How to make a (GameBoy) emulator?

Petar Veličković

University of Cambridge
Cambridge Coding Summer School

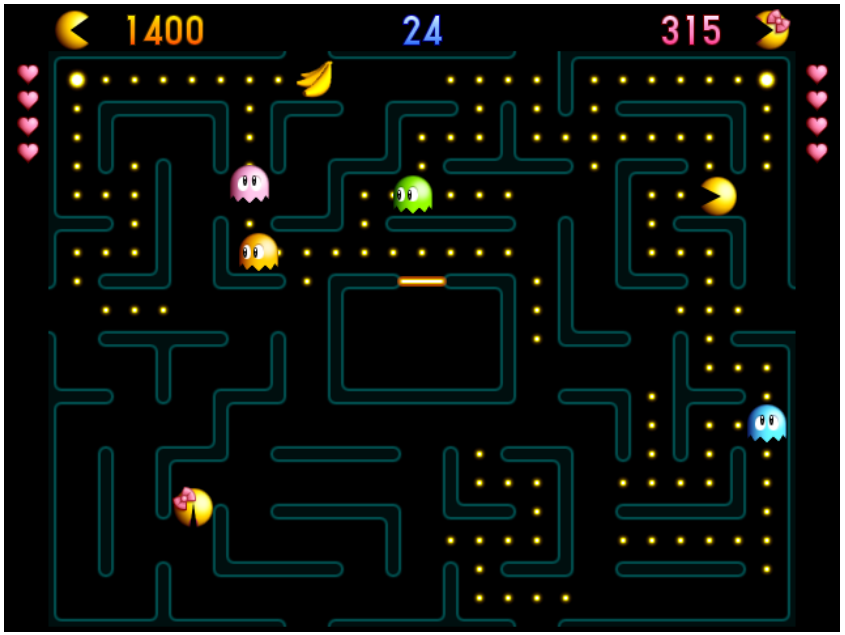
13 July 2016

Emulation

- **Emulation** is a computer science concept corresponding to *imitation of one computer system on another.*
- Contrast with **simulation**, which is just an imitation of a particular feature/functionality (e.g. of a video game).
- Emulation can be performed in both **hardware** and **software**; in this talk, we will focus only on software emulators.

Emulator vs simulator

PSX **emulator**



Pac-man **simulator**

An emulator executes the original code (ROM) written for the console, whereas the simulator represents an entirely new game.

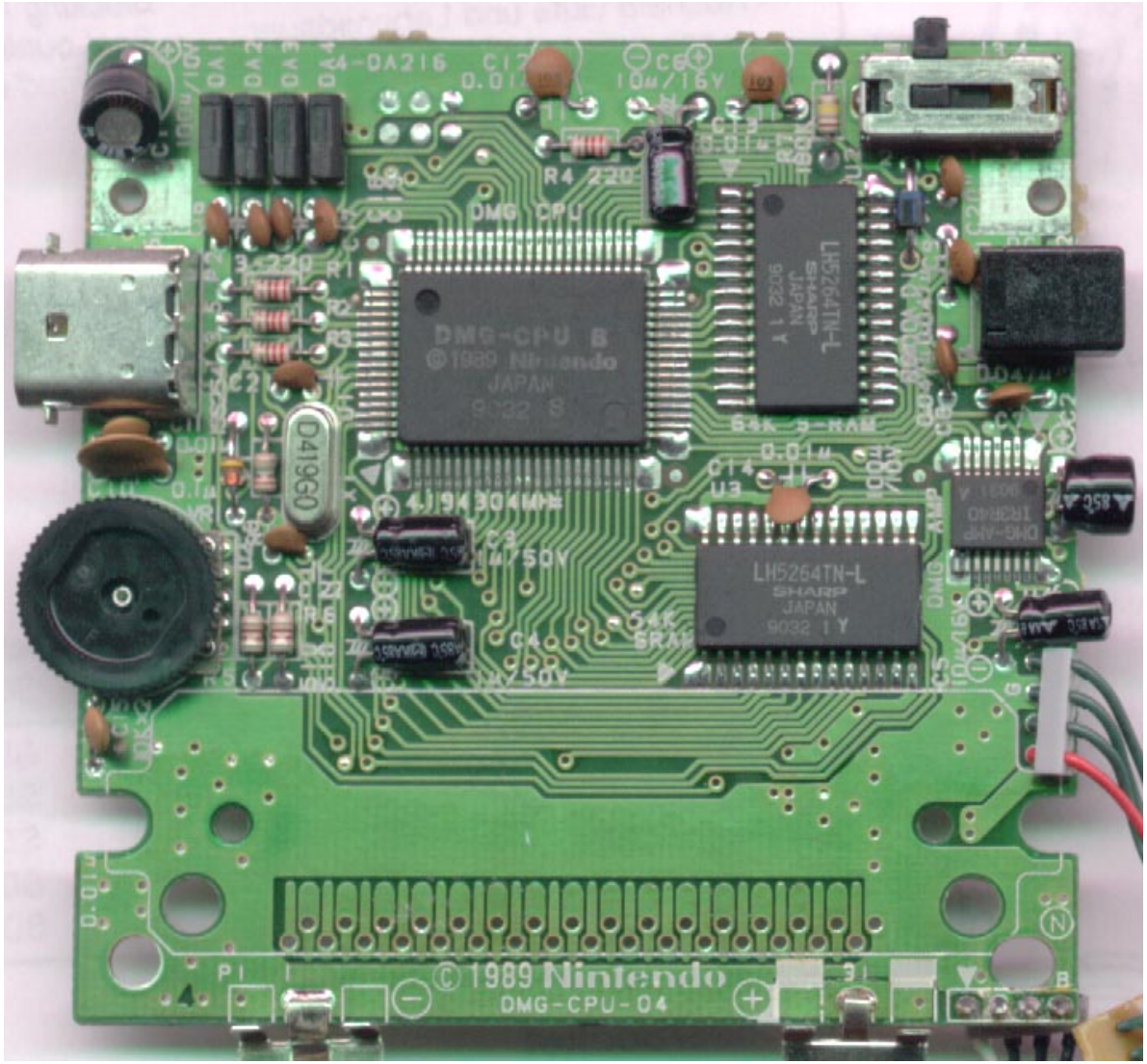
GameBoy

- The *GameBoy* is a video gaming console released by Nintendo in 1989.

- First notable *handheld* console, in spite of technologically superior competition:
  - Together with its successor (Game Boy Color), sold **over 119 million units** (by far the most popular handheld console prior to the release of the Nintendo DS).
  - The Tetris cartridge alone sold 35 million copies!

