INFO0940-1: Operating Systems
(Groups of 2)

Project 2: Emulating the Beta-machine

Prof. L. Mathy - V. Rossetto - G. Gain

Submission before Wednesday, May 24, 2023 (23:59:59)

1 Introduction

The beta-machine, as introduced in the Computation Structures course, is a toy RISC-like CPU based computer. It is as simple as possible for pedagogical purposes. While it is a nice tool to teach about CPU architecture, its very basic functionalities and instruction set make it a perfect subject to learn about full CPU emulation. In this project, you will write an emulator for the (extended) beta-machine. In addition, you will implement an interrupt handler in beta-assembly.

2 The extended beta-machine memory

The extended beta-machine's memory is organized according to the following general structure (Fig. 1).

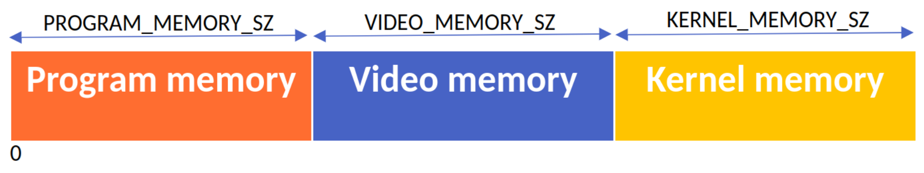

Figure 1: General memory organization

- Program memory contains the user-space program code and stack. An exact copy of the binary produced by BSim is inserted at the beginning of program memory. If there is empty space after it, it can be used as general-purpose memory by the user-space program.

- Video memory is a collection of RGB values that are displayed on the emulated screen. It is essentially a flattened matrix of pixels.
- Kernel memory contains a few basic data structures that are used by the interrupt handler, directly followed by the interrupt handler itself.

2.1 Video memory

The pixel matrix stored in video memory is a succession of rows from the pixel matrix (from top to bottom). Each row is composed by a succession of pixels (from left to right). A pixel is represented by 4 bytes: 1 byte for each primary color's intensity (Red, Green, Blue) and 1 byte of padding.

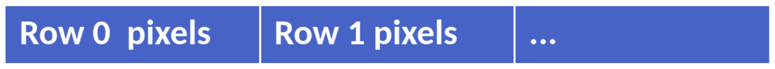

Figure 2: Video memory organization (row level)

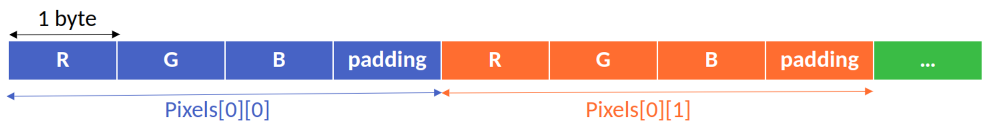

Figure 3: Video memory organization (pixel level)

2.2 Kernel memory

Kernel memory is organized as follows. It begins with a fixed-size block of kernel data structures followed by the interrupt handler's (binary) code (Fig. 4).

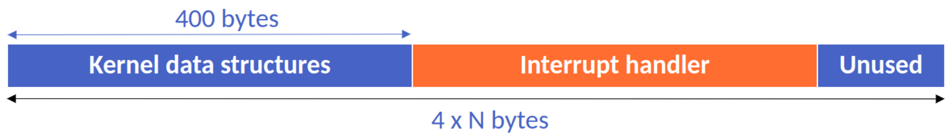

Figure 4: Kernel Memory organization

The following data structures (Fig. 5) are found before the interrupt handler.

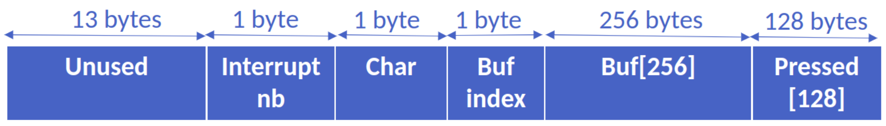

Figure 5: Kernel data structures

- `Interrupt nb`: Before handing control to the interrupt handler, the CPU will place the interrupt number in this specific place.
- `Char`: For keyboard-related interrupts, it is used to identify the key the interrupt is related to. It is placed in memory by the CPU before handing control to the interrupt handler.
- `Buf Index`: Index of the next cell to be written in `Buf`.
- `Buf`: Circular buffer of characters. Each key-pressed interrupts inserts the character's (ASCII) value inside of the buffer at the specified index and increments the index (with wraparound at the end of the buffer).
- `Pressed`: Each key-pressed interrupts sets `Pressed[char]` to 1 and each key-released interrupts sets `Pressed[char]` to 0.

3 Assignment

3.1 Emulator implementation

The main task of this assignment is to implement the emulator itself. You are provided with a graphical interface to test your emulator locally. You will implement the interface found in *emulator.h* in a file named *emulator.c*. You are free to provide additional source files for modularity purposes if you deem it is necessary. The programs passed as argument to the emulator are binaries, not assembly files. You can generate Beta-machine binaries using BSim from a Beta-assembly file (see project presentation slides).

