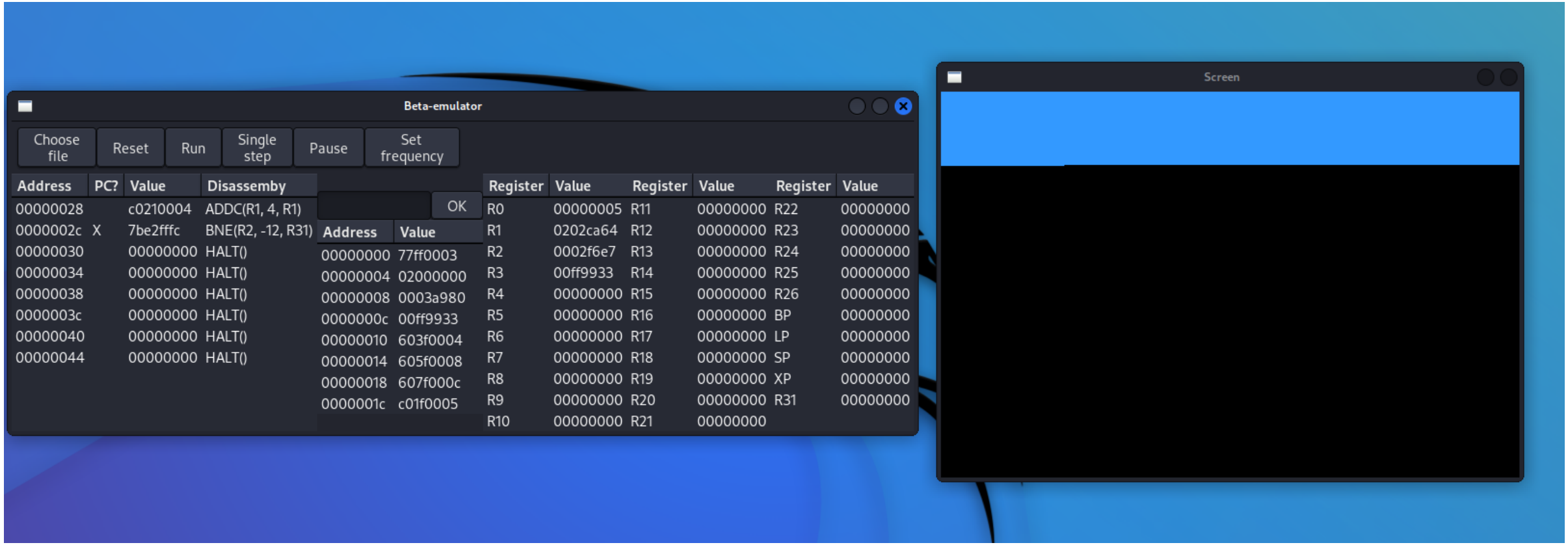
Operating systems
Project2
Emulating the beta-machine

Vincent Rossetto, Gaulthier Gain, Pr. Laurent Mathy

Remember? The beta-machine

- Minimal toy computer with a restricted instruction set
  - Used to teach computer architecture in the Computation Structures course
- Can be programmed using beta-assembly
- In this project, you will build an emulator for the beta-machine.

Objective: make this work (Demo)



