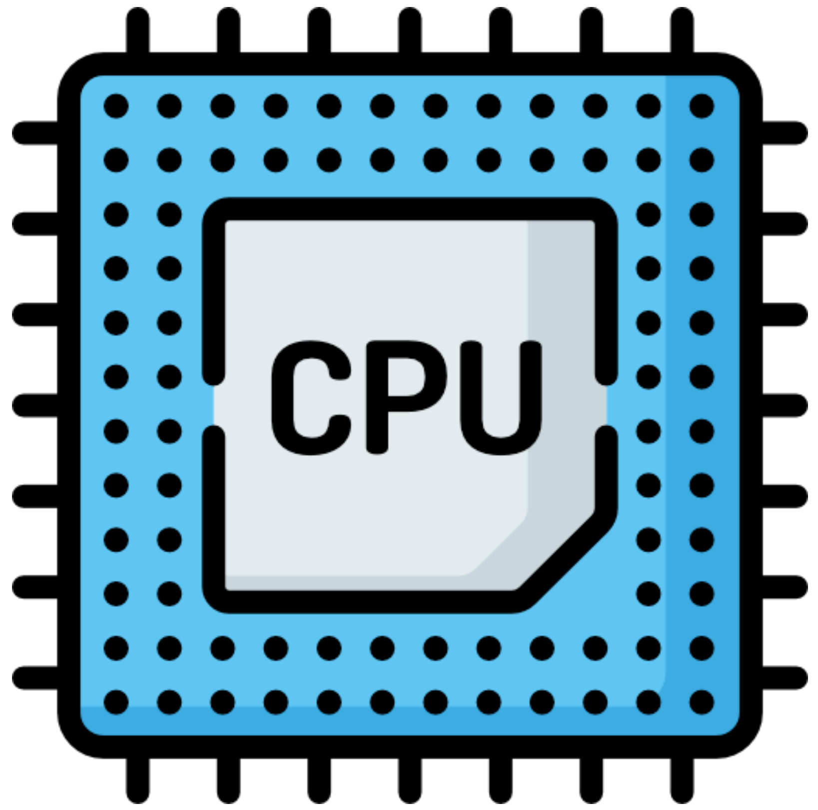
Memory and Pointers in C

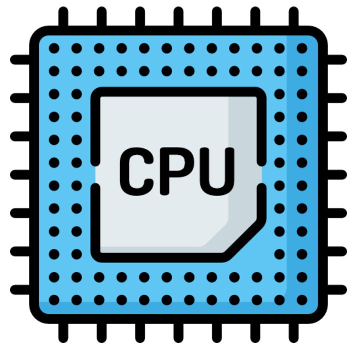
Why memory and pointers?