Console exterior

Under the hood



Today's task

Cartridge

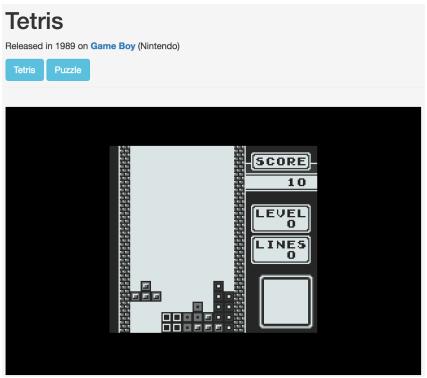
insert

GameBoy

1100 0011 0000 1100
0000 0010 0000 0000
0000 0000 0000 0000
0000 0010 1111 1111
1111 1111 1111 1111

ROM (Game code)

input



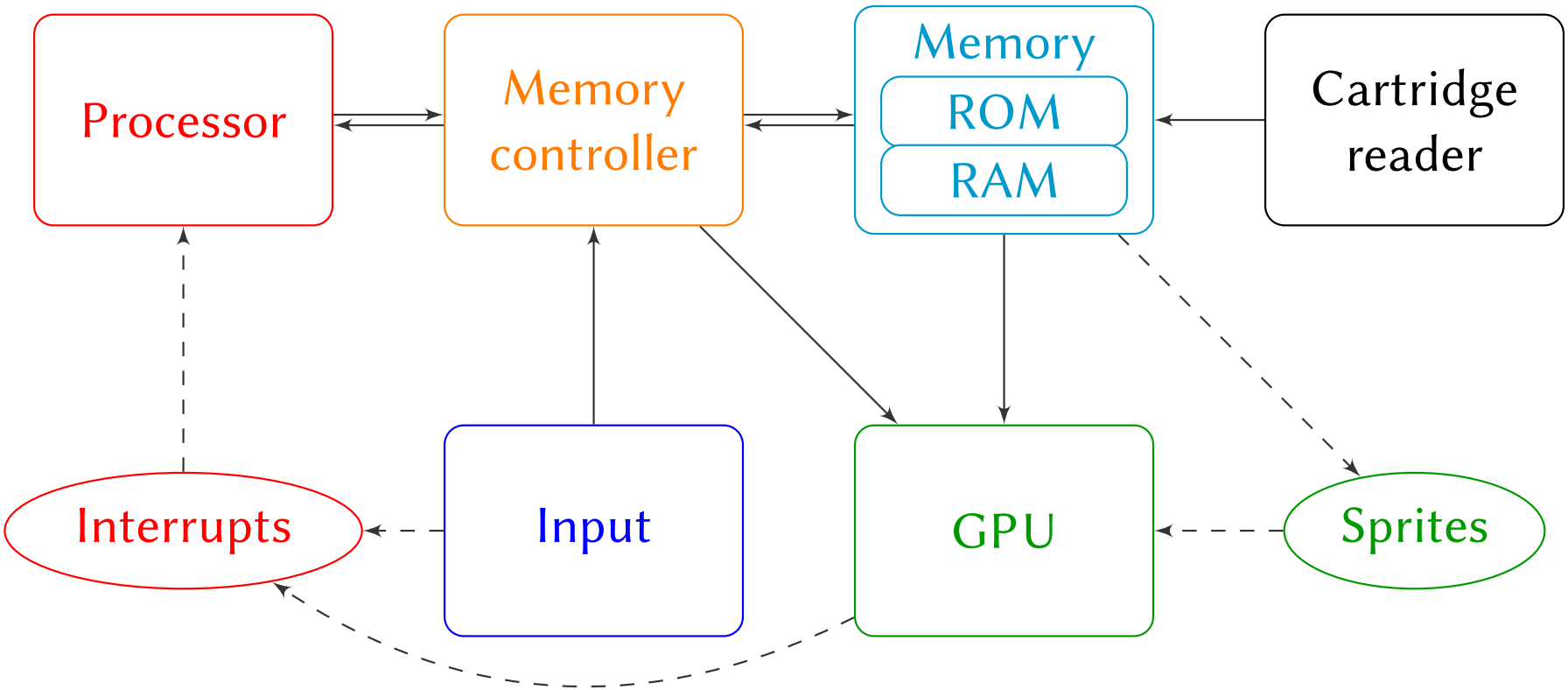
Emulator

Modular approach

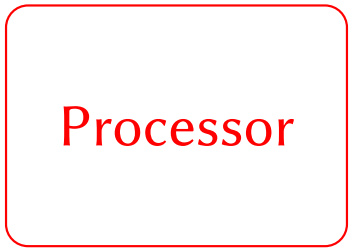
- We will adopt a *modular* approach in our implementation – the emulator will consist of (nearly) **independent** implementations of all of its subsystems ($\sim$ modules)!

- Why do this?
  - Debugging becomes much easier because the effects of programmer errors are local to a subsystem.
  - The modules may be reusable in other emulator projects – the hardware is *standardised*.
  - Amenable to *object-oriented programming*.

Overview of the emulator

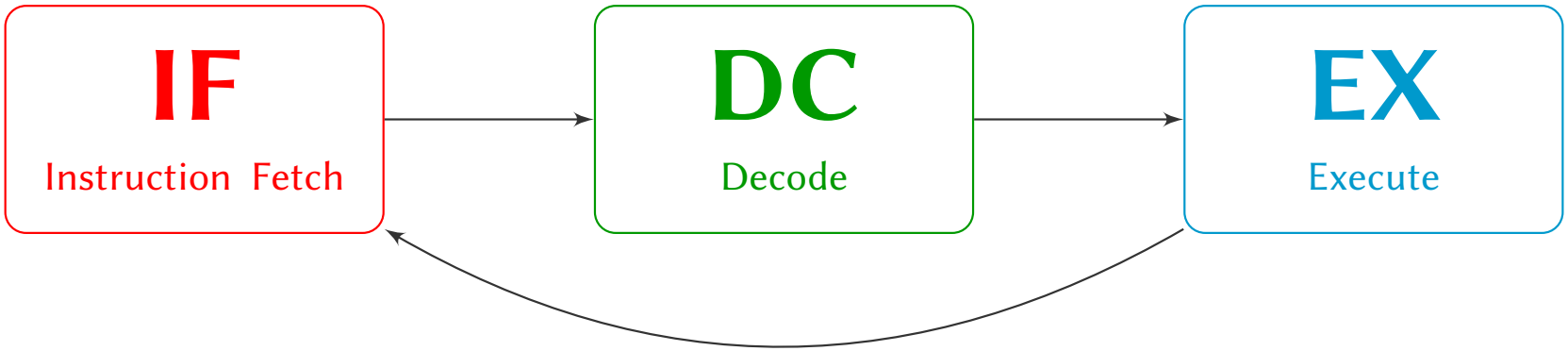


Processor



- Usually the most involved and complex module in the emulator implementation (which is why programmers usually tend to use pre-made CPU implementations).
- Several approaches are possible—we will focus on writing an **interpreter**, which "follows" the game code and simply simulates every instruction it encounters using the local CPU.