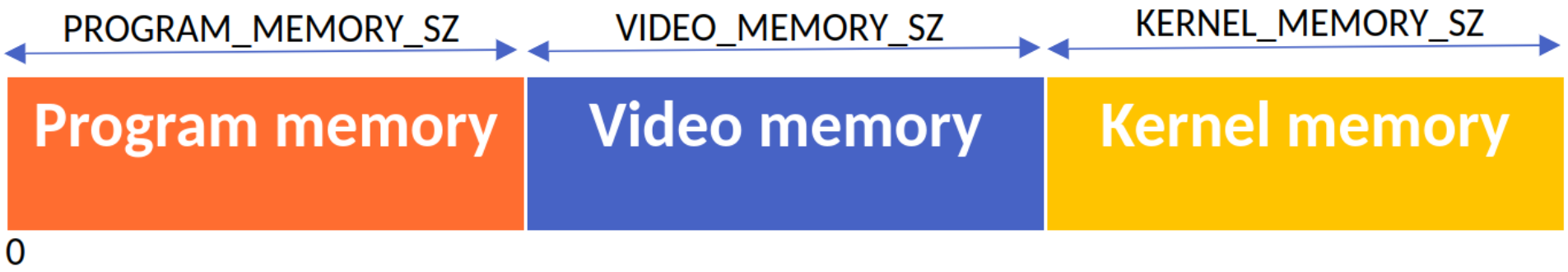
Concrete tasks

- The graphical interface is provided to you for free: *graphics.c*
- You are provided with an interface defining the functions to manipulate the emulated computer and exchange information with it in: *emulator.h*
- You will implement these functions in emulator.c
  - You can add other source files if you think it is best
  - You may add fields to the *CPU* and *Computer* structures
- You will implement an interrupt handler in beta-assembly (precisions on this later)

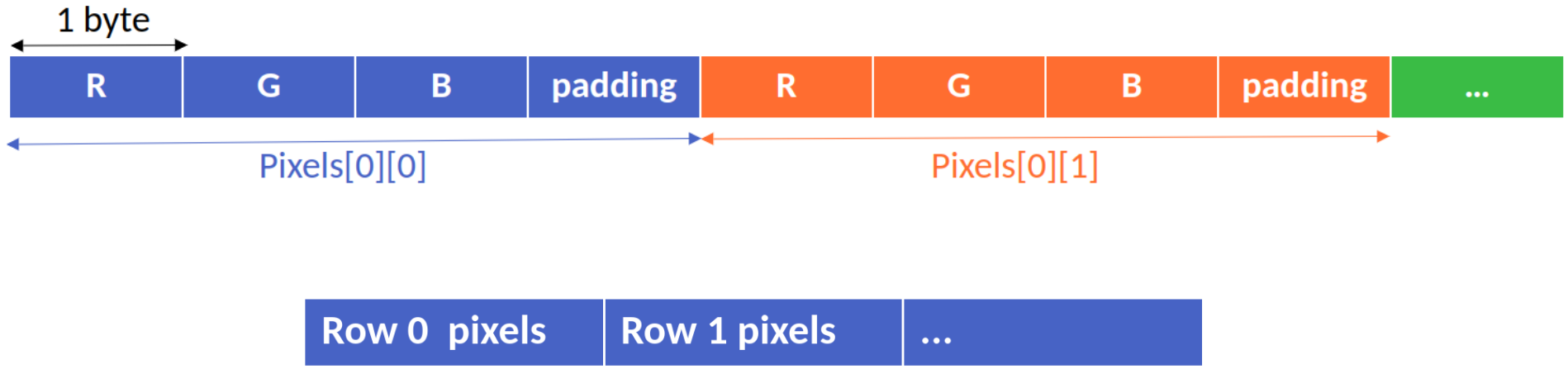
Organization of the computer's memory

- No virtual memory
- A single user-space program at the beginning of memory
- Two types of (keyboard) interrupts
  - Key pressed, key released

