Conception d'un emulateur CHIP-8

Projet assembleur x86-64

Auteur : Deroubaix Sasha

31 January 2026

Table des matieres

1 Introduction

Ce rapport presente la conception et le developpement d'un emulateur (interpreteur) CHIP-8 ecrit en **assembleur x86-64** (NASM), avec une couche d'interface graphique et audio en C utilisant la bibliotheque **Raylib**. Le projet s'inscrit dans le cadre du module assembleur de Master 1.

1.1 Contexte du projet

Le **CHIP-8** est un langage de programmation interprete, concu a l'origine dans les annees 1970 par Joseph Weisbecker pour le microprocesseur COSMAC VIP. Il s'agit d'une machine virtuelle simplifiee qui a ete pensee pour faciliter le developpement de jeux video sur les micro-ordinateurs de l'epoque.

Ses caracteristiques en font un excellent projet d'apprentissage de l'assembleur :

- **Architecture simple** : 16 registres 8 bits, 4 Ko de memoire, un ecran de 64x32 pixels
- **Jeu d'instructions reduit** : 35 opcodes seulement, chacun encode sur 2 octets
- **Pas de pipeline ni cache** : le cycle fetch-decode-execute est lineaire et previsible
- **Nombreuses ROMs de test** : une large variete de rom son desormait dans le domaine public

> **Attention**
>
> Un emulateur CHIP-8 n'est pas un emulateur materiel au sens strict : c'est un **interpreteur** qui simule une machine virtuelle. Il n'existe aucun processeur physique CHIP-8 – c'est un bytecode concu pour etre interprete par un programme hote.

1.2 Objectifs

Les objectifs du projet sont :

1. Comprendre et implementer un cycle **fetch-decode-execute** complet en assembleur x86-64
2. Interfacer de l'assembleur avec du C pour la partie graphique
3. Implementer les opcodes de la specification CHIP-8
4. Gerer l'affichage, le son, le clavier et les timers
5. Ajouter des fonctionnalites supplementaires

2 Architecture du projet

2.1 Vue d'ensemble

Le projet est organise en deux couches distinctes : le **coeur de l'emulateur** en assembleur x86-64 et la **couche d'interface** (graphique, audio, clavier) en C.

Composant	Role	Langage
`main.s`	Point d'entree, boucle principale, parsing des arguments	ASM
`chip8_state.s`	Etat du CPU : memoire, registres, pile, timers	ASM
`rom_loader.s`	Chargement de la ROM via syscalls Linux	ASM
`cpu.s`	Fetch des opcodes	ASM
`dispatcher.s`	Decodage et dispatch des 35 opcodes	ASM
`op_*.s` (17 fichiers)	Implementation de chaque opcode	ASM
`display.c`	Fenetre, rendu des pixels via Raylib	C
`audio.c`	Generation et lecture du beep 440 Hz(La)	C
`input.c`	Mapping clavier vers keypad CHIP-8	C
`timers.c`	Decrementation des timers delay et sound	C

Tableau 1. – Fichiers source du projet et leur role

2.2 Structure des repertoires

```
asm_chip_8/
├── Makefile
├── include/
│   ├── display.h
│   ├── audio.h
│   ├── input.h
│   └── timers.h
├── src/
│   ├── asm/
│   │   ├── main.s
│   │   ├── chip8_state.s
│   │   ├── rom_loader.s
│   │   ├── cpu.s
│   │   └── opcodes/
│   │       ├── dispatcher.s
│   │       ├── op_00E0.s ... op_FXxx.s
│   └── c/
│       ├── display.c
│       ├── audio.c
│       ├── input.c
│       └── timers.c
└── roms/
    ├── test/
    └── games/
```

2.3 Choix d'architecture : ASM + C

Le projet repose sur une separation claire des responsabilites :

- **Assembleur x86-64** : tout ce qui concerne la logique du CPU virtuel, initialisation de l'etat, chargement de la ROM, cycle fetch-decode-execute, implementation des 35 opcodes.
- **C avec Raylib** : tout ce qui concerne l'interaction avec le systeme d'exploitation pour l'affichage graphique, l'audio et le clavier. Raylib est une bibliotheque legere et simple d'utilisation qui evite la complexite de SDL ou OpenGL pur.

L'assembleur appelle les fonctions C via le mecanisme standard `call` / `extern`, en respectant la convention d'appel **System V AMD64 ABI** (parametres dans `rdi`, `rsi`, `rdx`…, alignement de la pile sur 16 octets).

> **Note**
>
> Le flag `-no-pie` est necessaire lors du linkage car l'assembleur utilise des adresses absolues pour acceder aux variables globales (MEMORY, REGISTERS, etc.). Sans ce flag, le linker genererait des erreurs de relocation.

3 Le CPU virtuel CHIP-8

3.1 Composants du CPU

Le CHIP-8 possede une architecture volontairement simple. Voici l'ensemble des composants de son CPU :

Composant	Taille	Variable ASM	Description
Memoire	4 096 octets	`MEMORY`	Espace d'adressage complet (0x000 – 0xFFF)
Registres generaux	16 x 1 octet	`REGISTERS`	V0 a VF – VF sert de flag
Registre d'index	2 octets (16 bits)	`REG_I`	Pointe vers des adresses memoire
Compteur de programme	2 octets (16 bits)	`PC`	Adresse de l'instruction courante
Pile	16 x 2 octets	`STACK`	Stocke les adresses de retour
Pointeur de pile	1 octet	`CH8_SP`	Index du sommet de pile (0-15)
Delay timer	1 octet	`DELAY_TIMER`	Decremente a 60 Hz, lisible par le programme
Sound timer	1 octet	`SOUND_TIMER`	Decremente a 60 Hz, emet un son quand > 0
Ecran	256 octets	`DISPLAY`	Buffer 64x32 pixels (1 bit par pixel)
Clavier	16 octets	`KEYPAD`	Etat des 16 touches (0 ou 1)

Tableau 2. – Composants du CPU virtuel CHIP-8

Comparaison avec x86-64

Le CHIP-8 possede 16 registres generaux comme le x86-64 (RAX, RBX, RCX…), mais ceux du CHIP-8 font seulement **8 bits** contre 64 bits pour le x86-64. Le registre **VF** joue un role similaire au **FLAGS** du x86 : il sert de carry, borrow et indicateur de collision.