Interface



Recall RISC-V from CS3410

Interface



- load / store instructions
- instruction / stack pointer registers

CS 2110 — data structures and algorithms

Object Reference

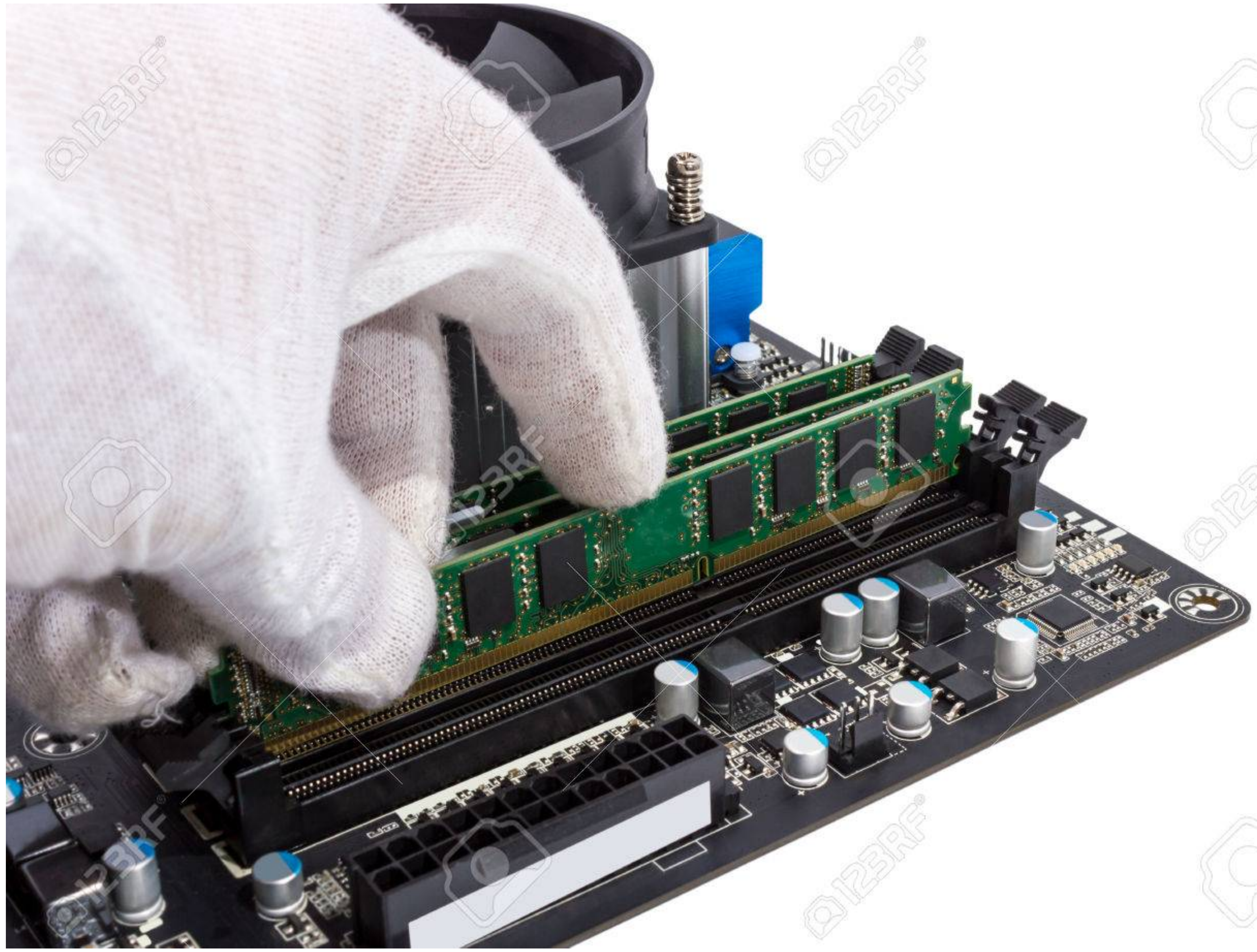
CS 4120 — Java, python™

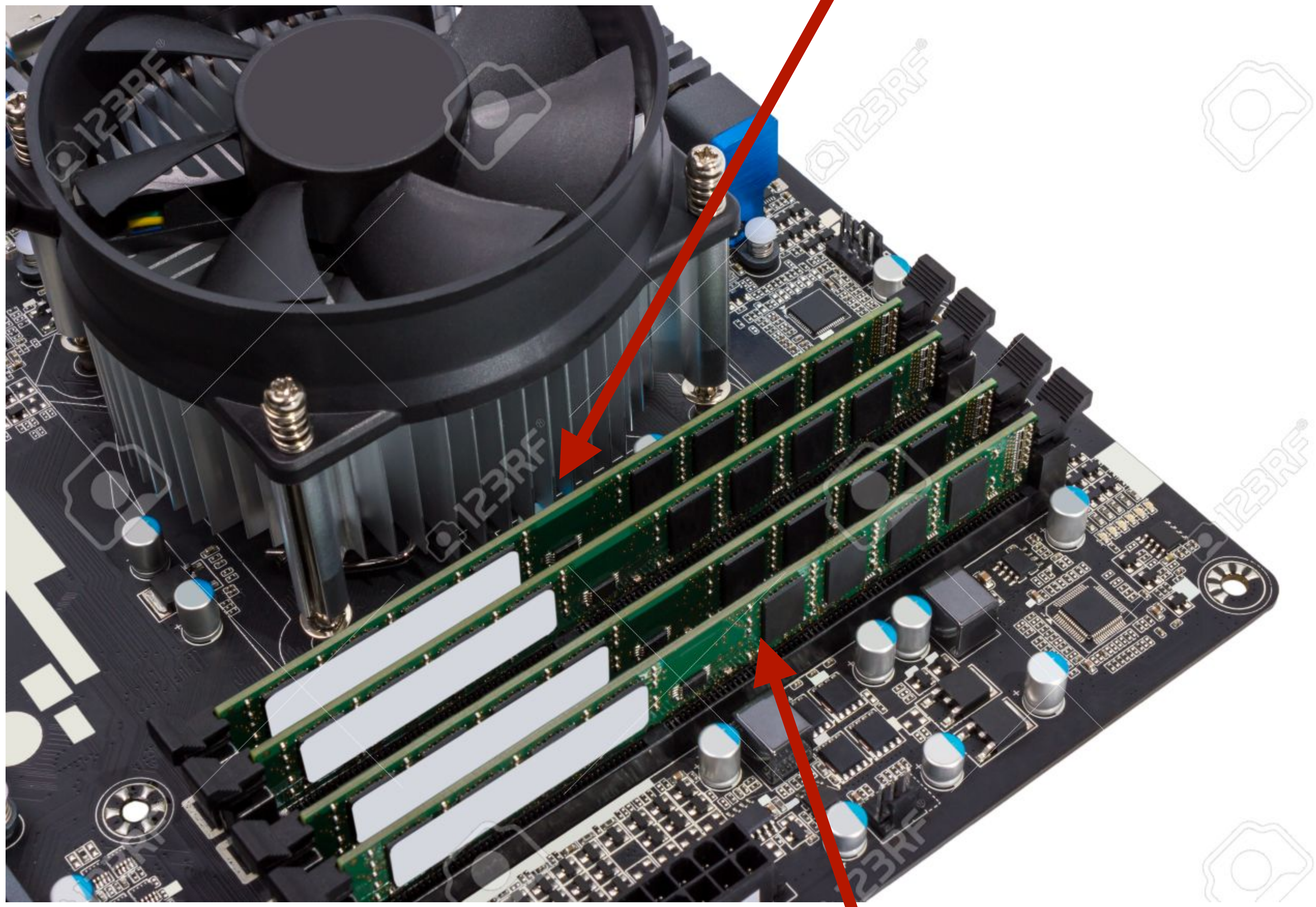
CS 4410
CS 4411 — Linux, mac OS, Windows

