eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Desarrollo de un emulador hardware para el lenguaje interpretado CHIP-8.

Autor/a

Imanol López Olivas

2021

eman ta zabal zazu

Universidad del País Vasco | Euskal Herriko Unibertsitatea

INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Ingeniería de Computadores

Trabajo de Fin de Grado

Desarrollo de un emulador hardware para el lenguaje interpretado CHIP-8.

Autor/a

Imanol López Olivas

Directore/a(s)

José Pascual

Resumen

Un emulador es un software que permite ejecutar programas informáticos en una plataforma diferente para la que fueron desarrollados originalmente. Según la tesis de Church-Turing cualquier ambiente funcional puede ser emulado dentro de cualquier otro, aunque estas plataformas puedan constar de diferentes características, como un sistema operativo o set de instrucciones diferente.

Uno de los usos más comunes para dar lugar al desarrollo de los emuladores de hardware es lograr la capacidad de poder revivir la experiencia original de ciertos sistemas que ya no estén en producción. Además, resulta más sencillo y práctico para los usuarios y programadores el hecho de poder ejecutar programas escritos para estas máquinas en cualquier computadora convencional, arquitectura o sistema operativo.

Este proyecto tiene como propósito desarrollar un emulador hardware del lenguaje de programación interpretado `CHIP-8` con el fin de imitar su funcionamiento en sistemas con arquitecturas más recientes como `x86-64` y los distintos entornos que podemos encontrar hoy en día. Este lenguaje fue inicialmente diseñado y creado por Joseph Weis-becker en 1977 con el fin de facilitar el desarrollo de programas interactivos para los microcomputadores COSMAC VIP y Telmac 1800 de 8 bits.

Describiendo los recursos presentes en el hardware original como el procesador, los diferentes registros y el intérprete de instrucciones como una estructura de datos en `C++` es posible definir una maquina virtual imitando los sistemas mencionados anteriormente. Además, el uso de plataformas *Simple DirectMedia Layer* para la gestión del procesamiento de imagen y gestión de las entradas del teclado permite ejecutar el emulador en cualquier plataforma que soporte esta librería, aumentando así su portabilidad.

Índice general

Resumen I

Índice general III

Índice de figuras VII

Índice de tablas IX

1. Introducción 1

2. Objetivos del proyecto 3

3. Análisis de las características de la máquina virtual 5

3.1. CPU 5

3.2. Memoria Principal 6

3.3. Registros 7

3.4. Pila 7

3.5. Puntero de pila 8

3.6. *Program Counter* 9

3.7. Temporizadores 9

3.8. Gráficos 9

3.8.1. Dibujado 10

3.9. Teclado 12

4. Conjunto de Instrucciones 13

4.1. Instrucciones . . . 13

4.1.1. Instrucciones aritméticas . . . 14

4.1.2. Instrucciones de salto . . . 16

4.1.3. Instrucciones a memoria . . . 17

4.1.4. Instrucciones de movimiento de bits . . . 20

4.1.5. Otras instrucciones . . . 21

5. Ciclos de ejecución del procesador e inicialización 23

5.1. Lectura de ROM e inicialización . . . 23

5.2. Ciclo del procesador . . . 24

5.2.1. Captación . . . 24

5.2.2. Descodificación . . . 25

5.2.3. Ejecución . . . 26

6. *Simple DirectMedia Layer* 27

6.1. SDL como plataforma . . . 27

6.2. Renderizado de gráficos . . . 28

6.3. Reproducción de sonido . . . 29

6.4. Eventos de entrada . . . 30

7. Pruebas y depuración 31

7.1. ROMs de pruebas unitarias . . . 31

7.1.1. Prueba unitaria 1 . . . 31

7.1.2. Prueba unitaria 2 . . . 32

7.2. Depurador . . . 34

8. Planificación del proyecto **35**
8.1. Tareas principales 35
8.2. Estimación de tiempo de las distintas tareas 36
8.3. Plan de riesgo general del proyecto 38
8.4. Análisis de las desviaciones y problemas generados 38

9. Conclusiones **41**
9.1. Trabajo futuro 42

Anexos

A. Instalación **45**
A.1. Dependencias 45
A.2. Compilación 46
A.3. Ejecución 46

Bibliografía **47**

Índice de figuras

1.1. COSMAC VIP y Telmac 1800 8-bit 2

3.1. Mapa de la memoria principal. 6

3.2. Registros disponibles para la CPU. 7

3.3. Pila de 16 niveles de 12 bit. 8

3.4. Puntero de pila. 8

3.5. Operación XOR entre dos figuras. 10

3.6. Disposición original del teclado hexadecimal. 12

4.1. Identificación y descodificado de la instrucción 8xy1 14

4.2. Operación BCD. 21

5.1. Ciclo del procesador. 24

5.2. Proceso de captación. 25

5.3. Proceso de descodificación. 26

6.1. Capas de abstracción de SDL en los distintos sistemas operativos. 28

6.2. Función *SDL_UpdateTexture()* 28

6.3. Representación gráfica de la onda sinusoidal. 29

6.4. Redistribución de las entradas del teclado. 30

7.1. Primera ROM utilizada para la prueba de unidades. 32

7.2. Segunda ROM utilizada para la prueba de unidades. 33

7.3. Emulador ejecutando una ROM de *Breakout* [Carmelo Cortez, 1979] con el depurador activo. 34

Índice de tablas

3.1. Representación binaria de los sprites predeterminados. 11

8.1. Estimación del tiempo dedicado de las distintas tareas. 37

8.2. Periodo de realización de las tareas en el tiempo. 37

1. CAPÍTULO

Introducción

El desarrollo de intérpretes y emuladores de ciertos sistemas o conjuntos de hardware es siempre un tema que despierta interés incluso entre los programadores más experimentados. Como idea, es un proyecto que mezcla diferentes aspectos que forman parte de lo que define una maquina virtual y la emulación del hardware original y la creación de un sistema que pueda leer y reproducir programas escritos para ese entorno.

El `CHIP-8` es un lenguaje de programación interpretado de bajo nivel que se usa como la especificación de una maquina virtual orientada a la ejecución de programas informáticos desarrollados para este entorno. Para ello se hace uso de un intérprete de instrucciones, programa que directamente descodifica y hace posible la ejecución de instrucciones escritas para un lenguaje específico, de alto o bajo nivel, sin necesidad de que haya sido compilado previamente.

El conjunto de instrucciones de este intérprete es sencillo y reducido, teniendo un máximo de 34 instrucciones funcionales basadas en códigos hexadecimales. Por ello, es adecuado para máquinas con poca memoria y capacidad de procesamiento. Este lenguaje ha recibido diferentes niveles de éxito a lo largo del tiempo, adaptándose a multitud de combinaciones de hardware y diferentes sistemas operativos, y hoy en día consta de una comunidad medianamente grande de personas que escriben aplicaciones interactivas y otros programas para ella.

Fue inicialmente diseñado y desarrollado por Joseph Weisbecker en 1977 en los laboratorios de la empresa RCA con el fin de permitir el desarrollo de programas interactivos para el microcomputador COSMAC VIP, aunque se utilizó también en sistemas posteriores

como la Telmac 1800 8-bit a mediados de la década de los setenta.

Figura 1.1: COSMAC VIP y Telmac 1800 8-bit

Los recursos que estas maquinas utilizaban para funcionar era ciertamente limitado: disponían de una memoria de `2KB` (2048 bytes) expansibles a `4KB`, de los cuales 512 eran utilizados para almacenar el sistema operativo, varios registros de 8 bits, una pantalla capaz de representar un total de 64x32 píxeles y un teclado hexadecimal, entre otros recursos. Además, disponía de un depurador integrado que permitía visualizar los valores de los registros y la pila de instrucciones. Los programas disponibles en su lanzamiento aparecían listados en el manual y debían ser introducidos en la memoria instrucción por instrucción por el usuario.

Se considera una de las máquinas más sencillas de emular, por lo tanto, gran parte de los desarrolladores que están interesados en el desarrollo de emuladores e intérpretes comienzan con `CHIP-8`, diseñando un programa informático que imita el diseño interno y la funcionalidad del hardware original. Como resultado, permite a los usuarios ejecutar programas diseñados para este sistema específico en una arquitectura totalmente diferente, como pueden ser las computadoras personales.

Este proyecto abordará los diferentes puntos que constituyen el desarrollo de un intérprete y la emulación de los recursos que componen una maquina virtual, siendo esta basada en el `CHIP-8`. El programa que se desarrollará tendrá como objetivo definir las características que definen una maquina virtual `CHIP-8` (su hardware) y diseñar un prototipo capaz interpretar y descodificar programas escritos para este entorno.

2. CAPÍTULO

Objetivos del proyecto

El objetivo principal de este proyecto es el estudio del lenguaje interpretado `CHIP-8` y los distintos recursos que forman parte del hardware del sistema a emular para hacer posible la construcción de una maquina virtual capaz de ejecutar programas desarrollados para este entorno.

Como primer paso se debe realizar la planificación del proyecto, dividiéndolo en lotes de trabajos y tareas. Asimismo, se debe diseñar un plan de riesgos para cualquier posible percance que pueda ocurrir durante el avance del proyecto.