Interpreter emulator



- Interpreters usually simulate the *fetch-decode-execute* cycle, present in most contemporary processors:
  - **IF** *Instruction Fetch* – fetch the current instruction;
  - **DC** *Decode* – determine the type of the instruction;
  - **EX** *Execute* – execute the current instruction, updating the state of the processor, memory, etc.
- This way, we can only emulate systems whose processors are **several orders of magnitude** slower than the native one.

Instruction Fetch

IF

Instruction Fetch

- *Where* is our program???
- With GameBoy, and most contemporary systems, both the code and data are stored in the same memory ($\sim$ *von Neumann architecture*).
- $\Longrightarrow$ We need methods for reading from and writing to memory, if we'd like to be able to fetch the current instruction.

Memory interface

- The GameBoy does all of its computations on the level of a single byte; therefore, it makes sense to expose two methods:

```
ReadByte(address);
WriteByte(address, value);
```

  to the CPU.

- For now, the processor only needs to know that these functions exist—we will consider their actual implementations later.

Decode

- After fetching the current byte of the program, we need to determine the instruction that it represents (and hence whether we need to read additional parameters from memory).
- This requires us to be well acquainted with the **instruction set architecture** (ISA) of GameBoy's processor (which is very similar to the *Zilog Z80* ISA).

GameBoy arhitecture

- Since we have read a single byte of our program as an instruction, this allows us to consider $2^8 = 256$ distinct instructions.
- It is necessary to implement an enormous branching (`switch-case` is an appropriate construct to use), which will select the instruction to execute based on this byte's value.
- This is easily the most cumbersome part of the implementation...

Execute

EX

Execute

- Each instruction may affect the processor's internal state (∼ **registers**), as well as the state of memory.
- Furthermore, every instruction requires a certain (predetermined) number of CPU ticks/cycles (**will be very important later!**).
- The previously mentioned `WriteByte` method may be used for effects on memory.
- In order to implement all these instructions, we need to implement and maintain the processor's internal state as well.

Processor state

The CPU's current state is fully specified by the following registers:

`PC` *Program Counter* — the memory address of the next instruction to be fetched;

`SP` *Stack Pointer* — the memory address of the top of the stack (omitted from this talk);

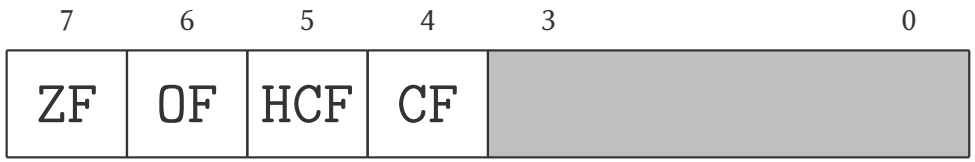
`F` *Flags*;

`A-E,H,L` General purpose registers.

- We will also maintain the number of elapsed ticks since the start of execution.

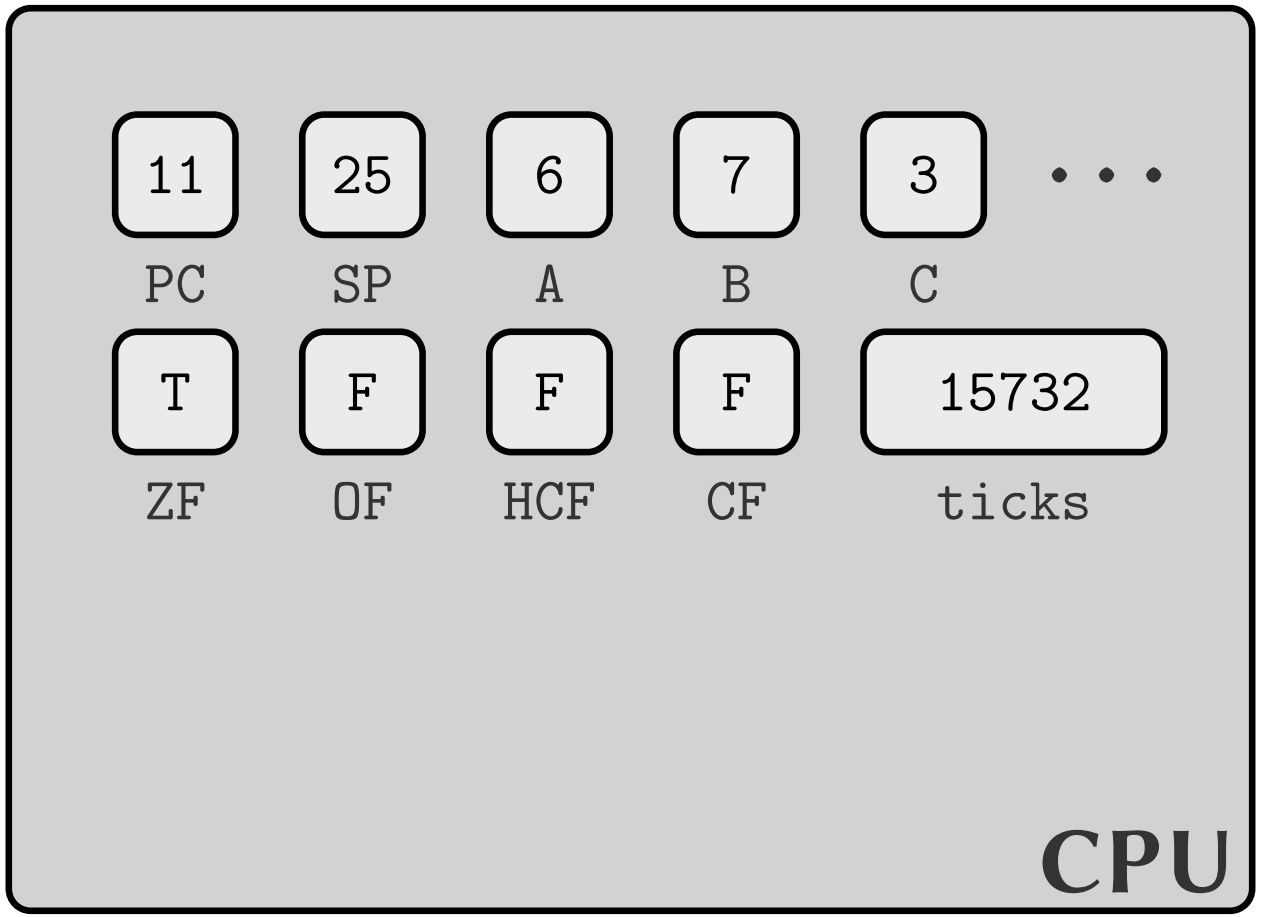
Flags

- Flags are logical (`true`/`false`) variables that allow the processor to determine the effects of the previously executed instruction.
- With GameBoy, the flags register looks like this:

