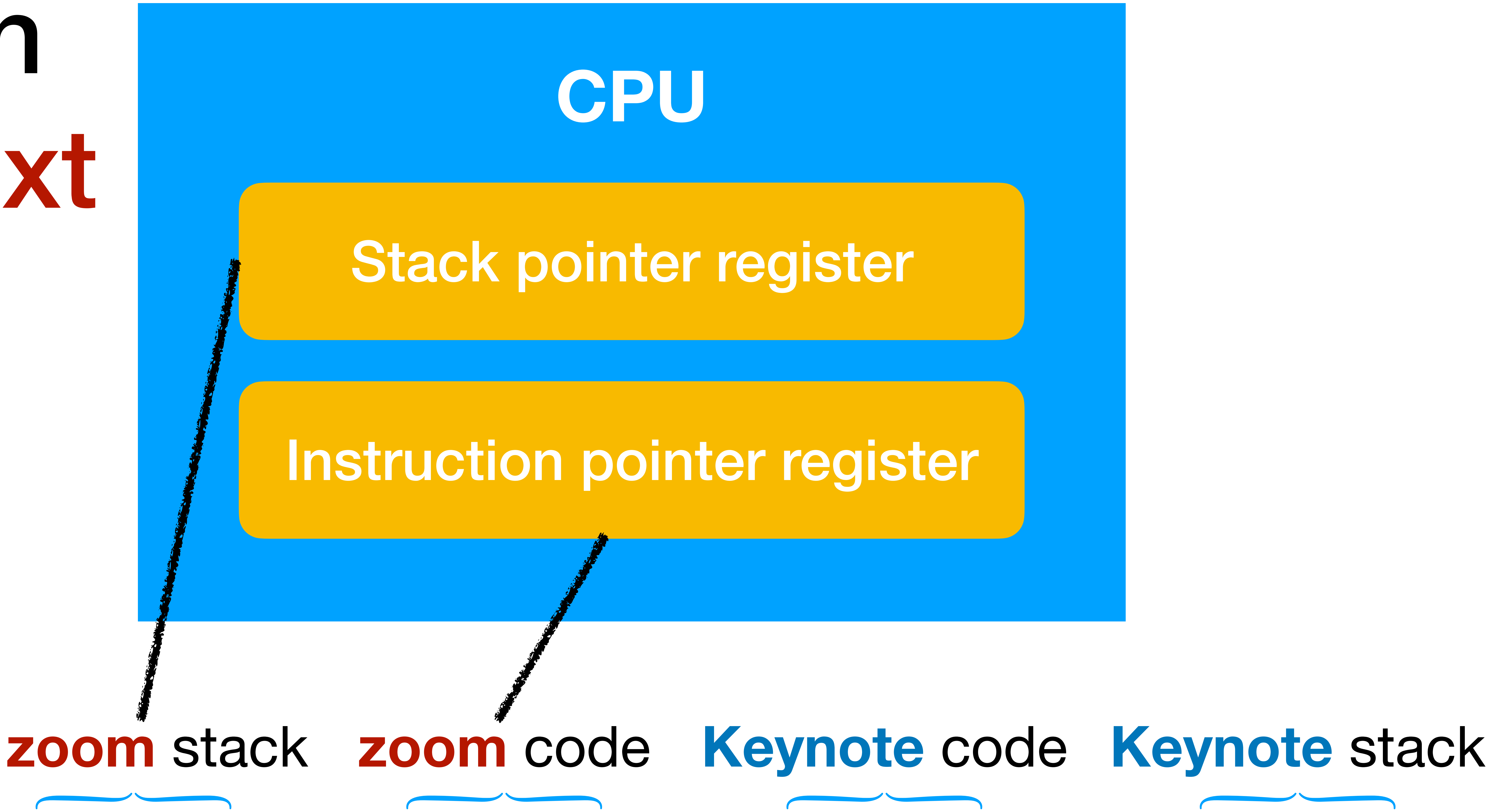
Stack pointer register

Instruction pointer register

zoom stack   zoom code   Keynote code   Keynote stack

Content	1st byte	......	......	......	2^32th byte
Address	0x0000 0000	......	......	......	0xFFFF FFFF