A continuación, se estudiarán y analizarán el funcionamiento de las distintas partes que conforman la maquina virtual a desarrollar, el ciclo de ejecución general de la máquina y la implementación del set de instrucciones y el lenguaje interpretado `CHIP-8` en `C++`. Asimismo, se buscará la manera de consolidar las distintas partes del proyecto de modo que sea posible escribir un prototipo capaz de ejecutar programas escritos para este lenguaje.

Por otra parte, se realizará una lectura de las distintas funcionalidades que la librería SDL puede ofrecer al desarrollador para la implementación de diferentes características que componen el entorno a imitar, como puede ser la gestión de la imagen y entradas del teclado.

Las fases específicas de este proyecto son las siguientes:

- 1. Planificación general del proyecto y los distintos lotes de tareas.
- 2. Estudio del hardware a emular (memoria principal, memoria gráfica, registros, pila, frecuencia de reloj, etc).
- 3. Estudio del conjunto de instrucciones del intérprete.
- 4. Estudio de la plataforma SDL para el renderizado de imagen, generación de sonido y gestión del teclado.
- 4. Planificación inicial del intérprete, ciclos de procesamiento y el hardware como estructura de datos del programa en `C++`.
- 5. Implementación general del programa.
- 6. Pruebas unitarias del prototipo desarrollado.
- 7. Mejora del prototipo añadiendo nuevas características (depurador de la máquina virtual y carga/descarga del estado del procesador).

3. CAPÍTULO

Análisis de las características de la máquina virtual

Para poder desarrollar la idea del proyecto es necesario conocer primero las distintas partes que forman parte del sistema a interpretar y emular. A continuación, se listarán en este capítulo todos los recursos que estarán presentes en la máquina virtual y se describirán las diferentes características que forman parte del hardware a emular.

3.1. CPU

La unidad de cómputo central o CPU será el recurso utilizado por el intérprete para la lectura de las instrucciones y comandos de entrada y de su posterior descodificación y procesado.

La ejecución de programas de la CPU se dividirá en ciclos de procesador. En cada ciclo se ejecutará una única instrucción y está formado por tres etapas: Captación (*Fetch*), descodificación (*decode*) y ejecución (*execution*), que se explicarán posteriormente (ver Sección 5.2).

La velocidad en la que estos tres pasos se desarrollan será dictado por los ciclos por segundo a los que rinda el procesador. Aunque el procesador de los sistemas que originalmente hacían uso del intérprete funcionaban a una frecuencia determinada, no existe ningún consenso en la comunidad que especifique a que velocidad debería funcionar la unidad de cómputo, por lo tanto este parámetro dependerá del desarrollador del intérprete y de los programas a ejecutar. En este prototipo el usuario podrá especificar el número

de ciclos que el procesador ejecutará por segundo (para el uso general del programa, ver Anexo A.2).

3.2. Memoria Principal

El intérprete del `CHIP-8` originalmente fue creado con el propósito de poder operar en maquinas que disponían de `4K bytes` de memoria. Estos sistemas disponían solamente de 4096 espacios de memoria, de 8 bits cada uno, una cifra muy pequeña para los estándares de hoy en día. Si se representa de forma hexadecimal, el espacio de memoria estaría representado desde la dirección `0x000` a `0xFFF`.

Este espacio de memoria estará dividido en tres segmentos (Figura 3.1):

- `0x000-0x1FF`: Originalmente, este espacio de memoria era el reservado por el propio intérprete para su funcionamiento. Dado que el intérprete será escrito en un nivel de abstracción más alto, no hará falta hacer uso de la mayoría de direcciones localizadas en este espacio.

- `0x050-0x0A0`: En este espacio de memoria se guardarán los *sprites* (ver Sección 3.8.1) que conformarán los diferentes caracteres de los que ciertas ROM harán uso, por lo tanto, es necesario definirlos.

- `0x200-0xFFF`: Se utilizará el resto de espacios de la memoria para almacenar las instrucciones de entrada y datos de programa.

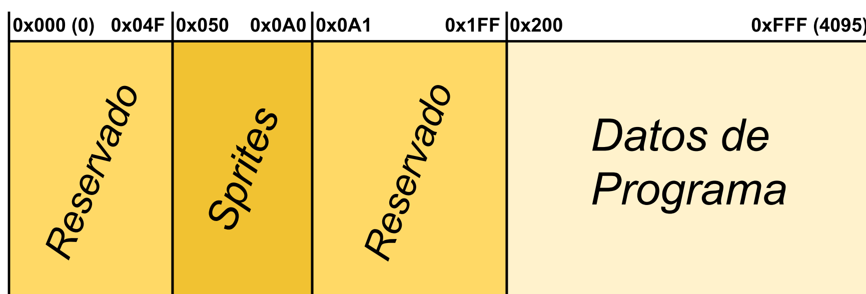

Figura 3.1: Mapa de la memoria principal.

3.3. Registros

Un registro es un recurso que encontramos en la mayoría de procesadores que permite a este último almacenar y acceder a cierta información de una forma inmediata. Muchas de las operaciones que realizará la unidad de cómputo central serán sobre los registros, como cargar información desde la memoria principal a los registros, operar con los valores disponibles en estos registros y escribir en la memoria principal.

El intérprete consta de 16 registros de 8 bits cada uno, `V0` hasta `VF`, siendo este último especial ya que se usará como *flag* para almacenar información sobre el resultado de operaciones, por ejemplo, como *carry flag* o detección de colisiones al dibujar en pantalla.

También constará de un registro de índices llamado `I` de 16 bits, en el que se almacenarán direcciones de memoria, usado por varias instrucciones para operar con la memoria.

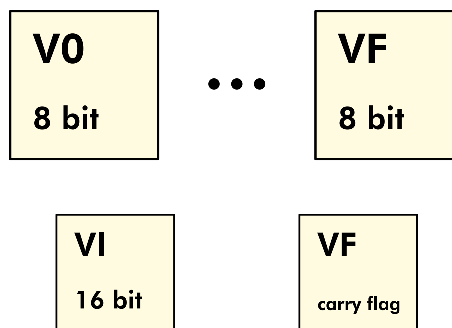

Figura 3.2: Registros disponibles para la CPU.

3.4. Pila

La pila es un recurso utilizado por la CPU para hacer un seguimiento del orden de ejecución de las instrucciones almacenadas en la memoria principal. Cuando se descodifica una instrucción `CALL`, el procesador comenzará a ejecutar instrucciones que se hallen en la dirección especificada, y cuando esa subrutina acabe (instrucción `RET`) la CPU tendrá que ser capaz de regresar a la instrucción siguiente que se encuentra después de la instrucción de llamada. La pila ayudará a almacenar la información necesaria para que la unidad de cómputo regrese a su estado anterior.

En la maquina virtual que se diseñará la pila dispondrá de 16 niveles capaz de guardar 16 PC o direcciones de memoria, de forma que el procesador pueda regresar a la dirección

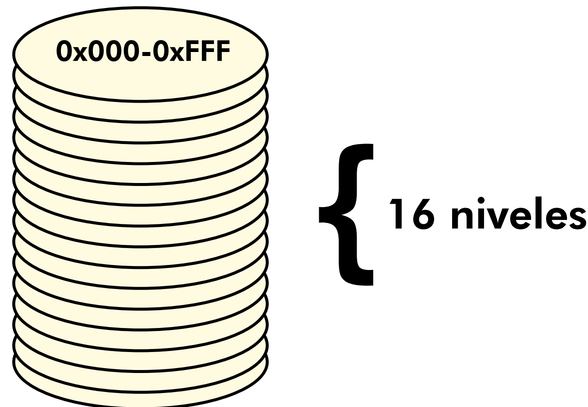

Figura 3.3: Pila de 16 niveles de 12 bit.

anterior (*pop*), o guardar su estado anterior (*push*). Esto permitirá también al procesador llamar a instrucciones `CALL` que se encuentren dentro de otra instrucción `CALL`.

3.5. Puntero de pila

El *Stack Pointer* o puntero de pila representa la localización de la siguiente instrucción a ejecutar dentro de la pila. Como la pila es de tipo LIFO, esa instrucción siempre se encontrará en la parte superior de la pila. Cuando el procesador haga uso de esa dirección para volver a su estado anterior (por ejemplo, cuando finalice de ejecutar una subrutina) se decrementará el puntero, y en el caso de almacenar una instrucción se aumentará este valor.

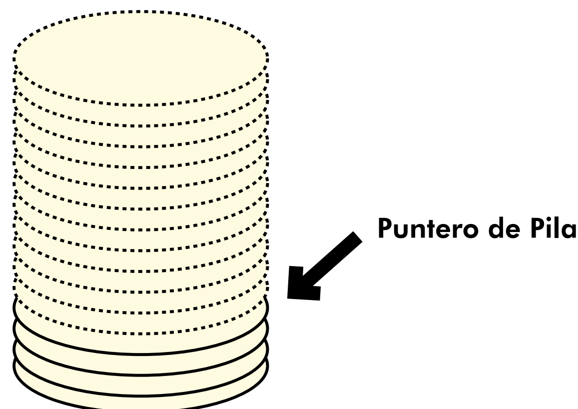

Figura 3.4: Puntero de pila.

El tamaño será de 8 bits para almacenar el valor máximo que puede se puede indexar

en la pila. Esencialmente, es un recurso que ayudará a la unidad de cómputo a hacer un seguimiento del actual tamaño de la pila.

