Emulators: from definition to implementation

Andrea Francesco Iuorio

Università degli Studi di Milano

Milan, 18 January 2016

Introduction to emulation

An emulator is a software or hardware that behaves like another software or hardware

- The system that executes the emulator is called host system
- The implemented hardware or software is called guest system

An emulator is NOT a virtual machine

- An emulator implements a specific software or hardware
- A virtual machine simulates parts of an architecture

Emulators

The great advantage of an emulator is that we can execute applications written for a specific architecture on another one

This makes emulators essential for

- Software preservation
- Executing legacy software on newer hardware
- Improving software security
- Adding additional features that the original hardware didn't have

Let's develop an emulator !

CHIP-8 system

We'll develop an emulator for the CHIP-8 system

- A complete version of this emulator is provided here: `https://bitbucket.org/aiuorio/panc-8`

The CHIP-8 system is an 8-bit architecture for videogames

The entire architecture is composed by 4 different components:

- An 8-bit CPU
- A 4096 byte (4 KB) RAM
- A monochromatic video screen (64x32)
- A keyboard with 16 keys

We need to emulate all these HW components

CHIP-8 CPU

The CHIP-8 CPU is an 8-bit cpu with a 16-bit address bus

- 15 8-bit registers (V0-VE)
- 8-bit flag register (VF)
- 16-bit address register (I)
- 16-bit stack pointer (SP)
- 16-bit program counter (PC)

An instruction is 16-bit long, and is stored in big-endian order

```
typedef struct c8_cpu{
    BYTE registers[16];
    WORD mem_register;
    WORD stack_pointer;
    WORD prog_counter;
} c8_cpu;
```


CHIP-8 memory

Emulators: from definition to implementation
Andrea Iuorio
Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

The CHIP-8 CPU uses an address space of 4096 bytes

The memory map is organised as follow:

- 0x000 - 0x1FF is reserved
- 0x200 - 0xE9F is where the application is loaded
- 0xEA0 - 0xEFF is used as call stack
- 0xF00 - 0xFFF is used for the screen's framebuffer

An application can only use the 0x200-0xE9F memory area

CHIP-8 memory

Emulators: from definition to implementation
Andrea Iuorio
Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

Today is easy to implement this memory map, since we can allocate all the 4 KB of RAM required

```
#define MEM_SIZE 0xFFF

#define PROG_MEM 0x200
#define STACK_MEM 0xEFF
#define FB_MEM 0xF00

BYTE memory[MEM_SIZE];
```

We can also create macros for helping us handling the memory map

pchip-8

Emulators: from definition to implementation
Andrea Iuorio
Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

Now that we have the structures for the CPU and the memory, we can start to emulate them

The main purpose of our emulated CPU is to work like a real one: we execute a list of instructions that will modify the state of both the CPU and the memory

The CPU cycle

A CPU clock cycle is usually composed by three different phases:

- Fetch the next instruction from the memory
- Decode the instruction, understanding what is does
- Execute the instruction, modifying the registers and the memory

Our emulator can be seen as an infinite loop in which we execute a clock cycle:

```
while(1){
        WORD opcode = fetch(cpu);
        decode(opcode, cpu);
        execute(opcode, cpu);
}
```


Fetch

Emulators: from definition to implementation
Andrea Iuorio

Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

The fetch phase is easy to implement (remember, the CHIP-8 is big endian):

```
WORD fetch(c8_cpu *cpu){
    WORD opcode = cpu->memory[cpu->prog_counter];
    opcode <<= 8;
    opcode |= cpu->memory[cpu->prog_counter+1];

    cpu->prog_counter+=2;

    return opcode;
}
```


Decode

Emulators: from definition to implementation
Andrea Iuorio

Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

The fetch phase provides us the opcode of the instruction. We have to "translate" it for understating what the instruction does

The CHIP8 uses a RISC-like encoding: the first 4 bit of an opcode indicates the type of the instruction

For example, 0x8014 means:

- 8 indicates a register-to-register operation
- 0 and 1 indicates which registers are used
- 4 indicates the operation (an addition with carry)

Decode
Implementation

Emulators: from definition to implementation
Andrea Iuorio

Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

An easy way to implement the decode phase is a jump table: we use the opcode as an index for calling a function in a table of functions

```
void decode(WORD opcode, c8_cpu *cpu){
    switch(opcode){
        case NOP_OPCODE:
            NOP_INSTR(cpu);
            break;
        case ADD_OPCODE:
            ADD_INSTR(cpu);
        ...
    };
}
```

This decoding method is called interpreted mode. There are other algorithms, like dynamic recompiling, but they are much more difficult to implement