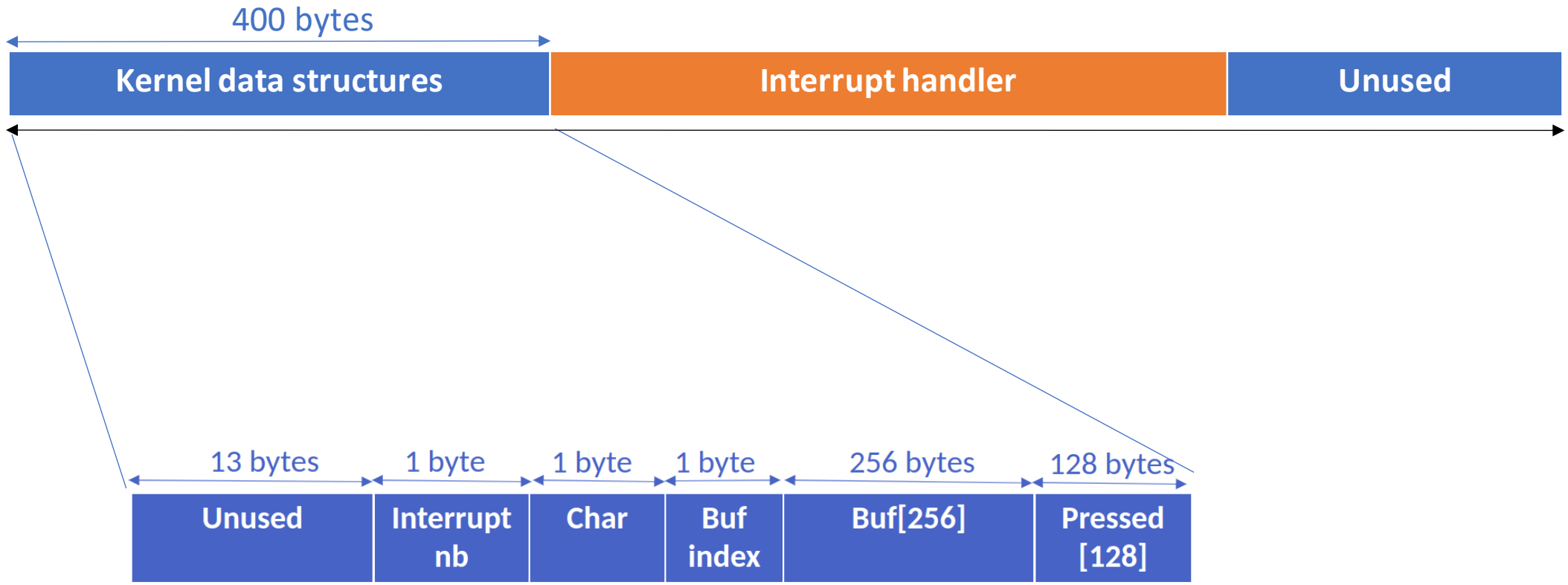
Memory - General organization



Memory - Video memory



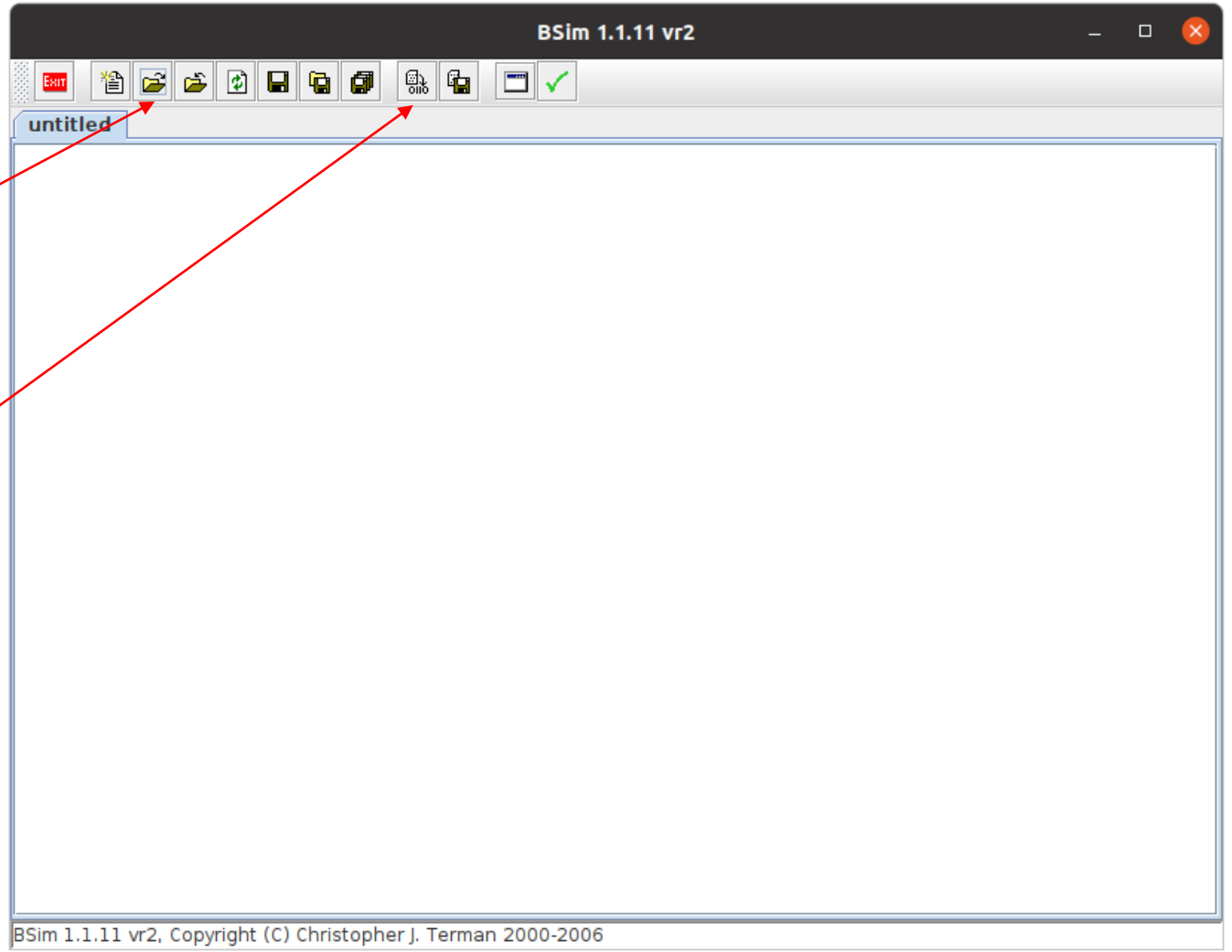
Memory - Kernel memory



Important remarks

- The inputs to your program are binary files, not assembly!
- Beta-assembly can be assembled into a binary using BSim
- The interrupt handler should work regardless of the sizes of the video memory, the user program memory and itself.

Create a binary with BSim



1) Select your assembly file

2) Build the associated binary

Use the provided BSim 1.1.11 vr2 and not BSim-export !

The BSim assembler

- All instructions translate directly into their 32-bit equivalent
- The *beta.uasm* file defines some useful macros which translate into one or several instructions
  - If you use those, they will be first replaced by their definitions, then the corresponding binary instructions will be replaced into their 32-bit representations
- Constants can be inserted into the binary
  - using LONG(value) to define a single 32-bit word of the given value
  - using STORAGE(size) to *insert* size words = 0
  - This is typically used for stack management but it can also be used to store large constants which cannot be represented inside of the instruction itself.
- We provide small examples to help you.

Emulator data structures

```
typedef struct{

    long program_counter;

    // add you own fields here !

} CPU;
```

```
typedef struct{

    CPU cpu;

    long memory_size; // = program_memory_size + video_memory_size +
    kernel_memory_size
    long program_memory_size;
    long video_memory_size;
    long kernel_memory_size;
    long latest_accessed; // address of the word most recently loaded/
    stored from/into memory
    bool halted; // was the HALT() instruction executed (stopping the
    program's execution)
    unsigned program_size; // user-space program size (code and stack)

    // add your own fields here!
} Computer;
```

Emulator interface (1/3)

```
/* Initializes the computer data structure, must be run before any