3.2 Organisation de la memoire

La memoire du CHIP-8 fait 4 096 octets (4 Ko) et est organisee de la facon suivante :

Adresses	Taille	Contenu
`0x000 -- 0x04F`	80 octets	Fontset : sprites des caracteres hexadecimaux 0-F
`0x050 -- 0x1FF`	432 octets	Zone reservee (historiquement : interpreteur)
`0x200 -- 0xFFF`	3 584 octets	Programme ROM charge ici

Tableau 3. – Carte memoire du CHIP-8

Le **PC** (Program Counter) demarre toujours a l'adresse `0x200` car les 512 premiers octets etaient historiquement occupes par l'interpreteur CHIP-8 lui-meme sur les machines d'origine.

3.3 Initialisation du CPU

La fonction `init_chip8` dans `chip8_state.s` prepare l'ensemble de l'etat du CPU :

```
init_chip8:
    •••
    ; PC = 0x200 (debut de la ROM)
    mov ax, 0x200
    mov [rel PC], ax

    ; SP = 0 (pile vide)
    xor eax, eax
    mov byte [rel CH8_SP], al

    ; I = 0
    xor ax, ax
    mov [rel REG_I], ax

    ; Effacer les 16 registres (V0-VF)
    xor rax, rax
    mov rcx, 0x10
    lea rdi, [rel REGISTERS]
    rep stosb

    ; Effacer les 4096 octets de memoire
    xor rax, rax
    mov rcx, 0x1000
    lea rdi, [rel MEMORY]
    rep stosb
    •••
```

L'instruction `rep stosb` est l'equivalent d'un `memset` en C : elle ecrit `AL` (ici 0) dans `RCX` octets consecutifs a l'adresse `RDI`. L'instruction `rep movsb` copie `RCX` octets de `RSI` vers `RDI`, soit l'equivalent d'un `memcpy`.

3.4 Le fontset

Le fontset est un ensemble de 16 sprites de 5 octets chacun (80 octets au total), representant les caracteres hexadecimaux 0 a F. Chaque sprite fait 8 pixels de large et 5 pixels de haut :

```
FONTSET:
    db 0xF0, 0x90, 0x90, 0x90, 0xF0  ; 0
    db 0x20, 0x60, 0x20, 0x20, 0x70  ; 1
    db 0xF0, 0x10, 0xF0, 0x80, 0xF0  ; 2
    ...
    db 0xF0, 0x80, 0xF0, 0x80, 0x80  ; F
```

Par exemple, le caractere « 0 » se decompose en binaire :

```
0xF0 = 1111 0000  ████
0x90 = 1001 0000  █  █
0x90 = 1001 0000  █  █
0x90 = 1001 0000  █  █
0xF0 = 1111 0000  ████
```

Chaque bit a 1 correspond a un pixel allume. L'opcode `FX29` permet de pointer le registre I vers un de ces sprites pour l'afficher ensuite avec `DXYN`.

4 Chargement de la ROM

Le chargement de la ROM est effectue via des **syscalls Linux**:

```
rom_loader:
    •••
    mov r12, rdi            ; Sauvegarder le chemin du fichier

    ; Syscall open(filename, O_RDONLY, 0)
    mov rax, 0x2
    mov rdi, r12
    xor rsi, rsi
    xor rdx, rdx
    syscall

    cmp rax, 0x0
    jl .error               ; Si fd < 0 : erreur

    mov rbx, rax            ; Sauvegarder le file descriptor

    ; Syscall read(fd, MEMORY+0x200, 0xE00)
    mov rax, 0x0
    mov rdi, rbx
    lea rsi, [rel MEMORY + 0x200]
    mov rdx, 0xE00          ; Lire jusqu'a 3584 octets max
    syscall

    ; Syscall close(fd)
    mov rax, 3
    mov rdi, rbx
    syscall
    •••
```

La ROM est lue en un seul appel `read` directement dans le tableau `MEMORY` a l'offset `0x200`. La taille maximale de lecture est `0xE00` (3 584 octets), ce qui correspond a l'espace disponible entre `0x200` et `0xFFF`.

5 Le cycle Fetch-Decode-Execute

Le coeur de tout processeur repose sur le cycle **fetch-decode-execute**. C'est une boucle qui se repete indefiniment : lire une instruction, la decoder, l'executer, puis passer a la suivante.