3.6. *Program Counter*

El contador de programa o *Program Counter* es un registro que encontramos en la mayoría de procesadores que almacena la dirección de memoria de la siguiente instrucción que este ejecutará.

Dado que las instrucciones se encuentran en un espacio de la memoria principal, el contador de programa deberá tener el mismo tamaño (12 bits, que corresponde a 4096 posibles direcciones) para retenerlas.

3.7. Temporizadores

El intérprete consta de dos temporizadores (siendo uno de estos usado para el sonido y otro de retardo) utilizados por ciertas instrucciones para operar con la memoria o para realizar ciertas funciones teniendo en cuenta la variable del tiempo, entre ellas la generación de valores aleatorios utilizando este valor como semilla. Estos temporizadores funcionarán a 60Hz y decrementarán su valor (debe ser positivo) 60 veces por segundo. La máquina virtual emitirá un sonido siempre que el valor del temporizador de sonido sea positivo.

3.8. Gráficos

La máquina virtual será capaz de muestrear gráficos en pantalla con una resolución de 64 (altura) por 32 píxeles (anchura). Cada píxel será representado de una forma binaria, pudiendo este estar apagado o encendido, es decir, será de carácter monocromático (blanco y negro).

El intérprete dispone de un *buffer* de memoria gráfica que almacenará el estado de cada píxel.

3.8.1. Dibujado

El dibujado en pantalla se realizará trabajando con los dos únicos estados en los que un píxel pueda encontrarse, realizando una operación bit a bit `XOR` entre el píxel a dibujar y el valor actual de este en la pantalla (para ver la instrucción de dibujado del intérprete, ver Sección 4.1.5).

Al realizar la operación de dibujado aplicando el operando `XOR` sobre un bit que ya esté activado, este se desactivará dado que se están superponiendo ambos valores. Ocurrirá lo contrario cuando se active un píxel en el que su valor previo era igual a 0 (ver Figura 3.5).

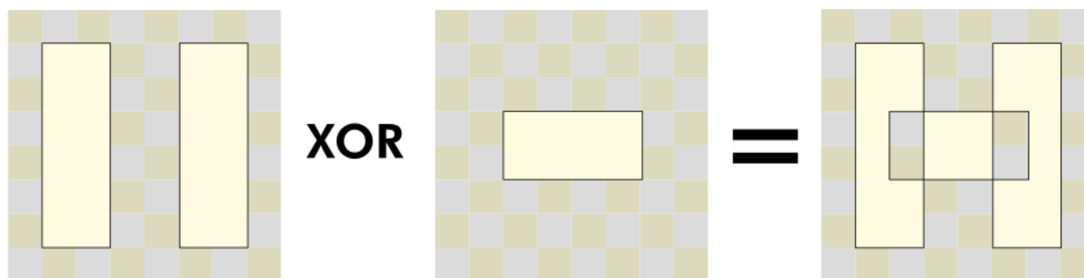

Figura 3.5: Operación XOR entre dos figuras.

Para realizar esta operación el procesador hará uso de los *sprites*. Un sprite es una representación binaria de la imagen deseada, que servirá como dato de entrada para formar distintos iconos y figuras. El desarrollador puede cargar sus propios sprites en la memoria principal gracias a las distintas instrucciones que incorpora el procesador (ver Sección 4.1.3, 4.1.4 y 4.1.3).

Por otro lado, el intérprete debe tener precargados en la memoria distintos sprites por defecto que los programadores utilizarán para mostrar en pantalla determinados caracteres. Existen un total de 16 caracteres, de `0` a `F`. Estos sprites ocupan cinco espacios en la memoria (cinco bytes) y cada valor almacenado es la representación binaria de cada fila que compone la figura, aunque solo se utilizarán los cuatro primeros bits más significativos como valor para las figuras predeterminadas.

A continuación se listan todos los sprites que deben inicializarse en la memoria y el valor de cada fila representado de forma binaria (ver Tabla 3.1):

0	Binario	Hex.
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0

1	Binario	Hex.
*	00100000	0x20
**	01100000	0x60
*	00100000	0x20
*	00100000	0x20
***	01110000	0x70

2	Binario	Hex.
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

3	Binario	Hex.
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

4	Binario	Hex.
* *	10010000	0x90
* *	10010000	0x90
****	11110000	0xF0
*	00010000	0x10
*	00010000	0x10

5	Binario	Hex.
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

6	Binario	Hex.
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0

7	Binario	Hex.
****	11110000	0xF0
*	00010000	0x10
*	00100000	0x20
*	01000000	0x40
*	01000000	0x40

8	Binario	Hex.
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0

9	Binario	Hex.
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
*	00010000	0x10
****	11110000	0xF0

A	Binario	Hex.
****	11110000	0xF0
* *	10010000	0x90
****	11110000	0xF0
* *	10010000	0x90
* *	10010000	0x90

B	Binario	Hex.
***	11100000	0xE0
* *	10010000	0x90
***	11100000	0xE0
* *	10010000	0x90
***	11100000	0xE0

C	Binario	Hex.
****	11110000	0xF0
*	10000000	0x80
*	10000000	0x80
*	10000000	0x80
****	11110000	0xF0

D	Binario	Hex.
***	11100000	0xE0
* *	10010000	0x90
* *	10010000	0x90
* *	10010000	0x90
***	11100000	0xE0

E	Binario	Hex.
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0

F	Binario	Hex.
****	11110000	0xF0
*	10000000	0x80
****	11110000	0xF0
*	10000000	0x80
*	10000000	0x80

Tabla 3.1: Representación binaria de los sprites predeterminados.

3.9. Teclado

Los computadores que originalmente hicieron uso del lenguaje interpretado `CHIP-8` disponían de un teclado hexadecimal de 16 entradas con la siguiente disposición (ver Figura 6.4).

Figura 3.6: Disposición original del teclado hexadecimal.

Cada tecla puede tener dos estados posibles: pulsado o no pulsado. El intérprete dispone de varias instrucciones que analizarán el valor de entrada del teclado y ejecutarán distintas operaciones en función de lo que el desarrollador desee procesar (ver Secciones 4.1.3, 4.1.3 y 4.1.3).

4. CAPÍTULO

Conjunto de Instrucciones

En este capítulo se realizará un análisis de las distintas instrucciones de las que hará uso el intérprete del `CHIP-8` para la descodificación y ejecución de los distintos programas de entrada para su posterior implementación en `C++`.

4.1. Instrucciones

Las instrucciones o *opcodes* son códigos que detallan los comandos que la unidad de procesamiento puede ejecutar, como operaciones de escritura y lectura sobre los registros o operaciones aritmético-lógicas.

Las instrucciones de entrada del `CHIP-8` tienen un tamaño de 16 bits, por lo tanto se necesitarán dos espacios de dos bytes para almacenarlas en la memoria principal antes de que estas sean descodificadas y ejecutadas por el procesador. De modo que sea posible la descodificación de cada código de entrada, todas las instrucciones siguen un patrón identificativo (ver Figura 4.1):

- *Opcode* o identificador: es el valor que identifica la instrucción a ejecutar. En la mayoría de casos se trata del valor hexadecimal más significativo.
- Datos: información que el procesador utilizará como entrada para ejecutar la instrucción una vez esta haya sido identificada.

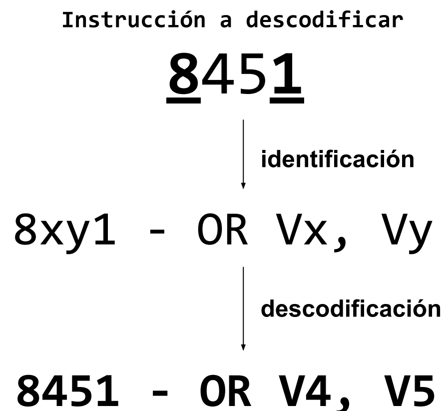

Figura 4.1: Identificación y descodificado de la instrucción 8xy1

Tomemos como ejemplo la instrucción `8451`. Dado que empieza por 8 y acaba en 1, el comando se interpretará como la operación bit a bit `OR` entre el valor del registro `V4` y el de `V5` (datos de entrada) - `AND V4, V5`.

Existen un total de 34 instrucciones que nuestro procesador debe ser capaz de interpretar para poder descodificar y procesar todas las entradas. El conjunto de instrucciones o *Instruction Set Arquitecture* recogerá todas estas especificaciones y ofrecerá al programador una manera de comunicarse imperativamente con el procesador.

A continuación se listarán todas las instrucciones de las que hará uso el intérprete para realizar el ciclo de ejecución de los programas.

4.1.1. Instrucciones aritméticas

7xkk - ADD Vx, byte

Adiciona `kk` al registro `Vx`.

Se actualizará el registro localizado en el índice x del *array* de registros adicionando `kk` a su valor actual.

8xy1 - OR Vx, Vy

Realiza la operación bit a bit *inclusive* OR entre el valor del registro `Vx` y `Vy` y lo almacena en `Vx`.

Para realizar esta operación en C++ podemos utilizar el operador |= que realizará la operación *inclusive* OR y asignará el resultado a `Vx`.

8xy2 - AND Vx, Vy

Realiza la operación bit a bit `AND` entre el valor del registro `Vx` y `Vy` y lo almacena en `Vx`.

Para realizar esta operación en C++ podemos utilizar el operador 'I&'= que realizará la operación AND y asignará el resultado a `Vx`.