Execute

The decode phase calls a function that implements what the instruction does

The CHIP-8 has 35 opcodes so we can't present them all here. We will provide some examples: The add operation

```
void ADD_INSTR(BYTE f, BYTE s, c8_cpu *cpu){
    cpu->registers[FLAG_REG] = 0;

    WORD value = cpu->registers[f] + cpu->registers[s];
    if(value > 0xFF)
        cpu->registers[FLAG_REG] = 1;

    cpu->registers[f] = value & 0xFF;
}
```

We also have to set the register flag when we have an overflow

CPU timing

Our machine is much more powerful compared to the CHIP-8, being able to execute a lot more clock cycles per second

This is a problem since programs and other hardware can be synced with the clock speed of the CHIP-8 CPU

We have to "slow down" the emulation.

Instruction timing

A CPU executes a certain number of cycles per second. Not all instructions require the same number of CPU clocks: some instruction are slower than others

- For example, the 16-bit operations of the Z80 require 8 clock cycles per instruction

Instruction timing is really important: a program often includes NOP operations for sync with the CPU pipeline or caches

- Every instruction of the CHIP-8 requires a single clock cycle
- The CHIP-8 CPU doesn't have pipelines or caches
- However, the programs use the CPU clock to handle timers and interrupts

CHIP-8 timing

The CHIP-8 uses a variable clock, based on the hardware implementation.
A good general value is 420 instructions per second, meaning we have to execute our loop only 420 times every second

```
while(1){
    for(int i = 0 ; i< 420 ; i++){
        WORD opcode = fetch(&c8);
        decode(opcode, &c8);
        execute(opcode,&c8);
    }
    wait(1000);
}
```

CHIP-8 timing
Delta Time

There are two main problems with this approach:

- Different CPUs executes our main loop at different speeds
- The execution time of our loop isn't constant

```
while(1){
    unsigned int timeStartFrame = GetTicks();
    for(int i = 0 ; i< 420 ; i++){
        WORD opcode = fetch(&c8);
        decode(opcode, &c8);
        execute(opcode,&c8);
    }
    float deltaT = (float)1000 - (float) (GetTicks()- timeStartFrame );
    if(deltaT>0)
        wait((unsigned int)deltaT);
}
```

This also presents a problem... but we solve it later !

CHIP-8 timing
Timers

The CHIP-8 CPU has 2 8-bit timers

- A delay timer, used for handling game timing
- A sound timer, used for sound management (beeps when non-zero)

Each timer count down every 16 ms (60 in a second). We should execute an updateTimers every (420/60) = 7 iterations

```
while(1){
    unsigned int timeStartFrame = GetTicks();
    for(int i = 0 ; i< 420/60 ; i++){
        WORD opcode = fetch(&c8);
        decode(opcode, &c8);
        execute(opcode,&c8);
    }
    updateTimers(&c8);
    float deltaT = (float)1000/60 - (float) (GetTicks()- timeStartFrame );
    if(deltaT>0)
        wait((unsigned int)deltaT);
}
```


GPU emulation

For our example, we can see a GPU as a chip that reads from a memory buffer representing the current frame and draws it on the screen

The application, modifying this memory, can change what appears on the screen. This memory area is called framebuffer

- On old systems, programs could directly access the framebuffer for drawing their graphics
- The CHIP-8 programs doesn't access the framebuffer directly but uses a special instruction: DXYN

CHIP-8 video system

The CHIP-8 uses a monochromatic screen with a resolution of 64x32, so each frame requires 2048 pixels

The CHIP-8 framebuffer is mapped to the address 0xF00 - 0xFFF. Each bit in this area represents a single pixel: if the pixel is 1, the pixel is activated (white color)

The screen has a refresh rate of 60 hz and it doesn't use neither a double framebuffer nor the V-Blank technique

CHIP-8 video system (cont.)

The CHIP-8 uses sprites

- Each bit is mapped to a pixel
- Fixed width (8 pixels)
- Variable height

A frame is a composition of sprites and this can cause some problems during its creation (like sprite collision). The CHIP-8 solves this problem using bit blit:

- We take all the sprite bitmaps
- We execute a xor operation between all the bitmaps

DXYN executes the blit operation.