Memory Pointer

CS 3410 — CPU, memory

Physical memory

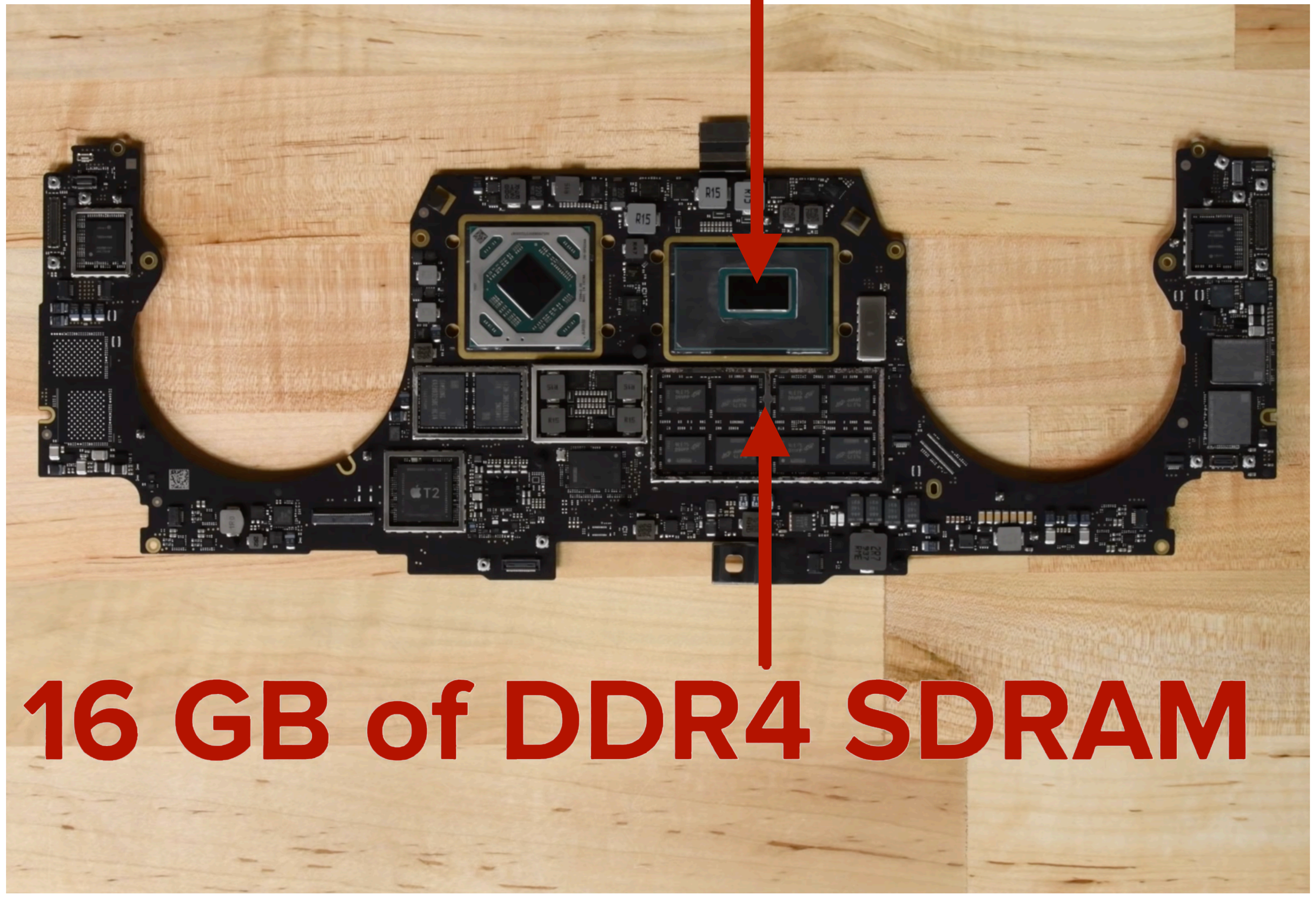




* Images from https://www.123rf.com/

Physical memory

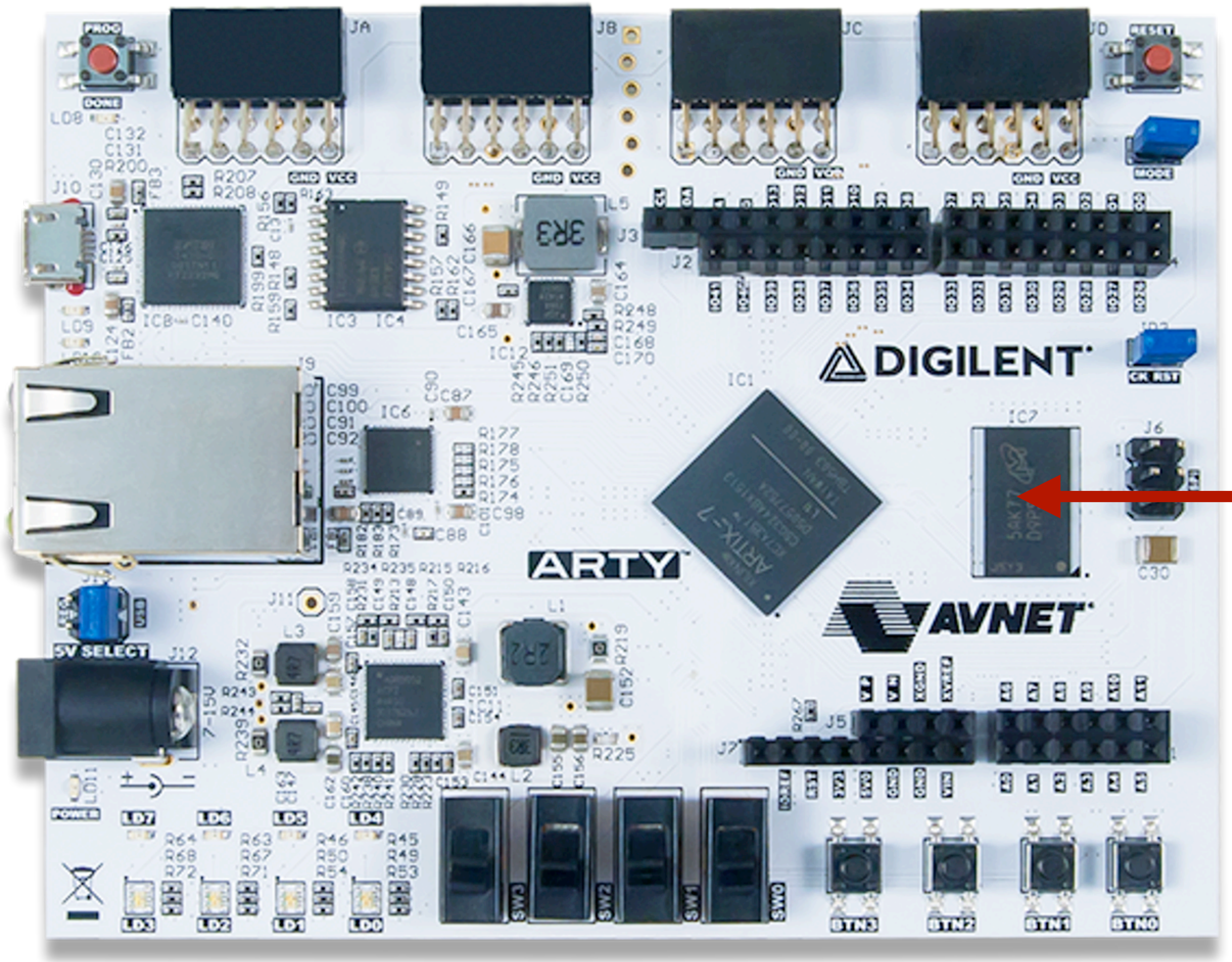
Intel i7 CPU



16 GB of DDR4 SDRAM

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

Physical memory



256MB of DDR3L

* https://digilent.com/shop/arty-a7-artix-7-fpga-development-board/

From physics to abstraction

- An ECE course would study voltage, current, etc.
- A CS course studies the abstraction of memory.
  - i.e., a simple math model, such as