8xy3 - XOR Vx, Vy

Realiza la operación bit a bit `AND` entre el valor del registro `Vx` y `Vy` y lo almacena en `Vx`.

Para realizar esta operación en C++ podemos utilizar el operador 'I&'= que realizará la operación AND y asignará el resultado a `Vx`.

8xy4 - ADD Vx, Vy

Adiciona los valores de los registros `Vx` y `Vy` y lo almacena en `Vx`.

Si el resultado es mayor a 255 (valor máximo decimal alcanzable en 8 bits) el valor del registro de *carry flag* `VF` deberá cambiar a 1 y a 0 si ocurre el caso contrario.

8xy5 - SUB Vx, Vy

Realiza una substracción entre los valores de los registros `Vx` y `Vy` y almacena el resultado en `Vx`.

Si el valor del registro de `Vx` es mayor al de `Vy`, el valor del registro de *carry flag* `VF` deberá cambiar a 1 y a 0 si ocurre el caso contrario.

8xy7 - SUBN Vx, Vy

Realiza una substracción entre los valores de los registros `Vy` y `Vx` y almacena el resultado en `Vx`.

Si el valor del registro de `Vy` es mayor al de `Vx`, el valor del registro de *carry flag* `VF` deberá cambiar a 1 y a 0 si ocurre el caso contrario.

Cxkk - RND Vx, byte

Carga el resultado de la operación bit a bit AND entre el valor del registro `Vx` y un byte aleatorio.

Normalmente, estos valores aleatorios se consiguen vía hardware mediante la lectura de los valores de voltaje que recibe un pin desconectado. Dado que se esta trabajando con un intérprete, se hará uso de las librerías propias de `C++` para esta función.

Fx1E - ADD I, Vx

Carga el resultado de la suma entre el valor del registro `Vx` y del registro `VI` en `VI`

4.1.2. Instrucciones de salto

00EE - RET

Regresa de una subrutina.

Las siglas RET hacen referencia a *Return*, por lo tanto es una instrucción de retorno desde una subrutina. La pila guarda la dirección de memoria donde estará la siguiente instrucción a ejecutar cuando el procesador vuelva a su estado anterior.

Utilizando el puntero de pila para conseguir la localización de la parte superior de la pila y realizando un *pop* en la propia pila esta instrucción conseguirá devolver a la CPU al contador de programa anterior a la ejecución de la subrutina.

1nnn - JP addr

Salta a la subrutina localizada en `nnn`.

Esta instrucción indicará al procesador que salte a la instrucción localizada en la dirección `nnn`. A diferencia de las instrucciones de tipo `CALL`, solo debemos actualizar el PC actual ya que no será necesario almacenar la dirección de retorno.

2nnn - CALL addr

Llama a la subrutina localizada en `nnn`.

Las instrucciones de tipo `CALL` llamarán a una subrutina localizada en la dirección especificada en `nnn`, actualizando el PC actual pero guardando el estado anterior del procesador, de modo que cuando se termine la ejecución de dicho bloque de código el procesador pueda recuperar el anterior valor del PC.

Para almacenar el valor del PC se hará uso de la parte superior de la pila, que a su vez actualizará el puntero de esta.

Bnnn - JP V0, addr

Salta a la subrutina localizada en la dirección resultante entre la suma de `nnn` y el valor del registro `V0`.

Tras calcular la suma entre la dirección de entrada `nnn` y el valor del registro `V0` se actualizará el valor del PC con el resultado.

4.1.3. Instrucciones a memoria

00E0 - CLS

Inicializa a 0 el *buffer* de gráficos.

Las siglas CLS corresponden con *Clear Display*, por lo tanto, esta instrucción será usada para restablecer nuestro *buffer* de memoria de gráficos.

Para interpretar esta funcion basta con inicializar todos los valores del *buffer* a 0 (píxel apagado).

3xkk - SE Vx, byte

Ignora la siguiente instrucción si el valor del registro `Vx` es igual a `kk`.

Se realizará una comparación entre el valor almacenado en el registro `Vx` y el especificado en `kkk`. Si estos coinciden, se actualizará el PC evitando la ejecución de la siguiente instrucción localizada en la memoria principal que se ejecutaría en el siguiente ciclo.

4xkk - SNE Vx, byte

Ignora la siguiente instrucción si el valor del registro `Vx` es distinto a `kk`.

Se realizará una comparación entre el valor almacenado en el registro `Vx` y el especificado en `kkk`. Si estos no coinciden, se actualizará el PC evitando la ejecución de la siguiente instrucción localizada en la memoria principal que se ejecutaría en el siguiente ciclo.

5xy0 - SE Vx, Vy

Ignora la siguiente instrucción si el valor del registro`Vx` es igual al del registro `Vy`.

Se realizará una comparación entre el valor almacenado en el registro `Vx` y en el registro `Vy`. Si estos coinciden, se actualizará el PC evitando la ejecución de la siguiente instrucción localizada en la memoria principal que se ejecutaría en el siguiente ciclo.

9xy0 - SNE Vx, Vy

Ignora la siguiente instrucción si los valores de los registros `Vx` y `Vy` no son iguales.

Se realizará una comparación entre el valor almacenado en el registro `Vx` y en el registro `Vy`. Si estos no coinciden, se actualizará el PC evitando la ejecución de la siguiente instrucción localizada en la memoria principal que se ejecutaría en el siguiente ciclo.

Ex9E - SKP Vx

Ignora la siguiente instrucción si la tecla con el mismo valor que el registro `Vx` esta presionada.

Se hará una lectura del valor de la tecla (es decir, si ha sido presionada o no) en el *array* que almacena los valores de los *inputs* en la dirección que dicte el valor del registro `Vx`. Si este es 1, la tecla ha sido presionada, y se incrementará el PC.

ExA1 - SKNP Vx

Ignora la siguiente instrucción si la tecla con el mismo valor que el registro `Vx` no esta presionada.

Se hará una lectura del valor de la tecla (es decir, si ha sido presionada o no) en el *array* que almacena los valores de los *inputs* en la dirección que dicte el valor del registro `Vx`. Si este valor es 0, la tecla no ha sido presionada, y se incrementará el PC.

6xkk - LD Vx, byte

Carga el valor `kk` en el registro `Vx`.

Se actualizará el valor del registro localizado en el índice `x` del *array* de registros a `kk`.

8xy0 - LD Vx, Vy

Carga el valor del registro `Vy` en `Vx`.

Se actualizará el registro `Vx` con el valor actual del registro `Vy`.

Annn - LD I, addr

Carga la dirección `nnn` en el registro de índices `VI`.

Fx29 - LD F, Vx

Carga la dirección especificado por el valor del registro `Vx` en el registro `I`, donde se localizará siguiente sprite a dibujar.

Como se ha explicado en la Sección 3.2, el intérprete utiliza las direcciones `0x050` en adelante para almacenar los sprites por defecto y los definidos por el desarrollador. Por ello, la dirección que se almacenará en el registro `I` será la definida a partir de este valor mas el contenido por el registro `Vx`.

Fx0A - LD Vx, K

Carga el valor de la próxima tecla presionada en el registro `Vx`.

El intérprete deberá esperar hasta que se detecte el siguiente input, comprobando todos los valores posibles. La instrucción se ejecutará una y otra hasta que esto ocurra.

Fx15 - LD DT, Vx

Carga el valor del registro `Vx` en el temporizador de retardo.

Fx18 - LD ST, Vx

Carga el valor del registro `Vx` en el temporizador de sonido.

Fx07 - LD Vx, K

Almacena el valor del temporizador en el registro `Vx`.

Fx55 - LD [I], Vx

Almacena los valores de todos los registros disponibles desde `V0` hasta `Vx` en la direccion de memoria localizada en el registro `I`.

Se iterarán todos los registros hasta `Vx` y escribiendo los valores de estos en los espacios de memoria a partir de `I`.

Fx65 - LD Vx, [I]

Carga los valores almacenados en la memoria principal a partir de la dirección localizada por el registro `I` en los registros disponibles desde `V0` a `Vx`.

Se iterarán todas las direcciones de memoria a partir la dirección localizada en el registro `I` y guardando los valores encontrados en los registros de `V0` a `Vx`.

4.1.4. Instrucciones de movimiento de bits

8xy6 - SHR Vx

Realiza una división entre dos del valor del registro `Vx` y almacena el resultado en `Vx`.

Para llevar a cabo esta operación se realizará un *bit shift* hacia la derecha (*Shift Logical Right*). Si el bit menos significativo es 1, el valor del registro de *carry flag* `VF` deberá cambiar a 1 y a 0 si ocurre el caso contrario.

8xyE - SHL Vx

Multiplica por dos el valor del registro `Vx` y almacena el resultado en `Vx`.

Para llevar a cabo esta operación se realizará un *bit shift* hacia la izquierda (*Shift Logical Left*). Si el bit más significativo es 1, el valor del registro de *carry flag* `VF` deberá cambiar a 1 y a 0 si ocurre el caso contrario.

Fx33 - LD B, Vx

Almacena el valor decimal codificado en binario del valor del registro Vx en las direcciones de memoria encontradas `I`, `I+1` e `I+2`.