CPU is in the context of zoom



CPU

Stack pointer register

Instruction pointer register

zoom stack

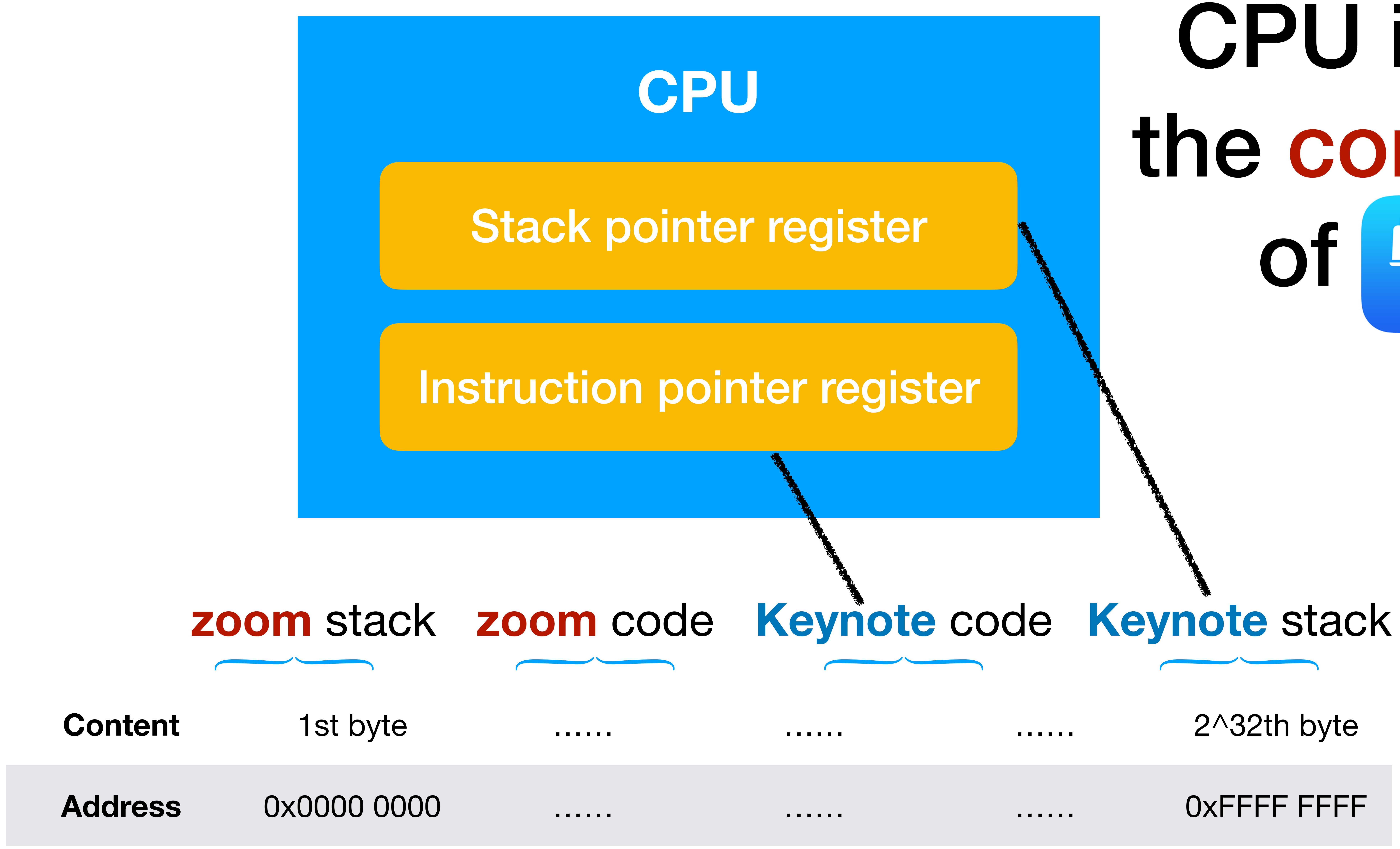
zoom code

Keynote code

Keynote stack

Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