The `Computer` and `CPU` data structures are defined in *emulator.h*. You are free to add any field you deem necessary into them.

```
typedef struct{

    long program_counter;

    // add your own fields here!

} CPU;
```

```c
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;
```

The header file also declares the interface you will need to implement. These methods are all used to manipulate the `Computer` data structure. The provided graphical interface will call these methods and allow you to visualize the internal state of the emulated computer as the code is running.

```c
/* 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);
```

```
/* Loads the binary at the beginning of the computer's memory,
   c -> program_size becomes the size of the binary in bytes. */
void load(Computer* c, FILE* binary);

/* Loads the interrupt handler binary in $c's kernel memory.
   $binary can be NULL, in which case the function does nothing.
   The $binary is placed after the kernel's data structures
   (see statement for a diagram of kernel memory). */
void load_interrupt_handler(Computer* c, FILE* binary);

/* Runs one fetch + decode + execute cycle of $c's CPU,
   If an interrupt line is raised (and the computer is not
   already executing the interrupt handler), the program counter
   becomes the start address of the interrupt handler
   and its first instruction is executed.
   Before handing control to the interrupt handler, the CPU
   places the interrupt number and associated character at
   the adequate place in kernel memory (see statement) and
   stores PC into XP so that the interrupt handler is able to
   return. */
void execute_step(Computer* c);

/* Raise an interrupt line of computer $c if no other already is.
   Otherwise, this does nothing.  $type is the interrupt number
   while $keyval is the associated character. */
void raise_interrupt(Computer* c, char type, char keyval);

/* Stores a textual representation of the disassembly of
   $instruction in the buffer $buf. We assume that $buf
   is large enough to store any disassembled instruction.
   If $instruction is not a valid instruction (see slides
   on the Beta-assembly instruction set), then the
   "INVALID" string is stored instead.
   Returns 0 if $instruction is valid, and a negative
   value otherwise. */
int disassemble(int instruction, char* buf);
```

3.2 Interrupt handler

You will write a generic interrupt handler in Beta-assembly. This interrupt handler will handle two types of interrupts. The interrupt handler must work regardless of start of address of kernel memory. Do not hardcode it as a constant. The interrupt handler must save the state of registers and restore it before handing back control to the user program. The XP register (number 30) contains the interrupt's return address (the address of the next instruction to run in the user-space program).

- *Interrupt nb = 0*: key-pressed interrupt, the interrupt handler will put the character's ASCII value into the circular buffer at the index found in the associated counter, increment said counter, and will set `Pressed[char] = 1` (where `char` is the ASCII value of the character.
- *Interrupt nb = 1*: key-released interrupt, the interrupt handler will set `Pressed[char] = 0`.

The provided graphical interface expects a binary interrupt handler named *interrupt_handler.asm.bin*. You can generate it from your Beta-assembly file using BSim. If no interrupt handler binary is provided, the graphical interface will still call your implementation of the functions from the *emulator.h* interface and will never trigger the interrupt mechanism regardless of keyboard keys pressed.

4 Bonus: Beta-assembly art (up to 2 points)

Provide Beta-assembly code to draw the following picture as close to the original as possible on the virtual screen of the emulated computer. You may hardcode the starting address of the video memory and the dimensions of the pixel matrix. Use the same values than for your user-defined macros in *emulator.h* if you modified the default values.

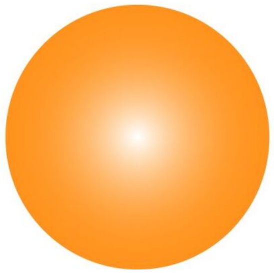

Figure 6: orange gradient circle

5 Generic Requirements

The following requirements also have to be satisfied:

- Your code must be readable. Use common naming conventions for variable names and functions.

- Your code must be robust and must not crash. Cleaning must be properly managed.
- The provided graphical interface relies on GTK version 4 (do not use other version). It should work on most recent Linux distributions but in case it does not, or behaves weirdly, check on the reference virtual machine which is provided in the Project 2 directory on eCampus.

Note that Gtk's non-critical warnings related to style, etc. can be safely ignored. However, if critical warnings start to appear during execution, it may be related to your emulator interface handing invalid data to the graphical interface.

6 Compilation

Your code should compile without warning using the following command:

```
gcc ‘pkg-config --cflags gtk4‘ *.c ‘pkg-config --libs gtk4‘ -lm
-Wno-deprecated-declarations
```

Note that it requires to install GTK-4 (the libgtk-4-dev package). Please put all source files into the same folder as the provided *graphics.c*

7 Evaluation and tests

No automatic tests are provided on the submission platform. Please test your implementation locally with the help of the provided graphical interface **AND** with your own test code. The evaluation will be based on achievement of the expected functionality, respect of the interface, style and robustness.

8 Submission

Projects must be submitted before the deadline. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^N - 1$ marks (where N is the number of started days after the deadline).

Submit a submission.tar.gz archive containing a folder named *code* in which can be found all your C source files, your *interrupt_handler.asm* source file as well as the corresponding your *interrupt_handler.asm.bin* binary. If you did the bonus, provide the assembly and associated binary file as well. Failure to compile will result in an awarded mark of 0.

Bon travail...