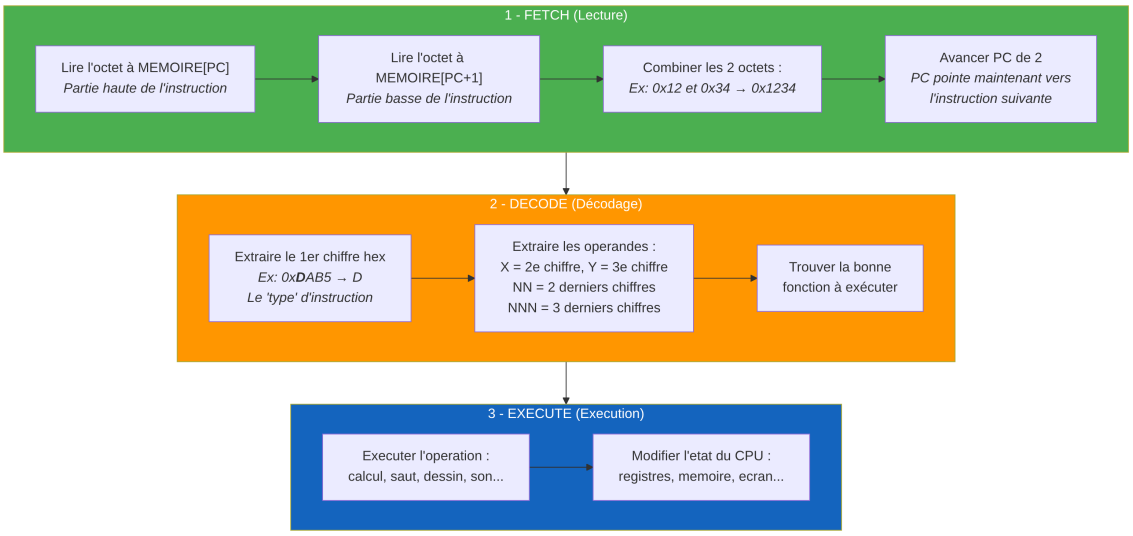

Fig. 1. – Schema du cycle Fetch-Decode-Execute du CHIP-8

5.1 Phase 1 : Fetch (Lecture)

Le **fetch** consiste a lire l'instruction courante depuis la memoire. Chaque instruction CHIP-8 fait **2 octets** (16 bits), stockes en **big-endian** (l'octet de poids fort en premier).

```
fetch_opcode:
    •••
    ; Lire le PC
    movzx rax, word [rel PC]
    •••
    ; Lire les 2 octets consecutifs
    lea rbx, [rel MEMORY]
    movzx rcx, byte [rbx + rax]      ; Octet haut
    movzx rdx, byte [rbx + rax + 1]  ; Octet bas

    ; Combiner en big-endian
    shl rcx, 8
    or rcx, rdx

    ; Avancer le PC de 2
    add word [rel PC], 2

    mov rax, rcx       ; Retourner l'opcode dans RAX
```

Exemple concret

Si `PC = 0x200` et que la memoire contient `0x12` a l'adresse `0x200` et `0x34` a l'adresse `0x201`, alors l'opcode fetche sera `0x1234`, soit l'instruction « Jump to address 0x234 ».

5.2 Phase 2 : Decode (Decodage)

Le decodage extrait les differentes parties de l'opcode de 16 bits. La structure d'un opcode CHIP-8 est la suivante :

Bits	Nom	Description
15-12	**Type**	Premier nibble : identifie la famille d'instruction
11-8	**X**	Deuxieme nibble : index du registre VX (0-F)
7-4	**Y**	Troisieme nibble : index du registre VY (0-F)
3-0	**N**	Dernier nibble : valeur immediate 4 bits
7-0	**NN**	Deux derniers nibbles : valeur immediate 8 bits
11-0	**NNN**	Trois derniers nibbles : adresse 12 bits

Tableau 4. – Structure d'un opcode CHIP-8 de 16 bits

L'extraction se fait par des operations de **decalage** (`shr`) et de **masquage** (`and`).

```
; Extraire le premier nibble (type d'instruction)
mov rax, r12          ; r12 contient l'opcode
shr rax, 12           ; Decaler de 12 bits vers la droite
and rax, 0xF          ; Garder seulement les 4 bits de poids faible

; Extraire X (index de registre)
mov rdi, r12
shr rdi, 8
and rdi, 0xF

; Extraire Y (index de registre)
mov rsi, r12
shr rsi, 4
and rsi, 0xF

; Extraire NN (valeur immediate 8 bits)
mov rsi, r12
and rsi, 0xFF

; Extraire NNN (adresse 12 bits)
mov rdi, r12
and rdi, 0x0FFF
```

Decodage en cascade

Certaines familles d'opcodes (0x0, 0x8, 0xE, 0xF) necessitent un **second niveau de decodage**. Par exemple, les opcodes `0x8XY_` sont 9 instructions differentes selon le dernier nibble (0, 1, 2, 3, 4, 5, 6, 7, E). Le dispatcher doit donc verifier deux niveaux pour identifier l'instruction correcte.

5.3 Phase 3 : Execute

Apres le decodage, le dispatcher appelle la fonction correspondante. Chaque opcode est implemente dans un fichier separe, ce qui facilite la maintenance. Le dispatcher renvoie 1 en cas de succes et 0 si l'opcode est inconnu.

5.4 Boucle principale

La boucle principale dans `main.s` orchestre l'ensemble du cycle. A chaque frame (60 FPS), elle execute **10 instructions** puis met a jour l'affichage :

```
.main_loop:
    •••
    mov r12, 10

.cpu_cycle:
    call fetch_opcode       ; Lire l'instruction
    test rax, rax
    jz .render_frame        ; Si invalide, passer au rendu

    mov rdi, rax
    call execute_opcode     ; Decoder et executer

    dec r12
    jnz .cpu_cycle          ; Boucler 10 fois

.render_frame:
    call render_display     ; Afficher + clavier + timers
    jmp .main_loop
```

Cela donne une vitesse de **600 instructions par seconde** (10 x 60 FPS), ce qui est une approximation raisonnable de la vitesse d'execution originale du CHIP-8.