CPU is in the context of



Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

Lesson2

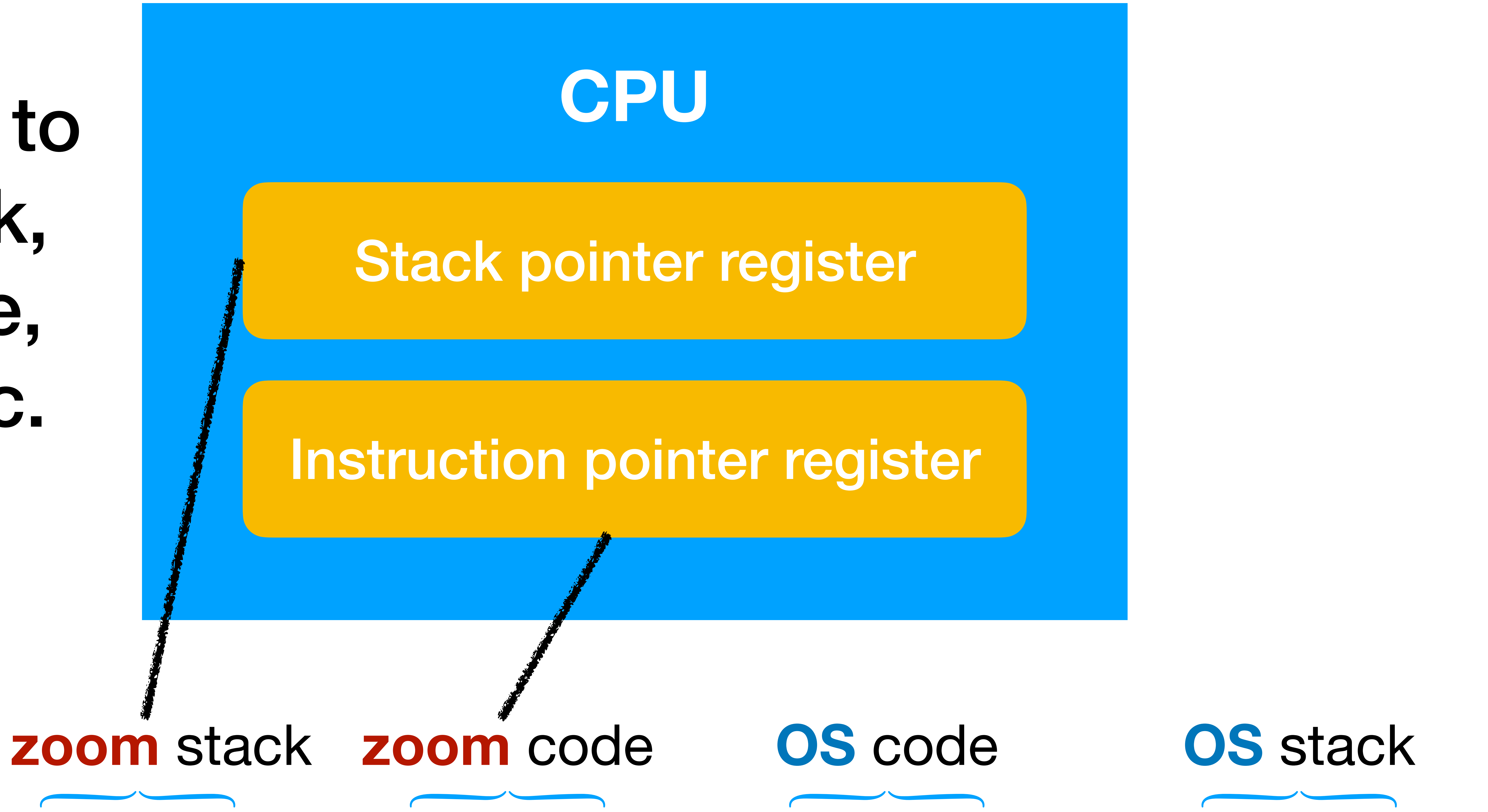
context = memory address space
+ stack pointer + instruction pointer