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

From abstraction to C variable

- A variable in C language is a range of memory.
- For example,

```
int val = 0x19950128;
// say a RISC-V C compiler puts val at 0x60000
//      li t0, 0x60000
//      li t1, 0x19950128
//      sw t1, 0(t0)
```

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

From C variable to address

```
int val = 0x19950128;
int* val_ptr = &val;
// say the compiler puts val_ptr at 0xC0000
```

val_ptr

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

val

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

From address back to C variable

```
int val = 0x19950128;
int* val_ptr = &val;
*val_ptr = 0x1234ABCD;
```

Step1: read val_ptr as an address

val_ptr

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

val

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

Dereference a pointer: 2 steps

```
int val = 0x19950128;
int* val_ptr = &val;
*val_ptr = 0x1234ABCD;
```

Step1: read val_ptr as an address

Step2: write a value to that address

val_ptr

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

val

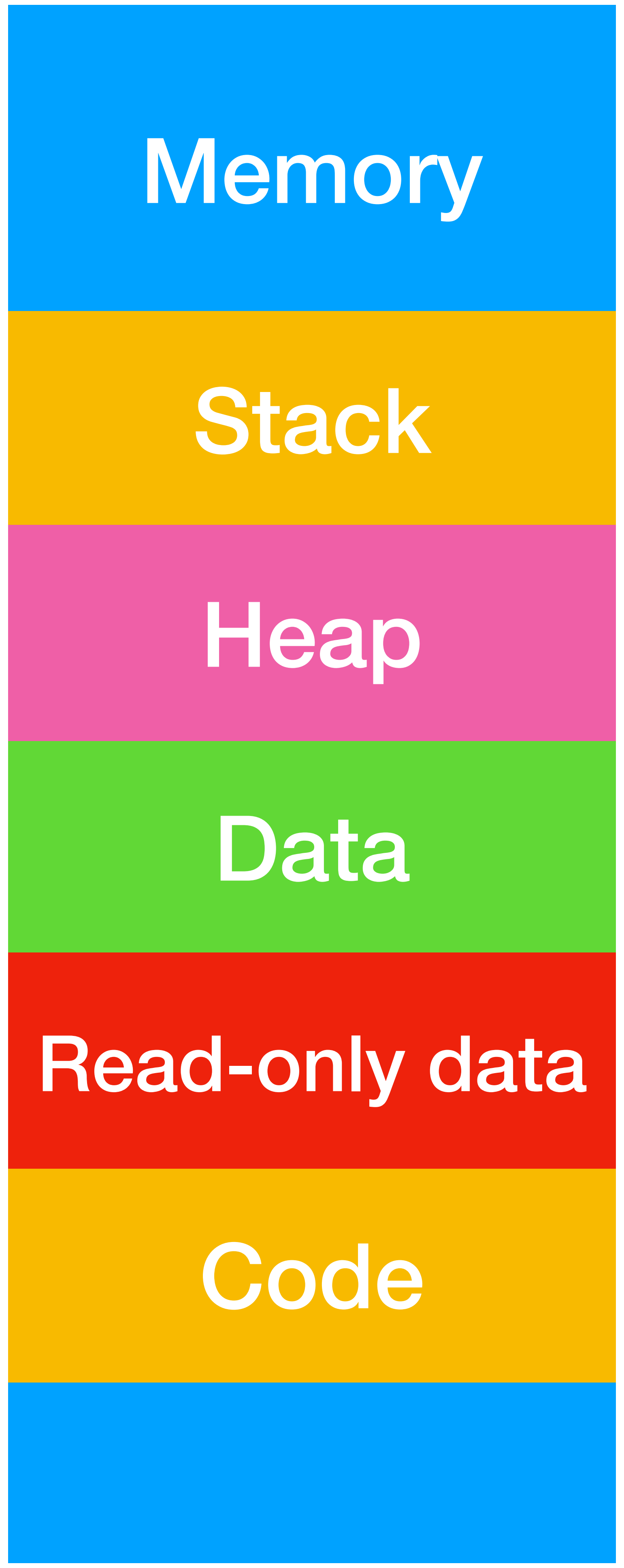
Content	**0x CD**	**0x AB**	**0x 34**	**0x 12**	......
Address	0x0006 0000	0x0006 0001	0x0006 0002	0x0006 0003	......

Program = variables + code

- Variables can be in
  - the read-only data section
  - the data section
  - the stack section
  - the heap section
- Machine code is in the code section