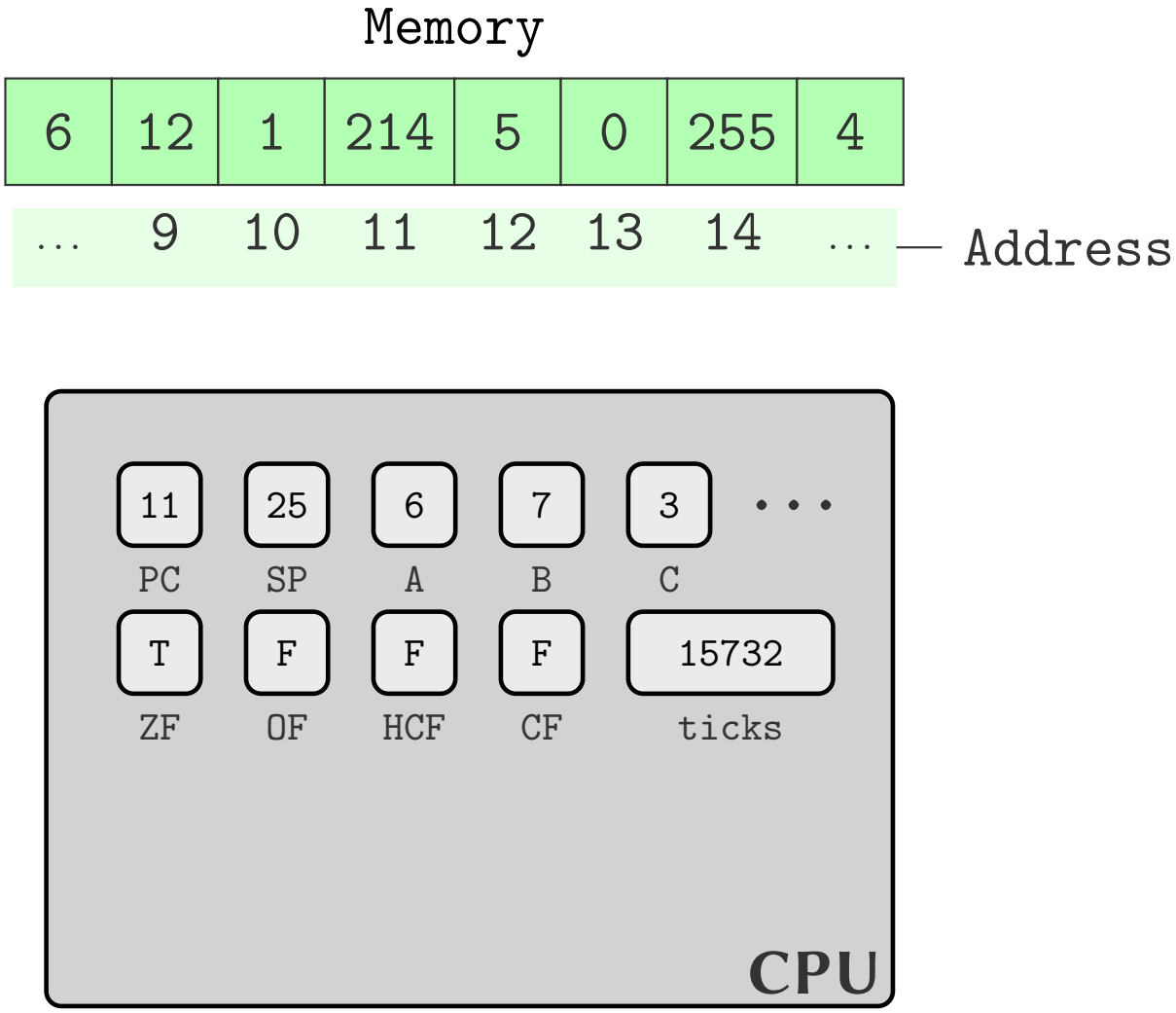


- The aforementioned flags are:
  - `ZF` *Zero Flag* — set if the result of the operation is zero;
  - `OF` *Operation Flag* — set if the operation was subtraction;
  - `HCF` *Half-Carry Flag* — set if there was an overflow in the lower half of the result (omitted);
  - `CF` *Carry Flag* — set if there was an overflow in the result.