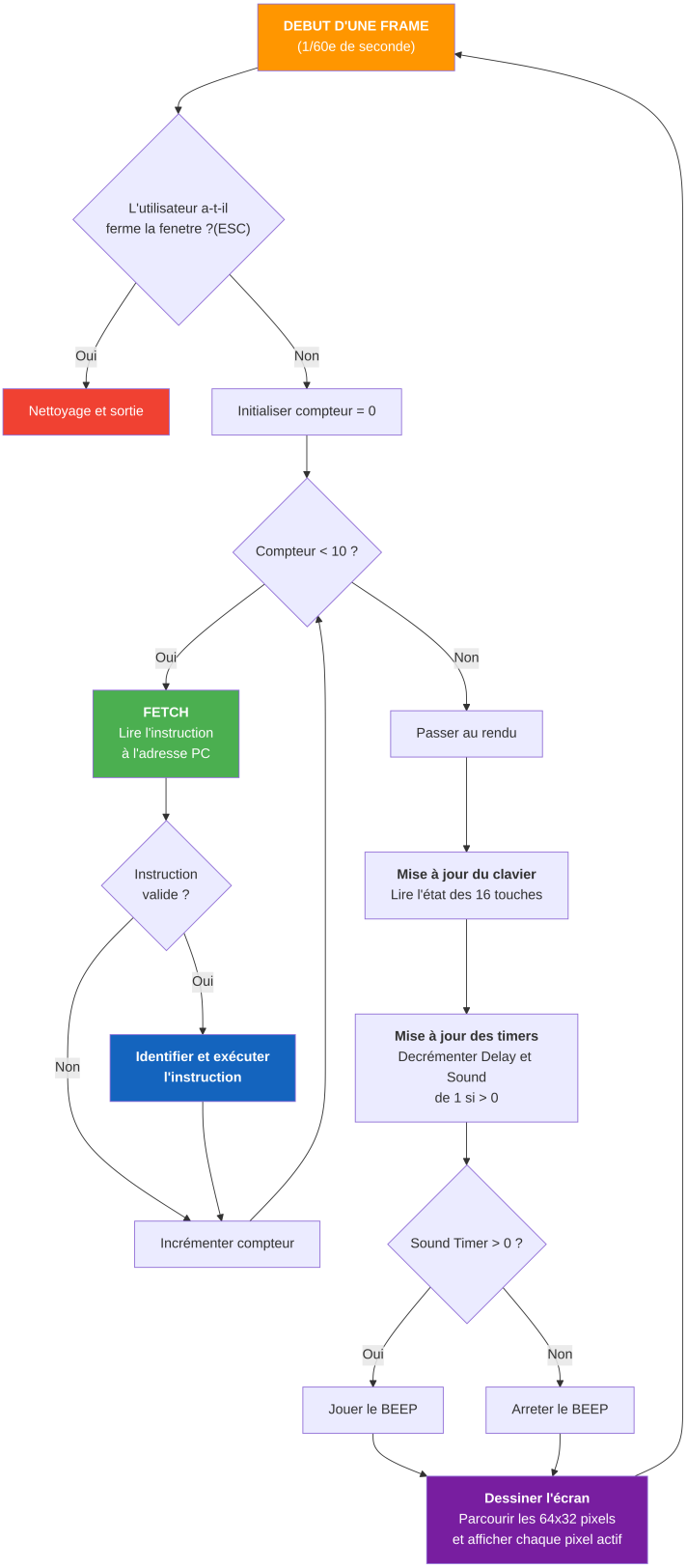

Fig. 2. – Schema detaille d'une frame : cycle CPU, clavier, timers et rendu

6 Workflow complet d'execution

Le schema ci-dessous presente le flux d'execution complet de l'emulateur, depuis le lancement en ligne de commande jusqu'a la fermeture :