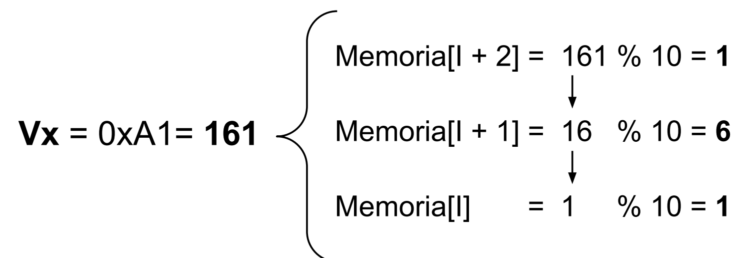

Figura 4.2: Operación BCD.

Para realizar esta operación el valor del registro `Vx` se dividirá en tres partes, siendo la más significativa almacenada en `I+2` y las otras dos en `I+1` e `I` respectivamente.

4.1.5. Otras instrucciones

0nnn - SYS addr

Salta a una rutina de código máquina en la dirección `nnn`.

Esta instrucción no se implementará en nuestro intérprete ya que su uso solo tendría sentido en los computadores donde el intérprete `CHIP-8` original fue implementado.

Dxyn - DRW Vx, Vy, nibble

Dibuja un sprite de n bytes (filas) empezando en la dirección de memoria indicada por el registro I y en la coordenada del píxel marcado por los registros `Vx` y `Vy`.

Se iterará sobre el sprite a dibujar que se encuentra en la dirección almacenada por I, linea a linea hasta el máximo marcado por `n` (altura). Como cada linea es representada por un `byte`, la figura constará de ocho columnas (anchura).

Tal y como se explica en la Figura 3.5, se realizará una operación `XOR` entre la figura a dibujar y los valores actuales de los píxeles sobre los que se quiere realizar la operación de dibujado. Si se va dibujar sobre un píxel que estaba previamente activado, se actualizará el registro `VF` (*carry flag*) a 1 ya que ha ocurrido una colisión, útil si otra instrucción o el desarrollador quiere tener constancia de ello.

5. CAPÍTULO

Ciclos de ejecución del procesador e inicialización

En este capítulo se describirán los distintos pasos que la unidad de cómputo central debe cumplir para hacer posible la ejecución de programas en el emulador: captación, descodificación y ejecución.

5.1. Lectura de ROM e inicialización

Antes de realizar la lectura de los distintos *opcodes* que forman parte del programa de entrada a procesar, es necesario inicializar todos los recursos que el emulador utilizará para esta tarea. Para ello, se inicializarán a cero todos los espacios de memoria de la máquina virtual, los registros, la pila y el puntero de pila, el PC y el *buffer* de memoria gráfica.

El emulador utilizará archivos de solo lectura llamados *Read Only Memory* como entrada para cargar en la memoria los comandos a descodificar y ejecutar.

Tal y como se ha especificado en el análisis del hardware de la máquina virtual (ver Sección 3.2), el procesador hará uso del tercer y último tramo de la memoria principal reservado para la carga de los distintos comandos que aparecerán en la ROM. Cada instrucción tiene un tamaño de 16 bits, por lo tanto, cada instrucción deberá almacenarse en dos partes, es decir, en dos direcciones de memoria contiguas, ya que solo se puede almacenar un `byte` en cada espacio.

Esta carga de datos comenzará en la dirección `0x200`. En consecuencia, el PC cargará este dato al iniciar el programa.

5.2. Ciclo del procesador

Cada ciclo del procesador se divide en tres pasos: captación (*fetch*), descodificación (*decode*) y ejecución (*execution*), y tiene como referencia el actual valor del PC, que apunta a la actual instrucción que se captará de la memoria principal, descodificará y se ejecutará.

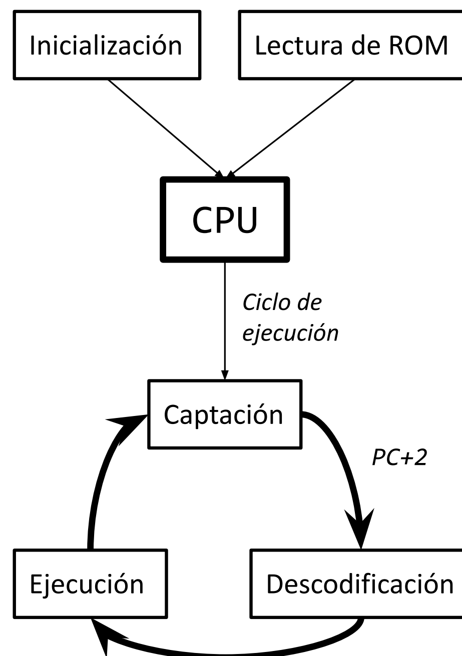

Figura 5.1: Ciclo del procesador.

La velocidad a la que se ejecuta cada ciclo es determinado por los ciclos por segundo a los que funciona el procesador. Cuanto más alto sea este valor, más ciclos se ejecutarán en el mismo intervalo de tiempo.

5.2.1. Captación

El proceso de captación consiste en la lectura del valor en la memoria que apunta el PC, pues es la información de entrada que el procesador necesita para descodificar la instrucción. En cada ciclo, solo se procesará una instrucción.

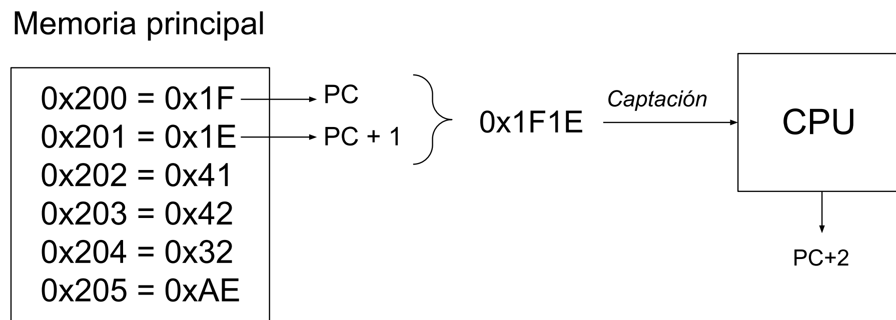

Figura 5.2: Proceso de captación.

Cada instrucción consta de cuatro valores hexadecimales divididos en dos direcciones de memoria contiguas. El procesador realizará una captación de estos comandos leyendo los valores almacenados en la dirección actual apuntada por el PC y PC+1 (ver Figura 5.2).

Una vez se haya conseguido este dato el PC se actualizará apuntando a la siguiente dirección de memoria.

5.2.2. Descodificación

La fase de descodificación consiste en interpretar la instrucción que previamente el procesador ha captado para su posterior ejecución. Tomando esos dos valores se formará una palabra de 16 bits, que el procesador dividirá hasta en cuatro partes diferentes (dependiendo del tipo que sea) para interpretar esa operación.

Tal y como se ha explicado en la Sección 4.1, para descodificar las instrucciones el intérprete debe identificar de que instrucción se trata y localizar los diferentes datos que utilizará para la ejecución.

Para descodificar el *opcode* el intérprete aplicará una máscara `AND` en el valor hexadecimal más significativo con el fin de localizar el rango de posibles instrucciónes. Una vez esto se haya realizado, dependiendo del grupo de instrucciones acotado se seguirán aplicando máscaras `AND` a los distintos valores posibles. Finalmente, al identificar el *opcode* se descodificarán los datos para la ejecución del mismo (ver Figura 5.3).

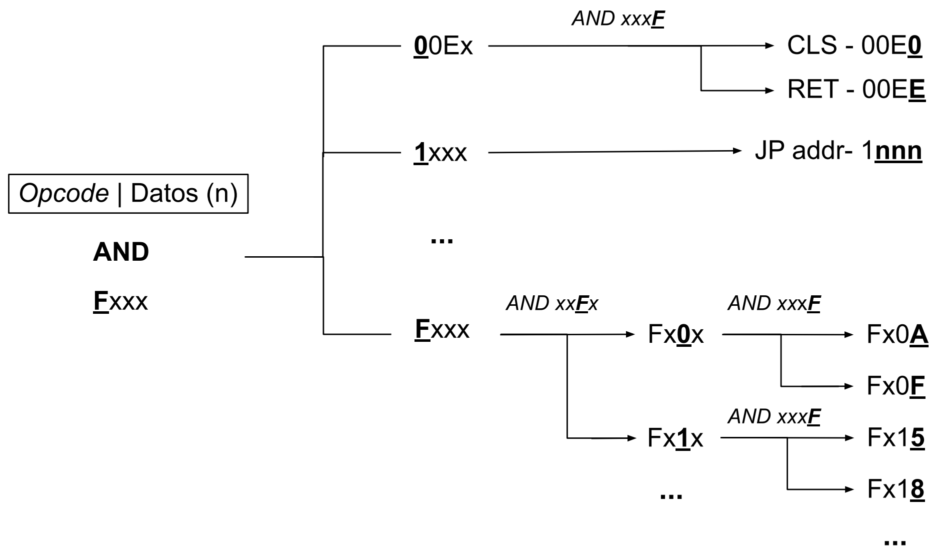

Figura 5.3: Proceso de descodificación.

5.2.3. Ejecución

Cuando el procesador haya identificado e interpretado la instrucción actual se ejecutará la rutina correspondiente a dicha operación.