Operating system is just a program

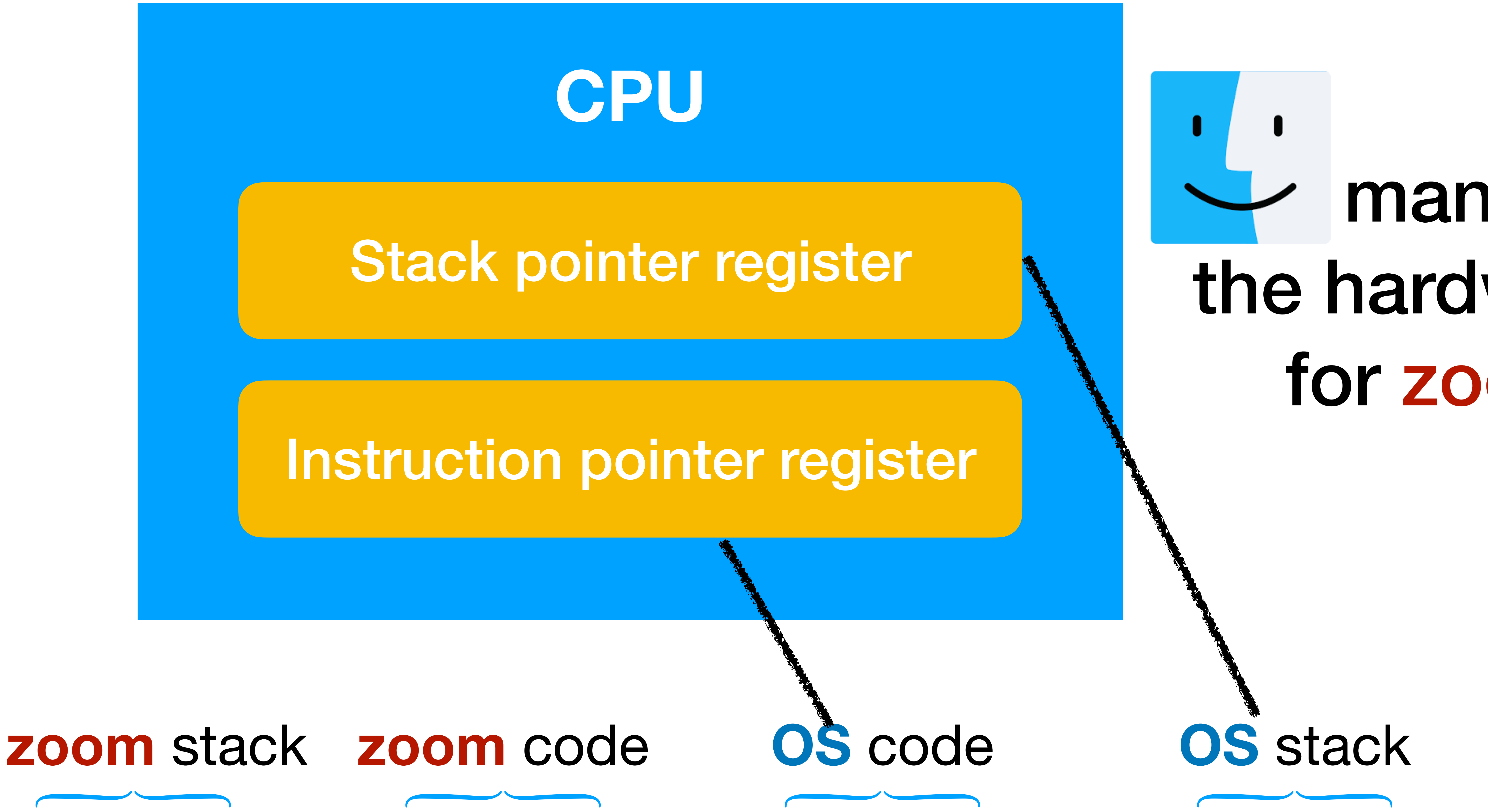
zoom stack zoom code OS code OS stack

Content	1st byte	......	......	......	2^32th byte
Address	0x0000 0000	......	......	......	0xFFFF FFFF

zoom wants to use network, microphone, speaker, etc.



Content	1st byte	......	......	......	2^32th byte
Address	0x0000 0000	......	......	......	0xFFFF FFFF



CPU

Stack pointer register

Instruction pointer register

zoom stack

zoom code

OS code

OS stack

manages the hardwares for zoom

Content	1st byte	……	……	……	2^32th byte
Address	0x0000 0000	……	……	……	0xFFFF FFFF

Summary

- **What is memory?**
  - A simple abstraction: address + content
  - A variable is just some bytes starting at some address
  - Bytes held by a pointer variable represent an address
- **What is context?**
  - The minimal memory requirement of any program is code & stack
  - context = memory address space + stack pointer + instruction pointer
  - At any moment, a CPU is in the context of some program