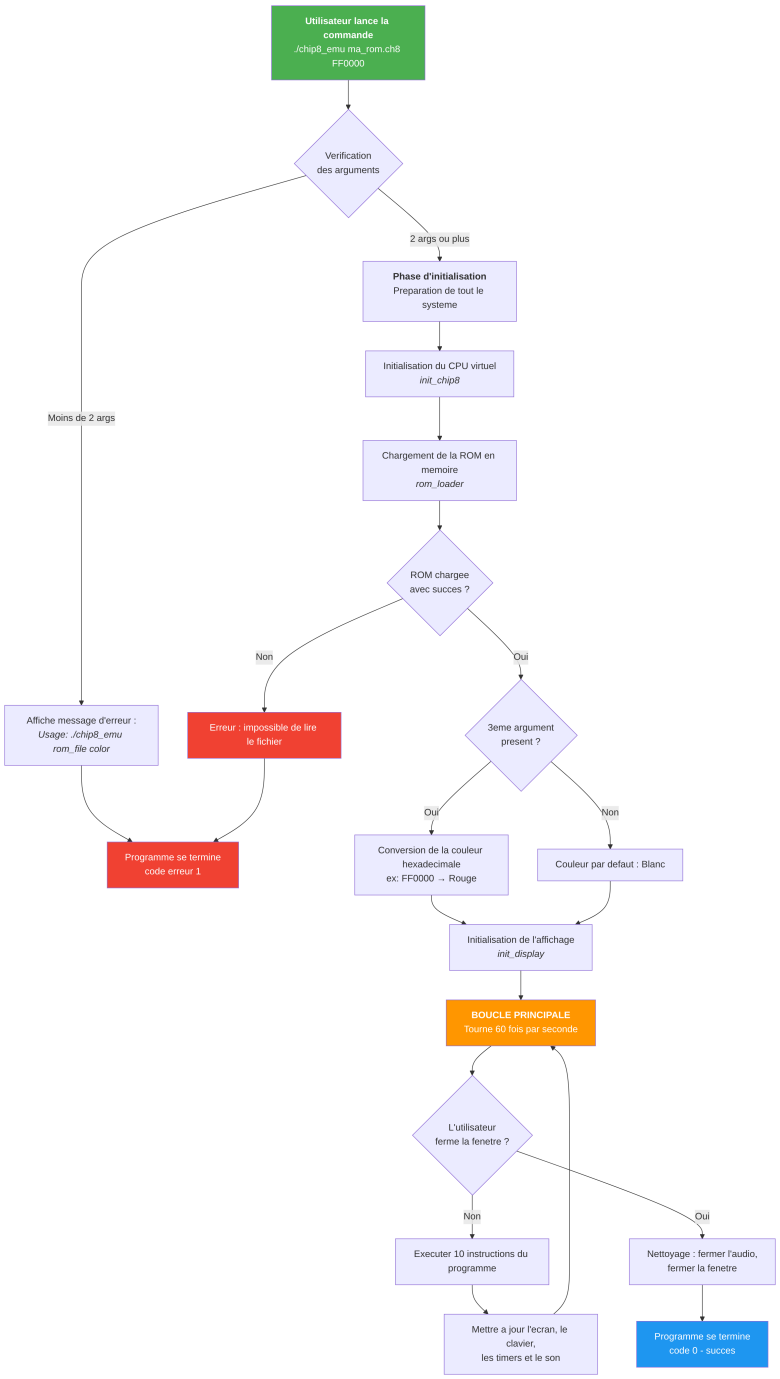

Fig. 3. – Diagramme complet du flux d'execution de l'emulateur

Le workflow se decompose en plusieurs phases :

1. **Validation des arguments** : verification qu'au moins un fichier ROM est passe en parametre
2. **Initialisation** : mise a zero de la memoire, des registres, chargement du fontset
3. **Chargement ROM** : lecture du fichier `.ch8` et ecriture dans `MEMORY` a partir de `0x200`
4. **Couleur optionnelle** : si un 3e argument est present, parsing de la couleur hexadecimale
5. **Ouverture de la fenetre** : creation de la fenetre 64x32 avec Raylib, initialisation audio
6. **Boucle principale** : 60 fois par seconde, execution de 10 instructions + rendu
7. **Nettoyage** : fermeture de la fenetre et liberation des ressources audio

7 Les opcodes du CHIP-8

L'ensemble des opcodes du CHIP-8 est implemente. Ils sont regroupes par famille selon leur premier nibble.

7.1 Opcodes de controle de flux

Ces opcodes sont comparables aux instructions de saut (`jmp`, `call`, `ret`) en assembleur.

Opcode	Nom	Action	Equiv. asm/C
`00E0`	Clear Screen	Efface l'ecran (256 octets a zero)	–
`00EE`	Return	Retour de sous-routine : PC = STACK[–SP]	`ret`
`1NNN`	Jump	Saut inconditionnel : PC = NNN	`jmp NNN`
`2NNN`	Call	Appel de sous-routine : STACK[SP++] = PC, PC = NNN	`call NNN`
`BNNN`	Jump V0	Saut avec offset : PC = V0 + NNN	`jmp [rax+NNN]`

Tableau 5. – Opcodes de controle de flux

7.2 Opcodes conditionnels (Skip)

Les instructions de « skip » sont le mecanisme de branchement conditionnel du CHIP-8. Elles n'ont pas de label de destination : elles sautent simplement l'instruction suivante (PC += 2) si la condition est vraie.

Opcode	Nom	Condition de saut	Equiv. x86
`3XNN`	Skip if EQ	Sauter si VX == NN	`cmp + je`
`4XNN`	Skip if NEQ	Sauter si VX != NN	`cmp + jne`
`5XY0`	Skip if EQ reg	Sauter si VX == VY	`cmp + je`
`9XY0`	Skip if NEQ reg	Sauter si VX != VY	`cmp + jne`
`EX9E`	Skip if key	Sauter si la touche VX est enfoncee	–
`EXA1`	Skip if not key	Sauter si la touche VX n'est pas enfoncee	–

Tableau 6. – Opcodes conditionnels

7.3 Opcodes registre

Ces opcodes manipulent les registres avec des valeurs immediates, de maniere similaire aux instructions `mov` et `add` en assembleur.

Opcode	Nom	Action	Equiv. x86
`6XNN`	Set VX	VX = NN	`mov reg, imm`
`7XNN`	Add VX	VX += NN (sans carry)	`add reg, imm`
`ANNN`	Set I	I = NNN	`mov reg, imm`
`CXNN`	Random	VX = random() AND NN	–

Tableau 7. – Opcodes registre/immediat

Comparaison `6XNN` / `mov`

L'opcode `6XNN` est le `mov` du CHIP-8 : il charge une valeur immediate dans un registre. Par exemple, `6A05` signifie « mettre la valeur 5 dans le registre VA », exactement comme `mov al, 5` en x86.

7.4 Opcodes arithmetiques et logiques (famille 8XY_)

La famille `0x8XY_` regroupe toutes les operations entre deux registres. Le dernier nibble determine l'operation :

Opcode	Nom	Action	Equiv. x86
`8XY0`	Set	VX = VY	`mov reg, reg`
`8XY1`	OR	VX = VX | VY	`or reg, reg`
`8XY2`	AND	VX = VX & VY	`and reg, reg`
`8XY3`	XOR	VX = VX ^ VY	`xor reg, reg`
`8XY4`	ADD	VX += VY (VF = carry)	`add reg, reg`
`8XY5`	SUB	VX -= VY (VF = not borrow)	`sub reg, reg`
`8XY6`	SHR	VX >>= 1 (VF = bit perdu)	`shr reg, 1`
`8XY7`	SUBN	VX = VY - VX (VF = not borrow)	–
`8XYE`	SHL	VX <<= 1 (VF = bit perdu)	`shl reg, 1`

Tableau 8. – Opcodes arithmetiques et logiques

Voici un exemple d'implementation de l'addition avec carry (`8XY4`) :

```
op_add_vx_vy:
    •••
    lea rax, [rel REGISTERS]
    movzx rcx, byte [rax + rdi]  ; VX
    movzx rdx, byte [rax + rsi]  ; VY

    add cl, dl                   ; Addition 8 bits
    •••
    mov byte [rax + rdi], cl     ; Stocker le resultat
    mov byte [rax + 0xF], bl     ; VF = carry flag
    •••
```