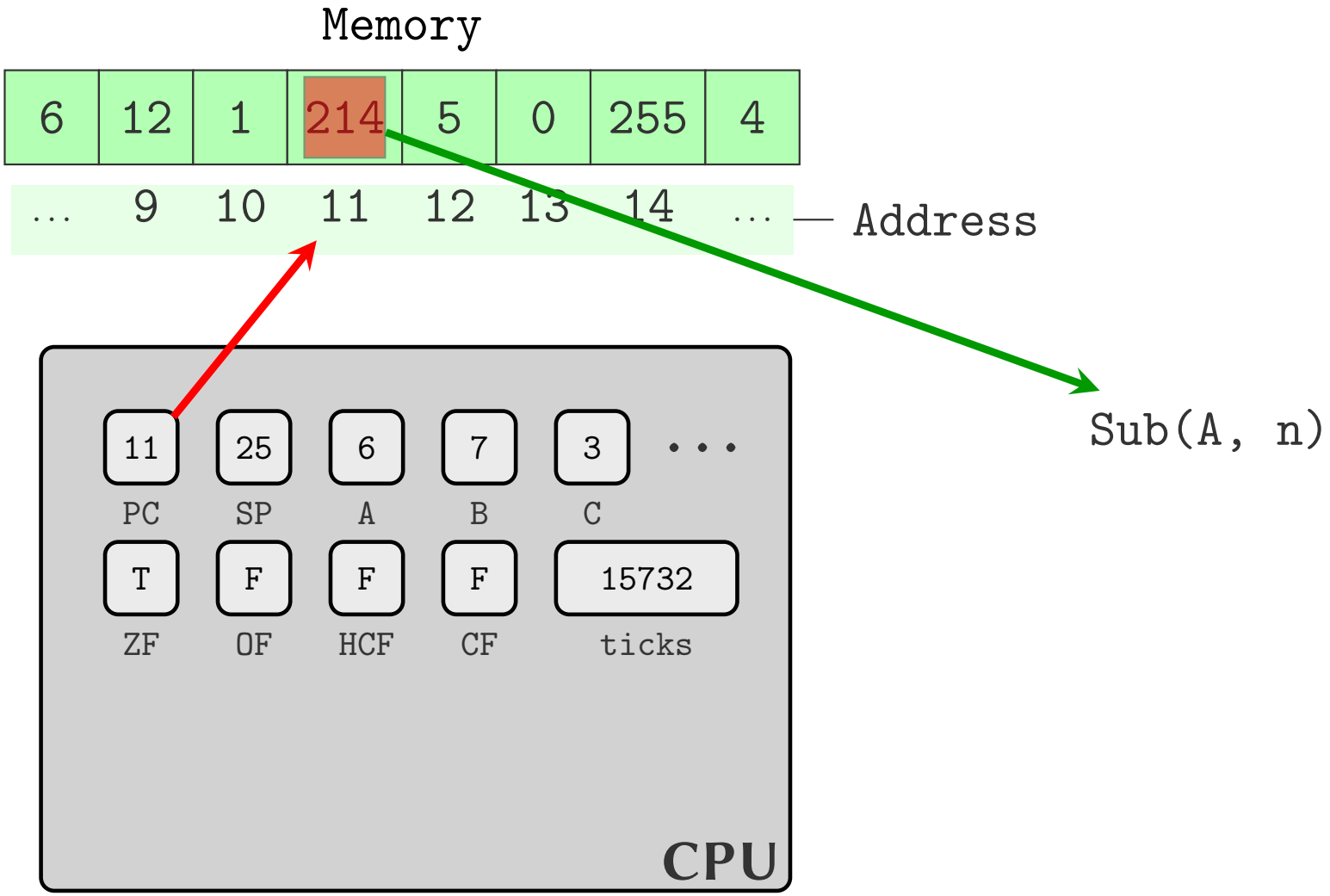
CPU state, *cont'd*



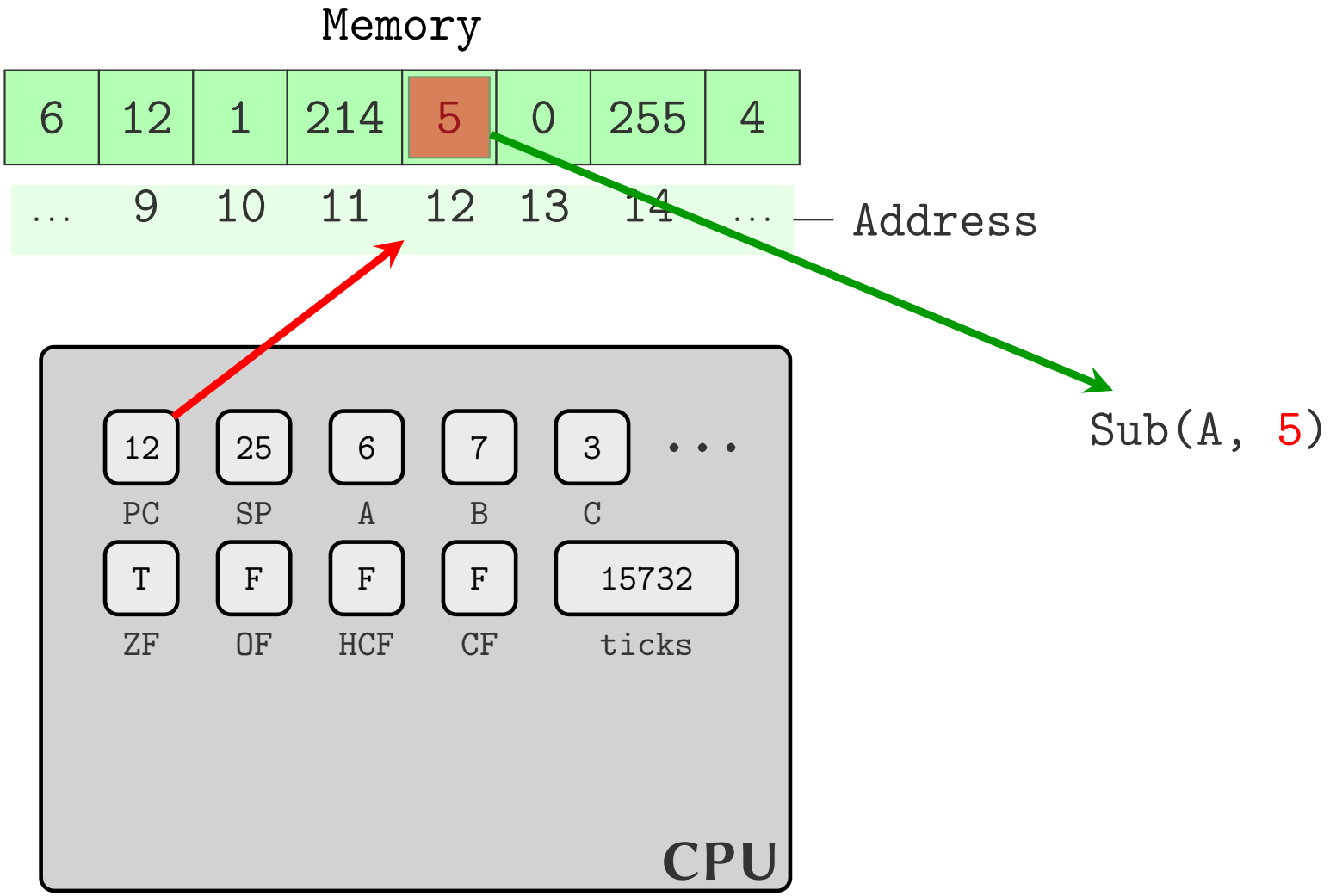
Example: Sub(A, 5)



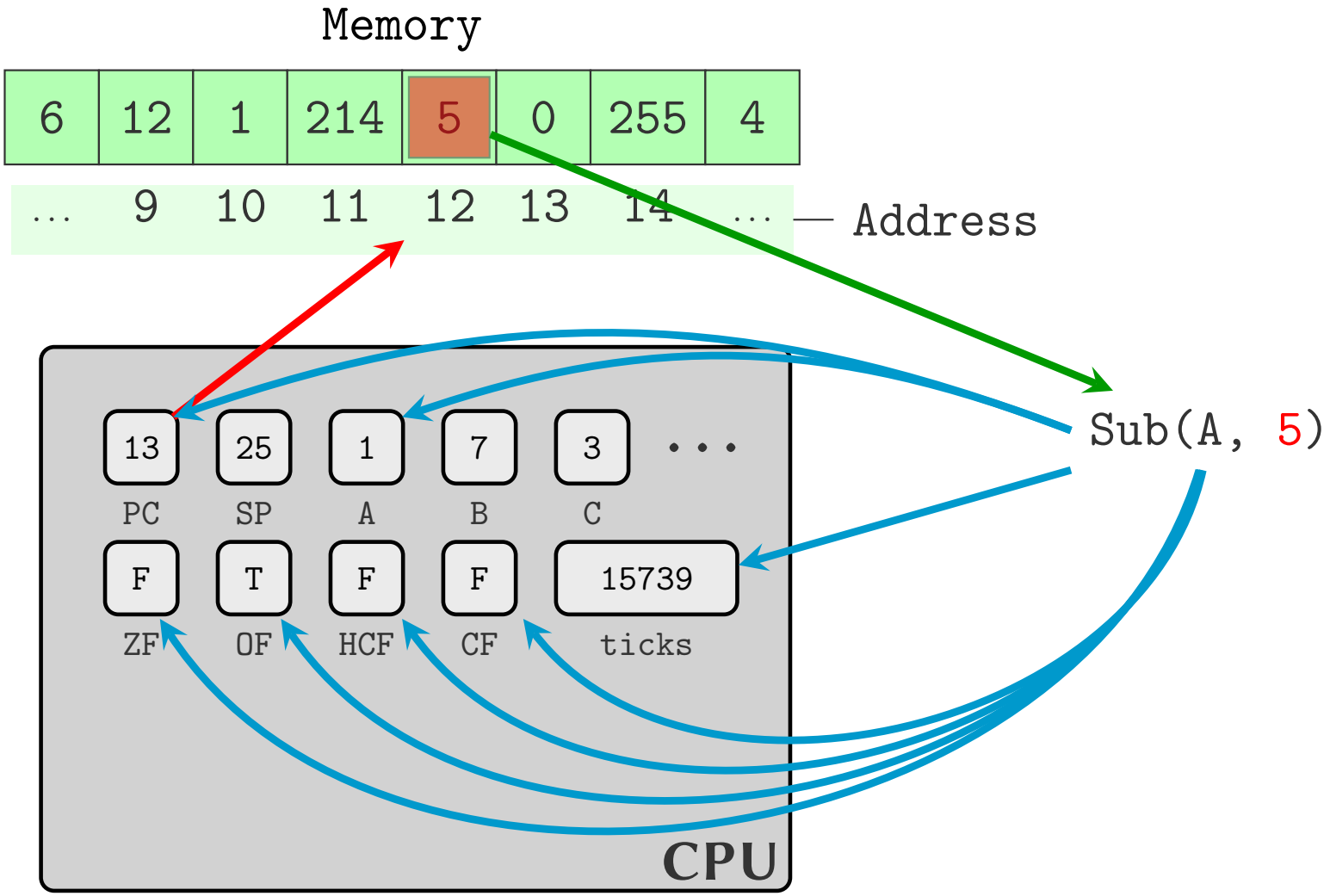
Example: Sub(A, 5)