Estas operaciones pueden ser de cuatro tipos diferentes:

- Instrucciones de movimiento de bits o *bitshifting*: Utilizado por ciertas operaciones de división o multiplicado, o para desplazar bits a la izquierda o derecha.
- Instrucciones aritméticas: Utilizado por las operaciones matemáticas y lógicas.
- Instrucciones de salto: En la que se cambia la siguiente el valor del PC, para ejecutar diferentes bloques de código o bucles.
- Instrucciones a memoria: Son con las que el procesador lee y escribe información en la memoria principal.

Todas las instrucciones de las que hará uso el intérprete se pueden encontrar en la Sección 4.1.

6. CAPÍTULO

Simple DirectMedia Layer

En este capítulo se realizará una breve descripción de las distintas funcionalidades que la librería *Simple DirectMedia Layer* proporcionará en la ejecución del emulador, además de una pequeña introducción a la plataforma.

6.1. SDL como plataforma

SDL o *Simple DirectMedia Layer* es una librería de desarrollo multiplataforma diseñada para proporcionar acceso de bajo nivel al hardware de audio, teclado, ratón y gráficos a través de *OpenGL* y *Direct3D*. Es utilizada por software de reproducción o renderizado de vídeo, emuladores y diversas aplicaciones interactivas.

Utilizando este recurso será posible renderizar en pantalla el *buffer* de gráficos de nuestro intérprete, reproducir sonido mientras el temporizador de sonido esté activo y detectar las entradas del teclado. Es posible implementar estas funciones para que funcionen junto con el ciclo de ejecución del intérprete.

SDL es oficialmente compatible con distintos sistemas operativos como Windows, Mac OS X, Linux, iOS y Android. Además, esta librería está escrita en `C` y funciona de forma nativa en `C++`.

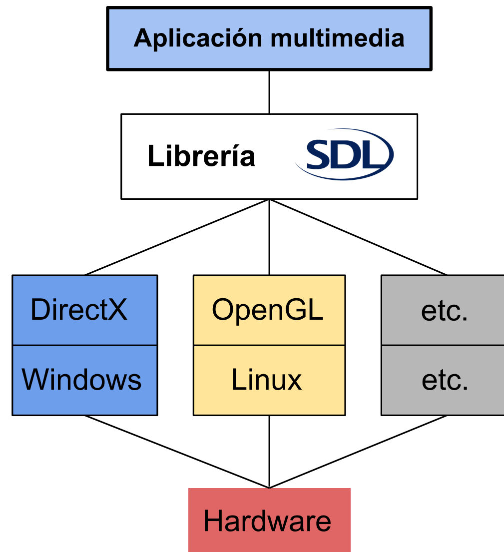

Figura 6.1: Capas de abstracción de SDL en los distintos sistemas operativos.

6.2. Renderizado de gráficos

Con SDL es posible hacer uso de la librería OpenGL o Direct3D (Windows) para renderizar gráficos en 2D.

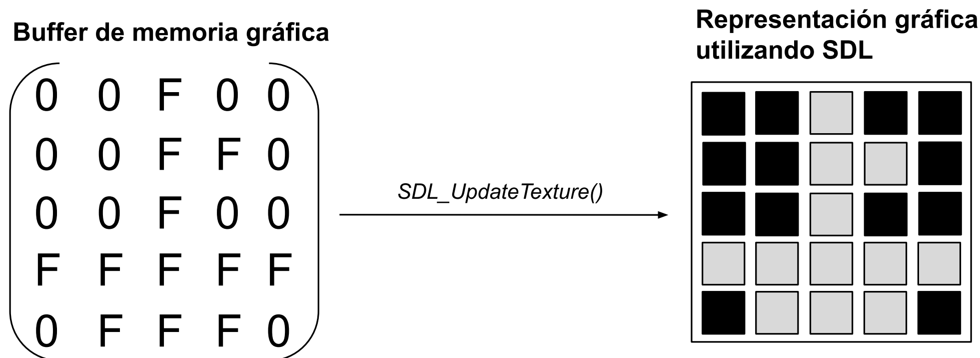

Figura 6.2: Función *SDL_UpdateTexture()*

Para crear la imagen 2D que representará el contenido del *buffer* de memoria de gráficos primero se utilizará la clase *SDL_Renderer* inicializando así nuestro acelerador de gráficos 2D. Además, podemos simular que la imagen 2D a renderizar se trate de una textura (con la clase *SDL_Texture*), siendo cada píxel de esta el contenido de la memoria gráfica.

Se utilizará el espacio de color RGB para mostrar las figuras en pantalla, siendo los píxeles

encendidos (color blanco) el valor máximo alcanzable en un entero sin signo de 32 `bits` (*uint32_t*) y lo contrario en los píxeles apagados (color negro).

En cada ciclo de ejecución se enviará como parámetro el valor actual de cada píxel a la función *SDL_UpdateTexture()*, que llamará al acelerador de gráficos 2D para actualizar la textura con los nuevos datos de entrada. Esto nos permitirá actualizar la textura actualizando las propiedades de la imagen presentada en cada ciclo del procesador.

6.3. Reproducción de sonido

Dado que solo necesitamos reproducir un único sonido o zumbido cuando el valor del temporizador de sonido sea mayor que cero generar una onda sonora de una frecuencia determinada es posible utilizando la librería *SDL_AUDIO* proporcionado por la plataforma SDL.

El *beep* que sonará será una onda sinusoidal con una frecuencia de muestreo de 44100 `Hz` y amplitud 128:

$$128 sin(2\pi \frac{1000t}{44100} + 1)$$

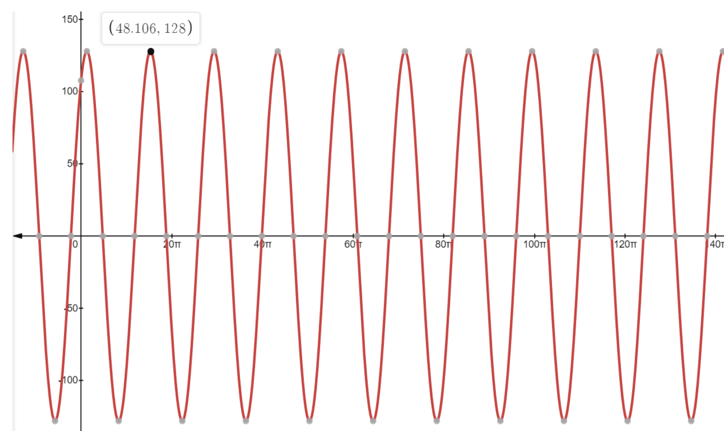

Figura 6.3: Representación gráfica de la onda sinusoidal.

Una vez inicializado el sistema de sonido con estos parámetros siempre que el temporizador de sonido tenga un valor positivo se ejecutará la función *SDL_PauseAudio(0)* o *SDL_PauseAudio(1)* si este valor es igual a cero.

6.4. Eventos de entrada

Es posible detectar pulsaciones de teclado utilizando la función de detección de eventos de entrada que la librería SDL incluye.

Cuando el programa principal detecte una entrada se actualizará el valor a uno del índice del *array* de teclas correspondiente a la tecla presionada por el usuario. Por otro lado, cuando se deje de pulsar la tecla, se cargará el valor de cero. De esta manera el intérprete será capaz de saber si una tecla esta siendo pulsada en el ciclo actual. Esta detección se realizará a través de la función *ProcessInput()* que hace uso de la clase *SDL_Event* para este propósito. Además, esta comprobación se realizará al inicio de cada iteración del bucle principal.

La redistribución de las entradas del teclado original se realizarán de esta forma (ver Figura 6.4).

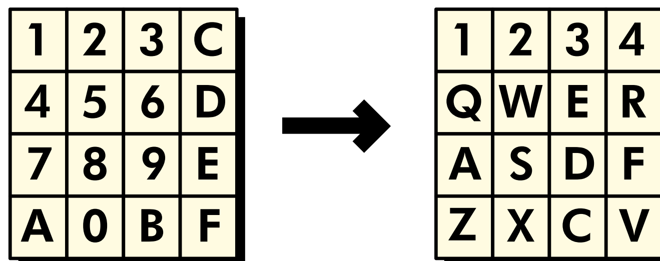

Figura 6.4: Redistribución de las entradas del teclado.

Por otro lado, ciertas teclas tendrán funciones específicas que el usuario puede hacer uso en cada periodo. Si el controlador de eventos detecta que se ha pulsado la barra espaciadora, la ejecución del ciclo actual se detendrá hasta que esta se vuelva a pulsar. En este estado se podrá pulsar la tecla de suma para avanzar en la ejecución ciclo por ciclo.

7. CAPÍTULO

Pruebas y depuración

En este capítulo se describirán las distintas pruebas realizadas sobre el prototipo, analizando la estabilidad de este y la comprobación del correcto funcionamiento de las multiples instrucciones implementadas.

7.1. ROMs de pruebas unitarias

Una manera sencilla y rápida de comprobar el correcto funcionamiento las implementaciones en `C++` de las distintas instrucciones que forman parte del ciclo de ejecución del procesador es utilizando ROMs de pruebas unitarias. Estos programas se pueden utilizar como entrada a la hora de analizar las distintas funciones, ya que incluyen distintos códigos de error que son útiles para identificar los posibles problemas que ocurran en la ejecución del programa.

A continuación se mostrarán los resultados obtenidos empleando dos *Unit Testing ROMs* que podemos encontrar en *GitHub*.