7.5 Opcodes memoire et timers (famille FX___)

Opcode	Nom	Action
FX07	Get Delay	VX = valeur du delay timer
FX0A	Wait Key	Attend une touche, stocke dans VX (bloquant)
FX15	Set Delay	delay timer = VX
FX18	Set Sound	sound timer = VX (declenche le beep)
FX1E	Add I	I += VX
FX29	Font	I = adresse du sprite font pour le chiffre VX
FX33	BCD	Stocke la representation BCD de VX dans M[I], M[I+1], M[I+2]
FX55	Store	Copie V0..VX dans MEMORY[I]..MEMORY[I+X]
FX65	Load	Copie MEMORY[I]..MEMORY[I+X] dans V0..VX

Tableau 9. – Opcodes de la famille FX

L'opcode `FX33` (BCD – Binary Coded Decimal) est notable : il convertit un nombre 8 bits en ses chiffres decimaux individuels. Par exemple, si VX = 123, il stocke 1, 2, 3 dans trois adresses memoire consecutives. Cela permet d'afficher des scores ou des compteurs a l'ecran en utilisant les sprites du fontset.

8 Systeme d'affichage

8.1 Configuration de l'ecran

L'ecran du CHIP-8 fait **64 x 32 pixels** monochromes. Chaque pixel est soit allume, soit eteint. Dans l'emulateur, chaque pixel est agrandi par un facteur **30** pour obtenir une fenetre plus grande pour un écrans standard d'aujourd'hui.

```
#define CHIP8_WIDTH 64
#define CHIP8_HEIGHT 32
#define SCALE 30
#define WINDOW_WIDTH (CHIP8_WIDTH * SCALE)
#define WINDOW_HEIGHT (CHIP8_HEIGHT * SCALE)
```

8.2 Representation du buffer d'affichage

Le buffer `DISPLAY` fait 256 octets et encode les 2 048 pixels (64 x 32) a raison de **1 bit par pixel** :

```
// Pour un pixel aux coordonnees (x, y) :
int byte_index = (y * 64 + x) / 8;
int bit_index = 7 - (x % 8);
bool pixel_on = (DISPLAY[byte_index] >> bit_index) & 1;
```

- `byte_index` : quel octet du buffer contient ce pixel
- `bit_index` : quel bit dans cet octet (le bit 7 est le plus a gauche)

8.3 Rendu avec Raylib

A chaque frame, la fonction `render_display` parcourt les 2 048 pixels et dessine un carre colore pour chaque pixel allume :

```
void render_display(void) {
    update_keypad();
    update_timers();

    BeginDrawing();
    ClearBackground(BLACK);

    for (int y = 0; y < CHIP8_HEIGHT; y++) {
        for (int x = 0; x < CHIP8_WIDTH; x++) {
            int byte_index = (y * CHIP8_WIDTH + x) / 8;
            int bit_index = 7 - (x % 8);

            if ((DISPLAY[byte_index] >> bit_index) & 1) {
                DrawRectangle(x * SCALE, y * SCALE,
                              SCALE, SCALE, pixel_color);
            }
        }
    }
    EndDrawing();
}
```

La couleur des pixels est configurable via la variable `pixel_color` (voir la section sur les parametres de lancement).

9 Systeme audio

Le CHIP-8 possede un systeme audio minimaliste : un unique **beep** a 440 Hz (la note La4, la reference standard de l'accordage musical).

Le son est genere par une **onde sinusoidale** calculee mathematiquement :

```
static void init_beep_sound(void) {
    int sample_count = (int)(BEEP_SAMPLE_RATE * BEEP_DURATION);
    // sample_count = 44100 * 0.1 = 4410 echantillons

    short *samples = (short *)wave.data;
    for (int i = 0; i < sample_count; i++) {
        float t = (float)i / BEEP_SAMPLE_RATE;
        samples[i] = (short)(sinf(2.0f * PI * 440.0f * t) * 32000);
    }
}
```

Le son est joue tant que le `SOUND_TIMER` est superieur a 0. Il est arrete des que le timer atteint 0.

10 Systeme d'input (clavier)

10.1 Mapping du clavier

Le CHIP-8 possede un clavier hexadecimal de 16 touches (0-F). Dans l'emulateur, ces touches sont mappees sur un clavier standard :

Clavier CHIP-8	Clavier physique
1 2 3 C 4 5 6 D 7 8 9 E A 0 B F	1 2 3 4 Q W E R A S D F Z X C V

Tableau 10. – Correspondance entre le keypad CHIP-8 et le clavier standard

10.2 Implementation

La mise a jour du clavier se fait a chaque frame (60 fois par seconde) via la fonction `update_keypad` :

```
static const int key_map[16] = {
    KEY_X,      // 0     KEY_ONE,   // 1
    KEY_TWO,    // 2     KEY_THREE, // 3
    KEY_Q,      // 4     KEY_W,     // 5
    KEY_E,      // 6     KEY_A,     // 7
    KEY_S,      // 8     KEY_D,     // 9
    KEY_Z,      // A     KEY_C,     // B
    KEY_FOUR,   // C     KEY_R,     // D
    KEY_F,      // E     KEY_V      // F
};

void update_keypad(void) {
    for (int i = 0; i < 16; i++)
        KEYPAD[i] = IsKeyDown(key_map[i]) ? 1 : 0;
}
```

Le tableau `KEYPAD` est defini en assembleur comme une variable globale de 16 octets, accessible depuis le C via `extern`.