Example: Sub(A, 5)



Example: Sub(A, 5)



Memory

- It is necessary to correctly implement the `ReadByte` and `WriteByte` methods: we need to emulate the **memory controller**, i.e. the memory management unit (*MMU*).

Naïve implementation

- PC has 16 bits, and therefore the GameBoy memory system supports $2^{16} = 65536$ addresses.
- The naïve implementation is very simple...

```
int memo[65536];
public int ReadByte(int address)
{
  return memo[address];
}
public void WriteByte(int address, int value)
{
  memo[address] = value;
}
```

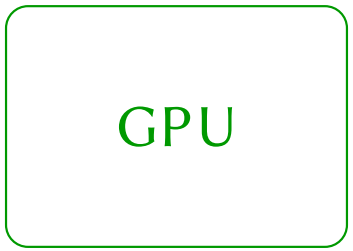
- Can you think of some reasons why this is not so easy?

Gotchas

- Generally, *logical* address $\neq$ *physical* address:
  - Not all writes should succeed (ROM…);
  - Not all addresses represent main memory (e.g. a special address allows the CPU to read keypad inputs…);
  - Two different addresses could be mapped to the same physical location.
  - …

- With GameBoy, **all of the above hold!**

GPU

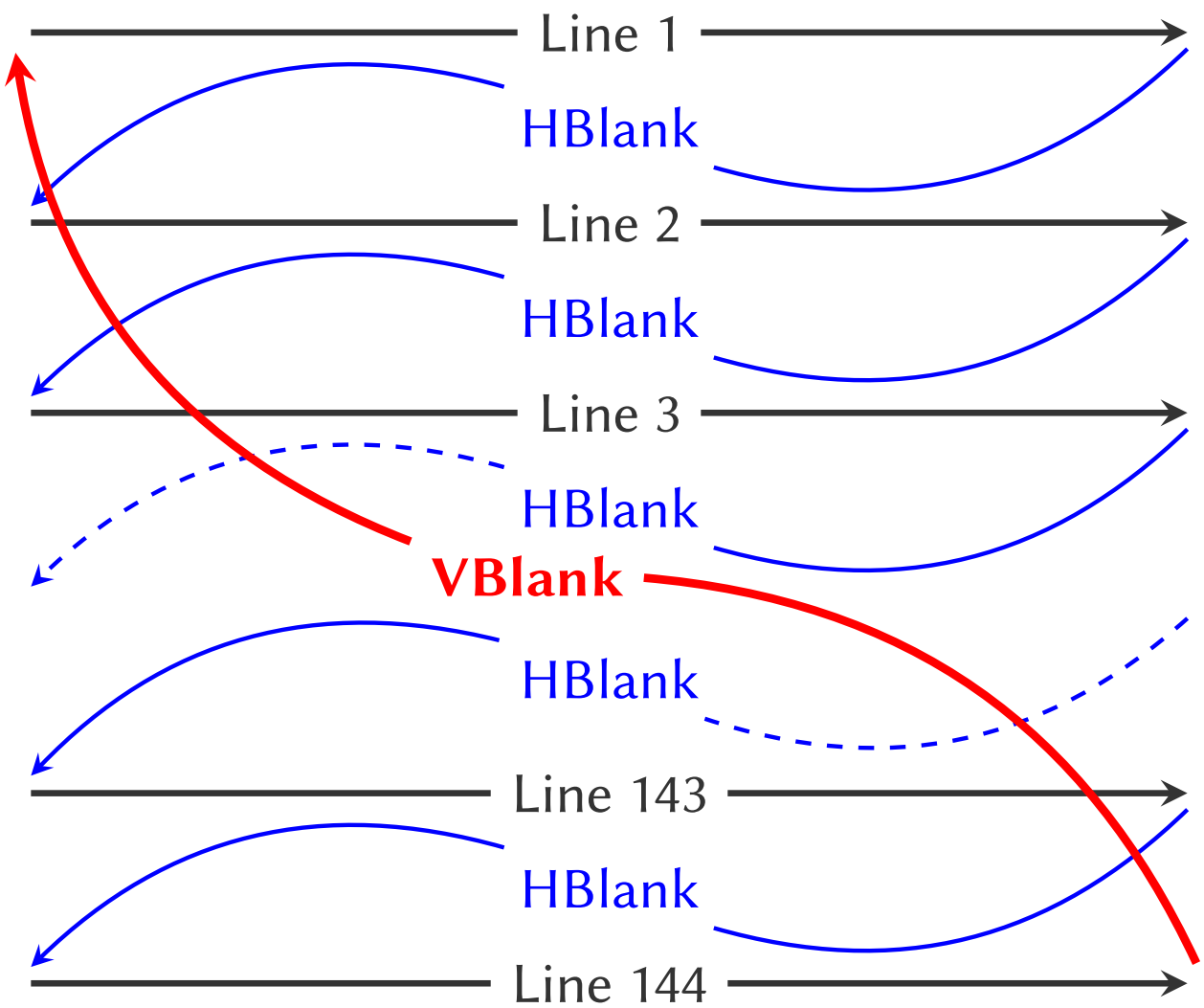


- So far, we have managed to design an emulator that can, step by step, execute any GameBoy program.
- Now we are interested in seeing the results of our work on a graphical output.