7.1.1. Prueba unitaria 1

La primera prueba desempeñada se ha realizado utilizando la ROM desarrollada por *corax89*. Esta prueba de unidades comprobará la implementación de las siguientes instrucciones:

- `3XNN 00EE 8XY5`
  `4XNN 8XY0 8XY6`
  `5XY0 8XY1 8XYE`
  `7XNN 8XY2 FX55`
  `9XY0 8XY3 FX33`
  `ANNN 8XY4 1NNN`

Si se utiliza esta ROM en el emulador, este es el resultado:

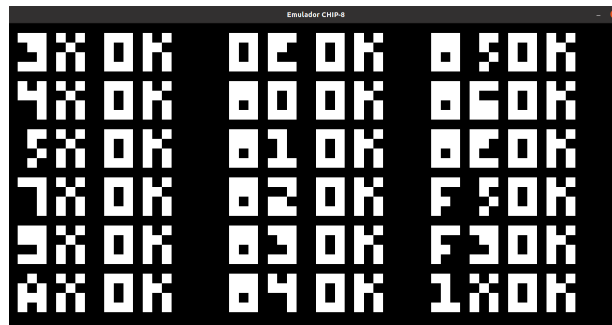

Figura 7.1: Primera ROM utilizada para la prueba de unidades.

Como se observa en la Figura 7.1, las instrucciones se han ejecutado correctamente ya que se puede leer un *ok* en cada una de ellas.

7.1.2. Prueba unitaria 2

Para la segunda prueba se ha utilizado la ROM desarrollada por *metteo*.

Esta ROM hace uso de códigos de error que se imprimirán en pantalla que especifican más detalladamente la instrucción mal implementada o cualquier error relacionado con esta. Los códigos de error que el desarrollador ha definido son los siguientes.

- ERROR INI - La inicialización del emulador ha fallado. Todos los registros deben inicializarse a 0.
- ERROR BCD - Problemas en la instrucción Fx33.
- ERROR 0 - Problemas con la instrucción Fx65. No se pueden cargar ceros en los registros desde la memoria principal.

- ERROR 1 - *Sprite* 8x5 no encontrado.
- ERROR 2 - Adición sin *overflow* (254+1). El *carry flag* debe estar definido a 0, pero después de la operación sigue estando a 1.
- ERROR 3 - Después de la operación 254+1, el registro V0 debe tener cargado el valor 255, pero no lo hace.
- ERROR 4 - Adición con *overflow* (255+1). El *carry flag* debe estar definido a 1, pero después de la operación sigue estando a 0.
- ERROR 5 - Resultado erroneo después de una operación de adición (255+1). El valor del registro debería ser 0.
- ERROR 6 - Después de la substracción 1-1 el *carry flag* debe estar a 1, pero no lo hace.
- ERROR 7 - Resultado erróneo después de realizar la operación 1-1. El resultado debe ser 0.
- ERROR 8 - Después de la substracción 0-1 el *carry flag* debe estar a 0, pero no lo hace.
- ERROR 9 - Resultado erróneo después de la operación 0-1. El registro V0 debe tener el valor de 255.

Si hacemos uso de esta ROM de prueba de unidades, como se puede observar en la Figura 7.2, el resultado es el siguiente:

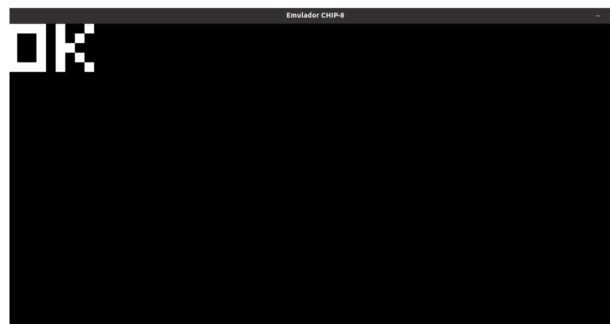

Figura 7.2: Segunda ROM utilizada para la prueba de unidades.

7.2. Depurador

El emulador consta de un depurador capaz de mostrar en la terminal los valores actuales de los recursos de la máquina virtual. Entre estos, podemos conocer el valor hexadecimal almacenado en los distintos registros, el registro de índices, los dos temporizadores, la pila y el puntero de pila, el contador de programa y la instrucción actual que se está ejecutando. Utilizar esta función es útil para el desarrollador ya que permite saber el estado actual de cada recurso utilizado en la máquina virtual.

Figura 7.3: Emulador ejecutando una ROM de *Breakout* [Carmelo Cortez, 1979] con el depurador activo.

Dado que probablemente el programa se esté ejecutando a muchas instrucciones por ciclo, mantener un seguimiento del depurador es una tarea complicada a esa velocidad. Por ello, se puede ejecutar el programa instrucción por instrucción si pausamos el programa con la barra espaciadora y pulsamos la tecla de adición para ir paso por paso, dándole al usuario la posibilidad de observar los valores actuales de los recursos del emulador.

Esta función es útil si el programador quiere saber con exactitud que está ocurriendo en cada ciclo del procesador.

8. CAPÍTULO

Planificación del proyecto

En este capítulo se describirá el plan propuesto al inicio del proyecto, y se especificarán las tareas a realizar y la planificación en el tiempo. Por otro lado, se hará una breve alusión al plan de riesgo teniendo en cuenta las posibles extensiones en el tiempo y su posterior análisis.

8.1. Tareas principales

Con el fin de cohesionar todas las fases que forman parte de este proyecto se ha dividido el trabajo a realizar en distintos lotes de trabajo.

- **Lote de Diseño de Solución (DS)**
  - **DS.L1 Interprete CHIP-8:** Estudio del lenguaje interpretado y funcionamiento general.
  - **DS.L2 Máquina virtual:** Análisis de los diferentes recursos que se emularán.
  - **DS.L3 Librería SDL:** Estudio de la librería SDL y su posible implementación en el programa.
- **Lote de Entorno de Trabajo (ET)**
  - **Preparación del entorno de trabajo:** Descarga e instalación de dependencias.

- **Lote de Desarrollo del Prototipo (D)**
  - **D.L1 Diseño general de la estructura del programa:** Diseño de la estructura general del prototipo.
  - **D.L2 Virtualización del hardware a emular:** Generación de la estructura de datos a utilizar para la máquina virtual.
  - **D.L3 Implementación del conjunto de instrucciones**.
  - **D.L4 Aplicación de la librería SDL:** Uso de la librería SDL para la gestión de imagen y sonido y control de entradas.
- **Lote de Prueba de Unidades (P)**
  - **P.L1 *Unit Testing ROMS*:** Prueba de la implementación de las distintas instrucciones del intérprete.
- **Lote de Implementación de Mejoras (M)**
  - **M.L1 Depurador:** Implementación de un depurador de los valores actuales del hardware emulado.
  - **M.L2 Carga/Descarga de estados:** Implementación funcional de guardado y carga de estado del procesador.
- **Lote de Redacción de Informe (I)**
  - **I.L1 Redacción del informe del proyecto.**

8.2. Estimación de tiempo de las distintas tareas

Se especificará el tiempo inicialmente estipulado para la realización de las distintas tareas que componen el proyecto y la dedicación final que estas han recibido en la Tabla 8.1. Por otro lado, se representará en la tabla 8.2 el periodo de realización en el tiempo de la mismas.

Tarea	**Horas estipuladas**	**Dedicación final**
Diseño de Solución	**50**	**65**
Estudio del intérprete CHIP-8	25	30
Análisis de los recursos de la máquina virtual	15	15
Estudio de la librería SDL	10	20
Entorno de Trabajo	**5**	**10**
Preparación del entorno de trabajo	5	10
Desarrollo del Prototipo	**125**	**130**
Diseño general de la estructura del programa	15	15
Virtualización de los recursos a emular	20	20
Implementación del conjunto de instrucciones	50	50
Aplicación de la librería SDL	40	45
Prueba de Unidades	**30**	**25**
Pruebas de la implementación de las instrucciones	30	25
Implementación de Mejoras	**40**	**45**
Depurador de hardware	15	15
Carga/Descarga de estados del procesador	25	30
Redacción de Informe	**50**	**60**
Redacción del informe del proyecto	50	60
TOTAL	**300**	**335**

Tabla 8.1: Estimación del tiempo dedicado de las distintas tareas.

	JUNIO	JULIO	AGO.	SEPT.
DS.L1				
DS.L2				
DS.L3				
ET.L1				
D.L1				
D.L2				
D.L3				
D.L4				
P.L1				
M.L1				
M.L2				
I.L1				

Tabla 8.2: Periodo de realización de las tareas en el tiempo.

8.3. Plan de riesgo general del proyecto

La planificación de riesgos es un proceso bastante estructurado que tiene como objetivo identificar los riesgos y cómo responder a ellos, y definir cómo controlar los problemas que surjan durante el desarrollo del proyecto.

En proyectos con una duración más o menos estimada (300 horas) es importante contar con un plan de riesgos que ayuden a prevenir e identificar las posibles desviaciones que puedan ocurrir durante el proceso de ejecución del trabajo de fin de grado. La estimación general de las horas que se tomarán para realizar las tareas es un proceso complicado debido a la alta duración de este.