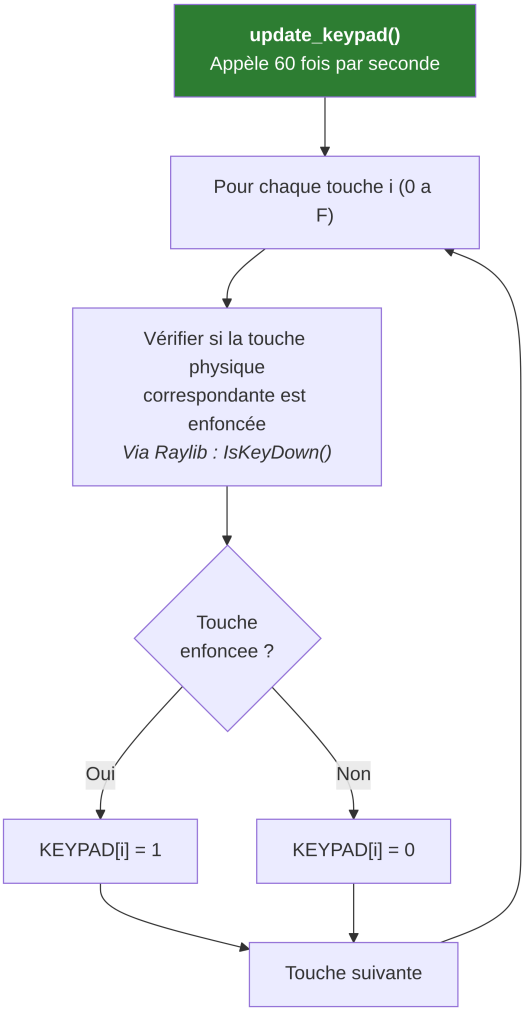

Fig. 4. – Schema de la mise a jour du clavier

11 Systeme de timers

Le CHIP-8 possede deux timers de 8 bits qui decrementent a **60 Hz** (une fois par frame) :

- **Delay Timer** : utilise par les programmes pour creer des delais (ex : tempo de jeu). Lisible via l'opcode `FX07`, modifiable via `FX15`.
- **Sound Timer** : quand il est superieur a 0, le beep retentit. Modifiable via `FX18`. Quand il atteint 0, le son s'arrete automatiquement.

```c
void update_timers(void) {
    if (DELAY_TIMER > 0)
        DELAY_TIMER--;

    if (SOUND_TIMER > 0) {
        play_beep();
        SOUND_TIMER--;
    } else {
        stop_beep();
    }
}
```

Cette fonction est appelee depuis `render_display()` a chaque frame, ce qui garantit une decrementation a exactement 60 Hz grace au `SetTargetFPS(60)` de Raylib.

12 Lancement et parametres

12.1 Utilisation en ligne de commande

L'emulateur se lance depuis le terminal avec la syntaxe suivante :

```
./chip8_emu <fichier_rom> [couleur_hex]
```

- `<fichier_rom>` : **obligatoire** – chemin vers un fichier ROM au format `.ch8`
- `[couleur_hex]` : **optionnel** – couleur des pixels au format hexadecimal RRGGBB

12.1.1 Exemples d'utilisation

```
# Lancement simple (pixels blancs par defaut)
./chip8_emu roms/games/tetris.ch8

# Pixels rouges
./chip8_emu roms/games/space_invaders.ch8 FF0000

# Pixels verts
./chip8_emu roms/test/test_ibm.ch8 00FF00
```

```
# Pixels bleu clair
./chip8_emu roms/games/Tron.ch8 3B82F6

# Couleur personnalisee (orange)
./chip8_emu roms/games/tetris.ch8 F28C38
```

12.2 Validation des arguments

Le programme verifie le nombre d'arguments et affiche un message d'usage en cas d'erreur :

```
main:
    mov r14, rdi            ; r14 = argc
    mov r15, rsi            ; r15 = argv

    ; Verifier qu'on a au moins 2 arguments
    cmp r14, 2
    jl .usage_error
```

12.3 Couleur personnalisee en hexadecimal

12.3.1 Fonctionnement du format RRGGBB

La couleur est specifiee au format hexadecimal **RRGGBB** (6 caracteres), ou chaque paire de caracteres represente un canal de couleur sur 8 bits (0-255) :

Hex	R	G	B	Couleur
`FF0000`	255	0	0	Rouge pur
`00FF00`	0	255	0	Vert pur
`0000FF`	0	0	255	Bleu pur
`FFFFFF`	255	255	255	Blanc (defaut)
`F28C38`	242	140	56	Orange
`3B82F6`	59	130	246	Bleu clair
`00000`	0	0	0	Noir (invisible !)

Tableau 11. – Exemples de couleurs hexadecimales

Attention

Attention : si vous specifiez `000000` (noir), les pixels seront invisibles sur le fond noir de l'emulateur.

12.3.2 Parsing de la couleur en assembleur

La conversion de la chaine hexadecimale en valeur 32 bits est effectuee entierement en assembleur dans la fonction `parse_hex_color` :

```
parse_hex_color:
    •••
    xor rax, rax

.parse_loop:
    movzx rbx, byte [rdi]     ; Lire un caractere
    test bl, bl                ; Fin de chaine (\0) ?
    jz .done

    shl rax, 4                ; resultat *= 16 (decaler de 4 bits)

    ; Conversion du caractere ASCII en valeur 0-15
    cmp bl, '0'
    jl .done
    cmp bl, '9'
    jle .digit                 ; '0'-'9' -> soustraire '0'

    cmp bl, 'A'
    jl .check_lower
    cmp bl, 'F'
    jle .upper_letter          ; 'A'-'F' -> soustraire 'A', ajouter 10

.check_lower:
    cmp bl, 'a'
    jl .done
    cmp bl, 'f'
    jg .done
    sub bl, 'a'               ; 'a'-'f' -> soustraire 'a', ajouter 10
    add bl, 10
    jmp .add_digit

.upper_letter:
    sub bl, 'A'
    add bl, 10
    jmp .add_digit

.digit:
    sub bl, '0'               ; Chiffre : valeur = caractere - '0'

.add_digit:
    or al, bl                 ; Ajouter le nibble au resultat
    inc rdi                   ; Caractere suivant
    jmp .parse_loop
```