GameBoy graphics

- GameBoy's graphical output is an LCD with a 160×144 pixel resolution $\Longrightarrow$ we can maintain a colour matrix of such size, and display it in our emulator.
- The values in this matrix have the following format: `0xAARRGGBB` (Alpha–Red–Green–Blue)
- GameBoy supports four shades of gray, so the matrix may contain only these four values:
  - `0xFF000000` (BLACK)
  - `0xFF555555` (DARK_GRAY)
  - `0xFFAAAAAA` (LIGHT_GRAY)
  - `0xFFFFFFFF` (WHITE)

Rendering a single frame ($\sim$ *CRT*)



When do we draw a new frame?

- Every stage in the previous figure takes a specific amount of time:

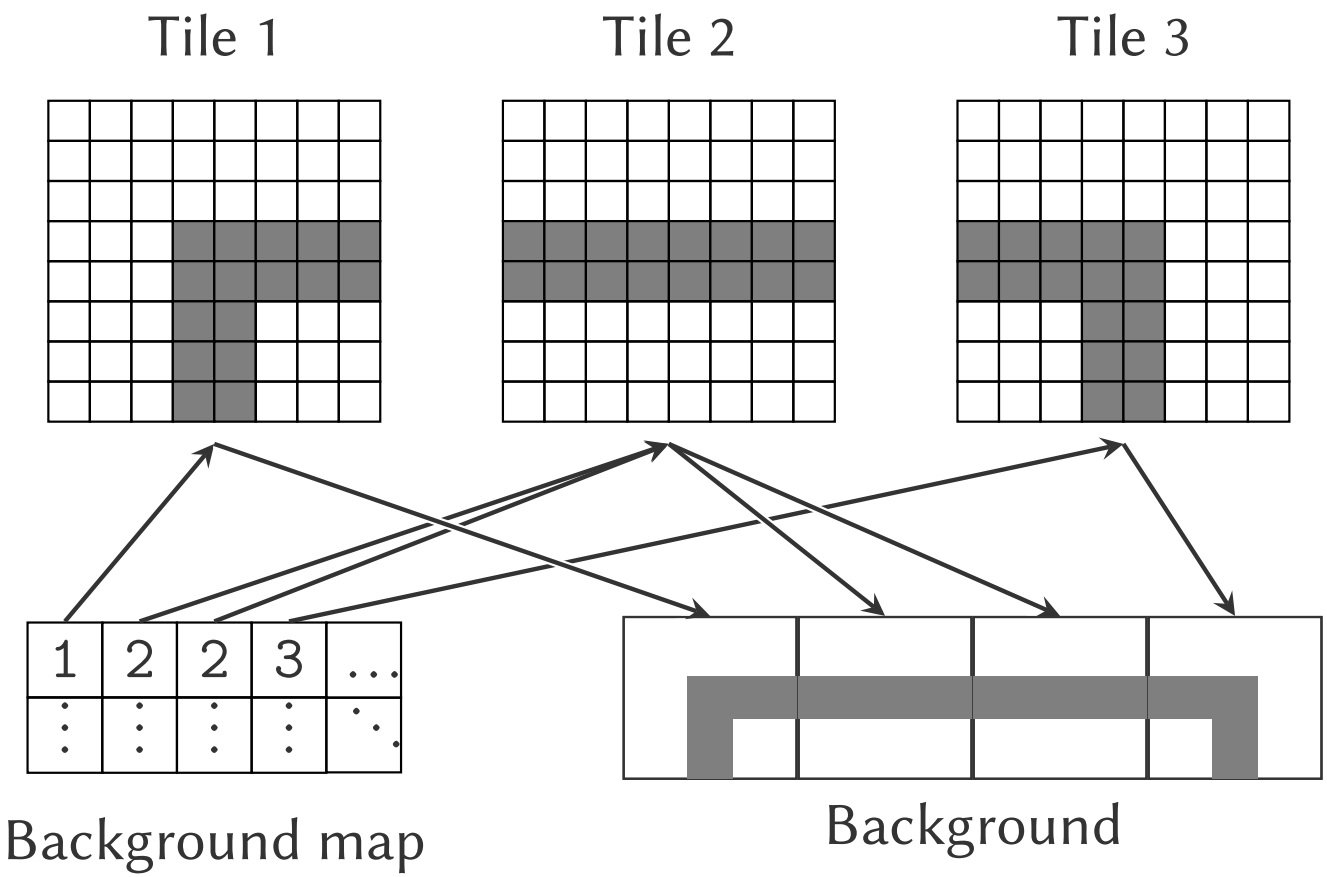
Phase	Ticks
Line (background)	172
Line (sprites)	80
Horizontal blank	204
Single line	456
Vertical blank	4560
Entire frame	70224

- How could we keep track of *when* to trigger each phase?

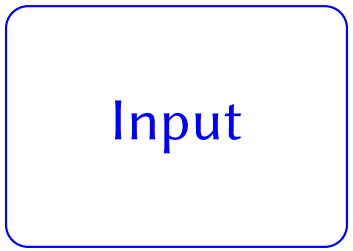
Tiling system

- GameBoy doesn't have enough memory to keep track of the entire matrix of pixels; instead, we use a *tiling* system.
- VRAM contains "tiles" of size 8×8, and the background is represented as a 32×32 matrix of pointers to the tiles.
- Observe that this gives us a total resolution of 256×256, yet GameBoy's LCD is 160×144!
  - Two special-purpose registers, `Scroll-X` and `Scroll-Y`, contain the coordinates of the point on the background that will be located in the upper-left corner of the LCD.

Tiling system, *cont'd*



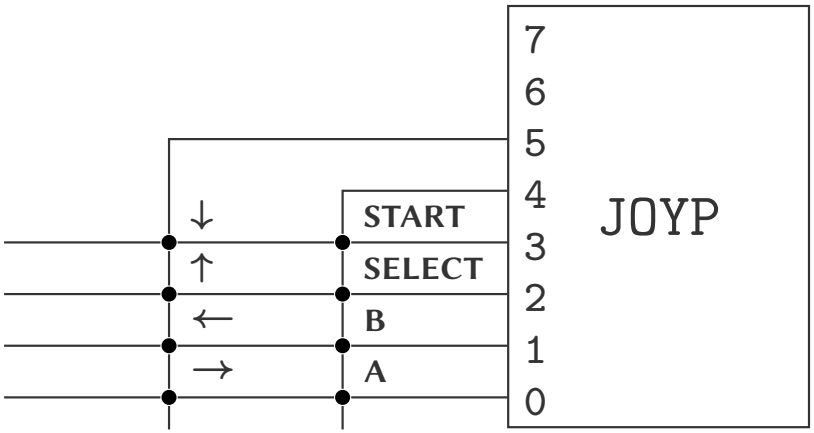
Input



- With the GPU implemented, the emulator is able to display graphical output, but the user/player still has no control over the game.
- Next step: emulating user input.