Para hacer frente a los imprevistos se ha decidido extender el tiempo límite estimado en un principio a las tareas más complejas de llevar a cabo, como la implementación del conjunto de instrucciones, las distintas pruebas unitarias a realizar y la solución errores generales del programa. Por otro lado, se ha reducido el numero de objetivos del proyecto definidos en un principio para respetar el límite de tiempo inicialmente determinado, como el desarrollo de nuevos añadidos.

8.4. Análisis de las desviaciones y problemas generados

La idea original del proyecto consistía en construir un prototipo capaz de emular los recursos incorporados por los primeros sistemas que utilizaron el interprete del `CHIP-8` en un hardware específico como puede ser una placa Arduino, un microcontrolador PIC o incluso una calculadora capaz de dibujar elementos en pantalla. Dado que tener acceso a este hardware es un paso más a tener en cuenta, y que se quería hacer hincapié en la portabilidad del proyecto, esta idea fue descartada.

Por otro lado, las razones de las distintas desviaciones que se han generado a la hora de construir el prototipo han estado principalmente ligadas a la situación actual que nos concierne a todos respecto al COVID-19. Debido a temas relacionados con la salud se han tenido que realizar varias paradas durante el proceso de desarrollo del proyecto, generando un desajuste general en el intervalo de tiempo principalmente propuesto para la finalización del mismo.

Los principales problemas surgidos durante el desarrollo del proyecto han tenido relación con la parte práctica del proyecto. Siendo este un proyecto con cierto grado de compleji-

dad a la hora de implementarlo en `C++`, muchos de estos imprevistos han tenido relación con no lograr los resultados esperados en un principio, como errores con la instrucción de dibujado o lectura y escritura errónea de datos en los distintos registros. Por otro lado, el desconocimiento total de la librería SDL y su posible aplicación en el proyecto demoró la finalización de este. Gracias a las distintas ROMs de pruebas unitarias que existen en internet se logró solventar este problema dedicandole el tiempo necesario.

9. CAPÍTULO

Conclusiones

En lo que respecta a la temática principal del desarrollo del emulador e intérprete del CHIP-8 se han logrado cumplir todos los objetivos propuestos al comienzo del proyecto.

En la realización general del proyecto he adquirido conocimientos teóricos y prácticos que han sido esenciales para permitir la continuidad del mismo, tanto a la hora de idear una estructura general del prototipo como del desarrollo de la máquina virtual y los distintos recursos que forman parte de esta.

En cuanto a las nociones teóricas presentes en el trabajo realizado, he aprendido a plasmar el concepto y la funcionalidad de un hardware físico en una arquitectura completamente distinta. Así mismo, la investigación llevada a cabo previamente al desarrollo del prototipo me ha ayudado a comprender mejor las diferentes capas de abstracción presentes en cada ciclo de ejecución de la unidad de procesamiento central. Adquirir esta idea general me ha proporcionado los recursos necesarios para desarrollar en un futuro emuladores de máquinas aún más complejas y con distintas peculiaridades.

En relación al apartado práctico del desarrollo del programa, he mejorado mis aptitudes para desarrollar programas en C++. Siendo esta una aplicación estrechamente relacionada con la programación orientada a objetos y la programación a bajo nivel, este lenguaje de programación es y ha sido el más adecuado para desarrollar la idea general del proyecto.

Por otra parte, he logrado hacer uso de la librería SDL para impulsar la portabilidad del programa, disponible en la mayor parte de sistemas operativos.

En cuanto al proceso de investigación previa al desarrollo general del proyecto, la bús-

queda de información sobre el intérprete no ha sido tarea sencilla pues esta se presenta de forma sesgada en la red, siendo esta fuertemente influenciada por la comunidad que forma parte del desarrollo de herramientas y prototipos basados este entorno, en muchas ocasiones haciendo complicado idear un modelo general del proyecto.

No hay un consenso general en como plantear el desarrollo del emulador y la estructura principal del proyecto, incluso de la implementación del mismo intérprete. Por ejemplo, ciertas ROMs no funcionarán si una instrucción se ha implementado de forma diferente a la que esperaba el desarrollador del programa a ejecutar, o no se mostrarán los resultados esperados. Aún así, ser consciente de las limitaciones principales y el alcance de mi implementación personal del emulador me ha ayudado a construir una idea general del programa a desarrollar y cumplir los objetivos propuestos al inicio del proyecto.

En lo que concierte al ámbito personal en la realización de este trabajo he de decir que me he centrado mayormente en fortalecer mis capacidades de sintetización de ideas. Principalmente, mi idea era redactar un documento fácil de entender y de leer, con una clara cohesión y progresión de conceptos. Dado que no es un proyecto complejo de entender pero un tanto complicado de implementar, escribir de una manera sencilla ayuda a que la recepción de ideas sea efectiva, con el fin de que más personas se animen a revivir la esencia de esas máquinas que fueron tan relevantes en el pasado, tanto a nivel comercial y de productividad como de progreso general de la tecnología.

9.1. Trabajo futuro

Dado que la escalabilidad es una característica fuertemente ligada al desarrollo de emuladores, en un futuro se podrían implementar las siguientes características:

- Implementar el prototipo realizado en hardware específico, como calculadoras con pantallas LCD o placas Arduino.
- Nuevas instrucciones funcionales para el intérprete (`Super CHIP-8`).
- Desarrollo de una interfaz general para el programa.

Anexos

A. ANEXO

Instalación

En este capítulo se resaltarán los diferentes recursos que el usuario necesita para generar el binario necesario para ejecutar el intérprete, y una pequeña guía para compilar el programa.

A.1. Dependencias

Para compilar el intérprete `CHIP-8` es necesario disponer de las siguientes librerías instaladas en el sistema operativo o entorno virtual en el que se vaya a ejecutar el programa:

- *CMake*: Gestiona el proceso de compilación en el sistema operativo y de forma independiente del compilador. *CMake* está diseñado para soportar jerarquías de directorios complejas y aplicaciones que dependen de varias bibliotecas.
- `GCC`: *GNU Compiler Collection* es un compilador integrado del proyecto GNU para `C`, `C++`, `Objective C` y `Fortran`. Es necesario para generar el archivo binario.
- `SDL`: Es necesario disponer de esta librería para compilar el código, que implementa todas las funciones multimedia que utilizará el programa.

Además, es necesario disponer de un procesador gráfico compatible con *OpenGL*.

A.2. Compilación

Para compilar todos los objetos, librerías, *headers* y código fuente se hará uso de un *CMake_lists* con la herramienta *CMake* para generar el archivo binario. Este archivo se incluye en el directorio principal del código fuente [1].

Para hacer uso de este archivo es necesario llamar a la herramienta de *CMake* desde la terminal. En la carpeta del proyecto introducimos los siguientes comandos (Linux):

Código A.1: Proceso de compilado.

```
cd bin
cmake .
make
```

A.3. Ejecución

Para ejecutar el programa debemos abrir la carpeta donde se localiza el archivo binario e introducir el siguiente comando:

Código A.2: Instrucciones para ejecutar el programa.

```
./chip8 <Ciclos por segundo> <Nombre de la ROM>
```

El usuario puede activar ciertas características mientras el programa esté en ejecución:

- Si el usuario presiona la tecla `F1`, el emulador se reiniciará.
- Si el usuario presiona la tecla `F2`, el depurador se iniciará.
- Si el usuario presiona la barra espaciadora, se detendrá el ciclo actual. En este estado se podrá ejecutar el programa ciclo por ciclo con la tecla de suma.
- Si el usuario presiona la tecla `F3`, se guardará el estado actual del procesador.
- Si el usuario presiona la tecla `F4`, se cargará un estado anteriormente guardado del procesador.

[1] https://github.com/iloupv/chip8_emulator

Bibliografía

[Copeland, 2000] Copeland, J. (2000). The Church-Turing Thesis. http://www.alanturing.net/turing_archive/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html. Visitado:.

[Greene, 1997] Greene, T. C. P. (1997). CHIP-8 Technical Reference v1.0. http://devernay.free.fr/hacks/chip8/C8TECH10.HTM. Visitado:.

[Lantinga, 2021] Lantinga, S. (2021). SDL Wiki. https://wiki.libsdl.org/. Visitado:.

[Mikolay, 2019] Mikolay, M. (2019). CHIP-8 Instruction Set. https://github.com/mattmikolay/chip-8/wiki/CHIP%E2%80%908-Instruction-Set. Visitado:.

[Mikolay, 2020] Mikolay, M. (2020). CHIP-8 Technical Reference. https://github.com/mattmikolay/chip-8/wiki/CHIP%E2%80%908-Technical-Reference. Visitado:.

[Morlan, 2019] Morlan, A. (2019). Building a CHIP-8 emulator. https://austinmorlan.com/posts/chip8_emulator/. Visitado:.

[*Technocupid*, 2021] *Technocupid* (2021). CMake community wiki. https://gitlab.kitware.com/cmake/community/-/wikis/Home. Visitado:.

[Weisbecker, 1978] Weisbecker, J. (1978). Rca cosmac vip instruction manual. http://www.bitsavers.org/components/rca/cosmac/COSMAC_VIP_Instruction_Manual_1978.pdf. Visitado:.

[Winter, 1998] Winter, D. (1998). A chip-8 / schip emulator version 2.2.0. http://vanbeveren.byethost13.com/stuff/CHIP8.pdf?i=1. Visitado:.