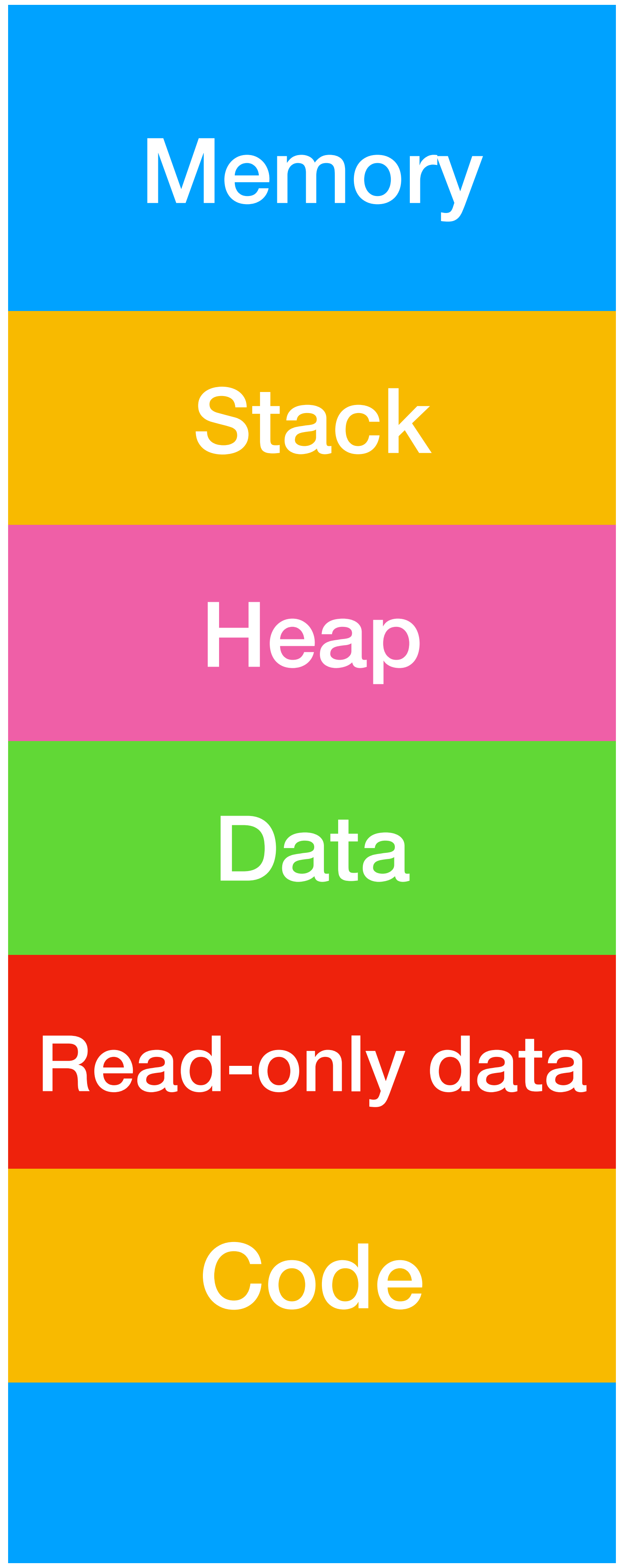
Quiz: variables in sections

```
int str_len = 14;

int main() {
        char* str = malloc(str_len);
        memcpy(str, “Hello World!\n”, str_len);
        printf(“%s”, str);
        return 0;
}
```



Answer: variables in sections

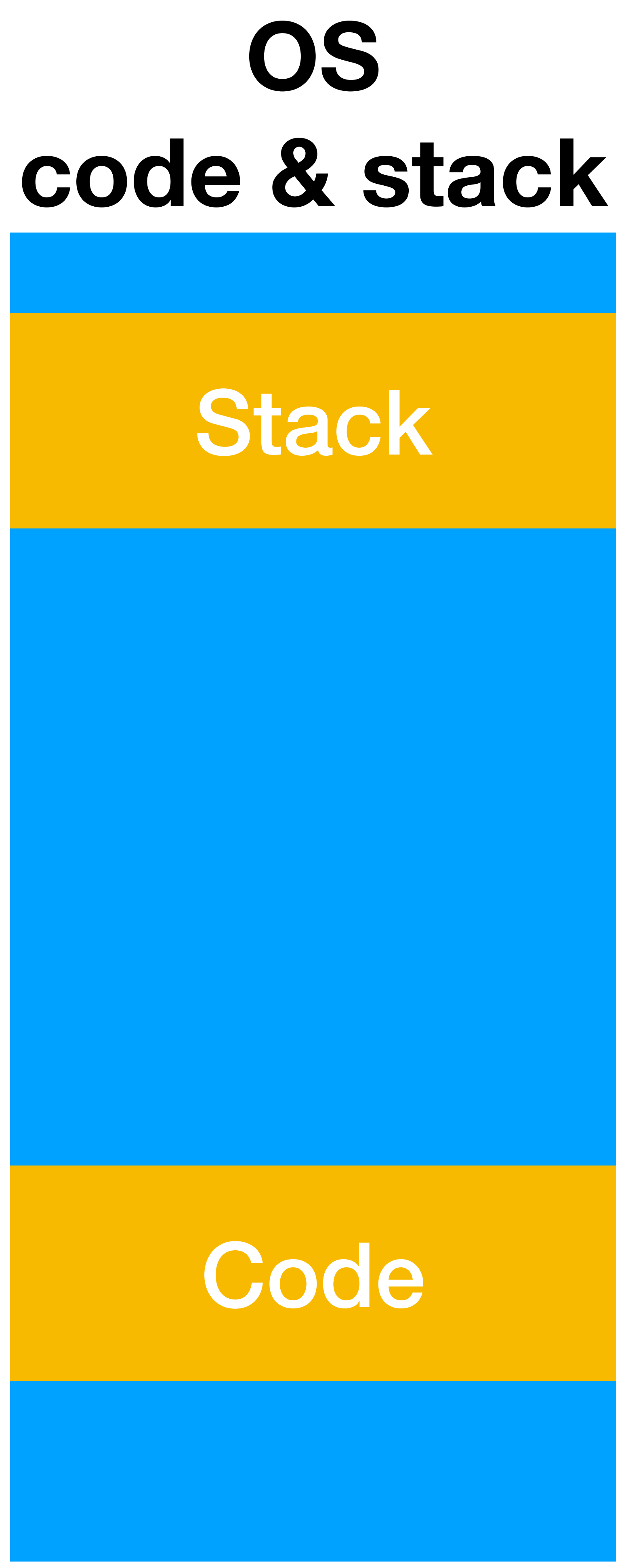
```
int str_len = 14;

int main() {
        char* str = malloc(str_len);
        memcpy(str, “Hello World!\n”, str_len);
        printf(“%s”, str);
        return 0;
}
```



Take-away #1: For each program, only code and stack are mandatory.