other function manipulating the computer */
void init_computer(Computer* c, long program_memory_size,
                                long video_memory_size,
                                long kernel_memory_size);

/*  Reads a 32-bit word at the address $addr from the computer's
    memory.
    Return value: the word found at addr. If addr > c ->
    memory_size, 0 will be returned. If addr is a valid address
    found at the boundary of the computer's memory (i.e. there is
    less than a full 4-byte word to return, then the function
    will return the valid bytes followed by a padding of 0-bytes. */
int get_word(Computer* c, long addr);

/* Returns the value of a given register of computer c, reg is
the register's number between 0 and 31. */
int get_register(Computer* c, int reg);

/* Frees all resources allocated for the computer data structure. */
void free_computer(Computer* c);
```

Emulator interface (2/3)

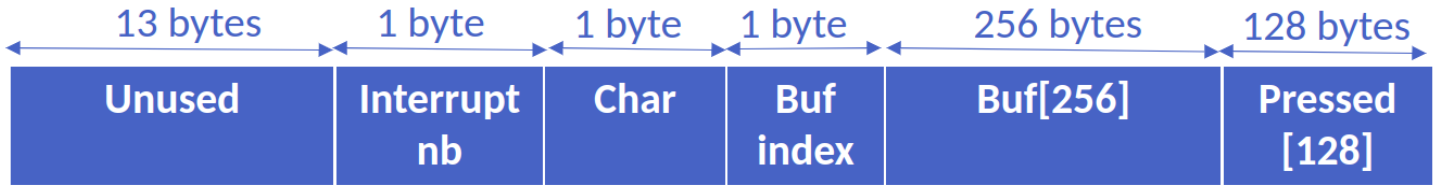
```
1  /* Loads the binary at the beginning of the computer's memory,
2     c -> program_size becomes the size of the binary in bytes. */
3  void load(Computer* c, FILE* binary);
4
5  /* Loads the interrupt handler binary in $c's kernel memory.
6     $binary can be NULL, in which case the function does nothing.
7     The $binary is placed after the kernel's data structures
8     (see statement for a diagram of kernel memory). */
9  void load_interrupt_handler(Computer* c, FILE* binary);
10
11 /* Runs one fetch + decode + execute cycle of $c's CPU,
12    If an interrupt line is raised (and the computer is not
13    already executing the interrupt handler), the program counter
14    becomes the start address of the interrupt handler
15    and its first instruction is executed.
16    Before handing control to the interrupt handler, the CPU
17    places the interrupt number and associated character at
18    the adequate place in kernel memory (see statement) and
19    stores PC into XP so that the interrupt handler is able to
20    return. */
21 void execute_step(Computer* c);
```