DXYN implementation


```
void DRAW(BYTE X, BYTE Y, BYTE N, c8_cpu *cpu){
    int coordx = cpu->registers[X];
    int coordy = cpu->registers[Y];

    cpu->registers[15] = 0 ;

    for (int yline = 0; yline < N; yline++)
    {
        BYTE data = cpu->memory[cpu->mem_register + yline];

        int xpixel = 0 ;
        int xpixelinv = 7 ;
        for(xpixel = 0; xpixel < 8; xpixel++, xpixelinv--)
        {
            BYTE work = data >> xpixelinv;
            work &= 0x1;
            int x = xpixel + coordx ;
            int y = coordy + yline ;
            BYTE tmp = cpu->video_memory[y][x];
            cpu->video_memory[y][x] ^= work;
            if((tmp == 1) && (cpu->video_memory[y][x] ==0) )
                cpu->registers[15] = 1;
        }
    }
}
```


Input emulation

The CHIP-8 uses a keyboard with 16 keys

The CPU has a 16-bit mask for the keyboard's state

In the real CHIP-8 the input mask is updated from the keyboard's hardware in "real time", but our emulator doesn't have a separate chip that can update the input mask

- We can use a separate thread
- We can update the input mask periodically (can introduce input lag)

Input emulation

We'll use the latter approach since it's simpler: when we update the timers and the screen, we also execute a function for updating the CPU's input mask

```
while(1){
    unsigned int timeStartFrame = GetTicks();
    for(int i = 0 ; i< 420/60 ; i++){
        WORD opcode = fetch(&c8);
        decode(opcode, &c8);
        execute(opcode,&c8);
    }
    getInput(&c8);
    updateTimer(&c8);
    renderVideoMemory(&c8,renderer, *SCALE);
    float deltaT = (float)1000/60 - (float) (GetTicks()- timeStartFrame );
    if(deltaT>0)
        wait((unsigned int)deltaT);
}
```

Input emulation
Opcodes

The CHIP-8 has 3 opcodes for detecting keystrokes

- 0xFX0A that uses a polling strategy

```
void FX0A(WORD opcode, WORD reg, c8_cpu *cpu){
    int i;
    for (i = 0 ; i < 16; i++)
        if (cpu->keymap[i] > 0)
            break;
    if (i == 16)
        cpu->prog_counter -= 2 ;
    else
        cpu->registers[reg] = i;
}
```


Emulators: from definition to implementation
Andrea Iuorio
Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

Input emulation
Opcodes

- 0xEX9E that skips the next instruction if the key is pressed
- 0xEXA1 that skips the next instruction if the key is released

```
void EX9E(WORD opcode, WORD reg, c8_cpu *cpu){
    BYTE val = cpu->registers[reg];
    if(cpu->keymap[val] == 1)
        cpu->prog_counter+=2;
}
void EXA1(WORD opcode, WORD reg, c8_cpu *cpu){
    BYTE val = cpu->registers[reg];
    if(cpu->keymap[val] == 0)
        cpu->prog_counter+=2;
}
```


Emulators: from definition to implementation
Andrea Iuorio
Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

pchip-8
Application loading

Our emulator is almost ready. We just need a couple of little tweaks:

We need to load the application in memory. As we said, the application is loaded starting from 0x200

```
void load(c8_cpu *cpu, char *rom){
    FILE *f = fopen(rom, "rb");
    if(!f){
        fprintf(stderr, "Can't load ROM\n");
        exit(2);
    }
    fread(cpu->memory+ PROG_MEM, 1, 0xE9F - 0x200, f);
    fclose(f);
}
```


Emulators: from definition to implementation
Andrea Iuorio
Introduction
The CPU cycle
CPU timing
GPU emulation
Input emulation
Conclusions

Reserved memory

The area 0x00-0x1FF, as we said, is reserved, but we need to put in it the font map (address 0x050)

```
void initCPU(c8_cpu *cpu){
    memset(cpu->memory, 0, MEM_SIZE);
    memcpy(cpu->memory,chip8_fontset, 0x50 );
    memset(cpu->registers, 0, 16);
    cpu->mem_register = 0;
    cpu->prog_counter = PROG_MEM;
    cpu->stack = (WORD *)(cpu->memory + STACK_MEM);
    memset(cpu->keymap, 0, 16);
    memset(cpu->video_memory, 0, 64*32);
    cpu->timer = 0;
    cpu->soundTimer = 0;
}
```

Conclusions

Now our emulator should be ready to run

One of the greater advantages of an emulator is that, since it is a software, we can easily modify and extend it

- We can easily implements an "online multiplayer mode", receiving the key pressed from the network instead of the keyboard
- We can use a bigger screen resolution
- We can support input devices that the CHIP-8 doesn't support, like a gamepad

The CHIP-8 system is really simple and it doesn't have some problems like VBlank syncing or nested interrupts, but now you have developed a little emulator from scratch