Input

- Input is processed using a special-purpose JOYP register, which the processor may access at address 65280. The wires leading to that register are connected in the following way:



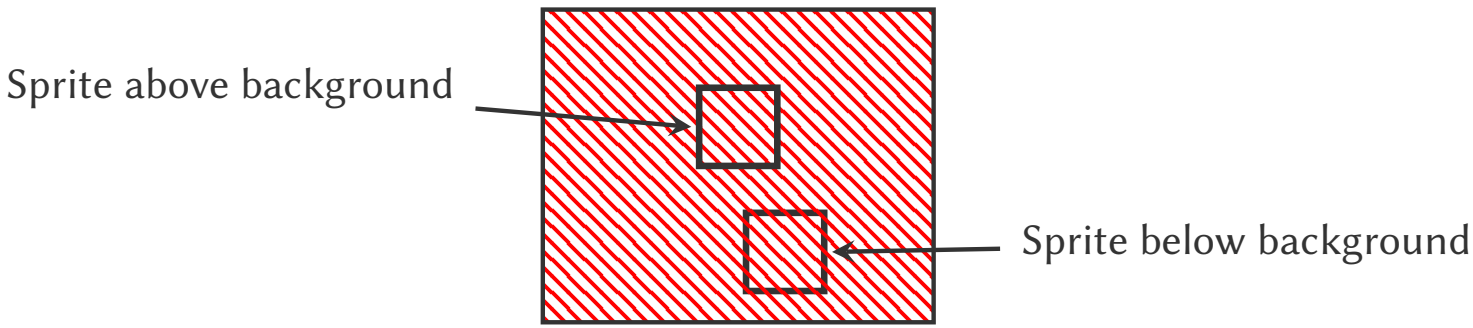
- The processor first writes 0x10 or 0x20 to the register, so it can activate one of two columns; then, after several ticks, it can read the least significant 4 bits in order to infer which buttons were pressed.

Sprites

- With the input processed, we can play some games – but we usually have to play blindly (there is no way the player can tell what his/her position is).
- This issue (and many others) is solved by using **sprites** – tiles which may be drawn and moved independent of the background.

Sprites: additional details

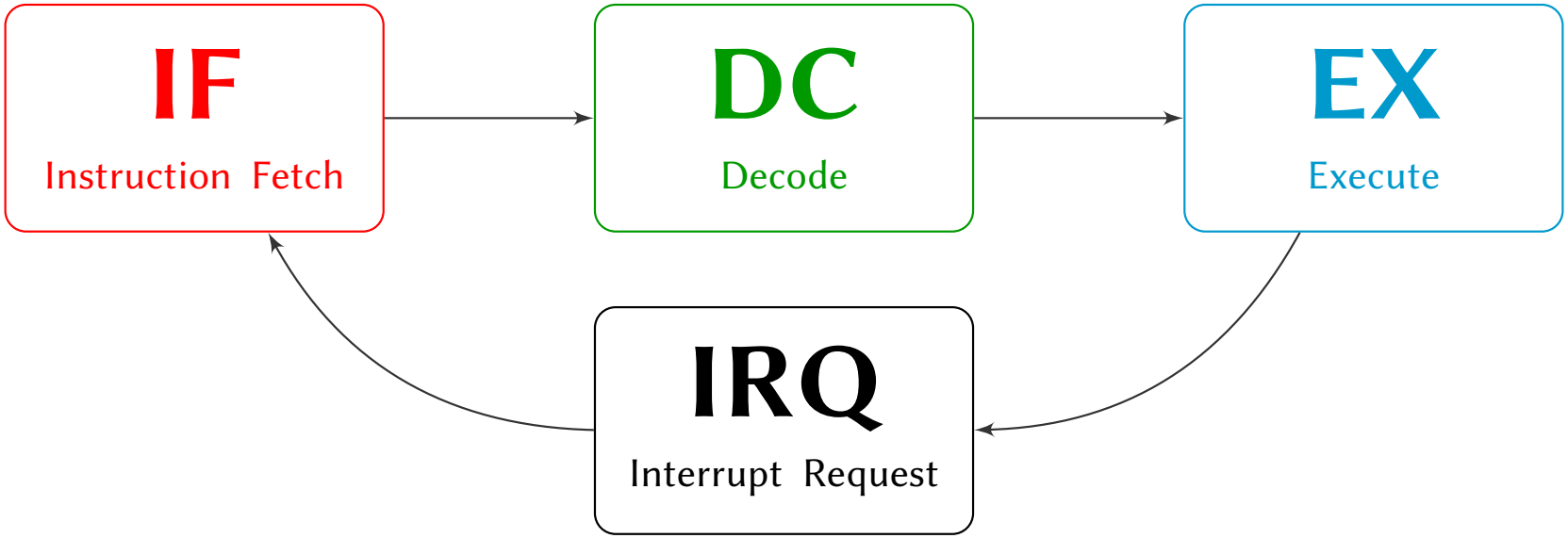
- Sprites use exactly the same type of 8×8 tiles as the background.
- Specific parameters, maintained for every sprite in the OAM:
  - (x, y) coordinates of the upper-left corner;
  - *priority:* is it in the foreground or behind the background;
  - *flip:* horizontal or vertical;
  - *size:* all sprites can be resized to 8×16 at the same time.
- The frame rendering algorithm: first draw the background, then infer (based on colours and sprite priority) whether to draw the sprite over the background.



Interrupts

- With sprite support, some simpler games work exactly as expected.
- Most games require a very important detail: notifying the processor **when** it may render the next frame.
- The most convenient time for changing the frame is **during VBlank**, because during the line change, the GPU doesn't access either VRAM nor OAM.
- The best (in terms of resource management and requirements) method for notifying the processor about an important event such as entering VBlank is using **interrupts**.

Fetch-decode-execute cycle, revisited



- After every executed instruction, the CPU needs to check whether an interrupt was raised; if an interrupt is detected, the CPU:
  - Remembers its current state;
  - Jumps (sets the PC) to the address of the *interrupt handler*.
  - Executes the interrupt handler code, which ends with a special `RETI` *(Return From Interrupt)* instruction, which restores state.

Interrupts: GameBoy specifics

- The processor has access to the *Interrupt Enable (IE)* registers, through which it can specify which interrupts (out of 5 in total) it is willing to handle at this point.
  - If multiple interrupts are detected, the one with the highest *priority* is handled first.

- The processor also has an *Interrupt Master Enable (IME)* switch, which allows it to completely deactivate interrupt handling (this is done during e.g. processing an interrupt, because we should not handle two of them at the same time).

GameBoy interrupts

Priority	Interrupt	Adress of the handler
0	*VBlank*	0x0040
1	*LCD Status*	0x0048
2	*Timer*	0x0050
3	*Serial*	0x0058
4	*Joypad*	0x0060

We're done!

- Even with only implementing the VBlank interrupt, our emulator is capable of running *Tetris*. :)
- All details about GameBoy's functionalities, as well as the Game Boy Color console, may be found on the following (extremely useful) reference page:
  `http://problemkaputt.de/pandocs.htm`
- For any further queries, feel free to contact me:
  `pv273@cam.ac.uk`
- **Thank you!**