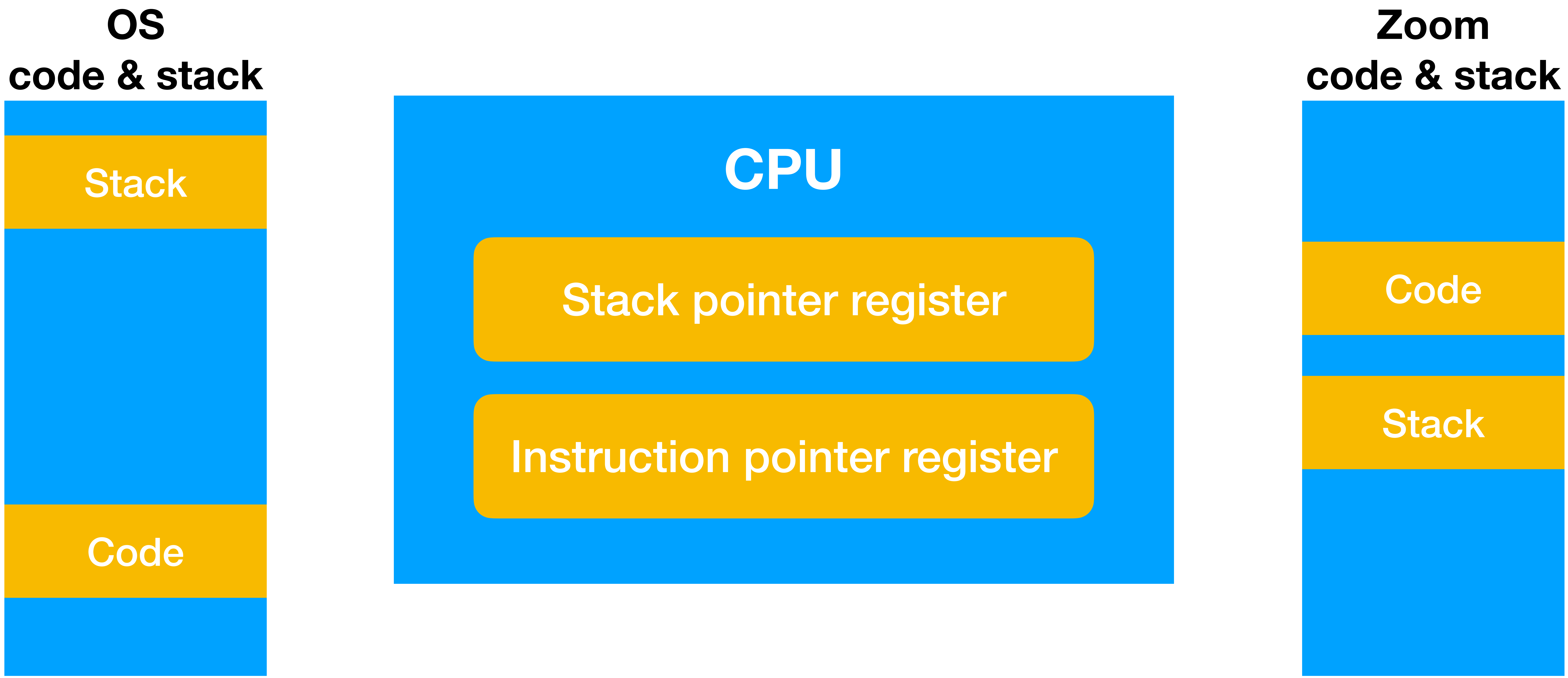
Heap | Data | Read-only data are optional.