Emulator interface (3/3)

```
23 /* Raise an interrupt line of computer $c if no other already is.
24    Otherwise, this does nothing.  $type is the interrupt number
25    while $keyval is the associated character. */
26 void raise_interrupt(Computer* c, char type, char keyval);
27
28 /* Stores a textual representation of the disassembly of
29    $instruction in the buffer $buf. We assume that $buf
30    is large enough to store any disassembled instruction.
31    If $instruction is not a valid instruction (see slides
32    on the Beta-assembly instruction set), then the
33    "INVALID" string is stored instead.
34    Returns 0 if $instruction is valid, and a negative
35    value otherwise. */
36 int disassemble(int instruction, char* buf);
```

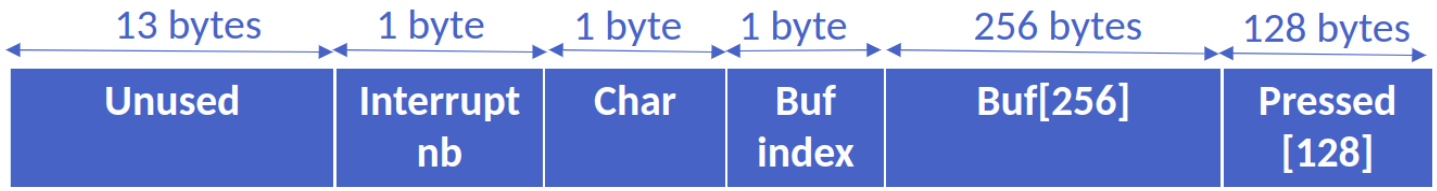
Interrupts

- You will write an interrupt handler in beta-assembly to deal with the following interrupts
- 2 types
  - Type = 0, Buf[index++] = Char; Pressed[Char] = 1



Interrupts

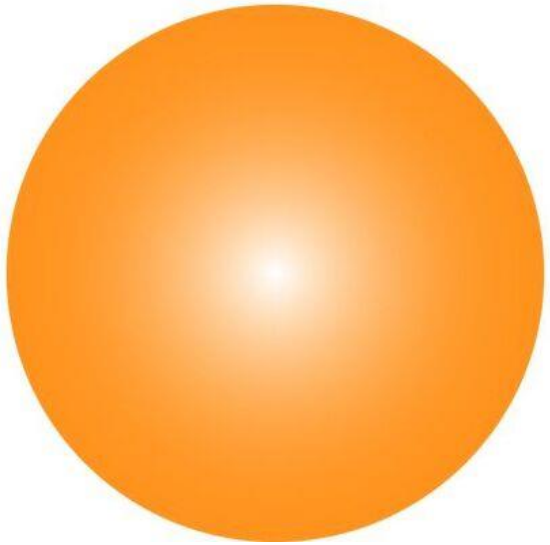
- You will write an interrupt handler in beta-assembly to deal with the following interrupts
- 2 types
  - Type = 1, Pressed[Char] = 0



- Do not use hardcoded addresses: program, video, and interrupt handler's memory sizes are user-defined and your interrupt handler should work with any.

Bonus (up to 2 points)

- Write an assembly user-space program to draw this on the emulated screen.



Requirements

- Same groups as for project 1
- Submit a *tar.gz* archive containing all your C source files (including header files such as your modified emulator.h), all your beta-assembly files as well as the corresponding beta-machine binaries
- **Deadline**: **24th May 2023 (23:59:59)**
- Late submissions are accepted but with a penalty of $2^N - 1$ marks per started day after the deadline
  - **Not** $2^{(N-1)}$!
  - Project 1 has been updated with the same rule.