Deroulement pour "F23838"

Voici le deroulement pas a pas pour la couleur `F23838` :

- “F” : resultat = 0x0 << 4 | 0xF = **0xF**
- “2” : resultat = 0xF << 4 | 0x2 = **0xF2**
- “3” : resultat = 0xF2 << 4 | 0x3 = **0xF23**
- “8” : resultat = 0xF23 << 4 | 0x8 = **0xF238**
- “3” : resultat = 0xF238 << 4 | 0x3 = **0xF2383**
- “8” : resultat = 0xF2383 << 4 | 0x8 = **0xF23838**

Le resultat final `0xF23838` est ensuite decompose en C : R=0xF2 (242), G=0x38 (56), B=0x38 (56).

13 Compilation et build

13.1 Chaine de compilation

Le projet utilise un Makefile qui orchestre la compilation en 3 etapes :

1. **NASM** : assembler les fichiers `.s` en fichiers objet `.o` (format ELF64)
2. **GCC** : compiler les fichiers `.c` en fichiers objet `.o`
3. **GCC (linker)** : lier tous les `.o` ensemble avec les bibliotheques necessaires

```
ASM = nasm
ASMFLAGS = -f elf64 -g

CC = gcc
CFLAGS = -Wall -Wextra -g -I$(INC_DIR)

LDFLAGS = -lraylib -lGL -lm -lpthread -ldl -lrt -lX11
```

13.2 Commandes disponibles

Commande	Action
`make`	Compiler le projet
`make clean`	Supprimer les fichiers objet
`make fclean`	Supprimer les fichiers objet et l’executable
`make re`	Recompilation complete
`make test`	Compiler et lancer l’emulateur
`make debug`	Compiler et lancer avec GDB

Tableau 12. – Cibles du Makefile

14 Demarche de developpement

Le developpement a suivi une approche **incrementale**, chaque etape validant la precedente avant de passer a la suivante.

14.1 Etape 1 : Initialisation du CPU

Le premier objectif etait de pouvoir creer et initialiser un « CPU » virtuel en assembleur : definir les variables globales (memoire, registres, pile, PC, timers) dans la section `.bss` et ecrire la fonction `init_chip8` qui met tout a zero et charge le fontset.

14.2 Etape 2 : Chargement de la ROM

L'objectif suivant etait de charger un fichier ROM dans la memoire du CPU virtuel. Cela a necessite l'utilisation des syscalls Linux (`open`, `read`, `close`). La validation consistait a verifier (avec GDB) que les octets de la ROM etaient bien presents dans `MEMORY` a partir de l'adresse `0x200`.

14.3 Etape 3 : Fetch des opcodes

Une fois la ROM chargee, il fallait implementer le mecanisme de **fetch** : lire 2 octets a l'adresse PC, les combiner en big-endian, et avancer le PC de 2. La fonction `print_opcode` (debug) a ete ecrite pour afficher chaque opcode fetche en hexadecimal et verifier visuellement que la lecture etait correcte.

14.4 Etape 4 : Afficher le logo IBM

C'est l'etape **charniere** du projet. La ROM `test_ibm.ch8` est le « Hello World » de l'emulation CHIP-8 : elle utilise uniquement les opcodes `00E0` (clear screen), `6XNN` (set register), `ANNN` (set I) et `DXYN` (draw sprite) pour afficher le logo IBM a l'ecran.

Pour y parvenir, il a fallu implementer :

- Le dispatcher minimal (4 opcodes)
- L'opcode de dessin `DXYN`
- L'integration avec Raylib pour le rendu graphique
- La boucle principale avec le timing a 60 FPS

Voir le logo IBM s'afficher a l'ecran a ete la premiere validation visuelle que l'emulateur fonctionnait.

14.5 Etape 5 : Emulateur complet

Apres le logo IBM, les opcodes restants ont ete implementes un par un. Les sous-systemes ont ete ajoutes progressivement :

- Clavier (input.c + opcodes EX9E/EXA1/FX0A)
- Timers (timers.c + opcodes FX07/FX15/FX18)
- Audio (audio.c + integration avec le sound timer)

Les jeux (Tetris, Space Invaders, Tron) ont servi de tests d'integration finale.

14.6 Etape 6 : Fonctionnalites supplementaires

Une fois l'emulateur fonctionnel, des ameliorations ont ete ajoutees :

- **Couleur personnalisable** : parsing hexadecimal en assembleur + application cote C
- **Generation pseudo-aleatoire** : implementation d'un LFSR (Linear Feedback Shift Register) pour l'opcode `CXNN`
- **Gestion d'erreurs** : verification des arguments, du chargement ROM, des limites du PC

15 Conclusion

Ce projet a permis de concevoir un emulateur CHIP-8 fonctionnel en assembleur x86-64, capable d'executer les opcodes de la specification, avec un affichage graphique, un systeme audio, une gestion du clavier et des timers.

15.1 Perspectives futures

- Ajouter un mode **debug pas-a-pas** avec affichage de l'etat des registres en temps reel
- Supporter le **rechargement a chaud** de ROM sans redemarrer l'emulateur

16 References

1. Cowgod's CHIP-8 Technical Reference : http://devernay.free.fr/hacks/chip8/C8TECH10.HTM
2. CHIP-8 Wikipedia : https://en.wikipedia.org/wiki/CHIP-8
3. Tobias V. Langhoff, Guide to making a CHIP-8 emulator : https://tobiasvl.github.io/blog/write-a-chip-8-emulator/
4. Documentation Raylib : https://www.raylib.com/
5. Documentation NASM : https://www.nasm.us/doc/
6. Intel x86-64 Architecture Manual : https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html