OS is a program

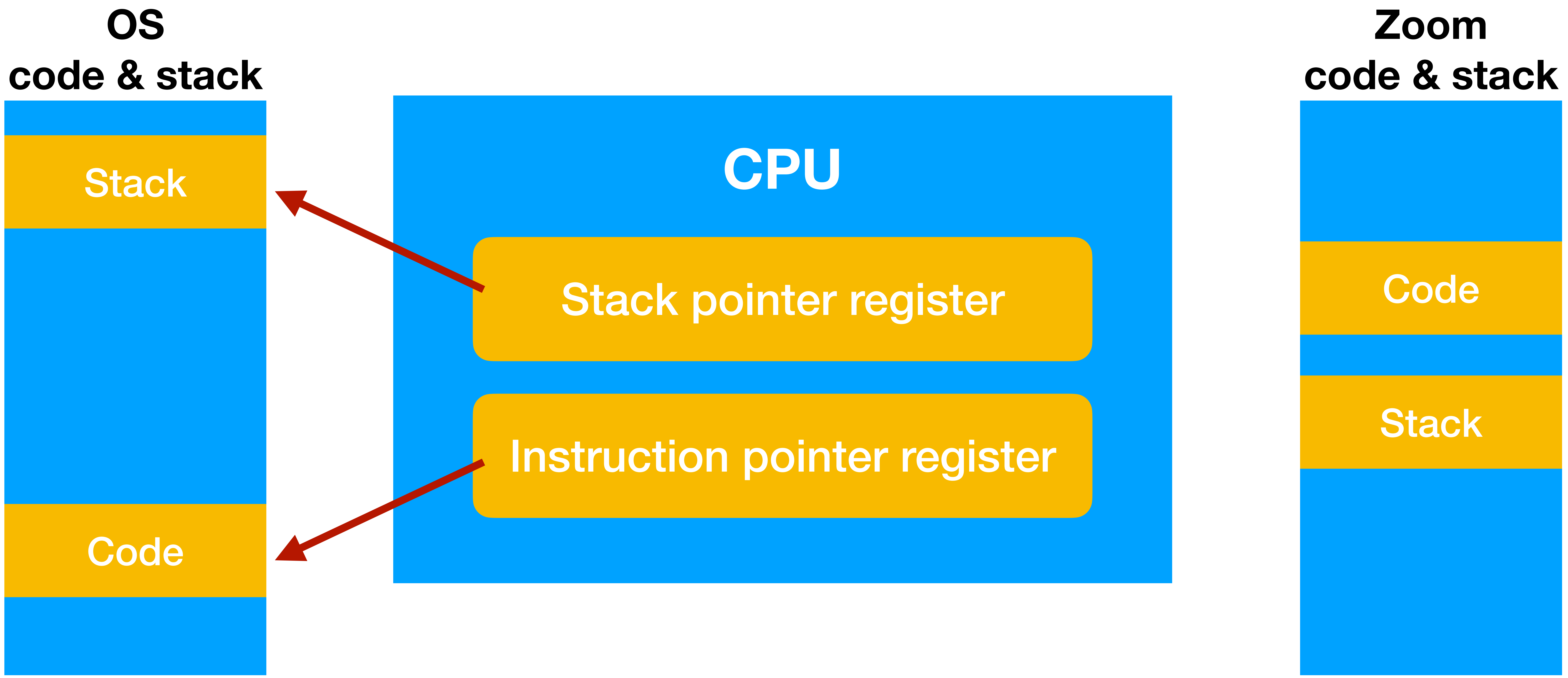


Zoom is another program

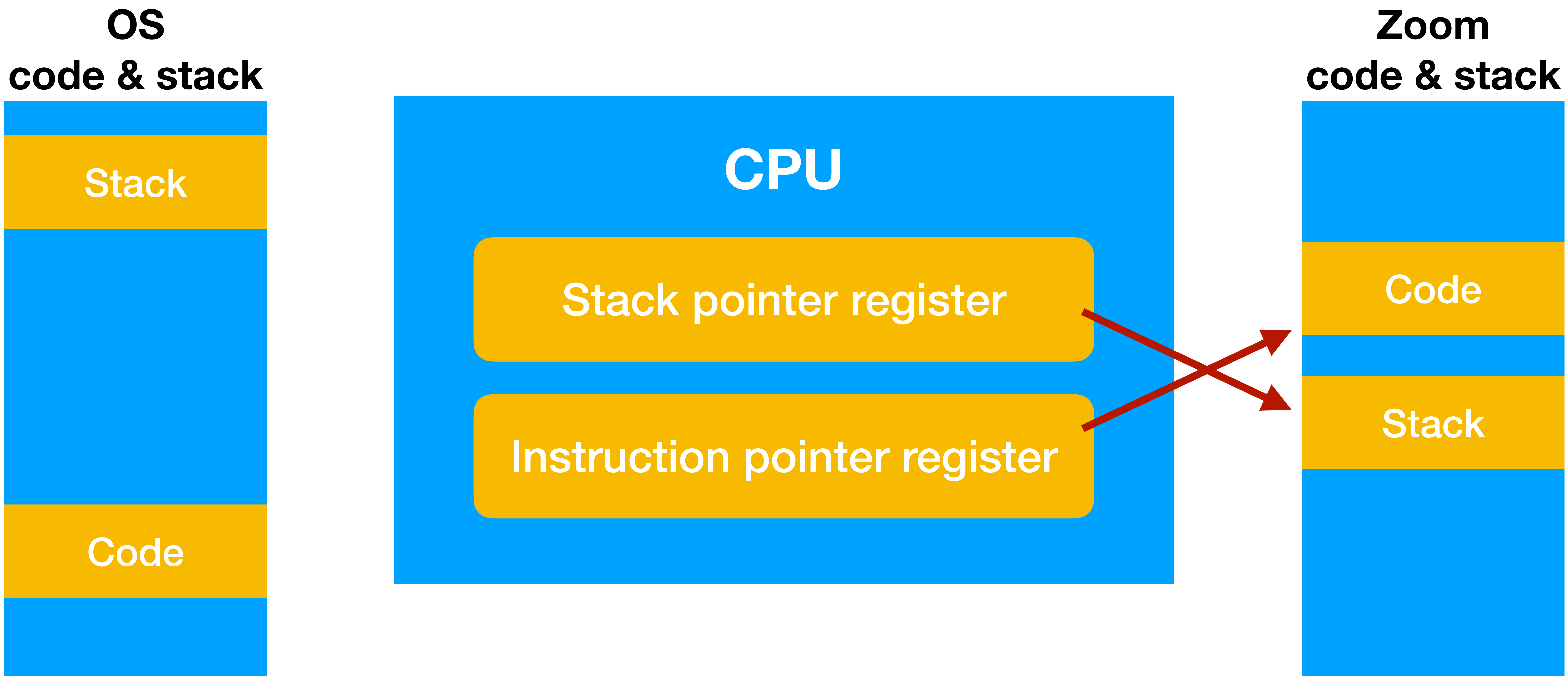
Add CPU into the picture



CPU in the context of OS



CPU in the context of Zoom



Take-away #2

CPU context = memory abstraction + stack pointer + instruction pointer

Different CPUs can be in the context of different programs.

A brief summary

- Physical memory is a piece of hardware.
- Memory abstraction is a simple math model.
- A C variable is a range in the memory abstraction.
- C variables can be in stack/heap/data/rodata sections.
- Program = variables + code; And code + stack is a must.
- CPU context = memory abstraction + stack / instruction pointers

Next: Types in C language

A variables has a type.

```
int val = 0x19950128;
```

A variable is a few bytes in memory.

Types tell the size of variables

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	N/A	void*	4	8

```
char c = ‘a’;                // type is char; sizeof(c) == 1
char* c_ptr = &c;            // type is char*
                             // sizeof(c_ptr) == 4 or 8
*c_ptr = ‘d’;                // ‘d’ has type char
                             // *c_ptr has type char
assert(c == *c_ptr && c == ’d’);

char** c_ptr_ptr = &c_ptr;   // type of &c_ptr is char**
                             // sizeof(c_ptr_ptr) == 4 or 8
assert(**c_ptr_ptr == ‘d’);
char another_c = ‘g’;
*c_ptr_ptr = &another_c;     // same as c_ptr == &another_c
                             // &another_c has type char*
                             // *c_ptr_ptr has type char*
                             // **c_ptr_ptr has type char
assert(**c_ptr_ptr == ‘g’);
```

Why pointer of pointer?

```
void g() {
    int a;              // a is 4 bytes on the stack of g()
                        // and a is not initialized

    f(&a);              // pass the address of a to f()
    assert(a == 123);
}

void f(int* ptr) {
    *ptr = 123;         // modify 4 bytes on the stack of g()
}
```

Similar for the dequeue() in P0

```
void test() {
      void* item;    // item is 4 or 8 bytes on the
                     // stack of test()

      queue_dequeue(…, &item);
}

int queue_dequeue(queue_t, queue, void** pitem) {
      *pitem = …;   // modify 4 or 8 bytes on the
                    // stack of test()
}
```

Why void*?

- A generic queue should store any type of variable.
- In C, any pointer type can cast to a void*.
- For example,

```
int a = 0x19950128;              // a 4-byte variable
void* ptr = (void*) &a;          // cast int* to void*
char* c_ptr = (char*) ptr;       // cast void* to char*
assert(*c_ptr == 0x28);          // dereferencing c_ptr will
                                 // read 1 byte instead of 4
```

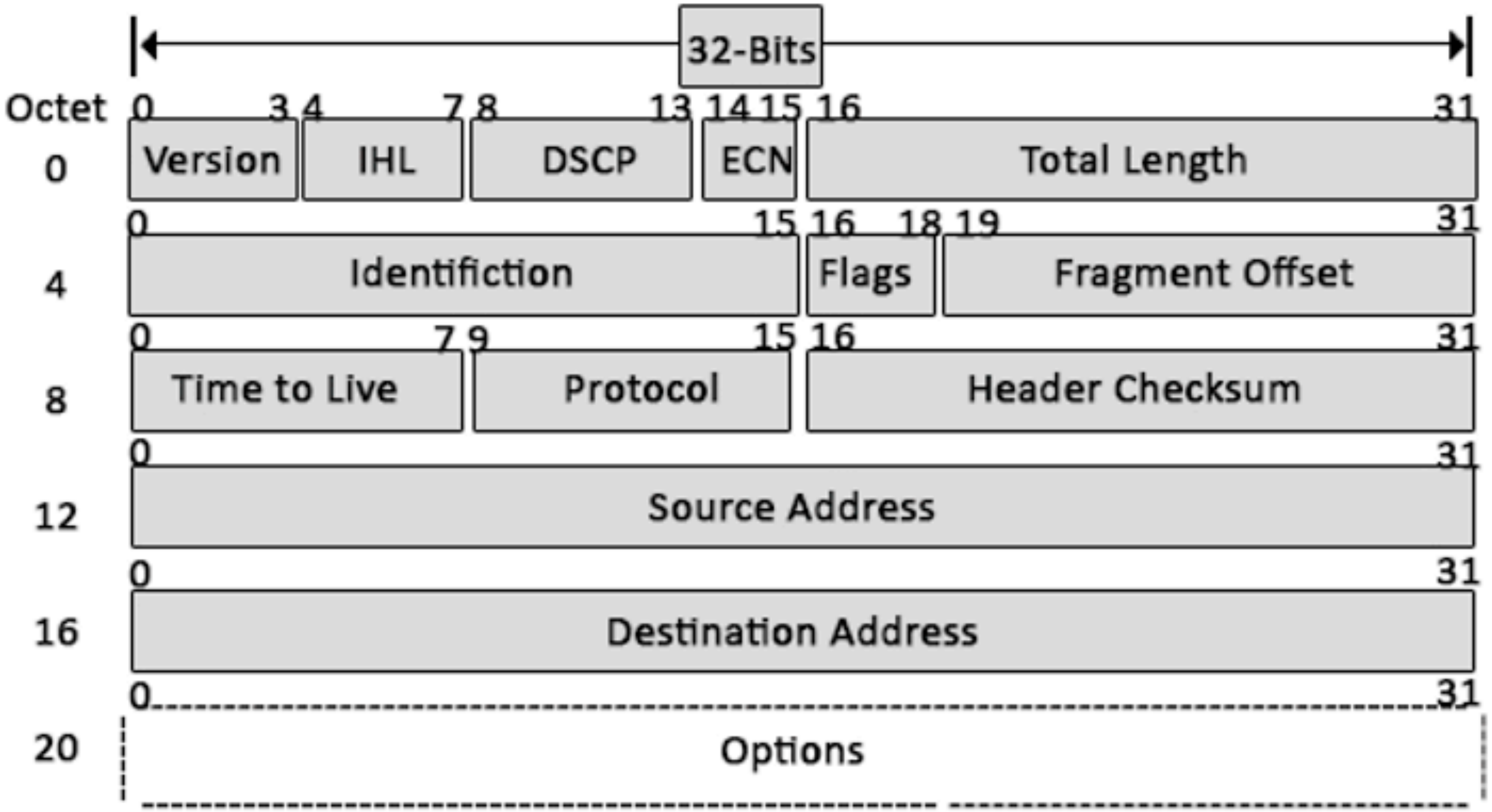
Some other C features

Typedef

```
typedef unsigned int   uint32_t;
uint32_t val;
// same as ‘unsigned int val’, but uint32_t is cleaner

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

Struct

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




```
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;
};
```

Why `malloc()` ?

```
// Consider a function receiving network packets
// The payload size is unknown

char* recv_packet() {
    struct header header;
    net_recv_header(&header);
    char* payload = malloc(header.payload_size);
    net_recv_payload(payload);

    return payload;
}
```

Why free() ?

```
void process_packet () {
    char* payload = recv_packet();

    …… // process the payload

    free(payload);
    // Otherwise, there is a memory leak which
    // can waste tons of memory in long-running programs
}
```

Summary

- Memory abstraction is a simple math model.
- C variables represent regions in the memory abstraction.
- CPU context = memory abstraction + stack / instruction pointers.
- Types specify the variables' size in order to reduce bugs in programs.
- Reference of a variable, &var, returns an address as a pointer variable.
- Dereference of a pointer variable, *ptr, returns the variable at address ptr.
- Other useful C language features: `typedef`, `struct`, `malloc()` and `free()`